

Texts in Computer Science

K. Erciyes

Guide to Graph Algorithms

Sequential, Parallel and Distributed

 Springer

Tabla de contenido

[Materia frontal](#)

[1. Introducción](#)

[1. Fundamentos](#)

[2. Introducción a las gráficas.](#)

[3. Algoritmos de grafos](#)

[4. Algoritmos de gráficos paralelos](#)

[5. Algoritmos de grafos distribuidos.](#)

[2. Algoritmos básicos de grafos](#)

[6. Árboles y gráficas transversales.](#)

[7. Gráficos ponderados](#)

[8. conectividad](#)

[9. emparejando](#)

[10. Independencia, dominación y cobertura de vértices.](#)

[11. colorear](#)

[3. Temas avanzados](#)

[12. Algoritmos de gráficos algebraicos y dinámicos](#)

[13. Análisis de gráficos grandes](#)

[14. Redes complejas](#)

[15. epílogo](#)

[Backmatter](#)

Textos en informática

Editores de serie

David Gries

Departamento de Ciencias de la Computación, Cornell University, Ithaca, Nueva York, EE.UU.

Orit Hazzan

Facultad de Educación en Tecnología y Ciencia, Technion — Instituto de Tecnología de Israel, Haifa, Israel

Fred B. Schneider

Departamento de Ciencias de la Computación, Cornell University, Ithaca, Nueva York, EE.UU.

[Más información sobre esta serie en http://www.springer.com/series/3191](#)

K. Erciyes

Guía de algoritmos de grafos

Secuencial, Paralela y Distribuida.



K. Erciyes

Instituto Internacional de Computación, Universidad Ege, Izmir, Turquía

ISSN 1868-0941e-ISSN 1868-095X

Textos en informática

ISBN 978-3-319-73234-3e-ISBN 978-3-319-73235-0

<https://doi.org/10.1007/978-3-319-73235-0>

Número de control de la Biblioteca del Congreso: 2017962034

© Springer International Publishing AG, parte de Springer Nature 2018

Esta obra está sujeta a derechos de autor. Todos los derechos están reservados por el Editor, ya sea en relación con la totalidad o parte del material, específicamente los derechos de traducción, reimpresión, reutilización de ilustraciones, recitación, transmisión, reproducción en microfilms o de cualquier otra forma física, y transmisión o almacenamiento de información, y recuperación, adaptación electrónica, software de computadora, o por metodología similar o diferente ahora conocida o desarrollada en lo sucesivo.

El uso de nombres descriptivos generales, nombres registrados, marcas registradas, marcas de servicio, etc. en esta publicación no implica, incluso en ausencia de una declaración específica, que dichos nombres están exentos de las leyes y regulaciones de protección relevantes y, por lo tanto, son gratis para todos. utilizar.

El editor, los autores y los editores pueden asumir que los consejos y la información de este libro se consideran verdaderos y precisos en la fecha de publicación. Ni el editor, ni los autores o los editores ofrecen una garantía, expresa o implícita, con respecto al material contenido en este documento o por cualquier error u omisión que se haya cometido. El editor permanece neutral con respecto a las reclamaciones jurisdiccionales en mapas publicados y afiliaciones institucionales.

Impreso en papel libre de ácido.

Esta impresión Springer es publicada por la compañía registrada Springer International Publishing AG parte de Springer Nature

La dirección registrada de la empresa es: Gewerbestrasse 11, 6330 Cham, Suiza.

A los recuerdos de Semra, Seyhun, Şebnem y Hakan.

Prefacio

Los gráficos son estructuras de datos clave para el análisis de varios tipos de redes, como las redes móviles ad hoc, Internet y redes complejas como las redes sociales y las redes biológicas. Se necesita el estudio de algoritmos de grafos para resolver de manera eficiente una variedad de problemas en dichas redes. Este estudio generalmente se centra en tres paradigmas fundamentales: algoritmos de grafos secuenciales; Algoritmos de grafos paralelos y algoritmos de grafos distribuidos. Los algoritmos secuenciales en general asumen un único flujo de control y tales métodos están bien establecidos. Para problemas de gráficos intratables que no tienen soluciones secuenciales en el tiempo polinomial, se pueden usar algoritmos de aproximación que han demostrado relaciones de aproximación a las soluciones óptimas. Muchas veces sin embargo, los algoritmos de aproximación no se conocen hasta la fecha y la única opción es el uso de heurísticas que son reglas de sentido común que se muestran para trabajar experimentalmente para una amplia gama de entradas. El diseñador de algoritmos se enfrenta con frecuencia a esta tarea de saber qué buscar o no; y qué camino seguir si la solución no existe. El primer objetivo de este libro es proporcionar un análisis completo y en profundidad de algoritmos de grafos secuenciales y guiar al diseñador sobre cómo abordar un problema difícil típico, mostrando cómo inspeccionar una heurística apropiada que generalmente se necesita en muchos casos.

Se necesitan algoritmos paralelos para acelerar el funcionamiento de los algoritmos de grafos. Los algoritmos paralelos de memoria compartida se sincronizan utilizando una memoria común y los algoritmos paralelos de memoria distribuida se comunican solo mediante el paso de mensajes. Los algoritmos de gráfico distribuido (o red) son conscientes de la topología de la red y pueden utilizarse para diversas tareas relacionadas con la red, como el enrutamiento. Algoritmos distribuidos es el término común usado para la memoria distribuida y algoritmos de grafos distribuidos, sin embargo, que llamaremos de memoria compartida y la memoria distribuida algoritmos de grafos paralelo algoritmos de grafos paralelas y algoritmos de grafos o de red distribuidos como algoritmos distribuidos. El diseño y análisis de algoritmos paralelos y distribuidos, así como algoritmos secuenciales para gráficos, serán el tema de este libro.

Un segundo y un objetivo fundamental de este libro es unificar estos tres métodos de algoritmos de grafía aparentemente diferentes cuando corresponda. Por ejemplo, el problema del árbol de expansión mínimo (MST) se puede resolver mediante cuatro algoritmos secuenciales clásicos: los algoritmos Boruvka, Prim, Kruskal y Reverse-Delete, todos con complejidades similares. Un algoritmo MST paralelo intentará encontrar el MST de una red grande en un número menor de procesadores únicamente para obtener una aceleración. En un algoritmo MST distribuido, cada procesador es un nodo del gráfico de red y participa para encontrar el MST de la red. Describiremos y compararemos los tres paradigmas para este y muchos otros problemas gráficos conocidos al considerar el mismo problema desde tres ángulos diferentes, lo que creemos ayudará a comprender mejor el problema y formar una visión unificadora.

Un tercer y un importante objetivo de este trabajo serán las conversiones entre la memoria secuencial, compartida y distribuida de algoritmos paralelos y distribuidos para gráficos. Este proceso no se implementa comúnmente en la literatura, aunque hay oportunidades en muchos casos. Vamos a ejemplificar este concepto mediante el algoritmo de coincidencia ponderada máxima en los gráficos. El algoritmo de aproximación secuencial para este propósito con una relación de 0.5 tiene una complejidad de tiempo de $O(m \log(m))$ con m siendo el número de aristas. Preis proporcionó un algoritmo secuencial localizado más rápido basado en el primer algoritmo con mejor $O(m)$ complejidad. Más tarde, Hoepmann proporcionó una versión distribuida del algoritmo de Preis. Más recientemente, Manne proporcionó una forma secuencial del algoritmo de gráfico distribuido de Hoepman y parallelizó este algoritmo. La secuencia de métodos empleados ha sido secuencial.

- secuencial - repartido - secuencial - paralelo para este problema gráfico. Aunque este ejemplo muestra una secuencia de transformación bastante larga, secuencial
- paralelo y secuencial - distribuidos son comúnmente seguidos por los investigadores en su mayoría por el sentido común. Algoritmos de grafos paralelos ↔ Gráficos distribuidos algoritmos de conversión de algoritmos muy rara vez se practica. Nuestro objetivo será sentar las bases de estas transformaciones entre paradigmas para convertir un algoritmo en un dominio a otro. Esto puede ser difícil para algunos tipos de algoritmos, pero los algoritmos de grafos son una buena premisa.

A medida que se desarrollan tecnologías más avanzadas, nos enfrentamos al análisis de big data de redes complejas que tienen decenas de miles de nodos y cientos de miles de bordes. También proporcionamos una parte sobre algoritmos para el análisis de grandes volúmenes de datos de redes complejas como Internet, redes sociales y redes biológicas en la célula. Para resumir, tenemos los siguientes objetivos en este libro:

- Un estudio exhaustivo y un estudio detallado de los principios fundamentales de los algoritmos de grafos secuenciales y los enfoques para problemas de NP-duro, algoritmos de aproximación y heurísticas.
- Un análisis comparativo de algoritmos de grafos secuenciales, paralelos y distribuidos que incluyen algoritmos para big data.
- Estudio de los principios de conversión entre los tres métodos.

Hay tres partes en el libro; En la primera parte, proporcionamos un breve resumen de los algoritmos de grafos, grafos secuenciales, paralelos y distribuidos. La segunda parte forma el núcleo del libro con un análisis detallado de algoritmos secuenciales, paralelos y distribuidos para problemas de gráficos fundamentales. En la última parte, nos centramos en los algoritmos de algoritmos algebraicos y dinámicos y los algoritmos de gráficos para redes muy grandes, que comúnmente se implementan utilizando heurísticas en lugar de soluciones exactas.

Revisamos la teoría tanto como sea necesario para el diseño de algoritmos de grafos secuenciales, paralelos y distribuidos, y nuestro énfasis para muchos problemas está en los detalles de la implementación en su totalidad. Nuestro estudio de los algoritmos de grafos secuenciales a lo largo del libro es exhaustivo, sin embargo, proporcionamos un análisis comparativo de los algoritmos secuenciales solo con los algoritmos grafos distribuidos y paralelos fundamentales. Mantuvimos el diseño de cada capítulo lo más homogéneo posible, describiendo el problema de manera informal y luego brindando

los antecedentes teóricos básicos. Luego describimos los algoritmos fundamentales describiendo primero la idea principal de un algoritmo; luego dando su pseudocódigo; Mostrando un ejemplo de implementación y finalmente el análisis de su corrección y complejidad.

Los destinatarios de este libro son los estudiantes senior / graduados de informática, ingeniería eléctrica y electrónica, bioinformática y cualquier investigador o persona con experiencia en matemáticas discretas, teoría de grafos básicos y algoritmos. Hay una página web para el libro de mantener erratas y otro material en: <http://Ube EGE.edu.tr/~Erciyes/GGA/..>.

Me gustaría agradecer a los estudiantes senior / graduados de la Universidad Ege, la Universidad de California Davis, la Universidad Estatal de California en San Marcos, y los estudiantes senior / graduados de la Universidad de Izmir que han tomado los algoritmos distribuidos y los cursos de redes complejas, a veces con nombres ligeramente diferentes, para su valiosa retroalimentación cuando se presentaron partes del material cubierto en el libro durante las conferencias. También me gustaría agradecer a los editores de Springer, Wayne Wheeler y Simon Rees, por su ayuda y fe en otro proyecto de libro que he propuesto.

K. ErciyesUn profesor emérito
Izmir, Turquía

Contenido

[1. Introducción](#)

[1. 1 Gráficos](#)

[1. 2 Algoritmos de grafos](#)

[1. 2. 1 Algoritmos de grafos secuenciales](#)

[1. 2. 2 Algoritmos de gráficos paralelos](#)

[1. 2. 3 Algoritmos de grafos distribuidos](#)

[1. 2. 4 Algoritmos para gráficos grandes](#)

[1. 3 retos en los algoritmos de grafos](#)

[1. 4 Esquema del libro](#)

Parte I Fundamentos

[2 Introducción a los Gráficos](#)

[2. 1 Introducción](#)

[2. 2 Notaciones y Definiciones.](#)

[2. 2. 1 grados de vértice](#)

[2. 2. 2 subgrafos](#)

[2. 2. 3 isomorfismo](#)

[2. 3 operaciones gráficas](#)

[2. 3. 1 unión e intersección](#)

[2. 3. 2 Producto cartesiano](#)

[2. 4 tipos de gráficas](#)

[2. 4. 1 Gráficos completos](#)

[2. 4. 2 Gráficos dirigidos](#)

[2. 4. 3 Gráficos ponderados](#)

[2. 4. 4 Gráficos bipartitos](#)

[2. 4. 5 Gráficos regulares](#)

[2. 4. 6 Gráficos lineales](#)

[2. 5 Caminatas, Caminos y Ciclos](#)

[2. 5. 1 Conectividad y Distancia](#)

[2. 6 Representaciones gráficas](#)

[2. 6. 1 Matriz de adyacencia](#)

[2. 6. 2 Lista de adyacencia](#)

[2. 6. 3 Matriz de incidencia](#)

[2. 7 árboles](#)

[2. 8 Gráficos y matrices.](#)

[2. 8. 1 valores propios](#)

[2. 8. 2 Matriz laplaciana](#)

[2. 9 Notas de capítulo](#)

[Referencias](#)

[3 algoritmos de grafos](#)

[3. 1 Introducción](#)

[3. 2 conceptos básicos](#)

[3. 2. 1 Estructuras](#)

[3. 2. 2 Procedimientos y funciones](#)

[3. 3 análisis asintótico](#)

[3. 4 Algoritmos Recursivos y Recurrencias](#)

[3. 5 Comprobación de la corrección de los algoritmos](#)

[3. 5. 1 Contraposición](#)

[3. 5. 2 Contradicciones](#)

[3. 5. 3 Inducción](#)

[3. 5. 4 fuerte inducción](#)

[3. 5. 5 invariantes de bucle](#)

[3. 6 Reducciones](#)

[3. 6. 1 Problemas de gráficos difíciles](#)

[3. 6. 2 Conjunto independiente a la reducción de la cubierta del vértice](#)

[3. 7 NP-Compleitud](#)

[3. 7. 1 Clases de complejidad](#)

[3. 7. 2 El primer problema NP-difícil: Satisfiabilidad](#)

[3. 8 Haciendo frente a NP-Completeness](#)

[3. 8. 1 algoritmos aleatorizados](#)

[3. 8. 2 Algoritmos de aproximación](#)

[3. 8. 3 Retroceso](#)

[3. 8. 4 Rama y Límite](#)

[3. 9 principales métodos de diseño](#)

[3. 9. 1 Algoritmos codiciosos](#)

[3. 9. 2 Divide y vencerás](#)

[3. 9. 3 Programación dinámica](#)

[3. 10 Notas de capítulo](#)

[Referencias](#)

[4 algoritmos de grafos paralelos](#)

[4. 1 Introducción](#)

[4. 2 conceptos y terminología](#)

[4. 3 arquitecturas paralelas](#)

[4. 3. 1 Arquitecturas de memoria compartida](#)

[4. 3. 2 Arquitecturas de memoria distribuida](#)

[4. 3. 3 Taxonomía de Flynn](#)

[4. 4 modelos](#)

4. 4. 1 Modelo PRAM

4. 4. 2 Modelo de paso de mensajes

4. 5 Análisis de los algoritmos paralelos.

4. 6 modos básicos de comunicación

4. 7 Métodos de diseño de algoritmos paralelos

4. 7. 1 paralelismo de datos

4. 7. 2 Paralelismo funcional

4. 8 Métodos de algoritmos paralelos para gráficos

4. 8. 1 Aleatorización y ruptura de simetría

4. 8. 2 particiones gráficas

4. 8. 3 Contracción del gráfico

4. 8. 4 puntero de salto

4. 8. 5 Descomposición del oído

4. 9 Asignación del procesador

4. 9. 1 Asignación estática

4. 9. 2 Equilibrio dinámico de carga

4. 10 Programación paralela

4. 10. 1 Programación de memoria compartida con hilos

4. 10. 2 Programación de memoria distribuida con MPI

4. 11 conclusiones

Referencias

5 algoritmos de grafos distribuidos

5. 1 Introducción

5. 2 tipos de sistemas distribuidos

5. 2. 1 redes móviles ad hoc

5. 2. 2 redes de sensores inalámbricos

[5. 3 modelos](#)

[5. 4 Comunicación y sincronización.](#)

[5. 5 Criterios de rendimiento](#)

[5. 6 Ejemplos de algoritmos de gráficos distribuidos](#)

[5. 6. 1 algoritmo de inundación](#)

[5. 6. 2 Construcción del árbol de expansión usando inundación](#)

[5. 6. 3 Comunicaciones básicas](#)

[5. 6. 4 Elección del líder en un anillo](#)

[5. 7 Notas del capítulo](#)

[Referencias](#)

Parte II Algoritmos básicos de grafos

[6 árboles y travesías gráficas](#)

[6. 1 Introducción](#)

[6. 2 árboles](#)

[6. 2. 1 Propiedades de los árboles](#)

[6. 2. 2 Encontrar la raíz de un árbol mediante el salto de puntero](#)

[6. 2. 3 Contando los árboles de expansión](#)

[6. 2. 4 Construyendo árboles de expansión](#)

[6. 2. 5 Travesías de árboles](#)

[6. 2. 6 árboles binarios](#)

[6. 2. 7 colas y montones de prioridad](#)

[6. 3 Búsqueda en profundidad](#)

[6. 3. 1 Un algoritmo recursivo](#)

[6. 3. 2 Un algoritmo iterativo DFS](#)

[6. 3. 3 DFS paralelo](#)

[6. 3. 4 Algoritmos distribuidos](#)

[6. 3. 5 Aplicaciones de DFS](#)

[6. 4 Búsqueda de amplitud](#)

[6. 4. 1 El algoritmo secuencial](#)

[6. 4. 2 BFS paralelo](#)

[6. 4. 3 Algoritmos distribuidos](#)

[6. 4. 4 Aplicaciones de BFS](#)

[6. 5 Notas de capítulo](#)

[Referencias](#)

[7 Gráficos ponderados](#)

[7. 1 Introducción](#)

[7. 2 árboles de expansión mínimos](#)

[7. 2. 1 Antecedentes](#)

[7. 2. 2 Algoritmos secuenciales](#)

[7. 2. 3 Algoritmos paralelos de MST](#)

[7. 2. 4 Algoritmo de Prim distribuido](#)

[7. 3 caminos más cortos](#)

[7. 3. 1 Senderos más cortos de una sola fuente](#)

[7. 3. 2 caminos más cortos para todos los pares](#)

[7. 4 Notas de capítulo](#)

[Referencias](#)

[8 conectividad](#)

[8. 1 Introducción](#)

[8. 2 teoría](#)

[8. 2. 1 Vertex y conectividad Edge](#)

[8. 2. 2 bloques](#)

[8. 2. 3 Teoremas de Menger](#)

- [8. 2. 4 Conectividad en Digraphs](#)
- [8. 3 algoritmos de conectividad secuencial](#)
- [8. 3. 1 Encontrar componentes conectados](#)
- [8. 3. 2 Búsqueda de puntos de articulación](#)
- [8. 3. 3 Bloque de descomposición](#)
- [8. 3. 4 Encontrando puentes](#)
- [8. 3. 5 Comprobación de conectividad fuerte](#)
- [8. 3. 6 Detección de componentes fuertemente conectados](#)
- [8. 3. 7 Búsqueda de vértices y conectividad perimetral](#)
- [8. 3. 8 Cierre transitivo](#)
- [8. 4 conectividad basada en flujo](#)
- [8. 4. 1 Flujos de red](#)
- [8. 4. 2 Búsqueda de conectividad de borde](#)
- [8. 4. 3 Búsqueda de conectividad de vértices](#)
- [8. 5 Búsqueda de conectividad paralela](#)
- [8. 5. 1 Cálculo de la matriz de conectividad](#)
- [8. 5. 2 Encontrar componentes conectados en paralelo](#)
- [8. 5. 3 Una encuesta de algoritmo de SCC paralelo](#)
- [8. 6 Algoritmos de conectividad distribuida](#)
- [8.6.1 Un algoritmo de conectividad k distribuida](#)
- [8. 7 Notas de capítulo](#)
- [Referencias](#)
- [9 a juego](#)
- [9. 1 Introducción](#)
- [9. 2 teoría](#)
- [9. 2. 1 Emparejamiento no ponderado](#)

[9. 2. 2 Matching ponderado](#)

[9. 3 Igualación de gráficos bipartitos no ponderados](#)

[9. 3. 1 Un algoritmo secuencial que usa rutas de aumento](#)

[9. 3. 2 Un algoritmo basado en el flujo](#)

[9. 3. 3 Hopcroft - Karp Algorithm](#)

[9. 4 Emparejamiento no ponderado en gráficas generales](#)

[9. 4. 1 Algoritmos secuenciales](#)

[9. 4. 2 Un algoritmo distribuido codicioso](#)

[9. 5 Igualación de gráficos bipartitos ponderados](#)

[9. 5. 1 Algoritmo codicioso](#)

[9. 5. 2 El método húngaro](#)

[9. 5. 3 El algoritmo de subasta](#)

[9. 6 Coincidencia ponderada en gráficas generales](#)

[9. 6. 1 algoritmo de Preis](#)

[9. 6. 2 Algoritmo de emparejamiento distribuido de Hoepman](#)

[9. 6. 3 Métodos de algoritmos paralelos](#)

[9. 7 Notas de capítulo](#)

[Referencias](#)

[10 Independencia, Dominación y Cobertura de vértices.](#)

[10. 1 Introducción](#)

[10. 2 Conjuntos independientes](#)

[10. 2. 1 Reducción a la camarilla.](#)

[10. 2. 2 Algoritmos secuenciales](#)

[10. 2. 3 Algoritmo MIS paralelo de Luby](#)

[10. 2. 4 Algoritmos distribuidos](#)

[10. 3 Conjuntos dominantes](#)

[10. 3. 1 un algoritmo secuencial codicioso](#)

[10. 3. 2 Un algoritmo distribuido para encontrar MDS](#)

[10. 4 cubierta de vértice](#)

[10. 4. 1 cubierta de vértice no ponderada](#)

[10. 4. 2 cubierta ponderada del vértice](#)

[10. 4. 3 Algoritmos paralelos](#)

[10. 5 Notas de capítulo](#)

[Referencias](#)

[11 colorear](#)

[11. 1 Introducción](#)

[11. 2 Colorear vértice](#)

[11. 2. 1 Relación con conjuntos independientes y camarillas](#)

[11. 2. 2 Algoritmos secuenciales](#)

[11. 2. 3 Algoritmos paralelos](#)

[11. 2. 4 Coloración de vértices distribuidos](#)

[11. 3 colores para bordes](#)

[11. 3. 1 Relación con la concordancia de gráficos](#)

[11. 3. 2 un algoritmo secuencial codicioso](#)

[11. 3. 3 Bipartite Graph Coloring](#)

[11. 3. 4 Coloración de bordes de gráficos completos](#)

[11. 3. 5 Un algoritmo paralelo](#)

[11. 3. 6 Un algoritmo distribuido](#)

[11. 4 Notas de capítulo](#)

[Referencias](#)

Parte III Temas avanzados

[12 Algoritmos de gráficos algebraicos y dinámicos](#)

[12. 1 Introducción](#)

[12. 2 matrices de grafos](#)

[12. 3 Algoritmos de gráficos algebraicos](#)

[12. 3. 1 Conectividad](#)

[12. 3. 2 Búsqueda de amplitud](#)

[12. 3. 3 caminos más cortos](#)

[12. 3. 4 árboles de expansión mínima](#)

[12. 3. 5 Emparejamiento algebraico](#)

[12. 4 Algoritmos de gráficos dinámicos](#)

[12. 4. 1 Métodos](#)

[12. 4. 2 Conectividad](#)

[12. 4. 3 Dynamic Matching](#)

[12. 5 Algoritmos de gráficos algebraicos dinámicos](#)

[12. 5. 1 Una biblioteca de matriz dinámica](#)

[12. 5. 2 Conectividad](#)

[12. 5. 3 Combinación perfecta usando la eliminación gaussiana](#)

[12. 6 Notas de capítulo](#)

[Referencias](#)

[13 Análisis de gráficos grandes](#)

[13. 1 Introducción](#)

[13. 2 parámetros principales](#)

[13. 2. Distribución de 1 grado](#)

[13. 2. 2 Densidad de gráfico](#)

[13. 2. 3 Coeficiente de agrupamiento](#)

[13. 2. 4 Índice de juego](#)

[13. 3 modelos de red](#)

[13. 3. 1 redes aleatorias](#)

[13. 3. 2 Redes del mundo pequeño](#)

[13. 3. 3 redes libres de escala](#)

[13. 4 centralidad](#)

[13. 4. 1 grado de centralidad](#)

[13. 4. 2 Centralidad de proximidad](#)

[13. 4. 3 Centralidad de intermediación](#)

[13. 4. 4 Centralidad de valores propios](#)

[13. 5 subgrafos densos](#)

[13. 5. 1 camarillas](#)

[13.5.2 k- puntuaciones](#)

[13. 5. 3 agrupamiento](#)

[13. 6 Notas de capítulo](#)

[Referencias](#)

[14 redes complejas](#)

[14. 1 Introducción](#)

[14. 2 redes biológicas](#)

[14. 2. 1 Clustering](#)

[14. 2. 2 Red Motif Search](#)

[14. 2. 3 Alineación de red](#)

[14. 3 redes sociales](#)

[14. 3. 1 Relaciones y equivalencia](#)

[14. 3. 2 Detección de la comunidad](#)

[14. 4 redes inalámbricas ad hoc](#)

[14.4.1 k -Conectividad](#)

[14. 4. 2 Agrupación en redes inalámbricas ad hoc](#)

14. 4. 3 Construcción de red troncal con conjuntos dominantes

14. 5 El Internet

14. 5. 1 Protocolo de vector de distancia

14. 5. 2 Algoritmo de estado de enlace

14. 5. 3 Enrutamiento jerárquico

14. 6 La web como red de información.

14. 6. 1 HITS Algoritmo

14. 6. 2 PageRank Algorithm

14. 7 Notas de capítulo

Referencias

15 epílogo

15. 1 Introducción

15. 2 Hoja de ruta para problemas difíciles

15. 3 ¿Son diferentes los algoritmos de grafos grandes?

15. 4 conversiones: ¿cuándo son útiles?

15. 5 Implementación

15. 5. 1 Secuencial, paralelo o distribuido

15. 5. 2 Clásico, algebraico, dinámico o todo?

15. 6 Un estudio de caso: Construcción de la red troncal en WSNs

15. 7 conclusiones

Referencias

Apéndice A: Convenciones de pseudocódigo

Apéndice B: Revisión de Álgebra Lineal

Índice

1. Introducción

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege, Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Los gráficos son estructuras discretas que se utilizan con frecuencia para modelar muchos problemas del mundo real, como las redes de comunicación, las redes sociales y las redes biológicas. Presentamos conceptos de algoritmos de grafos secuenciales, paralelos y distribuidos, desafíos en los algoritmos de grafos y el resumen del libro en este capítulo.

1.1 Gráficos

Los gráficos son estructuras discretas que se utilizan con frecuencia para modelar muchos problemas del mundo real, como las redes de comunicación, las redes sociales y las redes biológicas. Un gráfico consta de vértices y aristas que conectan estos vértices. Una gráfica se muestra como $G = (V, E)$ donde V es el conjunto de vértices y E es el conjunto de aristas que tiene. La Figura 1.1 muestra un ejemplo de gráfico con vértices y bordes entre ellos con $V = \{a, b, c, d\}$ y $E = \{(a, b), (a, c), (a, d), (b, c), (b, d), (b, e), (c, d), (d, e)\}$, (a, b) que denota el borde entre los vértices a y b, por ejemplo.

Los gráficos tienen numerosas aplicaciones que incluyen ciencias de la computación, computación científica, química y sociología, ya que son simples pero efectivas para modelar fenómenos de la vida real. Un vértice de una gráfica representa una entidad como una persona en una red social o una proteína en una red biológica. Una ventaja en tal red corresponde a una interacción social como la amistad en una red social o una interacción bioquímica entre dos proteínas en la célula.

El estudio de los gráficos tiene implicaciones tanto teóricas como prácticas. En este capítulo, describimos el objetivo principal del libro, que es proporcionar una vista unificada de los algoritmos de grafos en términos de algoritmos de grafos secuenciales, paralelos y distribuidos con énfasis en los algoritmos de grafos secuenciales. Describimos un problema de gráfica simple a partir de estas tres vistas y luego revisamos los desafíos en los algoritmos de gráfica al delinear finalmente el contenido del libro.

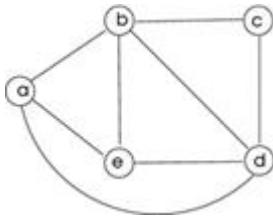


Fig. 1.1 Un ejemplo de gráfica consistente en vértices. $\{a, b, \dots, h\}$

1.2 Algoritmos de grafos

Un algoritmo consiste en una secuencia de instrucciones procesadas por una computadora para resolver un problema. Un algoritmo gráfico funciona en gráficos para encontrar una solución a un problema representado por los gráficos. Podemos clasificar los problemas del gráfico como secuenciales, paralelos y distribuidos, según el modo de ejecución de estos algoritmos en el entorno informático.

1.2.1 Algoritmos de grafos secuenciales

Un algoritmo secuencial tiene un único flujo de control y se ejecuta secuencialmente. Acepta una entrada, trabaja en la entrada y proporciona una salida. Por ejemplo, leer dos enteros, sumarlos e imprimir la suma es un algoritmo secuencial que consta de tres pasos.

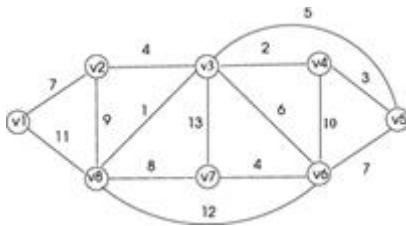


Fig. 1.2 Un ejemplo de gráfica

Supongamos que la gráfica de la Fig. 1.2 representa una pequeña red con vértices etiquetados $\{v_1, v_2, \dots, v_8\}$ y teniendo enteros 1, ..., 13 asociados con cada borde. El valor entero para cada borde puede denotar el costo de enviar un mensaje sobre ese borde. Comúnmente, los valores de borde se denominan pesos de bordes. Asumamos que nuestro objetivo es diseñar un algoritmo secuencial para encontrar el borde con el valor máximo. Esto puede tener algún uso práctico, ya que es posible que tengamos que encontrar el límite de costo más alto en la red para evitar ese enlace mientras se realiza alguna comunicación o transporte.

Formemos una matriz de distancia D para este gráfico, que tiene entradas $d(i, j)$ que muestran los pesos de los bordes entre vértices v_i y v_j como a continuación:

$$D = \begin{bmatrix} 0 & 7 & 0 & 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 4 & 0 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 2 & 5 & 6 & 14 & 1 \\ 0 & 0 & 2 & 0 & 3 & 10 & 0 & 0 \\ 0 & 0 & 5 & 3 & 0 & 13 & 0 & 0 \\ 0 & 7 & 6 & 10 & 13 & 0 & 4 & 12 \\ 0 & 7 & 14 & 0 & 0 & 4 & 0 & 8 \\ 11 & 9 & 1 & 0 & 0 & 12 & 8 & 0 \end{bmatrix}$$

Si dos vértices v_i y v_j no están conectados, insertamos un cero para que la entrada en D . Podemos tener un algoritmo secuencial simple que encuentre primero el valor más grande en cada fila de esta matriz y luego el valor máximo de todos estos valores más grandes en el segundo paso como se muestra en el Algoritmo 1.1.

Algorithm 1.1 Sequential_graph

```

1: int D[n, n] ← edge weights
2: int max[8], maximum
3: for i=1 to 8 do
4:   max[i] ← D[i, 1]
5:   for j=2 to 8 do
6:     if D[i, j] > max[i] then
7:       max[i] ← D[i, j]
8:     end if
9:   end for
10: end for
11: maximum ← max[1]
12: for i=2 to 8 do
13:   if max[i] > maximum then
14:     maximum ← max[i]
15:   end if
16: end for
17: output maximum

```

Este algoritmo requiere 8 comparaciones para cada fila para un total de 64 comparaciones. Necesita n^2 Comparaciones en general para una gráfica con n vértices.

1.2.2 Algoritmos de gráficos paralelos

Los algoritmos de gráficos paralelos apuntan al rendimiento como todos los algoritmos paralelos. Esta forma de acelerar los programas es necesaria especialmente para gráficos muy grandes que representan redes complejas, como las redes biológicas o sociales, que consisten en una gran cantidad de nodos y bordes. Tenemos una serie de procesadores que trabajan en paralelo en el mismo problema y los resultados generalmente se recopilan en un solo procesador para la salida. Los algoritmos paralelos pueden sincronizarse y comunicarse a través de la memoria compartida o se ejecutan como algoritmos de memoria distribuida que se comunican solo mediante la transferencia de mensajes. El último modo de comunicación es una práctica más común en la computación paralela debido a su versatilidad para realizar arquitecturas de red generales.

Podemos intentar parallelizar un algoritmo secuencial existente o diseñar un nuevo algoritmo paralelo desde cero. Un enfoque común en la computación paralela es la partición de los datos en varios procesadores para que cada elemento de computación funcione en una partición en particular. Otro enfoque fundamental es la partición de la computación entre los procesadores, como lo investigaremos en el Cap. 4 . Veremos que algunos problemas de gráficos son difíciles de partitionar en datos o cómputos.

Reconsideremos el algoritmo secuencial en la sección anterior e intentemos parallelizarlo utilizando la partición de datos. Dado que los datos del gráfico están representados por la matriz de distancia, lo primero a considerar sería la partición de esta matriz. De hecho, la partición por filas o por columnas de una matriz que representa un gráfico se usa comúnmente en algoritmos de gráficos paralelos. Permítanos tener un procesador de control que llamaremos al supervisor o la raíz y dos procesadores de trabajo para hacer el trabajo real. Este modo de operación, a veces llamado supervisor / trabajador , también es una práctica común en el diseño de algoritmos paralelos. Los procesadores son comúnmente llamados procesos para significar que el procesador real también puede estar haciendo

algún otro trabajo. Ahora tenemos tres procesos p_0 , p_1 y p_2 y p_0 es el supervisor. El proceso p_0 tiene la matriz de distancia inicialmente, y divide y envía la primera mitad de las filas de 1 a 4 a p_1 y 5 a 8 a p_2 . Como se muestra abajo:

$$D = \begin{bmatrix} 0 & 7 & 0 & 0 & 0 & 0 & 1 & p_1 \\ 7 & 0 & 4 & 0 & 0 & 0 & 9 & \\ 0 & 4 & 0 & 2 & 5 & 6 & 14 & 1 \\ 0 & 0 & 2 & 0 & 3 & 10 & 0 & 0 \\ \hline 0 & 0 & 5 & 3 & 0 & 13 & 0 & 0 & p_2 \\ 0 & 7 & 6 & 10 & 13 & 0 & 4 & 12 \\ 0 & 7 & 14 & 0 & 0 & 4 & 0 & 8 \\ 11 & 9 & 1 & 0 & 0 & 12 & 8 & 0 \end{bmatrix}$$

Cada trabajador ahora encuentra el incidente de borde más pesado en los vértices en las filas que se asigna utilizando el algoritmo secuencial descrito y envía este resultado al supervisor p_0 que encuentra el máximo de estos dos valores y lo produce. Una forma más general de este algoritmo con k procesos de trabajo se muestra en el algoritmo 1.2. Dado que los datos se dividen en dos procesos ahora, esperaríamos tener una disminución significativa en el tiempo de ejecución del algoritmo secuencial. Sin embargo, ahora tenemos costos de comunicación entre el supervisor y los trabajadores, lo que puede no ser trivial para las grandes transferencias de datos.

Algorithm 1.2 Parallel_graph

```

1: int  $D[n, n] \leftarrow$  edge weights of graph  $G$ 
2: int  $Max[n], E[n/k, n]\max$ 
3: if  $i = root$  then
4:   row-wise partition distance matrix  $D$  of graph into  $D_1, \dots, D_k$ 
5:   for  $i=1$  to  $k$  do
6:     send  $D_i$  to  $p_i$ 
7:   end for
8:   for  $i=1$  to  $k$  do
9:     receive  $largest_i$  from  $p_i$  into  $max[i]$ 
10:  end for
11:  find the maximum value of  $max$  using the sequential algorithm
12:  output  $maximum$ 
13: else
14:   receive my rows into  $E$ 
15:   find the maximum value in  $E$  using sequential algorithm
16:   send root my maximum value
17: end if

```

El diseño de algoritmos de gráficos paralelos puede no ser trivial como en este ejemplo, y en general necesitamos métodos más sofisticados. La operación que debemos realizar puede depender en gran medida de lo que se hizo antes, lo que significa que es posible que se requieran comunicaciones y sincronización significativas entre los trabajadores. La comunicación entre procesos a través de la red que conecta los nodos computacionales es costosa y podemos terminar diseñando un algoritmo de gráfico paralelo que no es eficiente.

1.2.3 Algoritmos de grafos distribuidos

Grafico distribuido Los algoritmos son una clase de algoritmos gráficos en los que tenemos un nodo computacional representado por un vértice del gráfico. Los problemas a resolver con tales algoritmos están relacionados con la red que representan; por ejemplo, puede ser necesario encontrar la distancia más corta entre dos nodos cualquiera en la red, de modo que cada vez que un paquete de datos llega a un nodo en la red, reenvía el paquete a uno de sus vecinos que está en la ruta de menor

costo para el destino. En tales algoritmos, cada nodo normalmente ejecuta el mismo algoritmo pero tiene diferentes vecinos para comunicarse y transferir su resultado local. En esencia, nuestro objetivo es resolver un problema general relacionado con el gráfico que representa la red mediante la cooperación de los nodos en la red.algoritmos locales .

En la versión distribuida de nuestro algoritmo de búsqueda de borde de peso máximo de muestra, tenemos nodos computacionales de una red de computadoras como los vértices del gráfico, y nuestro objetivo es que cada nodo en la red modelada por el gráfico reciba el borde de mayor peso del gráfico al final. Intentaremos resolver este problema utilizando rondas para la sincronización de los nodos. Cada nodo comienza la ronda, realiza alguna función en la ronda y no comienza la siguiente ronda hasta que todos los otros nodos también hayan terminado la ejecución de la ronda. Este modelo se usa ampliamente para algoritmos distribuidos, como describiremos en el Capítulo. [5](#)y no hay otro control central que no sea la sincronización de las rondas. Cada nodo comienza transmitiendo el mayor peso que incide a todos sus vecinos y recibiendo los valores de mayor peso de los vecinos. En las siguientes rondas, un nodo transmite el peso más grande que ha visto hasta ahora y después de un cierto número de pasos, el valor más grande se propagará a todos los nodos del gráfico como se muestra en el Algoritmo 1.3. El número de pasos es el diámetro del gráfico, que es el número máximo de bordes entre dos vértices cualquiera.

Algorithm 1.3 Distributed_graph

```

1: boolean finished, round_over ← false
2: message type start, result, step
3: while count ≤ diam( $G$ ) do
4:   receive max(j) from all neighbors
5:   find the maximum of all received values
6:   send the maximum value to all neighbors
7:   count ← count + 1
8: end while

```

Ahora podemos ver las diferencias fundamentales entre los algoritmos de gráficos paralelos y distribuidos utilizando este ejemplo de la siguiente manera.

- Los algoritmos de gráficos paralelos se necesitan principalmente para la aceleración que proporcionan. Hay una serie de elementos de procesamiento que funcionan en paralelo que cooperan para completar una tarea general. La relación principal entre el número de procesos y el tamaño del gráfico es que preferiríamos utilizar más procesos para gráficos grandes. Suponemos que cada elemento de procesamiento puede comunicarse entre sí en general, aunque existen algunas arquitecturas de computación paralela especiales, como los procesadores que forman una arquitectura de comunicación cúbica como en el hipercubo .
- En los algoritmos de gráficos distribuidos, los nodos computacionales son los vértices del gráfico en cuestión y se comunican con sus vecinos solo para resolver un problema relacionado con la red representada por el gráfico. Tenga en cuenta que el número de proceso es el número de vértices del gráfico para estos algoritmos.

Un objetivo importante de este libro es proporcionar una vista unificada de los algoritmos gráficos desde estos tres ángulos diferentes. Es posible que deseamos resolver un problema de red en un entorno de procesamiento paralelo, por ejemplo,

todas las rutas más cortas entre dos nodos de la red pueden necesitar ser almacenadas en un servidor central para ser transferidas a nodos individuales o con fines estadísticos. En este caso, ejecutamos un algoritmo paralelo para la red utilizando varios elementos de procesamiento. En una configuración de red, necesitamos que cada nodo trabaje para conocer las rutas más cortas desde él a otros nodos.

Un enfoque general es derivar algoritmos de gráficos paralelos y distribuidos a partir de uno secuencial, pero hay formas de convertir un algoritmo de gráficos paralelos a uno distribuido o viceversa para algunos problemas. Para el problema de ejemplo que tenemos, podemos tener cada fila de la matriz de distancia D asignada a un solo proceso. De esta manera, cada proceso puede ser representado por un nodo de red siempre que se comunique solo con sus vecinos. Las conversiones como tales son útiles en muchos casos, ya que no diseñamos un nuevo algoritmo desde cero.

1.2.4 Algoritmos para gráficos grandes

Los recientes avances técnicos en las últimas décadas han resultado en la disponibilidad de datos de redes muy grandes. Estas redes se denominan comúnmente redes complejas y consisten en decenas de miles de nodos y cientos de miles de enlaces entre los nodos. Uno de estos tipos de redes son las redes biológicas dentro de la célula de los organismos vivos. Una red de interacción proteína-proteína (PPI) es una red biológica formada por proteínas que interactúan fuera del núcleo de la célula.

Una red social que consiste en individuos que interactúan a través de Internet puede ser nuevamente una red muy grande. Estas redes complejas se pueden modelar mediante gráficos con vértices que representan los nodos y limita la interacción entre los nodos como cualquier otra red. Sin embargo, estas redes son diferentes de una red pequeña modelada por un gráfico en algunos aspectos. En primer lugar, tienen diámetros muy pequeños, lo que significa que la distancia más corta entre dos vértices es pequeña en comparación con sus tamaños. Por ejemplo, se encuentra que varias redes PPI que consisten en miles de nodos tienen un diámetro de solo varias unidades. Del mismo modo, las redes sociales y las redes tecnológicas como Internet también tienen diámetros pequeños. Este estado es conocido como pequeño mundo.propiedad. En segundo lugar, los estudios empíricos sugieren que estas redes tienen muy pocos nodos con un número muy alto de conexiones; y la mayoría de los otros nodos tienen pocas conexiones con vecinos. Esta propiedad llamada sin escala se exhibe nuevamente en la mayoría de las redes complejas. Por último, el tamaño de estas redes es grande y requiere algoritmos eficientes para su análisis. En resumen, necesitamos algoritmos eficientes y posiblemente paralelos que exploten varias propiedades, como las características de las redes de escala reducida y de mundo pequeño.

1.3 Desafíos en los algoritmos de grafos

Existen numerosos desafíos en los gráficos que deben resolverse mediante algoritmos de gráficos.

- Complejidad de los algoritmos gráficos : un algoritmo de tiempo polinomial tiene una complejidad que puede expresarse mediante una función

polinomial. Hay muy pocos algoritmos de tiempo polinomial para la mayoría de los problemas relacionados con los gráficos. Los algoritmos disponibles típicamente tienen complejidades de tiempo exponenciales, lo que significa que incluso para gráficos de tamaño moderado, los tiempos de ejecución son significativos. Por ejemplo, supongamos que un algoritmo A para resolver un problema gráfico P tiene complejidad de tiempo $\approx n^2$, siendo n el número de vértices en el gráfico. Podemos ver que A puede tener un bajo rendimiento incluso para gráficos con $n > 20$ vértices. Entonces tenemos las siguientes opciones:

Algoritmos de aproximación : busque un algoritmo de aproximación que encuentre una solución subóptima en lugar de una óptima. En este caso, debemos demostrar que el algoritmo de aproximación siempre proporciona una solución dentro de una relación de aproximación a la solución óptima. Se pueden emplear varias técnicas de prueba y no hay necesidad de experimentar el algoritmo de aproximación que no sea con fines estadísticos. Encontrar y probar los algoritmos de aproximación es difícil para muchos problemas de gráficos.

Algoritmos aleatorios : estos algoritmos deciden el curso de la ejecución en función de alguna elección aleatoria, por ejemplo, la selección de un borde al azar. La salida se presenta normalmente como se espera o con alta probabilidad, lo que significa que existe la posibilidad de que la salida no sea correcta, aunque sea ligeramente. Sin embargo, los algoritmos aleatorios proporcionan soluciones polinómicas a muchos problemas gráficos difíciles.

Heurísticas : en muchos casos, nuestra única opción es el uso de enfoques de sentido común llamados heurísticos en busca de una solución. La elección de una heurística se persigue comúnmente por intuición y necesitamos experimentar el algoritmo con la heurística para una amplia gama de entradas para demostrar que funciona de manera experimental .

Existen otros métodos, como el retroceso y la bifurcación, que funcionan solo para un subconjunto del espacio de búsqueda y, por lo tanto, tienen menos complejidades de tiempo. Sin embargo, estos enfoques pueden aplicarse solo a un subconjunto de problemas y no son generales. Ejemplifiquemos estos conceptos con un ejemplo. Una camarilla en un gráfico es un subgrafo que cada vértice en este subgrafo tiene conexiones con todos los otros vértices en el subgrafo como se muestra en la Fig. 1.3 . Encontrar camarillas en una gráfica tiene muchas implicaciones, ya que exhiben densas regiones de actividad. Encontrar la camarilla más grande de un gráfico G con n los vértices, que es la camarilla con el número máximo de vértices en el gráfico, no se pueden realizar en tiempo polinomial. Un algoritmo de fuerza bruta , que suele ser el primer algoritmo que viene a la mente, enumerará todos $\approx \frac{n(n-1)}{2}$ Subgrafos de G y verifique la condición de camarilla desde la más grande a la más pequeña En lugar de buscar un algoritmo de aproximación, podríamos hacer lo siguiente por intuición: comience con el vértice que tenga el mayor número de conexiones llamado grado; Verifique si todos sus vecinos tienen el mismo número de conexiones y si todos tienen, entonces tenemos una pandilla. Si esto falla, continúa con el siguiente vértice de grado más alto. Esta heurística funcionará bien, pero en general, debemos demostrar

experimentalmente que una heurística funciona para la mayoría de las variaciones de entrada, por ejemplo, para el 90%, pero un algoritmo que funcione bien durante el 60% del tiempo con entradas diversas no sería favorable heurístico.

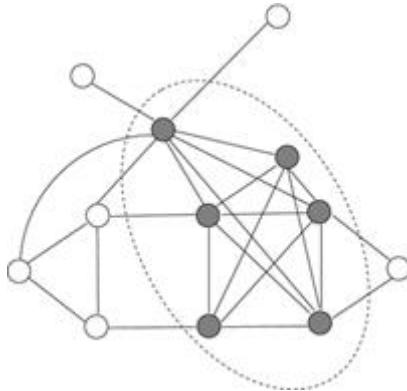


Fig. 1.3 La camarilla máxima de un gráfico de muestra se muestra mediante vértices oscuros. Todos los vértices en esta camarilla están conectados entre sí.

- Rendimiento: incluso con algoritmos de gráficos de tiempo polinomial, el tamaño del gráfico puede restringir su uso para gráficos grandes. El interés reciente en los grandes gráficos que representan grandes redes de la vida real exige algoritmos de alto rendimiento que se realizan comúnmente mediante computación paralela. Las redes biológicas y las redes sociales son ejemplos de tales redes. Por lo tanto, existe la necesidad de implementar algoritmos paralelos eficientes en estos gráficos grandes. Sin embargo, algunos problemas de gráficos son difíciles de parallelizar debido a la estructura de los procedimientos utilizados.
- Distribución : varias redes reales grandes se distribuyen en un sentido, cada nodo de la red es un elemento informático autónomo. Internet, la web, las redes móviles ad hoc y las redes de sensores inalámbricos son ejemplos de redes que pueden denominarse redes de computadoras en general. Estas redes pueden ser nuevamente modeladas convenientemente por gráficos. Sin embargo, los nodos de la red ahora participan activamente en la ejecución del algoritmo gráfico. Este tipo de algoritmos se denomina algoritmos distribuidos .

El objetivo principal de este libro es el estudio de algoritmos de grafos desde tres ángulos: algoritmos secuenciales, paralelos y distribuidos. Creemos que este enfoque proporcionará una mejor comprensión del problema en cuestión y su solución al mostrar también sus posibles áreas de aplicación. Seremos lo más completos posible en el estudio de algoritmos de gráficos secuenciales, pero solo presentaremos algoritmos de gráficos representativos para casos paralelos y distribuidos. Veremos que algunos problemas gráficos han complicado las soluciones algorítmicas paralelas informadas en estudios de investigación y proporcionaremos una encuesta de investigación contemporánea de los temas en estos casos.

1.4 Esquema del libro

Hemos dividido el libro en tres partes de la siguiente manera.

- Fundamentos : Esta parte tiene cuatro capítulos; El primer capítulo contiene una revisión densa de los conceptos básicos de la teoría de grafos. Algunos de estos conceptos se detallan en capítulos individuales. Luego describimos los algoritmos de grafos secuenciales, paralelos y distribuidos en secuencia en tres capítulos. En cada capítulo, primero proporcionamos los conceptos principales sobre el método de algoritmo y luego proporcionamos una serie de ejemplos en gráficos que utilizan el método mencionado. Por ejemplo, en los métodos de algoritmos secuenciales, proporcionamos un algoritmo de grafía codicioso a la vez que describimos algoritmos codiciosos. Esta parte básicamente forma el fondo para las partes II y III.
- Algoritmos de grafos básicos: Esta parte contiene el material central del libro. Analizamos los temas principales de la teoría de gráficos en cada capítulo, que son árboles y recorridos de gráficos; gráficos ponderados; conectividad pareo; subgrafos y colorear. Aquí, dejamos de lado algunos temas teóricos de la teoría de grafos que no tienen algoritmos significativos. Los temas que investigamos en el libro permiten métodos algorítmicos convenientemente y comenzamos cada capítulo con una breve reseña teórica para el análisis algorítmico. En otras palabras, nuestro tratamiento de los conceptos teóricos de gráficos relacionados no es exhaustivo, ya que nuestro objetivo principal es el estudio de algoritmos de gráficos en lugar de la teoría de gráficos por sí sola. En cada capítulo, Primero describimos los algoritmos secuenciales y esta parte es un lugar en el libro que intentamos ser lo más completo posible al describir la mayoría de los algoritmos bien establecidos del tema. A continuación, solo proporcionamos muestras de algoritmos paralelos y distribuidos sobre el tema investigado. Estos son típicamente uno o dos algoritmos conocidos en lugar de una lista completa. En algunos casos, los algoritmos paralelos o distribuidos son complicados. Para tales problemas, damos una encuesta de algoritmos con descripciones cortas.
- Temas avanzados : presentamos temas recientes y más avanzados en algoritmos de gráficos que la Parte II en esta sección del libro que comienza con algoritmos de gráficos algebraicos y dinámicos. Los algoritmos de gráficos algebraicos suelen hacer uso de las matrices asociadas con un gráfico y las operaciones en ellas mientras resuelven un problema de gráfico. Los gráficos dinámicos representan redes reales donde los bordes se insertan y eliminan de un gráfico en el tiempo. Los algoritmos para tales gráficos, llamados algoritmos de gráficos dinámicos, tienen como objetivo proporcionar soluciones en un tiempo más corto que ejecutar el algoritmo estático desde cero.

Los gráficos grandes que representan redes de la vida real, como las redes biológicas y sociales, tienden a tener propiedades interesantes e inesperadas, como hemos descrito. El estudio de tales gráficos se ha convertido en una importante dirección de investigación en la ciencia de redes recientemente. Por lo tanto, consideramos que es apropiado tener dos capítulos del libro dedicados para este propósito. Los algoritmos para estos gráficos grandes tienen objetivos de alguna

manera diferentes, y la detección de la comunidad que encuentra regiones densas en estos gráficos se ha convertido en uno de los principales temas de investigación. Primero proporcionamos un capítulo sobre la descripción general y el análisis de estos gráficos grandes junto con algoritmos para calcular algunos parámetros importantes. Luego revisamos los tipos de redes complejas básicas con algoritmos utilizados para resolver problemas fundamentales en estas redes.

Concluimos este capítulo enfatizando una vez más los objetivos principales del libro. Primero, sería apropiado declarar lo que este libro no es. Este libro no pretende ser un libro de teoría de gráficos, un libro de computación en paralelo o un libro de algoritmos distribuidos en gráficos. Asumimos una familiaridad básica con estas áreas, aunque proporcionamos una breve y densa revisión de estos temas en relación con los problemas del gráfico en la Parte I. Describimos la teoría básica del gráfico, incluida la notación y los teoremas básicos relacionados con el tema al comienzo de cada capítulo. Nuevamente, nuestro énfasis está en la teoría de gráficos que está relacionada con el algoritmo de gráficos que intentamos revisar. Intentamos ser lo más completos posible en el análisis de algoritmos de grafos secuenciales, pero solo revisamos algoritmos de grafos distribuidos y paralelos ejemplares. Nuestro enfoque principal es guiar al lector a los algoritmos de gráficas mediante la investigación y el estudio del mismo problema desde tres puntos de vista diferentes: una aproximación secuencial, un paralelo típico y enfoques algorítmicos distribuidos. Este enfoque es efectivo y beneficioso, no solo porque ayuda a entender mejor el problema en cuestión, sino que también es posible convertir de un enfoque a otro ahorrando una cantidad significativa de tiempo en comparación con el diseño de un algoritmo completamente nuevo.

Parte I

Fundamentos

2. Introducción a las gráficas.

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege, Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Los gráficos se utilizan para modelar muchas aplicaciones con vértices de un gráfico que representa los objetos o nodos y los bordes que muestran las conexiones entre los nodos. Revisamos las notaciones utilizadas para gráficos, definiciones básicas, grados de vértice, subgrafos, isomorfismo de gráficos, operaciones de gráficos, gráficos dirigidos, distancia, representaciones de gráficos y matrices relacionadas con los gráficos de este capítulo.

2.1 Introducción

Los objetos y las conexiones entre ellos se producen en una variedad de aplicaciones, como carreteras, redes de computadoras y circuitos eléctricos. Los gráficos se utilizan para modelar tales aplicaciones con vértices de un gráfico que representa los objetos o nodos y los bordes que muestran las conexiones entre los nodos.

Repasamos los conceptos teóricos de la gráfica básica en una forma bastante densa en este capítulo. Esta revisión incluye notaciones utilizadas, definiciones básicas, grados de vértice, subgrafos, isomorfismo de gráficos, operaciones de gráficos, gráficos dirigidos, distancia, representaciones de gráficos y matrices relacionadas con los gráficos. Cuando analizamos los algoritmos secuenciales, paralelos y distribuidos para estos problemas, dejamos la discusión de las propiedades más avanzadas de los gráficos, como el emparejamiento, la conectividad, los subgrafos especiales y los colores. También demoramos la revisión de métodos y parámetros para el análisis de gráficos grandes a la Parte III. Estos gráficos grandes se utilizan para modelar redes complejas como Internet o redes biológicas, que consisten en una gran cantidad de vértices y aristas.

2.2 Notaciones y definiciones

Una gráfica es un conjunto de puntos y un conjunto de líneas en un plano o un espacio 3D. Una gráfica se puede definir formalmente de la siguiente manera.

Definición 2.1 (gráfica) Una gráfica $G = (V, E, g)$ o $G = (V(G), E(G), g)$ es una estructura discreta que consiste en un conjunto de vértices V y un conjunto de aristas E y una relación g que asocia cada borde con dos vértices del conjunto V .

El conjunto de vértices consiste en vértices también llamados nodos , y un borde en el conjunto de bordes incide entre dos vértices llamados sus puntos finales . El conjunto de vértices de un gráfico G se muestra como $V (G)$ y el borde establecido como $E (G)$. Usaremos V para $V (G)$ y E para $E (G)$ cuando se conozca la gráfica en consideración. Una gráfica trivial tiene un vértice y no tiene aristas. Un gráfico nulo tiene un conjunto de vértices vacíos y un conjunto de bordes vacíos. Una gráfica se llama finita si ambas $V (G)$ y $E (G)$ son finitos. Consideraremos solo gráficos simples y finitos en este libro, a menos que se indique lo contrario. El número de vértices de un gráfico G se denomina orden y usaremos el literal n para este parámetro. El número de bordes de G se denomina tamaño y mostraremos este parámetro mediante el literal m . Un borde de una gráfica G entre sus vértices u y v se muestra comúnmente como (u , v) , uv o algunas veces $\llbracket u, v \rrbracket$; Adoptaremos la primera. Los vértices en los extremos de un borde se llaman sus puntos finales o vértices de extremo o simplemente termina . Para un borde (u , v) entre los vértices u y v , nos dicen u y v son incidente al borde (u , v) .

Definición 2.2 (bucle automático, borde múltiple) Un bucle automático es un borde con los mismos puntos finales. Los bordes múltiples tienen el mismo par de puntos finales.

Un borde que no es un auto-bucle se llama un borde adecuado . Un gráfico simple no tiene auto-loops o múltiples bordes. Una gráfica que contiene múltiples bordes se llama multigraph . Un gráfico subyacente de un multigraph se obtiene sustituyendo un borde único por cada borde múltiple. En la Fig. 2.1 se muestra un ejemplo de multigraph .

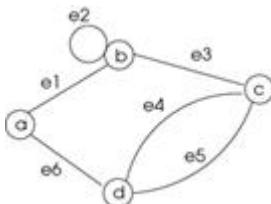


Fig. 2.1 Una gráfica con $V(G) = \{a, b, c, d\}$ y $E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. Borde e_2 es un auto-bucle y bordes e_4 y e_5 son bordes múltiples

Definición 2.3 (complemento de un gráfico) El complemento de un gráfico $G (V , E)$ es el gráfico $\overline{G(V, E)}$ con el mismo vértice establecido como G y cualquier borde $\llbracket u, v \rrbracket \in E'$ si y solo si $\llbracket u, v \rrbracket \notin E$.

De manera informal, tenemos el mismo conjunto de vértices en el complemento de un grafo G , pero sólo tienen bordes que no existen en G . Los complementos de dos gráficos se muestran en la Fig. 2.2 .

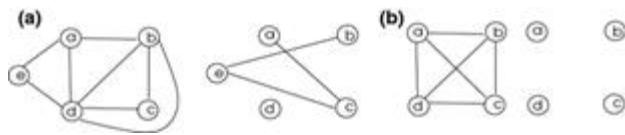


Fig. 2.2 a Complemento de un gráfico de muestra, b Complemento de un gráfico completamente conectado en el que cada vértice está conectado a todos los otros vértices

2.2.1 Grados de vértices

El grado de un vértice en un gráfico es un atributo útil de un vértice como se define a continuación.

Definición 2.4 (grado de un vértice) La suma del número de bordes apropiados y el doble del número de auto-bucles que inciden en un vértice v de un gráfico G se denomina grado y se muestra mediante grados (v).

Un vértice que tiene un grado de 0 se llama vértice aislado y un vértice de grado 1 se llama vértice colgante . El grado mínimo de una gráfica G se denota por $\delta(G)$ y el grado máximo por $\Delta(G)$. La siguiente relación entre el grado de un vértice v en G y este parámetro es válido:

$$0 \leq \delta(G) \leq \deg(v) \leq \Delta(G) \leq n - 1.$$

(2.1)

Dado que el número máximo de bordes en un gráfico simple no dirigido es $\frac{n(n-1)}{2}$, para cualquier gráfico,

$$0 \leq m \leq \frac{n(n-1)}{2} = \binom{n}{2}$$

Podemos, por lo tanto, concluir que hay como máximo $\binom{n}{2}$ Posibles gráficos simples no dirigidos que tienen n vértices. El primer teorema de la teoría de grafos , al que comúnmente se hace referencia como el lema del apretón de manos, es el siguiente.

Teorema 2.1 (Euler) La suma de los grados de un gráfico simple no dirigido $G = (V, E)$ Es el doble del número de sus bordes que se muestran a continuación.

$$\sum_{v \in V} \deg(v) = 2m$$

(2.2)

La prueba es trivial ya que cada borde se cuenta dos veces para encontrar la suma. Un vértice en una gráfica con n vértices puede tener un grado máximo de $n - 1$. Por lo tanto,

la suma de los grados en un gráfico completo donde cada vértice está conectado a todos los demás es $\frac{n(n-1)}{2}$. El número total de aristas es $\frac{n(n-1)/2}{2}$ en tal grafo. En una reunión de n personas, si todos se dieran la mano, el número total de apretones de manos sería $\frac{n(n-1)/2}{2}$ y de ahí el nombre de lema. El grado promedio de una gráfica es

$$\frac{\sum_{v \in V} \deg(v)}{n} = 2m/n,$$

(2.3)

Un vértice se llama impar o incluso dependiendo de si su grado es impar o par.

Corolario 2.1 El número de vértices de grados impares de un gráfico es un número par.

Prueba de los vértices de una gráfica. $G = (V, E)$ se puede dividir en grado parejo (v_p) y grado impar (v_i) vértices. La suma de grados puede ser declarada como

$$\sum_{v \in V} \deg(v) = \sum_{v_p \in V} \deg(v_p) + \sum_{v_i \in V} \deg(v_i)$$

Como la suma es pareada por el teorema 2.1, la suma de los vértices de grados impares también debería ser par, lo que significa que debe haber un número par de vértices de grados impares. \square

Teorema 2.2 Cada gráfica con al menos dos vértices tiene al menos dos vértices que tienen el mismo grado.

Prueba Probaremos este teorema usando la contradicción. Supongamos que no hay tal gráfico. Para una gráfica con n vértices, esto implica que los grados de vértice son únicos, de 0 a $n-1$. No podemos tener un vértice u con grado de $n-1$ y un vértice v con 0 grados en el mismo gráfico G que el anterior implica que u está conectado a todos los otros vértices en G y, por lo tanto, una contradicción. \square

Este teorema se puede poner en práctica en una reunión de personas donde algunos se conocen y el descanso no se conoce. Si las personas están representadas por los vértices de una gráfica donde un borde entre dos individuos, quienes se conocen entre sí están representados por un borde, podemos decir que hay al menos dos personas que tienen el mismo número de conocidos en la reunión.

2.2.1.1 Secuencias de Grado

La secuencia de grados de una gráfica se obtiene cuando los grados de sus vértices se enumeran en algún orden.

Definición 2.5 (secuencia de grados) La secuencia de grados de una gráfica G es la lista de los grados de sus vértices en no decreciente o no incremental, más comúnmente en

orden no incremental. La secuencia de grados de un dígrafo es la lista que consta de sus pares de grado y de grado.

La secuencia de grados del gráfico en la Fig. 2.1 es $\{2, 3, 3, 4\}$ para los vértices a, d, c, b en secuencia. Dada una secuencia de grado $D = (d_1, d_2, \dots, d_n)$, Que consiste en un conjunto finito de números enteros no negativos, D se llama gráfica de si representa una secuencia de grado de algunos gráfica G . Es posible que tengamos que comprobar si una secuencia de grados dada es gráfica. La condición que $\deg(v) < n - 1 \quad \forall v \in V$ Es la primera condición y también $\sum_{v \in V} \deg(v)$ debe ser par. Sin embargo, estos son necesarios pero no suficientes y se propone un método eficiente en el teorema primero probado por Havel [7] y luego por Hakimi utilizando un método más complicado [5].

Teorema 2.3 (Havel – Hakimi) Sea D una secuencia que no aumenta d_1, d_2, \dots, d_n con $n \geq 2$. Dejar D' ser la secuencia derivada de D mediante la eliminación d_1 y restando 1 de cada uno de los primeros. Elementos de la secuencia restante. Entonces D es gráfica si y solo si D' es gráfica.

Esto significa que si nos encontramos con una secuencia de grados que es gráfica durante este proceso, la secuencia de grados inicial es gráfica. Veamos la implementación de este teorema a una secuencia de grados mediante el análisis de la gráfica de la Fig. 2.3 a.



Fig. 2.3 una Un gráfico muestra para implementar Havel-Hakimi teorema b un gráfico que representa secuencia gráfica $\{1, 1, 1, 1\}$ c Una gráfica que representa una secuencia gráfica. $\{0, 1, 1\}$

La secuencia de grados para este gráfico es $\{4, 3, 3, 3, 2, 1\}$. Ahora podemos iterar de la siguiente manera comenzando con la secuencia inicial. Borrar 4 y restar 1 de los primeros 4 de los elementos restantes da

$\{2, 2, 2, 1, 1\}$

continuando de manera similar, obtenemos

$\{1, 1, 0, 0\}$

La última secuencia es gráfica, ya que puede realizarse como se muestra en la Fig. 2.3 b. Este teorema se puede implementar convenientemente usando un algoritmo recursivo debido a su estructura recursiva.

2.2.2 Subgrafos

En muchos casos, estaríamos interesados en parte de una gráfica en lugar de la gráfica en su conjunto. Un subgrafo G' de un gráfico G tiene un subconjunto de vértices de G y un

subconjunto de sus bordes. Es posible que tengamos que buscar un subgrafo de un gráfico que cumpla con alguna condición, por ejemplo, nuestro objetivo puede ser encontrar subgrafos densos que puedan indicar una mayor relación o actividad en esa parte de la red representada por el gráfico.

Definición 2.6 (subgrafo, subgrafo inducido) $G' = (V', E')$ es un subgrafo de $G = (V, E)$. Si $V' \subseteq V$ y $E' \subseteq E$. Un subgrafo $G' = (V', E')$ de un grafo $G = (V, E)$ se llama un subgrafo inducido de G si E' contiene todos los bordes en G que tienen ambos extremos en V' .

Cuando $G' \neq G$, G' se llama un subgrafo apropiado de G ; cuando G' es un subgrafo de G , G se llama un supergraph de G' . Un subgrafo que abarca G' de G es su subgrafo con $V(G) = V(G')$. Del mismo modo, un supergrafo que abarca G de G' tiene el mismo vértice establecido como G' . En la Fig. 2.4 se muestran un subgrafo que abarca y un subgraph inducido de un grafo .

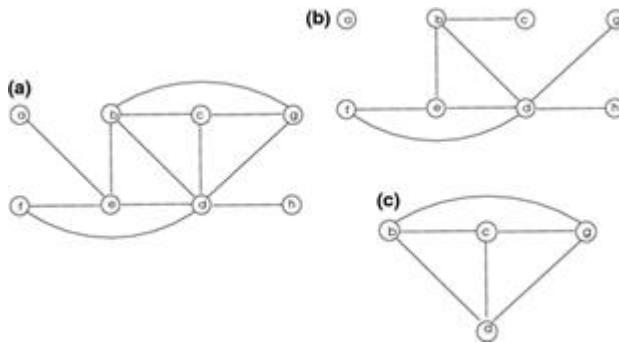


Fig. 2.4 a Un ejemplo de gráfico G , b Un subgrafo que abarca de G , c Un subgrafo inducido de G

Dado un vértice v de una gráfica G , el subgrafo de G mostrado por $G - v$ se forma eliminando el vértice v y todas sus aristas incidentes de G . El subgrafo $G - e$ se obtiene mediante la eliminación del borde e de G . El subgrafo inducido de G por el conjunto de vértices V' se muestra por $G[V']$. El subgrafo $G[V \setminus V']$ se denota por $G - V'$.

Todos los vértices en una gráfica regular tienen el mismo grado. Para un gráfico G , podemos obtener un gráfico H regular que contiene G como un subgrafo inducido. Simplemente duplicamos G junto a sí mismo y unimos cada par de vértices correspondientes por un borde si este vértice no tiene un grado de $\Delta(G)$ como se muestra en la figura 2.5. Si el nuevo gráfico G' no es $\Delta(G)$ -regular, continuamos este proceso duplicando G' . Hasta obtener la gráfica regular. Este resultado se debe a Konig, quien declaró que para cada gráfica de grado máximo r , existe una gráfica r - regular que contiene G como un subgrafo inducido.

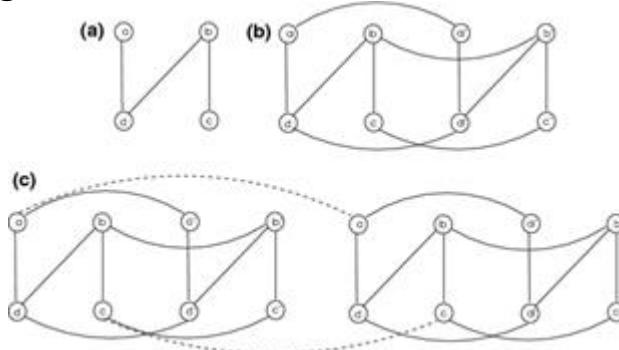


Fig. 2.5 Obtención de un gráfico regular a El gráfico b La primera iteración, c El gráfico de 3 resultados obtenidos en la segunda iteración que se muestra con líneas discontinuas

2.2.3 Isomorfismo

Definición 2.7 (gráfico isomorfismo) Un isomorfismo de un gráfico G_1 a otra grafica G_2 es una biyección $f: V(G_1) \rightarrow V(G_2)$ en el que cualquier borde $(u, v) \in E(G_1)$ si y solo si $f(u)f(v) \in E(G_2)$

Cuando esta condición se mantiene, G_1 se dice que es isomorfo a G_2 o, G_1 y G_2 son isomorfos En la figura 2.6 se muestran tres gráficos isomorfos . Probar si dos gráficas son isomorfas es un problema difícil y no se puede realizar en tiempo polinomial. Un isomorfismo de un gráfico a sí mismo se llama automorfismo . Un gráfico invariante es una propiedad de un gráfico que es igual en sus gráficos isomorfos. Dados dos gráficos isomorfos. G_1 y G_2 , sus órdenes y tamaños son los mismos y sus vértices correspondientes tienen los mismos grados. Por lo tanto, podemos decir que el número de vértices, el número de bordes y las secuencias de grados son invariantes de isomorfismo, es decir, no cambian en los gráficos isomorfos.

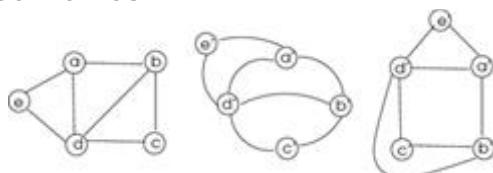


Fig. 2.6 Tres gráficas isomórficas. Vertex x se asigna al vértice x'

2.3 Operaciones gráficas

Es posible que necesitemos generar nuevos gráficos a partir de un conjunto de gráficos de entrada mediante ciertas operaciones. Estas operaciones están uniendo y encontrando la intersección de dos gráficos y encontrando su producto como se describe a continuación.

2.3.1 Unión e intersección

Definición 2.8 (unión e intersección de dos gráficos) La unión de dos gráficos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ es una gráfica $G_3 = (V_3, E_3)$ en el cual $V_3 = V_1 \cup V_2$ y $E_3 = E_1 \cup E_2$. Esta operación se muestra como $G_3 = G_1 \cup G_2$. La intersección de dos gráficos. $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ es una gráfica $G_3 = (V_3, E_3)$ en el cual $V_3 = V_1 \cap V_2$ y $E_3 = E_1 \cap E_2$. Esto se muestra como $G_3 = G_1 \cap G_2$.

La figura 2.7 representa estos conceptos.

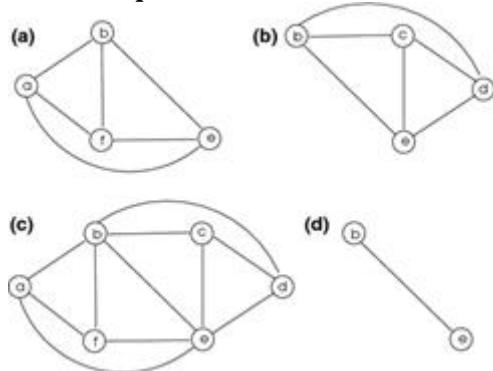


Fig. 2.7 Unión e intersección de dos gráficas. La gráfica en c es la unión de las gráficas en a y b, y la gráfica en d es su intersección.

Definición 2.9 (unión de dos gráficos) La unión de dos gráficos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ es una gráfica $G_3 = (V_3, E_3)$ en el cual $V_3 = V_1 \cup V_2$ y $E_3 = E_1 \cup E_2 \cup \{(u, v) : u \in V_1 \text{ y } v \in V_2\}$. Esta operación se muestra como $G_3 = G_1 \vee G_2$.

La operación de unión de dos gráficos crea nuevos bordes entre cada par de vértices, uno de cada uno de los dos gráficos. La figura 2.8 muestra la unión de dos gráficos. Todas las operaciones de unión, intersección y unión son conmutativas, es decir, $G_1 \cup G_2 = G_2 \cup G_1$, $G_1 \cap G_2 = G_2 \cap G_1$ y $G_1 \vee G_2 = G_2 \vee G_1$.

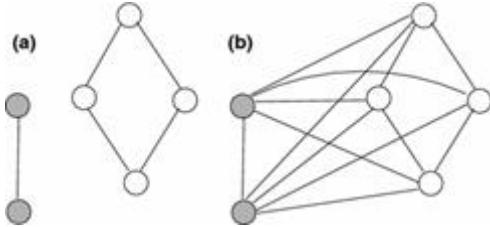


Fig. 2.8 Unión de dos gráficos, a Dos gráficos b Su combinación

2.3.2 Producto cartesiano

Definición 2.10 (producto cartesiano) El producto cartesiano o simplemente el producto de dos gráficos. $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ mostrado por $G_1 \square G_2$ o $G_1 \times G_2$ es una gráfica $G_3 = (V_3, E_3)$ en el cual $V_3 = V_1 \times V_2$ y una ventaja $((u_i, v_j), (u_p, v_q)) \in E_3$ es en $G_1 \times G_2$ si se cumple alguna de las siguientes condiciones:

$$i = p \text{ y } (v_j, v_q) \in E_2$$

1. $j = q$ y $(u_i, u_p) \in E_1$

Informalmente, los vértices que tenemos en el producto son el producto cartesiano de los vértices de los gráficos y, por lo tanto, cada uno representa dos vértices, uno de cada gráfico. La figura 2.9a muestra el producto del gráfico completo K_2 y la gráfica de trayectoria con 4 vértices, P_4 . El producto gráfico es útil en varios casos, por ejemplo, el hipercubo de dimensión n , Q_n , es un gráfico especial que es el producto gráfico de K_2 por sí mismo n veces. Puede ser descrito recursivamente como $Q_n = K_2 \times Q_{n-1}$. En la Fig. 2.9 b se muestra un hipercubo de dimensión 3.

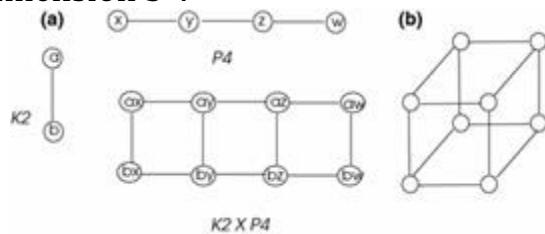


Fig. 2.9 un producto gráfico de K_2 y P_4 b Hipercubo de dimensión 3

2.4 Tipos de gráficos

Revisamos los principales tipos de gráficos que tienen varias aplicaciones en esta sección.

2.4.1 Gráficos completos

Definición 2.11 (gráfico completo) En un gráfico simple completo $G(V, E)$, cada vértice $v \in V$ está conectado a todos los otros vértices en V .

La búsqueda de subgrafos completos de una gráfica G proporciona regiones densas en G , lo que puede significar alguna funcionalidad importante en esa región. Un gráfico completo se denota por K_n donde n es el número de vértices. La figura 2.10 muestra K_1, \dots, K_5 . La gráfica completa con tres vértices, K_3 , se llama triángulo .

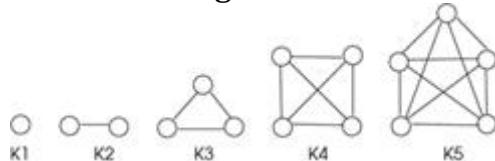


Fig. 2.10 Gráficos completos de tamaños 1 a 5.

El tamaño de un gráfico completo simple no dirigido K_n es $\frac{n(n-1)}{2}$ ya que el grado de cada vértice en K_n es $n-1$, hay n tales vértices, y necesitamos dividir por dos, ya que cada borde se cuenta dos veces para los dos vértices en sus puntos finales.

2.4.2 Gráficos dirigidos

Un borde dirigido o un arco tiene una orientación desde el punto final de la cabeza hasta el punto final de la cola que se muestra con una flecha. Los gráficos dirigidos consisten en bordes dirigidos.

Definición 2.12 (gráfico dirigido) Un gráfico dirigido o un dígrafo consiste en un conjunto de vértices y un conjunto de pares ordenados de vértices llamados bordes dirigidos . Un gráfico parcialmente dirigido tiene bordes tanto dirigidos como no dirigidos.

Si una ventaja $e = (u, v)$ es un borde dirigido en un gráfico dirigido G , decimos que e comienza en u y termina en v , o u es el origen de e y v es su destino, o e se dirige de u a v . El gráfico subyacente de un gráfico dirigido o parcialmente dirigido se obtiene al eliminar las direcciones en todos los bordes y al reemplazar cada borde múltiple por un solo borde. En la figura 2.11 se muestra un gráfico dirigido . A menos que se indique lo contrario, lo que establecemos para los gráficos será válido para los gráficos dirigidos y no dirigidos. En un digrafo simple y completo., hay un par de arcos, uno en cada dirección entre dos vértices cualquiera.

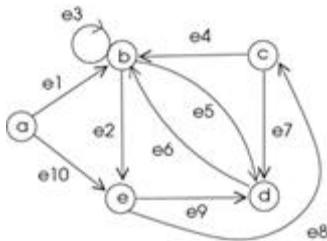


Fig. 2.11 Una gráfica dirigida con $V(G) = \{a, b, c, d, e\}$ y $E(G) = \{e_1, \dots, e_{10}\}$

Definición 2.13 (in-degree, out-degree) El in-grado de un vértice v en un dígrafo es el número de bordes dirigidos a v y el out-grado de v es el número de bordes que se originan en él. El grado de v es la suma de su grado y su grado.

La suma de los grados de los vértices en una gráfica es igual a la suma de los grados de salida que son iguales a la suma del número de bordes. Un gráfico dirigido que no tiene ciclos se denomina gráfico acíclico dirigido (DAG).

2.4.3 Gráficos ponderados

Hemos considerado gráficos no ponderados hasta este punto. Los gráficos ponderados tienen bordes y vértices etiquetados con números reales que representan pesos.

Definición 2.14 (ponderación de bordes, gráficas ponderadas de vértices) Una gráfica de ponderación de bordes $G(V, E, w)$, $w: E \rightarrow \mathbb{R}$. Tiene pesos que consisten en números reales asociados con sus bordes. De manera similar, un gráfico $G(V, E, w)$ ponderado por vértice, $w: V \rightarrow \mathbb{R}$. Tiene pesos de números reales asociados a sus vértices.

Los gráficos ponderados encuentran muchas aplicaciones reales, por ejemplo, el peso de un borde (u, v) puede representar el costo de moverse de u a v como en una carretera o el costo de enviar un mensaje entre dos enrutadores u y v en una red de computadoras. El peso de un vértice v puede asociarse con la capacidad almacenada en v , que puede utilizarse para representar una propiedad, como el volumen de almacenamiento de un enrutador en una red informática.

2.4.4 Gráficos bipartitos

Definición 2.15 (gráfica bipartita) Una gráfica $G = (V, E)$ se denomina gráfico bipartito si el conjunto de vértices V se puede dividir en dos subconjuntos V_1 y V_2 de tal manera que cualquier borde de G conecta un vértice en V_1 a un vértice en V_2 . Es decir, $\forall (u, v) \in E, u \in V_1 \wedge v \in V_2$ o $u \in V_2 \wedge v \in V_1$.

En una gráfica bipartita completa, cada vértice de V_1 está conectado a cada vértice de V_2 y tal gráfico se denota por $K_{m,n}$, donde m es el número de vértices en V_1 y n es el número de vértices en V_2 . El grafo bipartito completo $K_{m,n}$ tiene mn bordes y $m+n$ vértices. En la figura 2.12 se muestra un gráfico bipartito completo ponderado .

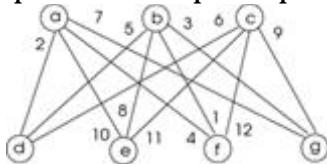


Fig. 2.12 Un gráfico bipartito completo ponderado $K_{3,4}$ con $V_1 = \{a, b, c\}$ y $V_2 = \{d, e, f, g\}$

2.4.5 Gráficos regulares

En una gráfica regular, cada vértice tiene el mismo grado. Cada vértice de una gráfica k -regular tiene un grado de k . Cada k -gráfico completo es un $k-1$ -Gráfico regular pero este último no implica el primero. Por ejemplo, un hipercubo d es un gráfico regular d , pero no es un gráfico completo d . En la figura 2.13 se muestran ejemplos de gráficos regulares . Cualquier gráfico de n ciclo único es un gráfico de 2 regulares. Cualquier gráfico regular con vértices de grados impares debe tener un número par de tales vértices para tener una suma de vértices de número par.

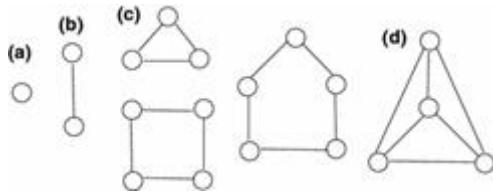


Fig. 2.13 a A 0-gráfico regular, b A 1-gráfico regular, c 2-gráfico regular; d Un gráfico de 3 regulares

2.4.6 Gráficos de líneas

Con el fin de construir una línea de gráfico que $L(G)$ de un simple gráfico G , cada vértice de L que representa un borde de G se forma. Luego, dos vértices u y v de L están conectados si los bordes representados por ellos son adyacentes en G , como se muestra en la Fig. 2.14 .

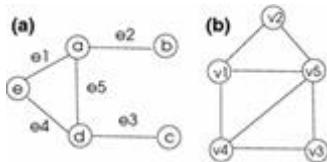


Fig. 2.14 un Un simple gráfico G , b Gráfico de líneas de G

2.5 Caminatas, Caminos y Ciclos

Necesitamos algunas definiciones para especificar el desplazamiento de los bordes y vértices de un gráfico.

Definición 2.16 (caminar) Un paseo W entre dos vértices v_0 y v_n de una gráfica G es una secuencia alterna de $n+1$ vértices y n bordes mostrados como $W = (v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n)$, donde e_i es incidente a los vértices v_{i-1} y v_i . El vértice v_0 se llama el vértice inicial y v_n se llama la terminación de vértice de la caminata W .

En una gráfica dirigida, una caminata dirigida se puede definir de manera similar. La longitud de un recorrido W es el número de bordes (arcos en dígrafos) incluidos en él. Un paseo puede tener aristas y vértices repetidos. Un paseo se cierra si comienza y termina en el mismo vértice y, de lo contrario, se abre . Un paseo se muestra en la figura 2.15 .

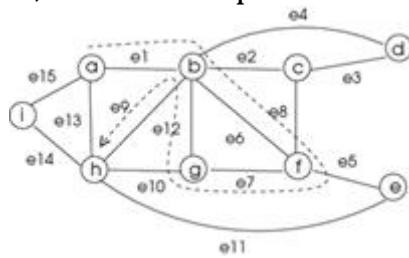


Fig. 2.15 Un paseo. $(a, e_1, b, e_6, f, e_7, g, e_{12}, b, e_9, h)$ en una gráfica de muestra mostrada por una curva discontinua

Un gráfico está conectado si hay un recorrido entre cualquier par de sus vértices. La conectividad es un concepto importante que encuentra muchas aplicaciones en las redes de computadoras y revisaremos los algoritmos de conectividad en el Capítulo 8 .

Definición 2.17 (rastro) Un sendero es un paseo que no tiene ningún bordes repetidos.

Definición 2.18 (ruta) Una ruta es una ruta que no tiene ningún vértice repetido con la excepción de los vértices iniciales y terminales.

En otras palabras, una ruta no contiene bordes o vértices repetidos. Las rutas se muestran únicamente por los vértices. Por ejemplo, (i , a , b , h , g) es una ruta en la figura 2.15 .

Definición 2.19 (ciclo) Una trayectoria cerrada que comienza y termina en el mismo vértice se denomina ciclo .

La longitud de un ciclo puede ser un entero impar, en cuyo caso se denomina ciclo impar . De lo contrario, se llama un ciclo par .

Definición 2.20 (circuito) Un camino cerrado que comienza y termina en el mismo vértice se denomina circuito .

Todos estos conceptos se resumen en la Tabla 2.1 .

Tabla 2.1 Propiedades de los recorridos de gráficos simples

Vértices repetidos	Bordes repetidos
Camina Sí	Sí
Caminos Sí	No
Caminos No excepto vértice inicial y terminal.	No
Circuitos Sí, comienza y termina en el mismo vértice. No	No
Ciclos No excepto inicial y plazo	No

Un sendero en la figura 2.15 es $(h, e_9, b, e_6, f, e_5, e)$. Cuando e_{11} y h se agregan a este camino, se convierte en un ciclo. Un recorrido de Euler es un sendero cerrado de Euler y un gráfico de Euler es un gráfico que tiene un recorrido de Euler. El número de aristas contenidas en un ciclo se denota su longitud l y el ciclo se muestra como C_l . Por ejemplo, C_3 es un triangulo

Definición 2.21 (Ciclo hamiltoniano) Un ciclo que incluye todos los vértices en un gráfico se denomina ciclo hamiltoniano y dicho gráfico se llama hamiltoniano . Un camino de Hamilton de un gráfico de G pasa a través de cada vértice de G .

2.5.1 Conectividad y Distancia

Nos interesaría encontrar si podemos alcanzar un vértice v desde un vértice u . En un gráfico G conectado , hay una ruta entre cada par de vértices. De lo contrario, G se desconecta . Los subgrafos conectados de un grafo desconectado se llaman componentes . Un gráfico conectado en sí es el único componente que tiene. Si el gráfico subyacente de un dígrafo G está conectado , G está conectado. Si hay un paseo dirigido entre cada par de vértices, G está fuertemente conectado .

En un gráfico conectado , es interesante descubrir cuán fácil es alcanzar un vértice desde otro. El parámetro de distancia definido a continuación proporciona esta información.

Definición 2.22 (distancia) La distancia d (u , v) entre dos vértices u y v en un gráfico G (dirigido) es la longitud de la ruta más corta entre ellos.

En un gráfico no ponderado (dirigido), d (u , v) es el número de bordes de la ruta más corta entre ellos. En un gráfico ponderado, esta distancia es la suma mínima de los pesos de los bordes de un camino fuera de todos los caminos entre estos vértices. El camino más corto entre dos vértices u y v en un gráfico G es otro término usado en lugar de la distancia entre los vértices u y v para tener el mismo significado. Los caminos más cortos entre los vértices h y e son h , b , f , e y h , g , f , e ambos con una distancia de 3 en la

figura 2.15 . Del mismo modo, las rutas más cortas entre los vértices i y b son $i \rightarrow b \rightarrow i$, $b \rightarrow i \rightarrow b$, tanto con una distancia de 2. La distancia dirigida desde un vértice u a v en un dígrafo es la longitud del pie más corta desde u a v . En un gráfico simple (ponderado) no dirigido, $G(V, E, w)$, se puede afirmar lo siguiente:

$$d(u, v) = d(v, u)$$

- 1.
2. $d(u, w) \leq d(u, v) + d(v, w) \quad \forall w \in V$

Definición 2.23 (excentricidad) La excentricidad de un vértice v en un gráfico conectado G es su máxima distancia a cualquier vértice en G .

La excentricidad máxima se denomina diámetro y el valor mínimo de este parámetro se llama el radio del gráfico. El vértice v de un gráfico de G con la excentricidad mínima en un gráfico conectado G se llama el vértice el centro de G . Encontrar el vértice central de un gráfico tiene implicaciones prácticas, por ejemplo, podemos querer ubicar un centro de recursos en una ubicación central en un área geográfica donde las ciudades estén representadas por los vértices de un gráfico y las carreteras por sus bordes. Puede haber más de un vértice central.

2.6 Representaciones gráficas

Necesitamos representar gráficos en formas adecuadas para poder realizar cálculos en ellos. Dos formas de representación ampliamente utilizadas son la matriz de adyacencia y los métodos de lista de adyacencia .

2.6.1 Matriz de adyacencia

Una matriz de adyacencia de un gráfico simple o un dígrafo es una matriz $A[n, n]$ donde cada elemento $a_{ij} = 1$ si hay un borde que une el vértice i a j y $a_{ij} = 0$ de otra manera. Para multigrafos, la entrada a_{ij} es igual al número de bordes entre los vértices i y j . Para un digraph, a_{ij} muestra el número de arcos desde el vértice i al vértice j . La matriz de adyacencia es simétrica para un gráfico no dirigido y es asimétrica para un dígrafo en general. Un dígrafo y su matriz de adyacencia se muestran en la Fig. 2.16 . Una matriz de adyacencia requiere $O(n^2)$ espacio.

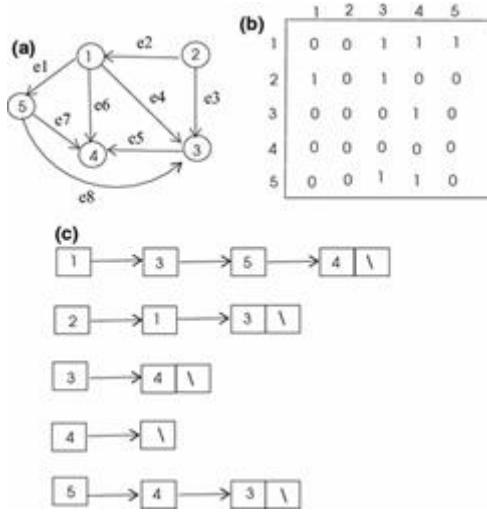


Fig. 2.16 a Dígrafo, b Matriz de adyacencia, c Lista de adyacencia. El final de las entradas de la lista están marcados con una barra invertida

2.6.2 Lista de adyacencia

Una lista de adyacencia de un gráfico simple (o un dígrafo) es una matriz de listas con cada lista que representa un vértice y sus vecinos (de salida) en una lista vinculada. El final de la lista está marcado con un puntero NULO. La lista de adyacencia de un gráfico se muestra en la figura 2.16 c. La matriz de adyacencia requiere $O(n + m)$ espacio. Para gráficos dispersos, se prefiere la lista de adyacencia debido a la dependencia del espacio en el número de vértices y aristas. Para gráficos densos, la matriz de adyacencia se usa comúnmente para buscar la existencia de un borde en esta matriz que se puede hacer en tiempo constante. Con la lista de adyacencia de un gráfico, el tiempo requerido para la misma operación es $O(n)$.

En un digraph $G = (V, E)$, la lista de predecesores $P_v \subseteq V$ de un vértice v se define de la siguiente manera.

$$P_v = \{u \in V : (u, v) \in E\}$$

y la lista sucesora de v , $S_v \subseteq V$ es,

$$S_v = \{u \in V : (v, u) \in E\}$$

Las listas de predecesores y sucesores de los vértices de la gráfica de la Fig. 2.16 a se enumeran en la Tabla 2.2.

Tabla 2.2 Listas de predecesores y listas de sucesores de vértices

v	P_v	S_v
1	{2}	{3, 4, 5}
2	{}	{1, 3}
3	{1, 2, 5}	{4}
4	{1, 3, 5}	{}
5	{1}	{3, 4}

2.6.3 Matriz de incidencia

Una matriz de incidentes $B [n , m]$ de un gráfico simple tiene elementos $b_{ij} = 1$ si el borde j es incidente al vértice i y $b_{ij} = 0$ de otra manera. La matriz de incidencia para un dígrafo se define de manera diferente como se muestra a continuación.

$$b_{ve} = \begin{cases} -1 & \text{if arc } e \text{ ends at vertex } v \\ 1 & \text{if arc } e \text{ starts at vertex } v \\ 0 & \text{otherwise} \end{cases}$$

La matriz de incidencia de la gráfica de la figura 2.16 a es la siguiente:

$$B = \begin{pmatrix} 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

En la representación de la lista de bordes de un gráfico, todos sus bordes se incluyen en la lista.

2.7 árboles

Una gráfica se llama árbol si está conectada y no contiene ningún ciclo. Las siguientes afirmaciones definen igualmente un árbol T :

T está conectado y tiene $n - 1$ bordes

- 1.
2. Cualquiera de los dos vértices de T están conectados exactamente por un camino.
3. T está conectado y cada borde es una eliminación cortar-borde de que desconecta T .

En un árbol arraigado T , hay un vértice especial r llamado raíz y todos los demás vértices de T tienen un camino dirigido a r ; El árbol está desraoteado de lo contrario. En la figura 2.17 se representa un árbol enraizado. Un árbol de expansión de una gráfica G es un árbol que cubre todos los vértices de G . Un árbol de expansión mínimo de un gráfico ponderado G es un árbol de expansión de G con el peso total mínima entre todos los árboles de expansión de G . Investigaremos estructuras de árbol y algoritmos con más detalle en el capítulo 6.

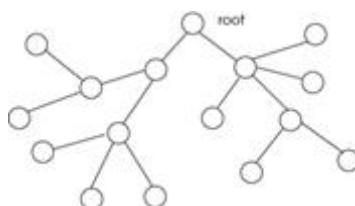


Fig. 2.17. Una estructura general del árbol.

2.8 Gráficos y matrices.

El análisis espectral de gráficos involucra operaciones en las matrices relacionadas con gráficos.

2.8.1 Valores propios

Multiplicando un $n \times n$ matriz A por una $n \times 1$ vector x resulta en otro $n \times 1$ vector y que se puede escribir como $Ax = y$ en forma de ecuación. En lugar de obtener un nuevo vector y, sería interesante saber si el producto Ax es igual a un vector que es una constante multiplicada por el vector x como se muestra a continuación.

$$Ax = \lambda x$$

(2.4)

Si podemos encontrar valores de x y λ para que esta ecuación se mantenga, λ se llama un valor propio de A y x como un vector propio de A. Habrá una serie de valores propios y un conjunto de vectores propios correspondientes a estos valores propios en general. Reescritura de la ec. 2.4 ,

$$Ax - \lambda x = 0$$

(2.5)

$$(A - \lambda I)x = 0$$

$$\det(A - \lambda I)x = 0$$

(2.6)

Para A [n , n], $\det(A - \lambda I) = 0$ se llama el polinomio característico que tiene un grado de n y por lo tanto tiene n raíces. Resolver este polinomio nos proporciona valores propios y sustituirlos en la ecuación. 2.4 proporciona los vectores propios de la matriz A .

2.8.2 Matriz laplaciana

Definición 2.24 (matriz de grado) La matriz de grado D de un gráfico G es la matriz diagonal con elementos d_1, \dots, d_n donde d_i es el grado de vértice i .

Definición 2.25 (Matriz laplaciana) La matriz Laplaciana L de un gráfico simple se obtiene restando su matriz de adyacencia de su matriz de grados.

Las entradas de la matriz Laplaciana L son las siguientes.

$$l_{ij} = \begin{cases} d_i & \text{if } i = j \\ -1 & \text{if } i \text{ and } j \text{ are neighbors} \\ 0 & \text{otherwise.} \end{cases}$$

La matriz L de Laplacía se denomina a veces Laplaciano combinatorio. El conjunto de todos los valores propios de la matriz laplaciana de un gráfico de G se denomina espectro laplaciano de G. La matriz laplaciana normalizada.^L Está estrechamente relacionado con el laplaciano combinatorio. La relación entre estas dos matrices es la siguiente:

$$\mathcal{L} = D^{-1/2} L D^{-1/2} = D^{-1/2}(D - A)D^{-1/2} = I - D^{-1/2}AD^{-1/2}.$$

(2.7)

Las entradas de la matriz laplaciana normalizada.^L entonces se puede especificar como a continuación.

$$\mathcal{L}_{ij} = \begin{cases} 1 & \text{if } i = j \\ \frac{-1}{\sqrt{d_i d_j}} & \text{if } i \text{ and } j \text{ are neighbors} \\ 0 & \text{otherwise.} \end{cases}$$

La matriz laplaciana y la matriz de adyacencia de un gráfico G se usan comúnmente para analizar las propiedades espectrales de G y diseñar algoritmos de gráficos algebraicos para resolver varios problemas de gráficos.

2.9 Notas del capítulo

Hemos revisado los conceptos básicos de la teoría de grafos, dejando el estudio de algunos de los antecedentes relacionados, incluidos los árboles, la conectividad, el emparejamiento, los flujos de red y la coloración a la Parte II cuando analizamos los algoritmos para estos problemas. El fondo principal de la teoría de grafos se presenta en varios libros, incluidos los libros de Harary [6], Bondy and Murty [1] y West [8]. La teoría de grafos con aplicaciones se estudia en un libro editado por Gross et al. [4]. La teoría de gráficos algorítmicos se centra más en los aspectos algorítmicos de la teoría de gráficos y los libros disponibles en este tema incluyen los libros de Gibbons [2], Columbic [3].

Ceremonias

Demuestre que la relación entre el tamaño y el orden de un gráfico simple está dada por $m \leq (n/2)^2$ Y decidir cuándo se mantiene la igualdad.

- 1.
2. Encuentra el orden de un gráfico de 4 regulares que tiene un tamaño de 28.
3. Demuestre que los grados mínimo y máximo de un gráfico G están relacionados por $\delta(G) \leq 2m/n \leq \Delta(G)$ desigualdad.
4. Demuestre que para un grafo bipartito regular $G = (V1, V2, E)$, $|V1| = |V2|$.

5. Dejar $G = (V_1, V_2, E)$ Ser un gráfico bipartito con particiones de vértice. V_1 y V_2 . Muestra esa

$$\sum_{u \in V_1} \deg(u) = \sum_{v \in V_2} \deg(v)$$

6. Una gráfica G tiene una secuencia de grados. $D = (d_1, d_2, \dots, d_n)$. Encuentra la secuencia de grados del complemento. \bar{G} de este grafo.
7. Demuestre que la unión de los complementos de un gráfico completo. K_p y otro grafo completo K_q es un grafo bipartito completo $K_{p,q}$.
8. Dibuja la gráfica lineal de K_3 .
9. Dejar $G = (V, E)$ ser una gráfica donde $\forall v \in V, \deg(v) \geq 2$. Demuestra que G contiene un ciclo.
10. Demuestre que si se desconecta un simple gráfico G , su complemento \bar{G} está conectado.

Referencias

1. Bondy AB, Murty USR (2008) Teoría de grafos. Postgrado en matematicas. Springer, Berlín. 1^a edición corregida 2008. 3^a edición 2008 (28 de agosto de 2008). ISBN-10: 1846289696, ISBN-13: 978-1846289699
2. Gibbons A (1985) Teoría de gráficos algorítmicos, 1^a ed . Cambridge University Press, Cambridge. ISBN-10: 0521288819, ISBN-13: 978-0521288811
3. Golumbic MC, Rheinboldt W (2004) Teoría de gráficos algorítmicos y gráficos perfectos, vol. 57, 2^a ed . Anales de la matemática discreta. Holanda Septentrional, Nueva York. ISBN-10: 0444515305, ISBN-13: 978-0444515308
4. Gross JL, Yellen J, Zhang P (eds) (2013) Manual de teoría de grafos, 2nd edn . Prensa CRC, Boca Raton
5. Hakimi SL (1962) Sobre la realizabilidad de un conjunto de enteros como grados de los vértices de una gráfica. SIAM J Appl Math 10 (1962): 496–506
[MathSciNet](#) [Crossref](#)
6. Harary F (1969) Teoría de grafos. Serie de Addison Wesley en Matemáticas. Addison – Wesley, Lectura
[Referencia cruzada](#)
7. Havel V (1955) Un comentario sobre la existencia de gráficos finitos. Casopis Pest Mat 890 (1955): 477–480
[MATES](#)
8. West D (2000) Introducción a la teoría de grafos, 2^a ed . PHI aprendizaje. Prentice Hall, Englewood Cliffs

3. Algoritmos de grafos

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege, Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Nosotros revisamos básica estructura de algoritmo y proporcionamos una breve y denso revisión de los principios fundamentales de diseño de algoritmos y análisis secuencial con el foco en algoritmos de grafos. A continuación, proporcionamos una breve encuesta de NP completa con ejemplos de problemas de NP-hard graph. Finalmente, revisamos brevemente los principales métodos de diseño de algoritmos que muestran sus implementaciones para problemas de gráficos.

3.1 Introducción

Un algoritmo es un conjunto finito de instrucciones para resolver un problema dado. Recibe un conjunto de entradas y calcula un conjunto de salidas utilizando las entradas. El diseño y análisis de algoritmos ha sido un tema clave en cualquier plan de estudios de informática.

Ha habido un interés creciente en el estudio de algoritmos para problemas teóricos de gráficos en las últimas décadas, principalmente debido a las numerosas aplicaciones cada vez mayores de gráficos en la vida real. Además, los avances tecnológicos recientes proporcionaron datos de redes muy grandes, como las redes biológicas, las redes sociales, la Web e Internet, que comúnmente se denominan redes complejas que se pueden representar mediante gráficos. Los problemas encontrados en estas redes pueden ser muy diferentes a los estudiados en la teoría de grafos clásica, ya que estas redes tienen tamaños grandes y no tienen estructuras aleatorias. Por lo tanto, hay una necesidad de nuevos métodos y algoritmos en estas redes.

El estudio de los algoritmos de gráficos se informa en varios encabezados, que incluyen la teoría de algoritmos de gráficos algorítmicos, gráficos y aplicaciones. Nuestro objetivo principal en este capítulo es proporcionar una breve y densa revisión de los principios fundamentales del diseño y análisis de algoritmos secuenciales con énfasis en los algoritmos de gráficos. Primero describimos los conceptos básicos de los algoritmos y luego revisamos las matemáticas detrás del análisis de los algoritmos. A continuación, proporcionamos una breve encuesta sobre la integridad de NP con un enfoque en los problemas de gráfica de NP. Cuando estamos lidiando con problemas gráficos tan difíciles, podemos usar algoritmos de aproximación que proporcionan soluciones subóptimas con

relaciones de aproximación probadas. Sin embargo, en muchos casos, nuestra única opción es utilizar algoritmos heurísticos que funcionen para la mayoría de las combinaciones de entrada, como describimos. Finalmente,

3.2 Fundamentos

Un algoritmo es un conjunto de instrucciones que trabajan en alguna entrada para producir una salida útil. Las propiedades fundamentales de cualquier algoritmo, así como cualquier algoritmo de gráfico, son las siguientes:

- Acepta un conjunto de entradas , procesa estas entradas y produce algunas salidas útiles .
- Debe proporcionar la salida correcta . La corrección es un requisito fundamental de cualquier algoritmo.
- Debe ejecutar un número finito de pasos para producir la salida. En otras palabras, no queremos que el algoritmo se ejecute para siempre sin producir ningún resultado. Es posible tener algoritmos que se ejecutan infinitamente, como los programas del servidor, pero estos producen resultados mientras se ejecutan.
- Un algoritmo debe ser efectivo . Debe realizar la tarea requerida utilizando un número mínimo de pasos. No tiene sentido tener un algoritmo que se ejecute 2 días para estimar el clima para mañana, ya que sabemos lo que sería para ese entonces. Dados dos algoritmos que realizan la misma tarea, preferiríamos el que tiene menos pasos.

Al presentar un algoritmo en este libro y en general, primero debemos proporcionar una descripción simple de la idea principal del algoritmo. Entonces tendríamos que detallar su operación usando pseudocódigo.sintaxis que muestra su ejecución utilizando estructuras básicas como se describe a continuación. Pseudocódigo es la descripción de un algoritmo de una manera más formal y estructurada que una descripción verbal pero menos formal que un lenguaje de programación. Luego, normalmente mostramos una operación de ejemplo del algoritmo en un gráfico de muestra. La segunda cosa fundamental que se debe hacer es demostrar que el algoritmo funciona correctamente, lo cual es evidente en muchos casos, trivial en algunos casos y necesita técnicas de prueba rigurosas para varios otros, como describimos en este capítulo. Finalmente, debemos presentar el análisis del peor de los casos, que muestra su desempeño como el número de pasos requeridos en el peor de los casos.

3.2.1 Estructuras

Tres estructuras fundamentales utilizadas en los algoritmos son la asignación , la decisión y los bucles . Una asignación proporciona la asignación de un valor a una variable como se muestra a continuación:

$b \leftarrow 3$

Aquí asignamos un valor entero de 3 a una variable b y luego asignamos el cuadrado de b a a . El valor final de a es 9. Las decisiones son las estructuras clave en los algoritmos como en la vida diaria. Necesitamos derivar a alguna parte del algoritmo basado en alguna condición. El siguiente ejemplo usa la estructura if ... then ... else para determinar el mayor de dos números de entrada:

```
input a,b  
if a > b then print a  
  else if a=b then print "they are equal"  
  else print b  
end if
```

Otra estructura fundamental en los algoritmos es la estructura de bucle para realizar una acción, respectivamente. Hay tres formas principales de realizar bucles en un algoritmo de la siguiente manera.

- Para bucles: los bucles for se utilizan comúnmente cuando sabemos cuántas veces debe ejecutarse el bucle antes de comenzar con el bucle. Hay una variable de bucle y condición de prueba. La variable de bucle se modifica al final del bucle de acuerdo con la línea de inicio y se compara con la condición especificada en la línea de prueba. Si esta condición produce un valor verdadero, se introduce el bucle. En el siguiente ejemplo, i es la variable de bucle y se incrementa en 1 al final de cada iteración de bucle y se compara con el valor del límite de 10. Este bucle simple imprime los cuadrados de enteros entre 1 y 10.

```
for i=1 to 10 step 1  
  print i * i  
end for
```

- Mientras bucles: En caso de que no sabemos cuántas veces se ejecutará el bucle, mientras que la estructura puede ser utilizado. Tenemos una condición de prueba al comienzo del bucle y si esto tiene éxito, se ingresa el bucle. El siguiente ejemplo ilustra el uso del bucle while en el que ingresamos Q para que el usuario lo abandone y, de lo contrario, sumamos los dos números dados por el usuario. No sabemos cuándo el usuario puede querer detenerse, por lo que el uso de while es apropiado aquí. También tenga en cuenta que necesitamos dos instrucciones de entrada para el control, una fuera del bucle que se ejecutará una vez y otra dentro del bucle para probar iterativamente ya que la verificación se encuentra al principio del bucle.

```
input chr  
while chr <> 'Q'  
  input a,b  
  print a+b  
  input chr  
end while
```

- Repetir .. hasta bucles: Esta estructura se utiliza en situaciones similares a mientras que los bucles cuando no sabemos cuántas veces se ejecutará el bucle. La principal diferencia es que hacemos la prueba al final del bucle, lo que significa que este bucle se ejecuta al menos una vez, mientras que el bucle while no puede ejecutarse ni una sola vez. Escribiremos el ejemplo anterior con la repetición ... hasta (o bucle ... hasta) estructura con un código claramente más corto.

```

repeat
    input a,b
    print a+b
until chr <> 'Q'

```

3.2.2 Procedimientos y funciones

Un procedimiento o una función es un bloque de código independiente para realizar una tarea específica. Estos módulos se proporcionan en lenguajes de programación para dividir grandes programas o tareas en pequeños para facilitar la depuración y el mantenimiento. Además, un procedimiento o una función se puede usar más de una vez, lo que resulta en un código simplificado con menos requisitos de espacio. Un procedimiento o una función se implementa utilizando el método de llamada / retorno. Se llaman desde el programa principal por la rutina de llamada . Un procedimiento o una función pueden ingresar parámetros para trabajar, y terminan con una declaración de retorno que lleva al programa en ejecución al punto posterior al que se llama. Una función siempre devuelve un valor, mientras que un procedimiento no puede. El algoritmo 3.1 muestra un procedimiento llamado Countque se llama desde el programa principal con el parámetro k . Muestra todos los enteros entre 1 y k . La ejecución de este algoritmo mostrará 1, 1 2, 1 2 3 y 1 2 3 4 en la salida llamando al procedimiento cuatro veces.

Algorithm 3.1 Proc_Example

```

1: Input : int n = 4
2: int i
3: procedure COUNT(k)
4:   int j = 1
5:   while j ≤ k do
6:     output j
7:     j ← j + 1
8:   end while
9: end procedure
10: for (i=1 to n do
11:   COUNT(i)
12: end for

```

▷ algorithm variable i
 ▷ procedure input variable k
 ▷ procedure local variable j

 ▷ main body of the algorithm
 ▷ procedure call

3.3 Análisis asintótico

Necesitamos evaluar el tiempo de ejecución de los algoritmos para varios tamaños de entrada para evaluar su desempeño. Podemos evaluar el comportamiento de un algoritmo de manera experimental, pero en la práctica se necesita una determinación teórica del número requerido de pasos en función del tamaño de entrada. Ilustremos estos conceptos escribiendo un algoritmo que busque un valor entero de clave en una matriz de enteros y devuelva su primera aparición como el índice de la matriz como se muestra en el Algoritmo 3.2. Queremos averiguar el tiempo de ejecución de este algoritmo como el número de pasos ejecutados y si podemos encontrar un algoritmo que tenga menos pasos para el mismo proceso, preferiríamos ese algoritmo. En dicho análisis, no estamos interesados en el número constante de pasos, sino que necesitamos encontrar el número de pasos necesarios a medida que aumenta el tamaño de la entrada. Por ejemplo,i necesita tiempo constante, sino que se realiza una sola vez, por lo tanto se puede despreciar, y esto es más significativo cuando $n \gg 1$. El número de veces que se ejecuta el bucle es importante, ya que afecta significativamente el rendimiento del algoritmo. Sin embargo, el número de pasos, digamos 2 o 3, dentro del bucle es nuevamente insignificante ya que $2n$ o $3n$ es invariable cuando n es muy grande.

Algorithm 3.2 Search_Key

```
1: Input : array  $A[n]$  of  $n$  integers, integer  $key$ 
2: Output : the first location of  $key$  or NOT_FOUND if it is not in  $A[n]$ 
3: int  $i$ 
4: for  $i=1$  to  $n$  do
5:   if  $key = A[i]$  then
6:     return  $i$ 
7:   end if
8: end for
9: return NOT_FOUND
```

Cuando ejecutamos este algoritmo, es posible que el valor clave se encuentre en la primera entrada de la matriz, en cuyo caso el algoritmo se completa en un paso. Este será el tiempo de ejecución más bajo del algoritmo. En el peor de los casos, necesitamos verificar cada entrada de la matriz A , ejecutando el bucle n veces. Estamos interesados principalmente en el peor tiempo de ejecución de un algoritmo, ya que esto es lo que puede esperarse como el peor de los casos.

Necesitamos analizar el tiempo de ejecución y los requisitos de espacio de un algoritmo como funciones del tamaño de entrada. Nuestro interés es determinar el comportamiento asintótico de estas funciones cuando el tamaño de entrada es muy grande. La cantidad de pasos necesarios para ejecutar un algoritmo se denomina complejidad de tiempo. Este parámetro se puede especificar de tres maneras: las complejidades de mejor caso, promedio y peor caso se describen a continuación, asumiendo que f y g son funciones de \mathbb{N} a \mathbb{R}^+ y n es el tamaño de entrada.

El peor análisis de caso

El peor tiempo de ejecución de un algoritmo es $f(n) = O(g(n))$, si existe una constante $c > 0$ tal que $\forall n_0 \geq n, f(n) \leq cg(n)$. Esto también se denomina notación de gran oh y establece que el tiempo de ejecución está limitado por una función $g(n)$ multiplicada por una constante cuando el tamaño de entrada es mayor o igual que un valor de entrada de umbral. Hay muchas funciones $O(g(n))$ para $f(n)$ pero buscamos el valor más pequeño posible para seleccionar un límite superior ajustado en $f(n)$.

Ejemplo 3.1 Let $f(n) = 5n + 3$ para un algoritmo, lo que significa que su tiempo de ejecución es esta función lineal de su tamaño de entrada n . Podemos tener una función $g(n) = n^2$ y $n_0 = 6$, y por lo tanto reclamo $cg(n) \geq f(n), \forall n \geq n_0$. Por lo tanto, $5n + 3 \in O(n^2)$ lo que significa que $f(n)$ tiene la peor complejidad de $O(n^2)$. Tenga en cuenta que cualquier complejidad mayor que n^2 , por ejemplo, $O(n^3)$, también es una complejidad válida en el peor momento para $f(n)$. De hecho, $O(n)$ es una complejidad más cercana para el peor de los casos para este algoritmo, ya que esta función se aproxima a n en el límite cuando n es muy grande. Normalmente lo adivinaríamos y procederíamos de la siguiente manera:

$$5n + 3 \leq cn$$

$$(c - 5)n \geq 3$$

$$n \geq 3/(c - 5)$$

Podemos seleccionar $c = 6$ y $n_0 = 4$ para que esta desigualdad se mantenga y, por tanto, complete la prueba de que $5n + 3 \in O(n)$. Como otro ejemplo, considere la complejidad del tiempo de $4\log n + 7$. Afirmamos que esto es $\Theta(\log n)$ y necesito encontrar c y n_0 valores tales que $4\log n + 7 \leq c\log n$ para $n \geq n_0$ que vale para $c = 12$ y $n_0 = 2$.

El análisis del mejor caso

El mejor o el menor tiempo de ejecución de un algoritmo, que también es el número mínimo de pasos para ejecutarlo, es $f(n) = \Omega(g(n))$, si existe una constante $c > 0$ tal que $f(n) \geq cg(n)$. De manera informal, este es el mejor tiempo de ejecución del algoritmo entre todas las entradas de tamaño n . En general, este parámetro no proporciona mucha información sobre el rendimiento general del algoritmo, ya que el algoritmo puede ser lento en otras combinaciones de entrada. Por lo tanto, puede que no sea confiable comparar los algoritmos en función de sus mejores tiempos de ejecución, pero este parámetro aún nos da una idea de qué esperar mejor.

Ejemplo 3.2 Sea $f(n) = 3\log n + 2$ para un algoritmo, y consideremos la función $g(n) = \log n$ para ser un límite inferior en el tiempo de ejecución del algoritmo. En este caso, tenemos que verificar $3\log n + 2 \geq c\log n$ para alguna constante c para todos $n \geq n_0$ valores para un umbral n_0 valor.

$$3\log n + 2 \geq c\log n$$

$$(3 - c)\log n \geq -2$$

$$\log n \geq -2/(3 - c)$$

y para $n_0 = 2$ y $c = 4$, esta ecuación se mantiene y por lo tanto afirma $3\log n + 2 \in \Omega(\log n)$. El punto clave aquí fue adivinar que esta función crece al menos a medida que $\log n$.

Notación theta

$\Theta(n)$ es el conjunto de funciones que crece al mismo ritmo que $f(n)$ y se considera un límite estricto para $f(n)$. Estas funciones están tanto en $O(g(n))$ y $\Omega(g(n))$. Formalmente, $g(n) \in \Theta(n)$ si existen constantes n_0 , c_1 y c_2 tal que $\forall n \geq n_0$, $c_1f(n) \leq g(n) \leq c_2f(n)$. La relación entre la tasa de crecimiento de una función $f(n)$, $O(n)$, $\Theta(n)$ y $\Omega(n)$ se representa en la figura 3.1.

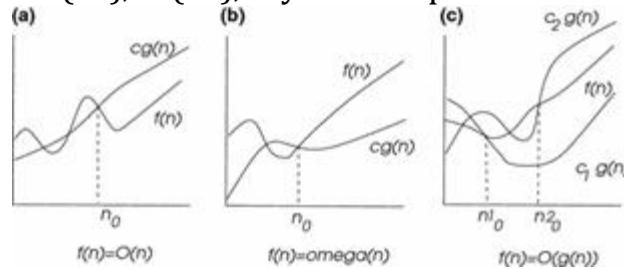


Fig. 3.1 La tasa de crecimiento de una función.

El análisis de casos promedio

Nuestro objetivo al determinar el caso promedio es encontrar el tiempo de ejecución esperado del algoritmo utilizando una entrada seleccionada al azar, asumiendo una distribución de probabilidad sobre las entradas de tamaño n . Este método en general es más difícil de evaluar que el peor o el mejor de los casos, ya que requiere un análisis probabilístico, pero puede proporcionar resultados más significativos. Otro punto de preocupación es el espacio de memoria que necesita un algoritmo. Esto se especifica como el número máximo de bits requerido por el algoritmo y se denomina complejidad de espacio del algoritmo.

Reglas generales

- El orden de crecimiento de los peores casos comúnmente encontrados en orden creciente es el siguiente:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3), \dots, O(n^k) \subset 2^n \subset O(n!) \subset O(n^n)$$

- Cuando el tiempo de ejecución de un algoritmo se determina como un polinomio, ignore los términos de orden inferior y el factor constante del término más grande, ya que tendrán un efecto muy pequeño en el tiempo de ejecución cuando el tamaño de entrada es muy grande. Por ejemplo, si el tiempo de ejecución de un algoritmo es $\frac{5n^4 + 3n^2 + 2n + 5}{O(n^4)}$, su complejidad en el peor de los casos es $O(n^4)$.
- Para cualquier constante c , $O(cf(n)) = O(f(n))$
- La suma de las complejidades de tiempo de peor caso de dos algoritmos es la complejidad de peor caso de la suma de los dos algoritmos de la siguiente manera.

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

- Del mismo modo, $O(f(n))O(g(n)) = O(f(n)g(n))$
- Desde $\log_{10} n \in \Theta(\log_2 n)$ $\log_{10} n = \frac{\log_2 n}{\log_2 10} = \frac{\log_2 n}{3.32}$ cual es $\log_2 n / 3.32 \in \Theta(\log_2 n)$. En general,

Aunque el análisis asintótico muestra el tiempo de ejecución del algoritmo a medida que el tamaño de la entrada aumenta a valores muy grandes, es posible que tengamos que trabajar solo con tamaños de entrada pequeños, lo que significa que los términos de orden inferior y las constantes pueden no ser ignorados. Además, dados los dos algoritmos, la elección de uno con una mejor complejidad promedio en lugar de uno con una mejor complejidad en el peor de los casos sería más sensata, ya que cubriría la mayoría de los casos.

3.4 Algoritmos recursivos y recurrencias

Tenemos dos métodos principales de diseño e implementación de algoritmos: recursión e iteración. Un algoritmo recursivo es el que se llama a sí mismo y estos algoritmos se emplean comúnmente para dividir un problema grande en partes más pequeñas, resolver las partes más pequeñas y luego combinar los resultados como veremos en este capítulo. Las soluciones iterativas siguen repitiendo el mismo

procedimiento hasta que se cumple la condición deseada. Los algoritmos recursivos comúnmente proporcionan soluciones más cortas pero pueden ser más difíciles de diseñar y analizar. Consideremos una función recursiva Potencia para encontrar la potencia n th de un entero x . La solución iterativa involucraría n veces la multiplicación de x por sí mismo en un for u otro tipo de bucle. En el algoritmo recursivo, tenemos la función que se llama a sí misma con valores decrecientes de n cada vez hasta que se encuentra el caso base cuando $n = 0$. Las llamadas anidadas a esta función comienzan a regresar a la persona que llama después de este punto, multiplicando cada vez x con el valor devuelto. El primer valor devuelto de la última llamada es 1 seguido de x^2 , x^3 hasta x^n como se muestra en el algoritmo 3.3.

Algorithm 3.3 Power

```

1: function POWER(x,n)
2:   if n = 0 then
3:     return 1
4:   else
5:     return x × power(x, n - 1)
6:   end if
7: end function

```

Para analizar el tiempo de ejecución de los algoritmos recursivos, necesitamos definir las relaciones de recurrencia, que son relaciones definidas recursivamente en términos de ellos mismos. Intentemos formar la relación de recurrencia para el algoritmo de poder recursivo definido anteriormente. Deje que $T(x)$ indique el tiempo empleado cuando x se ingresa a este algoritmo. Teniendo en cuenta dos constantes c_1 y c_2 Mostrar tiempo constante en cada paso del algoritmo para el caso base y el caso de recursión, respectivamente, podemos formar las siguientes ecuaciones de recurrencia:

$$T(0) = c_1$$

$$T(1) = c_2 + T(1)$$

$$T(n) = c_2 + T(n - 1)$$

$$T(n) = 2c_2 + T(n - 2)$$

$$T(n) = 3c_2 + T(n - 3)$$

...

$$T(n) = kc_2 + T(n - k)$$

cuando $k = n$,

$$T(n) = c_2 + T(n - n) = c_1 + nc_2 \in \Theta(n)$$

Así hemos demostrado que este algoritmo da n pasos. Intuitivamente, sustituimos la recurrencia por valores más bajos de n hasta que vimos un patrón que se denomina método de iteración para resolver las recurrencias. Sin embargo, resolver las recurrencias puede implicar procedimientos más complicados que este simple ejemplo. Un enfoque comúnmente usado para resolver relaciones de recurrencia es adivinar una solución y probar la solución por inducción. Por ejemplo, asumamos la siguiente función de repetición:

$$T(n) = 2T(n - 1) + 1$$

con

$$T(0) = 0$$

y supongo que la solución es $T(n) = 2^n - 1$ simplemente mirando los valores de esta función para los primeros valores de n que son 0, 1, 3, 7, 15, 31 y 63 para las entradas 0, 1, 2, 3, 4 y 5. Considerando la base Caso, encontramos que se mantiene.

$$T(0) = 2^0 - 1 = 0$$

Sustitución de los rendimientos generales del caso.

$$T(n) = 2T(n - 1) = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

Por lo tanto, concluimos que nuestra suposición era correcta.

El método maestro

El método Master se utiliza para resolver relaciones de recurrencia que se pueden expresar de la siguiente manera:

$$T(n) = aT(n/b) + f(n)$$

dónde $a \geq 1$ y $b > 1$ son constantes con una función $f(n)$ de n positiva. Hay tres casos a considerar:

$$f(n) = O(n^{\log a - \varepsilon}) \quad \text{para algunos } \varepsilon > 0 \quad T(n) = \Theta(n^{\log a})$$

1. $f(n) = \Theta(n^{\log a}) \quad T(n) = \Theta(n^{\log a} \log n)$
2. $f(n) = \Omega(n^{\log a + \varepsilon})$ para algunos $\varepsilon > 0$ y $af(n/b) \leq cf(n)$ para algunos $c < 1$ y $\forall n > m_1 \quad T(n) = \Theta(n^{\log a})$

La prueba se puede encontrar en [1].

3.5 Probando la corrección de los algoritmos

La corrección es un requisito fundamental para cualquier algoritmo. Puede ser fácil en muchos casos determinar que un algoritmo funciona correctamente para cualquier entrada, pero en muchos otros casos, necesitamos demostrar formalmente que el algoritmo funciona. Podemos usar varios métodos de prueba, como el método directo, la contraposición, la contradicción; Pero el método de inducción matemática se usa más que otros.

Una declaración lógica o una proposición "si p entonces q " se puede escribir como $p \rightarrow q$ donde p se llama la premisa y q es la conclusión. Necesitamos demostrar que la conclusión se basa en la premisa al mismo tiempo que probamos una proposición. La prueba directa implica llegar a la conclusión directamente desde la premisa.

Ejemplo 3.3 Si a y b son dos enteros pares, su producto ab es par.

Prueba podemos escribir $a = 2m$ y $b = 2n$ para algunos números enteros m y n , ya que son incluso, y por lo tanto son divisibles por 2. El producto $ab = 2m \cdot 2n = 4mn = 2(2mn)$ es un número par ya que es divisible por 2. \square

Las pruebas pueden ser tan simples como en este ejemplo. En muchos casos, sin embargo, una prueba implica un razonamiento más sofisticado para llegar a la conclusión. Veamos otro ejemplo que involucra una prueba directa, pero no tan fácil de obtener. Veamos otro ejemplo de prueba directa.

Ejemplo 3.4 Dados dos enteros a y b , si $a+b$ es par, entonces $a-b$ es par.

Prueba desde $a+b$ es par, podemos escribir $a+b=2m$ para algún entero m . Luego, la sustitución por b produce:

$$a+b=2m$$

$$b = 2m - a$$

$$a - b = a - 2m + a$$

$$= 2a - 2m$$

$$= 2(a - m)$$

lo que demuestra que la diferencia es un número par y completa la prueba. \square

3.5.1 Contraposición

Un contrapositivo de una afirmación lógica, $p \rightarrow q$ es $\neg p \rightarrow \neg q$. El contrario de tal declaración es equivalente a la declaración, y por lo tanto, podemos probar lo contrario a verificar la declaración original. Este método también se denomina prueba indirecta y se puede aplicar a una variedad de problemas.

Ejemplo 3.5 para cualquier entero $a > 2$, si a es un número primo, entonces a es un número impar.

Prueba Supongamos lo contrario de la conclusión, a es par. Entonces podemos escribir $a = 2n$ para algún entero n . Sin embargo, esto implica que a es divisible por 2, y por lo tanto, no puede ser un número primo que contradiga la premisa. \square

3.5.2 Contradicción

En este método de prueba, asumimos que la premisa p es verdadera y la conclusión q no es verdadera ($p \wedge \neg q$) y tratar de encontrar una contradicción. Esta contradicción puede ser contra lo que asumimos como hipótesis o simplemente ser algo contra lo que sabemos que es cierto, como $1 = 0$. En este caso, si encontramos ($p \wedge \neg q$) es falso, significa que p es falso o $\neg q$ es falso. Como suponemos que p es cierto como premisa, $\neg q$ debe ser falso, lo que significa q es verdadero si hay una contradicción y eso completa la prueba.

Ejemplo 3.6 Probemos la afirmación: $3n + 2$ es incluso cuando n es par.

Prueba aquí $p = "3n + 2"$ es par "y $q = "n"$ incluso". Vamos a asumir $\neg q$ que es "n es impar" y por lo tanto $n = 2k + 1$ para algún entero k . Sustituyendo en los rendimientos $p = 3(2k + 1) + 2 = 6k + 5 = 2(3k + 2) + 1$. Sustituyendo $r = (3k + 2)$ que un entero, da como resultado $3n + 2 = 2r + 1$ que es impar por la definición de un entero impar. Por lo tanto, llegamos a $\neg p$. \square

3.5.3 Inducción

En la inducción, se nos da una secuencia de proposiciones en la forma $P(1), \dots, P(n)$ y realizamos dos pasos:

Paso básico : establecer $P(1)$ es verdadero.

- 1.
2. Paso de inducción : si $P(k)$ es verdadero para cualquier k dada , entonces establezca $P(k+1)$ También es cierto.

Si estos dos pasos proporcionan resultados verdaderos, podemos concluir que $P(n)$ es verdadero para cualquier n . Es uno de los métodos más utilizados para probar algoritmos secuenciales, paralelos y distribuidos.

Ejemplo 3.7 Ilustremos este método demostrando que la suma S de los primeros n números impares $1 + 3 + 5 \dots$ es $\frac{n^2}{n^2}$.

Prueba

Paso base : $P(1) = 1 = 1^2$, por lo que el paso base da una respuesta verdadera.

- 1.
2. Paso de inducción : Suponiendo $P(k) = k^2$, tenemos que demostrar que $P(k+1) = (k+1)^2$. Dado que el elemento k th de $P(n)$ se expresa como $2k-1$ Como es un número impar, se puede decir lo siguiente:

$$P(k+1) = P(k) + 2(k+1) - 1 = k^2 + 2k + 1 = (k+1)^2$$

Por lo tanto, esta proposición es verdadera para todos los enteros positivos.

□

3.5.4 fuerte inducción

En inducción, intentamos probar la proposición. $P(k+1)$ asumiendo que la afirmación $P(k)$ es verdadera. En fuerte inducción, tenemos los siguientes pasos:

Paso base : $P(1) = 1 = 1^2$, entonces el paso base es verdadero.

- 1.
2. Paso de inducción fuerte : Suponiendo que $P(1), \dots, P(k)$ son todos verdaderos, necesitamos establecer que $P(k+1)$ es verdad.

Este método de prueba es útil cuando $P(k+1)$ no depende de $P(k)$ sino de algunos valores más pequeños de k . De hecho, los dos métodos de inducción son equivalentes.

Ejemplo 3.8 Cada entero mayor que 1 es un número primo o puede escribirse como el producto de números primos.

Prueba Necesitamos considerar el caso base y el caso de inducción fuerte.

- Caso base : $P(2) = 2$ Es primordial por lo que el caso base es cierto.
- Fuerte paso de inducción : asumiendo cada entero n con $2 \leq n \leq k$ es un número primo o un producto de números primos, necesitamos probar $(k+1)$ es primo o un producto de números primos. Tenemos dos casos como sigue:

$(k+1)$ es un número primo: entonces $P(k+1)$ es verdad.

$(k+1)$ es un número compuesto y no un número primo: entonces $\exists a$ y b con $2 \leq a, b \leq k$ tal que $k+1 = a \cdot b$. Por la etapa de inducción fuerte, a y b son o bien números primos o producto de números primos. Por lo tanto, $k+1 = a \cdot b$ Es un producto de números primos. \square

3.5.5 Invariantes de bucles

Un algoritmo con un bucle comienza al inicializar algunas variables en función de algunas entradas, ejecuta un bucle y produce un resultado en función de los valores de sus variables. Un invariante de bucle es una afirmación sobre el valor de una variable después de cada iteración de un bucle particular, y el valor final de esta variable se utiliza para determinar la corrección del algoritmo.

Una condición previa es un conjunto de declaraciones que son verdaderas antes de que se ejecute el algoritmo, que se representa comúnmente como un conjunto de entradas, y una condición posterior es un conjunto de declaraciones que permanecen verdaderas después de que se ejecuta el algoritmo, que son las salidas. Utilizamos invariantes de bucle para ayudarnos a comprender por qué un algoritmo es correcto. Debemos mostrar tres cosas acerca de un bucle invariante:

Inicialización : el bucle invariante debe ser verdadero antes de la primera iteración del bucle.

- 1.
2. Mantenimiento : Si el bucle invariante es verdadero para la n iteración, debe ser cierto para $(n+1)$ th iteración.
3. Terminación : el invariante es verdadero cuando el bucle termina.

Las dos primeras propiedades de una variante de bucle afirman que la variante es verdadera antes de cada iteración de bucle, similar al método de inducción. La inicialización es como el caso base de la inducción, y el mantenimiento es similar al paso inductivo. No hay reglas definidas para elegir una variante de bucle y procedemos por sentido común en la mayoría de los casos. Consideraremos el siguiente , mientras que bucle. Nuestro objetivo es probar que este bucle funciona correctamente y termina.

```
a > 0;  
b = 0;  
while ( a != b )  
    b = b + 1;
```

Nosotros elegiremos $a \geq b$ como la variante de bucle L . Necesitamos mostrar las tres condiciones: L es verdadera antes de que comience el ciclo; si L es verdadera antes de una iteración, permanece verdadera después de la iteración y, por último, debe establecer la condición posterior. Ahora podemos verificar estas condiciones de la siguiente manera y podemos determinar que este bucle funciona correctamente y termina:

Inicialización : $a \geq 0$ y $b = 0$ antes de que comience el bucle es cierto .

- 1.
2. Mantenimiento : $(a \geq 0) \wedge (a \neq b) \rightarrow b = b + 1$
3. Terminación : $(a \geq 0) \wedge \neg(a \neq b) \rightarrow a = b$

3.6 Reducciones

Podemos querer probar que algunos problemas computacionales son difíciles de resolver. Con el fin de verificar esto, tenemos que mostrar un poco de problema X es al menos tan duro como un problema conocido Y. Una forma elegante de la prueba de esta afirmación es reducir problema Y a X. Supongamos que tenemos un problema P_1 que tiene una solución algorítmica A_1 y un problema similar P_2 . Eso no tiene ninguna solución. La similitud puede implicar que podemos tomar prestadas algunas de las soluciones encontradas para P_1 . Si podemos encontrar una reducción del problema P_2 a P_1 .

Definición 3.1 (reducción) Una reducción de un problema P_2 a P_1 transforma las entradas de P_2 a las entradas de P_1 , obtiene salidas de P_1 ejecutando algoritmo A_1 y representa estas salidas como las soluciones de P_1 . Un problema P_2 es reducible a problema P_1 . Si hay una función f que toma una entrada arbitraria para P_2 , lo transforma en una entrada para P_1 , resuelve allí y obtiene la solución a x . Esto se muestra como $P_2 \leq P_1$.

Especificar un límite superior en la transformación de la entrada es sensato, ya que este proceso en sí puede llevar mucho tiempo, por ejemplo, puede ser exponencial. Si se transfiere de alguna entrada al problema P_2 al problema P_1 . Se puede realizar en tiempo polinomial, P_2 se dice que es polinomial-tiempo reducible a P_1 y se muestra como $P_2 \leq_p P_1$. Nuestro interés se centra principalmente en las reducciones de tiempo polinómico al resolver problemas similares.

3.6.1 Problemas de gráficos difíciles

Consideremos el problema de la cubierta del vértice. Una cubierta de vértice (VCOV) de un gráfico $G = (V, E)$ es un subconjunto V' de sus vértices tal que cualquier borde $e \in E$ es incidente a al menos un vértice $v \in V'$. De manera informal, tratamos de cubrir todos los bordes de G por los vértices en este subconjunto. La forma de decisión de este problema (VCOV) pregunta: Dado un gráfico $G = (V, E)$ y un entero k , ¿ G tiene una cubierta de vértice de tamaño como máximo k ? El problema de la optimización de VCOV es encontrar la cubierta de vértice con el número mínimo de vértices entre todas las cubiertas de vértice de un gráfico.

Un conjunto independiente de una gráfica G es un subconjunto V' de sus vértices tales que ningún vértice en V' es adyacente a cualquier otro vértice en V' . La forma de decisión de este problema (IND) busca responder a la pregunta: Dado el gráfico G y un entero k , ¿ G contiene un conjunto independiente de al menos k vértices? El problema de la optimización de IND es encontrar el conjunto independiente con el número máximo de vértices entre todos los conjuntos independientes de un gráfico.

Un problema relacionado con la gráfica es encontrar el conjunto dominante (DOM) de una gráfica $G = (V, E)$ que es un subconjunto V' de sus vértices tal que cualquier $v \in V \setminus V'$ es adyacente a al menos un vértice en V' . La forma de decisión de DOM busca responder a la pregunta: Dado el gráfico G y un entero k , ¿ G contiene un conjunto dominante de k vértices como máximo? Estos subgrafos se muestran en una gráfica de muestra en la Fig. 3.2. El problema de la optimización del DOM es encontrar el conjunto dominante con el número mínimo de vértices entre todos los conjuntos dominantes de un gráfico. Las versiones mínimas o máximas de todos estos problemas son para encontrar subconjuntos de vértices que no pueden reducirse o ampliarse mediante la eliminación / adición de cualquier otro vértice. Revisaremos estos problemas con más detalle en el cap. 10.

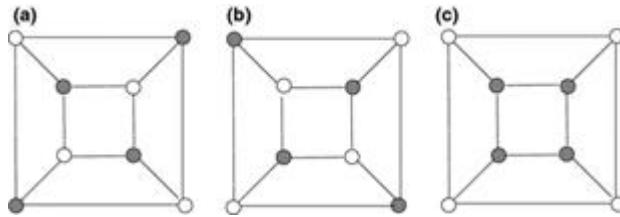


Fig. 3.2 Algunos problemas gráficos difíciles. a Una cubierta de vértice mínimo, b un conjunto independiente y c un conjunto dominante mínimo. Tenga en cuenta que a es también un conjunto máximo independiente y un conjunto dominante mínimo, b también es una cobertura de vértice mínimo y un conjunto dominante mínimo, pero c es solo un conjunto dominante mínimo para este gráfico

3.6.2 Conjunto independiente a la reducción de la cubierta del vértice

Mostraremos que un conjunto independiente de una gráfica se puede reducir a una cubierta de vértice considerando primero el teorema a continuación.

Teorema 3.1 Dada una gráfica $G = (V, E)$, $S \subset V$ es un conjunto independiente de G si y solo si $V - S$ Es una cubierta de vértice.

Prueba Consideremos una ventaja $(u, v) \in E$. Si $S \subset V$ es un conjunto independiente en G , ya sea $u \in S$ o $v \in S$ pero no ambos. Esto significa que al menos uno de los puntos finales de (u, v) está en $V - S$ de ahí $V - S$ Es una cubierta de vértice. Necesitamos probar el teorema en la otra dirección; Si $V - S$ Es una cubierta de vértice, consideremos dos vértices $u \in S$ y $v \in S$. Entonces, $(u, v) \notin E$ ya que si existiera tal borde, no estaría cubierto en $V - S$, y por lo tanto, $V - S$ No sería una cubierta de vértice.

La Figura 3.3 muestra la equivalencia de estos dos problemas en un gráfico de muestra. Dados los cinco vértices en (a) con $k=5$, podemos ver que forman un conjunto independiente. Ahora transformamos esta entrada en una entrada del problema de cobertura de vértice en tiempo polinomial, que son los vértices blancos en (a). Verificamos si estos forman una cubierta de vértice en (b) nuevamente en tiempo polinomial y concluimos que lo hacen. Nuestro algoritmo de prueba simplemente marca los bordes incidentes en vértices negros en el tiempo $O(k)$ y verifica si alguno de los bordes se deja sin marcar al final. Todos los bordes están cubiertos por estos cuatro vértices en este caso. Por lo tanto, podemos deducir que los cinco vértices negros en (a) son, de hecho, una solución al problema de decisión de conjunto independiente para este gráfico. Hemos mostrado un ejemplo de $\text{IND} \leq_p \text{VCOV}$.

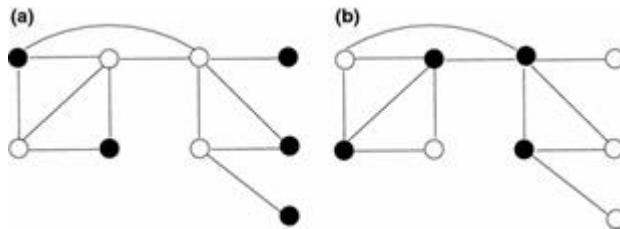


Fig. 3.3 Una gráfica de muestra con un conjunto independiente en a y una cubierta de vértice en b que se muestra con círculos negros en ambos casos. El conjunto independiente en b está formado por los vértices blancos en una

3.7 NP- Integridad

Los problemas a los que nos enfrentamos se pueden clasificar según el tiempo que se tarda en resolverlos. Una función polinomial. $O(n^k)$, con n como la variable y k como la constante,

está delimitado por n^k como vimos. Las funciones exponenciales se refieren a funciones tales como $O(2^n)$ o $O(n^n)$ que crecen muy rápidamente con el aumento del tamaño de entrada n . Un algoritmo de tiempo polinomial tiene un tiempo de ejecución limitado por una función polinomial de su entrada, y un algoritmo exponencial es el que no tiene un rendimiento de tiempo limitado por una función polinomial de n . Un problema manejable es solucionable mediante un algoritmo de tiempo polinomial, y un problema intratable no puede resolverse mediante un algoritmo de tiempo polinomial. La búsqueda de un valor clave en una lista se puede realizar mediante un algoritmo de tiempo polinomial, ya que necesitamos verificar cada entrada de una lista de tamaño n en n tiempo y enumerar todas las permutaciones de n . Los números son un ejemplo de un algoritmo exponencial. De hecho, la tercera clase de problemas no tiene algoritmos polinómicos conocidos, pero tampoco se ha demostrado que sean intratables. Cuando se nos presenta un problema intratable, podemos hacer una de las siguientes acciones:

- Intente resolver una versión más simple o restringida de un problema que se puede lograr en tiempo polinomial.
- Implementar un algoritmo probabilístico de tiempo polinomial que proporcione la solución correcta solo con una probabilidad muy alta. Esto significa que puede fallar en raras ocasiones.
- Use un algoritmo de aproximación de tiempo polinomial con una relación de aproximación probada. En este caso, tenemos una solución subóptima al problema que puede ser aceptable en muchas aplicaciones.
- Cuando todo falla, podemos usar algunas heurísticas que son reglas de sentido común para diseñar algoritmos de tiempo polinómico. Necesitamos demostrar experimentalmente que el algoritmo heurístico funciona bien con un amplio espectro de combinaciones de entrada. Muchos problemas en los gráficos son intratables y este método se emplea con frecuencia especialmente en gráficos grandes, como veremos.

3.7.1 Clases de complejidad

Los problemas manejables tienen algoritmos de tiempo polinomial para resolverlos. Los problemas intratables, por otro lado, se pueden dividir en dos subclases; los probados para no tener algoritmos de tiempo polinomial y otros que tienen algoritmos de solución de tiempo exponencial. En este punto, tendremos que clasificar los problemas según los resultados esperados de ellos como problemas de optimización o problemas de decisión. En un problema de optimización, intentamos maximizar o minimizar una función objetivo en particular y un problema de decisión devuelve un sí o un no como respuesta a una entrada determinada. Consideremos el problema de la IND tanto como una optimización como un problema de decisión. El problema de optimización pide encontrar el conjunto independiente de un gráfico. $G = (V, E)$ Con el pedido más grande. El problema de decisión que vimos busca una respuesta a la pregunta: dado un entero $k \leq |V|$, ¿ G tiene un conjunto independiente con al menos k vértices? Tratar con los problemas de decisión es ventajoso no solo porque estos problemas son en general más fáciles que sus versiones de optimización, sino que también pueden proporcionar una transferencia al problema de optimización. En el problema de decisión de IND, podemos probar todos

los valores k posibles y encontrar el más grande que proporcione un conjunto independiente para encontrar una solución al problema de optimización de IND.

Clase de complejidad p

La primera clase de complejidad que consideraremos es P, que contiene problemas que pueden resolverse en tiempo polinomial.

Definición 3.2 (clase P) La clase de complejidad P se refiere a problemas de decisión que pueden resolverse en tiempo polinomial.

Para que un problema A esté en P, tiene que haber un algoritmo con el peor tiempo de ejecución $O(n^k)$ para un tamaño de entrada n y una constante k que resuelve A . Por ejemplo, encontrar el valor más grande de una matriz de enteros se puede realizar en tiempo O (n) para una matriz de tamaño n , y por lo tanto es un problema en P.

Clase de complejidad NP

En los casos en que no conocemos un algoritmo de tiempo polinomial para resolver un problema de decisión dado, se puede realizar una búsqueda de un algoritmo que resuelva una instancia de entrada de un problema en el tiempo polinomial. La entrada específica se llama certificado y el algoritmo de tiempo polinomial que comprueba si el certificado es aceptable, lo que significa que es una solución al problema de decisión, se denomina certificador . Ahora podemos definir una nueva clase de complejidad de la siguiente manera.

Definición 3.3 (clase NP) La clase de complejidad Polinomio no determinista (NP) es el conjunto de problemas de decisión que se pueden verificar mediante un algoritmo polinomial.

La clase NP incluye P (P ⊂ NP) clase ya que todos los problemas en P tienen certificadores en tiempo polinomial pero si P ⊂ NP no se ha determinado y sigue siendo un gran desafío en Ciencias de la Computación. Muchos problemas son difíciles de resolver, pero una instancia de entrada se puede verificar en tiempo polinomial, ya sea que se obtenga una solución o no. Por ejemplo, dada una gráfica. $G = (V, E)$ y un certificado $S \subseteq V$, Podemos comprobar en tiempo polinómico si S es un conjunto independiente de G . El programa del certificador se muestra en el algoritmo 3.4, donde simplemente verificamos si hay dos vértices en S adyacentes. Si se encuentran estos dos vértices, la respuesta es NO y la entrada se rechaza. El algoritmo ejecuta dos bucles anidados en $O(n^2)$ tiempo, y por lo tanto, tenemos un certificador de tiempo polinomial que muestra IND ∈ NOTARIO PÚBLICO.

Algorithm 3.4 IS_Certifier

```

1: Input :  $G = (V, E)$ ,  $S \subseteq V$ 
2: Output : yes or no
3: for all  $u \in S$  do
4:   for all  $v \in S$  do
5:     if  $(u, v) \in E$  then
6:       return No
7:     end if
8:   end for
9: end for
10: return Yes

```

Problemas NP-duros

Definición 3.4 (clase NP-Duro) Un problema de decisión P_i es NP-difícil si todos los problemas en NP son polinomiales reducibles a P_i . En otras palabras, si podemos

resolverlo.^{Pi} En el tiempo polinomial, podemos resolver todos los problemas de NP en el tiempo polinomial.

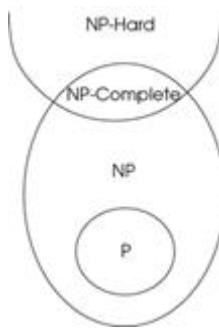


Fig. 3.4 Relación entre clases de complejidad.

Un problema difícil de NP^{Pi} No necesita estar en NP. Un problema difícil de NP^{Pi} medio^{Pi} Es tan difícil como cualquier problema en NP.

Problemas NP-completos

Un problema^{Pi} es NP-completo si es NP-duro y también es miembro de la clase NP.

Definición 3.5 (clase NP-Completa) Un problema de decisión^{Pi} es NP-Completo si está en NP y NP-duro.

La figura [3.4](#) muestra la relación entre las clases de complejidad. La clase P está contenida en la clase NP, ya que cada problema en P tiene un certificador, y los problemas NP-completos se encuentran en la intersección de los problemas NP y los problemas NP-difíciles.

Para demostrar que un problema^{Pi} es NP-Completo, tenemos que hacer lo siguiente:
Probar^{Pi} Es un problema de decisión.

- 1.
2. Show^{Pi} ∈ NOTARIO PÚBLICO.
3. Show^{Pi} Es tan difícil como cualquier problema que sea NP-duro.

3.7.2 El primer problema NP-difícil: la satisfacción

El problema de satisfacción (SAT) indica que, dada una fórmula booleana, ¿hay una manera de asignar valores de verdad a las variables en la fórmula de manera que la fórmula evalúe el valor verdadero? Consideremos un conjunto de variables lógicas.

x_1, x_2, \dots, x_n Cada uno de los cuales puede ser verdadero o falso. Una cláusula está formada por la disyunción de variables lógicas como $(x_1 \vee \overline{x}_2 \vee x_3)$. Una fórmula CNF es una conjunción de las cláusulas como $C_1 \wedge C_2, \dots \wedge C_k$ como a continuación:

$$(x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$$

(3.1)

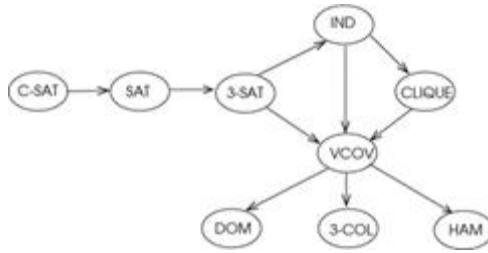


Fig. 3.5 La relación de reducción entre varios problemas NP-difíciles

La fórmula CNF se satisface si cada cláusula en ella da un valor verdadero . El problema de satisfacción busca una asignación a variables x_1, x_2, \dots, x_n de tal manera que se satisfaga la fórmula CNF. El problema 3-SAT requiere que cada cláusula sea de longitud 3 sobre las variables x_1, x_2, \dots, x_n . El problema SAT no tiene un algoritmo de tiempo polinomial conocido, pero tampoco podemos concluir que sea intratable. Podemos probar todas las combinaciones de la entrada en 2^n . Es hora de encontrar la solución. Sin embargo, podemos tener una entrada distinta y verificar si esta entrada es aceptada por el circuito SAT y, por lo tanto, concluir que SAT está en NP. Se demostró que este problema es NP-duro y, por lo tanto, Cook lo completó en 1970 [3]. Por lo tanto, podemos usar el problema 3-SAT como base para probar que otros problemas están completos o no en NP. Las relaciones entre varios problemas se representan en la figura 3.5 .

Vamos a mostrar cómo reducir el problema 3-SAT al problema IND. En el primero, sabemos que tenemos que establecer al menos un término en cada cláusula para que sea verdadero y no podemos establecer ambos x_i y \bar{x}_i para ser verdad al mismo tiempo. Primero dibujamos triángulos para cada cláusula de 3-SAT con cada vértice que representa el término dentro de la cláusula. Un verdadero literal de cada cláusula es suficiente para obtener un valor verdadero para la fórmula 3-SAT. Añadimos líneas entre un término y su inverso, ya que no queremos incluir ambos en la solución. La gráfica dibujada de esta manera para la ec. 3.1 se muestra en la figura 3.6 .

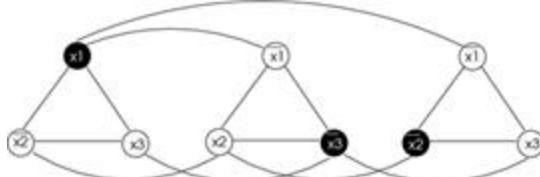


Fig. 3.6 La gráfica para la ecuación 3-SAT de la ecuación. 3.1 . Los vértices negros x_1 , \bar{x}_3 y \bar{x}_2 representan el conjunto independiente de este gráfico, que también es la solución al 3-SAT de la ecuación. 3.1 con $x_1 = 1$, $x_2 = 0$ y $x_3 = 0$ valores

Ahora reclamamos lo siguiente.

El teorema 3.2 de 3-SAT fórmula F con k cláusulas es satisfactorio si y solo si la gráfica formada de esta manera tiene un conjunto independiente de tamaño k .

Prueba Deje que la gráfica formada de esta manera sea $G = (V, E)$. Si la fórmula F es satisfactoria, debemos tener al menos un literal verdadero de cada cláusula. Formamos el conjunto de vértices $V' \subset V$ seleccionando un vértice de cada triángulo, y también al no seleccionar una variable x y su complemento \bar{x} al mismo tiempo, ya que una variable y su complemento no pueden ser verdaderos al mismo tiempo. V' es un conjunto independiente ya que no hay bordes entre los vértices seleccionados. Para probar la afirmación en la dirección inversa, consideraremos que G tiene un conjunto independiente de tamaño k . El conjunto V' no puede tener dos vértices del mismo grupo y no tendrá una variable y su

complemento al mismo tiempo ya que es un conjunto independiente. Además, cuando establecemos valores verdaderos para todas las variables en V , habremos cumplido con la fórmula F del SAT. La transformación del problema 3-SAT al problema IND se puede realizar en tiempo polinomial; por lo tanto, deducimos que estos dos problemas son equivalentes, y encontrar una solución para uno significa que la solución también se descubre. \square

3.8 Haciendo frente a NP-Completeness

Muchos de los problemas de optimización de gráficos son NP-hard sin soluciones conocidas en tiempo polinomial. Sin embargo, los métodos para obtener soluciones la mayor parte del tiempo dentro de un límite de probabilidad específico (algoritmos aleatorios), o resultados que están cerca de la solución exacta dentro de un margen específico al resultado exacto (algoritmos de aproximación); o los métodos que eliminaremos en algunos de los resultados intermedios no deseados para lograr una mejora en el rendimiento (backtracking y branch and bound) son los temas que revisaremos en esta sección.

3.8.1 Algoritmos aleatorizados

Los algoritmos aleatorios se utilizan con frecuencia para algunos de los problemas gráficos difíciles, ya que son simples y proporcionan soluciones eficientes. Los números generados aleatoriamente o las elecciones aleatorias se utilizan normalmente en estos algoritmos para decidir los cursos de cómputo. La salida de un algoritmo aleatorio y su tiempo de ejecución varía para diferentes entradas e incluso para la misma entrada. Dos clases de algoritmos aleatorios son los algoritmos de Las Vegas y Monte Carlo. El primero siempre devuelve una respuesta correcta, pero el tiempo de ejecución de tales algoritmos depende de las elecciones aleatorias realizadas. El algoritmo se ejecuta una cantidad de tiempo constante, pero la respuesta puede o no ser correcta en los algoritmos de Monte Carlo.

El costo promedio del algoritmo sobre todas las elecciones aleatorias nos da sus límites esperados y un algoritmo aleatorio se especifica comúnmente en términos de su tiempo de ejecución esperado para todas las entradas. Por otro lado, cuando decimos que un algoritmo se ejecuta en tiempo $O(x)$ con alta probabilidad, significa que el tiempo de ejecución de este algoritmo no estará por encima del valor de x con alta probabilidad. Los algoritmos aleatorios se usan comúnmente en dos casos: cuando se elige una configuración aleatoria inicial y para decidir una solución local cuando hay varias opciones. La elección aleatoria se puede repetir con diferentes semillas y luego se devuelve la mejor solución [2].

Algoritmo de corte mínimo de Karger

Nos vamos a describir cómo la aleatorización ayuda a encontrar el recorte mínimo (mincut en adelante) de un gráfico. Dado un grafo $G = (V, E)$, encontrar un error de G es dividir los vértices de la gráfica en dos conjuntos separados V_1 y V_2 tal que el número de bordes entre V_1 y V_2 es mínimo. Hay una solución a este problema utilizando el flujo máximo, como veremos en el capítulo 8, aquí describiremos un algoritmo aleatorio debido a Karger [4].

Este sencillo algoritmo selecciona un borde al azar, crea un supervertex desde los puntos finales del borde seleccionado utilizando la contracción y continúa hasta que

quedan exactamente dos supervertices como se muestra en el algoritmo 3.5. Los vértices en cada supervertex final son los vértices de las particiones.

Algorithm 3.5 Kargel_mincut

```

1: Input :  $G(V, E)$ 
2: Output mincut  $V_1$  and  $V_2$  of  $G$ 
3:
4:  $G' = (V', E') \leftarrow G(V, E)$ 
5: repeat
6:   select an edge  $(u, v) \in E'$  at random
7:   contract vertices  $u$  and  $v$  into a super vertex  $uv$ 
8: until there are only two super vertices  $u_1$  and  $u_2$  left
9:  $V_1 \leftarrow$  all of the vertices in  $u_1$ 
10:  $V_2 \leftarrow$  all of the vertices in  $u_2$ 
```

La contracción de dos vértices u y v se realiza de la siguiente manera:

- Eliminar todos los bordes entre u y v .
- Reemplaza u y v con un supervertex uv .
- Conecte todos los bordes incidentes a u y v a supervertex uv .

Veamos cómo funciona este algoritmo en el gráfico simple de la figura 3.7. Los bordes seleccionados al azar se muestran dentro de las regiones discontinuas y el corte final consta de tres bordes entre $V_1 = \{b, h, a\}$ y $V_2 = \{c, f, d, e, g\}$ como se muestra en (h). Este no es el mincut, sin embargo, el corte mínimo consiste en los bordes (b, c) y (g, f) como se muestra en (i).

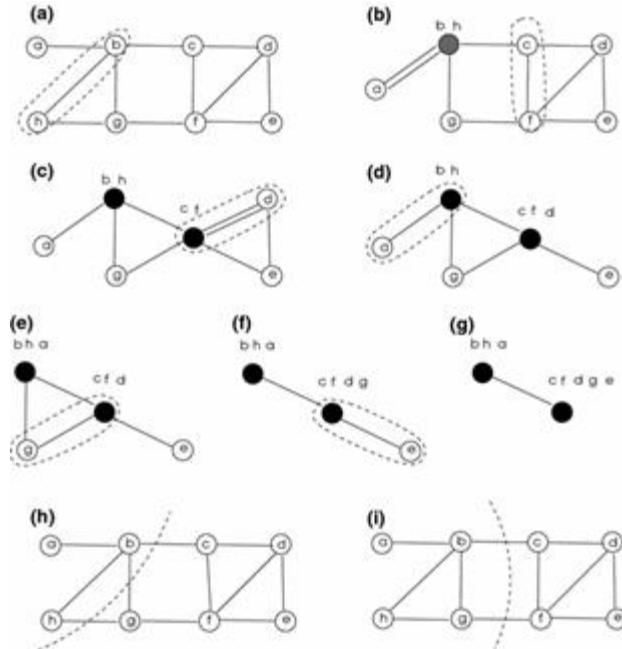


Fig. 3.7 Ejecución del algoritmo de Kargel en un gráfico simple

El algoritmo de Karger encontrará el corte mínimo correcto si nunca selecciona un borde que pertenezca al corte mínimo. En nuestro ejemplo, seleccionamos el borde (g, f) que pertenece al mincut en la Fig. 3.7 e deliberadamente para dar como resultado un corte que no es mínimo. Por otro lado, los bordes mincut tienen la menor probabilidad de ser seleccionados por este algoritmo, ya que tienen menos bordes que todos los bordes que no pertenecen al mincut. Antes de su análisis, permítanos indicar algunas observaciones acerca del tamaño de una gráfica.

Observación 1 El tamaño de un fragmento de una gráfica G es a lo sumo el grado mínimo $\delta(G)$ de g .

Esto es válido desde el mincut no es mayor que cualquier corte de G . Por lo tanto, $\delta(G)$ establece un límite superior en el tamaño de la mincut. Ya que no podemos determinar $\delta(G)$ fácilmente, verifiquemos si existe un límite superior en términos de tamaño y orden de G.

Teorema 3.3 El tamaño de la mincut es a lo sumo $2 m / n$

Prueba Supongamos que el tamaño de la mincut es k . Entonces, cada vértice de G debe tener al menos un grado de k . Por lo tanto, por el teorema de Euler (lema de apretón de manos),

$$m = \frac{\sum_{v \in V} \deg(v)}{2} \geq \frac{\sum_{v \in V} k}{2} = \frac{nk}{2}$$

(3.2)

lo que significa $k \leq 2m/n$. \square

Corolario 3.1 Dado un gráfico G con un mincut C , la probabilidad de seleccionar un borde $(u, v) \in C$, $P(E_1)$ es a lo más $2 / n$.

Prueba Hay m bordes y a lo sumo $2 m / n$ están en la mincut por Teorema 3.3 , por lo que $P(E_1) = m/(2m/n) = 2/n$. \square

Observación 2 Dado un gráfico G con un mincut C , el algoritmo no debe seleccionar ningún borde $(u, v) \in C$.

Por lo tanto, la probabilidad de no seleccionar el primer borde de la mincut es

$$P(\bar{E}_1) = 1 - P(E_1) \geq 1 - \frac{2}{n} \geq \frac{n-2}{n}$$

(3.3)

Escojamos un corte mínimo C con tamaño k y encontremos la probabilidad de que un borde $(u, v) \in C$ no está contraído por el algoritmo, que nos dará la probabilidad de que el algoritmo encuentre el resultado correcto. Primero evaluaremos la probabilidad de seleccionar un borde. $(u, v) \in C$ en la primera ronda, $P(E_1)$ que es k / m para un mincut con tamaño k . Por lo tanto,

$$P(E_1) = \frac{k}{m} \leq \frac{k}{nk/2} = \frac{2}{n}$$

(3.4)

Dejar $P(E_C)$ Ser la probabilidad de que el corte final sea mínimo. Esta probabilidad es el producto de las probabilidades de la probabilidad de que el primer borde seleccionado no

esté en mincut, la probabilidad de que el segundo borde seleccionado no esté en mincut, etc., hasta que se formen los dos últimos supervértices. Una contracción de un borde da como resultado un vértice menos en el nuevo gráfico. Por lo tanto,

$$P(e_C) \geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \left(1 - \frac{2}{n-3}\right) \cdots \left(1 - \frac{2}{3}\right)$$

(3.5)

ya que los nominadores y los denominadores se cancelan en cada dos términos, excepto los dos primeros denominadores,

$$= \left(\frac{n-2}{n}\right) \left(\frac{n-3}{n-1}\right) \left(\frac{n-4}{n-2}\right) \cdots \left(\frac{1}{3}\right)$$

$$= \frac{2}{n(n-1)}$$

Por lo tanto, la probabilidad de que el algoritmo devuelva el mincut C es al menos $\frac{2}{n(n-1)}$. Por lo tanto podemos concluir que este algoritmo tiene éxito con probabilidad $p \geq \frac{2}{n^2}$ y ejecutandolo $O(n^2 \log n)$. El tiempo proporciona un corte mínimo con alta probabilidad. Usando la matriz de adyacencia del gráfico, podemos ejecutar cada iteración en $O(n^3)$ tiempo, y el tiempo total es $O(n^4 \log n)$.

3.8.2 Algoritmos de aproximación

Un enfoque común en la búsqueda de una solución a los problemas de NP-hard graph es relajar los requisitos e inspeccionar los algoritmos de aproximación que proporcionan soluciones subóptimas.

Definición 3.6 (relación de aproximación) Sea A ser un algoritmo de aproximación para el problema P, I sea un instante del problema P, y OPT (I) el valor del coste óptimo de solución para I. La relación de aproximación del algoritmo A se define como

$$\alpha_A = \max_I \frac{A(I)}{OPT(I)}$$

(3.6)

Ejemplo 3.9 Ya habíamos investigado el problema de cobertura de vértice en la Secta. 3.6. Encontrar la cobertura mínima de vértices que tenga el menor número de vértices entre todas las cubiertas de vértices de una gráfica es NP-duro. Como nuestro objetivo es cubrir todos los bordes del gráfico con un subconjunto de vértices, podemos diseñar un algoritmo que seleccione cada borde en un orden aleatorio y, como no podemos

determinar qué vértice se usará para cubrir el borde, incluimos ambos extremos de El borde en la cubierta como se muestra en el algoritmo 3.6. Para cada borde seleccionado (u, v), debemos eliminar los bordes que inciden en u o v del gráfico, ya que estos bordes están cubiertos.

Algorithm 3.6 Approx_Verex_Cover

```

1: Input  $G(V, E)$ 
2:  $E' \leftarrow E, V' \leftarrow \emptyset$ 
3: while  $E' \neq \emptyset$  do
4:   select randomly an edge  $(u, v) \in E'$ 
5:    $V' \leftarrow V' \cup \{u, v\}$ 
6:    $E' \leftarrow E' \setminus \{ \text{all edges incident to either } u \text{ or } v \}$ 
7: end while

```

Las iteraciones de este algoritmo en un gráfico de muestra se muestran en la Fig. 3.8 que proporciona una cobertura de vértice con una aproximación de 2.

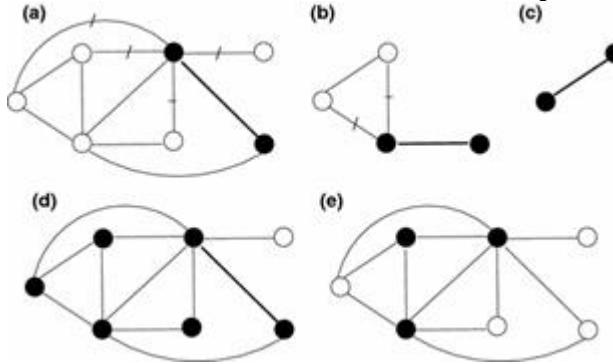


Fig. 3.8 Una posible iteración del algoritmo 3.6 en una gráfica de muestra, que muestra el borde seleccionado en negrita y los vértices incluidos en los puntos finales de este borde como negro en cada paso, desde a - c . La cubierta final del vértice tiene seis vértices como se muestra en d, y la cubierta mínima del vértice para este gráfico tiene tres vértices como se muestra en e, lo que resulta en la peor relación de aproximación de 2

El teorema 3.4 El algoritmo 3.6 proporciona una cobertura de vértice en tiempo $O(m)$ y el tamaño de MVC es $2^{\lfloor \text{MinVC} \rfloor}$.

Prueba Dado que el algoritmo continúa hasta que no quedan más bordes, se cubre cada borde, por lo tanto, la salida de Seql_MVC Es un MVC, tomando $O(m)$ tiempo. El conjunto de bordes escogidos por este algoritmo es una coincidencia, ya que los bordes elegidos están separados y es máximo ya que no es posible agregar otro borde. Como se cubren dos vértices para cada borde coincidente, la relación de aproximación para este algoritmo es 2.

□

3.8.3 Retrocesos

En muchos casos de diseño de algoritmos, nos enfrentamos a un espacio de búsqueda que crece exponencialmente con el tamaño de la entrada. El enfoque de búsqueda bruta o exhaustiva busca todas las opciones disponibles. Retroceder es una forma inteligente de buscar las opciones disponibles mientras busca una solución. En este método, nos fijamos en una solución parcial y si podemos seguir adelante, lo hacemos. De lo contrario, retrocedemos al estado anterior y procedemos desde allí, ya que proceder del estado actual viola los requisitos. De esta manera, guardamos algunas de las operaciones necesarias a partir del estado actual en adelante. Las opciones que se pueden hacer se colocan en un árbol de búsqueda de estadodonde los nodos, excepto las hojas, corresponden a soluciones parciales y los bordes se utilizan para expandir las soluciones

parciales. No se busca un subárbol que no lleve a una solución y retrocedemos a la matriz de dicho nodo. El retroceso se puede implementar convenientemente por medio de la recursión, ya que necesitamos volver a la opción anterior si encontramos que la opción actual no conduce a una solución.

Veamos cómo funciona este método usando un ejemplo. En el problema de suma de subconjuntos , se nos da un conjunto de S de n enteros distintos y se nos pide que encuentren posiblemente más de un subconjunto de la suma de S de los cuales es igual a un entero M dado . Por ejemplo, si $S = \{1, 3, 5, 7, 8\}$ y $M = 11$, entonces $S_1 = \{1, 3, 7\}$ y $S_2 = \{3, 8\}$ son las soluciones Tenemos 2^n posibles subconjuntos y un árbol binario que representa el árbol de espacio de estado tendrá 2^n Hojas con una o más hojas que proporcionan las soluciones si existen.

Dado $S = \{2, 3, 6, 7\}$ y $M = 9$, se puede formar un árbol de espacio de estado como se muestra en la Fig. 3.9 . Los nodos del árbol muestran la suma acumulada hasta ese punto desde la raíz hacia abajo del árbol y comenzamos con 0 suma. Consideramos cada elemento del conjunto S en orden creciente y con una longitud i desde la raíz, el elemento considerado es el elemento i th de S. En cada nodo, tenemos la rama izquierda que muestra la decisión si incluimos el elemento y la rama derecha pasa al subárbol cuando no incluimos ese elemento en la búsqueda. Los nodos que se muestran en círculos discontinuos son los puntos en la búsqueda en los que no necesitamos continuar ya que el requisito no se puede cumplir si lo hacemos. Por ejemplo, cuando se incluyen los dos primeros elementos del conjunto, tenemos 5 como la suma acumulada y añadiendo el tercer elemento, 6 a dar 11 que es más que M . Por lo tanto, no tomamos el subárbol arraigado en la rama izquierda del nodo 5. De forma similar, la selección de 3 y 6 nos da la solución y la informamos, pero todavía retrocedemos ya que todas las soluciones son necesarias. Si solo se requiriera una solución, nos hubiéramos detenido allí.

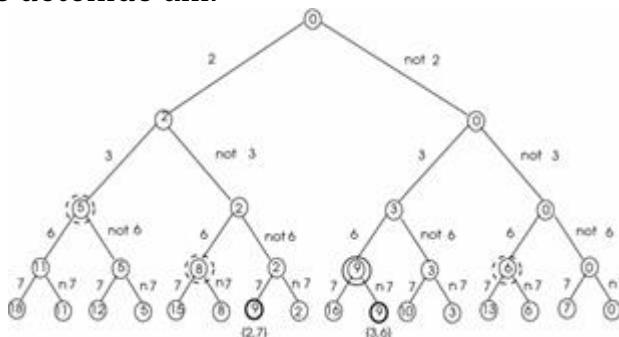


Fig. 3.9 Árbol de espacio de estado para el ejemplo de suma de subconjunto

3.8.4 Rama y Límite

El método de ramificación y límite es similar al enfoque de retroceso utilizado para los problemas de decisión, que se modifica para los problemas de optimización. El objetivo de resolver un problema de optimización es maximizar o minimizar una función objetivo y el resultado encontrado es la solución óptima al problema. Los algoritmos de ramificación y límite emplean los árboles de espacio de estado como en el retroceso con la adición del registro de la mejor solución mejor encontrada hasta ese momento en la ejecución. Además, a medida que avanza la ejecución, necesitamos el límite del mejor valor $netxt_i$ que se puede obtener para cada nodo i del árbol si continuamos procesando desde ese nodo. De esta manera, podemos comparar estos valores y si $netxt_i$ no es mejor que lo mejor , no hay necesidad de procesar el subárbol enraizado en el nodo i .

Ilustraremos la idea general de este método por el problema del vendedor ambulante (PST) en el que un vendedor comienza su viaje desde una ciudad, visita cada ciudad y regresa a la ciudad original utilizando una distancia total mínima. Esto, de hecho, es el problema del ciclo hamiltoniano con el peso de los bordes. Dejar $G = (V, E)$ Ser el gráfico no dirigido que representa las ciudades y carreteras entre ellas. Para el ejemplo de gráfico de la figura 3.10, podemos ver que hay seis posibles ciclos hamiltonianos ponderados y solo dos proporcionan la ruta óptima. Las rutas en ambos son iguales pero el orden de las visitas se invierte. Tenga en cuenta que a partir de cualquier vértice proporcionaría las mismas rutas. De hecho, hay $(n - 1)!$ Rutas posibles en una gráfica totalmente conectada con n vértices.

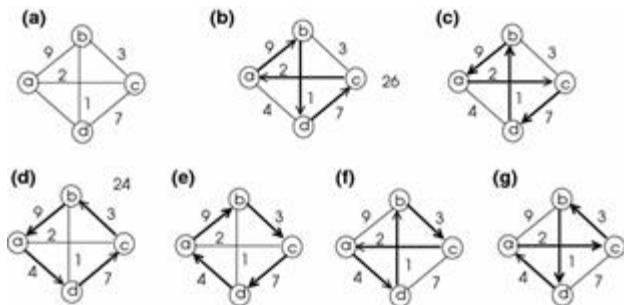


Fig. 3.10 Ciclos hamiltonianos de un gráfico simple ponderado

Un enfoque de fuerza bruta comenzaría desde un vértice a , por ejemplo, y buscaría todos los ciclos hamiltonianos posibles usando el árbol de espacio de estado y luego registraría la mejor ruta encontrada. Necesitamos definir un valor de límite inferior (lb) para el algoritmo de ramificación y límite. El valor de lb se calcula como la suma total de los dos bordes de peso mínimo de cada vértice dividido por dos para obtener el valor promedio. Para el gráfico representado en la figura 3.10, este valor es

$$((2 + 4) + (3 + 1) + (3 + 2) + (4 + 1))/2 = 10$$

Ahora podemos comenzar a construir el árbol de espacio de estado y cada vez que consideremos agregar un borde a la ruta existente, modificaremos los valores en la función de límite inferior según la selección de ese borde y calcularemos un nuevo límite inferior. Luego, seleccionaremos el borde con el valor de límite inferior mínimo entre todos los bordes posibles. El árbol de espacio de estado de la rama y el algoritmo unido para TSP en el gráfico de la figura 3.10 se muestra en la figura 3.11. Los ciclos hamiltonianos óptimos son a, c, b, d, a y a, d, b, c, a . Estos caminos corresponden a los caminos (f) y (g) de la figura 3.10.

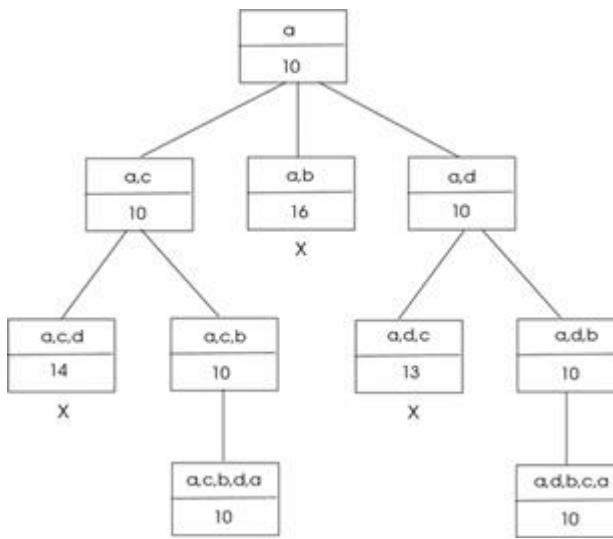


Fig. 3.11 El árbol de espacio de estado para la gráfica de la Fig. 3.10 . Cada nodo de árbol tiene la ruta y el valor de límite inferior utilizando esta ruta

La clave para el funcionamiento de cualquier rama y algoritmo de límite es la especificación del límite inferior. Cuando el espacio de búsqueda es grande, necesitamos que este parámetro se calcule fácilmente en cada paso pero que sea lo suficientemente selectivo como para podar los nodos no deseados del árbol de espacio de estado temprano. Para el ejemplo TSA, otro límite inferior es la suma de las entradas mínimas en la matriz de adyacencia A de la gráfica G . Este es un límite inferior sólido ya que estamos considerando el borde de peso más ligero de cada vértice y sabemos que no podemos hacerlo mejor que esto. Para nuestro ejemplo de la figura 3.10 , el límite inferior calculado de esta manera es 9. Calcular el límite inferior en cada paso implicaría entonces eliminar la fila a y la columna b de Acuando se considera el borde (a , b) y luego se incluyen los bordes más claros de cada uno de los vértices restantes que no están conectados a a o b para calcular el límite inferior para incluir el borde (a , b) en la ruta.

3.9 Principales métodos de diseño

Dado un problema que debe resolver un algoritmo, podemos enfocar el diseño utilizando varios métodos. Un enfoque básico es el método de fuerza bruta en el que básicamente evaluamos todas las posibilidades. Es simple y fácil de aplicar, pero por lo general no tiene un rendimiento favorable. Otras técnicas comúnmente empleadas son el método codicioso, el método de dividir y vencer, y la programación dinámica.

3.9.1 Algoritmos codiciosos

El codicioso metodobusca soluciones que sean óptimamente locales basadas en el estado actual. Elige la mejor alternativa disponible utilizando la información actualmente disponible. Un ejemplo de la vida real es el cambio provisto por un cajero en un supermercado. Comúnmente, el cajero seleccionará la moneda más grande para obtener el menor número de monedas en el siguiente paso, lo cual es óptimo en algunas combinaciones de monedas, como en los EE. UU. En muchos casos, seguir la mejor solución local en cada paso no proporcionará un óptimo general solución. Sin embargo, en ciertos problemas, el método codicioso proporciona soluciones óptimas. Encontrar los caminos más cortos entre los nodos de un gráfico ponderado y construir un árbol de expansión mínimo de un gráfico son ejemplos de métodos codiciosos que se pueden usar de manera

eficiente para encontrar las soluciones. Los algoritmos codiciosos también pueden usarse para encontrar soluciones aproximadas a algunos problemas. Describiremos el algoritmo de Kruskal para encontrar el árbol de expansión mínimo (MST) como un ejemplo de algoritmo de gráfico codicioso.

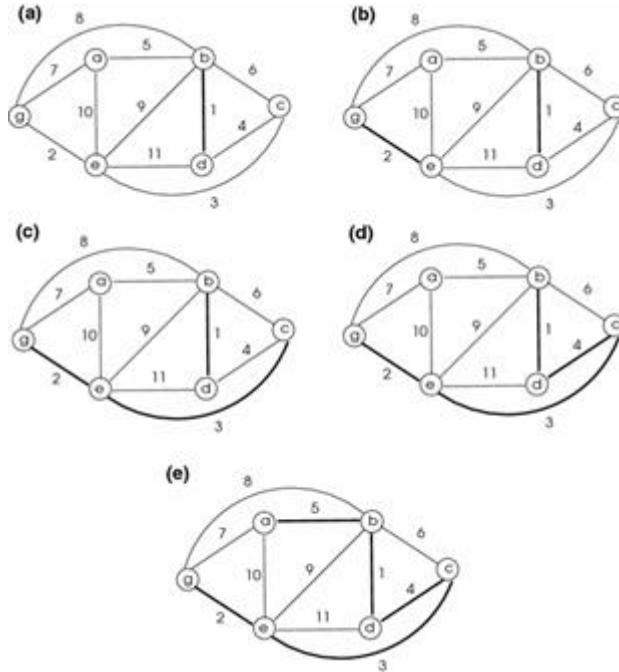


Fig. 3.12 Ejecución del algoritmo MST de Kruskal para un gráfico de muestra

3.9.1.1 Algoritmo de Kruskal

El algoritmo MST de Kruskal ordena los bordes del gráfico ponderado $G(V, E, w)$ en pesos no decrecientes. Luego, a partir del borde de peso más ligero, los bordes se incluyen en el MST parcial. T^* siempre que no formen ciclos con los bordes ya contenidos en T^* . Este proceso continúa hasta que se procesan todos los bordes de la cola. Este algoritmo consta de los siguientes pasos:

Entrada : Un gráfico no dirigido ponderado $G = (V, E, w)$

- 1.
2. Salida : MST T de G
3. Ordene los bordes de G en orden no decreciente y colóquelos en Q
4. $T \leftarrow \emptyset$
5. mientras que Q no está vacío
6. Elija el primer elemento (u, v) de Q que no forme un ciclo con ningún borde de T
7. $T \leftarrow T \cup \{(u, v)\}$

La Figura 3.12 muestra las iteraciones del algoritmo de Kruskal en una gráfica.

La complejidad de este algoritmo es $O(n^2)$ a medida que se ejecuta el bucle while para todos los vértices y la búsqueda del borde de peso mínimo para cada vértice tomará

tiempo $O(n)$. Esta complejidad puede reducirse a $\Theta(m \log n)$ tiempo mediante el uso de montones binarios y listas de adyacencia. Revisaremos este algoritmo con más detalle cuando investiguemos los algoritmos MST para los gráficos en el Capítulo 7.

Análisis

Los pesos de los bordes de la gráfica G se pueden clasificar en $\Theta(m \log m)$ hora. Podemos utilizar la estructura de datos union-find descrita en el cap. 7 que resulta en $\log n$. El tiempo para buscar y probar m aristas se puede hacer en $\Theta(m \log n)$ hora.

3.9.2 Divide y vencerás

La estrategia de dividir y conquistar implica dividir una instancia de problema en varias instancias más pequeñas de tamaños posiblemente similares. Las instancias más pequeñas se resuelven y las soluciones a las instancias más pequeñas se combinan para proporcionar la solución global. La solución de las instancias más pequeñas y la combinación de estas soluciones se realiza de forma recursiva.

3.9.2.1 Tomando el poder con la división y la conquista

Hemos visto el algoritmo recursivo para encontrar el n^{a} potencia de un número entero en la Sección 3.4. Tratemos de encontrar el n^{a} potencia de un número entero utilizando el método de dividir y conquistar este momento. La división del paso implica reducir a la mitad el problema hasta que se encuentre el caso base. Cada llamada a la función resulta en llamar a la misma función con la mitad de la potencia para n y la mitad de uno menos que n para enteros impares, como se muestra en el algoritmo 3.7.

```
Algorithm 3.7 Power_DivideConq
1: function POWER(int a, int n)
2:   if n = 1 then                                > base case
3:     return a
4:   else
5:     if n MOD 2 = 0 then
6:       return Power(a, n/2) × Power(a, n/2)
7:     else return Power(a, (n - 1)/2) × Power(a, (n - 1)/2) × a
8:     end if
9:   end if
10:  end function
```

Por ejemplo, para calcular 2^9 , hacemos las siguientes llamadas que se muestran en la Fig. 3.13.

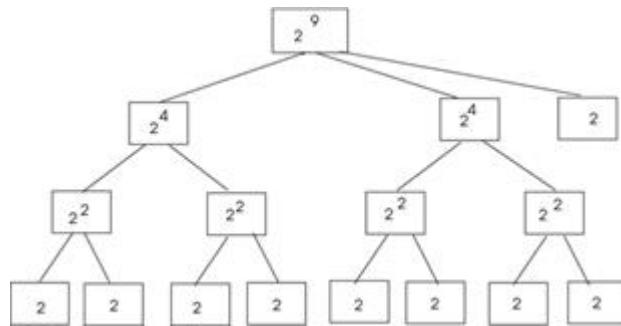


Fig. 3.13 Llamadas recursivas para computar 2^9 por algoritmo 3.7

Este algoritmo tiene la siguiente relación de recurrencia:

$$T(n) = T(n/2) + \Theta(1)$$

que tiene una solución como $\Theta(\log n)$.

3.9.2.2 Serie Fibonacci con Dividir y Conquistar

Probemos otro ejemplo para este método; queremos encontrar la serie de Fibonacci que tiene 0 y 1 como los dos primeros elementos, y cada término posterior es la suma de los dos términos anteriores con $0, 1, 1, 2, 3, 5, \dots, F(n - 1) + F(n - 2)$ como los primeros n términos. Esta serie simple tiene numerosas aplicaciones diversas. Podemos formar este algoritmo de forma recursiva de una manera sencilla como se muestra en el algoritmo 3.8 utilizando el método de dividir y vencer.

Algorithm 3.8 Fibo_DivideConq

```
1: function Fibo(int n)
2:   if n ≤ 1 then
3:     return 1                                ▷ base case
4:   else
5:     return Fibo(n - 1) + Fibo(n - 2)          ▷ recursive call
6:   end if
7: end function
```

Intentemos analizar las llamadas recursivas a esta función. Por ejemplo, $F(9)$ es $F(8) + F(7)$; $F(8)$ es $F(7) + F(6)$ y moviéndonos en esta dirección, alcanzaremos los valores básicos de $F(1)$ y $F(0)$. Note que el valor de $F(7)$ se calcula dos veces para encontrar $F(9)$. Esta relación de recurrencia tiene complejidad exponencial temporal; sin embargo, podemos ver que algunas de las llamadas se repiten, por ejemplo, $F(3)$ tiene que calcularse dos veces y deducir que esta no es la mejor manera de resolver este problema. La solución de programación dinámica proporciona una solución con mayor complejidad, como veremos a continuación.

3.9.3 Programación dinámica

La programación dinámica es un método algorítmico para resolver problemas que tienen subproblemas que se superponen. La palabra programación significa un método tabular en lugar de una programación de computadora real. Las elecciones realizadas se basan en el estado actual y, por lo tanto, se utiliza la palabra dinámica. Un algoritmo de programación dinámica requiere una relación de recurrencia; el cálculo tabular y el rastreo para proporcionar la solución óptima.

En este método, el problema se divide primero en casos más pequeños, los pequeños problemas se resuelven y los resultados se almacenan para usarse en la siguiente etapa. Es similar a dividir y conquistar el método de una manera que divide de forma recursiva la instancia del problema en instancias más pequeñas. Sin embargo, calcula la solución para instancias más pequeñas, las registra y no vuelve a calcular estas soluciones como dividir y conquistar.

3.9.3.1 Serie Fibonacci con Programación Dinámica

Intentemos formar la serie Fibonacci usando programación dinámica esta vez. Todavía dividimos el problema en instancias más pequeñas, pero guardamos los resultados intermedios que se utilizarán más adelante. La solución de programación dinámica para la serie Fibonacci se muestra en el Algoritmo 3.9, donde los resultados intermedios se almacenan en una tabla y se usan posteriormente, lo que resulta en una complejidad de tiempo $O(n)$.

Algorithm 3.9 Dynamic_Fibo

```

1: Input : int  $n$ 
2: Output : array  $F[n]$ 
3: int  $i$ 
4:  $F[0] \leftarrow 0; F[1] \leftarrow 1$ 
5: for  $i=2$  to  $n$  do
6:    $F[i] \leftarrow F[i - 1] + F[i - 2]$ 
7: end for

```

3.9.3.2 Algoritmo de Bellman-Ford

Como ejemplo de aplicación de programación dinámica en gráficos, consideraremos el problema de la ruta más corta. Se nos da un gráfico ponderado que tiene bordes etiquetados con algunos números reales, que muestran los costos asociados con los bordes. Dado un ponderado $G = (V, E, w)$ con $w: E \rightarrow \mathbb{R}$ y un vértice fuente s , nuestro objetivo es encontrar el camino más corto desde el vértice s a todos los otros vértices, la trayectoria que tiene el peso total mínimo entre la fuente y un vértice destino. Bellman – Ford proporcionó un algoritmo de programación dinámica para este propósito que puede trabajar con bordes de peso negativo. Proporciona todas las rutas más cortas y detecta ciclos negativos si existen. La idea principal de este algoritmo es utilizar las rutas más cortas calculadas previamente y, si la ruta calculada actualmente tiene un peso menor, actualice la ruta más corta con la actual. Es un algoritmo de programación dinámico, ya que utilizamos los resultados anteriores sin recalcularlos. En cada iteración, extendemos los vértices accesibles desde la fuente en un salto más como se muestra en el algoritmo 3.10.

Algorithm 3.10 BellFord_SSSP

```

1: Input :  $G = (V, E), s$             $\triangleright$  a directed or undirected edge-weighted graph, a source vertex  $s$ 
2: Output :  $d_u, \forall u \in V$            $\triangleright$  distances to  $s$ 
3:  $d_g \leftarrow 0;$ 
4: for all  $i \neq s$  do            $\triangleright$  initialize distances and predecessors
5:    $d_i \leftarrow \infty$ 
6: end for
7: for  $i = 1$  to  $n - 1$  do
8:   for all  $(u, v) \in E$  do            $\triangleright$  update distances
9:      $d_u \leftarrow \min\{d_u, d_v + w(u, v)\}$ 
10:  end for
11: end for
12: for all  $(u, v) \in E$  do            $\triangleright$  report negative cycle
13:   if  $d_u + w(u, v) > d_v$  then
14:     report "Graph contains a negative cycle"
15:   end if
16: end for

```

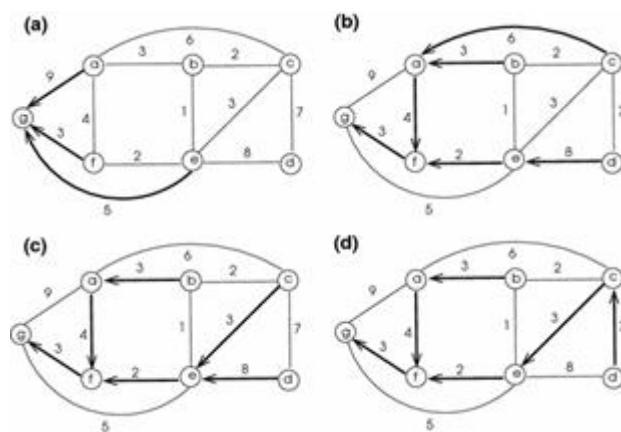


Fig. 3.14 Un ejemplo de ejecución del algoritmo Bellman-Ford en un gráfico de muestra para el vértice fuente g . Las vértices primera alcanzables son una , f , y e que están incluidos en el menor árbol de ruta T . Este árbol se actualiza en cada iteración cuando se encuentran rutas de menor costo en comparación con las anteriores

Ejemplo

En la Fig. [3.14 se](#) muestra un gráfico de muestra no dirigido y ponderado por el borde, donde implementamos este algoritmo con el vértice de origen g . El número máximo de cambios de la ruta más corta para un vértice es $n - 1$, lo que requiere $n - 1$ iteraciones del bucle externo en la línea 8. Cada bucle requiere, como máximo, una verificación de borde m , lo que resulta en una complejidad de tiempo $O(nm)$ para este algoritmo. Veremos una versión más detallada de este algoritmo que también proporciona una estructura de árbol en el Capítulo [7](#).

3.10 Notas de capítulo

En este capítulo proporcionamos una revisión densa de los métodos algorítmicos básicos con enfoque en los algoritmos de grafos. Primero revisamos las estructuras de algoritmos básicos y describimos el análisis asintótico de la complejidad del tiempo de los algoritmos generales. El análisis de caso peor, mejor y promedio de un algoritmo nos da una idea del tiempo que requiere cuando el tamaño de entrada es grande. Notamos la complejidad del tiempo en el peor de los casos, llamada notación grande- O Es comúnmente requerido para que los algoritmos predigan su tiempo de ejecución. Nuestro enfoque al presentar algoritmos es describir verbalmente la idea principal con puntos clave del algoritmo. Luego proporcionamos el pseudocódigo del algoritmo para detallar su operación en un lenguaje de programación como la sintaxis, que se acompaña de una operación de ejemplo en un pequeño gráfico de muestra en muchos casos. Luego mostramos que el algoritmo funciona correctamente usando varios métodos de prueba. También proporcionamos el análisis de temporización del caso más desfavorable del algoritmo de gráfico en consideración. Revisamos los antecedentes necesarios para todos estos pasos, incluidas las principales estrategias de prueba, como las variantes de bucle y la inducción, que se utilizan comúnmente.

El segundo tema que investigamos fue NP-exhaustividad. Clasificamos los problemas como los de P, en NP y los problemas que son NP-hard y NP-complete. Un problema que tiene una solución en el tiempo polinomial está en P; Un problema de decisión que tiene un certificador que puede proporcionar una respuesta de sí o no a un certificado se encuentra en NP. Un problema de NP-completo está en NP y es tan difícil como cualquier otro problema que sea NP-difícil. Los problemas C-SAT y SAT fueron los primeros problemas que se demostró que eran NP-duros y las reducciones se usan para probar que un problema es tan difícil como otro problema. Una vez que sabemos que un problema es NP-difícil, podemos hacer uno de los siguientes. Podemos buscar un algoritmo de aproximación que proporcione una solución subóptima en tiempo polinomial. Se debe probar que un algoritmo de aproximación tiene una relación de aproximación a la solución óptima del problema. Las pruebas de tales algoritmos pueden ser complejas, ya que no conocemos el algoritmo óptimo en sí para mostrar cómo el algoritmo de aproximación converge a él. Podemos usar heurísticas que son reglas de sentido común que se muestran para trabajar con una amplia gama de aportes en la práctica. Sin embargo, no hay garantía de que funcionarán para cada entrada. Sin embargo, para muchos problemas de gráficos en aplicaciones como la bioinformática, el uso de la heurística sigue siendo el único método viable.

En la parte final, describimos los métodos algorítmicos fundamentales que son el método codicioso, la estrategia de dividir y conquistar y la programación dinámica. Los métodos codiciosos intentan encontrar una solución global óptima seleccionando siempre las soluciones óptimas locales. Estas opciones de soluciones locales se basan en lo que se conoce hasta el momento y pueden no, y no conducen a una solución óptima en general. Sin embargo, vimos que los algoritmos codiciosos brindan soluciones óptimas en algunos problemas de gráficos, incluidas las rutas más cortas y los árboles de expansión mínima. En el método de dividir y vencer, el problema en cuestión se divide en una serie de problemas más pequeños que se resuelven y las soluciones se fusionan para encontrar la solución final. Estos algoritmos a menudo emplean la recursión debido a la naturaleza de su operación. La programación dinámica también divide el problema en partes más pequeñas pero hace uso de las soluciones parciales encontradas para obtener la solución general. Los antecedentes que hemos revisado están relacionados principalmente con los algoritmos de grafos secuenciales y veremos que se necesitan más antecedentes y consideraciones para los algoritmos de grafos distribuidos y paralelos en los siguientes capítulos. Hay una serie de libros de algoritmos que proporcionan información básica sobre los algoritmos, incluido el de Skiena [6], Cormen et al. [1], y por Tardos y Kleinberg [5].

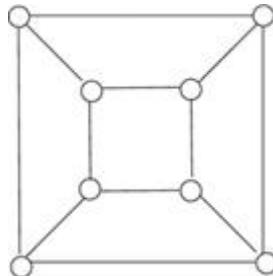


Fig. 3.15 Gráfico de muestra para el Ejercicio 8

Ceremonias

Calcule mediante pruebas los peores tiempos de ejecución para lo siguiente:

$$f(n) = 4n^3 + 5n^2 - 4$$

1.

a.

segundo. $f(n) = 2^n + n^7 + 23$

do. $f(n) = 2n \log n + n + 8$

2. Probar $\frac{3n^4}{n^3}$ no es $O(n^3)$. Tenga en cuenta que debe mostrar que no hay una constante válida c y un umbral n_0 valores para este peor caso para mantener.
3. Escriba el pseudocódigo de un algoritmo recursivo para encontrar la suma de los primeros n enteros positivos. Forme y resuelva la relación de recurrencia de este algoritmo para encontrar su complejidad en el peor momento.
4. Probar $n! \leq n^n$ Por el método de inducción.
5. Una camarilla es el subgrafo de una gráfica en la que cada vértice es adyacente a todos los otros vértices de la camarilla. Dado un grafo $G = (V, E)$ y su subgrafo V' , V' es una camarilla máxima de G si y solo si $G - V'$ es un conjunto máximo independiente. Demuestre que el problema de decisión de encontrar si un gráfico tiene una camarilla máxima de tamaño k es NP-completo por reducción de IND.

6. Encuentre todas las cubiertas mínimas de vértices, los conjuntos máximos independientes y los conjuntos dominantes mínimos del gráfico de muestra que se muestra en la Fig. 3.15 .
 7. Calcule la cubierta de vértices para el gráfico de muestra en la Fig. 3.16 encontrando primero su conjunto independiente máximo y luego la reducción a la cubierta de vértices.
 8. Encuentre la solución a la siguiente fórmula 3-SAT utilizando la reducción a un conjunto independiente:
- $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$
9. Calcule la ruta óptima para el problema TSA en el gráfico que se muestra en la Fig. 3.17 utilizando el límite inferior calculado como la suma de los pesos medios de los dos bordes de peso más ligero de cada vértice. Dibuje el árbol de espacio de estado mostrando todos los nodos podados.
 10. Encuentre el recorrido TSA óptimo para la gráfica de la figura 3.17 esta vez, estableciendo el límite inferior como la suma de los pesos de los bordes de peso más ligero en cada vértice.
 11. Encuentre la cobertura de vértice mínima para el gráfico de muestra en la Fig. 3.18 usando el algoritmo de aproximación de 2. Muestre cada iteración de este algoritmo y calcule la aproximación lograda por sus iteraciones.
 12. Encuentre el MST de la gráfica de muestra de la figura 3.19 utilizando el algoritmo de Kruskal mostrando cada iteración.

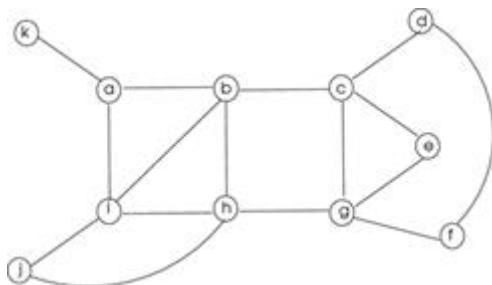


Fig. 3.16 Gráfico de muestra para el ejercicio 9

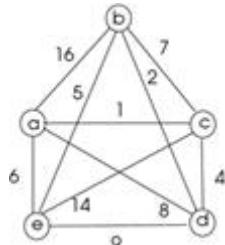


Fig. 3.17 Gráfico de muestra para los ejercicios 11 y 12

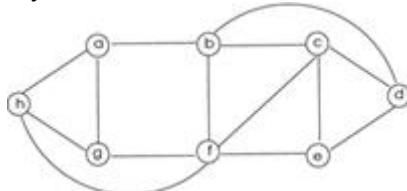


Fig. 3.18 Gráfico de muestra para el ejercicio 13

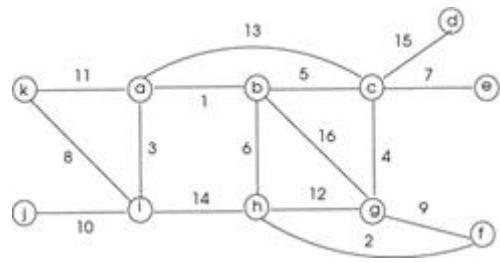


Fig. 3.19 Gráfico de muestra para el ejercicio 14

Referencias

1. Cormen TH, Stein C, Rivest RL, Leiserson CE (2009) Introducción a los algoritmos, 3^a ed. MIT Press, Cambridge ISBN-13: 978-0262033848
2. Dasgupta S, Papadimitriou CH, Vazirani UV (2006) Algoritmos. McGraw-Hill, Nueva York
3. Garey MR, Johnson DS (1979) Computadoras e intratabilidad: una guía para la teoría de la integridad de NP. WH Freeman, Nueva York
4. Karger D (1993) Reducciones mínimas globales en RNC y otras ramificaciones de un algoritmo de mincut simple. En: Actas del 4to simposio anual ACM-SIAM sobre algoritmos discretos
5. Kleinberg J, Tardos E (2005) Diseño de algoritmo, 1^a ed. Pearson, Londres ISBN-13: 978-032129535
6. Skiena S (2008) El manual de diseño de algoritmos. Springer, Berlín ISBN-10: 1849967202

4. Algoritmos de gráficos paralelos

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

En este capítulo investigamos métodos para el diseño de algoritmos paralelos con énfasis en los algoritmos de grafos. La memoria compartida y el procesamiento paralelo de la memoria distribuida son los dos modelos fundamentales en hardware, sistema operativo, programación y niveles algorítmicos de computación paralela. Revisamos estos métodos y describimos el equilibrio de carga estático y dinámico en sistemas de computación en paralelo.

4.1 Introducción

Nosotros hemos revisado los conceptos básicos en algoritmos y métodos principales para diseñar algoritmos secuenciales con énfasis en algoritmos de grafos en el capítulo anterior. Nuestro objetivo en este capítulo es investigar métodos para el diseño de algoritmos paralelos con énfasis en los algoritmos de gráficos nuevamente. El procesamiento paralelo se usa comúnmente para resolver tareas de gran tamaño computacional e intensivas en datos en varios nodos computacionales. El objetivo principal del uso de este método es obtener resultados mucho más rápidos que los que se obtendrían mediante el procesamiento secuencial y, por lo tanto, mejorar el rendimiento del sistema. El procesamiento paralelo requiere el diseño de algoritmos paralelos eficientes y esto no es una tarea trivial como veremos.

Hay varias tareas involucradas en la ejecución paralela de algoritmos; Primero necesitamos identificar las subtareas que se pueden ejecutar en paralelo. Para algunos problemas, este paso se puede realizar convenientemente; sin embargo, muchos problemas son inherentemente secuenciales y veremos que varios algoritmos de grafos se encuentran en esta categoría. Suponiendo que una tarea T se puede dividir en n subtareas paralelas t_1, \dots, t_n , el siguiente problema es asignar estas subtareas a los procesadores del sistema paralelo. Este proceso se denomina asignación y se denota como una función del conjunto de tareas T al conjunto de procesadores P como $M : T \rightarrow P$. Las subtareas pueden tener dependencias para que una subtarea t_j no puede comenzar antes de una subtarea anterior t_i acabados Este es ciertamente el caso cuando t_j envía algunos datos a t_i en cuyo caso empezando t_j sin estos datos no tendría sentido. Si conocemos todas las dependencias de la tarea y también características como los tiempos de ejecución, podemos distribuir las tareas de manera uniforme a los procesadores antes de ejecutarlos y este proceso se

denomina programación estática . En muchos casos, no tenemos esta información de antemano y el equilibrio de carga dinámico se utiliza para proporcionar a cada procesador una parte justa de la carga de trabajo en tiempo de ejecución en función de la variación de carga de los procesos.

Podemos tener un procesamiento paralelo de memoria compartida en el que los nodos computacionales se comunican a través de una memoria compartida y, en este caso, la memoria global debe estar protegida contra accesos concurrentes. En la memoria distribuida, el procesamiento paralelo, la comunicación y la sincronización entre tareas paralelas se manejan enviando y recibiendo mensajes a través de una red de comunicación sin ninguna memoria compartida. La programación paralela implica escribir el código paralelo real que se ejecutará en los procesadores paralelos. Para el procesamiento paralelo de memoria compartida, los procesos ligeros denominados subprocesos son ampliamente utilizados y para aplicaciones de procesamiento paralelo en arquitecturas de memoria distribuida; La interfaz de paso de mensajes (MPI) es un estándar de interfaz comúnmente implementado. Veremos que podemos tener diferentes modelos en los modos de hardware, algoritmo y programación. Estos modelos están relacionados entre sí en cierta medida como se describirá. Por ejemplo, el modelo de paso de mensajes a nivel algorítmico requiere memoria distribuida a nivel de hardware que puede implementarse mediante un modelo de programación de paso de mensajes que ejecuta el mismo código con diferentes datos o diferentes códigos posiblemente con diferentes datos.

Comenzamos este capítulo describiendo conceptos fundamentales del procesamiento paralelo seguidos de la especificación de modelos de computación paralela. Luego investigamos los métodos de diseño de algoritmos paralelos que se centran en algoritmos de gráficos paralelos que requieren técnicas específicas. Los métodos de equilibrio de carga estáticos y dinámicos para distribuir uniformemente las tareas paralelas a los procesadores se resumen y concluimos ilustrando los entornos de programación paralelos.

4.2 Conceptos y terminología

La computación paralela implica el empleo simultáneo de múltiples recursos computacionales para resolver un problema computacional. Los pasos comúnmente aplicados de la computación en paralelo son la partición de la tarea general en una serie de subtareas paralelas: el diseño de la comunicación entre tareas y la asignación de subtareas a los procesadores. Estos pasos dependen de la selección de ciertos criterios en un paso que afecta a los otros procesos. Investigaremos estas pistas en detalle más adelante en este capítulo. Parte de la terminología común utilizada en la computación paralela se puede describir a continuación.

- Paralelismo versus concurrencia : el paralelismo indica que al menos dos tareas se ejecutan físicamente al mismo tiempo en diferentes procesadores. La concurrencia se logra cuando dos o más tareas se ejecutan en el mismo marco de tiempo, pero es posible que no se ejecuten al mismo tiempo físico. Las tareas que se comunican utilizando la memoria compartida en un sistema de un solo procesador son concurrentes pero no son paralelas. La concurrencia es más general que el paralelismo y abarca el paralelismo.
- Paralelismo de grano fino frente a grano grueso : cuando el cómputo se divide en varias tareas, el tamaño de las tareas, así como el tamaño de los datos en los que trabajan, afectan su tiempo de ejecución. En el paralelismo de grano fino , las

tareas se comunican y sincronizan con frecuencia y el paralelismo de grano grueso implica tareas con tiempos de computación más grandes que se comunican y sincronizan con mucha menos frecuencia.

- Vergonzosamente paralelo : el cálculo paralelo consiste en tareas independientes que no tienen interdependencias en este modo. En otras palabras, no hay relaciones de precedencia o comunicación de datos entre ellos. La aceleración alcanzada por estos algoritmos puede estar cerca del número de procesadores utilizados en el sistema de procesamiento paralelo.
- Computación multi-core : un procesador multi-core contiene más de uno de los elementos de procesamiento llamados núcleos . La mayoría de los procesadores contemporáneos son multi-core y la computación multi-core se ejecuta en programas en paralelo en procesadores multi-core. El algoritmo paralelo debe hacer uso de la arquitectura de múltiples núcleos de manera efectiva y el sistema operativo debe proporcionar una programación efectiva de las tareas a los núcleos en estos sistemas.
- Multiprocesamiento simétrico : un multiprocesador simétrico (SMP) contiene una serie de procesadores idénticos que se comunican a través de una memoria compartida. Tenga en cuenta que los SMP están organizados en una escala más gruesa que los procesadores de múltiples núcleos que contienen núcleos en un solo paquete de circuitos integrados.
- Multiprocesadores : un multiprocesador consta de varios procesadores que se comunican a través de una memoria compartida. Por lo general, tenemos un conjunto de microprocesadores conectados por un bus paralelo de alta velocidad a una memoria global. El arbitraje de memoria a nivel de hardware es necesario en estos sistemas.
- Multicomputador : cada procesador tiene memoria privada y generalmente se comunica con otros microcomputadores enviando y recibiendo mensajes. No hay memoria global en general.
- Computación en clúster : un clúster es un conjunto de computadoras conectadas que se comunican y sincronizan mediante mensajes a través de una red para terminar una tarea común. El usuario visualiza un grupo como una sola computadora y actúa como una sola computadora mediante el uso de un software adecuado. Tenga en cuenta que un clúster es una vista más abstracta de un sistema multiprocesador con capacidades de software como el equilibrio dinámico de carga.
- Computación en cuadrícula : una cuadrícula es una gran cantidad de computadoras distribuidas geográficamente que funcionan y cooperan para lograr un objetivo común. Grid computing proporciona una plataforma de computación paralela principalmente para aplicaciones vergonzosamente paralelas debido a retrasos impredecibles en la comunicación.
- Computación en la nube : la computación en la nube permite compartir recursos de computación en red para varias aplicaciones que usan Internet. Brinda servicios de entrega tales como almacenamiento en línea, potencia de cómputo y aplicaciones de usuario especializadas para el usuario.
- Procesamiento paralelo con GPU : una unidad de procesamiento gráfico (GPU) es un coprocesador con capacidades mejoradas de procesamiento gráfico. CUDA

es una plataforma de computación paralela que utiliza GPU para el procesamiento en paralelo formado por NVIDIA [2].

4.3 Arquitecturas paralelas

Una unidad de procesamiento tiene un procesador, una memoria y una unidad de entrada / salida en su forma más básica. Necesitamos una serie de procesadores que deben ejecutarse en paralelo para realizar subtareas de una tarea más grande. Estas subtareas necesitan dos operaciones básicas: la comunicación para transferir los datos producidos y la sincronización. Podemos tener estas operaciones en memoria compartida o configuraciones de memoria distribuida como se describe a continuación. Una tendencia general en la computación paralela es emplear procesadores estándar de propósito general conectados por una red debido a la simplicidad y la escalabilidad de tales configuraciones.

4.3.1 Arquitecturas de memoria compartida

En una arquitectura de memoria compartida, como se muestra en la figura 4.8 a, cada procesador tiene algo de memoria local, y la comunicación y la sincronización entre procesos se realizan mediante una memoria compartida que proporciona acceso a todos los procesadores. Los datos se leen y escriben en las ubicaciones de memoria compartida; sin embargo, debemos proporcionar algún tipo de control sobre el acceso a esta memoria para evitar condiciones de carrera. Podemos tener una serie de módulos de memoria compartida, como se muestra en la Fig. 4.1 b, con la interfaz de red para estos módulos que proporcionan accesos simultáneos a diferentes módulos por diferentes procesadores.

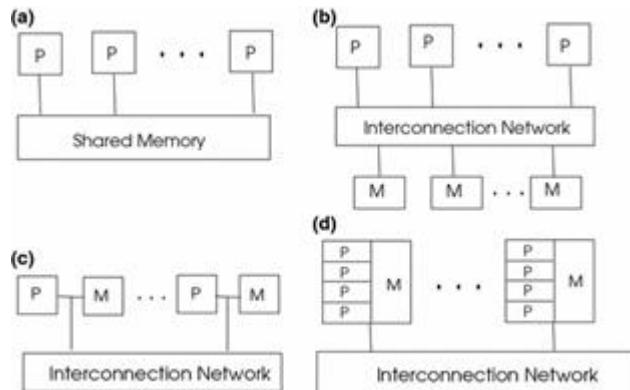


Fig. 4.1 arquitecturas de computación en paralelo, un una arquitectura de memoria compartida con una única memoria global, b una arquitectura de memoria compartida con un número de módulos de memoria, c una arquitectura de memoria distribuida, d una arquitectura de memoria distribuida y compartida

La principal ventaja de los procesadores paralelos de memoria compartida es el rápido acceso de datos a la memoria. Sin embargo, la memoria compartida debe estar protegida contra accesos concurrentes por las tareas paralelas y este proceso debe ser controlado por el programador en muchos casos. Otra desventaja importante del enfoque de memoria compartida es el número limitado de procesadores que se pueden conectar debido al cuello de botella en el bus al acceder a la memoria compartida. En conclusión, podemos afirmar que los sistemas de memoria compartida no son escalables.

4.3.2 Arquitecturas de memoria distribuida

No hay memoria compartida en las arquitecturas de memoria distribuida, y la comunicación y la sincronización se realizan exclusivamente mediante el envío y la recepción de mensajes. Por esta razón, estos sistemas también se denominan paso de mensajes. Cada procesador tiene su propia memoria local, y el espacio de direcciones no se comparte entre los procesadores. Una ventaja importante de los sistemas de memoria distribuida es que podemos usar computadoras disponibles y conectarlas mediante una red para tener un sistema de computación paralelo. El acceso a las memorias locales es más rápido que el acceso a la memoria global; sin embargo, el diseñador de algoritmos debe asumir la responsabilidad de cómo y cuándo sincronizar el procesador y la transferencia de datos. La principal ventaja de los sistemas de memoria distribuida es su escalabilidad y uso de las computadoras disponibles en el mercado. Además, no hay gastos generales en la gestión de la memoria como en la memoria compartida. Como principal inconveniente, la tarea del diseñador de algoritmos es gestionar la comunicación de datos mediante mensajes; y la comunicación a través de la red es comúnmente serial, que es más lenta que la comunicación paralela del sistema de memoria compartida. Además, algunos algoritmos se basan en compartir la conversión de datos global y la asignación de los cuales a la memoria distribuida puede no ser una tarea trivial.

En muchos casos, las aplicaciones de computación paralela contemporáneas utilizan arquitecturas de memoria compartida y distribuida. Los nodos del sistema de computación paralelo son procesadores múltiples simétricos (SMP) que están conectados a través de una red a otros SMP en muchos casos. Cada nodo SMP funciona en modo de memoria compartida para ejecutar sus tareas, pero se comunica con otros nodos en modo de memoria distribuida, como se muestra en la Fig. [4.1 d](#).

4.3.3 La taxonomía de Flynn

La taxonomía de computadoras paralelas de Flynn clasifica las computadoras paralelas según cómo procesan las instrucciones y los datos de la siguiente manera [[3](#)]:

- Computadoras de instrucción única (SISD): estas computadoras ejecutan una instrucción por unidad de tiempo en un elemento de datos. Primero, las computadoras con un procesador operaron en este modo. La mayoría de los procesadores modernos tienen varios núcleos en su unidad de procesamiento.
- Computadoras de datos múltiples (SIMD) de instrucción única : tenemos procesadores paralelos síncronos que ejecutan la misma instrucción en diferentes datos en esta arquitectura. Las matrices de procesadores y las tuberías de vectores son los dos ejemplos principales de este modelo. Estos procesadores paralelos son adecuados para aplicaciones científicas que tratan con grandes datos.
- Computadoras de datos únicos de instrucción múltiple (MISD): estos procesadores ejecutan diferentes instrucciones en los mismos datos. Este modelo no es práctico, excepto en el caso de las etapas de canalización dentro del procesador.
- Computadoras de datos múltiples (MIMD) de instrucción múltiple : en este caso, tenemos procesadores que ejecutan instrucciones diferentes en datos diferentes. Este modelo exhibe el modelo más común y versátil de computación paralela. Los clústeres de equipos paralelos con comunicaciones de red, cuadrículas y supercomputadoras se incluyen en esta categoría.

Las arquitecturas especiales proporcionan enlaces de comunicación que pueden transferir datos entre varios pares de nodos fuente-destino en paralelo. En un hipercubo, los procesadores están conectados como los vértices de un cubo, como se muestra en la Fig. 4.2a para un hipercubo de tamaño 4. La distancia más grande entre dos procesadores es $\log n$ en un hipercubo con n procesadores, y un hipercubo de tamaño d tiene

$n = 2^d$ procesadores. Cada nodo tiene una etiqueta de número entero que tiene una diferencia de un bit de cualquiera de sus vecinos, lo que proporciona una forma conveniente de detectar vecinos mientras se diseñan algoritmos paralelos en el hipercubo.

Una matriz lineal consiste en procesadores conectados conectados en una línea, cada uno con un vecino izquierdo y derecho, excepto el procesador de inicio y terminación, como se muestra en la Fig. 4.2 b. Una red en anillo tiene procesadores conectados en un ciclo como en la Fig. 4.2c. La arquitectura de malla tiene una matriz 2-D de procesadores conectados como una matriz y la arquitectura de árbol equilibrada es un árbol con nodos como procesadores, cada uno de los cuales tiene dos hijos, excepto las hojas, como se muestra en la Fig. 4.2 c, d. Pocas computadoras paralelas, incluyendo Cray T3D, SGI e IBM Blue Gene, tienen estructuras de malla.

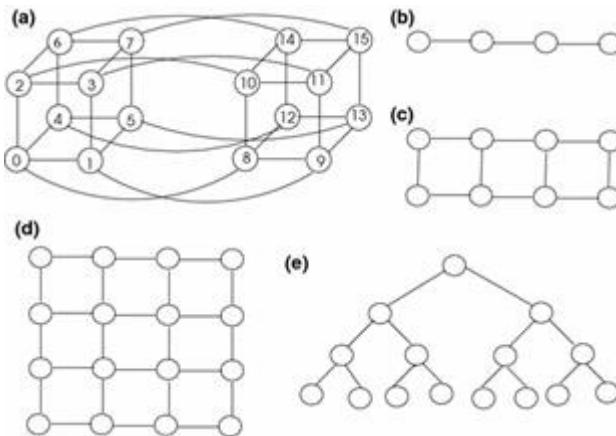


Fig. 4.2 arquitecturas especiales de procesamiento en paralelo, a) un hipercubo de dimensión 4, b) una disposición lineal, c) una malla de dos dimensiones, d) un árbol binario equilibrado

4.4 modelos

Necesitamos un modelo de computación paralela que especifique qué se puede hacer en paralelo y cómo se pueden realizar estas operaciones. A continuación se describen dos modelos básicos basados en restricciones arquitectónicas.

4.4.1 Modelo PRAM

La memoria de acceso aleatorio paralelo (PRAM) extiende el modelo de RAM básico a la computación paralela. El procesador es idéntico a la memoria local, y existe una memoria compartida global utilizada para la comunicación y la sincronización. Por lo tanto, asume la arquitectura de memoria compartida descrita en la Secta. 4.3.1. Los procesadores funcionan sincrónicamente usando un reloj global y en cada unidad de tiempo, un procesador puede realizar una lectura desde una ubicación de memoria global o local; ejecutar una sola operación de RAM y escribir en una ubicación de memoria global o local. Los modelos de PRAM se clasifican de acuerdo con los derechos de acceso de lectura o escritura a la memoria global de la siguiente manera.

- El modelo de escritura exclusiva de lectura exclusiva (EREW) requiere que cada procesador lea o escriba exclusivamente en ubicaciones de memoria global, un procesador a la vez.
- El modelo de escritura exclusiva de lectura concurrente (CREW) permite la lectura concurrente de la misma ubicación de memoria global por más de un procesador; Sin embargo, la escritura en una ubicación de memoria global debe hacerse exclusivamente.
- El modelo de escritura concurrente de lectura concurrente (CRCW) es el más versátil de todos, ya que permite tanto lecturas concurrentes como escrituras concurrentes. En el caso de escrituras simultáneas, se necesita un mecanismo para decidir qué se escribirá como último en la ubicación de la memoria. Puede haber escrituras arbitrarias o escrituras con prioridad.

El modelo PRAM es idealista ya que asume un número ilimitado de procesadores y un tamaño ilimitado de memoria compartida. Sin embargo, simplifica en gran medida el diseño del algoritmo paralelo al abstraer los detalles de la comunicación y la sincronización. Tenemos n procesadores que funcionan de manera síncrona y podemos tener como máximo n operaciones en cualquier unidad de tiempo. Por lo tanto, podemos usar este modelo para comparar varios algoritmos paralelos para resolver un problema dado. Veremos ejemplos de algoritmos de gráficos PRAM en la Secta. [4.8](#).

4.4.2 Modelo de paso de mensajes

El modelo de paso de mensajes se basa en la arquitectura de memoria distribuida. No hay memoria compartida, y las tareas paralelas que se ejecutan en diferentes procesadores se comunican y sincronizan utilizando dos primitivas básicas: enviar y recibir. Las llamadas a estas rutinas pueden estar bloqueando o no bloqueando. Un envío bloqueado detendrá a la persona que llama hasta que se reciba un acuse de recibo del receptor. Por otra parte, una recepción de bloqueo evitará que la persona que llama continúe hasta que se reciba un mensaje específico / general. Podemos asumir que la red es confiable y transfiere los mensajes a su destino con alta confiabilidad y la tarea de recepción necesita los datos que recibe antes de continuar. Por lo tanto, es una práctica general tener un envío no bloqueante y un par de recepción bloqueante como se muestra en la Fig. [4.3 a](#). Es posible que necesitemos el par de rutinas de envío y recepción de bloqueo, como se muestra en la Fig. [4.3b](#) para aplicaciones que requieren alta confiabilidad, como en sistemas paralelos de tiempo real.

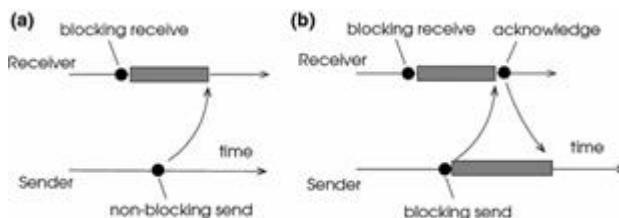


Fig. 4.3 modos de comunicación de bloqueo y no bloqueantes, un una de no bloqueo enviar y un bloqueo recibir, b una de no bloqueo enviar y una de no bloqueo recibir. Los tiempos bloqueados se muestran mediante rectángulos grises. Los retrasos en la red causan la duración entre el envío y la recepción de un mensaje

4.5 Análisis de los algoritmos paralelos.

Necesitamos evaluar la eficiencia de un algoritmo paralelo para decidir su bondad. El tiempo de ejecución de un algoritmo, la cantidad de procesadores que utiliza y su costo se

utilizan para determinar la eficiencia de un algoritmo paralelo. El tiempo de ejecución de un algoritmo paralelo. T_p se puede especificar como

$$T_p = t_{fin} - t_{ini}$$

dónde t_{ini} es la hora de inicio del algoritmo en el primer procesador (el más antiguo) y t_{fin} es el tiempo de finalización del algoritmo en el último (último) procesador. La profundidad D_p de un algoritmo paralelo es el mayor número de pasos dependientes realizados por el algoritmo. La dependencia en este contexto significa que no se puede realizar un paso antes de que termine el anterior, ya que necesita la salida de este paso anterior. Si T_s es el peor tiempo de ejecución de un algoritmo particular A para un problema Q usando p procesadores idénticos y S_p es el peor tiempo de ejecución del algoritmo secuencial más rápido conocido para resolver Q, la aceleración S_p . Se define de la siguiente manera:

$$S_p = \frac{T_s}{T_p}$$

(4.1)

Necesitamos que la aceleración sea lo más grande posible para la eficiencia. El tiempo de procesamiento paralelo. T_p aumenta con el aumento de los costos de comunicación entre procesos , lo que resulta en una menor aceleración. Colocar tareas paralelas en menos procesadores para reducir el tráfico de red disminuye el paralelismo y estos dos enfoques contradictorios deben considerarse cuidadosamente al asignar tareas paralelas a los procesadores. La eficiencia de un algoritmo paralelo se define como

$$E_p = \frac{S_p}{p}$$

(4.2)

Se dice que un algoritmo paralelo es escalable si su eficiencia permanece casi constante cuando aumentan tanto la cantidad de procesadores como el tamaño del problema. La eficiencia de un algoritmo paralelo está entre 0 y 1. Un programa que es escalable con la aceleración se acerca a p. La eficiencia se acerca a 1. Analicemos la suma de n números usando los procesadores k suponiendo que se distribuyen n / k elementos a cada procesador. Cada p_i , $0 \leq i < k$, encuentra su suma local en $\Theta(n/k)$ hora. Luego, se suman las sumas parciales en $\log(k)$. Tiempo por k procesadores resultando en un tiempo total paralelo. $T_p = \Theta(n/k + \log k)$. El algoritmo secuencial tiene una complejidad de tiempo de $T_s = \Theta(n)$. Por lo tanto, la eficiencia de este algoritmo es

$$E_p = \frac{T_s}{kT_p} = \frac{n}{n + k \log k}$$

(4.3)

Necesitamos E_p permanecer como una constante c y resolviendo para n rendimientos.

$$n = \frac{c}{1-c} k \log k$$

(4.4)

Al enumerar los valores de eficiencia contra el tamaño del problema n y el número de procesadores k , podemos ver para (n , k) valores de (64,4), (192,8) y (512,16), la eficiencia es 80% [6] para un máximo de (512,32) valor; Todos los demás valores de eficiencia son más bajos. El costo total o simplemente el costo de un algoritmo paralelo es el tiempo colectivo que toman todos los procesadores para resolver el problema. El costo en pcomputadoras paralelas es el tiempo gastado multiplicado por el número de procesadores como pT_p , y por lo tanto

$$T_p \geq \frac{T_s}{p}$$

que se llama la ley del trabajo. Estamos interesados en la cantidad de pasos tomados en lugar de la duración del tiempo físico del algoritmo paralelo. Un algoritmo paralelo tiene un costo óptimo si el trabajo total realizado W es

$$W = pT_p = \Theta(T_s)$$

En otras palabras, cuando su costo es similar al costo del algoritmo secuencial más conocido para el mismo problema, el paralelismo alcanzado [¶] Se puede especificar en términos de estos parámetros como a continuación:

$$\varphi = \frac{W}{S}$$

Ilustremos estos conceptos con otro algoritmo paralelo para sumar 8 números; esta vez cada procesador hace una sola adición. En la Fig. 4.8 se muestra una posible implementación de este algoritmo utilizando cuatro procesadores . Podemos ver que la cantidad de pasos dependientes que es la profundidad de este algoritmo es 3 y el trabajo total realizado es 7, ya que las adiciones de 4, 2 y 1 se realizan en los pasos 1, 2 y 3. Estos problemas se denominan problemas de circuito. que incluyen la búsqueda de valores mínimos / máximos en una matriz y su profundidad es $\log n$ como puede verse. El trabajo total realizado en estos algoritmos es n -1 (Fig. 4.4).

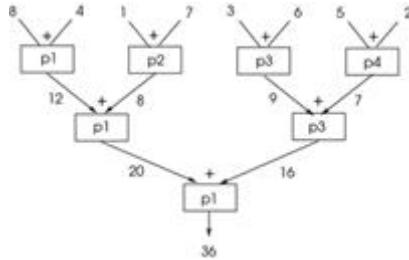


Fig. 4.4. Un algoritmo de suma paralela con cuatro procesadores. p_1, p_2, p_3, p_4 sumar 8 numeros

4.6 Modos básicos de comunicación

Los procesos que residen en diferentes nodos informáticos del sistema paralelo deben comunicarse para finalizar una tarea general. Podemos distinguir los dos modos básicos de comunicación entre los procesos: o todos los procesos involucrados en la tarea paralela se comunican o un grupo de procesos se comunican entre sí. Visto desde otro ángulo, también está la arquitectura del hardware que debe considerarse al especificar estos modos de comunicación. Las siguientes son las comunicaciones básicas entre todos los n procesos del sistema o el grupo [6].

- Transmisión única : un proceso p_i transmite un mensaje m a todos $n - 1$ Otros procesos. El mensaje m se copia en las memorias privadas de todos los procesos en el modelo de memoria distribuida y en la memoria global en el modelo de memoria compartida. En una red con estructura de árbol, este modo se puede realizar mediante la raíz enviando un mensaje m a sus hijos que transfieren m a sus hijos hasta que las hojas del árbol lo reciban. Esta transferencia se logra en $\Theta(\log n)$ Tiempo por la transferencia de mensajes $O(n)$.
 - Reducción de todos a uno : los datos de cada proceso del sistema paralelo se combinan mediante un operador asociativo y el resultado se almacena en una única ubicación de memoria de un proceso específico.
 - Transmisión de todos a todos y reducción de todos a todos : tenemos cada proceso p_i , $0 \leq i < n$ de n procesos enviando mensaje m_i a todos los demás procesos simultáneamente. Cada proceso almacena mensajes. m_i , $0 \leq i < n$ Al final de esta comunicación. En la reducción de todos a todos , el modo es la operación inversa de la transmisión de todos a todos en la que cada proceso p_i almacena n mensajes distintos enviados por otros procesos al final.
 - Operación de dispersión y recolección : se envía un mensaje único personalizado para cada proceso desde un proceso en la operación de dispersión . El proceso fuente p_i tiene mensajes m_j , $0 \leq j < n$, $j \neq i$ Para cada uno de los procesos y cada proceso. p_j , $j \neq i$, recibe su mensaje m_j en el final. El procedimiento inverso se realiza en la operación de recopilación , y cada proceso envía su mensaje único para que se recopile en un proceso específico.
 - Comunicaciones personalizadas para todos : en este modo, cada proceso p_i tiene un mensaje distinto m_i y envía este mensaje a todos los demás procesos. Todos los procesos en el sistema reciben mensajes. m_i , $0 \leq i < n$ Al final de esta comunicación.

Intentemos implementar la comunicación de difusión general en un hipercubo de dimensión d . Cada nodo i tiene un mensaje distinto m_i . Al principio y requerimos que cada nodo tenga todos los mensajes. $m_i, 0 \leq i < n$, con $n = 2^d$. Al final del algoritmo. Podemos hacer que

todos los vecinos intercambien sus mensajes a lo largo del eje x en el primer paso, luego intercambiar el resultado obtenido a lo largo del eje y , y finalmente a lo largo del eje z en el último paso para un hipercubo 3D como se muestra en la Fig. 4.2 . Tenga en cuenta que el tamaño de los mensajes transmitidos se duplica en cada paso y el número total de pasos es d (Fig. 4.5).

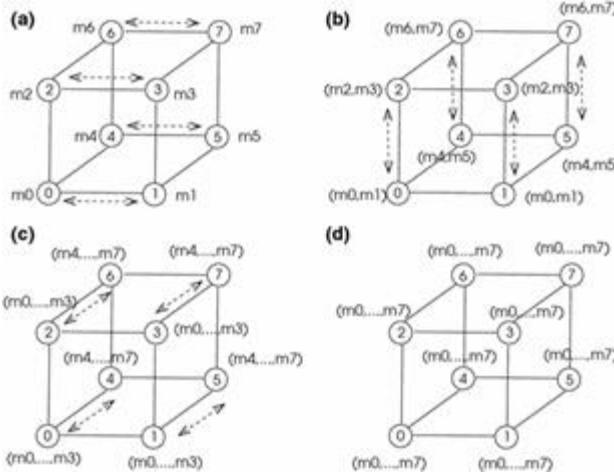


Fig. 4.5 Comunicación de todos a todos en un hipercubo 3D. El primer intercambio de datos entre vecinos es en la dirección x , luego y y finalmente en las direcciones z en a , b y c consecutivamente

El problema aquí es escribir un algoritmo que proporcione la transferencia de datos especificada anteriormente. Podemos hacer uso del etiquetado de los nodos en un hipercubo que difiere en un bit de cualquier vecino. La diferencia de bit menos significativa (LSB) de un nodo desde un vecino está en la dirección x , la segunda diferencia LSB está en la dirección y , y así sucesivamente. Por lo tanto, podemos hacer OR exclusivo a nivel de bit de una identidad de un nodo para encontrar la identidad de su vecino. Por ejemplo, el nodo 5 (0101B) cuando XOR con resultados 0001b en 4, que es el vecino del nodo 5 en x dirección. Algoritmo 4.1 hace uso de esta propiedad y selecciona vecinos (relincho) para la transferencia que tienen diferencia de 1 bit en cada iteración. El número total de pasos es la dimensión.d del hipercubo.

4.7 Métodos de diseño de algoritmos paralelos

Podemos emplear una de las siguientes estrategias al diseñar un algoritmo paralelo; modifique un algoritmo secuencial existente mediante la detección de subtareas que se pueden realizar en paralelo, que es, con mucho, uno de los enfoques más utilizados. Alternativamente, podemos diseñar un algoritmo paralelo desde cero o podemos iniciar el mismo algoritmo secuencial en varios procesadores pero con condiciones iniciales diferentes, posiblemente aleatorias, y el primero que termina se convierte en el ganador. Foster propuso un enfoque de diseño de cuatro pasos para el procesamiento paralelo que se detalla a continuación [4]. Veremos estos pasos con más detalle en las siguientes secciones.

Algorithm 4.1 All-to-All Broadcast on Hypercube

```

1: procedure ATA_BCAST_HC(my_id, my_msg, d, all_msgs)
2:   for i ← 0 to d - 1 do
3:     neigh ← my_id ⊕ 2i
4:     send all_msgs to neigh
5:     receive msg from neigh
6:     all_msgs ← all_msgs ∪ msg
7:   end for
8: end procedure

```

Particionamiento : los datos, la tarea general o ambos, se pueden dividir en varios procesadores. La partición de datos se denomina datos o descomposición de dominio y la partición de código se denomina descomposición funcional .

- 1.
2. Comunicación : La cantidad de datos y el envío y la recepción de subtareas paralelas se determinan en este paso.
3. Aglomeración : las subtareas determinadas en los dos primeros pasos se organizan en grupos más grandes con el objetivo de reducir la comunicación entre ellos.
4. Mapeo : los grupos formados se asignan a los procesadores del sistema paralelo. Cuando se construye el gráfico de tareas que describe las subtareas y su comunicación, los dos últimos pasos de esta metodología se reducen al problema de partición de gráficos, como lo haremos.

4.7.1 Paralelismo de datos

El método de paralelismo o descomposición de datos comprende la ejecución simultánea de la misma función en los elementos de un conjunto de datos. Dividiendo datos a k procesadores. p_1, \dots, p_k , cada uno de los cuales trabaja en su partición puede aplicarse tanto a la PRAM como a los modelos de paso de mensajes, y este método se implementa ampliamente en muchos problemas de computación en paralelo. Ilustremos este enfoque mediante la multiplicación de matrices. Necesitamos formar el producto C de dos. $n \times n$ las matrices A y B y las dividimos como $n / 2, n / 2$ submatrices para cuatro procesos de la siguiente manera:

$$\begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \times \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

Las tareas a realizar por cada proceso. p_1, \dots, p_4 Ahora se puede indicar de la siguiente manera:

$$C_1 = (A_1 \times B_1) + (A_2 \times B_3) \rightarrow p_1$$

$$C_2 = (A_1 \times B_2) + (A_2 \times B_4) \rightarrow p_2$$

$$C_3 = (A_3 \times B_1) + (A_4 \times B_3) \rightarrow p_3$$

$$C_4 = (A_3 \times B_2) + (A_4 \times B_4) \rightarrow p_4$$

Simplemente podemos distribuir las particiones asociadas de matrices a cada proceso en el modelo de paso de mensajes, por ejemplo A_1, A_2, B_1 y B_3 a p_1 , o hacer que los procesos trabajen en sus particiones relacionadas en la memoria compartida en el modelo PRAM. En el modelo supervisor / trabajador de procesamiento paralelo, tenemos un proceso,

digamos p_1 , que tiene todas las entradas que son las matrices A y Ben este caso. Este nodo supervisor es responsable de la distribución de los datos iniciales a los procesos de trabajo y luego recopila los resultados. También puede involucrarse en el cálculo de los resultados si hay un desequilibrio de carga tal que el supervisor permanezca inactivo cuando otros procesos participan en el cálculo. Intuitivamente, cuando hay una gran cantidad de procesos con una comunicación densa necesaria, el rol del supervisor se puede limitar a administrar el flujo de datos básico y proporcionar el resultado. Alternativamente, en el modelo totalmente distribuido, todos los procesos son iguales a los datos de entrada proporcionados a todos. Cada nodo en la red trabaja en su partición pero intercambia mensajes para tener el resultado total almacenado en ellos. Este modo se puede usar en el primer paso de una tarea paralela si cada proceso necesita resultados totales como datos de entrada para el siguiente paso de los cálculos individuales. Hemos usadola partición de bloques de las matrices de entrada en este ejemplo, donde la matriz se divide en bloques de igual tamaño como se muestra en la Fig. 4.6 para una 8×8 matriz donde tenemos 16 procesos, p_1 a p_{16} cada uno tiene un 2×2 Partición de la matriz.

1	2	3	4	5	6	7	8
p_1	p_2	p_3	p_4				
p_5	p_6	p_7	p_8				
p_9	p_{10}	p_{11}	p_{12}				
p_{13}	p_{14}	p_{15}	p_{16}				

Fig. 4.6 Bloqueo de partición de una matriz.

Particionamiento de filas

Consideremos una matriz A [n , n] con n filas y n columnas. En la partición de filas, simplemente dividimos la matriz A a k partes de manera que p_i consigue $\frac{(i-1)n/k+1}{n}$ a en $\frac{n}{k-1}$ filas. Dicha partición se representa en la Fig. 4.7a para una 8×8 matriz con cuatro procesos p_1 , p_2 , p_3 y p_4 .

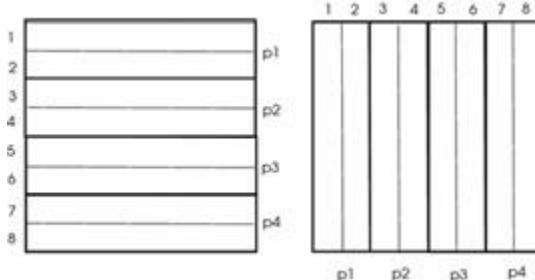


Fig. 4.7 Partición por filas y por columnas de una matriz

Partición de la columna en forma de columna

En partición por columnas de una matriz A [n , n], cada proceso tiene ahora n / k columnas consecutivas de A . Colocación en columna de una 8×8 La matriz con cuatro procesos se muestra en la figura 4.7 b. La partición por filas y por columnas de una matriz se denomina partición 1-D y la partición de bloque se denomina comúnmente partición 2-D de una matriz.

4.7.2 Paralelismo funcional

El paralelismo funcional o el enfoque de descomposición funcional implica aplicar diferentes operaciones a datos posiblemente diferentes simultáneamente. También

llamado paralelismo de tareas , este modelo implica la ejecución simultánea de varias funciones en múltiples procesadores en el mismo conjunto de datos o en diferentes. El método algorítmico de dividir y vencer se puede implementar de manera eficiente utilizando este modelo. Hemos revisado el método de división y conquista secuencial en el que dividimos recursivamente el problema en instancias más pequeñas, resolvimos los casos más pequeños y fusionamos los resultados de forma recursiva para obtener el resultado final. Los algoritmos de dividir y conquistar son buenos candidatos para la parallelización, ya que las recursiones se pueden hacer en paralelo. Consideraremos agregar nNúmeros usando divide y vencerás como se muestra en el algoritmo 4.2.

Algorithm 4.2 Recursive Sum

```

1: function SUM(A[1..n])
2:   if n = 1 then                                ▷ base case
3:     return A[1]
4:   else
5:     x ← Sum(A[1..n/2])
6:     y ← Sum(A[n/2 + 1..n])
7:     return x + y                               ▷ recursive call
8:   end if
9: end function

```

El análisis de este algoritmo muestra que obedece a la relación de recurrencia. $T(n) = 2T(n/2) + 1$ que tiene solución $T(n) = O(n)$. Para tener una versión paralela de este algoritmo, notamos que las llamadas recursivas son independientes ya que operan en diferentes particiones de datos; por lo tanto, podemos realizar estas llamadas en paralelo simplemente realizando las operaciones dentro de la sentencia else entre las líneas 4 y 8 del algoritmo 4.2 en paralelo. La relación de recurrencia para este algoritmo en este caso es $T(n) = T(n/2) + 1$ que tiene solución $T(n) = O(\log n)$.

4.8 Métodos de algoritmos paralelos para gráficos

El método de dividir y conquistar requiere que la estructura del gráfico se divida en gráficos más pequeños y esto no es una tarea fácil debido a las estructuras irregulares de los gráficos. La partición de datos para algoritmos de gráficos paralelos significa una partición equilibrada de gráficos entre los procesadores, lo que es un problema NP-difícil. Necesitamos métodos radicalmente diferentes para los algoritmos de gráficos paralelos, y la contracción de gráficos, el salto de puntero y la aleatorización son los tres enfoques fundamentales para este propósito.

4.8.1 Aleatorización y ruptura de simetría

Un algoritmo aleatorio toma ciertas decisiones en función del resultado de los lanzamientos de monedas durante la ejecución del algoritmo, como vimos en el Capítulo. 3 . Estos algoritmos asumen que cualquier combinación de entrada es posible. Las dos clases principales de algoritmos aleatorios son los algoritmos de Las Vegas y Monte Carlo , como hemos descrito.

Los algoritmos aleatorios se pueden usar de manera efectiva para la solución paralela de varios problemas de gráficos. El descubrimiento de los componentes conectados de un gráfico, la búsqueda de conjuntos independientes máximos y la construcción de árboles de expansión mínimos de un gráfico pueden realizarse mediante algoritmos aleatorios paralelos, como veremos en la Parte II.

La ruptura de simetría en un algoritmo de gráfico paralelo implica la selección de un subconjunto de un gran conjunto de operaciones independientes que utilizan alguna propiedad del gráfico. Por ejemplo, la búsqueda de todos los vértices candidatos para el

conjunto máximo independiente (MIS) de una gráfica se puede hacer en paralelo. Sin embargo, no podemos tener ambos vértices adyacentes incluidos en el MIS, ya que esto viola la definición de MIS. Un procedimiento de ruptura de simetría puede seleccionar el vértice con un identificador inferior o un grado inferior. En general, la ruptura de simetría se puede emplear para corregir la salida cuando las operaciones paralelas independientes en los vértices o bordes de un gráfico producen un resultado grande y posiblemente incorrecto.

4.8.2 Partición de gráficos

Dado un gráfico no dirigido no ponderado $G = (V, E)$, la tarea de partición de gráficos es dividir el conjunto de vértices V en conjuntos de vértices separados V_1, \dots, V_k de tal manera que el número de vértices en cada partición es aproximadamente igual y el número de bordes entre los subgrafos inducidos por los vértices en la partición es mínimo. Este proceso se representa en la Fig. 4.8, donde un gráfico se divide en tres subgrafos equilibrados. El vértice y los bordes pueden tener pesos asociados que representan algún parámetro físico relacionado con la red representada por un gráfico. En tal caso, nuestro objetivo en la partición es tener una suma de pesos de vértices aproximadamente igual en cada partición con una suma mínima total de ponderaciones de borde entre las particiones.

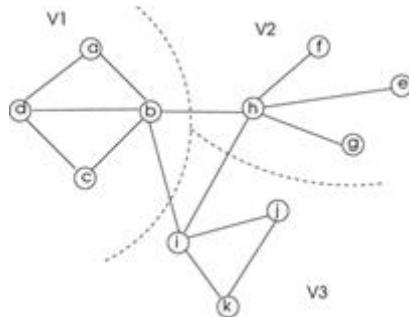


Fig. 4.8 Partición de una gráfica de muestra en tres conjuntos de vértices desunidos; $V_1 = \{a, b, c, d\}$, $V_2 = \{f, g, h, e\}$ y $V_3 = \{j, k, l\}$. Un número de vértices en estas particiones son 4, 4 y 3, respectivamente, y hay un total de tres bordes entre las particiones

En PRAM y modelo de memoria distribuida, cada proceso. ^{p1} Funciona en su partición. Asumir que el trabajo realizado por un procesador es una función de la cantidad de vértices y bordes en su partición, la carga se distribuye uniformemente. Sin embargo, los bordes entre particiones y los vértices de borde deben manejarse con cuidado al obtener la solución general al problema. Los vértices de borde duplicados en las particiones a las que no pertenecen se llaman vértices fantasma y el uso de estos nodos ayuda a superar las dificultades encontradas en los límites de la partición. Kernighan y Lin propusieron un algoritmo simple y efectivo a principios de 1970 para dividir una gráfica de forma recursiva [9]. Básicamente se utiliza para mejorar una partición existente mediante el intercambio de vértices entre las particiones para reducir el costo de los bordes entre particiones. En el método de partición de gráficos de varios niveles, el gráfico $G = (V, E)$ se convierte en una pequeña gráfica $G' = (V', E')$. Usando heurísticas adecuadas, una partición de k -way de G' se calcula, y la partición obtenida se proyecta de nuevo al gráfico original G [7]. Una formación paralela de este método que utiliza la coincidencia máxima durante la fase de engrosamiento se presenta en [8].

4.8.3 Contracción del gráfico

El método de contracción de gráficos implica obtener gráficos más pequeños al reducir el gráfico original en cada paso. Este esquema es útil para diseñar algoritmos de gráficos

paralelos eficientes en dos aspectos. Se puede realizar convenientemente en $O(\log n)$ pasos como el tamaño de la gráfica se reduce por un factor constante en cada paso. Por lo tanto, si podemos encontrar alguna forma de contraer un gráfico en paralelo mientras resolvemos simultáneamente un problema de gráfico durante la contracción, entonces tenemos un algoritmo de gráfico paralelo adecuado. Como veremos, es posible buscar una solución a algunos problemas de gráficos, como la expansión mínima de árboles durante la contracción. Además, la selección cuidadosa del método de contracción mantiene las propiedades básicas del gráfico y, por lo tanto, podemos resolver el problema en un gráfico más pequeño en paralelo y luego combinar las soluciones para encontrar la solución para el gráfico original.

Supongamos que tenemos un gráfico de entrada G y obtenemos G_1, \dots, G_k . Pequeños gráficos después de la contracción. Podemos resolver el problema en paralelo en estos pequeños gráficos y luego fusionar las soluciones. Una plantilla de algoritmo de contracción gráfica que se muestra en el Algoritmo 4.3 sigue una estructura de algoritmo recursiva típica con el caso base y un caso inductivo. Cuando alcanzamos el caso base, comenzamos a calcular la función requerida en el gráfico pequeño y luego recursionamos en este gráfico pequeño. Tenga en cuenta que la partición de vértice debe ser desarticulada.

Algoritmo 4.3 Graph Contraction

```

1: procedure CONTRACT_GRAPH( $G = (V, E)$ )
2:    $G \leftarrow G_i$ 
3:   if graph  $G_i$  is small enough then ▷ base case
4:     compute the required task on  $G_i$ 
5:     return
6:   else
7:     compute vertex partitioning  $V_i = V_1, \dots, V_k$  of  $G_i$ 
8:     contract each  $V_j \in V_i$  into a supervertex  $V_k$ 
9:     remove edges inside each supervertex  $V_k$ 
10:    merge multiple edges between supervertices
11:     $G_j(V_j, E_j) \leftarrow$  contracted graph
12:    return Contract_Graph( $G_j(V_j, E_j)$ ) ▷ recursive call
13:   end if
14: end procedure

```

Consideremos el ejemplo de la Fig. 4.10 donde dividimos el conjunto de vértices en cuatro subconjuntos que tienen un vértice representativo como se muestra. Suponiendo que el gráfico recién formado sea lo suficientemente pequeño, ahora podemos resolver el problema original en el gráfico pequeño que es el caso base y pedir ayuda para obtener la solución completa (Fig. 4.9).

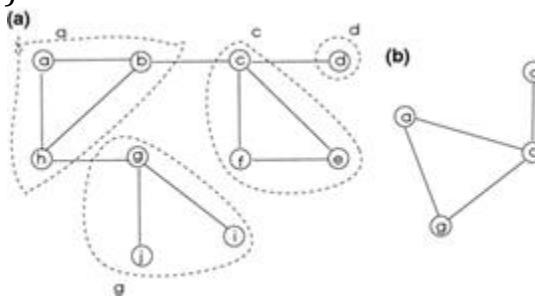


Fig. 4.9 Contracción de una muestra gráfica. El conjunto de vértices en a se divide en cuatro subconjuntos y cada subconjunto está representado por uno de los vértices en el subconjunto para obtener el gráfico contraído en b

4.8.3.1 Contracción del borde

La división de bordes de un gráfico implica seleccionar bordes distintos o vértices aislados que formarán las particiones. Luego, contratamos los vértices en pares que inciden en estos bordes seleccionados en este método. Dado que dos vértices y el borde entre ellos se contraen para tener un solo vértice, los bordes seleccionados no deben compartir puntos

finales, lo que de hecho es el problema de coincidencia de gráficos que se describe a continuación.

Definición 4.1 Una coincidencia en un gráfico $G = (V, E)$ es un conjunto $E' \subseteq E$ de sus bordes tal que cualquier $v \in V$ es incidente a lo sumo una ventaja en E' .

En otras palabras, los bordes en la coincidencia son desunidos sin vértices compartidos. Un acoplamiento máximo en un gráfico de G no se puede ampliar mediante la adición de nuevos bordes y la máxima adaptación en G es el juego con el tamaño máximo entre todos los matchings en G . Por lo tanto, podemos ver la partición de borde y el problema de contracción como encontrar de forma recursiva la coincidencia máxima E' en una gráfica G , contracción del borde en E para obtener G' y continuando con la búsqueda de máxima coincidencia en G' , y así. Para la contracción del gráfico, se puede usar una coincidencia suficientemente grande en lugar de una máxima. Buscaremos el problema de coincidencia de gráficos con más detalle en el capítulo [9](#).

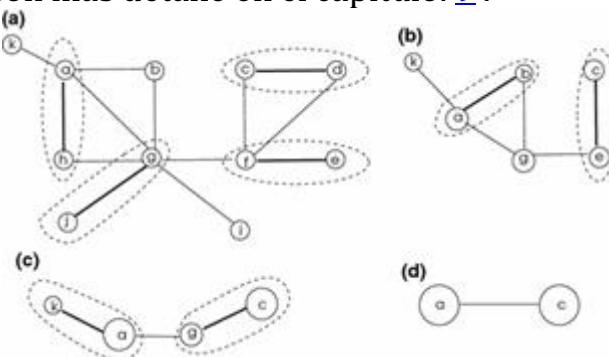


Fig. 4.10 Contracción del borde de un gráfico de muestra en a . Se encuentra una coincidencia máxima y se contrata la gráfica para obtener la gráfica en b . El tamaño de las particiones se amplía a medida que se incluyen más vértices y la etiqueta de una partición es la etiqueta más pequeña que tiene lexicográficamente. La gráfica final tiene dos supervertices.

Con el fin de encontrar una contracción borde de un gráfico, podemos utilizar un simple algoritmo voraz que recoge un borde (u, v) de manera arbitraria en G , borra u, v y todos aristas incidentes a U y V de G , y repite hasta que no se los bordes quedan Sin embargo, realizar una comparación paralela no es tan simple, ya que más de un vértice puede seleccionar el mismo vértice para la comparación. Una forma de superar este problema es usar la aleatorización para romper la simetría. Podemos seleccionar cada borde para que se incluya en la coincidencia de M con una probabilidad de 0.5. Si se selecciona un borde (u, v) para estar en M y todos los demás bordes incidentes en u o v no están incluidos en M , entonces (u, v) se decide que esté en M [1]. Este esquema asegura encontrar la correspondencia correcta en paralelo.

4.8.3.2 Contracción en estrella

Un gráfico de estrellas es un gráfico no dirigido con un vértice central y vértices que están conectados directamente por vértices al centro como se muestra en la Fig. [4.11](#) . En la contracción en estrella , seleccionamos un vértice en cada componente, ya que el centro de la estrella y todos los demás vértices conectados directamente a este centro, los satélites llamados , se contraen para formar un supervertece .

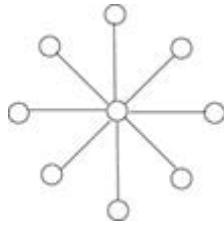


Fig. 4.11 Una red de estrellas con un centro y ocho satélites.

En una configuración secuencial, podemos seleccionar un vértice arbitrariamente como centro y contraer a todos sus vecinos, eliminar la estrella de la gráfica y continuar hasta que (no queden vértices). La figura 4.12 muestra la contracción secuencial de la estrella.

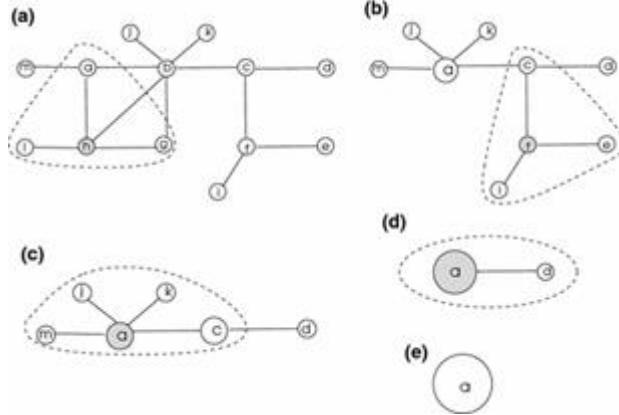


Fig. 4.12 Iteraciones de la contracción en estrella de un gráfico de muestra. Los vértices contraídos se muestran como círculos agrandados y los centros sombreados.

Un algoritmo de contracción en estrella paralela puede hacer uso de la aleatorización y la ruptura de simetría como en la contracción del borde. Esta vez, cada vértice selecciona ser centro o un satélite con probabilidad 0.5. Un vértice u que selecciona ser un centro se convierte en un centro; sin embargo, un vértice v que selecciona ser un satélite busca un vecino que se haya convertido en un centro. Si tal vecino existe, se convierte en un satélite de ese vecino. De lo contrario, el vértice v se convierte en un centro. Si hay más de un centro vecino de v , selecciona uno arbitrariamente como su centro [1].

4.8.4 Salto del puntero

Consideremos una lista enlazada de n elementos con cada elemento que apunta al siguiente elemento de la lista. El método de salto de puntero proporciona a cada elemento para que apunte al final de la lista después de $\log n$ pasos. En cada paso del algoritmo, cada elemento apunta al elemento señalado por su sucesor como se muestra en el algoritmo 6.1. Este algoritmo se puede ejecutar en paralelo, ya que cada actualización del puntero se puede realizar en paralelo.

Algorithm 4.4 Pointer Jumping

```

1: Input : a linked list  $L$  with  $n$  elements
2: Output : modified  $L$ 
3: for  $i = 1$  to  $\lceil \log n \rceil$  do
4:   for each list element  $a$  in parallel do
5:      $a.next \leftarrow (a.next).next$ 
6:   end for
7: end for

```

La operación de este algoritmo para una lista enlazada de ocho elementos se muestra en la Fig. 4.13. Podemos usar esta plantilla para el diseño de algoritmos de gráficos paralelos que usan listas vinculadas, ya que los gráficos se pueden representar por

adyacencia o listas de bordes. El método de salto de puntero es adecuado para el modelo PRAM con memoria compartida.

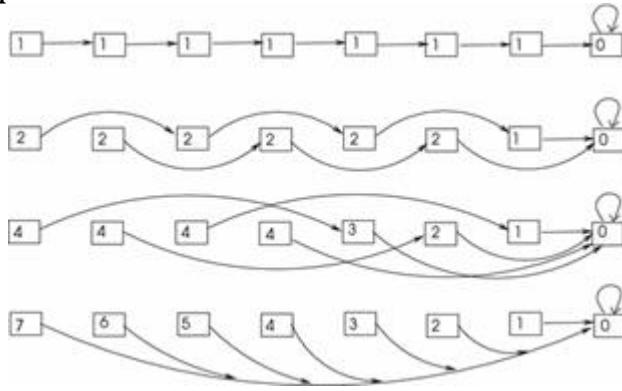


Fig. 4.13 Método de salto de puntero en una lista enlazada de ocho elementos. Después de tres pasos, todos los elementos apuntan al final de la lista. El algoritmo de clasificación de lista también se muestra en esta figura al final de la cual todos los nodos tienen distancias a la cabeza almacenada

Ranking de listas

Dada una lista enlazada L , la búsqueda de la distancia desde cada nodo de L al nodo terminal se denomina clasificación de lista . El algoritmo 6.1 se puede modificar para calcular estas distancias como se muestra en el algoritmo 4.5.

Algorithm 4.5 List Ranking

```

1: Input : a linked list  $L$  with  $n$  elements as ( $dist, next$ )
2: Output : distances of list elements to the list head
3: for each list element  $a$  in parallel do                                > initialization
4:   if  $a.next = \emptyset$  then
5:      $a.dist \leftarrow 0$ 
6:   else  $a.dist \leftarrow 1$ 
7:   end if
8: end for
9: for  $i=1$  to  $\lceil \log n \rceil$  do
10:   for each list element  $a$  in parallel do
11:     if  $a.next \neq \emptyset$  then
12:        $a.dist \leftarrow a.dist + (a.next).dist$ 
13:        $a.next \leftarrow (a.next).next$ 
14:     end if
15:   end for
16: end for
```

La línea 12 de este algoritmo para un nodo a implica leer la distancia del siguiente nodo de a y luego agregar esta distancia para poseer y escribir la suma como la nueva distancia a a . Estas dos operaciones consecutivas se pueden realizar en tiempo constante solo en el modelo CRCW PRAM. Para proporcionar la versión EREW de este algoritmo, necesitamos reemplazar la línea 12 por una línea de lectura y una línea de escritura a continuación, ambas deben ejecutarse en el modo EREW. La complejidad del tiempo de este algoritmo es $O(\log n)$.

```

temp = (a.next).d
a.d = temp
```

4.8.5 Descomposición del oído

Una descomposición de oído de un gráfico se define de la siguiente manera.

Definición 4.2 (descomposición de oído) Una descomposición de oído de un gráfico $G = (V, E)$ es la unión de caminos P_0, P_1, \dots, P_n con P_0 siendo un ciclo simple y P_i ($1 \leq i \leq n$) es un camino con ambos puntos finales en $P_0 \cup P_1 \cup \dots \cup P_{i-1}$.

En la figura 4.14 se representa una descomposición de la oreja de un gráfico . Una gráfica tiene una descomposición de la oreja si y solo si no tiene puentes. Una descomposición de

oído de un gráfico se puede usar para determinar si dos bordes forman parte de un ciclo común que se puede usar para resolver algunas propiedades del gráfico, como la conectividad y la planaridad. Una descomposición de oído de un gráfico se puede encontrar en tiempo logarítmico en paralelo y, por lo tanto, estos problemas se pueden resolver en paralelo convenientemente utilizando este método.

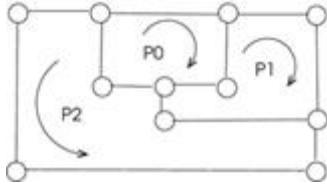


Fig. 4.14. Una descomposición de oído de una gráfica.

4.9 Asignación del procesador

La asignación de subtareas de una tarea a procesadores de modo que cada procesador se utilice de manera eficiente durante el cómputo es una de las tareas importantes en el cómputo paralelo. Podemos conocer las características de la subtarea, como su tiempo de cálculo, tiempos de comunicación y duraciones, y las subtareas que comunica de antemano, en cuyo caso podemos aplicar estrategias de programación estática. Si no se conocen estos parámetros, los métodos de equilibrio dinámico de carga se usan comúnmente para mantener a todos los procesadores ocupados en todo momento.

4.9.1 Asignación estática

La asignación estática de tareas a los procesadores se realiza antes de la ejecución del algoritmo. Necesitamos conocer el tiempo de ejecución de las tareas, su interacción y el tamaño de los datos transferidos entre tareas para poder realizar esta asignación. Asumiremos el tiempo de cálculo de una subtarea. t_i ; Sus antecesores son las subtareas que deben terminar antes. t_i y sus sucesores que pueden comenzar cuando t_i . Los acabados son conocidos de antemano con la duración de cada comunicación. En este caso, podemos dibujar un gráfico de dependencia de tarea que ilustra estos parámetros como se muestra en la Fig. 4.15 para seis subtareas de una tarea. Suponiendo que tenemos un conjunto de tareas T que consta de n subtareas t_1, \dots, t_n y un conjunto de procesadores $P = p_1, \dots, p_k$. De los k elementos, este problema se reduce a encontrar la función óptima. $F : T \rightarrow P$. Este proceso se denomina comúnmente asignación o programación estática. Desafortunadamente, el conjunto de soluciones para F crece exponencialmente con un aumento de n y con frecuencia se emplean métodos heurísticos para la asignación estática de tareas. El gráfico en la figura 4.15 b que muestra la asignación de tareas a los procesadores en función del tiempo se denomina diagrama de Gantt diseñado por Henry Gantt en 1910 [5].

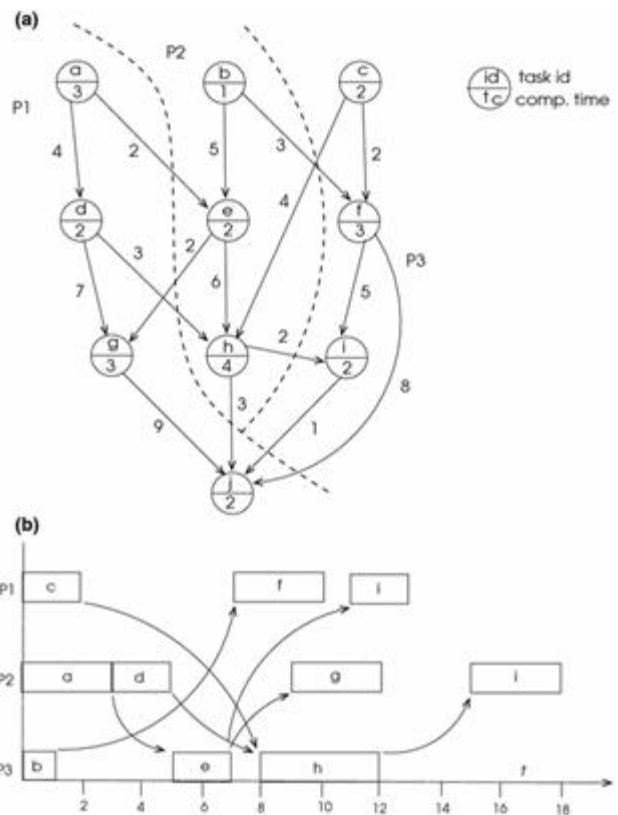


Fig. 4.15 Asignación de tareas estáticas utilizando el gráfico de dependencia de tareas. Tenemos un gráfico de nueve tareas en a y una posible asignación a tres procesadores se muestra en b . La partición aquí intenta colocar tareas que se comunican en gran medida con el mismo procesador al tratar de mantener la carga de trabajo en cada procesador similar

4.9.2 Equilibrio dinámico de carga

En el equilibrio de carga dinámico , asignamos las tareas al procesador durante la ejecución del algoritmo paralelo. Se necesita este método cuando características de la tarea no se conocen en priori . Es posible que tengamos un esquema de equilibrio de carga centralizado en el que un proceso central comúnmente denominado supervisor Gestiona la distribución de la carga. Cada vez que un procesador queda inactivo, se informa al supervisor, que proporciona una nueva subtarea / datos al trabajador inactivo. Los enfoques totalmente distribuidos no tienen procesos de supervisión y el monitoreo de la carga en cada procesador se realiza por trabajadores iguales que intercambian sus estados durante la ejecución del programa paralelo. El proceso sobrecargado puede iniciar la transferencia de trabajo a un proceso ligeramente cargado o la transferencia de trabajo puede iniciarse mediante el proceso de recepción. El primer enfoque se llama balanceo de carga dinámico iniciado por el remitente y este último iniciado por el receptor .

4.10 Programación paralela

La programación paralela implica escribir el código real que se ejecutará en la máquina paralela. Revisaremos la programación paralela con ejemplos en memoria compartida y modelos de memoria distribuida que pasan mensajes. Esta tarea se puede modelar como un paradigma de datos múltiples de programa único (SPMD) en el que todos los procesos ejecutan el mismo código en datos diferentes o un modelo de datos múltiples de programas múltiples (MPMD) con cada proceso ejecutando códigos diferentes en datos diferentes en ambos estos modelos

4.10.1 Programación de memoria compartida con hilos

Los sistemas operativos se basan en el concepto de un proceso que es la unidad básica de código que se debe programar y que contiene datos como registros, pilas y memoria privada. La organización de las funciones principales de un sistema operativo, que son la administración de recursos y la conveniente interfaz de usuario en torno a esta percepción, tiene muchas ventajas. En el sentido más básico, muchos procesos se pueden programar de forma independiente en los sistemas operativos multitarea, lo que evita las esperas innecesarias debido a los dispositivos de entrada / salida lentos, como los discos. Un proceso puede estar en uno de los tres estados básicos en cualquier momento: ejecutándose cuando se está ejecutando, bloqueado cuando no puede ejecutarse debido a la falta de disponibilidad de un recurso, o listo cuando el único recurso que necesita es el procesador. El uso de procesos proporciona el cambio del procesador entre diferentes procesos. El entorno actual del proceso en ejecución, como sus registros, punteros de archivo y datos locales, se almacena y el entorno guardado del nuevo proceso que se ejecuta se restaura en el cambio de contexto . Otro problema encontrado al usar este modelo de computación es la protección de la memoria compartida entre procesos cuando los datos en esta área se deben leer o escribir. El código de un proceso o el sistema operativo que realiza el acceso a la memoria compartida con otros procesos se denomina la sección crítica. Si bien en teoría podemos usar procesos para el procesamiento paralelo en un entorno de memoria compartida, dos dificultades principales son la costosa sobrecarga del cambio de contexto y la protección de los segmentos de memoria compartida contra las operaciones de lectura / escritura simultáneas realizadas por los procesos.

4.10.1.1 Hilos

Los sistemas operativos modernos admiten subprocessos que son procesos ligeros dentro de un proceso. Un hilo tiene un contador de programa, registros, pila y una pequeña memoria local que hace que el cambio de contexto sea al menos un orden menos costoso que los procesos de cambio. Existe un área global del proceso que debe protegerse ya que los subprocessos necesitan acceder a esta área a menudo. Hay dos tipos principales de subprocessos: subprocessos del núcleo y subprocessos de usuario. El núcleo de un sistema operativo conoce un subprocesso del núcleo y, por lo tanto, puede programarse de forma independiente, en contra de los subprocessos de usuario que solo se identifican en el espacio del usuario. Los subprocessos de usuario son administrados por el tiempo de ejecución, lo que resulta en un orden de disminución en su cambio de contexto en comparación con los subprocessos del núcleo. Sin embargo, un hilo de usuario bloqueado en una operación de entrada / salida bloquea todo el proceso, ya que estos hilos no están identificados por el núcleo.

4.10.1.2 Hilos POSIX

La interfaz de sistema operativo portátil (POSIX, por sus siglas en inglés) es un conjunto de estándares que surgieron por necesidad como esfuerzos conjuntos de IEEE e ISO para proporcionar software compatible para diferentes sistemas operativos [[13](#)]. El estándar de subprocessos POSIX es una interfaz de programación de aplicaciones (API) que especifica una serie de rutinas para la administración de subprocessos. Las funciones de administración de subprocessos fundamentales en esta biblioteca se pueden clasificar de la siguiente manera.

- Función de subprocesso : el propio subprocesso se declara como un procedimiento con parámetros de entrada y también los posibles parámetros de retorno que puede invocar el subprocesso principal.
- Creación de subprocessos : esta llamada al sistema crea un subprocesso y comienza a ejecutarlo.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attributes, void *(*start_function)(void *), void *arguments);
```

dónde *thread_id* es la variable el identificador del hilo creado se almacenará después de esta llamada al sistema, ciertas propiedades de un hilo pueden ser inicializadas por la variable de atributos , start_function es la dirección del código del hilo, y los argumentos son las variables que se pasan al hilo creado.

- Esperando la terminación del hilo : el hilo principal espera los hilos que ha creado usando esta función de llamada.

```
int pthread_join(pthread_t thread, void **status);
```

donde subprocesso es el identificador del subprocesso a esperar y el estado se utiliza para el estado de retorno o devolver una variable al subprocesso principal.

- Sincronización de subprocessos : los subprocessos deben sincronizarse para las secciones críticas y también para notificar eventos entre sí mediante el uso de estructuras de datos tales como variables de exclusión mutua , variables de condición y semáforos .

Exclusión mutua

La protección de las variables globales mediante subprocessos POSIX se proporciona mediante variables de exclusión mutua. En el código de ejemplo C utilizando los hilos POSIX continuación, tenemos dos hilos T 1 y T 2, un globales variables compartidas de datos entre ellos, y una variable de exclusión mutua m que es inicializado por el hilo principal, que también activa hilos y finalmente les espera para terminar. Cada hilo se bloquea mantes de ingresar a su sección crítica, lo que evita la interrupción en esta sección. Al salir, desbloquea m para permitir que cualquier otro subprocesso entre en su sección crítica protegida por m . El sistema operativo garantiza que el bloqueo y desbloqueo. Las operaciones en las variables de exclusión mutua se ejecutan atómicamente y, por lo tanto, no se pueden interrumpir.

```
#include <pthread.h>

int data;
pthread_t thread1, thread2;
pthread_mutex_t m;

T1() {
    ...
    pthread_mutex_lock(&m);
    data=data+1;
    pthread_mutex_unlock(&m);
    ...
}

T2() {
    ...
    pthread_mutex_lock(&m);
    data=data*4;
    pthread_mutex_unlock(&m);
    ...
}

main() {
    pthread_mutex_init(&m);
    pthread_create(&thread1,NULL,T1,"void");
    pthread_create(&thread2,NULL,T2,"void");
    ...
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
}
```

Sincronización

Los hilos, como procesos, deben sincronizarse en condiciones. Asumamos dos hilos, uno de los cuales produce algunos datos (productor) y necesita informar a otro hilo de que ha finalizado esta tarea para que el segundo hilo pueda recuperar y procesar estos datos (consumidor). El hilo del consumidor no puede continuar antes de que el primer productor declare la disponibilidad de datos utilizando algún método de señalización. Los semáforos son estructuras de datos que consisten en un entero y comúnmente una cola de proceso asociada con ellos. Los procesos y subprocesos pueden realizar dos acciones atómicas principales en los semáforos: esperar en el que la persona que llama puede esperar o continuar dependiendo de la condición que pretende esperar, y señalar la llamada proporciona una señalización de la finalización del evento esperado y también libera cualquier proceso de espera para ese evento.

El código C que se proporciona a continuación muestra cómo dos subprocesos se sincronizan utilizando dos semáforos sema1 y sema2 . Tenemos un hilo productor que introduce algunos datos y escribe estos datos a la ubicación de memoria compartida de datos . Luego señala al otro usuario del hilo que lee estos datos y los procesa. La sincronización es necesaria para que los datos no leídos no sean sobrescritos por el productor y los datos no sean leídos y procesados más de una vez por el consumidor . El semáforo sema1 se inicializa a valor verdadero en el hilo principal, lo que permite al productor del hilo. eso ejecuta una espera en él para continuar sin esperar en primera instancia, ya que inicialmente no tiene nada que esperar. El segundo semáforo sema2 se inicializa a un valor falso ya que no queremos que el consumidor proceda antes de que el productor indique la disponibilidad de datos para el productor . Los semáforos también se pueden usar para la exclusión mutua, pero se debería preferir el empleo de variables de exclusión mutua para la exclusión mutua para mejorar la legibilidad del programa y también para el desempeño.

```
#include <pthread.h> pthread_t t1, t2;
sem_t sema1,sema2;
int data;

void producer()
{ while(true) {
    wait(sema1);
    input some_data;
    data = some_data;
    signal(sema2);
}
}

void consumer()
{ while(true) {
    wait(sema2);
    my_data = data;
    signal(sema1);
    process my_data;
}
}

main()
{
pthread_create(&t1,NULL,producer,"void");
pthread_create(&t2,NULL,consumer,"void");
sem_init(&sema1,1,0);
sem_init(&sema2,1,0);
...
}
```

Un programa multiproceso con la biblioteca de subprocesos POSIX (lpthread) se puede compilar y vincular de la siguiente manera en el entorno UNIX:

```
cc -o sum sum.c -lpthread
```

Un algoritmo paralelo con hilos POSIX

Usemos subprocesos POSIX para calcular el número PI utilizando el procesamiento paralelo de memoria compartida. Este número puede definirse por varios métodos y una de esas técnicas es tomar la integral de $\frac{4}{(1+x^2)}$ entre 0 y 1 para encontrar una solución aproximada. En nuestro diseño, dividiremos el área bajo esta curva en n número de tiras; calcule el área de cada tira en paralelo utilizando las hebras POSIX y agregue estas

áreas para encontrar toda el área bajo la curva entre 0 y 1, usando el hecho de que la integral de una función entre dos límites es el tamaño del área dentro de estos límites y el eje x . Cada hilo tiene un identificador único de hilo dado por el sistema operativo en tiempo de ejecución; sin embargo, pasamos un entero secuencial en aumento que comienza desde 1 a cada uno para definir el área de la curva en la que funcionará el hilo. La variable de memoria global `total_area` cada subproceso debe actualizarlo y, por lo tanto, debe estar protegido contra accesos concurrentes mediante un bloqueo de exclusión mutua denominado m 1.

```
#include <stdio.h>
#include <pthread.h>

#define n 100
#define n_threads 1024

pthread_t threads[n_threads];
pthread_mutex_t m1;

int total_area;// global variable to store PI

/*****************
thread code to be invoked n_threads times
******/

void *worker(void *id)
{ int me=((int *) id);
  int i, n_slice;
  double width, my_sum;

  width = 1/(double) n_threads;
  x = my_id * width;
  y = 4.0 / (double)(1+(x*x));
  my_area = x * y; // find my_area
  pthread_mutex_lock(&m1); // update total_area
  total_area = total_area + my_area;
  pthread_mutex_unlock(&m1);
}

/*****************
main thread
*****/

main()
{ pthread_t threads[n_threads];
  int i;
  pthread_mutex_init(&m1);
  for(i=1; i<=n_threads; i++)
    pthread_create(&threads[i],NULL,worker,i);
  for(i=1; i<=n_threads; i++)
    pthread_join(threads[i],NULL);
  printf("Approximate PI is:
```

La Figura [4.16](#) muestra la curva PI entre 0 y 1 valores de eje x donde asumimos cinco hilos paralelos para simplificar. La precisión puede mejorarse ya sea aumentando el número de hilos y / o calculando el área como el valor medio de los valores del área del borde como se muestra en la figura.

Los hilos se pueden utilizar convenientemente para el procesamiento en paralelo en los modernos procesadores de múltiples núcleos. Se comunican utilizando la memoria compartida y, por lo tanto, no necesitan enviar mensajes, lo que evita los retrasos en la comunicación de datos a expensas de los gastos generales causados por la protección de la memoria compartida. OpenMP es una plataforma de procesamiento paralelo ampliamente utilizada que utiliza multihilo [[12](#) , [15](#)].

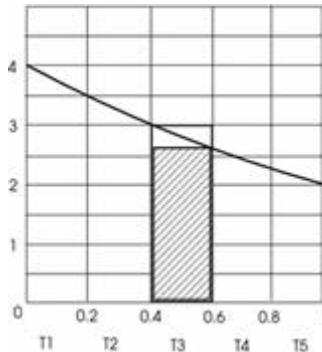


Fig. 4.16 La curva PI entre 0 y 1 valores de eje x . El área debajo de la curva se puede aproximar mejor si cada hilo calcula el promedio de las áreas de dos rectángulos formados por valores de borde como se muestra para T_3 que debería encontrar áreas para $x = 0.4$ y $x = 0.6$ valores y promedios

4.10.2 Programación de memoria distribuida con MPI

El estándar de interfaz de paso de mensajes (MPI) especifica una biblioteca de rutinas de paso de mensajes para proporcionar un método portátil, flexible y eficiente para escribir programas de paso de mensajes [10 , 11]. Aunque está dirigido principalmente a programas de memoria distribuida, los desarrollos posteriores y las versiones de MPI proporcionan memoria distribuida, memoria compartida o implementaciones híbridas. La máquina virtual paralela (PVM) es otra herramienta ampliamente utilizada para el procesamiento en paralelo [14]. MPI consiste en rutinas para enviar y recibir datos entre procesos y varios otros modos de comunicación como la difusión de un mensaje a todos los procesos en el sistema o la multidifusión en la que se envía un mensaje a un grupo de mensajes. Los programas MPI comienzan al inicializar el entorno, realizando cálculos paralelos enviando y recibiendo mensajes entre los procesos y luego terminando. Para definir el conjunto de procesos que se ejecutarán, se emplean objetos llamados comunicadores . Cada proceso en MPI pertenece a un comunicador y dentro de un comunicador, cada proceso tiene un identificador único llamado su rango . Las siguientes rutinas de C se utilizan para inicializar el entorno de computación en paralelo y luego terminar.

- MPI_Init (int * argc , char ** argv): ingresa un puntero al número de argumentos y un puntero al vector de argumentos. Estos parámetros se pasan desde la línea de comandos para especificar el número de procesos a invocar.
- Int MPI_Comm_size (MPI_Comm comm , int * size): El número de procesos paralelos en la comunicación del comunicador se devuelve en tamaño en la comunicación del comunicador .
- Int MPI_Comm_rank (MPI_Comm comm , int * rank): devuelve el rango de un proceso en el grupo comm . Los rangos se ordenan 0, ..., tamaño -1.
- Int MPI_Finalize (void): Cada proceso llama a esta rutina antes de salir para limpiar la biblioteca y terminar.

Los procedimientos principales para la transferencia de datos son las rutinas de envío y recepción con muchas variaciones. El envío de bloqueo y la recepción de bloqueo se especifican a continuación:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
            int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
            int source, int tag, MPI_Comm comm, MPI_Status *status)
```

donde buf es la dirección del búfer de envío / recepción, cuenta el número de elementos en el búfer, el tipo de datos es el tipo de datos de cada elemento del búfer de envío / recepción, dest / source es el rango entero de destino / fuente, la etiqueta es el tipo de mensaje, el comunicador es el comunicador y el estado es el objeto de estado que se examinará. Tenga en cuenta que dos procesos pueden usar la etiqueta de mensaje para realizar diferentes acciones por etiquetas diferentes. Ahora podemos escribir una aplicación MPI simple de dos procesos enviando y recibiendo mensajes en lenguaje de programación C de la siguiente manera:

```
#include <mpi.h>

int main(int argc, char** argv) {
    int my_rank, size, data;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (my_rank == 0) { // rank 0 is the sender
        data = 23;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("This is process
    } else if (my_rank == 1) { // rank 1 is the receiver
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("This is process
    }
    MPI_Finalize();
}
```

El mismo código se ejecuta en todos los procesadores en este modelo SPMD, que es muy común en las aplicaciones MPI debido a la dificultad de escribir diferentes códigos para diferentes procesos. Las instrucciones que debe ejecutar cada proceso están separadas por el uso de identificadores de proceso. En este ejemplo, el rango 0 es el remitente y el rango 1 es el receptor. Necesitamos compilar y ejecutar este código llamado mess.c en el entorno UNIX de la siguiente manera:

```
mpicc -o mess mess.c
mpirun -np 8 mess
```

La primera línea es el comando de compilación y vinculación que usa el compilador / vinculador mpicc y la segunda línea comienza a ejecutar el programa ejecutable con ocho procesos. Tenga en cuenta que pasamos este argumento de ocho procesos al programa principal en el que MPI_Initutiliza para inicializar el entorno y comienza a ejecutar estos procesos idénticos en un entorno de hardware que no conocemos. De hecho, podríamos haber instalado MPI en una sola computadora y ocho procesos podrían ejecutarse en la misma computadora. El siguiente ejemplo muestra el uso de MPI para calcular PI utilizando el mismo método para encontrar el área bajo la curva $\frac{4}{1+x^2}$. Entre $x=0$ y $x=1$. Como hicimos con los hilos POSIX.

```

#include <mpi.h>
#include <math.h>

int main(int argc,char **argv)

{
    int i, n, my_id, n_slices, n_procs, n_sl;
    double my_area, total_area, width, x, my_sum;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&n_procs);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

    if (myid == 0) {
        printf("Enter the number of slices for each process:");
        scanf("%d",&n_slices);
        MPI_Bcast(&n_slices, 1, MPI_INT, 0, MPI_COMM_WORLD);
        for(i= 1; i < n_procs; i++) {
            MPI_Recv(&part_area, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                     1, MPI_COMM_WORLD, &status);
            total_area += part_area;
        }
        printf("Approximate PI is: %f", total_area);
    }
    else {
        MPI_Recv(&n_sl, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        my_sun = 0.0;
        width = 1.0 / (double) (n_sl + 1);
        for (i = my_id + 1; i <= n_sl; i++) {
            x = width * (double)i;
            my_area += 4.0 / (1.0 + x*x);
        }
        MPI_Send(&my_area, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}

```

Tenga en cuenta que este es un modelo de supervisor / trabajador de procesamiento de mensajes que pasa paralelo que utiliza el paradigma SPMD. El proceso del supervisor, a veces llamado raíz , tiene un rango de 0 y comienza por la inicialización como todos los demás procesos seguidos de pedir al usuario el número de cortes que debe procesar cada proceso bajo la curva. Luego transmite este valor a todos los procesos. Cada proceso luego calcula el área debajo de su porción de la curva para el número de rebanadas y envía esta área parcial al supervisor que las agrega y genera. Podríamos tener al supervisor también involucrado en el cálculo de PI (Ver Ejercicio 6).

4.11 conclusiones

Hemos revisado los conceptos fundamentales de computación en paralelo en este capítulo con énfasis en los métodos de diseño para algoritmos de grafos paralelos. Los algoritmos paralelos pueden usar la memoria compartida o el modelo de paso de mensajes en un sentido general. El modelo PRAM es un método idealista para diseñar algoritmos paralelos en la plataforma de memoria compartida; además, proporciona un modelo abstracto que oculta los detalles de la implementación y, por lo tanto, se puede utilizar para el diseño de alto nivel y la comparación de algoritmos paralelos de memoria compartida. El modo de acceso a la memoria compartida es importante en este modelo, y las lecturas y escrituras se pueden realizar en modos concurrentes o exclusivos. El modelo de paso de mensajes es adecuado para procesadores de memoria distribuidos que se comunican y sincronizan enviando y recibiendo mensajes únicamente. Los modos de comunicación básicos en un sistema de computación paralelo pueden clasificarse según el origen y el destino de la transferencia de datos. Agrupando las comunicaciones bajo operaciones tales comoLos modos uno a todos o todos a todos alivian la carga de escribir un algoritmo paralelo, ya que simplemente podemos especificar el modo necesario en lugar de escribir

el algoritmo real para las comunicaciones. Los métodos de diseño para algoritmos paralelos pueden clasificarse ampliamente como datos o descomposición funcional. Los datos se descomponen en varios conjuntos para ser procesados por procesadores paralelos que utilizan el mismo algoritmo en la primera y diferentes tareas se asignan a diferentes procesadores en el segundo método.

Vimos que los gráficos requieren métodos especiales de computación en paralelo y la contracción de gráficos es un método comúnmente utilizado para permitir operaciones de gráficos en paralelo. La gráfica en consideración se hace más pequeña en cada paso, utilizando métodos como la contracción del borde o la estrella. La contracción del gráfico se puede realizar en paralelo y también la resolución del problema en un gráfico más pequeño se puede hacer de manera más conveniente. Algunos problemas de gráficos se pueden resolver de manera eficiente utilizando algoritmos secuenciales; sin embargo, los mismos problemas no tienen soluciones algorítmicas paralelas simples debido a las dependencias involucradas. Los métodos de aleatorización y ruptura de simetría proporcionan algoritmos de gráficos paralelos simples y elegantes para diversos problemas de gráficos, tales como conjuntos independientes máximos y árboles de expansión mínima. La partición de datos para los gráficos a menudo implica dividir la matriz de adyacencia por filas, por columnas, o en bloque a varios procesadores. Las relaciones de tareas paralelas se pueden representar mediante un gráfico de dependencia de tareas y la asignación de estas tareas a los procesadores es una variación del problema de partición de gráficos. Este enfoque requiere que los tiempos de cálculo de tareas y su interacción se conozcan por adelantado, lo que puede no ser realista en muchas aplicaciones de la vida real. El equilibrio de carga dinámico implica mantener las cargas en los procesos incluso en tiempo de ejecución. Finalmente, revisamos dos plataformas de uso común para la memoria compartida y la programación de memoria distribuida: los subprocesos POSIX proporcionan una API conveniente para implementar algoritmos paralelos de memoria compartida y MPI se usa ampliamente para la programación de memoria distribuida. Las relaciones de tareas paralelas se pueden representar mediante un gráfico de dependencia de tareas y la asignación de estas tareas a los procesadores es una variación del problema de partición de gráficos. Este enfoque requiere que los tiempos de cálculo de tareas y su interacción se conozcan por adelantado, lo que puede no ser realista en muchas aplicaciones de la vida real. El equilibrio de carga dinámico implica mantener las cargas en los procesos incluso en tiempo de ejecución. Finalmente, revisamos dos plataformas de uso común para la memoria compartida y la programación de memoria distribuida: los subprocesos POSIX proporcionan una API conveniente para implementar algoritmos paralelos de memoria compartida y MPI se usa ampliamente para la programación de memoria distribuida. Las relaciones de tareas paralelas se pueden representar mediante un gráfico de dependencia de tareas y la asignación de estas tareas a los procesadores es una variación del problema de partición de gráficos. Este enfoque requiere que los tiempos de cálculo de tareas y su interacción se conozcan por adelantado, lo que puede no ser realista en muchas aplicaciones de la vida real. El equilibrio de carga dinámico implica mantener las cargas en los procesos incluso en tiempo de ejecución. Finalmente, revisamos dos plataformas de uso común para la memoria compartida y la programación de memoria distribuida: los subprocesos POSIX proporcionan una API conveniente para implementar algoritmos paralelos de memoria compartida y MPI se usa ampliamente para la programación de memoria distribuida. El

equilibrio de carga dinámico implica mantener las cargas en los procesos incluso en tiempo de ejecución. Finalmente, revisamos dos plataformas de uso común para la memoria compartida y la programación de memoria distribuida: los subprocesos POSIX proporcionan una API conveniente para implementar algoritmos paralelos de memoria compartida y MPI se usa ampliamente para la programación de memoria distribuida. El equilibrio de carga dinámico implica mantener las cargas en los procesos incluso en tiempo de ejecución. Finalmente, revisamos dos plataformas de uso común para la memoria compartida y la programación de memoria distribuida: los subprocesos POSIX proporcionan una API conveniente para implementar algoritmos paralelos de memoria compartida y MPI se usa ampliamente para la programación de memoria distribuida.

En un nivel más abstracto, podemos ver el modelado de todo el proceso de computación paralela en cuatro niveles relacionados: hardware, sistema operativo, programación y niveles algorítmicos como se muestra en la Fig. 4.17. En todas estas capas de diseño, la distinción principal es si se utiliza la memoria compartida o distribuida. Encontramos que el sistema operativo y el middleware deberían proporcionar diferentes servicios en estos niveles. El principal problema con el enfoque de la memoria compartida es la protección de la memoria durante accesos concurrentes, y esto lo proporcionan las estructuras del sistema operativo, como los semáforos y los bloqueos. En el nivel de programación, los hilos que son procesos ligeros se utilizan ampliamente para la programación de memoria compartida. La biblioteca de subprocesos POSIX proporciona todas las rutinas necesarias para la sincronización de subprocesos y la exclusión mutua. El estándar MPI se usa ampliamente para la computación paralela de memoria distribuida con una amplia gama de procedimientos de paso de mensajes requeridos. A nivel algorítmico, el modelo PRAM que asume que la memoria compartida no es práctico, ya que asume una memoria compartida infinitamente grande y un número infinito de procesadores; sin embargo, se usa para comparar varios algoritmos paralelos para el mismo problema. Consideraremos principalmente plataformas de memoria distribuida en nuestro análisis de algoritmos de gráficos paralelos en este libro para proporcionar soluciones implementables, excepto en algunos lugares donde describimos los algoritmos de PRAM.

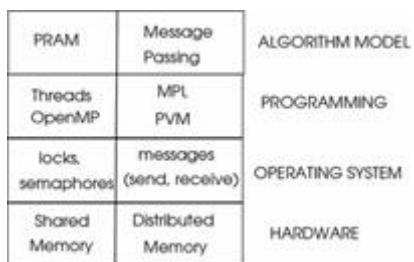


Fig. 4.17 La jerarquía de modelos de procesamiento paralelo de memoria compartida y distribuida

Ceremonias

Escriba el pseudocódigo de un algoritmo EREW PRAM que usa los procesadores p para encontrar la suma de los valores de p en $\tilde{O}(\log p)$ hora.

- 1.
2. La suma de prefijos de una matriz $A[1..n]$ es una matriz $B[1..n]$ tal que $B[j] = \sum_{i=1}^j A[i]$. Diseñe un algoritmo paralelo de EREW PRAM para encontrar la suma de prefijos de una matriz de entrada.
3. Escriba el pseudocódigo de la rutina de comunicación de difusión uno a todos en un hipercubo de dimensión d . Esta rutina ingresa la dimensión d y el identificador i del nodo y el mensaje. Suponga

que el nodo 0 es la fuente del mensaje y primero se transmite en la dirección x (1), luego en la dirección y , y finalmente en la dirección z (3) como se muestra en la figura 4.18 . Calcula el número de pasos y el trabajo realizado.

4. Necesitamos dividir el gráfico de tareas de la figura 4.19 en dos procesadores paralelos. Queremos colocar procesos que tengan una comunicación significativa con el mismo procesador para mantener las comunicaciones entre procesos lo más pequeñas posible y también queremos mantener la carga total (tiempo de ejecución de las tareas) en cada procesador similar. El método a emplear es encontrar una coincidencia de la gráfica de manera que se seleccionen los bordes de separación más pesados en cada iteración. Luego, los vértices en cada extremo de los bordes seleccionados se contraen para obtener supervertices . Usa la contracción gráfica hasta solo dos supervertices permanecen cada uno de los cuales se pueden asignar a un solo procesador. Calcule el tiempo total de ejecución paralela dibujando un diagrama de Gantt de la programación de los procesos. Entrene también el costo total de la comunicación entre procesos, ignorando los costos de comunicación de las tareas asignadas al mismo procesador.
5. Escriba un programa en C utilizando la API de subprocesos POSIX que encuentre la suma de una matriz de tamaño n almacenada en la memoria global. Use el paralelismo de datos para descomponer los datos por igual a k hilos. La suma total se mantiene en la memoria global y debe protegerse contra accesos concurrentes.
6. Modifique el código C para el cálculo de PI usando MPI de la Sección. 4.10.2 para que el supervisor también calcule el área bajo su porción de curva.
7. Ocho procesadores con etiquetas en orden creciente están conectados en una estructura de anillo unidireccional como se muestra en la Fig. 4.20 . Cada proceso tiene un valor entero y necesitamos sumar estos valores y generar la suma por proceso p_0 . Escribe un programa MPI en lenguaje C que comienza por p_0 enviando su valor a su próximo vecino, que agrega el valor que recibe con el valor que tiene y envía la suma a su próximo vecino. Las sumas parciales se propagan de esta manera hasta p_0 recibe la suma total y la genera como se muestra en la figura. Modificar este programa de tal manera que p_0 tiene una matriz de enteros de tamaño n y distribuye las porciones de esta matriz a los procesos que utilizan la descomposición de los datos que se realizan como antes para encontrar la suma total de la matriz.
8. Se diseñará un servidor de archivos multiproceso que reciba un mensaje de parte del hilo frontal, y este hilo invoca uno de los hilos de apertura ,lectura y escritura según la acción requerida en el mensaje entrante. El hilo de lectura lee el número de bytes del archivo que se envía al remitente, el hilo de escritura escribe el número especificado de bytes contenidos en el mensaje en el archivo especificado. Escribe este programa en C con hilos POSIX.

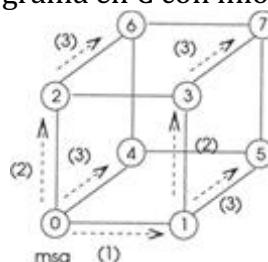


Fig. 4.18 Comunicación hipercubo uno a todos.

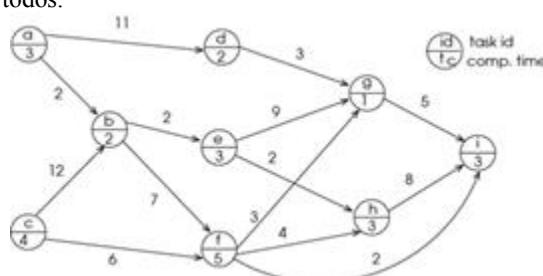


Fig. 4.19 Gráfico de dependencia de tareas para el Ejercicio 4

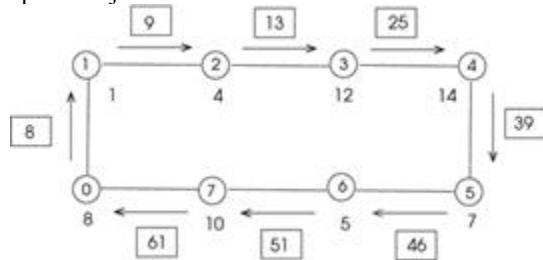


Fig. 4.20 Un anillo unidireccional de ocho procesos utilizado para calcular la suma de los enteros almacenados en cada nodo. Los enteros almacenados se muestran junto a los nodos y los mensajes contienen las sumas parciales transferidas

Referencias

1. Belloch G (2015) Diseño de algoritmo: paralelo y secuencial, libro borrador
2. http://www.nvidia.com/object/cuda_home_new.html
3. Flynn MJ (1972) Algunas organizaciones de computadoras y su efectividad. IEEE Trans. Comput . C21 (9): 948960
[MATES](#)
4. Foster I (1995) Diseño y creación de programas paralelos: conceptos y herramientas para la ingeniería de software en paralelo. Addison-Wesley, Boston
[MATES](#)
5. Gantt HL (1910) Trabajo, salarios y ganancias, La revista de ingeniería, Nueva York
6. Grama A, Karypis G, Kumar V, Gupta A (2003) Introducción a la computación paralela, 2^a ed . Addison-Wesley, Boston
[MATES](#)
7. Karypis G, Kumar V (1995) Esquema de partición k-way multnivel para gráficos irregulares. Informe técnico TR 95-064, Departamento de Ciencias de la Computación, Universidad de Minnesota
8. Karypis G, Kumar V (1997) Una formulación paralela de grano grueso de algoritmo de partición de gráficos de k-way multnivel. En: 8^a conferencia SIAM sobre procesamiento paralelo para computación científica.
9. Kernighan BW, Lin S (1970) Un procedimiento heurístico eficiente para particionar gráficos. Bell Syst Tech J 49: 291307
[Referencia cruzada](#)
10. <http://www.mcs.anl.gov/research/projects/mpi/>
11. <http://www.open-mpi.org/>
12. <http://openmp.org/wp/>
13. POSIX.1 FAQ. El grupo abierto, 5 oct 2011.
14. <http://www.csm.ornl.gov/pvm/>
15. Quinn M (2003) Programación paralela en C con MPI y OpenMP , 1^a ed . McGraw-Hill Ciencia / Ingeniería / Matemáticas, Nueva York

5. Algoritmos de grafos distribuidos.

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Un sistema distribuido consiste en una serie de nodos computacionales conectados por una red de comunicación. Los nodos de un sistema distribuido cooperan y se comunican para lograr un objetivo común. Primero describimos el tipo de sistemas distribuidos, los métodos de comunicación y sincronización utilizados en estos sistemas. Luego investigamos algunos algoritmos distribuidos fundamentales que incluyen la construcción de un árbol de expansión, las operaciones de transmisión y convergencia sobre un árbol de expansión y la elección del líder.

5.1 Introducción

Un sistema informático distribuido o un sistema distribuido, como se denomina más comúnmente, consiste en una serie de nodos computacionales conectados por una red de comunicación. Los nodos de computación son autónomos y la red puede ser un medio cableado; Un canal de comunicación inalámbrica o ambos. Los nodos de un sistema distribuido cooperan y se comunican para lograr un objetivo común. Es evidente que la sincronización entre los cálculos en los nodos de dicho sistema es necesaria para proporcionar esta coordinación.

Un sistema distribuido aparece a los usuarios como un único sistema informático. En ese sentido, una nube es un sistema distribuido, ya que existen numerosos elementos informáticos y bases de datos en una nube, sin embargo, aparece como un sistema único para un usuario. Los sistemas distribuidos son necesarios porque proporcionan un acceso conveniente a los recursos remotos para los usuarios y las aplicaciones. En muchos casos, la aplicación en sí se distribuye de forma inherente. Por ejemplo, muchos usuarios utilizan un sistema de reservas de líneas aéreas y proporciona toda la comunicación y sincronización necesarias. Los sistemas distribuidos proporcionan tolerancia a fallos, en cuyo caso la falla de un nodo o un enlace no perjudica el funcionamiento del sistema, ya que estos son reemplazados por otros nodos o enlaces. Los sistemas distribuidos suelen ser dinámicos en los que los nodos y enlaces pueden insertarse o eliminarse de la red debido a fallas o movimientos de los nodos como en el caso de una red móvil. Una operación de rescate que consiste en mover nodos es un ejemplo de una red móvil.

Un sistema distribuido se puede modelar convenientemente mediante un gráfico en el que los vértices del gráfico representan los nodos computacionales y un borde entre dos

nodos representa una facilidad de comunicación entre ellos. Los algoritmos que se ejecutan en los nodos de un gráfico que representa el sistema distribuido se denominan comúnmente gráficos distribuidos o algoritmos de red. Tenga en cuenta que los algoritmos de uso de memoria distribuida en un entorno de procesamiento paralelo también se denominan algoritmos distribuidos en la literatura, pero en el contexto de este libro, usaremos algoritmos distribuidos (gráfico) para indicar los algoritmos que se ejecutan en una red representada por un gráfico. Veremos que diseñar una versión distribuida de un algoritmo de grafo secuencial no es una tarea trivial. Comenzamos este capítulo describiendo las plataformas de sistemas distribuidos comunes. Luego investigamos los algoritmos de gráficos distribuidos, los clasificamos y mostramos el funcionamiento de algunos algoritmos de gráficos distribuidos básicos.

5.2 Tipos de sistemas distribuidos

Las aplicaciones de sistemas distribuidos varían de grupos de computadoras a redes de sistemas integrados. Podemos clasificar los sistemas distribuidos como sistemas informáticos distribuidos, sistemas de información distribuidos y sistemas generalizados distribuidos [5]. Los sistemas de computación distribuida consisten típicamente en un grupo de computadoras homogéneas conectadas por una red de área local. The Grid es también un sistema informático distribuido, que consta de numerosos sistemas informáticos heterogéneos con muchos usuarios diferentes que cooperan para lograr un objetivo común [3]. La computación en la nube es más general que la computación en cuadrícula y proporciona a los usuarios diversos recursos como almacenamiento, administración de datos, hospedaje de sitios web y computación [4]. La tolerancia a fallos debida a la falla de los nodos y enlaces, y el equilibrio de carga son los principales problemas que deben manejarse tanto en Grid como en la nube.

Los sistemas de información distribuidos comúnmente involucran grandes aplicaciones de bases de datos, como un sistema de procesamiento de transacciones. Un sistema bancario en línea con millones de usuarios es un ejemplo de tal sistema. Los sistemas generalizados distribuidos consisten típicamente en computadoras pequeñas y, a veces, móviles que se comunican mediante un medio inalámbrico. Examinaremos estos sistemas más de cerca, ya que, a diferencia de Grid o de una nube, estos se pueden modelar convenientemente mediante un gráfico. Internet es la red más grande del mundo que conecta redes personales, de infraestructura , inalámbricas o de cualquier otro tipo.

Las redes inalámbricas se comunican mediante comunicación inalámbrica y medios de red. Se pueden clasificar en términos generales como infraestructuras y ad hoc . Una red inalámbrica con infraestructura tiene una red troncal cableada fija que consta de enruteadores y puntos de acceso para proporcionar comunicación entre los hosts de la red, como una red celular. En contraste, las redes inalámbricas ad hoc no tienen esta estructura y cada nodo en dicha red actúa como un enruteador para la transferencia de mensajes. Las redes ad hoc se utilizan ampliamente debido a la facilidad y la velocidad en su implementación. Dos tipos de redes inalámbricas han ganado importancia recientemente; Redes móviles ad hoc y redes de sensores inalámbricos .

5.2.1 Redes móviles ad hoc

Una red móvil ad hoc (MANET) es una red inalámbrica sin infraestructura que consiste en nodos que se mueven dinámicamente. Las redes ad hoc para vehículos (VANET) que proporcionan comunicación entre vehículos en movimiento, los MANET militares

utilizados por los militares y los MANET utilizados en operaciones de rescate son ejemplos de tales sistemas. Cada nodo en un MANET actúa como un enrutador para comunicaciones de múltiples saltos entre hosts en los que un mensaje se transfiere entre varios pares de hosts antes de que llegue a su destino.

Uno de los principales desafíos en un MANET es el enrutamiento , que es el proceso de transferencia de un mensaje entre un remitente y un receptor de la manera más eficiente. Los nodos son móviles, lo que significa que las rutas deben calcularse dinámicamente y requieren algoritmos de enrutamiento eficientes. Mantenerse conectado en un MANET también es otro problema que debe resolverse. Una red de robots es otro ejemplo de MANET y es necesario mantener la red conectada en todo momento para las operaciones coordinadas de robots en dicha red.

5.2.2 Sensores Inalámbricas Redes

Una red de sensores inalámbricos (WSN) consiste en una red de sensores con transceptores y controladores de radio. Estas redes de nodos físicamente pequeños en la mayoría de los casos, tienen muchas aplicaciones que incluyen control ambiental, salud electrónica y edificios inteligentes. Un nodo sensor tiene una potencia muy limitada y los sensores normalmente están controlados por un nodo central llamado sumiderocon más capacidades computacionales. Los datos registrados por los nodos sensores se recopilan en el sumidero para su posterior procesamiento. El enrutamiento de los mensajes de datos al receptor de manera eficiente utilizando los protocolos de red y manteniendo la red conectada son los principales problemas que deben abordarse en las WSN. Las redes de sensores son en su mayoría estacionarias y requieren una operación de baja potencia, lo que es más crítico que la administración de la energía en los MANET.

Un MANET o un WSN se pueden modelar convenientemente mediante un gráfico y los problemas como el enrutamiento, la conectividad se pueden transferir al dominio del gráfico para resolverlos con métodos desarrollados para gráficos. Por ejemplo, un problema de enrutamiento eficiente se puede resolver con la ayuda del método de encontrar la distancia más corta entre dos nodos de un gráfico ponderado. Sin embargo, estos problemas ahora deben resolverse de manera distribuida sin ningún conocimiento global, lo que hace que el problema sea más difícil que un problema de grafo ordinario. Un nodo en un gráfico que representa una WSN solo puede comunicarse con sus vecinos, pero necesitamos tener una decisión global utilizando los datos recopilados de todos los sensores. La Figura 5.1 muestra una red inalámbrica con nodos que pueden transmitir y recibir señales de radio dentro de un radio de rmetros Luego podemos conectar los nodos que están dentro de los rangos de transmisión entre sí por un borde y obtener el gráfico que se muestra.

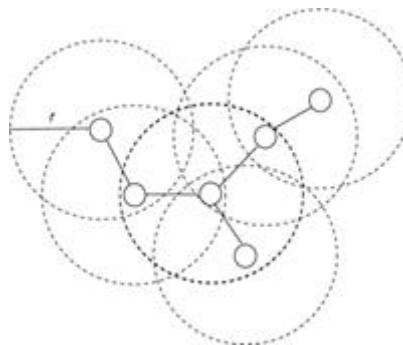


Fig. 5.1 La representación gráfica de una red inalámbrica. El rango de transmisión de un nodo se muestra mediante círculos discontinuos centrados en ese nodo

5.3 modelos

Los mensajes son cruciales para el correcto funcionamiento de un algoritmo distribuido. Podemos definir formalmente el modelo de paso de mensajes ampliamente aceptado de un sistema distribuido de la siguiente manera [1 , 6]:

- Un proceso p_i en un nodo i se comunica con otros procesos intercambiando mensajes solamente.
- Cada proceso p_i tiene un estado $s_i \in S$, donde S es el conjunto de todos los estados posibles que un proceso p_i puede ser.
- La configuración de un sistema consiste en un vector de estados como $C = [s_1, \dots, s_n]$
- La configuración de un sistema puede cambiar ya sea por un evento de entrega de mensajes o un evento de cálculo .
- Un sistema distribuido pasa continuamente por ejecuciones como $C_0, \phi_1, C_1, \phi_2, \dots$ donde ϕ es un evento de cómputo o de entrega de mensaje.

Una máquina de estado finito (FSM) o un autómata de estado finito es un modelo matemático para representar un sistema complejo. Un FSM consiste en estados, entradas y salidas. Puede cambiar su estado según su estado actual y la entrada que recibe. Los FSM se utilizan ampliamente para diseñar algoritmos, protocolos de red y análisis de secuencias en bioinformática. Formalmente, un FSM determinista es un quintuple (I, S, S_0, δ, F) donde

- I es un conjunto de señales de entrada.
- S es un conjunto finito de estados no vacíos.
- $S_0 \in S$ Es el estado de inicio inicial.
- $\delta : S \times I \rightarrow S$ Es la función de transición de estado tal que
- $O \in S$ Es el conjunto de estados de salida.

El siguiente estado de un FSM está determinado por su estado actual y la entrada que recibe. La misma entrada puede causar diferentes acciones en diferentes estados. Como ejemplo cotidiano, consideremos a los estudiantes de una escuela que, por simplicidad, pueden tener solo dos estados: `in_class` o `out_class`, lo que significa que pueden estar en la clase o fuera de ella. Cuando la campana suena en estado `in_class` , significa que pueden salir y la campana que suena en estado `out_class` significa que deben ir en la clase. Un diagrama FSM o un diagrama de transición de estadoEs una ayuda visual para entender el comportamiento de un FSM. Los círculos en dicho diagrama denotan sus estados y las transiciones entre estados se muestran mediante arcos dirigidos que están etiquetados como a / b , donde a es el conjunto de entradas recibidas y b es el conjunto de salidas producidas cuando se reciben estas entradas. Un doble círculo denota el estado de aceptación.

Una tabla de estado proporciona una forma alternativa de representar un FSM. Tiene estados del FSM como filas y entradas como columnas y los elementos de la tabla pueden ser el siguiente estado del FSM y las acciones que se deben tomar cuando se recibe la entrada. La salida de un tipo de máquina Moore de FSM es el siguiente estado, mientras que la salida en un tipo de máquina Mealy de FSM contiene salidas, así como el estado siguiente.

Ejemplo 5.1 Diseñaremos un FSM simple para un elevador que solo puede ir a los pisos 0, 1 y 2. Hay dos botones en el elevador: arriba y abajo que suben y bajan el elevador respectivamente. Podemos asociar el estado actual del ascensor con el piso que se encuentra actualmente; por lo tanto, tenemos tres estados 0, 1 y 2. En cada estado, el botón arriba o abajo se puede presionar representado por dos entradas arriba por 0 y abajo por 1. El diagrama de FSM para este ejemplo se muestra en la Fig. 5.2 que muestra todas las transiciones de estado, considerando que habrá dos entradas en cada estado. No podemos bajar desde el estado 0 y tampoco se permite mostrar subidas desde el segundo piso mediante bucles en estos estados.

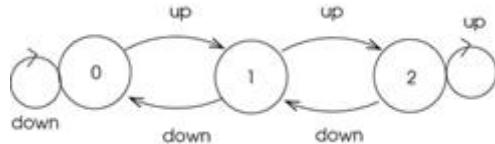


Fig. 5.2 FSM del ejemplo del elevador.

Ahora podemos formar la tabla de estado para este FSM con entradas que muestran el siguiente estado del FSM cuando la entrada que se muestra en columnas se recibe en el estado que se muestra en filas como se muestra en la Tabla 5.1. Esta forma de expresar un FSM proporciona una manera muy conveniente de escribir su algoritmo. Podemos formar una matriz 2-D con cada elemento como un puntero de función. Luego definimos las funciones a realizar para cada entrada de la tabla; por ejemplo, recibir el estado “0” (arriba) en el estado “1” (primer piso) debe causar una transición al estado 2 (el elevador debe moverse al segundo piso), lo cual se realiza al cambiar el estado actual a “2”. La ejecución del algoritmo es entonces sencilla; cada vez que se recibe una entrada, activamos la función mostrada por la entrada de la tabla FSM como se muestra en el código de lenguaje de programación C a continuación.

Tabla 5.1 Tabla de estado del elevador

Estado 0 (arriba) 1 (abajo)

0	1	0
1	2	0
2	2	1

```

#include <stdio.h>
#define UP    0
#define DOWN  1

void *fsm_tab[3][2]();
int input;

void act00(){curr_state=1;}
void act01(){curr_state=0;}
void act10(){curr_state=2;}
void act11(){curr_state=0;}
void act20(){curr_state=2;}
void act21(){curr_state=1;}


main()
{
    curr_state=0;           // initialize curr_state
    fsm_tab[0][0]=act00;   // initialize FSM table
    fsm_tab[0][1]=act01;
    fsm_tab[0][2]=act02;
    fsm_tab[1][0]=act10;
    fsm_tab[1][1]=act11;
    fsm_tab[1][2]=act12;

    while (true)
    {
        printf("Type 0 for up, and 1 for down \n");
        scanf("%d", &input);
        *fsm_tab[curr_state]{input};
        printf("now at floor %d", curr_state)
    }
}

```

5.4 Comunicación y sincronización.

Los algoritmos que se ejecutan en los nodos de un sistema distribuido deben sincronizarse para lograr un objetivo común. Este proceso se puede realizar en varios niveles. Veamos cómo se puede manejar la sincronización localmente en tres niveles principales de jerarquía; El hardware, el sistema operativo y la aplicación. En el nivel más bajo, el hardware puede proporcionar sincronización en un cierto número de tics de reloj periódicamente. En un nivel superior, una de las tareas principales de los sistemas operativos locales en cada nodo es la sincronización de los procesos que residen en ese nodo. Además, esta función se puede extender a los procesos que se ejecutan en los nodos del sistema distribuido en el nivel de la aplicación.

Sin embargo, necesitamos un mecanismo para proporcionar sincronización entre los nodos que se debe traducir a los mecanismos de sincronización local descritos anteriormente. Un método muy utilizado en un sistema distribuido es la sincronización a través de mensajes. En este llamado modelo de paso de mensajes , cada sistema operativo local o middleware proporciona dos primitivas básicas; Enviar y recibir para enviar y recibir mensajes. Estos procedimientos pueden ser ejecutados en forma de bloqueo o no bloqueo . Un envío bloqueado detiene a la persona que llama hasta que se recibe un acuse de recibo del receptor. Una recepción de bloqueo significa que el receptor debe esperar hasta que se reciba un mensaje. El bloqueorecibir tal vez selectiva en el que un mensaje de un remitente en particular se esperó y la ejecución se reanuda sólo después de esto sucede. Es una práctica común emplear un envío sin bloqueo con una recepción de bloqueo, ya que se supone que el mensaje enviado se entrega correctamente, mientras que las acciones de un receptor dependen de si el mensaje se recibe y también de su contenido y, por lo tanto, se utiliza frecuentemente una recepción de bloqueo .

El envío y la recepción se suelen emplear indirectamente mediante el uso de estructuras de datos denominadas puertos o buzones . Estos son lugares de depósito para mensajes, y la colocación o eliminación de mensajes se puede realizar de forma asíncrona desde estas estructuras. En un sistema distribuido, el procedimiento de envío ejecutado localmente generalmente deposita el mensaje en el buzón del proceso de red que agrega

los encabezados de red necesarios y transfiere el mensaje a la red a través del software de la capa de red inferior. El proceso de la red receptora elimina los encabezados de la red y deposita el mensaje en el buzón del receptor, que lo toma desde allí como se muestra en forma simplificada en la Fig. 5.3. Hay tres módulos de software principales en cada nodo de un sistema distribuido: sistema (OS), pila de protocolos de red (N / W) y la aplicación (APP) como se muestra en esta figura.

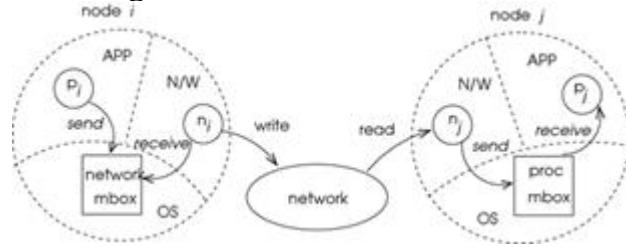


Fig. 5.3 Comunicación distribuida a través de buzones. Proceso p_i en el nodo i envía un mensaje para procesar p_j en el nodo j usando buzones a través de procesos de red n_i y n_j

En resumen, el sistema operativo y los procesos de red proporcionan sincronía entre dos procesos p_i y p_j que ejecutan algoritmos distribuidos en los nodos i y j de la red. En un nivel más alto de abstracción en la aplicación, la sincronización entre algoritmos distribuidos en diferentes nodos puede lograrse mediante rondas que se ejecutan de forma escalonada. En este caso, un proceso especial en un nodo comienza cada ronda y cada proceso espera a que todos los demás procesos finalicen la ejecución antes de que comience la siguiente ronda. La sincronización al comienzo de una ronda se puede lograr mediante la transmisión de un mensaje especial y el final de una ronda se puede identificar por la convergencia de mensajes, que son dos operaciones de comunicación básicas, como veremos en breve. Comúnmente un proceso p_i de un sistema distribuido realiza los siguientes pasos en cada ronda:

enviar mensaje

- 1.
2. recibir mensaje
3. hacer algunos cálculos.

Suponemos aquí que un proceso envía los resultados de su cálculo desde la ronda k - 1 en la ronda k th. Sin embargo, este orden no es estricto, podríamos tener una secuencia de procesamiento , envío y recepción , lo que significaría que cada proceso ahora computa los nuevos resultados en la ronda k según lo que haya recibido en la ronda anterior y envía los nuevos resultados en la ronda actual. Los algoritmos distribuidos que funcionan de forma asíncrona y no tienen estas rondas de ejecución sincrónica se denominan algoritmos asíncronos. Detectar la terminación de algoritmos distribuidos es necesario para detener el algoritmo cuando se cumple una determinada condición y esto no es una tarea trivial. Aunque el inicio y el final de una ronda causan sobrecargas en términos de mensajes adicionales necesarios, el diseño de algoritmos distribuidos síncronos es más sencillo que los algoritmos asíncronos en general. Los algoritmos asíncronos requieren una lógica de control más compleja y la detección de terminación en tales algoritmos también es más difícil.

Otra distinción es si un solo iniciador inicia el algoritmo distribuido o si hay más de un iniciador. Un solo iniciador que también controla la ejecución general del algoritmo significa un único punto de supervisión que es más fácil de administrar que los procesos controlados individualmente. Por lo tanto, podemos clasificar los algoritmos distribuidos según la sincronización a nivel de aplicación o algoritmo de la siguiente manera.

- Algoritmos de iniciador único sincrónico (SSI): hay un iniciador único que inicia el algoritmo, sincronizando el inicio y el final de cada ronda e iniciando la terminación. Estos algoritmos son más sencillos de diseñar que otros, ya que hay un solo proceso que controla la operación.
- Algoritmos de iniciador único asíncrono (ASI): este tipo de algoritmos tiene un único iniciador, pero la actividad en cada nodo se realiza de forma asíncrona desde los otros nodos. La sincronización y la detección de terminación son más difíciles para tal algoritmo que un algoritmo sincrónico.
- Algoritmos de iniciador simultáneo sincrónico (SCI): estos algoritmos se ejecutan de forma sincrónica, pero pueden ser iniciados por iniciadores concurrentes.
- Algoritmos de iniciador simultáneo asíncrono (ACI): hay más de un iniciador en este caso y las actividades son asíncronas. Este modo de operación es el caso más general, pero puede requerir un control complejo.

En el caso de un algoritmo SSI, se puede usar convenientemente un árbol de expansión creado previamente para transferir mensajes de control. En base a lo anterior, una posible plantilla de algoritmo SSI se traza en el algoritmo 5.1. Todos los procesos comienzan la ronda k cuando reciben el mensaje de inicio sobre el árbol de expansión T , que es básicamente una operación de transmisión sobre T , como veremos en breve. Las tres acciones de la ronda son enviar los resultados de la ronda anterior a todos los vecinos, recibir los resultados de la ronda anterior de los vecinos y preparar nuevos resultados para la próxima ronda. Cuando un proceso termina de ejecutar una ronda, espera a que todos sus hijos en T terminen antes de poder enviar la parada.mensaje a su parent. Cuando la raíz del árbol de expansión T recibe un mensaje de detención de todos sus hijos, la ronda k ha terminado y la raíz ahora puede comenzar la ronda $k+1$. Usaremos esta estructura con frecuencia mientras diseñamos algoritmos de gráficos distribuidos.

```

Algorithm 5.1 SSI_template
1: boolean finished, round_over ← false
2: message type start, result, stop
3: while ¬round_over do
4:   receive msg(i)
5:   case msg(i).type of
6:     start : send result(i) to all neighbors
7:     receive result(j), from each neighbor j
8:     compute result(i), finished ← true
9:   stop : if stop received from all children and finished then
10:    send stop to parent
11:    round_over ← true, finished ← false
12: end while

```

5.5 Criterios de rendimiento

El rendimiento de un algoritmo distribuido se evalúa en términos de tiempo, mensaje, espacio y complejidad de bits que se describen a continuación:

- **Complejidad de tiempo** : la complejidad de tiempo es el número de pasos necesarios para que el algoritmo distribuido termine como en un algoritmo secuencial. Para los algoritmos distribuidos sincrónicos, estaríamos interesados principalmente en el número de rondas como complejidad de tiempo.

- Complejidad del mensaje : este parámetro se considera comúnmente como el costo dominante de un algoritmo distribuido, ya que muestra directamente la utilización de la red e indica los costos de sincronización entre los nodos de la red. Transferir un mensaje a través de una red es una magnitud de órdenes más costosa que hacer cálculos locales.
- Complejidad de bits : la longitud de un mensaje también puede afectar el rendimiento de un algoritmo distribuido, especialmente si la longitud del mensaje aumenta a medida que el mensaje atraviesa la red. Para una red grande modelada por un gráfico con muchos vértices y bordes, la complejidad de bits puede ser significativa, lo que afecta directamente el rendimiento de la red.
- Complejidad del espacio : Este es el almacenamiento requerido en un nodo del sistema distribuido para el algoritmo en consideración.

5.6 Ejemplos de algoritmos de gráficos distribuidos

Ahora estamos listos para diseñar e implementar algoritmos de gráficos distribuidos simples. Describiremos algoritmos básicos de muestra que siguen una secuencia lógica. El primer algoritmo utiliza una técnica llamada inundación para enviar un mensaje desde un nodo del gráfico que representa la red a todos los demás nodos. A continuación, hacemos uso de este algoritmo para construir un árbol de expansión de la red que puede ser utilizado para la difusión eficiente y convergecast de mensajes de la red como se describe a continuación.

5.6.1 algoritmo de inundación

Nuestro objetivo es enviar un mensaje desde un solo nodo a todos los nodos del gráfico. Esta operación se denomina difusión y tiene muchas aplicaciones en redes reales, por ejemplo, para informar a todos los nodos de una condición de alarma que ocurre en un nodo. En el caso más simple, podemos tener las siguientes reglas como un primer intento para resolver este problema:

El iniciador i envía un mensaje $msg(i)$ a todos sus vecinos.

- 1.
2. Cualquier nodo que reciba el mensaje msg lo envía a sus vecinos, excepto al que lo recibió.

Este algoritmo funciona bien y todos los nodos recibirá el mensaje MSG enviado por finalmente. Sin embargo, podemos obtener un algoritmo más eficiente con menos mensajes transferidos entre los nodos mediante una simple modificación: un nodo envía msg a sus vecinos solo cuando lo recibe por primera vez. De esta manera, se evita la transmisión duplicada a lo largo de un borde del gráfico en la misma dirección. Ahora necesitamos una forma de detectar si un mensaje se recibe por primera vez o no, lo que se puede implementar simplemente usando una variable como la que se visita, que es falsa inicialmente y se vuelve verdadera cuando llega el mensaje msg por primera vez. Sin embargo, este algoritmo modificado es fácil de implementar por un FSM que tiene dos estados, como se muestra en la Fig. 5.4, lo que también nos ayudará a comprender el uso de los FSM en algoritmos distribuidos.

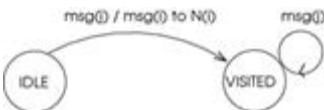


Fig. 5.4 FSM del algoritmo de Inundación.

Podemos implementar este algoritmo basado en el FSM como se muestra en el algoritmo 5.2. Cuando el mensaje msg llega por primera vez, se ingresa al estado VISITADO y se ignoran las recepciones posteriores de msg .

Algorithm 5.2 Flooding

```

1: [ code for process  $i$ , message received from process  $j$  ]
2:  $currstate \leftarrow IDLE$                                 // start with IDLE state
3: if  $i = initiator$  then
4:   send msg( $i$ ) to  $N(i)$ 
5:    $currstate \leftarrow VISITED$ 
6: end if
7:
8: while true do
9:   receive(msg( $j$ ))                                     // all nodes execute this part
10:  case currstate of
11:    IDLE: send msg( $i$ ) to  $N(i)/j$ 
12:    currstate  $\leftarrow VISITED$ 
13:    VISITED:                                         // do nothing
14:  end while

```

Podemos tener algunas mejoras a este algoritmo de la siguiente manera.

- En lugar de esperar para siempre en la línea 8, podemos tener una condición de terminación. Un proceso i puede esperar ciertos momentos, tales como el diámetro $diam(G)$ de la gráfica. Esto es lógico ya que $diam(G)$ es la ruta más larga que puede atravesar el mensaje msg . Una vez que el mensaje MSG se recibe $diam(G)$ veces, proceso i termina.
- Podemos tener un contador comúnmente llamado time-to-live (TTL) contenido en el mensaje. Cada vez que se recibe, el TTL se decrementa y un mensaje con un valor de 0 TTL no se reenvía a los vecinos.

Análisis

Una mirada cuidadosa a este algoritmo revela que cada borde del gráfico se recorrerá a lo sumo dos veces, una vez en cada dirección cuando ambos nodos en los extremos de un borde comienzan a enviar el mensaje msg al mismo tiempo. Por lo tanto, la complejidad del mensaje es $O(m)$. Suponiendo que hay al menos una transferencia de mensaje en cada unidad de tiempo, el tiempo que toma este algoritmo es la distancia más larga entre dos vértices del gráfico, que es su diámetro y, por lo tanto, la complejidad del tiempo es $\Theta(diam(G))$.

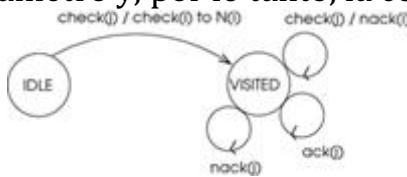


Fig. 5.5 FSM de la construcción de un árbol de expansión usando inundación

5.6.2 Construcción del árbol de expansión mediante inundación

Podemos diseñar una construcción de árbol de expansión de una red utilizando el algoritmo de Inundación con algunas modificaciones. La construcción de un árbol de expansión en un entorno de red significa que cada nodo conoce a su padre y sus hijos en el sentido general. No intentaremos almacenar toda la estructura de árbol en un nodo especial o en cada nodo del gráfico, ya que la relación padre / hijo en cada nodo es adecuada para transferir mensajes a través del árbol de expansión. Tenemos un solo iniciador como en el algoritmo de Inundación y este nodo se convierte en la raíz del árbol

de expansión que se formará. La primera modificación que tenemos es asignar el remitente j del mensaje $\text{msg}(j)$ como padre del receptor i . Si se recibe $\text{msg}(j)$ por primera vez. Dado que también requieren que los padres estar al tanto de sus hijos, el nodo i debería enviar un mensaje de acuse de recibo $\text{ACK}(i)$ a j para informar j de esta situación. De lo contrario, si el nodo i ya tiene un parente, lo que significa que ha sido visitado anteriormente, envía un mensaje de confirmación negativo $\text{nack}(i)$ al nodo j . Tenemos, por lo tanto, tres tipos de mensajes; cheque, ack , y nack . La determinación de los tipos de mensajes es crucial en el diseño de algoritmos de gráficos distribuidos; además, la determinación de los estados se realiza mediante mensajes si queremos usar un FSM. Modifiquemos el FSM de la figura 5.4 para reflejar lo que hemos estado discutiendo. Podemos ver que los estados pueden seguir siendo los mismos ya que un nodo puede estar en estado IDLE o VISITED como antes. Según su estado y el tipo de mensaje, es posible que debamos realizar diferentes acciones. El FSM modificado se muestra en la Fig. 5.5 con el estado VISITADO teniendo todos los tipos de mensajes posibles como entrada ahora.

Este FSM se puede traducir directamente a un algoritmo distribuido como se muestra en el Algoritmo 5.3 donde, además, también se especifica una condición de terminación para un nodo. La actividad de cualquier nodo se termina cuando se ha recibido ACK o NACK mensajes de todos sus vecinos excepto el remitente del mensaje que ha recibido por primera vez.

Algorithm 5.3 Flooding2

```

1: int parent ← Ø
2: set of int childs ← {Ø} , others ← {Ø}
3: message types check, ack, nack
4:
5: if  $i = \text{initiator}$  then
6:   send check to  $N(i)$ 
7:   currstate ← VISITED
8: end if
9:
10: while ( $\text{childs} \cup \text{others} \neq (N(i) \setminus \{parent\})$ ) do           /* all nodes execute this part
11:   receive(mg(j))
12:   case currstate ∧ msg(j).type of
13:     IDLE ∧ check:   parent ← j
14:                 send check to  $N(i) \setminus j$ 
15:                 send ack to  $j$ 
16:                 currstate ← VISITED
17:     VISITED ∧ check: send nack to  $j$ 
18:     VISITED ∧ ack:   childs ← childs ∪ {j}                      /* j is now a child
19:     VISITED ∧ nack: others ← others ∪ {j}                         /* j is not a child
20:   end if

```

El funcionamiento de este algoritmo se muestra en la Fig. 5.6.

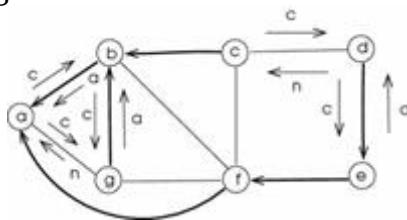


Fig. 5.6. Un árbol de expansión construido en una gráfica que usa inundación. La rama (g, b) está en el árbol, pero (g, a) no está desde que el mensaje de verificación (c) del nodo a llega a g más tarde que c de b , que es respondido por un mensaje nack (n). Se muestra una situación similar para la rama (e, d) donde el mensaje c del nodo d se responde con un mensaje ack (a) y (d, e) está incluido en el árbol

Podríamos haber implementado fácilmente este algoritmo sin usar un FSM, un nodo que tiene un parente o no básicamente muestra su estado como IDLE o VISITED. Teniendo esto en cuenta, este algoritmo se muestra en el algoritmo 5.4 como en [2]. Sin embargo,

para algoritmos distribuidos complicados, el uso de FSMs facilitaría el diseño y la implementación.

Algorithm 5.4 Flooding³

```

1: int parent ← ⊥
2: set of int childx ← {0} , others ← {0}
3: message types check, ack, nack
4:
5: if i = initiator then                                ▷ root initiates tree construction
6:   send check to N(i)
7:   parent ← i
8: end if
9:
10: while (childx ∪ others) ≠ (N(i) \ {parent}) do
11:   receive msg(j)
12:   case msg(j).type of
13:     check : if parent = Ø then                      ▷ check received first time
14:       parent ← j
15:       send ack to j
16:       send msg(i).check to N(i) \ {j}
17:     else                                           ▷ check received before
18:       send msg(i).reject to j
19:     ack : childx ← childx ∪ {j}                      ▷ j is a child
20:     nack : others ← others ∪ {j}                      ▷ j is not a child
21:   end while

```

Análisis

Cada borde de la gráfica será atravesado al menos dos veces por pares de mensajes de verificación / ack o chequeo / nack y como máximo cuatro veces cuando dos nodos comienzan a enviarse mensajes de verificación entre sí simultáneamente. Por lo tanto, la complejidad del mensaje de este algoritmo es $O(m)$. La profundidad del árbol construido será a lo sumo $n - 1$, asumiendo que se construye una red lineal. Si hay al menos una transferencia de mensajes por unidad de tiempo, la complejidad del tiempo es $O(n)$.

5.6.3 Comunicaciones básicas

Hay una serie de operaciones de comunicación básicas realizadas en un sistema distribuido. Uno de estos procesos es la difusión que es iniciada por un nodo enviando un mensaje y todos los nodos en el sistema distribuido tienen una copia del mensaje al final de la operación de difusión. Otra primitiva fundamental es la transmisión por convergencia en la que los datos de cada nodo se recopilan en un nodo especial del sistema. Veremos estas dos operaciones en esta sección. Otra actividad es el envío de mensajes de multidifusión en los que un mensaje se entrega solo a un subconjunto específico de procesos.

Difusión sobre un árbol de expansión

Para la operación de difusión, asumiremos que un gráfico representa la red del sistema distribuido y un árbol de expansión T ya está construido por un algoritmo similar al que hemos analizado. La difusión es iniciada por un nodo enviando msg a todos sus hijos. Cualquier nodo en el árbol T que tenga hijos simplemente reenvía msg a todos sus hijos. Dado que msg se transfiere solo sobre los bordes del árbol, el número de mensajes será $n - 1$. Para una gráfica con n vértices. El tiempo empleado será la profundidad de T, asumiendo el envío simultáneo de mensajes en cada nivel. La profundidad de T puede ser un máximo de $n - 1$ asumiendo una red lineal.

Convergecast sobre un árbol de expansión

En ciertas redes, los datos de todos los nodos se deben recopilar en un nodo con más capacidades y este nodo especial puede analizar y evaluar todos los datos, proporcionar informes que contienen estadísticas que pueden transferirse a centros de cómputo más avanzados o usuarios para su posterior procesamiento. Esta situación se encuentra comúnmente en las redes de sensores inalámbricos donde los datos detectados deben pasar por estos pasos de operación. La recopilación de datos se simplifica mucho cuando

se utiliza un árbol de expansión construido de antemano. En este caso, las hojas del árbol envían sus datos a sus padres, los padres combinan sus propios datos con los de las hojas y los envían a sus padres. Un nodo intermedio puede, de hecho, realizar alguna operación simple en datos, como calcular el promedio o encontrar valores extremos. De esta manera, Los datos enviados hacia arriba en el árbol no tienen que aumentar mucho en cada nivel. Este proceso de recolección se llama convergecast continua hasta que todos los datos se recopilan en el nodo especial, comúnmente denominado sumidero en las redes de sensores. El algoritmo 5.5 muestra el pseudocódigo para el proceso de convergecast en un árbol de expansión. Las hojas del árbol inician el algoritmo y cualquier nodo intermedio en el árbol debe esperar hasta que se reciben los datos de todos sus hijos antes de combinar estos datos con los suyos para enviarlos a su parent, como se observó en la línea 8 del algoritmo. La condición de terminación para la raíz del árbol se cumple cuando recibe los mensajes de convergecast de todos sus hijos en la línea 12. Para todos los demás, la terminación está en la línea 17 cuando envían sus datos a sus padres.

Algorithm 5.5 Convergecast

```

1: int parent
2: set of int child, received ← {0}, data ← {0}
3: message types convergecast
4:
5: if child = {0} then
6:   send convergecast to parent
7: else
8:   while child ≠ received do           ▷ leaf nodes start convergecast
9:     receive convergecast(j)          ▷ any intermediate node or root
10:    received ← received ∪ {j}
11:    data ← data ∪ convergecast(j)
12:  end while
13: end if
14: if i ≠ root then
15:   combine data into convergecast
16:   send convergecast to parent
17: end if

```

Las complejidades de mensaje y tiempo para este algoritmo son las mismas que el algoritmo de difusión que utiliza un razonamiento similar. La Figura 5.7 muestra el funcionamiento del algoritmo de Convergecast utilizando el árbol de expansión construido en la Fig. 5.6 . Los mensajes están etiquetados con par (a , b); a que muestra el período de tiempo y b es la duración del mensaje. Podemos ver el nivel más alto de convergecast deacabados de árbol en 5 unidades de tiempo, ya que esta es la duración más larga, seguido de 6 unidades en el nivel 2 y 2 unidades en el nivel 1 para un total de 13 unidades de tiempo.

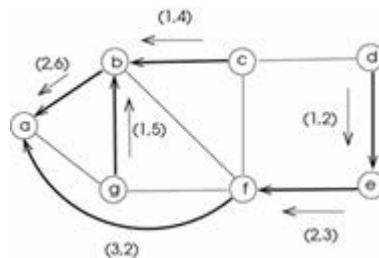


Fig. 5.7 Convergecast sobre el árbol de expansión de la Fig. 5.6 . Los valores de la etiqueta del mensaje muestran la transferencia concurrente de hermanos y la duración de los mensajes

5.6.4 Elección del líder en un anillo

La elección de líder o coordinador es necesaria en sistemas distribuidos, ya que este nodo especial puede iniciar un algoritmo y supervisar la ejecución general del algoritmo como en un algoritmo SSI. El líder también puede tomar acciones correctivas cuando se encuentran fallas en la ejecución de un algoritmo. Los nodos y los enlaces de comunicación

pueden fallar físicamente y, aunque inicialmente podemos asignar un nodo como líder de la red, debemos elegir un nuevo líder cuando ocurra una falla. Los algoritmos de elección proporcionan formas de asignar un nuevo líder en la red cuando el líder actual falla.

Hay muchos algoritmos de elección de líderes en la literatura para sistemas distribuidos. Como ejemplo de algoritmo distribuido de introducción, consideraremos la elección del líder en un anillo con nodos que tienen identificadores únicos. La transferencia de mensajes es en una sola dirección. Este ejemplo puede ser descrito convenientemente por un FSM simple con los siguientes estados:

- LEAD: Los nodos tienen un líder en este estado estable.
- ELECT: la elección se está llevando a cabo cuando un nodo está en este estado.

La idea principal de este algoritmo es que cualquier nodo que detecte la falla del líder actual inicia el algoritmo enviando un mensaje de elección que contiene su identificador a su vecino a su derecha, asumiendo un anillo unidireccional en el sentido de las agujas del reloj. Un nodo que recibe este mensaje cambia su estado a ELEGIR. Si el identificador en el mensaje es mayor que su propio identificador, simplemente pasa el mensaje de elección a su próximo vecino. De lo contrario, inserta su identificador que es mayor que el identificador en el mensaje entrante y lo envía al vecino. Tenemos dos mensajes en este ejemplo:

- Elección : Enviado por cualquier nodo que detecte falla de líder. Este mensaje puede ser enviado por más de un iniciador.
- Líder : el nuevo líder emite este mensaje para notificar a todos los nodos que la elección ha terminado.

Cuando un proceso con identificador i recibe un mensaje con un identificador j en él, comprueba y realiza una de las siguientes acciones:

- : El proceso i reemplaza j con i en el mensaje y lo pasa al siguiente nodo. $i > j$
- : Proceso i simplemente pasa el mensaje a la siguiente nodo. $i < j$
- : El proceso i se convierte en el líder y envía el mensaje del líder a su próximo vecino. $i = j$

En el último caso, el mensaje de elección que se origina en el nodo i se ha vuelto a sí mismo, lo que significa que tiene el identificador más alto entre todos los procesos activos. Básicamente, el identificador más alto se transfiere entre todos los nodos en funcionamiento y cuando el originador recibe su propio mensaje, determina que es el líder y envía el mensaje de líder a su vecino, que luego se transmite a todos los nodos mediante transferencias de vecinos. El FSM para este algoritmo se muestra en la Fig. 5.8 .

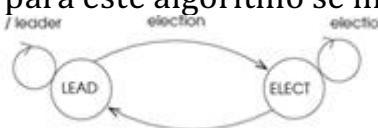


Fig. 5.8 FSM del algoritmo de elección del líder de anillo

Análisis

El peor de los casos ocurre cuando los nodos se ordenan de menor a mayor con respecto a sus identificadores en el sentido de las agujas del reloj y se inician las elecciones simultáneamente en el sentido contrario a las agujas del reloj. El mensaje del identificador más grande viaja a través de todos los nodos n veces, el segundo identificador más grande se transfiere $n - 1$ tiempos y en total habrá $\sum_{i=1}^n = n(n + 1)/2$ mensajes como se muestra en la

Fig. 5.9 a. El mejor caso ocurre para un total de $2 n - 1$ mensajes cuando los mensajes se transmiten en el sentido de las agujas del reloj como se muestra en la Fig. 5.9 b. En este caso, incluso si todos los nodos inician la elección al mismo tiempo, sus mensajes serán eliminados por los siguientes nodos para $n - 1$ veces y solo el mensaje del nodo identificador más alto, que es 7 en este caso, atravesará el anillo hasta el originador en n pasos. El número total de pasos será entonces $2n - 1$, excluyendo el mensaje de declaración enviado por el líder.

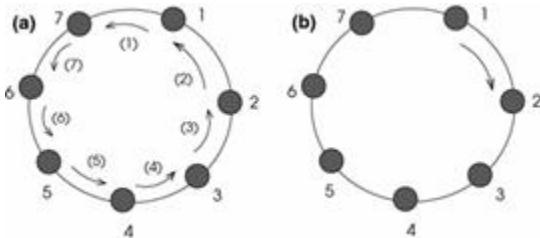


Fig. 5.9 Algoritmo de elección del líder de anillo: peores y mejores escenarios. En a , cada mensaje del originador está etiquetado con la cantidad de enlaces que viaja. Por ejemplo, el mensaje que se origina en el nodo 7 está etiquetado con 7 ya que pasa por 7 bordes hasta el nodo 7. El mejor caso se muestra en b

5.7 Notas del capítulo

Hemos descrito los sistemas distribuidos y los problemas fundamentales en el diseño de algoritmos para tales sistemas en este capítulo. Los sistemas distribuidos son necesarios, ya que proporcionan recursos compartidos, tolerancia a fallos y, en varias implementaciones, la aplicación se distribuye de forma inherente, como un sistema de control de fábrica. Las plataformas comunes para implementar algoritmos distribuidos son Internet, redes móviles ad hoc, redes de sensores inalámbricas, Grid y Cloud. En muchos casos, podemos modelar estas redes adecuadamente mediante gráficos con vértices de un gráfico que representa nodos computacionales y bordes que muestran los enlaces de comunicación entre ellos.

Los sistemas distribuidos requieren algoritmos distribuidos que se ejecutan en los nodos de dicho sistema. La sincronización y la comunicación son dos requisitos básicos en el diseño eficiente de tales algoritmos. La sincronización puede realizarse en varios niveles; Hardware, sistema operativo / middleware y nivel de aplicación. Vemos que la sincronización en el nivel de la aplicación mediante el uso de mensajes se usa comúnmente debido a la versatilidad y facilidad de implementación que luego se puede transferir a primitivas de sincronización locales. En este llamado mensaje de paso. Los sistemas distribuidos, la comunicación principal y la sincronización se logran únicamente mediante mensajes. El receptor de un mensaje decide qué hacer a continuación, principalmente por el tipo de mensaje que llega. Un algoritmo distribuido síncrono normalmente se ejecuta en rondas y la siguiente ronda no se inicia hasta que todos los nodos terminan de ejecutar la ronda actual. La sincronización al principio y al final de la ronda se realiza comúnmente mediante mensajes especiales enviados por un nodo especial.

Los algoritmos distribuidos pueden ser modelados por las FSM, que son modelos matemáticos que incluyen estados y transiciones entre estados, tal como lo hemos descrito. Podemos diseñar un algoritmo distribuido sin un FSM, pero para algoritmos complicados, los FSM proporcionan un algoritmo más ordenado con ayuda visual y menos

propensos a errores que los algoritmos que de otra manera podrían involucrar muchas declaraciones de toma de decisiones.

Luego describimos algunos algoritmos de gráficos distribuidos de muestra que incluyen la construcción de un árbol de expansión del gráfico, las operaciones de transmisión y convergecast sobre un árbol de expansión, y un algoritmo de elección líder para encontrar el nuevo coordinador de nodos en un anillo cuando falla el líder. Necesitamos probar que un algoritmo distribuido logra correctamente lo que está destinado; y las complejidades de tiempo, mensaje, bit y espacio de un algoritmo distribuido se utilizan para evaluar su rendimiento. En general, la complejidad del mensaje se considera como el costo dominante de un algoritmo distribuido.

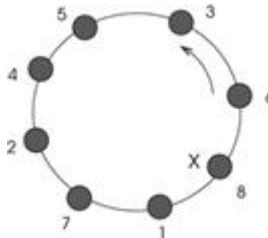


Fig. 5.10 Estructura anular para los ejercicios 4.

Ceremonias

El algoritmo elevador del Ejemplo 5.1 se modificará de modo que se agregue una luz verde que muestre el movimiento hacia arriba y una luz amarilla para el movimiento hacia

1. abajo. Proporcione las modificaciones necesarias al diagrama FSM, la tabla FSM y el código C para incorporar estas dos salidas.
2. Una cadena de bits binarios S tiene paridad par si el número de bits en S es un número par y, de lo contrario, tiene paridad impar. Proporcione el diagrama FSM, la tabla FSM y el código C de un algoritmo que lee una cadena binaria bit a bit y decide estar en estado par o impar después de cada lectura. Utilice el estilo de programación que se muestra en el código C del Ejemplo 5.1 .
3. Necesitamos modificar el algoritmo de difusión sobre un árbol de expansión para que el iniciador se dé cuenta de que cada nodo ha recibido el mensaje de difusión. Esto se puede realizar simplemente si cada nodo aplaza para enviar un acuse de recibo al remitente del mensaje hasta que reciba el acuse de recibo de todos sus hijos, similar a la operación de convergecast que debe iniciarse a partir de las hojas del árbol de expansión una vez que reciben la emisión. mensaje. Escriba el pseudocódigo para este algoritmo con comentarios y calcule su tiempo y las complejidades de los mensajes.
4. Muestre la ejecución del algoritmo de elección de anillo para los nodos que se muestran en la Fig. 5.10 . Suponga que los nodos 2 y 5 encuentran simultáneamente que el líder no está trabajando y deciden realizar una elección.
5. En un gráfico totalmente conectado con cada nodo con identificadores únicos, se puede usar el algoritmo de intimidación para elegir un nuevo líder. Un nodo u que encuentre que el líder no está funcionando puede iniciar este algoritmo enviando un mensaje de elección a todos los nodos que tengan identificadores más altos que ellos. Cualquier nodo v que reciba este mensaje devuelve y recibe un mensaje de confirmación al nodo u, que luego abandona la elección. El nodo v ahora comienza la elección y este proceso continúa hasta que hay un ganador que es el nodo activo con el identificador más alto. El nuevo líder lo transmite ganando con un mensaje especial a todos los nodos. Escriba el pseudocódigo para este algoritmo y encuentre su complejidad de tiempo y mensaje. Muestre su operación en un gráfico completo de 8 nodos donde los nodos 4 y 6 encuentran simultáneamente que el líder 8 está caído.

Referencias

1. Attiya H, Welch J (2004) Computación distribuida: fundamentos, simulaciones y temas avanzados, 2^a ed . Wiley, nueva york
2. Erciyes K (2013) Algoritmos de grafos distribuidos para redes informáticas. Springer serie de comunicaciones y redes informáticas. Springer, Berlín. ISBN-10: 1447151720 (16 de mayo de 2013)
3. Foster I, Kesselman C (2004) La cuadrícula: modelo para una nueva infraestructura informática. Morgan Kaufmann, San Mateo
4. Mell P, Grance T (2011) La definición NIST de computación en la nube. Instituto Nacional de Estándares y Tecnología, Departamento de Comercio de los Estados Unidos, publicación especial, 800145
5. Tanenbaum AS, Steen MV (2007) Sistemas distribuidos, principios y paradigmas, 2^a ed . Pearson-Prentice Hall, Upper Saddle River. ISBN 0-13-239227-5
6. Tel G (2000) Introducción a los algoritmos distribuidos, 2^a ed . Cambridge University Press, Cambridge

Parte II

Algoritmos básicos de grafos

6. Árboles y gráficas transversales.

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Un árbol es un gráfico acíclico conectado y un bosque está formado por árboles. En este capítulo, primero describimos la estructura de árbol, los algoritmos para construir un árbol de expansión de un gráfico y los algoritmos de recorrido de árbol. Dos métodos principales de recorrido de gráficos son la búsqueda en profundidad y la búsqueda en primer lugar. Revisamos algoritmos secuenciales, paralelos y distribuidos para estos recorridos junto con sus diversas aplicaciones.

6.1 Introducción

Un árbol es un gráfico acíclico conectado y un bosque está formado por árboles. Los árboles encuentran muchas aplicaciones en la informática y en situaciones de la vida real. Por ejemplo, la organización de una universidad o de cualquier establecimiento generalmente se muestra en un árbol, y los árboles familiares ilustran las relaciones de los padres entre los individuos. En informática, los árboles se utilizan para el almacenamiento eficiente de datos y los algoritmos basados en árboles encuentran una amplia gama de aplicaciones. Un árbol de expansión de una gráfica es su subgrafo de árbol que incluye todos los vértices de la gráfica. Una gráfica puede tener una cantidad de árboles en expansión. Hemos descrito brevemente los árboles en el cap. 2 ; Ahora, proporcionamos un análisis más detallado de los árboles con algoritmos relacionados en este capítulo. Comenzamos definiendo la estructura de árbol e indicando sus propiedades. Luego describimos los algoritmos para construir árboles de expansión y árboles transversales y revisamos brevemente los tipos de árboles especiales.

Es necesario atravesar todos los vértices o todos los bordes de un gráfico en algún orden en varias aplicaciones, por ejemplo, para encontrar todos los vértices alcanzables en un gráfico. Los algoritmos que realizan recorridos también pueden usarse como bloques de construcción de algoritmos de grafos más complicados. En un recorrido de gráfico indireccional , todos los bordes se consideran, mientras que solo los bordes salientes de un nodo se consideran en un gráfico dirigido. Dos métodos principales de traversal gráfico son la búsqueda en profundidad y en amplitud búsqueda. En el primer método, partimos de cualquier vértice de una gráfica y profundizamos tanto como

podemos visitando a los vecinos de cada vértice visitado. La búsqueda de amplitud implica visitar primero a todos los vecinos de un vértice, luego visitar a todos los vecinos de estos vecinos y proceder de esta manera hasta que se visiten todos los vértices. Ambos métodos producen árboles de expansión arraigados en el vértice de inicio. Describimos los algoritmos secuenciales, paralelos y distribuidos para ambos enfoques con sus posibles aplicaciones en este capítulo.

6.2 árboles

Un gráfico es un árbol si está conectado y no contiene ningún ciclo. Un bosque es un gráfico sin ciclos. Cada camino es un árbol y un árbol T es el camino si y sólo si el grado máximo de T es 2. Un árbol puede ser arraigado o sin raíces . Un nodo designado llamado raíz en un árbol con raíz está en la parte superior de la jerarquía y todos los demás vértices de T tienen una ruta a la raíz; El árbol está desraoteado de lo contrario. Un árbol binario consiste en nodos que tienen como máximo dos hijos. Las siguientes afirmaciones definen igualmente un árbol T :

T está conectado y tiene $n - 1$ bordes

- 1.
2. Cualquiera de los dos vértices de T están conectados exactamente por un camino.
3. T está conectado y cada borde es una eliminación cortar-borde de que desconecta T .

Definición 6.1 (nivel) El nivel de un vértice en un árbol arraigado es su distancia a la raíz. El nivel de un vértice también se llama su profundidad en el árbol.

Definición 6.2 (padre, hijo) Un vértice v que está conectado al vértice u (el predecesor de u) en la ruta a la raíz se llama el padre de u y v se llama el hijo de u .

Un vértice v puede tener solo un parente, ya que tener más de un parente produce un ciclo y, por lo tanto, la estructura resultante no será un árbol.

Definición 6.3 (hoja, vértice interno, hermanos) Una hoja es un vértice del árbol que no tiene hijos. Un vértice interno de un árbol tiene un parente y uno o más hijos. Los hermanos en un árbol tienen el mismo parente.

El nivel máximo de una hoja es la altura (o profundidad) del árbol, ya que es el vértice más lejano a la raíz.

Definición 6.4 (árbol de expansión) Un árbol de expansión $T = (V, E')$ de un grafo $G = (V, E)$ tiene el mismo conjunto de vértices de G con un conjunto de borde que es un subconjunto de los bordes de G .

Un árbol de expansión mínima MST (G) de un grafo ponderado, no dirigido G es un árbol de expansión de G con el coste peso borde total mínima entre todos los árboles de expansión de G . Los MST encuentran numerosas aplicaciones e investigaremos los algoritmos secuenciales, paralelos y distribuidos para los MST en el Cap. 8 .

Definición 6.5 (árbol m- ario $(m \geq 2)$) Una m - aria árbol es un árbol con raíz en el que cada vértice aparte de las hojas tiene a lo sumo m niños. En un árbol binario , .

Definición 6.6 (completar m- ary árbol) Una completa m - aria árbol es un m - aria árbol en el que cada vértice interno del árbol tiene exactamente m niños y todas las hojas tienen la misma profundidad.

Definición 6.7 (árbol ordenado) En un árbol ordenado, hay un ordenamiento lineal de los hijos de cada nodo. Esto significa que podemos identificar a los niños como primero, segundo, etc.

La figura [6.1](#) muestra estos conceptos.

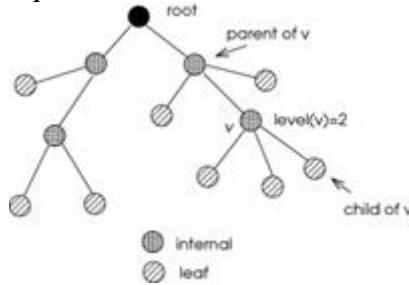


Fig. 6.1 Los vértices de un árbol de muestra que es un árbol de 3 arios ya que todos los vértices que no sean las hojas tienen como máximo tres hijos

Definición 6.8 (centro de un árbol) Un centro de un árbol T es un vértice v tal que $\max(d(u, v)) \forall u \in V$ Es mínimo o dos centros adyacentes con esta propiedad.

Podemos encontrar el (los) centro (s) de un árbol retirando recursivamente sus hojas hasta que queden uno o dos centros. Este procedimiento se ilustra en la figura [6.2](#) .

6.2.1 Propiedades de los árboles

Teorema 6.1 un grafo no dirigido G es un árbol si y sólo si existe un camino simple única entre cualquier par de vértices de G .

Prueba Nos primera asumiremos G es un árbol que significa que no tiene ciclos. Ahora, si hay dos rutas simples entre cualquier par de vértices (u , v) en G , la ruta total de u a v y luego de vuelta a u formará un ciclo; sin embargo, a partir de la definición de un árbol, sabemos que G no tiene un ciclo y, por lo tanto, una contradicción. En la otra dirección de la declaración, supongamos que G es un gráfico en el que dos vértices u y v están conectados por una ruta única. Si hubiera dos rutas distintas entre un par de vértices u y v , estas rutas formarían un ciclo y dado que G es un árbol y no contiene un ciclo, tenemos una contradicción. □

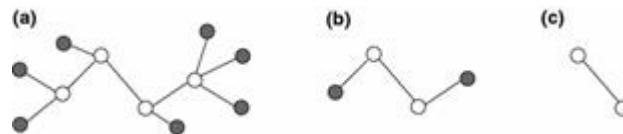


Fig. 6.2 Encontrando dos centros de un árbol sin raíz. Las hojas se retiran recursivamente de a - c para obtener los dos centros

Teorema 6.2 Cada árbol de orden n tiene tamaño $m = n - 1$.

Prueba Nosotros demostraremos este teorema por inducción n . La hipótesis de la inducción es que un árbol con n nodos tiene $n - 1$ bordes Para el caso base, cuando $n = 1$, el grafo trivial no tiene aristas; por lo tanto, el caso base se mantiene. Consideremos un árbol T con $n + 1$ nodos La eliminación de un nodo hoja v con su borde incidente de T deja un árbol $T' = (V', E')$. Ya que no hemos creado un ciclo al hacerlo, T' También es un árbol, digamos con p aristas y q vértices. Con la hipótesis inductiva, $p = q - 1$. Desde que eliminamos un borde y un nodo de T , $p = m - 1$, $q = n - 1$. Sustitución de rendimientos. $p = n - 2$, cuando $q = n - 1$ de ahí $m = n - 1$

□

Teorema 6.3 Cualquier gráfica conectada G con n vértices y $n - 1$ Los bordes son un árbol.

Prueba que tenemos que mostrar que G es acíclico. Supongamos lo contrario que G tiene al menos un ciclo y eliminamos de forma iterativa los bordes de los ciclos hasta que tengamos un gráfico G' Que es acíclica y por tanto es un árbol. Podemos concluir G' tiene $n - 1$ Bordes por el teorema [6.2](#) . Desde que hemos eliminado al menos un borde para obtener G' , G tenía un tamaño al menos n que es uno mayor que el tamaño de G' Y de ahí una contradicción. □

6.2.2 Encontrar la raíz de un árbol mediante el salto de puntero

En el método de salto de puntero, tendríamos que cada elemento de una lista vinculada apunte al enlace del elemento al que apunta en cada paso. Este método puede usarse convenientemente para encontrar la raíz de un árbol con raíz como se muestra en el algoritmo 6.1. Después $\lceil \log_{\text{depth}} \rceil$ pasos, todos los vértices del árbol apuntarán a la raíz. Tenga en cuenta que este método es inherentemente paralelo.

Algorithm 6.1 Finding Root of a Tree

```

1: Input: a rooted tree  $T = (V, E)$  with  $n$  elements
2: Output: each vertex  $v$  points to the root
3: for  $i=1$  to  $\lceil \log_{\text{depth}} \rceil$  do
4:   for each tree vertex  $v \in V$  in parallel do
5:      $(v \rightarrow \text{parent}) \leftarrow ((v \rightarrow \text{parent}) \rightarrow \text{parent})$ 
6:   end for
7: end for

```

Encontrar la raíz de un árbol con profundidad 3 se muestra en la Fig. [6.3](#) . Se necesitan tres iteraciones para que todos los nodos apunten a la raíz.

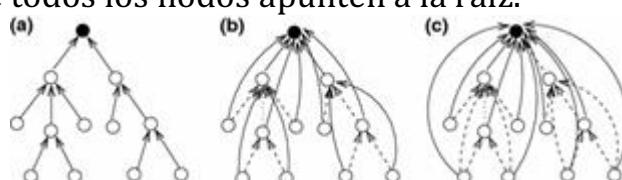


Fig. 6.3 La raíz de un árbol se encuentra en dos iteraciones por todos los nodos

6.2.3 Contando los árboles de expansión

Un árbol de expansión de una gráfica G es su subgrafo que es un árbol y contiene todos los vértices de G . Cada gráfico conectado y no dirigido tiene al menos un árbol de expansión. Los árboles de expansión encuentran diversas aplicaciones, como el suministro de una infraestructura de comunicación en redes informáticas y análisis de clústeres. Denotemos el número de árboles que abarcan una gráfica G por $t(G)$. Cayley proporcionó una fórmula para encontrar el número de árboles que abarcan un gráfico completo etiquetado K_n que tiene un identificador único para cada uno de sus vértices como sigue [4]:

$$\tau(K_n) = n^{n-2}$$

(6.1)

Los 3 árboles que se extienden de K_3 y 16 árboles que abarcan de etiquetado K_4 se muestran en la Fig. 6.4 y la Fig. 6.5 .

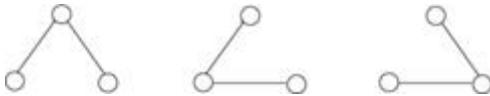


Fig. 6.4 Todos los posibles tres árboles que se pueden etiquetar de K_3

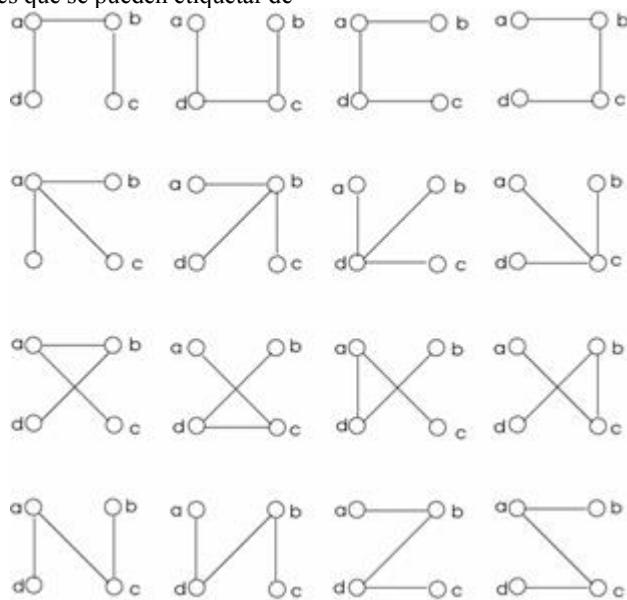


Fig. 6.5 Todos los posibles 16 árboles que se pueden etiquetar de K_4

6.2.4 Construyendo árboles de expansión

Describimos tres algoritmos simples para construir un árbol de expansión de un gráfico no dirigido, simple y conectado en esta sección.

6.2.4.1 El primer algoritmo

Como un enfoque simple, comenzaremos con todos los bordes de un gráfico que pertenece a T que, de hecho, puede que no sea un árbol, y luego eliminaremos de manera iterativa los bordes de T que no darán como resultado un gráfico G desconectado . Continuamos hasta que T tenga una de las propiedades básicas del árbol descritas anteriormente; es decir, tiene $n - 1$ bordes, es acíclico, etc. El pseudocódigo de este algoritmo se encuentra en el algoritmo 6.2, donde verificamos la propiedad del árbol de que cada borde de un árbol es un borde cortado, es decir, la eliminación de cualquier borde deja un árbol desconectado.

Algorithm 6.2 ST_Algo

```

1: Input:  $G = (V, E)$ 
2: Output: Spanning Tree  $T$  of  $G$ 
3:  $T \leftarrow E$ 
4: repeat
5:   pick any edge  $e \in T$  removal of which does not disconnect  $T$ 
6:    $T \leftarrow T - \{e\}$ 
7: until  $T$  any edge removal leaves  $T$  disconnected

```

Los pasos de operación de este algoritmo se muestran en la Fig. 6.6. Eliminamos un borde en cada iteración, y por lo tanto el *repeat-until*. El bucle se ejecuta $O(m)$ veces. Alternativamente, podríamos comprobar que la estructura resultante después de cada borde tiene $n-1$ bordes y está conectado. Comprobar lo primero es simple manteniendo un contador y disminuyéndolo después de cada eliminación de borde. Sin embargo, verificar la conexión de la gráfica no es trivial como veremos en la segunda parte de este capítulo. Tenga en cuenta que requerimos ambas propiedades según el teorema 6.3. Veremos que podemos construir árboles de expansión con propiedades especiales en tiempo lineal en el Cap. 6.

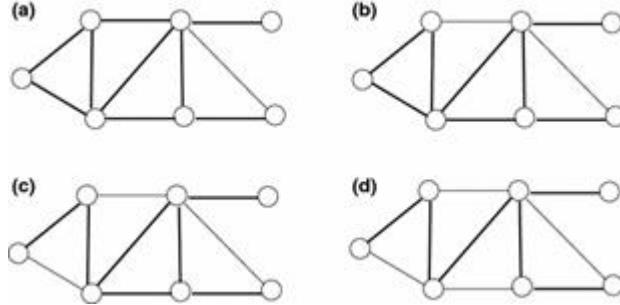


Fig. 6.6 Pasos del algoritmo 6.2 en una gráfica de muestra. Todos los bordes del gráfico que se muestran en negrita se incluyen inicialmente en el árbol de expansión. Luego, una eliminación de bordes que no desconecta el árbol se elimina en cada iteración a partir de a hasta que esto no sea posible

6.2.4.2 El segundo algoritmo

Un algoritmo alternativo para construir un árbol de expansión T de un gráfico $G = (V, E)$ Se puede construir de la siguiente manera. Esta vez, comenzamos con un árbol vacío T y recoger un vértice arbitrario u , e incluyen T en T . A partir de entonces, nos seleccionar arbitrariamente cualquier arco de salida (u, v) , que es una ventaja con un punto final decir T en T y el otro extremo v fuera T , e incluyen (u, v) en T . Procedemos de esta manera hasta que todos los vértices se procesen como se muestra en el algoritmo 6.3.

Algorithm 6.3 ST_mAlg2

```

1: Input:  $G = (V, E)$ 
2: Output: Spanning Tree  $T$  of  $G$ 
3:  $T \leftarrow$  an arbitrary vertex  $u$ 
4:  $V' \leftarrow \emptyset$ 
5: while  $V' \neq V$  do
6:   select any outgoing edge  $(u, v)$  from  $T$  vertices with  $u \in T \wedge v \notin T$ 
7:    $T \leftarrow T \cup \{(u, v)\}$ 
8:    $V' \leftarrow V' \cup \{v\}$ 
9: end while

```

La corrección es evidente ya que cualquier borde saliente no producirá un ciclo con bordes ya incluidos en el árbol. Por lo tanto, la estructura resultante estará libre de ciclos y, por lo tanto, será un árbol. Una posible operación de este algoritmo en un pequeño gráfico de muestra se muestra en la Fig. 6.7. La complejidad del tiempo es $O(n)$, que es el número de veces que se ejecuta el bucle while. Tenga en cuenta que no necesitamos ningún procesamiento adicional, como verificar la propiedad del árbol o la conectividad como en el algoritmo anterior. La selección de bordes salientes del conjunto de bordes ya incluido en el árbol será útil para formar árboles de expansión mínima, como veremos en el siguiente capítulo.

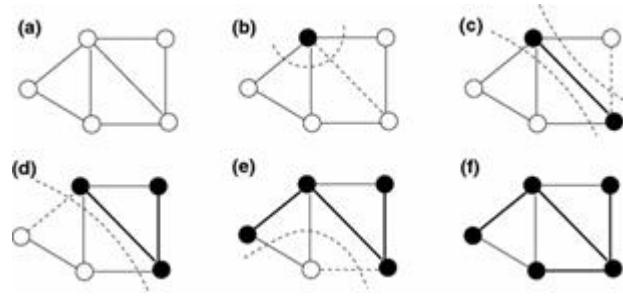


Fig. 6.7 Construcción de un árbol de expansión de un pequeño gráfico de muestra utilizando el concepto de borde saliente. El borde saliente seleccionado en cada iteración se muestra mediante una línea discontinua

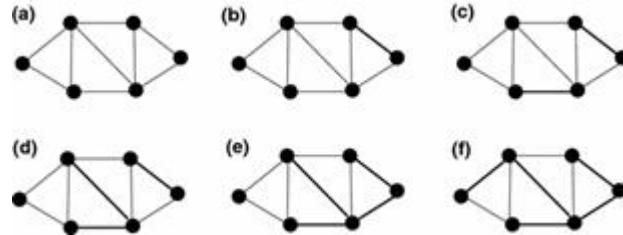


Fig. 6.8 Ejecución del tercer algoritmo de construcción de árbol de expansión en un gráfico de muestra. Los bordes del árbol de expansión se muestran en negrita y cada vértice es un árbol inicialmente

6.2.4.3 *El tercer algoritmo*

Otro enfoque para construir un árbol de expansión es comenzar primero con cada vértice que es un árbol y fusionar dos árboles hasta que esto no sea posible. La idea principal y el argumento de corrección relacionado aquí es que la fusión de dos árboles siempre producirá otro árbol simplemente porque cada árbol anterior a la fusión es acíclico y un borde que se une a ellos no producirá un ciclo con los árboles más pequeños. Tenemos nuevamente $O(n)$ pasos de este algoritmo para formar el árbol de expansión como en el caso del gráfico lineal con la fusión iterativa de cada vértice con el vértice vecino. La operación de este algoritmo en un gráfico de muestra se muestra en la Fig. 6.8.

Este método se presta al procesamiento paralelo ya que son posibles formaciones de subárbol independientes. De hecho, también es adecuado para el procesamiento distribuido en un entorno de red. Cada vértice es un nodo de red y solicita la operación de combinación de un nodo vecino. Debemos tener cuidado de no formar ciclos cuando las solicitudes concurrentes son realizadas por dos nodos desde el mismo subárbol hasta dos nodos que coexisten en otro subárbol vecino. Este problema se puede manejar seleccionando un líder para cada subárbol que controla las solicitudes para fusionar mediante un protocolo adecuado.

6.2.5 Travesías de árboles

La travesía del árbol es el proceso de visitar recursivamente cada nodo del árbol una sola vez. Atravesar árboles en una secuencia determinada es útil en muchas aplicaciones gráficas. Podemos clasificar los recorridos de árboles por el orden en que se visitan los vértices como preorden y postorden para árboles generales.

Previa Traversal

En el recorrido de la preorden de un árbol enraizado, se visita un vértice antes de sus descendientes, como se muestra en el algoritmo 6.4. La complejidad temporal de este recorrido es $O(n)$ para un árbol con n vértices.

Algorithm 6.4 Preorder

```

1: procedure PREORDER(root)
2:   Input: tree T and its root root
3:   Output: preorder traversal of T
4:   if root ≠ Ø then
5:     process(root)
6:     PREORDER(root → left)
7:     PREORDER(root → right)
8:   end if
9: end procedure

```

El recorrido de la preorden del árbol de muestra de la figura 6.9 da como resultado una secuencia de procesamiento de vértice de a , b , c , d , e, f , g , h , i , j , k , l , n , o , m .

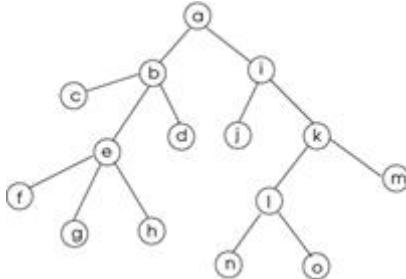


Fig. 6.9 Un árbol de muestra para recorridos de árboles.

orden posterior Transversal

El recorrido posterior al pedido de un árbol implica visitar un vértice de árbol después de visitar a sus descendientes como se muestra en el algoritmo 6.5. El recorrido posterior al pedido del árbol de muestra de la figura 6.5 proporciona la secuencia de procesamiento de vértice de c , f , g, h , e , d , b , j , n , o , l , m , k , i , a . Como cada vértice se visita exactamente una vez, la complejidad del tiempo de este método es O (n).

Algorithm 6.5 Postorder Traversal

```

1: procedure POSTORDER(root)
2:   Input: tree T and its root root
3:   Output: postorder traversal of T
4:   if root ≠ Ø then
5:     POSTORDER(root → left)
6:     POSTORDER(root → right)
7:     process(root)
8:   end if
9: end procedure

```

6.2.6 Árboles binarios

Definición 6.9 (árbol binario) Un árbol binario es un árbol arraigado en el que cada vértice tiene a lo sumo dos hijos y cada hijo de un vértice v es hijo izquierdo o hijo derecho de v .

Un árbol binario completo de profundidad d tiene $2^{d+1} - 1$ vértices que es el orden máximo de cualquier árbol binario. Los árboles binarios se pueden recorrer como preorden o postorden como en los árboles generales. Un método de recorrido adicional para árboles binarios es el recorrido de árbol inorder .

Inorder Traversal

En este modo de recorrido de árbol binario, los vértices de cada subárbol izquierdo de un vértice se procesan primero, el vértice se procesa en segundo lugar y los vértices de subárbol derecho del vértice se procesan finalmente. El pseudocódigo para esta operación se muestra en el algoritmo 6.6. La complejidad temporal de este algoritmo también es O (n).

Algorithm 6.6 Inorder Traversal

```
1: procedure INORDER(root)
2:   Input: tree T and its root root
3:   Output: inorder traversal of T
4:   if root ≠ Ø then
5:     inoder(root → left)
6:     process(root)
7:     inoder(root → right)
8:   end if
9: end procedure
```

Podemos almacenar expresiones aritméticas o lógicas en un árbol binario donde las hojas del árbol son operandos y los nodos internos son operadores unarios o binarios, como se muestra en la Fig. 6.10 . Dicho árbol binario se conoce comúnmente como un árbol de expresión . Este método es utilizado por los compiladores para analizar y evaluar varias expresiones. Podemos ver que el desplazamiento inorder de un árbol de expresiones produce la expresión.

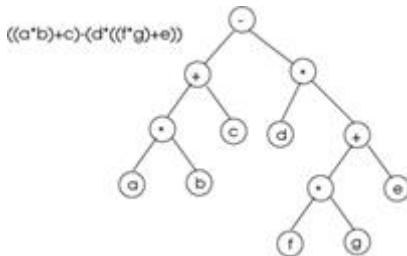


Fig. 6.10 Evaluación de una expresión utilizando un árbol binario.

Un árbol de búsqueda binario (BST) es un árbol binario en el que el valor en cada nodo es mayor que todos los valores en el subárbol izquierdo del nodo y menor que todos los valores en el subárbol derecho del nodo como se muestra en la Fig. 6.11 . Los BST se utilizan en diversas aplicaciones, como la clasificación o la búsqueda de datos.

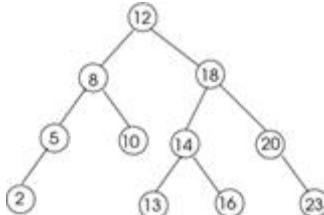


Fig. 6.11 Ejemplo de un árbol binario de búsqueda.

6.2.7 Colas y montones de prioridad

Una cola de prioridad es una estructura de datos para almacenar un conjunto de elementos, cada uno con un valor llamado clave . Esta estructura de datos es útil para implementar varios algoritmos de gráficos, como veremos al revisar los algoritmos de los gráficos ponderados en el Cap. 7 . Una cola de prioridad mínima proporciona las siguientes operaciones.

- \square Insert (S, x) : inserta el elemento de x en el conjunto S .
- \square Mínimo (S) : se devuelve el elemento S con la clave más grande.
- \square ExtractMin (S) : Similar al mínimo (x) pero el elemento x se elimina de S .
- \square DecreaseKey (S, x, k) : el valor de la clave de x se reduce a k .

Cuando estamos tratando con una cola de máxima prioridad, las operaciones en dicha cola son Máximo (S) , ExtractMax (S) y IncreaseKey (S, x, k) que encuentran el elemento máximo de la cola, extraiga este valor, y aumentar el valor de la clave de un elemento a su vez.

Heap como una cola de prioridad

Un montón min binario es un árbol binario completo, excepto posiblemente las hojas en las que las claves de los hijos de cualquier vértice u son mayores o iguales a la clave de u . Por lo tanto, a lo largo de cada ruta desde la raíz, las claves aumentan monótonamente y la raíz tiene el valor de clave mínimo como se muestra en la Fig. 6.12. Podemos tener un montón de binario máximo en el que los valores clave disminuyen desde la raíz hacia abajo.

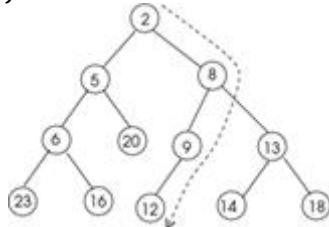


Fig. 6.12 Ejemplo de un montón. La ruta discontinua desde la raíz a una hoja muestra los valores clave que aumentan monótonamente

Todas las operaciones de cola de prioridad definidas anteriormente se pueden implementar con montones como `HeapInsert`, `HeapExtract`, `HeapMin` y `HeapDecrease` en $O(\log n)$ el tiempo y la construcción del montón binario lleva tiempo $O(n)$. Una descripción detallada de la estructura del montón se proporciona en [5].

6.3 Búsqueda en profundidad

La búsqueda en profundidad (DFS) es un método básico para recorrer todos los nodos de un gráfico. Básicamente atraviesa una gráfica al ir lo más profundo posible desde un vértice dado y, por lo tanto, el nombre. Exploramos un camino hasta donde podemos ir marcando los vértices que visitamos a lo largo del camino y cuando no podemos ir más lejos ya que encontramos un vértice con un grado de 1 o todos los vecinos de un vértice son visitados, volvemos a donde viene de. Este método puede ser descrito mejor por una persona en un laberinto que lleva una tiza y una cuerda. Cada habitación (vértice) tiene una cantidad de puertas (vecinos) y la persona que ingresa a la habitación selecciona una de las puertas sin marcar, la marca con la tiza y pasa por esa puerta a otra habitación. Si eso no tiene puertas sin marcar (todos los vecinos visitados) o ninguna otra puerta que no sea de la que vino (un vértice con un borde), ella regresa a donde vino. La cadena se utiliza para realizar un seguimiento de dónde vino. El algoritmo DFS tiene dos versiones recursivas e iterativas que se describen a continuación.

6.3.1 Un algoritmo recursivo

Suponemos que el gráfico está conectado y, si no lo está, el algoritmo DFS se realiza en cada componente del gráfico. La última versión de los algoritmos se llama `DFS_Forest`. Seleccionamos cualquier vértice u del gráfico G y este vértice es la raíz del árbol DFS que se formará. Luego seleccionamos una arista (u, v) que es incidente para u . Este borde es un borde de árboles y se incluye en el árbol DFS T , y el vértice u es el padre del vértice v en T . Procedemos de esta manera seleccionando siempre los bordes no explorados que inciden en vértices que no se visitan. Si se exploran todos los bordes incidentes en un vértice u , lo que significa que todos sus vecinos son visitados, volvemos al parente de u y continuar la búsqueda desde allí. Cuando encontramos un borde inexplorado (u, v) en un vértice v , el borde (u, v) se atraviesa, se incluye en T y u se convierte en el parente de v como en el caso raíz. Si v ha sido visitado anteriormente, y (u, v) no está

explorado, (u, v) no es un borde de árbol y no está incluido en T . El pseudocódigo de un algoritmo que realiza el procedimiento descrito se proporciona en el algoritmo 6.7. También registramos los tiempos de visita para cada vértice; La primera vez de la visita cuando se descubre un vértice es en $d[v]$ y la última vez que regresamos de la llamada recursiva a v se almacena en $f[v]$.

Algorithm 6.7 DFS_Recursive_Forest

```

1: Input:  $G(V, E)$ , directed or undirected graph
2: Output:  $Pred[n]$ ;  $d[n]$ ,  $f[n]$            > place of a vertex in DFS tree and its visit times
3: int  $time \leftarrow 0$ ; boolean  $Marked[1..n]$ 
4: for all  $u \in V$  do                                > initialize
5:    $Marked[u] \leftarrow \text{false}$ 
6:    $Pred[u] \leftarrow \perp$ 
7: end for
8: for all  $u \in V$  do
9:   if  $Marked[u] = \text{false}$  then
10:     $DFS(u)$                                > call for each connected component
11:   end if
12: end for
13:
14: procedure  $DFS(u)$ 
15:    $Marked[u] \leftarrow \text{true}$ 
16:    $time \leftarrow time + 1$ ;  $d[u] \leftarrow time$           > first visit
17:   for all  $(u, v) \in E$  do                      > visit neighbors
18:     if  $Marked[v] = \text{false}$  then
19:        $Pred[v] \leftarrow u$ 
20:        $DFS(v)$ 
21:     end if
22:   end for
23:    $time \leftarrow time + 1$ 
24:    $f[u] \leftarrow time$                            > return visit
25: end procedure

```

La salida del algoritmo es un árbol DFS almacenado en la matriz $Pred$ que muestra los predecesores de cada vértice. Hay algunas cosas a tener en cuenta sobre este algoritmo de la siguiente manera.

- Podemos seleccionar los vecinos del vértice visitado de forma arbitraria o mediante algún ordenamiento en la línea 17. Si los vértices están etiquetados con enteros únicos, el ordenamiento puede ser lineal, desde los identificadores de vértices más pequeños hasta los más grandes. Cuando los vértices están etiquetados con letras, podemos tener una primera opción lexicográficamente. Como consecuencia, el árbol DFS obtenido como resultado de este algoritmo no es único, ya que el orden de la selección de bordes no explorados afecta la estructura de este árbol.
- Si se usa la representación matricial de adyacencia del gráfico, debemos verificar toda la fila que pertenece al vértice u en las líneas 17 y 18 para n un gráfico con n vértices. El uso de la lista de adyacencia significa que la verificación en estas líneas será un máximo de $A(G)$ veces; por lo tanto, podemos deducir que usar la lista de adyacencia es una mejor opción para este algoritmo que usar la matriz de adyacencia.
- El procedimiento DFS finaliza cuando regresa al vértice raíz desde el que se llama. El algoritmo termina cuando el procedimiento DFS se ejecuta en todos los componentes del gráfico.
- El primer tiempo de visita $d[v]$ de un vértice v se denomina número de vértice u de primera profundidad y corresponde al número que se le dio durante el recorrido de la preorden del árbol formado. Veremos que la primera y la última visita de los vértices se pueden usar para varias aplicaciones DFS.
- Si sabemos que el gráfico G está conectado antes de ejecutar el algoritmo, simplemente podemos ejecutar el procedimiento entre las líneas 14 y 25 para que un vértice u encuentre un árbol DFS enraizado en u .

Análisis

Los bordes incluidos en el árbol DFS forman un árbol de expansión dirigido de G . Esto es cierto ya que nunca formamos un ciclo al nunca seleccionar un borde entre dos vértices marcados (líneas 18 y 19) y todos los vértices están marcados y se incluyen en el árbol DFS al final. Necesitamos invocar el procedimiento DFS n veces, uno para cada vértice. Cada activación de este procedimiento implica verificar cada entrada en la matriz de adyacencia por un total de n veces. El tiempo tomado usando la matriz de adyacencia es por lo tanto $\Theta(n^2)$. Utilizar la lista de adyacencia significa verificar cada borde en G dos veces para cada vértice en sus extremos más el tiempo necesario para la inicialización, lo que da como resultado un tiempo total de $\Theta(n + m)$.

Ejemplo

El funcionamiento de DFS_Forest El algoritmo en una gráfica de muestra con dos componentes se muestra en la Fig. 6.13. Podemos ver que se forman dos árboles DFS que se dirigen a los árboles de dos componentes.

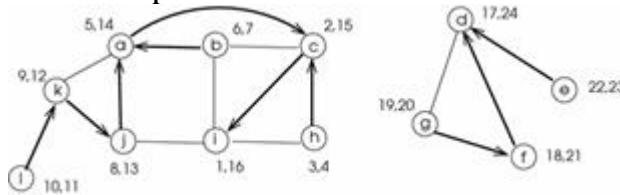


Fig. 6.13 Ejecución de $DFS_Recursive_Forest$ Algoritmo en una muestra de gráfico desconectado con tiempos de primera y última visita mostrados junto a los vértices. Los vértices de origen son i y d en los componentes y las flechas apuntan a los padres en el árbol

Para cualquiera de los dos vértices u y v en el árbol DFS formado con u como antecesor de v ,

$$d[u] < d[v] < f[v] < f[u]$$

ya que el vértice u se descubre antes del vértice v y volvemos al vértice u después de completar el procesamiento del vértice v . Cuando estos vértices están en diferentes árboles o en el mismo árbol pero no tienen una relación parental, entonces tampoco $f[u] < d[v]$ o $f[v] < d[u]$.

Propiedades de borde

La ejecución de DFS en un gráfico dirigido funciona de manera similar, excepto que tenemos que considerar solo los bordes salientes de un vértice al buscar a sus vecinos. El algoritmo DFS que funciona de esta manera divide los bordes de un gráfico dirigido en los siguientes tipos.

- **Aristas del árbol** : estos son los bordes del árbol formado. Un borde (u, v) pertenece al árbol DFS si $DFS(u)$ llama a $DFS(v)$.
- **Bordes posteriores** : cuando el vértice u de un borde (u, v) es un descendiente distinto de sus hijos del vértice v en el árbol, (u, v) es un borde posterior .
- **Bordes delanteros** : cuando el vértice u de un borde (u, v) es un antepasado del vértice v que no sea su parente en el árbol, (u, v) es un borde frontal .
- **Bordes transversales** : cualquier borde que no sea un árbol, la parte posterior o frontal se denomina borde transversal .

También existe la siguiente relación entre el tiempo de descubrimiento d y el tiempo de finalización f de los vértices en un gráfico G , comúnmente denominado teorema de paréntesis . Otros ordenamientos de descubrimiento y tiempos de finalización no son posibles.

- El vértice u es un descendiente de v en el bosque DFS si y solo si [d (u), f (u)] es un subintervalo de [d (v), f (v)].
- Vértice v es un descendiente de u en el bosque DFS si y solo si [d (v), f (v)] es un subintervalo de [d (u), f (u)].
- El vértice u no está relacionado con v en el bosque DFS si y solo si [d (u), f (u)] y [d (v), f (v)] son intervalos disjuntos.

La figura 6.14 muestra estos bordes en un gráfico de muestra. Descubrir un borde posterior en un árbol DFS nos ayuda a descubrir varias propiedades de un gráfico. Modifiquemos el procedimiento DFS en el algoritmo 6.7 para que un gráfico dirigido clasifique estos bordes utilizando los tiempos de descubrimiento y finalización de sus puntos finales descritos anteriormente. Usaremos tres colores para cada vértice en esta implementación: un vértice es blanco cuando no está explorado, es gris cuando se explora pero no está terminado, y es negro cuando está terminado. El color de la matriz se inicializa en blanco para todos los vértices y mantiene el color de cada vértice. El pseudocódigo para el DFS modificado se muestra en el algoritmo 6.8 [5].

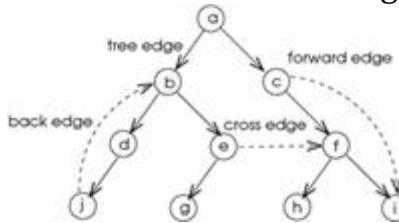


Fig. 6.14 Bordes DFS de un digrafo; (j , b): borde posterior, (c , i): borde delantero, (e , f): borde transversal y todos los demás bordes son bordes de árbol DFS

Algorithm 6.8 DFS2_Edges

```

1: procedure DFS(u)
2:   color[u]  $\leftarrow$  gray
3:   time  $\leftarrow$  time + 1; d[u]  $\leftarrow$  time                                > first visit
4:   for all  $(u, v) \in E$  do
5:     if color[v] = black then
6:       if d[u] < d[v] then
7:         type(u, v)  $\leftarrow$  forward_edge
8:       else type(u, v)  $\leftarrow$  cross_edge
9:       end if
10:      if color[v] = gray then
11:        type(u, v)  $\leftarrow$  back_edge
12:      end if
13:      if color[v] = white then
14:        type(u, v)  $\leftarrow$  tree_edge
15:        DFS(v)
16:      end if
17:    end if
18:    color[u]  $\leftarrow$  black
19:    Pred[v]  $\leftarrow$  u
20:    time  $\leftarrow$  time + 1
21:    f[u]  $\leftarrow$  time                                         > return visit
22:  end for
23: end procedure

```

Buscamos a todos los vecinos de un vértice u que el procedimiento toma como entrada; si encontramos un borde con un punto final v que se completa antes, entonces (*u* , *v*) es un borde delantero si $d[u] < d[v]$ de lo contrario es una arista transversal . El vértice v puede ser gris, lo que significa que está explorado pero no está terminado, en cuyo caso (*u* , *v*) es un borde posterior, de lo contrario es un borde de árbol. En una gráfica no dirigida, no hay bordes delanteros o transversales; por lo tanto, solo tenemos

que comprobar el color del vértice vecino v del vértice u ; si es gris , tenemos un borde posterior y, de lo contrario, v es blanco y (u , v) es un borde de árbol. El tiempo de ejecución para este algoritmo de clasificación de borde es $\Theta(n + m)$ ya que solo hemos agregado operaciones de tiempo constante al procedimiento DFS en el algoritmo 6.7.

6.3.2 Un algoritmo iterativo DFS

Cuando sabemos que la profundidad de recursión de un gráfico es muy grande, por ejemplo, más de unos pocos miles, podemos reemplazar las llamadas recursivas con una pila y obtener un algoritmo DFS no recursivo. El algoritmo DFS iterativo se muestra en el algoritmo 6.9 se inicia desde el vértice fuente y en cada iteración, los vecinos del vértice u bajo consideración son empujados en una pila S . De esta manera, se garantiza la visita a todos los vecinos de u . Una vez que se termina este paso, se salta un vértice w desde S , se marca como visitado y el padre de w se marca como u . Este paso se repite ahora para vértice w. Tenga en cuenta que este algoritmo realiza visitas recursivas de vértices utilizando la pila S que realiza un seguimiento de los vértices vistos pero no procesados. A diferencia del algoritmo DFS recursivo, tenemos el último vértice insertado en la pila que se procesa primero. Tenemos la misma complejidad de tiempo. $\Theta(n + m)$.

Algorithm 6.9 DFS_Iterative

```

1: Input:  $G(V, E)$  and a vertex  $v$ , directed or undirected graph
2: Output:  $Pred[n]$ ;  $d[n]$ ;  $f[n]$             $\triangleright$  place of a vertex in DPS tree and its visit times
3:  $int time \leftarrow 0$ ; boolean  $Marked[n]$ ; stack  $S \leftarrow \emptyset$ 
4: for all  $u \in V$  do                                 $\triangleright$  initialize
5:    $Marked[u] \leftarrow false$ 
6:    $Pred[u] \leftarrow \perp$ 
7: end for
8:  $push(S, v)$                                       $\triangleright$  push source vertex in stack
9: while  $S \neq \emptyset$  do
10:    $u \leftarrow pop(S)$ 
11:   if  $visited[u] \neq true$  then
12:      $visited[u] \leftarrow true$ 
13:      $Pred[u] \leftarrow v$ 
14:     for all  $(u, w) \in E$  do                       $\triangleright$  push neighbors onto stack
15:        $push(S, w)$ 
16:     end for
17:   end if
18: end while

```

6.3.3 DFS paralelo

Debido a la naturaleza de su ejecución, el algoritmo DFS es difícil de parallelizar. Una forma sencilla de proporcionar procesamiento paralelo es dividir el espacio de búsqueda entre los procesadores. Sin embargo, esta asignación estática generalmente resulta en un balance de carga deficiente, ya que el tamaño de los subárboles puede variar significativamente. El espacio de búsqueda en general se forma dinámicamente y es difícil estimarlo de antemano. Luego se puede usar el equilibrio dinámico de carga para el procesamiento DFS paralelo.

Un equilibrio de carga dinámico simple para DFS paralelo puede funcionar de la siguiente manera. Un proceso ¹⁰ Funciona en un espacio de búsqueda dado y cuando termina su trabajo, solicita trabajo de otros procesos. Esto puede ser manejado por un proceso central o de una manera verdaderamente distribuida sin control central. En términos de implementación, todo el espacio de búsqueda puede darse a un solo proceso y todos los demás procesos pueden tener espacios de búsqueda vacíos inicialmente como se describe en [8]. El espacio de búsqueda se divide entre procesos cuando solicitan trabajo.

6.3.4 Algoritmos distribuidos

En una configuración de red, nuestro objetivo es hacer que los nodos de la red cooperen para encontrar el DFS de toda la red. Podemos usar el árbol DFS formado para varias aplicaciones, como encontrar nodos conectados en un entorno distribuido de este tipo. La información del árbol DFS se puede recopilar en la raíz, que luego puede transferir los vértices conectados en la red a una utilidad de administración que puede tomar medidas correctivas si hay nodos desconectados. Hay varios algoritmos para este propósito y revisaremos uno básico que imita el algoritmo secuencial que hemos visto, usando un mensaje especial llamado token. Este mensaje especial proporciona un único punto de ejecución que es el titular del token. Cualquier nodo que posea el token puede ejecutar el algoritmo mientras los otros permanecen inactivos. La ficha sirve a un segundo propósito; contiene los identificadores de los nodos que se visitan para evitar volver a visitarlos.

Este algoritmo llamado *Token_DFS* [6] que funciona utilizando el mismo principio que en el procedimiento DFS se muestra en pseudocódigo en el algoritmo 6.10 [6]. Ahora tenemos una raíznodo que inicia el algoritmo que es donde comenzaríamos el algoritmo secuencial. Un nodo que recibe el token por primera vez registra al remitente como su padre. Luego verifica si tiene un vecino no visitado comparando la lista contenida en el token con sus vecinos. Si tal vecino existe, envía el token a ese nodo. De lo contrario, el token se devuelve al parente, que básicamente está imitando el retorno del procedimiento recursivo en el algoritmo secuencial. La terminación correcta del algoritmo distribuido es un problema fundamental en una configuración distribuida. Cualquier nodo que no sea la raíz termina cuando devuelve el token a su parente. La raíz tiene una terminación diferente, se detiene cuando se le devuelve el token y no tiene otros vecinos no visitados.

Algorithm 6.10 Token_DFS

```

1: int parent ← ⊥
2: set of int children ← {0}, others ← {0}, list ← {0}
3: boolean used[] ← false
4: message types token
5:
6: if i = root then                                ▷ root node starts the algorithm
7:   parent ← i, choose j ∈ N(i)
8:   send token([i]) to j
9: end if
10:
11: while true do
12:   receive token(j, list)
13:   if parent = ⊥ then                            ▷ token received first time
14:     parent ← j
15:   end if
16:   if ∃ j ∈ {N(i) \ [list]} | then            ▷ choose an unvisited node if it exists
17:     choose j ∈ {N(i) \ [list]}
18:     send token(list ∪ {i}) to j
19:   else if i = root then                      ▷ if i is the root and all visited, terminate
20:     exit
21:   else                                         ▷ if all visited and i is not root, return token to parent
22:     send token(list ∪ {i}) to parent
23:   exit                                         ▷ all nodes except root terminate
24: end if
25: end while

```

Análisis

Teorema 6.4 El *Token_DFS* construye correctamente un árbol DFS en $2n - 2$ tiempo usando $2n - 2$ mensajes

Prueba Dado que la operación es básicamente la misma del algoritmo DFS secuencial, el árbol DFS se construirá correctamente, es decir, visitando cada nodo de manera DFS y formando un árbol sin ningún ciclo. El árbol DFS construido tendrá $n - 1$ bordes ya que cualquier árbol con n vértices tiene $n - 1$ bordes, y solo los bordes de este árbol se habrán atravesado dos veces, una vez en cada dirección, dando como resultado un total de $2^{n - 2}$

transferencias de token entre los nodos que resultan en $2n - 2$ mensajes. Los bordes que no son de árbol no se atravesarán ya que siempre buscamos nodos no visitados. Hay una sola actividad en cualquier momento dictada por la posesión del token, y cada transferencia de mensaje toma una sola unidad de tiempo que resulta en $2n - 2$ hora.

Seguimos una ruta muy simple para diseñar un algoritmo distribuido siguiendo la misma lógica que la secuencial. Una buena parte de este algoritmo es que el token se reenvía solo a lo largo de los bordes del árbol. Sin embargo, un problema conocido con este algoritmo es que el tamaño del token depende del orden del gráfico. Suponiendo que necesitamos $\log n$ bits para contener el identificador de un nodo, el tamaño requerido del token sería $n \log n$ lo que significa que tenemos que transferir un mensaje de $O(n \log n)$ tamaño a través $n - 1$ bordes del árbol DFS. Además, la operación es secuencial, ya que existe una actividad única por parte del titular del token en cualquier momento.

Ejemplo

Una operación de este algoritmo en un gráfico de muestra se muestra en la figura 6.15. Podemos ver que el token es transferido 12 veces lo que es $2 \times n - 2$, siendo $n = 7$ en este caso. Este es también el tiempo que toma el algoritmo. El número de bits requeridos en el token es $O(n \log n) = 21$.

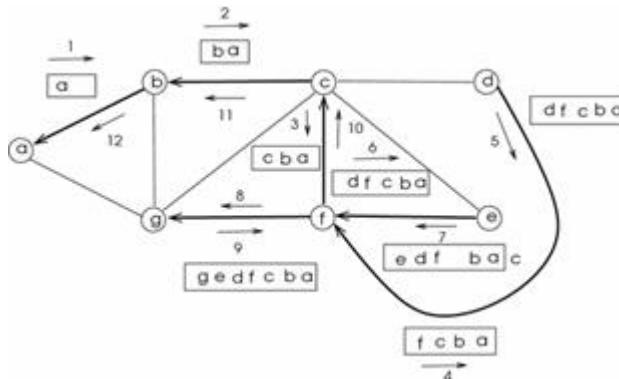


Fig. 6.15 Ejecución de *Dist_BFS*. Algoritmo en una gráfica de muestra. El contenido del token se muestra solo cuando hay un cambio. El árbol dirigido tiene su raíz en el vértice a y las flechas muestran la secuencia de ejecución

Como un intento de proporcionar un algoritmo que use un token como antes pero sin contener los identificadores de los nodos visitados, Awerbuch propuso un algoritmo distribuido para la construcción del árbol DFS [1]. En este algoritmo, un nodo informa a sus vecinos mediante un mensaje especial llamado *vis* que se visita por primera vez. Los vecinos responden con mensajes de *ack*. Los nodos pueden mantener un seguimiento de los vecinos visitados y no visitados a expensas de mensajes adicionales. Este algoritmo requiere mensajes de 4 m en $4n - 2$ hora.

El algoritmo DFS distribuido clásico también implementa un token y funciona con las siguientes reglas [6]. Este algoritmo necesita mensajes de 2 m y 2 m de tiempo para construir un árbol DFS.

Un nodo nunca reenvía el token a través del mismo borde.

- 1.
2. Cualquier otro nodo que no sea la raíz reenvía el token a su padre cuando la primera regla no es aplicable.

3. Un nodo que recibe el token lo envía de vuelta a través del mismo borde si las reglas 1 y 2 lo permiten.

6.3.5 Aplicaciones de DFS

Podemos usar el algoritmo DFS para probar la conectividad de gráficos dirigidos o no dirigidos, para detectar ciclos en gráficos dirigidos o no dirigidos y para el orden topológico.

6.3.5.1 Prueba de conectividad

El algoritmo DFS se puede usar para encontrar los componentes conectados de un gráfico. Si una sola llamada al procedimiento DFS visita todos los vértices de G , entonces está conectado. Como consecuencia, también podemos determinar el número de componentes conectados de G ; esto se puede lograr simplemente agregando un contador para encontrar el número de veces que se llama al procedimiento DFS en la línea 10 del algoritmo 6.7. Si el recuento es 1, solo hay un componente en el gráfico, es decir, el gráfico está conectado.

DFS también fue utilizado por Hopcroft y Karp para la comparación bipartita [9]. Hopcroft y Tarjan propusieron otro uso del método DFS para probar la planaridad de una gráfica en tiempo lineal [10] y Even y Tarjan proporcionaron algoritmos de conectividad de vértice y borde basados en DFS [7]. Una aplicación esencial de DFS es el ordenamiento topológico o la clasificación topológica de un gráfico acíclico dirigido (DAG), como describiremos.

6.3.5.2 Detección de ciclos

Un ciclo en una gráfica es una ruta que comienza y termina en el mismo vértice. Podemos usar el algoritmo DFS para detectar ciclos en gráficos no dirigidos o dirigidos por la observación de que no habrá bordes posteriores, como un borde (u, v) , siendo v el antecesor de u en el árbol en gráficos acíclicos. Podemos hacer uso del siguiente teorema para detectar un borde posterior.

Teorema 6.5 Dado un grafo dirigido o no dirigido $G = (V, E)$, $(u, v) \in E$ es un borde trasero si y solo si $d(v) < d(u) < f(u) < f(v)$.

Prueba Si (u, v) es un borde posterior, conecta el vértice u a su antecesor v ; Por lo tanto, vértice u es un descendiente de vértice v . Por el teorema de paréntesis, el intervalo $[d(u), f(u)]$ es un subintervalo de $[d(v), f(v)]$, por lo que se mantiene la dirección de avance del teorema. Para la dirección inversa, consideremos nuevamente el teorema de paréntesis. Cuando $d(v) < d(u) < f(u) < f(v)$, el vértice u es un descendiente de v . Esto significa que el borde (u, v) conecta un vértice a su antepasado y, por lo tanto, es un borde posterior. \square

Por lo tanto, podemos detectar ciclos en un gráfico dirigido o no dirigido simplemente ejecutando el algoritmo DFS en $O(n+m)$ tiempo para asignar los tiempos de descubrimiento y finalización para cada vértice y luego verificar el tipo de bordes formados en el tiempo $O(m)$ para esta propiedad, lo que resulta en un total de $O(n+m)$ hora. Para detectar los bordes posteriores en línea cuando estamos realizando un DFS, podemos verificar el tiempo de finalización $f(u)$ de un vértice u con el tiempo de descubrimiento de sus vecinos. Si existe un vecino v tal que $d(v) < f(u)$, entonces v es un antepasado de u y (u, v) es un borde posterior. Tenga en cuenta que esta prueba es adecuada para determinar

$d(v) < d(u) < f(u) < f(v)$ ya que sabemos $d(v) < f(v)$ y $d(u) < f(u)$. Para cualquier par de vértices u, v de la gráfica. En el gráfico de ejemplo de la figura 6.13, el borde (h, i) es un borde posterior, ya que el vértice i es un antecesor del vértice h ; $d(h), f(h) = 3, 4$; $d(i), f(i) = 3, 4$ y por lo tanto $d(h) < f(i)$.

Como un enfoque alternativo, podemos usar el esquema de coloración del algoritmo 6.8 donde el blanco no está explorado, el gris se descubre pero no está terminado, y el negro significa que hemos descubierto y regresado de ese vértice. Cuando consideramos un vértice vecino v de un vértice u como en la línea 18 del algoritmo 6.8, verificamos su color. Si es gris, el vértice v tiene que ser un antepasado del vértice u en el árbol DFS y, por lo tanto, (u, v) es un borde posterior que significa que el gráfico G no es acíclico.

6.3.5.3 Orden topológico

El ordenamiento topológico es un proceso importante en los dígrafos donde buscamos un ordenamiento lineal de vértices tal que para cualquier borde (u, v) en el gráfico, u preceda a v en el orden. Supongamos que hay n tareas, algunas de las cuales dependen de otras tareas para comenzar, similares a las que tenemos en un gráfico de dependencia computacional de tareas descrito en la Secta. 4. Para realizar estas tareas, necesitamos organizarlas en el orden en que se muestran sus dependencias. La clasificación topológica en un dígrafo con un ciclo no es posible y, por lo tanto, asumiremos que el dígrafo de entrada no tiene ciclos. Básicamente, si un vértice v puede ser alcanzado desde un vértice u , entonces el vértice u debe tener un orden inferior (Fig. 6.16).

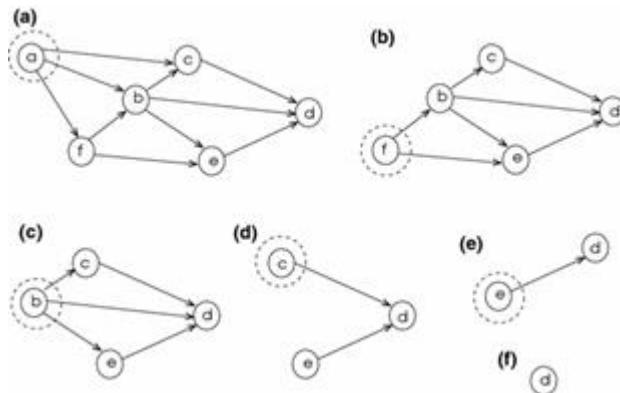


Fig. 6.16 Iteraciones del algoritmo simple para ordenamiento topológico en una gráfica de muestra. El vértice seleccionado sin bordes entrantes en cada iteración se muestra dentro de un círculo discontinuo. El ordenamiento formado es $[a, f, b, c, e, d]$ en el orden de vértices eliminados

Definición 6.10 (orden topológico) Un orden topológico \prec de una gráfica acíclica dirigida $G = (V, E)$ es un orden total del conjunto de vértices V tal que para todos los bordes $(u, v) \in E$ $u \prec v$.

Un algoritmo simple

Podemos tener un algoritmo simple para orden topológico de la siguiente manera. Primero encontramos un vértice v con in grado 0. Siempre hay un vértice de este tipo en un DAG, ya que está libre de bucles. Si hay más de uno de estos vértices, se realiza una selección arbitraria. Este vértice se coloca en la lista de salida ordenada; y v con todos sus bordes salientes se elimina del gráfico. Este proceso se repite hasta que no quedan vértices. La corrección está garantizada, ya que si la eliminación de los bordes salientes de un vértice v colocado en la lista deja un vértice u sin bordes entrantes, lo que sabemos $v \prec u$

. El algoritmo 6.11 muestra esta rutina en pseudocódigo donde tenemos L como la lista de salida ordenada. El funcionamiento de este algoritmo se muestra en la figura 6.16 .

Algorithm 6.11 BFS

```

1: Input :  $G(V, E)$ 
2: Output :  $L$ 
3:  $G' = (V', E') \leftarrow G = (V, E)$                                  $\triangleright$  A directed, connected graph  $G$ 
4: while  $V' \neq \emptyset$  do                                          $\triangleright$  List of ordered elements
5:   find  $v \in V'$  such that  $indeg(v) = 0$                                  $\triangleright$  do until  $V'$  is empty
6:    $L \leftarrow L \cup \{v\}$ 
7:    $G' \leftarrow G' \setminus \{v\} \cup \{\text{all outgoing edges of } v\}$ 
8: end while

```

Análisis

En términos de implementación, podemos tener una matriz A de listas de adyacencia de la gráfica y también una matriz D que muestra los grados de cada vértice como se muestra a continuación para la gráfica de ejemplo de la Fig. 6.17 . Necesitamos verificar cada entrada de D para un valor de 0 y si hay más de un vértice, seleccionamos uno arbitrariamente. Luego, eliminamos este vértice del gráfico insertando un -1 en la matriz de grados D y eliminando este vértice y eliminándolo de la matriz A de las listas de adyacencia de sus vecinos salientes. Este proceso se repite hasta que haya un vértice con un grado de 0.

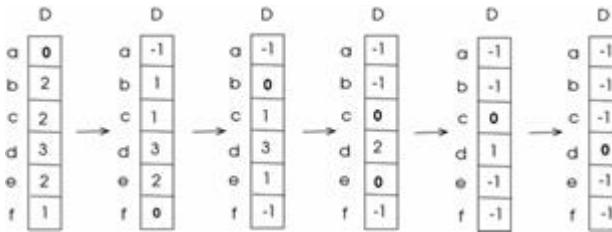


Fig. 6.17 Los valores de la matriz D durante las iteraciones del algoritmo de ordenamiento topológico simple para el gráfico de la Fig. 6.17

Iniciar la matriz de grados D lleva tiempo $O(m)$, buscar cada entrada es tiempo $O(n)$, lo que da como resultado $O(n^2)$ tiempo para toda la matriz D . Luego, reducimos el grado de inminencia de todos sus vecinos en tiempo $O(m)$, lo que resulta en un tiempo de ejecución total de $O(n^2 + m)$ para este algoritmo.

Algoritmo basado en DFS

Podemos hacer uso de las propiedades de borde descubiertas durante el algoritmo DFS recursivo para realizar un orden topológico. Cuando ejecutamos el algoritmo DFS recursivo en un gráfico G para obtener un DAG G' , cada borde (u, v) en G' tiene $f(v) < d(u)$ ya que no hay bordes posteriores en G' . Esto nos proporciona la información necesaria para formar el orden topológico de G : simplemente liste los vértices de G' Desde el tiempo de finalización más grande hasta el más pequeño. Podemos obtener esta lista agregando la identidad del vértice al frente de una lista L cuando terminemos con ella y cuando se complete DFS, la lista L contiene el orden topológico de los vértices. El tiempo total tomado por lo tanto es $\Theta(n + m)$ Como en el DFS recursivo. La Figura 6.18 muestra un árbol DFS obtenido en el mismo gráfico de la Fig. 6.17 y la clasificación de los tiempos de finalización de los vértices proporciona el mismo ordenamiento topológico que el algoritmo anterior en este gráfico.

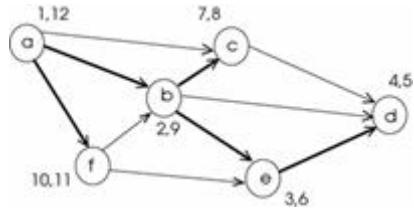


Fig. 6.18 Ordenamiento topológico por DFS en el gráfico de muestra de la Fig. 6.18 . El árbol DFS se muestra en negrita y el primer y último momento de una visita a cada vértice se muestra a su lado. El ordenamiento utilizando los tiempos de la última visita de los vértices encontrados usando este método es $\{a, f, b, c, e, d\}$

Análisis

Teorema 6.6 El algoritmo de ordenamiento topológico basado en DFS proporciona correctamente un tipo topológico de un dígrafo acíclico $G = (V, E)$.

Prueba Tenemos que demostrar que para cualquier borde dirigido $(u, v) \in E$, $f(v) < f(u)$. Para la corrección de este algoritmo. Para cualquier borde (u, v) considerado durante el DFS, el vértice v no puede ser un antepasado del vértice u ya que G recibe un ácido acíclico, por lo tanto $f(v) < f(u)$. En términos de nuestra codificación de colores de vértices, el vértice v no puede ser gris ; puede ser blanco (no explorado) o negro (explorado). Si el vértice v es blanco , se procesará y se volverá negro antes de que u se convierta en negro y, por tanto, $f(v) < f(u)$. Si es negro , esto significa que ya se ha procesado para tener determinado su tiempo de finalización $f(v)$ y el vértice u también tendrá un tiempo de finalización mayor que v en este caso. □

6.4 Búsqueda de amplitud

La idea principal del método de búsqueda de amplitud (BFS) es visitar a todos los vecinos de un vértice antes de visitar otros vértices, y de ahí el nombre. Partiendo del vértice fuente s , primero visitamos todos los vecinos de s en un orden arbitrario. Estos vértices vecinos, $N(s)$, todos tienen una distancia de unidad del vértice s después de la visita. Luego visitamos vecinos de vértices en $N(s)$ que están etiquetados con una distancia de 2 a vértice s . Este proceso continúa hasta que se visitan todos los vértices.

6.4.1 El algoritmo secuencial

Podemos implementar este algoritmo insertando los vértices adyacentes del vértice actualmente visitado en una cola, y luego eliminándolos uno por uno y repitiendo el proceso. Necesitamos mantener un registro de los vértices visitados como en los algoritmos DFS para evitar que un vértice sea visitado nuevamente. El algoritmo 6.12 muestra una forma de implementar el procedimiento descrito.

Algorithm 6.12 BFS

```

1: Input :  $G(V, E), s$             $\triangleright$  undirected, connected graph  $G$  and a source vertex  $s$ 
2: Output :  $D[n]$  and  $Pred[n]$      $\triangleright$  distances and predecessors of vertices in BFS tree
3: for all  $v \in V \setminus \{s\}$  do           $\triangleright$  initialize all vertices except source  $s$ 
4:    $D[v] \leftarrow \infty$ 
5:    $Pred[v] \leftarrow \perp$ 
6: end for
7:  $D[s] \leftarrow 0$                     $\triangleright$  initialize source  $s$ 
8:  $Pred[s] \leftarrow s$ 
9:  $Q \leftarrow s$ 
10: while  $Q \neq \emptyset$  do            $\triangleright$  do until  $Q$  is empty
11:    $v \leftarrow \text{dequeue}(Q)$          $\triangleright$  deque the first element  $v$ 
12:   for all  $(u, v) \in E$  do        $\triangleright$  process all neighbors of  $v$ 
13:     if  $D[u] = \infty$  then
14:        $D[u] \leftarrow D[v] + 1$ 
15:        $Pred[u] \leftarrow v$ 
16:        $\text{enqueue}(Q, u)$ 
17:     end if
18:   end for
19: end while

```

Análisis

La inicialización de distancia y predecesora para vértices lleva tiempo $O(n)$. Cada vértice se pone en cola como máximo una vez y cada borde se explora como máximo dos veces, una por cada vértice que incide en él. Por lo tanto, el tiempo total empleado para construir el árbol BFS es $O(n + m)$.

Ejemplo

La ejecución del algoritmo BFS en un gráfico de muestra se muestra en la Fig. 6.19. El árbol BFS construido muestra las rutas más cortas desde el vértice raíz a .

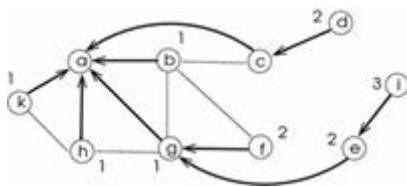


Fig. 6.19 Ejecución del algoritmo BFS en un gráfico de muestra del vértice a . El árbol dirigido tiene sus raíces en a y los niveles de vértices se muestran junto a ellos. Tenga en cuenta que marcamos los bordes de los árboles con flechas para señalar a sus padres a lo largo del camino hacia el vértice de origen

Propiedades

Las propiedades del recorrido del gráfico BFS son las siguientes.

- Un árbol de expansión dirigido del gráfico G se obtiene como resultado de este algoritmo. Actualizamos distancias de vértices con distancias infinitas solo en la línea 13 que no se visitan y, por lo tanto, no se forman ciclos. Tenga en cuenta que no necesitamos una variable para verificar si un vértice está visitado o no como en el algoritmo DFS, ya que el valor de la distancia del infinito muestra que el vértice no está visitado. Todos los vértices serán procesados al final del algoritmo y por lo tanto, la salida es un árbol de expansión de G .
- El orden que seleccionamos y, por lo tanto, encolamos a los vecinos de v en la línea 12 del algoritmo afecta la estructura del árbol BFS obtenido y, por lo tanto, este árbol no es único.
- El algoritmo BFS divide los bordes de un gráfico no dirigido en bordes de árbol y bordes posteriores .

6.4.2 BFS paralelo

Cuando vemos el procedimiento BFS, podemos ver que todos los vértices en el mismo nivel se pueden procesar en paralelo. Por ejemplo, podemos tener un bucle paralelo para explorar todos los vértices vecinos de los vértices en el nivel i para encontrar los vértices

en el nivel $i+1$. Sin embargo, podemos tener una situación en la que dos vértices u y v en el nivel i serán tanto intento de establecer el nivel de un vecino sin explorar vértice w de $i+1$ y establecerse como el padre de w . Sin embargo, esta condición aparentemente carrera no causa ningún problema, ya que no importa qué vértice establece el nivel de w de $i+1$ y también es irrelevante si u o v es el padre de w . Además, debemos sincronizar todos los procesos en el nivel i para asegurarnos de que todos terminen antes de iniciar el procesamiento paralelo a nivel $i+1$ para encontrar vértices a nivel $i+2$.

Este enfoque de nivel sincrónico se implementa en varios algoritmos de BFS paralelos como se describe en [3]. Un enfoque basado en PRAM puede funcionar de manera similar a lo que hemos descrito al proporcionar procesamiento de bucle paralelo por varias unidades de procesamiento y actualizaciones de nivel atómico con sincronización de barrera entre procesamiento de nivel. El tiempo total de ejecución basado en este modelo sería entonces $O(\text{diam}(G))$ ya que el número de niveles no excedería el diámetro del gráfico G . Varios algoritmos paralelos, ya sean de memoria compartida o distribuida, adaptan esta estrategia con un posible equilibrio de carga agregado durante el procesamiento paralelo del ciclo de exploración. En [2] se informa un algoritmo BFS paralelo de grano fino que se ejecuta en la memoria compartida del sistema Cray MTA-2. En un sistema de procesamiento paralelo de memoria distribuida, la partición del gráfico para procesar los nodos es comúnmente perseguida. Un algoritmo paralelo de memoria distribuida que utiliza partición de gráficos 2-D se implementa en BlueGene / L en [12].

6.4.3 Algoritmos distribuidos

Revisamos dos algoritmos distribuidos para formar un árbol BFS en una red: un algoritmo síncrono que funciona en rondas y un algoritmo asíncrono. La terminación debe ser manejada con cuidado en ambos casos.

6.4.3.1 Un algoritmo de BFS síncrono

Nuestro punto de partida será nuevamente el algoritmo BFS secuencial e intentaremos tener su versión distribuida. Esta vez, sin embargo, asumiremos el método del algoritmo distribuido síncrono de iniciador único (SISD). Hay un nodo raíz (supervisor) r que es donde se arraigará el árbol BFS y el algoritmo se ejecuta en rondas síncronas [6]. La raíz amplía el árbol BFS actual. r Con una nueva capa en cada ronda. Los mensajes utilizados en este algoritmo son los siguientes.

- Round : enviado por la raíz r al principio de cada round sobre el árbol parcial r . Cada nodo en r Transmisiones redondas a sus hijos.
- Sonda : Enviada por hojas de T a todos los nodos vecinos excepto el padre.
- Ack : un nodo que no es un árbol responde al mensaje de la sonda mediante un mensaje de acuse de recibo . Marca al remitente de la sonda como su padre y el receptor de ack marca al remitente como uno de sus hijos. Si se reciben más de un mensaje de sondeo simultáneamente, el nodo receptor escoge uno de ellos arbitrariamente.
- Nack : si el destinatario de un mensaje de sonda ya tiene asignado un parent, termina enviando un mensaje nack al remitente .
- Upcast : enviado por hojas de r a sus padres para informar a la ronda ha terminado.

Una vez que se forma una nueva capa, las hojas de la ronda anterior recogen todos los ACK y NACK mensajes e iniciar un convergecast operación que se describe en la Sección. 5.6.3 sobre los bordes de τ . Cuando la raíz recibe mensajes upcast de todos sus hijos, puede comenzar la siguiente ronda. El algoritmo 6.13 muestra la ronda k del algoritmo BFS síncrono distribuido ejecutado por cualquier nodo, excepto la raíz. El conjunto de elementos secundarios es el conjunto de elementos secundarios para los nodos raíz e intermedios; otros son el conjunto de vecinos de un nodo hoja que no son sus hijos y el conjunto recopilado se utiliza para realizar un seguimiento de los niños que han enviado un mensaje de difusión a un nodo intermedio.

Algorithm 6.13 Synchron_BFS

```

1: int parent ← Ø
2: set of int child� ← {Ø}, others ← {Ø}, collected ← {Ø}
3: message types round, probe, ack, nack
4: boolean visited ← false; round_over ← false
5: while ¬round_over do
6:   receive msg(j)
7:   case msg(j).type of
8:     round(k):
9:       if leaf_node then
10:         send probe(k) to  $N(i) \setminus \{j\}$                                 ▷ if leaf check neibours
11:       else
12:         send probe(k) to child�
13:         if ¬visited then
14:           parent ← j; visited ← true
15:           send ack(k) to j                                         ▷ inform parent I am child
16:         else
17:           send nack(l) to j                                     ▷ else reject sender
18:           child� ← child� ∪ {j}                                ▷ include sender in children
19:           others ← others ∪ {j}                                ▷ include sender in unrelated
20:     upcast(k):
21:       collected ← collected ∪ {j}                                ▷ collect upcast signals
22:     if (leaf_node ∧ (child� ∪ others) =  $N(i)$ ) ∨ (¬leaf_node ∧ (collected = child�))
23:     then
24:       send upcast to parent
25:       round_over ← true; collected ← {Ø}
26:   end if
27: end while

```

Podemos ver que este algoritmo funciona de forma asincrónica en una ronda síncrona. No prestamos atención al orden de los mensajes recibidos, aunque conocemos cada nodo en τ primero recibirá el mensaje redondo y actuará de manera diferente dependiendo de si es una hoja o un nodo intermedio. Un nodo hoja busca vecinos para ser incluidos en la siguiente capa y un nodo intermedio simplemente actúa como una puerta de enlace al enviar el mensaje redondo a sus hijos. Al finalizar la ronda, un nodo intermedio recopila los mensajes de upcast de sus hijos y envía un mensaje de upcast a su padre. Verificamos si un nodo de hoja ha recibido mensajes de ack o nack de todos sus vecinos; o un nodo que no está en la hoja ha recibido mensajes upcast de todos sus hijos para decidir si la ronda ha terminado. Cuando esto se decide, un upcast mensaje se envía a los padres.

Al igual que con todos los algoritmos distribuidos, se necesita la detección de terminación. En otras palabras, la raíz debe saber cuántas rondas tiene que iniciar como máximo. Podemos ver que el número de rondas necesarias es la distancia más lejana entre la raíz y cualquier nodo de la red, que es el diámetro del gráfico de la red y, por lo tanto, la raíz debe conocer el diámetro antes de ejecutar el algoritmo. Desafortunadamente, esto crea otro problema; sin embargo, una inspección detallada del algoritmo puede proporcionar una salida más fácil y elegante de la siguiente manera. Un nodo de hoja l en el gráfico termina cuando tiene un grado de unidad o todos sus vecinos lo han rechazado, lo que significa que son hijos de otros nodos de árbol intermedios. Esto significa que no hay razón para l a seguir, ya que no tiene hijos a transferir Mensajes redondos o upcast ni ningún vecino a sondear. Podemos elaborar sobre esta condición y cuando esto sucede, un especial de terminar mensaje puede ser convergecast hacia la raíz. Cuando todos los hijos de un nodo intermedio v envían un mensaje de terminación , el nodo v termina y envía

un mensaje de terminación a su padre y la raíz termina cuando todos sus hijos reciben un mensaje de terminación (consulte el Ejemplo 10).

Ejemplo

Un ejemplo de operación de este algoritmo se muestra en la Fig. 6.20 donde el nodo raíz g inicia el algoritmo enviando mensajes de sondeo a sus vecinos a y f en la primera ronda que responden mediante mensajes de acuse de recibo y los bordes (g, a) y (g, f) convertirse en aristas del árbol BFS. Tenga en cuenta que los nodos a y f se envían mensajes de sondeo entre sí en la siguiente ronda (ronda 2) que se rechazan. El número de rondas iniciadas por el nodo g es 4, que es el diámetro del gráfico.

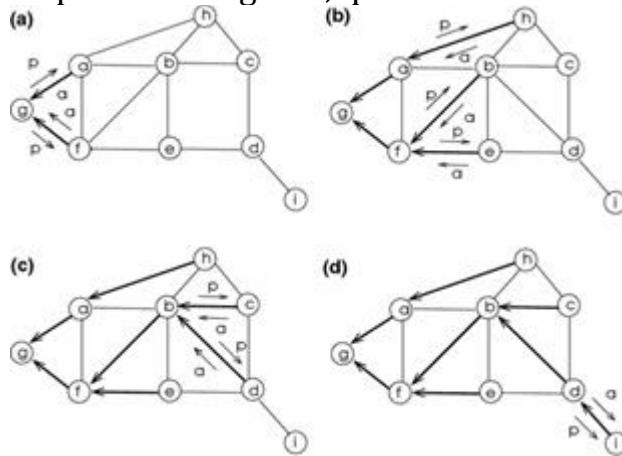


Fig. 6.20 Ejecución del algoritmo 6.13 en un gráfico de muestra para cuatro rondas. Solo se muestran los mensajes de sondeo (p) que se confirman con ack (a) mensajes

Análisis

Teorema 6.7 Algoritmo $Synchon_BFS$ construye correctamente un árbol BFS en $O(diam^2(G))$ tiempo usando $O(n \cdot diam(G) + m)$ mensajes

Prueba En cada paso del algoritmo, solo las hojas en la capa k del árbol parcial formado enviarán mensajes de sondeo para formar la capa $k+1$. Hojas que estarán agrandando el árbol parcial de BFS de una capa. Por lo tanto, la propiedad del árbol BFS se obedece para formar el árbol BFS final.

En cada paso k del algoritmo, el tiempo empleado será proporcional al nivel k actual, ya que los mensajes se transmiten y converger a través de k capas. Podemos tener a lo sumo niveles de $diam(G)$ y, por lo tanto, totalizar

$$\sum_{k=1}^{diam(G)} k = O(diam^2(G))$$

Un análisis de la complejidad del mensaje es el siguiente. Habrá un máximo de n mensajes de sincronización en cada ronda para un total de $O(n \cdot diam(G))$ ya que habrá como máximo diámetros (G). Además, cada borde será atravesado a lo sumo dos veces por pares de mensajes sonda / ack o sonda / nack, lo que dará como resultado mensajes $O(m)$ para consultas. El número total de mensajes necesarios será la suma de estos como $O(n \cdot diam(G) + m)$. \square

6.4.3.2 Un algoritmo asíncrono de BFS

Intentaremos proporcionar una versión distribuida del algoritmo dinámico Bellman-Ford descrito en la Sección. 3.9.3.2 . Este algoritmo proporcionó distancias más cortas en un gráfico ponderado; sin embargo, lo utilizaremos para un gráfico no ponderado con el mismo razonamiento que en [6 , 11]. La versión distribuida es un algoritmo asíncrono de una sola fuente y, por lo tanto, no sabemos el orden de los mensajes que se recibirán. Por lo tanto, necesitamos modificar las distancias de los nodos al nodo de origen en cada mensaje entrante.

Tenemos tres niveles de mensajes distintos (k), ack (k) y nack (k). El nodo raíz inicia el algoritmo enviando nivel(0) mensaje a sus vecinos. Todos los nodos inicializan su distancia a la raíz como infinito primero y cualquier nodo que reciba este mensaje compara su valor de distancia con la raíz con el valor contenido en el mensaje. Si la ruta a la raíz a través del vecino que envía el mensaje es más corta, el vecino que lo envía se asigna como padre y la distancia se modifica como se muestra en el algoritmo 6.14. La distancia modificada se transmite a los vecinos para que también puedan actualizar sus distancias. Esto es necesario ya que la ruta recién encontrada puede afectar las distancias más cortas de los vecinos a la raíz. Tenemos los conjuntos childs y otros como en el algoritmo síncrono, ya que necesitamos un nodo para estar al tanto de los vecinos que son hijos y que no están relacionados.

Algorithm 6.14 Asynchron_BFS

```

1: int parent ← Ø, my_layer ← ∞, count=1
2: set of int childs← [Ø], neighbors ← [Ø], others ← [Ø]
3: message types level, ack, nack
4: if  $i = \text{root}$  then
5:   send level(0) to  $N(i)$                                  $\triangleright$  Only root executes this part
6: end if
7: while count ≤ diam( $G$ ) do
8:   receive msg( $j$ )
9:   case msg( $j$ ).type of
10:    layer( $l$ ) :      if my_layer >  $l + 1$  then            $\triangleright$  update my distance
11:      parent ←  $j$ 
12:      my_layer ←  $l + 1$ 
13:      send ack( $l$ ) to  $j$                                  $\triangleright$  inform parent i am child
14:      send my_layer to  $N(i) \setminus \{j\}$              $\triangleright$  inform neighbors of new level
15:    else
16:      send nack( $l$ ) to  $j$                                  $\triangleright$  else reject sender
17:    ack( $l$ ) :      childs ← childs ∪  $\{j\}$            $\triangleright$  include sender in children
18:    nack( $l$ ) :      others ← others ∪  $\{j\}$            $\triangleright$  include sender in unrelated
19:   count ← count+1
20: end while

```

Se puede ver que este proceso eventualmente construye un árbol BFS a partir de la raíz. La condición de terminación sería la de desplazamiento de la trayectoria más corta más larga entre dos nodos cualesquiera que sería el diámetro de la gráfica G . Por lo tanto, cada nodo debe esperar un máximo de diam (G) de los mensajes de la red. A diferencia del algoritmo síncrono, esta vez no tenemos una solución fácil para la terminación ya que no conocemos el diámetro en priori . Podemos incluir un campo de tiempo de vida en cada mensaje que se inicializa a un límite superior del valor del diámetro y disminuye en cada recepción en un nodo. Cuando este campo se convierte en cero, el mensaje ya no se transmite a los vecinos.

Teorema 6.8 Algoritmo *Asynchron_BFS* construye correctamente un árbol BFS en tiempo $O (\text{diam} (G))$ usando mensajes $O (nm)$.

Proof Después de los pasos diam (G), todos los nodos habrán recibido $\text{layer}(\text{diam}(G) - 1)$ Mensaje y establecerá su distancia a este valor y, por lo tanto, se construirá el árbol BFS. El tiempo necesario es el diámetro de la red para alcanzar el nodo más alejado del nodo raíz, por lo

que la complejidad del tiempo es $O(\text{diam}(G))$. El camino más largo en la red tendrá una longitud de $n-1$ y un nodo que tenga este valor por primera vez puede necesitar cambiarlo $n-2$ veces y enviará a lo sumo $n \cdot \deg(v)$ mensajes que resultan en el número total de mensajes a continuación [11].

$$\sum_{v \in V} n \cdot \deg(v) = O(nm)$$

□

6.4.4 Aplicaciones de BFS

Podemos averiguar si un gráfico G está conectado o no usando este algoritmo como en el algoritmo DFS. Si todos los vértices de G se procesan al final, entonces está conectado. Este método también proporciona la ruta más corta desde el vértice raíz a todos los demás en un modo no dirigido. El algoritmo BFS en un gráfico simple no ponderado proporciona la distancia entre un vértice v y el vértice de origen; esta distancia es simplemente el nivel de v en el árbol BFS formado.

6.4.4.1 Prueba de bipartitismo

Un gráfico $G(V, E)$ es bipartito si su conjunto de vértices V se puede dividir en dos subconjuntos V_1 y V_2 tal que cada $(u, v) \in E$ tiene un punto final en V_1 y el otro punto final en V_2 . En otras palabras, no hay bordes entre dos vértices en V_1 y sin bordes entre dos vértices en V_2 . Podemos usar el algoritmo BFS para verificar si una gráfica G es bipartita o no de la siguiente manera. Un borde que une dos capas del árbol BFS significa que G tiene un ciclo de longitud impar y, por lo tanto, no puede ser bipartito. Damos el mismo color a cada nodo de G en la misma capa descubierta por el algoritmo BFS. Claramente, podemos colorear todos los nodos usando dos colores, por ejemplo, blanco y gris. Por lo tanto, si G es bipartito, no habrá ningún borde de G que tenga dos vértices del mismo color en sus puntos finales. Específicamente, tenemos los siguientes pasos de este algoritmo.

Entrada : $G = (V, E)$

- 1.
2. Resultado : Evaluar G como bipartito o no bipartito
3. Seleccionar $s \in V$ y establecer $\text{color}(s) \leftarrow \text{white}$
4. Ejecutar algoritmo BFS modificado a partir de vértice $s \in V$ como sigue.

Colorea un vértice v en el nivel i con el color gris si vértices en el nivel $(i-1)$ son blancos A la inversa color v blanco si nivel $(i-1)$ Los vértices son grises.

a.
segundo. Dejar $\text{color}(v) = c$ para un vértice v coloreado como el anterior.

do. Si $\exists u \in N(v)$ tal que $\text{color}(v) = \text{color}(u)$

re. Salida " G no es bipartita". Detener.

5. Si todos los vértices están coloreados correctamente, entonces la salida " G es bipartita"

La Figura 6.21 muestra un gráfico de muestra dividido en dos conjuntos por el algoritmo BFS y podemos ver que no es bipartito ya que hay un borde que une dos vértices del mismo color.

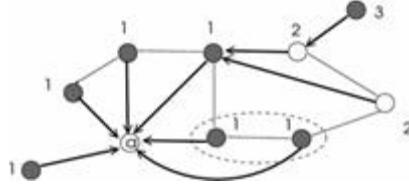


Fig. 6.21 Un gráfico que está dividido en vértices blanco y gris por el algoritmo BFS de un vértice fuente a . Los niveles de vértices se muestran junto a ellos. El borde encerrado en la elipse discontinua está entre dos vértices que son grises, por lo tanto, este gráfico no es bipartito

Este algoritmo funciona correctamente con el siguiente razonamiento para mostrar que solo uno de los casos es válido.

- Si no hay un borde entre dos vértices en la misma capa, los vértices en las capas adyacentes se colorearán con colores opuestos. Por lo tanto, G es bipartito en este caso.
- Supongamos que hay un borde (u, v) entre los vértices u y v de la misma capa L_j . Sea w el antepasado menos común de u y v en el árbol BFS T en la capa L_i . Entonces, $w - u = v - w$ Es un ciclo de la gráfica que tiene longitud $2(j - i) + 1$ que es un ciclo impar. Por lo tanto, G no es bipartito.

BFS se logra en $\mathcal{O}(n + m)$ El tiempo y la exploración de los bordes se pueden realizar en tiempo $O(m)$, lo que resulta en un tiempo total de $\mathcal{O}(n + m)$ para este algoritmo.

6.5 Notas del capítulo

En la primera parte de este capítulo describimos la estructura de árbol en gráficos. Los árboles tienen numerosas implementaciones en informática y en aplicaciones de la vida real. Revisamos los algoritmos para construir árboles de expansión en una gráfica y luego los algoritmos de recorrido de árboles.

En la segunda parte del capítulo, revisamos dos métodos de recorrido de gráficos fundamentales: DFS y BFS. Vimos que DFS se puede implementar como un algoritmo recursivo efectivo con $\mathcal{O}(n + m)$ complejidad de tiempo y también tiene una versión iterativa que usa una pila con la misma complejidad de tiempo. Puede usarse para probar la conectividad y para encontrar el número de componentes de un gráfico. Una implementación importante de DFS es encontrar el orden topológico de un gráfico acíclico dirigido en el que los vértices tienen relaciones de precedencia. El algoritmo DFS es difícil de paralelizar debido a su naturaleza secuencial y dependiente de la ejecución entre cada paso de algoritmo. Sin embargo, el método de contracción del gráfico en el que obtuvimos un gráfico más grueso de un paso anterior se puede utilizar para la construcción de DFS en paralelo. La construcción de árbol DFS distribuido implica nodos de una red de comunicación que cooperan para construir este árbol. Podemos convertir el algoritmo DFS secuencial a uno distribuido mediante un mensaje especial llamado tokenentre los

nodos. Cualquier nodo que posea el token puede ejecutarse, y por lo tanto, de hecho, tenemos un algoritmo secuencial que se ejecuta de manera distribuida. Hay varios otros algoritmos DFS distribuidos que logran un mejor paralelismo a expensas de un mayor número de mensajes, como hemos revisado.

El algoritmo BFS visita los vértices de un gráfico mediante la búsqueda de capa por capa y se puede usar para encontrar distancias desde un vértice de origen a todos los otros vértices en un gráfico no dirigido y no ponderado. Para un gráfico ponderado, necesitamos modificar este algoritmo para encontrar distancias como veremos en el siguiente capítulo. El algoritmo BFS se puede usar para probar la bipartidad de un gráfico no dirigido como vimos. La versión paralela del algoritmo BFS usa la contracción del gráfico como en el algoritmo DFS paralelo. Hay pocos algoritmos BFS distribuidos y uno de ellos funciona de manera síncrona en rondas bajo el control de un nodo especial llamado supervisor. Este nodo amplía el árbol BFS capa por capa en cada ronda y, de hecho, imita el algoritmo BFS secuencial en una configuración distribuida. Además de resolver problemas explícitos como la conectividad, el orden topológico y la bipartida , estos dos métodos básicos de recorridos de gráficos proporcionan bloques de construcción de varios algoritmos de gráficos más complejos, como veremos. Las implementaciones de estos algoritmos en gráficos dirigidos son similares, y deberíamos considerar solo los bordes salientes de un vértice en un dígrafo.

Ceremonias

Escriba el pseudocódigo del algoritmo de búsqueda del centro de árbol recursivo de la secta. [6.2](#) y muestre la ejecución paso a paso de este algoritmo en el árbol de muestra representado en la Fig. [6.22](#) .

1. Construya un posible árbol de expansión del gráfico representado en la Fig. [6.23](#) usando el segundo algoritmo de árbol de expansión de la Secta. [6.2.4.2](#) .
2. Diseñe y forme el pseudocódigo de un algoritmo distribuido que forma un árbol de expansión basado en el tercer algoritmo de la Secta. [6.2.4.3](#) para la construcción del árbol de expansión. Muestre una posible ejecución de este algoritmo en el gráfico de red que se muestra en la Fig. [6.24](#) .
3. Calcule un posible árbol DFS arraigado en el vértice a en el dígrafo de la figura [6.25](#) mostrando los tiempos de descubrimiento y finalización de cada vértice. Muestre también los bordes de los árboles, los bordes delanteros, los bordes traseros y los bordes transversales en el gráfico.
4. El algoritmo DFS basado en token se ejecutará en el gráfico de muestra de la figura [6.26](#) . Elabore un posible árbol DFS enraizado en el vértice a . Muestre las iteraciones del algoritmo en este gráfico con el contenido del token.
5. Escriba el pseudocódigo del algoritmo de detección de ciclos basado en DFS que usa tiempos de descubrimiento y tiempos de finalización de vértices. Muestre la ejecución paso a paso de este algoritmo en la gráfica de la figura [6.27](#) .
6. Calcule el orden topológico de los vértices en el DAG de la figura [6.28](#) utilizando el algoritmo simple y el algoritmo basado en DFS.
7. Encuentre el árbol BFS enraizado en el vértice g en la Fig. [6.29](#) mostrando los niveles para cada vértice.
8. Diseñar el algoritmo BFS síncrono distribuido de la Sect. [6.13](#) (Algoritmo 6.13) con FSMs. Dibuje el diagrama FSM y escriba el pseudocódigo para este algoritmo.
9. Modifique el algoritmo 6.13 de modo que se use la terminación que utiliza un mensaje de terminación especial upcast by leaves.

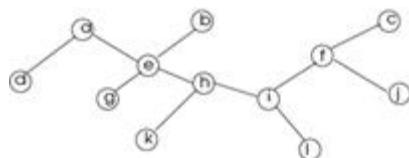


Fig. 6.22 Gráfico de muestra para el Ejercicio 1

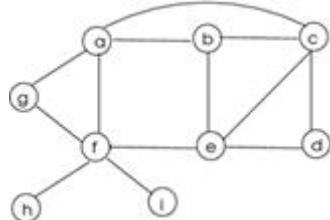


Fig. 6.23 Gráfico de muestra para el ejercicio 2

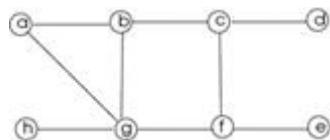


Fig. 6.24 Gráfico de muestra para el Ejercicio 3

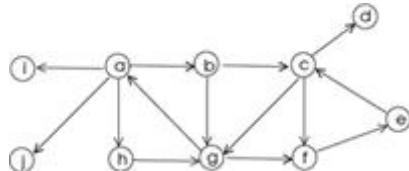


Fig. 6.25 Gráfico de muestra para el Ejercicio 4

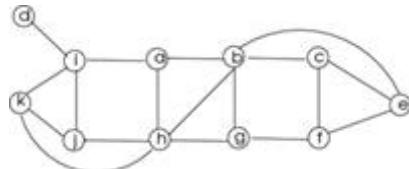


Fig. 6.26 Gráfico de muestra para el ejercicio 5

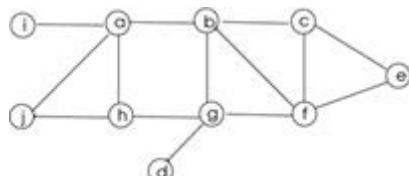


Fig. 6.27 Gráfico de muestra para el ejercicio 6

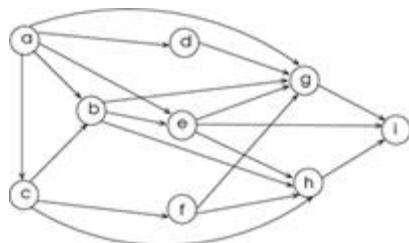


Fig. 6.28 DAG para el ejercicio 7

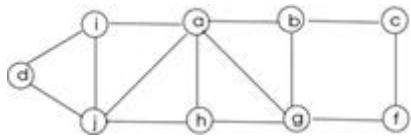


Fig. 6.29 Gráfico de muestra para el ejercicio 8

Referencias

1. Awerbuch A (1985) Un nuevo algoritmo de búsqueda de profundidad distribuida. *Inf Proceso Lett* 20: 147150
[Referencia cruzada](#)
2. Bader DA, Madduri K (2006) Diseño de algoritmos para multiproceso primero en amplitud búsqueda y st -Conectividad en el Cray MTA-2. En: Actas de la 35^a conferencia internacional sobre procesamiento paralelo (ICPP 2006), pp 523–530
3. Buluc A, Madduri K (2011) Búsqueda paralela de amplitud en sistemas de memoria distribuida. En: Conferencia internacional para computación de alto rendimiento, redes, almacenamiento y análisis (SC'11), Artículo 65
4. Cayley A (1857) Sobre la teoría de las formas analíticas llamadas árboles. *Philos Mag* 4 (13): 172176
5. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introducción a los algoritmos, 3^a ed . MIT Press, Cambridge, Capítulo 22
6. Erciyes K (2013) Algoritmos de gráficos distribuidos para redes informáticas, Chaps. 4 y 5. Serie de comunicaciones y redes informáticas Springer. Springer, Berlín (2013). ISBN-10: 1447151720
[Referencia cruzada](#)
7. Even S , Tarjan RE (1975) Flujo de red y conectividad de gráficos de prueba. *SIAM J Comput* 4 (4): 507–518
[MathSciNet Crossref](#)
8. Grama A, Gupta A, Karypis G, Kumar V (2003) Introducción a la computación paralela, 2^a ed . Capítulo 11. Addison Wesley, Boston
9. Hopcroft JE, Karp RM (1973) An $n^{5/2}$ Algoritmo para máxima coincidencia en gráficos bipartitos. *SIAM J Comput* 2 (4): 225-231
[MathSciNet Crossref](#)
10. Hopcroft JE, Karp RM (1974) Pruebas de planaridad eficientes. *J ACM* 21 (4): 549–568
[MathSciNet Crossref](#)
11. Peleg D (2000) Computación distribuida: un enfoque sensible a la localidad. Monografías de SIAM sobre matemáticas discretas y aplicaciones, Capítulo 5
12. Yoo A, Chow E, Henderson K, McLendon W, Hendrickson B, Catalyuurek UV (2005) Un algoritmo de búsqueda escalable en paralelo de amplitud en primer lugar en BlueGene / L. En: Actas de la conferencia ACM / IEEE sobre computación de alto rendimiento (SC2005)

7. Gráficos ponderados

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Un gráfico ponderado puede tener pesos asociados con sus bordes o sus vértices. El peso en un borde generalmente denota el costo de atravesar ese borde y los pesos de un vértice comúnmente muestran su capacidad para realizar alguna función. En este capítulo, revisamos los algoritmos secuenciales, paralelos y distribuidos para los gráficos ponderados para dos tareas específicas; el problema del árbol de expansión mínimo y el problema de la ruta más corta.

7.1 Introducción

Una gráfica puede tener pesos asociados con sus bordes o sus vértices. El peso en un borde generalmente denota el costo de atravesar ese borde y los pesos de un vértice comúnmente muestran su capacidad para realizar alguna función. Nuestro objetivo en este capítulo es revisar algoritmos para gráficos ponderados para dos tareas específicas; el problema del árbol de expansión mínimo y el problema de la ruta más corta.

Un árbol es un gráfico conectado sin ciclos y un árbol de expansión de un gráfico conectado es un árbol que incluye todos los nodos del gráfico, como vimos en el capítulo anterior. Un árbol de expansión mínima(MST) de un gráfico ponderado, no dirigido y conectado es el árbol de expansión con el costo total mínimo de bordes entre todos los árboles de esa gráfica. Puede haber más de un MST en un gráfico si los pesos de los bordes no son distintos. Los MST encuentran una amplia gama de aplicaciones, como la conexión de varias ciudades, componentes u otros objetos. En general, nuestro objetivo en la búsqueda de un MST de un gráfico es utilizar una cantidad mínima de carreteras, cables o cualquier otro medio de conexión para conectar los objetos en cuestión. Los MST también se utilizan para agrupar grandes redes que constan de decenas de miles de nodos y cientos de miles de bordes, como las redes biológicas. La eliminación de una serie de bordes de mayor peso da como resultado agrupaciones en dichas redes.

Revisamos los algoritmos secuenciales, paralelos y distribuidos para la construcción de MST en la primera parte de este capítulo. En la segunda parte de este capítulo, observamos los algoritmos para encontrar las rutas más cortas entre los vértices de un gráfico. Este problema tiene muchos usos prácticos, especialmente en redes de computadoras cuando queremos transferir un paquete de datos entre los dos nodos de una red con costos mínimos. Describimos algoritmos secuenciales, paralelos y distribuidos para encontrar

rutas más cortas entre un vértice y todos los demás vértices, y entre cada par de vértices en una gráfica.

7.2 Árboles de expansión mínimos

En esta sección, primero describiremos los cuatro algoritmos secuenciales principales para construir MST de gráficos ponderados. Luego investigaremos formas de obtener algoritmos paralelos de estos algoritmos, seguidos de la ilustración de un algoritmo distribuido que se puede usar para encontrar el MST de una red de computadoras. También consideraremos la conversión entre algoritmos paralelos y distribuidos para este problema.

7.2.1 Antecedentes

Dado un gráfico ponderado, no dirigido y conectado $G = (V, E, w)$, estamos buscando el MST $T \subseteq G$ de tal manera que $w(T)$ dada a continuación se minimiza.

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

(7.1)

En la búsqueda de una solución a este problema, consideraremos algunas heurísticas aparentemente razonables. En primer lugar, no queremos bordes pesados en el MST e intentamos incluir tantos bordes claros como sea posible. También debemos prevenir los ciclos ya que se requiere un árbol. Por último, los puentes de una gráfica deben incluirse en la MST, ya que la exclusión de estos bordes deja la MST desconectada. Dos reglas definidas a continuación ayudarán a formar MSTs.

Teorema 3 (propiedad Cortar) Considere un gráfico ponderado, no dirigido y conectado $G = (V, E, w)$. Vamos A ser un subconjunto de los bordes que se contiene en algunos MST de G. Para cualquier corte $(S, V \setminus S)$ de G que no tiene bordes de A que crucen el corte, sea (u, v) el borde de menor peso a través de este corte de manera que $u \in S$ y $v \notin S$. A continuación, el borde (u, v) está contenido en algunos MST de G. Si los pesos de borde de G son distintos, entonces hay un MST único T^* de G que contiene el borde (u, v) .

Prueba Considere una MST T que contenga el conjunto A y no contenga el borde (u, v) . Luego debe haber una ruta p que conecta el vértice u al vértice v, ya que la MST T debe cubrir todos los vértices. Combinemos el borde (u, v) con la trayectoria de p para formar un ciclo C en G. El borde (u, v) está a través del corte, lo que significa que debe haber al menos un borde $(w, z) \in C$ que pasa por el corte. Ahora reemplazamos (w, z) con (u, v) para formar un nuevo árbol $T' = T \cup \{(u, v)\} - \{w, z\}$. T' es un árbol de expansión de G ya que agregamos un borde y eliminamos un borde que resulta en $n-1$ bordes. Además, $w(T') = w(T) + w(u, v) - w(w, z) \leq w(T)$ ya que $w(u, v) \leq w(w, z)$ lo que significa T' que contiene el borde (u, v) es un MST de G. \square

La propiedad de corte es útil para formar un MST de un gráfico. Cualquier borde de menor peso a través de cualquier corte del gráfico puede incluirse en el MST hasta que tengamos

$n - 1$ bordes que significa que el árbol formado es un MST. La propiedad de ciclo también es una característica útil de un MST.

Teorema 7.1 (propiedad de ciclo) Deje C ser cualquier ciclo en G y (u, v) ser el borde máximo de peso en C . No hay MST de G que contenga (u, v) .

Prueba Sea T una MST de G y suponga lo contrario que T contiene (u, v) . Eliminar (u, v) de T da como resultado dos subgrafos con vértices V_T y $V - V_T$. El ciclo C tiene otra ventaja. $(w, z) \neq (u, v)$ que tiene exactamente un punto final en V_T y $w(w, z) < w(u, v)$ desde borde (u, v) es el borde peso máximo de C . Formar un nuevo arbol $T' = T - \{(u, v)\} \cup \{(w, z)\}$. El peso total de T' es menor que el peso total de T que es una contradicción. \square

Teorema 7.2 Deje $G = (V, E, w)$ ser un gráfico conectado, ponderado, no dirigido. Si los pesos de borde de G son distintos, entonces G tiene un MST único.

Prueba Let T ser un MST de G . Para cada borde $(u, v) \in T$, el árbol $T' = (V, T \setminus \{(u, v)\})$ tiene dos componentes conexas dicen P y Q . El borde (u, v) es el único borde de T a través del corte entre P y Q y es el borde único con menor peso entre estos dos conjuntos por la propiedad de corte y porque los bordes de G tienen pesos distintos. Por lo tanto, cada MST de G debe contener (u, v) y si tenemos en cuenta todos los bordes de T , cada MST de G debe contener todos los bordes del T . Por lo tanto cada MST es igual a T . \square

7.2.2 Algoritmos secuenciales

Podemos tener un algoritmo genérico para construir el MST de un gráfico G de la siguiente manera. Comenzamos con un MST $T = \emptyset$ de G y siempre añadir bordes seguras a T que debería estar en el MST de G . Los algoritmos fundamentales para construir el MST de un gráfico ponderado con este método se deben a Prim, Kruskal y Boruvka, como veremos a continuación.

7.2.2.1 Algoritmo de Prim

Este algoritmo fue propuesto inicialmente por Jarnik y luego por Prim para encontrar MST de un gráfico $G = (V, E, w)$ conectado, ponderado, dirigido o no dirigido [16]. La idea de este algoritmo es seleccionar siempre un borde seguro mediante la propiedad de corte del Teorema 3 para ser incluido en el conjunto A que tiene un subconjunto de bordes de un MST de G . Este algoritmo asume que los bordes en A forman un solo árbol. Se parte de un vértice arbitrario s y lo incluye en el MST. Luego, en cada paso del algoritmo, el peso mínimo saliente borde (MWOE) (u, v) del fragmento de árbol actual T tal que $u \in T$ y $v \in G \setminus T$ se encuentra y se añade al árbol T . Si T es un fragmento MST de G , $T \cup (u, v)$ También es un fragmento MST de G por el Teorema 3. Procediendo de esta manera, el algoritmo finaliza cuando todos los vértices se incluyen en la MST final, como se muestra en el Algoritmo 7.1. La figura 7.3 muestra las iteraciones del algoritmo de Prim en un gráfico.

Algorithm 7.1 Prim_MST

```
1: Input :  $G(V, E, w)$ 
2: Output : MST  $T(V, E_T)$  of  $G$ 
3:  $V_T \leftarrow \{a\}$ 
4:  $T \leftarrow \emptyset$ 
5: while  $V_T \neq V$  do ▷ continue until all vertices are visited
6:   select the edge  $(u, v)$  with minimal weight such that  $u \in T$  and  $v \in G \setminus T$ 
7:    $V_T \leftarrow V_T \cup \{v\}$ 
8:    $E_T \leftarrow E_T \cup \{(u, v)\}$ 
9: end while
```

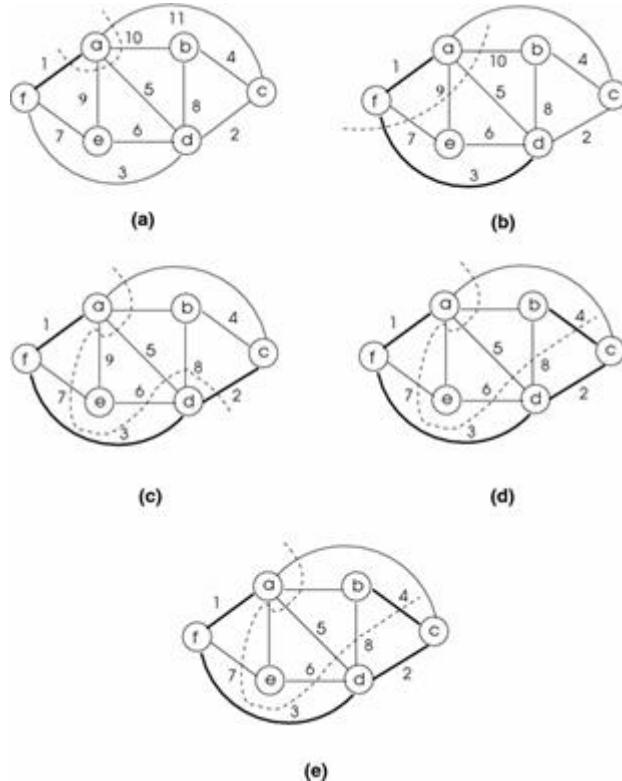


Fig. 7.1 Ejecución del algoritmo MST de Prim en un pequeño gráfico a partir del vértice a

Análisis

Teorema 7.3 (corrección) El algoritmo de Prim proporciona un MST del gráfico de entrada $G = (V, E, w)$

Prueba La propiedad de corte garantiza la corrección de este algoritmo, ya que siempre seleccionamos el MWOE que forma parte del MST de esta propiedad. Mostraremos una prueba alternativa. En cada iteración, sumamos un vértice $v \in V - V_T$ que está conectado a un vértice $u \in V_T$ sobre el borde más claro (u, v) entre los dos conjuntos. Como el borde (u, v) siempre está entre dos conjuntos separados, no puede formar un ciclo con vértices de V_T . Por lo tanto, T es siempre un árbol a lo largo del algoritmo en ejecución. Además, desde V_T contiene todos los vértices de G en el extremo como prueba en la línea 5, T es un árbol de expansión de G .

Ahora debemos verificar si T es un MST de todos los árboles que abarcan G y lo haremos por inducción. Dado que cada vértice debe estar cubierto por el MST, se demuestra la base de la inducción. Ahora queremos mostrar que si T_{i-1} es parte del MST, y luego agregar MWOE al mismo proporcionará T_i que también formará parte de MST. Supongamos T_i no es parte de la MST y dejar T_{i-1} es un MST parcial de G y al agregar el MWOE (u, v) de T_{i-1} para obtener T_i . De acuerdo con la regla del algoritmo de Prim. Vamos a asumir $(u, v) \notin T_{i-1}$ que es el MST final construido. En este caso, hay otro borde (w, z) en

el cutset entre T_{t-1} y T_t eso es parte de T_{t-1} . Además, (u, v) y (w, z) son bordes de un ciclo. Eliminar (w, z) de G da como resultado otro árbol T' de G que tiene un peso total menor que T_{t-1} ya que $w(u, v) \leq w(u, v)$. Esto significa T_t Está incluido en otro MST de G que contradice nuestro supuesto inicial. \square

Teorema 7.4 (complejidad) El algoritmo de Prim se ejecuta en $O(m \log n)$ hora.

Prueba La operación principal realizada por estos algoritmos es la selección del MWOE en cada iteración en la línea 6 del algoritmo 7.1. Una matriz dpuede contener las distancias mínimas a cualquier nodo en V_T para $\forall v \in V - V_T$, llamemos a este conjunto V'_T . Luego podemos encontrar el valor mínimo del vértice v de esta matriz para incluirlo en V_T con el borde correspondiente (u, v) en E_T . También debemos actualizar las entradas en esta matriz, ya que la inclusión del nuevo nodo v puede resultar en un cambio de los valores de sus vecinos. Tenemos que comprobar $n-1$ vértices para encontrar el valor mínimo de d en la primera iteración, seguido de $n-2$ iteraciones en el segundo paso con un paso menos que el anterior en cada paso, lo que resulta en $O(n^2)$ pasos. La actualización de los valores d requiere verificar los vecinos del nodo v en cada paso, lo que da como resultado la suma de los grados de los nodos en total, que es de $2m$. El tiempo total tomado por lo tanto es $O(m + n^2)$. Podemos usar una estructura de datos de pila para almacenar los valores de V'_T . Como veremos en detalle en la siguiente sección. Encontrar el valor mínimo de V'_T en el montón se puede hacer en $\log n$ Tiempo en cada paso en este caso. Actualizar los valores de d para los vecinos de v requiere una mayor $\deg(v) \log n$ en cada paso Sumar estas dos operaciones para n pasos resulta en $O(n \log n + m \log n)$. Para una gráfica densa, podemos asumir $m \gg n$ y el tiempo resultante es $O(m \log n)$. \square

Implementación

Encontrar el MWOE a partir del fragmento MST es clave para la operación de este algoritmo. Usaremos una cola de prioridad mínima basada en un atributo clave como se describe en la Secta. [6.2.7](#). Definimos la clave (v) de un vértice v como el peso mínimo de cualquier borde que conecte v con un vértice en el árbol que se inicializa ∞ . Ya que no tenemos un árbol al inicio. La cola Q contiene todos los vértices de la gráfica inicialmente y extraer un vértice con el valor de clave mínimo de Q en cada iteración de la mientas bucle. También asignamos el padre de cada vértice v en $P[v]$ para formar la estructura de árbol durante las iteraciones. El procedimiento ExtractMin (Q) elimina el elemento con la tecla más baja de Q . Por lo tanto, invocamos este procedimiento para encontrar MWOE hasta que Q quede vacío. El pseudocódigo para este algoritmo se muestra en el algoritmo 7.2.

Algorithm 7.2 Prim_MST²

```

1: Input :  $G(V, E, w)$  undirected, connected, and weighted graph
2:  $s$  source vertex
3: Output : MST  $T(V, E_T)$  of  $G$ 
4: for all  $u \in V$  do
5:    $key(u) \leftarrow \infty$ 
6:    $P[u] \leftarrow \emptyset$ 
7: end for
8:  $key(s) \leftarrow 0$ 
9:  $T \leftarrow \emptyset$ 
10:  $Q \leftarrow V$ 
11: while  $Q \neq \emptyset$  do                                > continue until all vertices are visited
12:    $u \leftarrow ExtractMin(Q)$ 
13:   for all  $(u, v) \in E$  do
14:     if  $v \in Q$  and  $w(u, v) < key(v)$  then
15:        $P[v] \leftarrow u$ 
16:        $key(v) \leftarrow w(u, v)$ 
17:     end if
18:   end for
19: end while

```

Análisis de la implementación de la cola de prioridad

Probaremos la exactitud de esta implementación con una cola de prioridad utilizando variantes de bucle como en [5]. Especificamos una variante de bucle con tres componentes antes de cada iteración del tiempo de bucle de la siguiente manera:

El conjunto $V' = \{(v, P[v]) : v \in V - \{s\} - Q\}$ es mantenido. Cuando seleccionamos el borde más claro (u, v) con vértice $u \in Q$ añadimos u al conjunto $V - Q$ cuales son los vértices de los árboles. Por lo tanto, el borde ($u, P[u]$) se añade a MST T .

- 1.
2. El conjunto $V - Q$ contiene vértices que se incluyen en el MST T . Esto es válido, ya que cada vez que eliminar un vértice de Q , lo añadimos al conjunto de vértices en el MST T .
3. Para cualquier vértice $v \in Q$, Si $P[v] \neq \emptyset$ entonces $key(v) < \infty$ y $key(v) = w(u, v)$ con $u \in T$. Es decir, cualquier vértice v que esté fuera del MST parcial construido con un parente definido tiene un valor de clave inferior a ∞ y está conectado a un vértice u en el MST asignado como su parente. Actualizamos la clave y el elemento primario de cada vértice vecino de u sin alterar ninguno de estos valores en el árbol T y, por lo tanto, esta parte de la variante del bucle se mantiene.

La complejidad del tiempo de esta implementación depende de cómo está estructurada la cola de prioridad mínima Q . Si el binario min-heap se describe en la sec. 6.2. Se usa 7, necesitamos formar el montón en $O(n)$ inicialmente. El tiempo de bucle se ejecuta n veces con la ExtractMin toma procedimiento $O(\log n)$ en cada carrera que resulta en $O(n \log n)$ veces para llamar a ExtractMin. La prueba de los valores clave de los vecinos del vértice u utilizando la lista de adyacencia requiere $2m$ de tiempo en total, verificando cada borde dos veces. Comprobando si $v \in Q$ se puede realizar en tiempo constante manteniendo una variable booleana para cada vértice. La asignación del valor clave en la línea 16 se puede realizar mediante el procedimiento DecreaseKey que requiere $O(\log n)$ hora. El tiempo total tomado por lo tanto es $O(n \log n + m \log n) = O(m \log n)$.

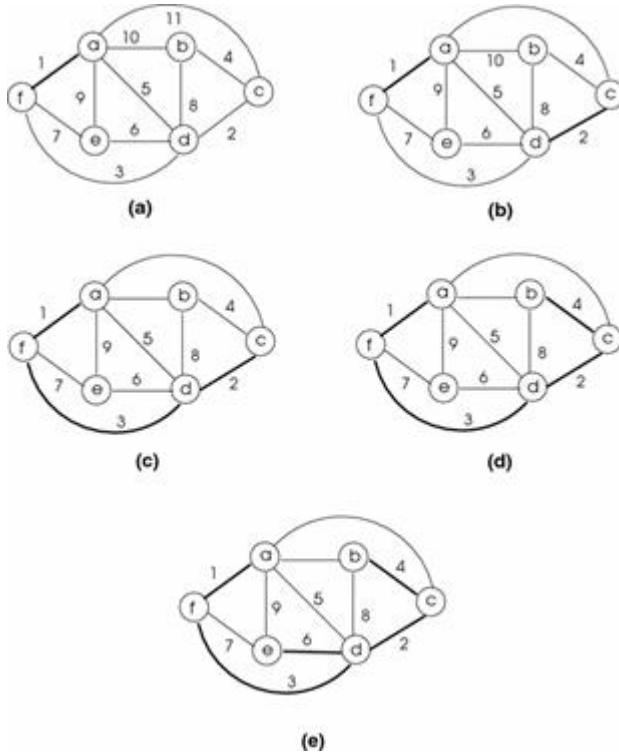


Fig. 7.2 Ejecución del algoritmo MST de Kruskal para el mismo gráfico de la Fig. 7.1 . El mismo MST se obtiene a medida que los pesos de los bordes son distintos.

7.2.2.2 Algoritmo de Kruskal

El algoritmo MST de Kruskal adopta un enfoque diferente al ordenar los bordes del gráfico $G = (V, E, w)$ En pesos no decrecientes . Luego, a partir del borde de peso más ligero, los bordes se incluyen en el MST parcial. τ siempre que no formen ciclos con los bordes ya contenidos en τ . Este proceso continúa hasta que todos los bordes de la cola se procesan como se muestra en el algoritmo 7.3.

Algorithm 7.3 Kruskal_MST

```

1: Input :  $G = (V, E, w)$ 
2: Output : MST  $T$  of  $G$ 
3:  $T \leftarrow \emptyset$ 
4:  $Q \leftarrow$  sorted edges in nondecreasing weights of  $E$ 
5: while  $Q \neq \emptyset$  do ▷ continue until all edges are checked
6:   pick an edge  $(u, v)$  from  $Q$ 
7:   if  $(u, v)$  does not make a cycle with vertices in  $T$  then
8:     add  $(u, v)$  to  $T$ 
9:   end if
10: end while

```

La figura 7.2 muestra las iteraciones del algoritmo de Kruskal en una gráfica. Probar si la adición de un borde crea un ciclo es crucial en la operación de este algoritmo. Cuando tenemos un bosque de árboles, debemos determinar si los puntos finales del borde (u, v) deben considerarse como pertenecientes al mismo árbol. Si lo son, incluir este borde en el MST creará un ciclo y, por lo tanto, deberíamos descartar este borde. También debemos fusionar dos árboles por el borde en consideración si hacerlo no crea un ciclo. Podemos usar la estructura de datos de Union-Find para este propósito que tiene las siguientes operaciones definidas:

- \square **MakeSet (x)**: crea un nuevo conjunto que consiste solo en el elemento x .
- \square **FindSet (x)**: encuentra el nombre (puntero al representante) del conjunto que contiene el elemento x .

- \square Unión (x): fusiona los conjuntos que contienen x e y para formar un nuevo conjunto. Se eliminan los conjuntos antiguos.

Implementación

Basándonos en estas operaciones, podemos reestructurar el algoritmo de Kruskal como se muestra en el Algoritmo 7.4. Todos los vértices de la gráfica son componentes del bosque primero. Los bordes se clasifican y se insertan en la cola Q , luego verificamos si los puntos finales de un borde (u , v) retirados de Q están en el mismo árbol. Si lo son, sabemos que agregar (u , v) a la MST T creará un ciclo y descartaremos esta ventaja. De lo contrario, los árboles de u y v se fusionan mediante la operación de la Unión para formar un nuevo árbol.

Algorithm 7.4 Kruskal_MST2

```

1: Input :  $G = (V, E, w)$  undirected, weighted graph
2: Output : MST  $T = (V, E_T)$  of  $G$ 
3:  $T \leftarrow \emptyset$ 
4:  $Q \leftarrow$  sorted edges in nondecreasing weights of  $E$ 
5: for all  $v \in V$  do
6:    $MakeSet(v)$ 
7: end for
8: while  $Q \neq \emptyset$  do                                 $\triangleright$  continue until all vertices are visited
9:    $(u, v) \leftarrow \text{dequeue}(Q)$ 
10:  if  $FindSet(u) \neq FindSet(v)$  then            $\triangleright$  check cycles
11:     $E_T \leftarrow E_T \cup \{(u, v)\}$                    $\triangleright$  include edge in MST
12:     $Union(u, v)$                                 $\triangleright$  merge trees if edge not in cycle
13:  end if
14: end while

```

Análisis

Teorema 7.5 (corrección) El algoritmo de Kruskal proporciona un MST del gráfico de entrada $G (V , E , w)$.

Prueba En cada iteración, agregamos un vértice $v \in V_T$ que está conectado a un vértice $u \in V_T$ sobre el borde más barato (u , v) entre los dos conjuntos. Como el borde (u , v) siempre está entre dos conjuntos separados, no puede formar un ciclo con vértices de V_T . Por lo tanto, T es siempre un árbol a lo largo del algoritmo en ejecución. Además, desde V_T contiene todos los vértices de G como prueba en la línea 5, T es un árbol de expansión de G . Ahora tenemos que demostrar T es un MST de G . Durante el i ^a iteración del algoritmo, y mucho A_i Ser el subconjunto de aristas de la MST final. r . Tenga en cuenta que a diferencia del algoritmo de Prim, A_i Puede contener un conjunto de bordes separados. No habrá ninguna ventaja en A_i que tiene un peso mayor que cualquier filo en $E \setminus A_i$ simplemente porque incluimos bordes de bajo peso en A_i a partir de la más baja. Esto significa que si el nuevo borde (u , v) que se va a agregar crea un ciclo con los bordes existentes en A_i , es el borde de mayor peso en ese ciclo. Por la propiedad de ciclo, estamos rechazando un borde que no pertenece al MST. Por otro lado, cada vez que aceptamos un borde, pertenece a la MST por la propiedad de corte. \square

La inicialización por MakeSet toma n pasos. Ya que habrá $n - 1$ Bordes MST, tenemos que ejecutar el procedimiento de unión . $n - 1$ veces. También necesitamos probar cada borde dos veces para cada uno de sus puntos finales mediante el procedimiento Find-Set que resulta en una invocación de 2 m . Usando listas vinculadas, el procedimiento de unión lleva tiempo O (n) y MakeSet y FindSet toman tiempo O (1), lo que resulta en $O(n^2)$ Tiempo para estos procedimientos. Además, los pesos de los bordes del gráfico G se pueden clasificar en $O(m \log m)$ Tiempo que es el tiempo dominante para este

algoritmo. Asumiendo $m < n^2$, $\log m < 2 \log n$ y por lo tanto la complejidad puede ser asumida como $O(m \log n)$.

7.2.2.3 Algoritmo de borrado inverso

Como otro método para construir un MST de un gráfico, podemos comenzar con todos los bordes del gráfico y eliminar los bordes que nunca se incluirán en el MST hasta que tengamos un gráfico conectado que tenga la propiedad del árbol, es decir, sea acíclico o tiene $n - 1$ bordes. Eliminamos bordes en el orden de pesos decrecientes siempre que la eliminación de un borde de este tipo no desconecte el gráfico, ya que cualquier puente del gráfico debe estar contenido en el MST. Más específicamente, el algoritmo consta de los siguientes pasos:

Entrada : Un gráfico ponderado no dirigido $G = (V, E, w)$

- 1.
2. Salida : Un MST T de G
3. Ordenar los bordes de G en orden no decreciente en Q
4. Dejar $T = G$
5. Repetir
6. Retira el primer borde (u , v) de Q
7. Retire (u , v) de T
8. Si tal eliminación deja T desconectado entonces
9. Unir (u , v) a T
10. Hasta $Q = \emptyset$

La ejecución de este algoritmo en un pequeño gráfico se muestra en la Fig. [7.3](#).

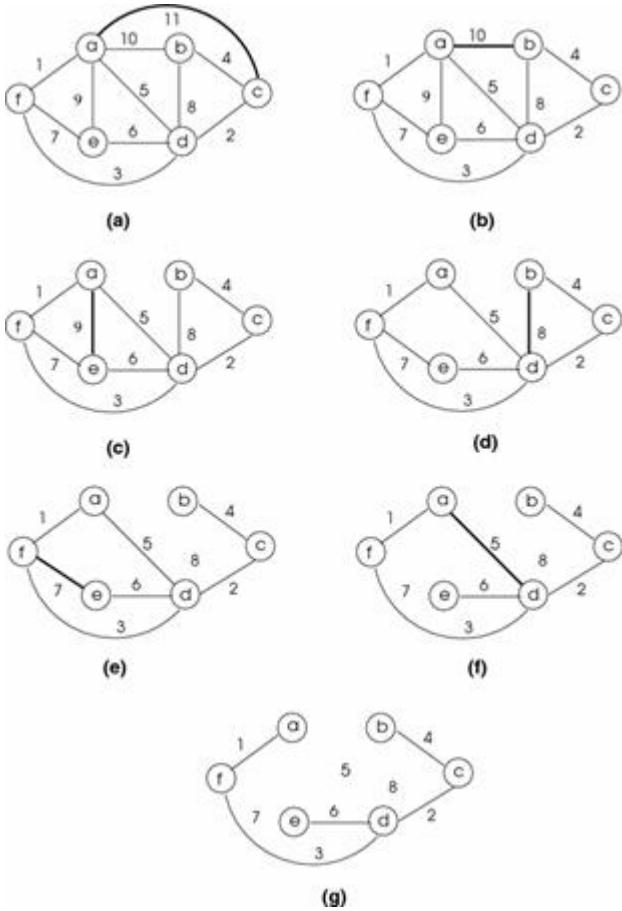


Fig. 7.3 Ejecución del algoritmo de borrado de borde inverso para el mismo gráfico de la Fig. 7.1 . El mismo MST se obtiene como en los algoritmos de Prim y Kruskal ya que los pesos de los bordes son distintos

Análisis

Teorema 7.6 (corrección) El algoritmo de eliminación de borde inverso proporciona un MST del gráfico de entrada $G = (V, E, w)$.

Prueba Este algoritmo produce un árbol de expansión, ya que la estructura resultante no contiene ningún ciclo, ya que eliminamos el borde de mayor peso que se encuentra en una eliminación de ciclo del cual no se desconecta el gráfico. Mostraremos que el árbol de expansión resultante T es un MST de G de la siguiente manera. Sea (u, v) el borde eliminado durante una iteración del algoritmo. Antes de la retirada, que debe haber sido en un ciclo de C como de lo contrario dicha eliminación desconectaría G . Puesto que es el primer borde encontrado en C , es el borde peso más pesado en el ciclo C . Por la propiedad del ciclo, el borde (u, v) No pertenece a ninguna de MST G . Por lo tanto, este algoritmo da como resultado un MST de Gya que elimina los bordes que no pueden estar contenidos en cualquier MST de la gráfica G . □

Los pesos de los bordes de la gráfica G se pueden clasificar en $O(m \log m)$ tiempo o $O(m \log n)$ Tiempo para una gráfica densa. El principal problema con este algoritmo es la prueba de la conexión del gráfico. Esto puede ser realizado por el algoritmo DFS o BFS en $O(n + m)$ tiempo después de cada eliminación de borde que resulta en $O(nm + m^2)$ hora. El tiempo total tomado es entonces $O(m \log m + nm + m^2)$. En [18] se muestra que la eliminación de un borde, la comprobación de la conectividad después de la eliminación y la reinserción del borde si el gráfico está desconectado se puede realizar en $O(m \log n (\log \log n)^3)$ Tiempo por operación.

7.2.2.4 Algoritmo de Boruvka

El algoritmo de Boruvka fue el primer algoritmo informado para construir un MST de un gráfico ponderado $G(V, E, w)$. Fue diseñado para construir una red eléctrica de Moravia en la República Checa en 1926. Comienza por encontrar el borde de menor peso (u, v) que incide en cada vértice $v \in V$. Luego contrae u y v para tener un componente C que contenga u, v y el borde (u, v) en él, y el borde (u, v) se incluye en el MST. Es posible que el borde (u, v) sea el borde incidente más claro al vértice v , pero otro borde, digamos (u, w) , es el borde más claro que incide al vértice u . En tal caso, ambos bordes (u, v) y (u, w) se incluyen en el MST, y estos bordes con todos los vértices que inciden en estos bordes se colocan en el mismo componente. Luego, en todos los pasos después de la inicialización; dos componentes C_x y C_y se contraen para formar un componente más grande. C_z utilizando el borde de peso más ligero (u, v) entre ellos y el borde (u, v) se incluye en el MST. Si componente C_x y / o C_y tiene bordes más claros que (u, v) incidentes a ellos, estos bordes también se incluyen en el MST y estos bordes con sus vértices incidentes se colocan en el nuevo componente C_z . Este proceso se repite hasta que solo hay un componente que contiene todos los vértices que proporcionan un árbol de expansión y los bordes seleccionados son los bordes de MST como se muestra en el algoritmo 7.5.

Algorithm 7.5 Boruvka_MST

```

1: Input :  $G(V, E, w)$ 
2: Output : MST  $T(V_T, E_T)$  of  $G$ 
3: Let each vertex  $v \in V$  be a component
4:  $T \leftarrow \emptyset$ 
5: while there is more than one component do
6:   combine two neighbor components  $C_x$  and  $C_y$  using the lightest edge  $(u, v)$  between them
7:    $E_T \leftarrow E_T \cup \{(u, v)\}$                                  $\triangleright$  include lightest edge in  $T$ 
8:    $V_T \leftarrow V_T \cup \{u, v\}$                                  $\triangleright$  include vertices in  $T$ 
9:   if  $\exists(p, q) \in C_x : w(p, q) > w(u, v)$  and/or  $\exists(r, z) \in C_y : w(r, z) > w_{u,v}$  then
10:     $E_T \leftarrow E_T \cup \{(p, q)\}$  and/or  $E_T \leftarrow E_T \cup \{(r, z)\}$ 
11:     $V_T \leftarrow V_T \cup \{p, q\}$  and/or  $V_T \leftarrow V_T \cup \{r, z\}$ 
12:   end if
13: end while
```

La figura 7.4 muestra dos iteraciones del algoritmo de Boruvka en una gráfica.

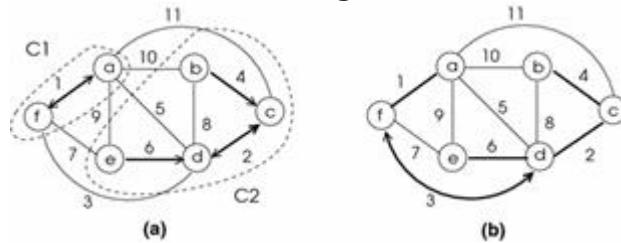


Fig. 7.4 Ejecución del algoritmo MST de Boruvka en el gráfico G de la Fig. 7.1. El primer paso en un divide el gráfico en dos componentes $C_1 = G[a, f]$ y $C_2 = G[b, c, d, e]$ con los bordes incidentes de peso más ligero incluidos en MST como se muestra en negrita. El borde más claro entre estos componentes es (f, d) que se usa para fusionarlos y este borde se convierte en parte del MST como se muestra en b. Las flechas muestran al vértice que inciden los bordes más claros. El mismo MST que en los algoritmos de Prim, Kruskal y los algoritmos de borrado inverso se obtienen a medida que los pesos de los bordes son distintos

Tenga en cuenta que podemos terminar descubriendo el MST completo incluso con una iteración de este algoritmo si se seleccionan los bordes más claros que inciden en cada vértice, como se muestra en la Fig. 7.5.

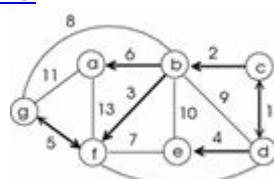


Fig. 7.5 Ejecución del algoritmo MST de Boruvka cuando todos los bordes MST se seleccionan en una iteración. Los bordes de MST se muestran en negrita y las flechas muestran el vértice en el que inciden los bordes más claros

Análisis

Lema 7.1 Supongamos los pesos de los bordes de la gráfica. $G = (V, E, w)$ son distintos. Dejar e_v ser el borde de menor peso incidente a un vértice $v \in V$. El MST T de G contiene todos esos bordes e_v .

Prueba Supongamos que el MST T , que es único debido a los distintos grosores de los bordes, no contiene ningún borde. $e_v = (u, v)$. Agregar el borde (u, v) a T crea un ciclo. Deje que el vértice w sea un vecino del vértice v , entonces $w(u, v) < w(v, w)$ ya que (u, v) es el borde más claro incidente en v . Si eliminamos (v, w) de T y agregamos (u, v) a T , obtenemos $T' = T - \{(v, w)\} \cup \{(u, v)\}$ que sigue siendo un árbol que tiene $n - 1$ Bordes y tiene un peso total menor que el peso de T dando como resultado una contradicción. \square

Teorema 7.7 (corrección) El algoritmo de Boruvka produce el MST del gráfico de entrada $G = (V, E, w)$ que tiene distintos pesos de borde.

Prueba El componente final T es un árbol, ya que unimos dos componentes de árbol mediante exactamente un borde que evita ciclos en cada iteración. Este componente T incluye todos los vértices de G a medida que continuamos hasta que hay un componente, que es un árbol de expansión de G . Los bordes que deben incluirse en el árbol de T en cada iteración son parte del MST de G por el Lema 7.1, por tanto, el árbol resultante T es el MST de G . \square

Tenga en cuenta que requerimos que los pesos de los bordes sean distintos para seleccionar un único borde de menor peso que incide en un vértice. Esta restricción se puede relajar variando ligeramente los pesos de los bordes de igual peso para tener pesos de borde únicos.

Teorema 7.8 (complejidad) El algoritmo de Boruvka se ejecuta en $O(m \log n)$ hora.

Prueba Cada paso del algoritmo reduce el número de vértices en al menos un factor de 2 y, por lo tanto, el número total de pasos es $\log n$. Cada paso requiere $O(m)$ tiempo para la contracción que resulta en $O(m \log n)$ hora. \square

7.2.3 Algoritmos paralelos de MST

Fuera de los cuatro algoritmos que hemos revisado, de Boruvka algoritmo es el más adecuado para el procesamiento en paralelo debido a las operaciones de contracción relativamente independientes que participan. Sin embargo, primero veremos formas de paralelizar el algoritmo de Prim y luego describiremos brevemente cómo se puede ejecutar el algoritmo de Boruvka en un sistema paralelo de memoria distribuida.

7.2.3.1 Algoritmo de Prim paralelo

Dado el gráfico de entrada $G = (V, E, w)$, consideraremos el algoritmo de Prim para encontrar el MST $T = (V, E_T)$ de G en paralelo. Comenzamos considerando la versión modificada del algoritmo con la matriz $d[n]$ que contiene los valores de distancia mínima de los nodos que están en $V - V_T$ a V_T para funcionamiento en paralelo. El algoritmo 7.6 muestra la

operación del algoritmo secuencial como en [10] donde seleccionamos arbitrariamente un vértice s y primero inicializamos los valores de matriz para los vecinos de s . Luego, en cada iteración, encontramos el valor mínimo de la matriz d , que es el MWOE de la iteración actual. El nodo v que tiene este valor con borde (u, v) como MWOE se incluye en el MST. Necesitamos actualizar los valores de la matriz como la distancia a la nueva v_T . El vértice v tiene que ser considerado ahora.

Algorithm 7.6 Prim_MST3

```

1: Input :  $G = (V, E, w)$ 
2: Output : MST  $T$  of  $G$ 
3:  $V_T \leftarrow \{s\}$ 
4: for all  $u \in V - V_T$  do
5:   if  $(u, z) \in E$  then
6:      $d[u] \leftarrow w(u, z)$ 
7:   end if
8: end for
9:  $T \leftarrow \emptyset$ 
10: while  $V_T \neq V$  do           > continue until all vertices are visited
11:   select the minimum element of  $d$  with vertex  $v$  and edge  $(u, v)$ 
12:    $V_T \leftarrow V_T \cup \{v\}$ 
13:    $E_T \leftarrow E_T \cup \{(u, v)\}$ 
14:   for all  $u \in N(v)$  do      > update distances to  $V_T$  considering the new node  $v$ 
15:      $d[u] \leftarrow \min[d[u], w(u, v)]$ 
16:   end for
17: end while

```

Este algoritmo es inherentemente secuencial a medida que buscamos el MWOE en cada iteración. Sin embargo, la búsqueda de MWOE se puede hacer en paralelo dentro de una única iteración. La idea general del algoritmo paralelo es dividir los vértices de k procesos y hacer que encuentren los MWOE en sus particiones. El MWOE global se puede encontrar luego mediante un proceso especial que lo transmite a todos los demás para actualizaciones locales, como en las líneas 14 a 16 del algoritmo secuencial. En la implementación, tenemos k procesos, P_0, \dots, P_{k-1} con P_0 como el supervisor. Dividimos los vértices en k subconjuntos donde cada proceso P_i obtiene n / k vértices en el conjunto V_i . La matriz d es un bloque 1-D particionado para k procesos y la matriz de adyacencia ponderada A también está particionada en columnas para k procesos. Cada proceso luego encuentra el valor mínimo de la matriz d en su partición utilizando la matriz A y el mínimo global se calcula utilizando la reducción de todos a uno en el proceso raíz P_0 que luego lo transmite a todos los procesos. Los procesos actualizan sus distancias y forman su partición de d para los nodos que son responsables. Este proceso continúa hasta que la matriz d ya no tiene elementos, lo que significa que todos los nodos están en V_T . El pseudocódigo para este algoritmo se muestra en el algoritmo 7.6.

Algorithm 7.7 Parallel_Prim_MST

```

1: Input :  $A[n, n]$  : weighted adjacency matrix of  $G = (V, E, w)$ 
2:    $P = \{p_0, p_1, \dots, p_{k-1}\}$                                  $\triangleright$  set of  $k$  processes
3: Output : MST  $T = (V_T, E_T)$  of  $G$ 
4: if  $p_i = p_0$  then                                          $\triangleright$  if I am the root process
5:   for  $i = 1$  to  $k - 1$  do
6:     send column  $((i - 1)n/k) + 1$  to  $in/k$  of  $A$  to  $p_i$ 
7:   end for
8:   for  $round = 1$  to  $n - 1$  do                                      $\triangleright$  loop for  $n-1$  rounds
9:     all-to-one receive MWOEa in values
10:     $min\_val(u, v) \leftarrow min(values)$                                  $\triangleright$  find the global minimum distance
11:    one-to-all broadcast new_MWOE( $u, v$ ) to all processes
12:     $V_T \leftarrow V_T \cup \{v\}$ 
13:     $E_T \leftarrow E_T \cup \{(u, v)\}$ 
14:   end for
15: else                                                  $\triangleright$  I am a worker process
16:    $round \leftarrow 0$ 
17:   receive my column partition  $D[my\_cols]$  from  $p_0$ 
18:   while  $round < k$  do                                          $\triangleright$  get local minimum values from processes
19:      $d_{p_i}(u, v) \leftarrow$  minimum distance between  $u$  and  $v$  in  $d[my\_columns]$ 
20:     send proc_min( $u, v, d_{p_i}$ ) to  $p_0$ 
21:   receive new_MWOE( $u, v$ ) from  $p_0$ 
22:   for all  $u \in N(v) \wedge u \in d[my\_columns]$  do  $\triangleright$  update distances to  $V_T$  considering the
      new node  $v$ 
23:      $d[u] \leftarrow min[d[u], w(u, v)]$ 
24:   end for
25:    $round \leftarrow round + 1$ 
26: end while
27: end if

```

Mostraremos la implementación de este algoritmo paralelo utilizando cuatro procesos. p_0, \dots, p_3 . Para el mismo gráfico que hemos usado para demostrar algoritmos secuenciales. Matriz de adyacencia ponderada, a veces llamada matriz de distancia, A se forma y se divide de la siguiente manera:

	p_1		p_2		p_3
una	segundo	dore	miF		
una	0 10	11 5 9 1			
segundo	10 0	4 8 ∞ ∞			
do	11 4	0 2 ∞ ∞			
re	5 8	2 0 6 3			
mi	9 ∞	∞ 6 7 7			
F	1 ∞	∞ 3 7 0			

Las primeras entradas de la matriz d serían las siguientes:

segundo dore miF

10 11 5 9 1

El proceso raíz recopila los valores mínimos de 10, 5 y 1 de los procesos. p_1 , p_2 y p_3 respectivamente y determina el valor mínimo global de 1 entre los nodos a y f. A continuación, transmite este valor que se incluye en V_T , el nodo f se elimina de V_T y todos los bordes vecinos se prueban para obtener la nueva d como se muestra a continuación:

segundo dore mi

10 11 3 9

Esta vez, el nodo d se transmite a todos los procesos, se elimina de V_T y d se actualiza para obtener d con (8, 2, 6) para los nodos b, c y e respectivamente. Tres rondas más del algoritmo paralelo proporcionan el mismo MST encontrado por otros métodos.

Análisis

Cada proceso p_i encuentra el valor mínimo y realiza las actualizaciones en $\Theta(n/k)$ tiempo y el tiempo total es $\Theta(n^2/k)$ para n rondas. Se necesita $\log k$ tiempo para realizar la comunicación de uno a todos en cada ronda, lo que resulta en $\Theta(n \log k)$ Tiempo total para la comunicación. El tiempo total tomado es

$$T_p = \Theta(n^2/k) + \Theta(n \log k)$$

(7.2)

Dado que el tiempo secuencial es $\Theta(m \log n)$ para el algoritmo de Prim, la aceleración obtenida es

$$S = \frac{\Theta(m \log n)}{\Theta(n^2/k) + \Theta(n \log k)}$$

(7.3)

7.2.3.2 Algoritmo de Kruskal paralelo

El algoritmo de Kruskal crece múltiples árboles que se pueden realizar en paralelo. Implementamos una estrategia similar a la del algoritmo paralelo de Prim dividiendo la matriz de adyacencia entre k procesos. El algoritmo paralelo de Kruskal consta de los siguientes pasos [14]:

Cada proceso p_i ordena los bordes en su partición V_i .

- 1.
2. Cada proceso p_i construye un MST o un bosque de MST en su partición utilizando el algoritmo de Kruskal .
3. Los MST encontrados por proceso se combinan por pares mediante un proceso que envía sus bordes MST a otro proceso. Este paso se puede manejar rompiendo simetrías usando identificadores. El proceso de identificador inferior puede enviar sus bordes MST a un proceso de identificador superior seleccionado arbitrariamente. El proceso del identificador inferior debería detenerse.
4. El paso 3 se repite hasta que hay un proceso que contiene el MST.

La fusión de bordes en el paso 2 se puede realizar utilizando el algoritmo de Kruskal . La Figura 7.6 muestra el funcionamiento de este algoritmo paralelo en un pequeño gráfico donde dividimos la matriz de adyacencia en cuatro procesos. El cálculo de los bordes de MST en cada partición toma $\Theta(n^2/k)$ tiempo y hay $\Theta(\log k)$ operaciones de fusión cada uno con un costo de $\Theta(n^2 \log k)$ y cada proceso envía $O(n)$ bordes en una combinación que resulta en un tiempo paralelo total de $\Theta(n^2/k) + \Theta(n^2 \log k)$ [14].

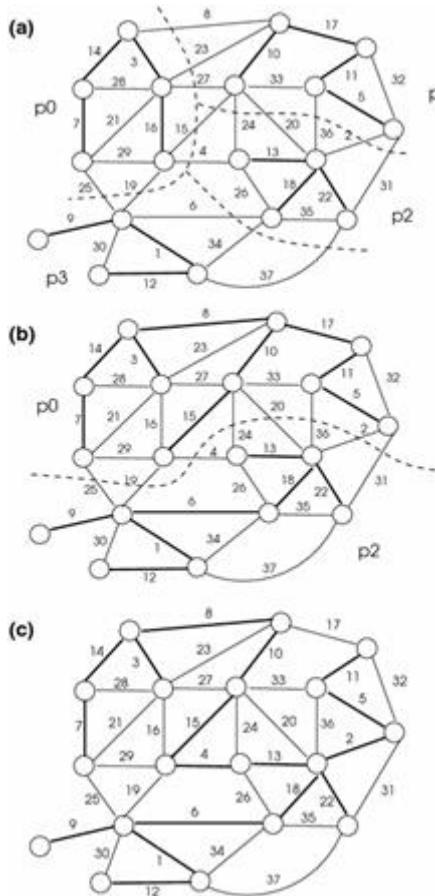


Fig. 7.6 Ejecución del algoritmo de Kruskal en paralelo en un gráfico de muestra. Hay 4 procesos y la matriz de adyacencia del gráfico se divide en cuatro procesos p_0 , p_1 , p_2 y p_3 como se muestra por líneas discontinuas

7.2.3.3 Algoritmo de Boruvka paralelo

El algoritmo de Boruvka puede funcionar en paralelo como lo mostramos en el pseudocódigo de alto nivel del algoritmo 7.8.

Algorithm 7.8 Par_Boruvka

```

1: Input :  $G(V, E, w)$  undirected, weighted graph
2: Output : MST  $T$  of  $G$ 
3: for all  $v \in V$  in parallel do
4:   Find the lightest edge incident to  $v$ 
5: end for
6: Contract edges
7: Merge adjacency lists
8: Recuse until all edges are processed

```

Varios algoritmos MST paralelos se basan en el algoritmo de Boruvka . Uno de estos enfoques se describe en [4] donde los súper vértices resultantes después de la contracción consisten en árboles de vértices. La información de vecindad se mantiene en listas de bordes, una para cada vértice. Hay a lo sumo $\frac{n-1}{2}$ Elementos en cada lista de bordes. Los pasos del algoritmo de Boruvka en este estudio consisten en los siguientes pasos:

1. Elija el borde más claro : la lista de bordes de cada vértice se busca para encontrar el borde de menor peso que incide en ese vértice para formar componentes. Los ciclos se eliminan para tener un árbol para cada componente.
2. Buscar raíz : cada vértice encuentra la raíz del árbol al que pertenece mediante el método de salto de puntero.
3. Renombrar vértices : cada proceso p_i , $1 \leq i \leq k$ Determina el nuevo nombre de cada vértice listado en sus listas de bordes.

4. Fusionar : las listas de bordes de cada componente se fusionan con la lista de bordes de raíz para reducirla en un solo super-vértice.
5. Limpiar : cada proceso p_i ejecuta el algoritmo MST secuencial en su lista de bordes.

El tiempo de ejecución paralelo para este algoritmo se da como $\Theta((t_s + t_w)(m \log n/p))$ lo que resulta en una aceleración comparable al número de procesos paralelos pero la constante $(s + t_w)$ puede ser muy grande [4]. La contracción se puede realizar utilizando los métodos de contracción de borde o estrella que hemos revisado en el cap. 3 para obtener un algoritmo de Boruvka paralelo [1].

7.2.4 Algoritmo de Prim distribuido

En la versión distribuida de este problema, estamos interesados en encontrar el MST de una red en la que cada nodo esté involucrado en la construcción. Consideraremos cada uno de los tres algoritmos secuenciales para este propósito. Como primer intento, la investigación del algoritmo de Prim revela que es básicamente de naturaleza secuencial. Sin embargo, el modelo de iniciador único síncrono (SSI) de procesamiento distribuido puede ser conveniente para este propósito. Construiremos y utilizaremos el MST para la transferencia correcta del mensaje entre la raíz y otros nodos en el árbol. El procesamiento se realiza en rondas síncronas en este modelo y tenemos un proceso de raíz a el cual inicia cada ronda. En la primera ronda, incluye el borde más claro que le incumbe en el MST. En cada ronda posterior, la raíz solicita el MWOE de cada hoja de la T parcial que son convergecast a la raíz. Luego encuentra el MWOE más pequeño recibido de niños y lo transmite a los miembros de T que pueden actualizar sus estados. En esencia, estamos procesando el gráfico exactamente como en el algoritmo secuencial, pero como no conocemos el MWOE global de antemano, la raíz del proceso especial debe recibir a todos los candidatos de cada hoja de T y determinar el borde más claro. Describiremos una posible implementación de esta idea similar a [7 , 15]. Los mensajes necesarios son los siguientes.

- Inicio : Esto es enviado por la raíz a sus hijos en cada ronda. Tiene un doble propósito; la iniciación de una nueva ronda k y llevando el MWOE (u , v) determinaron en la ronda k -1 del árbol parcial T .
- Respuesta : Este es el mensaje de convergecast desde las hojas a la raíz. En cada nodo intermedio, se reúnen los MWOE de los hijos, en comparación con el propio MWOE, y el más pequeño de ellos se envía mediante el mensaje de respuesta al padre.
- Cheque : este mensaje es necesario para evitar ciclos. El nodo v recién agregado lo envía a los vecinos para verificar los que ya están en Ty, para tal nodo u , el borde (u , v) se marca como interno, por lo que no se considerará como MWOE de v en futuras rondas.
- Estado : este mensaje es devuelto por un nodo u como una respuesta para verificar el mensaje desde v y contiene información sobre si $u \in T$ o no.

El algoritmo 7.9 muestra una forma de implementar el procedimiento que hemos descrito. La sincronización es proporcionada solo por mensajes y la raíz comienza la siguiente ronda solo después de que se reciben todos los mensajes de convergecast de sus hijos. Selecciona el borde más claro (u , v) con v como el nuevo vértice y lo envía a los nodos de la T parcial en la siguiente ronda. Cualquier nodo x que tiene un borde de vértice v marca este borde (x , v) como interno para evitar ciclos en T . El

vértice v comprueba si sus vecinos están en T_0 no. Esto es de nuevo necesario para evitar ciclos como un vecino pudo haber pasado a formar parte de la T . Las hojas inician el proceso de convergecast que termina en la raíz con los MWOE recibidos de los niños.

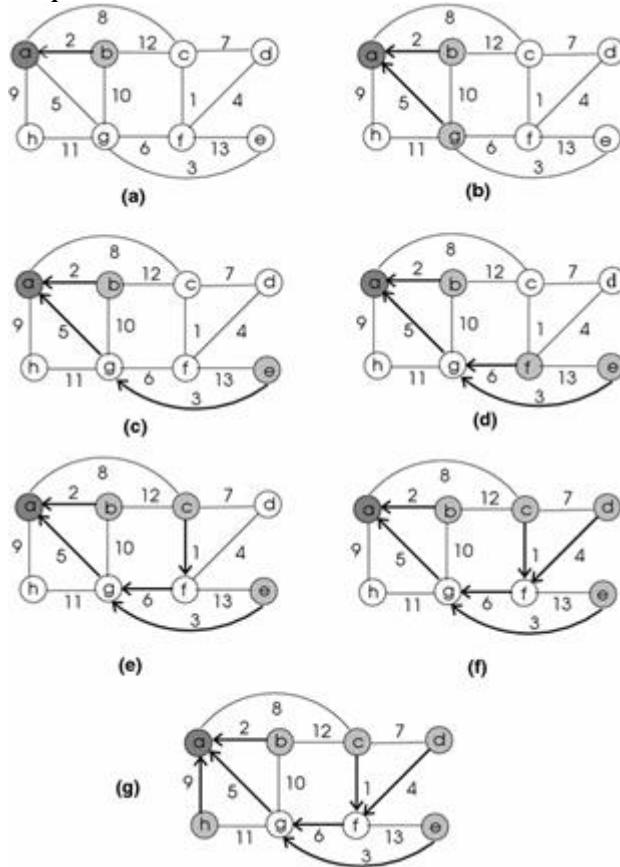


Fig. 7.7 Ejecución del algoritmo MST de Prim distribuido en una gráfica del vértice a durante siete rondas. Los nodos gris oscuro muestran las hojas del árbol formado. El downcasting de la sonda demensaje está en la dirección inversa de las flechas de líneas en negrita de padres a hijos y upcasting de ack mensaje está en la dirección de las flechas de los niños a los padres

Algorithm 7.9 Distributed_Prim_MST

```

1: Input :  $G(V, E, w)$ 
2: Output : MST  $T(V_T, E_T)$  of  $G$ 
3: if  $a = root$  then
4:   let  $(a, b)$  be the lightest edge
5:    $T \leftarrow (a, b)$                                 > a becomes the parent of b
6: repeat                                         > round k
7:   send  $start(v, (u, v), k)$  to children
8:   receive  $reply((p, q)_i, w_{(p, q)_i})$  from each child  $i$ 
9:   if  $reply = \emptyset$  from all children then
10:    send  $stop$  to all children
11:   else
12:      $(u, v)$  with new node  $v$  is the lightest edge of all MWOEs received from children
13:      $T \leftarrow T \cup (u, v)$                           > include  $(u, v)$  in  $T$ 
14:   end if
15:   until  $reply = \emptyset$  from all children
16: else                                         > I am node  $x$  on  $T$ 
17:   while  $stop$  not received do
18:     receive  $start(v, (u, v), k)$ 
19:     if  $(v, x) \in N(x)$  then                      > check cycles
20:        $state(v, x) \leftarrow internal$ 
21:     else if  $x = v$  then
22:       for all  $u \in N(x)$  do
23:         send  $check$  to  $u$ 
24:       end for
25:       for all  $u \in N(x)$  do
26:         receive  $status(res)$  from  $u$ 
27:          $state(u, x) \leftarrow res$                       > mark neighbor edges as internal or external
28:       end for
29:     end if
30:     if  $x$  is a leaf then
31:       let  $(x, z)$  be my MWOE such that  $state(x, z) = external$ 
32:       send  $((x, z), w_{(x, z)})$  to parent
33:     else                                         > convergence data of children
34:       receive  $reply((p, q)_i, w_{(p, q)_i})$  from each child  $i$ 
35:       find the lightest edge  $(p, q)$ , including my MWOE
36:       send  $((p, q), w_{(p, q)})$  to parent
37:     end if
38:   end while
39: end if

```

La ejecución de este algoritmo en un pequeño gráfico de red se muestra en la Fig. 7.7, donde la construcción del MST se completa en siete rondas.

Este algoritmo encuentra correctamente el MST de un gráfico, ya que imita el algoritmo Prim secuencial en una configuración distribuida. Cada paso k del algoritmo requiere tiempo $O(k)$ y mensajes. El tiempo y las complejidades del mensaje son, por lo tanto, $O(n^2)$.

Al observar otros algoritmos secuenciales, el algoritmo de Kruskal es difícil de implementar por parte de los nodos en la red, ya que requiere un ordenamiento global de los pesos de los bordes. Sin embargo, Boruvka Los algoritmos implican pasos independientes. Como cada nodo del gráfico ahora es un nodo en la red, cada nodo puede encontrar el borde de incidente más ligero en la fase inicial en un solo paso. Sin embargo, necesitamos encontrar formas de contratar y administrar los nodos contratados. Un enfoque simple pero efectivo es elegir un líder para cada componente contratado que pueda encontrar el MWOE de los nodos en su componente y pedirle al componente conectado en el otro extremo de este MWOE la operación de fusión. El líder puede ser el nodo identificador más bajo o el nodo más nuevo en el componente. Los líderes de cada componente se comunican con los líderes vecinos y deciden el borde más claro entre ellos. Un método similar se emplea en el algoritmo de Gallager , Humboldt y Spira para encontrar el MST de una red [8]. La naturaleza del algoritmo de Boruvka proporciona un procesamiento distribuido convenientemente. Sin embargo, la elección del líder debe ser realizada.

7.3 Caminos más cortos

En diversas situaciones de la vida real, nos puede interesar encontrar el camino más corto, es decir, el camino con el peso total mínimo entre todos los otros caminos entre dos vértices. Por ejemplo, puede requerirse la ruta más corta entre dos ciudades. Revisamos los algoritmos para este problema en esta sección. En todos estos algoritmos,

emplearemos una técnica llamada relajación que se puede describir a continuación. Dado un gráfico ponderado $G = (V, E, w)$ Queremos encontrar rutas más cortas desde un vértice fuente. $s \in V$ a todos los otros vértices en el gráfico G . Definimos un valor de distancia. d_v que muestra la mejor estimación de la distancia actual de v a s y un vértice u predecesor de v que es su padre en el árbol formado enraizado en el vértice de origen s . Formaremos un árbol de expansión T arraigado en s al final de un algoritmo de ruta más corta, a veces denominado árbol de ruta más corta en el que la suma de pesos de una ruta desde s hasta un vértice v en este árbol será mínima entre todas posibles caminos de s a v. La distancia de cada vértice desde el vértice de origen se establece en infinito y su parent no está definido inicialmente. Luego, la relajación implica verificar si se encuentra una ruta más corta de un vértice v a través de un vértice vecino u que su distancia actual, en cuyo caso se actualiza su distancia para atravesar ese vértice vecino u y su parent se establece en u como se muestra en los siguientes pasos Realizado para cada vértice v . Necesitamos agregar el peso del borde entre estos dos vértices a la distancia del vértice u para obtener la distancia real.

para todos $u \in N(v)$

- 1.
2. Si $d_v > d_u + w(u, v)$
3. $d_v = d_u + w(u, v)$
4. $P_v = u$

Otro tema de preocupación es si la gráfica tiene pesos negativos y / o ciclos negativos.

7.3.1 Rutas más cortas de una sola fuente

En el caso más general, podemos buscar rutas más cortas desde un solo vértice a todos los otros vértices, lo que se denomina problema de ruta más corta de una sola fuente (SSSP) para el cual revisamos un algoritmo fundamental debido a Dijkstra .

7.3.1.1 El algoritmo de Dijkstra

Dijkstra propuso un algoritmo iterativo para encontrar las distancias más cortas desde un vértice de una sola fuente a todos los otros vértices en un gráfico ponderado, dirigido o no dirigido $G = (V, E, w)$ [6]. La idea general de este algoritmo es que empezar desde un vértice fuente s e inicialmente etiquetar valores de distancia de todos los vecinos de s con el peso de los bordes de s , 0 y el resto de los vértices con valores de distancia infinito. y el valor de la distancia de s a sí mismo como 0. El vértice vecino v que tiene la distancia más pequeña a s se incluye en el conjunto de vértices visitados. A partir de entonces en cada iteración, el valor de la distancia y el predecesor de cualquier vecino u del vértice incluido recientemente v ha sido actualizado si la distancia a través de v Es más pequeño que su distancia actual. Este algoritmo procesa todos los vértices y eventualmente forma un árbol de expansión arraigado en el vértice de origen s . Tenemos un conjunto de vértices S que muestra los vértices a procesar, una matriz D con D [i] que muestra la distancia más corta actual del vértice i al vértice fuente s y otra matriz P con P [i] que muestra la predecesora actual de a vértice i a lo largo de este camino como se muestra en el algoritmo 7.3.

Algorithm 7.10 Dijkstra_SSSP

```

1: Input :  $G(V, E, w)$                                  $\triangleright$  connected, weighted graph  $G$  and a source vertex  $s$ 
2: Output :  $D[n]$  and  $P[n]$                            $\triangleright$  distances and predecessors of vertices in the tree
3:   tree vertices  $T$ 
4: for all  $v \in V \setminus \{s\}$  do                       $\triangleright$  initialize all vertices except source  $s$ 
5:    $D[v] \leftarrow \infty$ 
6:    $P[v] \leftarrow \perp$ 
7: end for
8:  $D[s] \leftarrow 0$ ;  $P[s] \leftarrow s$ 
9:  $V' \leftarrow V \setminus T$ 
10: while  $V' \neq \emptyset$  do
11:   find  $v \in S$  with minimum distance value           $\triangleright$  update neighbor distances to  $v$ 
12:   for all  $(u, v) \in E$  do
13:     if  $D[u] > D[v] + w(u, v)$  then
14:        $D[u] \leftarrow D[v] + w(u, v)$ 
15:        $P[u] \leftarrow v$                                  $\triangleright$  update tree structure
16:     end if
17:   end for
18:    $V' \leftarrow V' \setminus \{v\}$                           $\triangleright$  remove new vertex from searched
19:    $T \leftarrow T \cup \{v\}$                              $\triangleright$  add it to tree vertices
20: end while

```

La ejecución de este algoritmo se muestra en la figura 7.8. El vértice de origen es f y el vértice más cercano a f es un que se incluye en los vértices buscados. Luego, todos los vecinos de a que son b y e se verifican si tienen una distancia más corta al vértice fuente f a a . Dado que estos vértices tenían distancias infinitas inicialmente, sus distancias se modifican para valores más pequeños a través de a . Luego, encontramos que el vértice etiene el valor de distancia más pequeño y lo incluimos en los vértices buscados y los valores de distancia de actualización de sus vecinos. Tenga en cuenta que vérticeb tiene un valor de distancia menor de 7 a e, por lo que su distancia se actualiza y su predecesor se convierte en e . Este proceso continúa hasta que buscamos en todos los vértices que se realiza eliminando la distancia más corta v del conjunto de vértices inicial V' en cada iteración

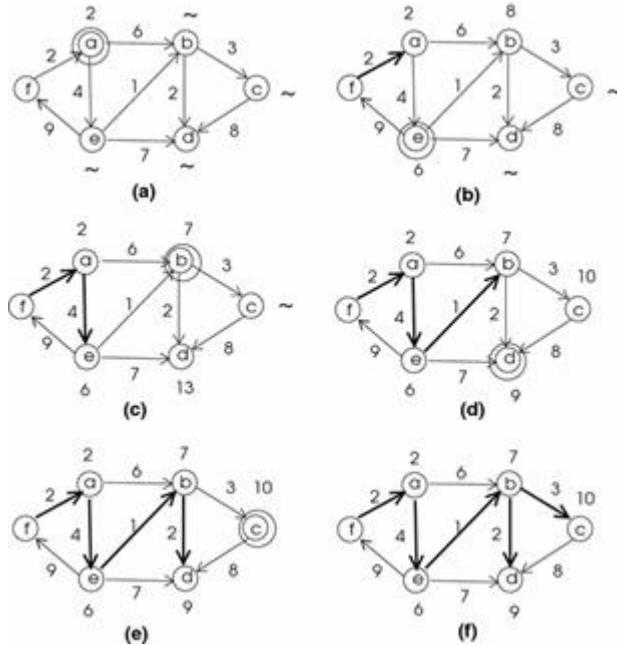


Fig. 7.8 Ejecución del algoritmo SSSP de Dijkstra en un ejemplo de gráfico dirigido desde el vértice de origen f . En cada iteración, se selecciona el vértice con el valor de distancia mínima mostrado por un círculo grande

Análisis

Teorema 7.9 (corrección) para cada vértice $v \in S$ en cualquier momento durante la ejecución del algoritmo de la ruta más corta de Dijkstra , la ruta $P_{s,v}$ obtenido por el algoritmo es la ruta más corta entre el vértice de origen s y el vértice v .

Habremos demostrado la corrección del algoritmo al demostrar este teorema, ya que el conjunto S contendrá todos los vértices de la gráfica al final del algoritmo.

Prueba Usaremos la inducción para la prueba como en [11]. Para el caso base, $d(s) = 0$ y $S = \{s\}$ cuando $|S| = 1$ y por lo tanto $P_{s,s}$ Es el camino más corto.

Supongamos que agregando un vértice $v \notin S$ a S cuando el tamaño de S es k y $u \in S$ Es un vértice vecino de v en el camino más corto. $P_{s,v}$ del vértice fuente s al vértice v . Consideré cualquier camino arbitrario P de s a v . Nuestra hipótesis es que el peso total de este camino es al menos tan alto como el peso total de $P_{s,v}$. Deje que el vértice a sea el último vértice en P justo antes de que salga de S y el vértice $b \in V \setminus S$. Sea el primer vértice que es el vecino del vértice a en este camino, como se muestra en la figura 7.9. Sabemos el peso total del camino. $w(P_{s,u})$, $w(P_{s,a})$, $w(a,b)$ y $w(v,u)$. Es la distancia mínima al vértice u de la fuente de vértice s por la hipótesis inductiva. Si $w(a,b) < w(u,v)$ el algoritmo habría seleccionado el borde (a, b) en lugar del borde (u, v) . Por lo tanto, $w(P) \geq w(P_{s,v})$ lo que significa $P_{s,v}$ encontrado durante la iteración $k + 1$ del algoritmo es el camino más corto desde el vértice s al vértice v . Tenga en cuenta que nos hemos basado en la inexistencia de bordes de peso negativo. □

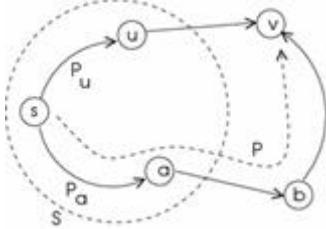


Fig. 7.9 Caminos alternativos entre dos vértices.

Necesitamos para ejecutar el bucle del algoritmo de $O(n)$ veces para n vértices, ya que procesar un solo vértice en cada iteración. También tenemos que encontrar la distancia más pequeña de vértices sin procesar al vértice fuente s de $O(m)$ tiempo ya que es posible que tengamos que considerar todos los bordes para encontrar el valor mínimo. Por lo tanto, la complejidad del tiempo de este algoritmo es $O(nm)$ en esta implementación directa.

Podemos mejorar el rendimiento de este algoritmo utilizando una cola de prioridad. En este caso, utilizaremos tres operaciones de colas de prioridad: Insertar, ExtractMin, y DecreaseKey. Necesitamos insertar todos los vértices en la cola Q mediante la operación Insertar, encontrar el valor mínimo de la cola mediante la operación ExtractMin y DecreaseKey durante la relajación, donde actualizamos los valores de distancia de los vecinos del vértice seleccionado. Cuando se utiliza un min-heap binario como la cola de prioridad, el tiempo para construir la cola lleva tiempo $O(n)$. Necesitamos n operaciones de ExtractMin para n vértices cada uno con $O(\log n)$ tiempo y $O(m)$ pasos de relajación con DecreaseKey durante la relajación, cada uno con $O(\log n)$ hora. Por lo tanto, el tiempo total tomado es $O((n+m)\log n)$. Un montón de Fibonacci que tiene un amortizado $O(\log n)$ el tiempo para la operación ExtractMin y el tiempo amortizado $O(1)$ para la operación DecreaseKey se pueden usar en lugar del binario min-heap. En esta implementación, la complejidad del tiempo se reduce a $O(n \log n + m)$.

7.3.1.2 Algoritmo de Bellman-Ford

Se es posible tener pesos negativos en algunos gráficos y en tales casos, de Dijkstra algoritmo SSSP no proporciona correctas rutas más cortas desde un vértice fuente

como hemos contado con pesos no negativos para el correcto funcionamiento del algoritmo. El algoritmo de Dijkstra se basa en el supuesto de que una ruta más corta consiste en subpaths más cortos y pequeños . Dejar $p = \{v_1, v_2, \dots, v_k\}$ Ser un camino más corto desde un vértice fuente. v_1 a un vértice de destino v_k . Entonces para $2 \leq i < k$, $p = \{v_1, \dots, v_i\}$ También es una ruta más corta para que el algoritmo funcione correctamente. Claramente, esta suposición es válida solo para ponderaciones de borde no negativas. Los bordes de peso negativo se encuentran en algunas aplicaciones de la vida real, como el comercio de divisas y los flujos de costos mínimos, por lo tanto, existe la necesidad de un algoritmo de ruta más corta en presencia de bordes de peso negativo.

El algoritmo dinámico proporcionado por Bellman y Ford funciona en presencia de bordes con pesos negativos, sin embargo, solo detectará ciclos negativos cuando haya uno [2]. Un ciclo negativo en una gráfica G es un ciclo $\{v_0, v_1, \dots, v_k, v_0\}$ tal que $w(v_0, v_1) + w(v_1, v_2) + \dots + w(v_k, v_0) < 0$

El funcionamiento de este algoritmo es simple, realiza la relajación para cada vértice progresivamente que es $1, \dots, n-1$ salta lejos del vértice de origen s para permitir cambios a lo largo del camino más largo que es $n-1$ saltos como se muestra en el algoritmo

7.11. Utilizamos la matriz D para los valores de distancia y la matriz P para las identidades de los vértices predecesores en el árbol. La ejecución de este algoritmo en un ejemplo de gráfico no dirigido se muestra en la Fig. 7.10 .

Algorithm 7.11 BellFord_SSSP

```

1:  $D[x] \leftarrow 0$ 
2: for all  $i \neq s$  do
3:    $D[i] \leftarrow \infty$ 
4:    $P[i] \leftarrow \perp$ 
5: end for
6: for  $k = 1$  to  $n-1$  do
7:   for all  $(u, v) \in E$  do
8:     if  $D[u] > D[v] + w(u, v)$  then
9:        $D[u] \leftarrow D[v] + w(u, v)$ 
10:       $P[u] \leftarrow v$ 
11:    end if
12:   end for
13: end for
14: for all  $(u, v) \in E$  do
15:   if  $D[u] + w(u, v) > D[v]$  then
16:     return false
17:   end if
18: end if
19: end for

```

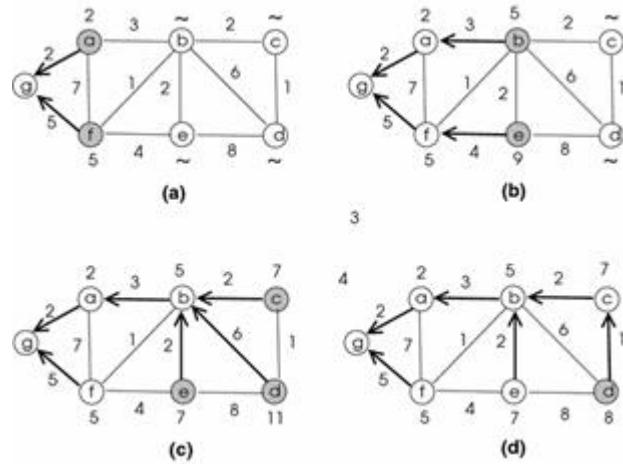


Fig. 7.10 Ejecución del algoritmo de Bellman-Ford desde el vértice de origen g en 4 iteraciones. Los vértices visitados en cada iteración se muestran en gris y el valor de distancia actual de un vértice se muestra a su lado

Análisis

El siguiente lema ayuda a probar la exactitud de este algoritmo [12]:

Lemma 7.2 (principio de optimalidad) $G = (V, E)$ un grafo ponderado sin ciclos negativos y dejar que u y v sea dos vértices de G . Sea P el camino más corto entre u y v con un máximo de k bordes y sea (w, v) el borde final en este camino. Entonces, (o) es un camino más corto de u a w con a lo más b bordes

Prueba de que R es un más corto $u - w$ camino que P y $|E(R)| \leq k - 1$. Entonces, $w(E(R)) + w(w, v) < w(E(P))$. Si $v \notin R$, entonces $R \cup \{(v, w)\}$ es un corto $u - v$ camino que P (Caso 1). De otra manera $R \cup \{(w, v)\}$ Es un ciclo no negativo y por lo tanto el costo de $R_{\{u,v\}}$, $w(E(R_{\{u,v\}})) = w(E(R_{\{u,v\}}) + w(w, v) - w(E(R_{\{u,w\}}) + w(w, v)) < w(E(P)) - w(E(R_{\{u,w\}})(w + v)) \leq w(E(P))$ (Caso 2). En ambos casos, existe una contradicción con el supuesto de que P es el camino más corto entre los vértices u y v con un máximo de k bordes. \square

Teorema 7.10 (corrección) Bellman-Ford algoritmo calcula correctamente caminos SSSP de una fuente vértice s a todos los demás vértices de un grafo no dirigido o dirigido cuando no hay ciclos negativos. Dejar $d(s, v_i)$ ser la etiqueta de distancia de vértice v_i después de la iteración i y $dist(s, v_i)$ Ser la distancia (camino más corto) del vértice v_i a la fuente del vértice s . Más específicamente, afirmamos que después de k iteraciones del algoritmo, $d(s, v_i)$ es a lo más $dist(s, v_i)$.

Prueba Nosotros demostraremos este teorema por inducción sobre el número de iteraciones.

- Caso Base : Las etiquetas de distancia de cada vértice distinto de la fuente de vértice s tener etiquetas infinito cuando el algoritmo comienza. Después de la iteración $i = 1$, solo los vecinos del vértice s tendrán una etiqueta de distancia tal que $\forall v \in N(s), d(v, s) = w(s, u)$. Por lo tanto, todos estos vecinos tendrán la distancia más corta a s cuando $k = 1$.
- Caso inductivo : Consideremos la iteración k y asumamos que el teorema es válido para todos $i < k$. Sea P el más corto $s - v_k$ camino con en la mayoría de k bordes y y (v_{k-1}, v_k) ser el ultimo borde de P . Por el lema 7.2, la parte del camino P hasta el vértice v_{k-1} ($P_{s, v_{k-1}}$) es un camino más corto entre s y v_{k-1} ; y por la hipótesis de la inducción, $dist(s, v_{k-1}) \leq w(E(P_{s, v_{k-1}}))$ después de la $(k-1)$ la iteración Después de la k^{a} iteración, tenemos $dist(s, v_k) \leq dist(s, v_{k-1}) + w(v_{k-1}, v_k) \leq w(E(P))$

El razonamiento anterior es válido cuando no hay ciclos negativos en el gráfico. Ahora queremos probar el uso de la contradicción de que este algoritmo devuelve falso cuando hay un ciclo negativo. Supongamos que el gráfico G contiene un ciclo negativo $C = \{v_0, v_1, \dots, v_k, v_0\}$ tal que $\sum_{i=1}^k w(v_i, v_{i+1}) < 0$ con $v_{k+1} = v_0$ y el algoritmo devuelve verdadero . Hay un camino desde la fuente vértice s a v_k Y a todos los demás vértices de C y deja $d(v_i)$ Ser la distancia obtenida en la primera parte del algoritmo utilizando la relajación. Desde que asumimos que el algoritmo devuelve verdadero sin detectar ciclos negativos, $d(v_{i+1}) \leq d(v_i) + w(v_i, v_{i+1})$ para $i = 1, \dots, k$. Cuando sumamos todos los vértices del ciclo, obtenemos

$$\sum_{i=1}^k d(v_{i+1}) \leq \sum_{i=1}^k (d(v_i) + w(v_i, v_{i+1}))$$

$$\sum_{i=1}^k d(v_{i+1}) \leq \sum_{j=1}^k d(v_j) + \sum_{j=1}^k w(v_i, v_{j+1})$$

Como sumamos durante el ciclo C , $\sum_{i=1}^k d(v_{i+1}) = \sum_{i=1}^k d(v_i)$ y cancelar en la ecuación anterior resulta en lo siguiente.

$$0 \leq \sum_{j=1}^k w(v_i, v_{j+1})$$

Esto contradice nuestra suposición inicial y, por lo tanto, el algoritmo de Bellman-Ford devuelve falso cuando hay un ciclo negativo en la gráfica. \square

Teorema 7.11 (complejidad) El algoritmo de Bellman-Ford tiene una complejidad de tiempo de $O(nm)$.

Prueba tenemos que tener $n-1$ iteraciones del bucle for externo para considerar la ruta más larga en un gráfico, ya que puede haber $n-1$ cambios de la distancia de un vértice sobre este camino más largo. Puede haber a lo sumo m verificación de borde en cada iteración del bucle interno en la línea 7 y, por lo tanto, la complejidad de tiempo total de este algoritmo es $O(nm)$. Por lo tanto, es un algoritmo más lento que el algoritmo SSSP de Dijkstra , sin embargo, permite bordes de peso negativo que pueden ser necesarios en aplicaciones de la vida real. \square

7.3.1.3 Algoritmo de Dijkstra en paralelo

Nos podemos formar una versión paralela de Dijkstra algoritmo de SSSP de una manera similar al método del algoritmo de Prim paralelo. La matriz de adyacencia ponderada A se divide en columnas , de manera que cada proceso p_i se asigna n / k columnas consecutivas de A . Entonces cada uno p_i calcula los valores n / p de la matriz l . La comunicación entre los procesos es similar al algoritmo Prim paralelo y el rendimiento es el mismo que este algoritmo [10].

7.3.1.4 Algoritmos distribuidos

En un entorno de red, nuestro objetivo es que cada nodo de la red calcule las rutas más cortas de sí mismo a todos los demás nodos de la red.

7.3.1.5 Algoritmo de Bellman – Ford distribuido sincrónico

Nos podemos esbozar un algoritmo distribuido síncrono (DBF_SSSP) usando el método de algoritmo Bellman-Ford en un entorno de red. Existe un nodo especial llamado la raíz que inicia la ronda síncrona por el mensaje de ronda en un árbol de expansión construido antes de la ejecución del algoritmo. Cada nodo intercambia su valor de distancia al nodo fuente con sus vecinos mediante el mensaje de actualización en cada ronda. Cualquier nodo i que encuentre que tiene un camino más corto al nodo fuente a través de un vecino j hace que j sea su padre y actualiza su distancia a la fuente agregando el peso del borde (i , j) a la distancia del nodo j . Toda esta operación es análoga al algoritmo de Bellman-Ford en una red. Habrá $n-1$ rondas para ser iniciadas por la raíz como en el caso secuencial, por lo tanto, la raíz debe conocer el número de nodos en la red. Debido a la

incertidumbre en la secuencia de entrega de mensajes en una ronda, un mensaje de actualización puede llegar a un nodo antes de un mensaje de ronda. Por lo tanto, hemos incluido una variable booleana round_recieve d que se comprueba por cada nodo antes de actualizar las distancias. Cuando todos los mensajes de los vecinos se reciben junto con un mensaje de ronda, se realiza la actualización y otra variable booleana round_overse establece en true para habilitar la convergencia de mensajes de sincronización. Una sola ronda de este algoritmo se muestra en el algoritmo 7.12.

Algorithm 7.12 SDBF_SSSP

```

1: message types round, update
2: int i, j, my_dist, dist
3: set of int received ← Ø;
4: boolean round_over ← false, round_recv ← false
5: while ~round_over do           // A single round executed by each node except the source
6:   receive msg(i)
7:   case msg(i).type of
8:     round(k) :      send update(k,my_dist) to N(i)
9:     round_recv ← true
10:    update(k,dist) : received ← received ∪ {j}
11:      if received = N(i) ∧ round_recv then
12:        for all j ∈ N(i) do
13:          if my_dist > (dist + wij)
14:            my_dist ← dist + wij
15:            parent ← j
16:        round_over ← true
17: end while

```

Teorema 7.12 *SDBF_SSSP* el algoritmo encuentra correctamente las distancias APSP desde un nodo fuente en rondas $O(n)$ usando mensajes $O(nm)$.

Prueba Dado que el algoritmo distribuido tiene la misma lógica que el algoritmo secuencial, podemos concluir que cada nodo encuentra su distancia a un nodo fuente correctamente. Hemos notado que la raíz necesita ejecutarse. $n - 1$ las rondas para tener en cuenta la ruta más larga de la red, por lo tanto, la complejidad del tiempo en las rondas para este algoritmo es $O(n)$. Cada borde en la red se usa para enviar mensajes de actualización en ambas direcciones en cada ronda, lo que da como resultado un total de $2m$ de mensajes por ronda. El número total de mensajes intercambiados será, por lo tanto, $O(nm)$. \square

7.3.2 Rutas más cortas en todos los pares

En un caso más general, es posible que tengamos que descubrir las rutas más cortas de todos los vértices a todos los otros vértices en el gráfico, que se denomina problema de rutas más cortas de todos los pares (APSP). Como primer enfoque, podemos ejecutar el algoritmo SSSP de Dijkstra para cada vértice del gráfico, lo que resulta en $O(n^2 \log n)$ complejidad del tiempo. Cuando el gráfico tiene bordes con pesos negativos, no podemos usar el algoritmo de Dijkstra y usar el algoritmo de Bellman-Ford para este propósito produce una complejidad de tiempo de $O(n^2m)$. Considera correrlo por n vértices. Buscaremos algoritmos con mejores rendimientos cuando tratemos con bordes de peso negativo y uno de estos enfoques se debe a Floyd-Warshall, que se describe en la siguiente sección.

7.3.2.1 Floyd–Algoritmo Warshall

Floyd–Warshall Algoritmo (*FW_APSP*) resuelve el problema APSP en tiempo lineal usando programación dinámica. Como se practica a menudo en la programación dinámica, el problema se divide en subproblemas más pequeños que luego se resuelven para obtener resultados intermedios que se utilizarán en la solución general. En este algoritmo, se

permiten bordes de peso negativo, pero no los ciclos negativos. Utiliza el método de relajación que hemos visto, esta vez para la distancia entre todos los pares de vértices usando cada vértice como un pivote en secuencia. Por conveniencia, los vértices están etiquetados con números enteros. Consideremos el camino más corto, p_{ij} , con peso $d_{ij}^{(k)}$ entre dos vértices cualesquiera $i, j \in V$ con elementos tomados de $1, 2, \dots, k$. No hay vértices intermedios cuando $k = 0$ y por lo tanto $d_{ij}^{(0)} = w_{ij}$. Podemos definir $d_{ij}^{(k)}$ recursivamente como sigue [5]:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1 \end{cases}$$

(7.4)

Ahora podemos implementar el algoritmo Floyd-Warshall como se muestra en el algoritmo 7.6. Tenemos dos bucles anidados para aplicar la relajación a la distancia entre cada par de vértices (i, j) para ver si la distancia entre estos vértices es más corta a través de un vértice de pivote k que su distancia actual. Cada vértice se asigna como el pivote en secuencia y, por lo tanto, tenemos otro bucle externo para seleccionar los vértices de pivote que dan como resultado tres bucles anidados como se muestra en

Tenemos la matriz de distancia $D[n, n]$ con elementos d_{ij} mostrando la distancia actual entre los vértices i y j que se inicializa a infinito para los vértices que no están conectados directamente y al peso del borde entre ellos si son vecinos. La matriz predecesora $P[n, n]$ tiene entradas p_{ij} que muestra el primer vértice sobre la ruta más corta actual entre los vértices i y j .

Algorithm 7.13 FW_APSP

```

1: Input :  $G(V, E, w)$                                  $\triangleright$  connected, weighted directed, or undirected graph  $G$ 
2: Output :  $D[n, n]$  and  $P[n, n]$                    $\triangleright$  distances and predecessors of vertices
3:
4: for  $i = 1$  to  $n$  do                                      $\triangleright$  initialize
5:   for  $j = 1$  to  $n$  do
6:     if  $(i, j) \in E$  then
7:        $D[i, j] \leftarrow w(i, j)$ ,  $P[i, j] \leftarrow j$ 
8:     else
9:        $D[i, j] \leftarrow \infty$ ,  $P[i, j] \leftarrow \perp$ 
10:    end if
11:   end for
12: end for
13:
14: for  $k = 1$  to  $n$  do                                      $\triangleright$  pivot vertex
15:   for  $i = 1$  to  $n$  do
16:     for  $j = 1$  to  $n$  do
17:       if  $D[i, k] + D[k, j] > D[i, j]$  then            $\triangleright$  relaxation
18:          $D[i, j] \leftarrow D[i, k] + D[k, j]$ ,  $P[j] \leftarrow k$ 
19:       end if
20:     end for
21:   end for
22: end for

```

La corrección se deriva de la regla de relajación, ya que siempre mejoramos los caminos más cortos utilizando todos los pivotes posibles. Tenemos tres bucles anidados que se ejecutan n veces, lo que resulta en una complejidad de tiempo de $O(n^3)$ para este algoritmo con la inicialización teniendo $O(n^2)$ hora. En la figura 7.11 se muestra un pequeño gráfico de ejemplo.

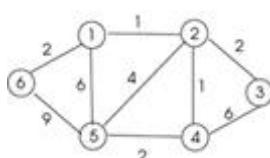


Fig. 7.11 Gráfico de muestra para el algoritmo FW_APSP

Los contenidos de la matriz de distancia inicialmente y para $k = 1$ en secuencia se muestran a continuación con los contenidos modificados mostrados en negrita

$$D^{(0)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & \infty & \infty & 6 & 2 \\ 1 & 0 & 2 & 1 & 4 & \infty \\ 2 & \infty & 2 & 0 & 6 & \infty \\ 3 & \infty & 1 & 6 & 0 & 2 \\ 4 & \infty & 6 & 4 & \infty & 2 \\ 5 & 6 & 4 & \infty & 2 & 0 \\ 6 & 2 & \infty & \infty & 9 & 0 \end{pmatrix} \rightarrow D^{(1)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & \infty & \infty & 6 & 2 \\ 1 & 0 & 2 & 1 & 4 & 3 \\ 2 & \infty & 2 & 0 & 6 & \infty \\ 3 & \infty & 1 & 6 & 0 & 2 \\ 4 & \infty & 6 & 4 & \infty & 2 \\ 5 & 6 & 4 & \infty & 2 & 0 \\ 6 & 2 & 3 & \infty & \infty & 8 \\ 7 & 0 & \infty & \infty & 8 & 0 \end{pmatrix}$$

La matriz D se muestra abajo para $k = 2$. No hay cambio en los valores de distancia cuando el vértice 3 es el pivote y, por lo tanto, mostramos valores D cuando $k = 4$.

$$D^{(2)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & \mathbf{3} & 2 & 5 & 2 \\ 1 & 0 & 2 & 1 & 4 & 3 \\ 2 & 3 & 2 & 0 & \mathbf{3} & \mathbf{6} \\ 3 & 2 & 1 & 3 & 0 & 2 \\ 4 & 5 & 4 & 6 & 2 & 0 \\ 5 & 2 & 3 & 6 & 4 & 7 \\ 6 & 2 & 3 & 6 & 4 & 7 \end{pmatrix} \rightarrow D^{(4)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 3 & 2 & 5 & 2 \\ 1 & 0 & 2 & 1 & 4 & 4 \\ 2 & 3 & 2 & 0 & 3 & 6 \\ 3 & 2 & 1 & 3 & 0 & 2 \\ 4 & 5 & 4 & 6 & 2 & 0 \\ 5 & 2 & 3 & 6 & 4 & 0 \\ 6 & 2 & 3 & 6 & 4 & 0 \end{pmatrix}$$

No hay más cambios en los contenidos de la matriz D cuando $k = 5$ y $k = 6$. La matriz P se muestra a continuación, que muestra el primer vértice en la ruta más corta desde un vértice i hasta j.

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 2 & 2 & 2 & 2 & 6 \\ 1 & 0 & 3 & 4 & 4 & 1 \\ 2 & 2 & 2 & 0 & 2 & 2 \\ 3 & 2 & 2 & 2 & 0 & 5 \\ 4 & 2 & 4 & 4 & 4 & 0 \\ 5 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

7.3.2.2 APSP paralelo utilizando el algoritmo de Dijkstra

Nosotros podemos utilizar de Dijkstra algoritmo PSSS a encontrar rutas APSP de dos maneras diferentes métodos como vértices particiones y cálculo particiones como se describe en [10].

Vías SSSP particionadas en vértice

En el primer enfoque, los vértices se dividen de manera uniforme en todos los procesos del sistema de procesamiento paralelo. Cada p_i luego calcula los SSSP para todos los vértices de los que es responsable. La matriz de distancia se replica en cada nodo y, por lo tanto, no hay comunicación entre procesos. El tiempo de ejecución paralelo, en este caso, es $T_p = \Theta(n^2)$ y dado que el tiempo secuencial es $T_s = \Theta(n^3)$ cuando el algoritmo SSSP de Dijkstra de $\Theta(n^2)$ la complejidad del tiempo se ejecuta para n vértices, la aceleración S obtenida y la eficiencia E es

$$S = \frac{\Theta(n^3)}{\Theta(n^2)} = n, \quad E = \Theta(1)$$

(7.5)

Si el número de procesos k es menor que el número de nodos, este algoritmo tiene un buen rendimiento, de lo contrario, se escalará mal.

Caminos SSSP con particiones de computación

Podemos tener el algoritmo SSSP ejecutándose en una serie de procesos paralelos cuando $k > n$ como sigue. Suponiendo que tenemos k procesos disponibles para computación paralela, asignamos procesos k / n a cada vértice y luego ejecutamos procesos k / n en paralelo para cada vértice como se describe en la Sección. 7.3.1.3 cuando se paraliza el algoritmo SSSP de Dijkstra . En otras palabras, tenemos n cálculos SSSP paralelos, cada uno de los cuales se maneja mediante procesos n / k .

$$T_p = \Theta(n^3/k) + \Theta(n \log k)$$

$$S = \frac{\Theta(n^3)}{\Theta(n^3/k) + \Theta(n \log k)} = n, \quad E = \frac{1}{1 + \Theta((k \log k)/n^2)}$$

(7.6)

7.3.2.3 Floyd paralelo - Algoritmo Warshall

Nos podemos formar una versión paralela de FW_APSP Algoritmo dividiendo la tarea del cálculo de la matriz D entre los procesos p . Describiremos una posible partición utilizando el mapeo de bloques 2-D para este problema como se describe en [10]. En este enfoque, D se divide en bloques de tamaño. $(n/\sqrt{p}) \times (n/\sqrt{p})$ A cada proceso se le asigna un solo bloque. Los procesos se posicionan en una grilla de \sqrt{p} por \sqrt{p} y un proceso $p_{i,j}$ tiene un subbloque con esquina superior izquierda $((i-1)n/\sqrt{p}+1, (j-1)n/\sqrt{p}+1)$ y una esquina inferior derecha $in/\sqrt{p}, jn/\sqrt{p}$ como se muestra en la figura 7.12 .

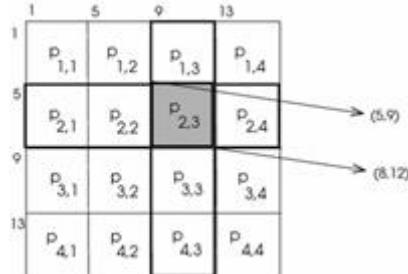


Fig. 7.12 partición 2-D de 16×16 Matriz D para una gráfica con 16 vértices a 16 procesos en PFW_APSP algoritmo. El proceso $p_{2,3}$ por ejemplo, tiene las coordenadas de la esquina superior izquierda $(5, 9)$ y las coordenadas de la parte inferior derecha $(8, 12)$. Este proceso necesita todas las entradas de matriz mantenidas por procesos. $p_{2,1}, p_{2,2}, p_{2,4}$ En su fila y por procesos. $p_{1,3}, p_{3,2}, p_{4,3}$ en su columna para poder calcular sus valores de subbloques D para la iteración actual

Cada proceso $p_{i,j}$ calcula su subbloque de D durante cada iteración, pero necesita los elementos mantenidos por los procesos en su fila y columnas para poder realizar este cálculo. Por lo tanto, necesitamos una transmisión de valores D a lo largo de filas y columnas mantenidas por procesos. El algoritmo 7.14 muestra una forma de realizar este algoritmo utilizando la partición 2-D.

Algorithm 7.14 PFW_APSP

```

1: Input : subBlock of  $D^0$                                  $\triangleright$  my position of the distance matrix
2: Output :  $D_{i,j}^k$                                       $\triangleright$  shortest path values for my subblock
3:
4: for  $j = 1$  to  $n$  do
5:   broadcast my segment of  $D^{(k-1)}$  to all processes in       $\triangleright$  update distances and next node
6:   broadcast my segment of  $D^{(k-1)}$  to all processes in
7:   receive  $D^{(k-1)}$  values from processes in my row and column
8:   compute  $D^{(k)}$  for my subblock
9: end for

```

Cada proceso $p_{i,j}$ sostiene $\frac{n}{\sqrt{p}}$ Elementos de la fila o columna k th que se emiten en $\Theta(n \log p / \sqrt{p})$ hora. La sincronización en la línea 7 requiere $\Theta(\log p)$ tiempo y computación de $\frac{n^2}{p}$. Los valores asignados a un proceso requieren $\Theta(n^2/p)$ tiempo resultante un tiempo total de procesamiento paralelo de

$$T_p = \Theta\left(\frac{n^3}{p}\right) + \Theta\left(\frac{n^2}{\sqrt{p}} \log p\right)$$

(7.7)

Sabemos que el algoritmo secuencial tiene una complejidad de tiempo de $\Theta(n^3)$. Por lo tanto, la aceleración S se puede establecer de la siguiente manera:

$$S = \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta(n^2 \log p / \sqrt{p})}$$

(7.8)

Por lo tanto la eficiencia E es,

$$E = \frac{1}{1 + \Theta(\sqrt{p} \log p / n)}$$

(7.9)

De las ecuaciones de aceleración y eficiencia, podemos concluir que este algoritmo puede emplear $\Theta(n^2 / \log^2 n)$ procesos. El paso de sincronización se puede omitir para dar como resultado una versión canalizada más rápida del algoritmo 2-D con eficiencia $1/(1 + \Theta(p/n^2))$ [10].

7.3.2.4 Floyd distribuido - Algoritmo Warshall

En una configuración distribuida, necesitamos n nodos de la red, cada uno de los cuales determina su SSSP a todos los demás nodos al final del algoritmo. Podemos tener una versión distribuida síncrona del algoritmo FW_APSP con la modificación de que cada nodo i ahora solo tiene un vector $D_{i,n}$ que muestra su mejor estimación de la distancia más corta a todos los demás nodos de la red. Este vector, de hecho, corresponde a la fila que el nodo i tiene en la matriz de distancia D en el algoritmo secuencial. Un vector local $P_{i,n}$ mantenido en el nodo i muestra el primer nodo a lo largo de la estimación de la ruta más corta actual del nodo i a todos los demás nodos. Para poder adaptar el algoritmo

secuencial a este entorno de red en su totalidad, necesitamos que el nodo pivote k transmita su vector local $D_k[n]$ de manera que cada nodo i puede comparar los valores de D_k con los valores en D_i y actualizar D_i y P_i en consecuencia, como en el algoritmo 7.15, donde se muestra una sola ronda para un nodo i . Asumimos lo siguiente:

- Hay un nodo especial llamado la raíz que inicia cada ronda.
- Un árbol de expansión se construye de antemano para enviar y recibir mensajes de control como, por ejemplo, redondeo de difusión y redondeo convergecast .
- Los nodos tienen identificadores enteros únicos en el rango $\{1, \dots, n\}$.
- La raíz envía el número r redondo en cada ronda, que se interpreta como el parámetro k en el algoritmo secuencial. Cualquier nodo que encuentre su identificador igual a r emitirá sus valores D para que todos los demás nodos lo comparen.

Algorithm 7.15 DFW_APSP

```

1: set of int  $D[n], P[n]$                                 ▷ local distance and next node vectors
2:
3:  $D[i] \leftarrow 0, P[i] \leftarrow i$                          ▷ initialize
4: for  $j = 1$  to  $n$  do
5:   if  $j \in N(i)$  then
6:      $D[j] \leftarrow w(i, j), P[j] \leftarrow j$ 
7:   else
8:      $D[j] \leftarrow \infty, P[j] \leftarrow \perp$ 
9:   end if
10: end for
11:                                                       ▷ a single round
12: receive  $round(r)$  message
13: if  $i = r$  then broadcast  $D_k[n]$                       ▷ if I am the pivot, broadcast  $D_k[n]$ 
14: else receive  $D_k[n]$  from node  $k$                    ▷ otherwise receive the pivot vector
15: end if
16: for  $j = 1$  to  $n$  do
17:   if  $D[j] + D[k] < D[j]$  then
18:      $D[j] \leftarrow D[j] + D[k]$ 
19:      $P[j] \leftarrow k$ 
20:   end if
21: end for

```

La Figura 7.13 muestra la ejecución de este algoritmo en una red pequeña. Difusión de D_k Vector es el principal cuello de botella en este algoritmo. Podemos hacer que el nodo r envíe su vector D a la raíz, que luego transmite este vector a todos los nodos sobre el árbol de expansión. Tóquegproporcionó una versión asíncrona de este algoritmo al reducir el conjunto de nodos que deberían recibir la D_k Valores con una complejidad de tiempo. $O(n^2)$ y un mensaje de complejidad $O(nm)$ [19].

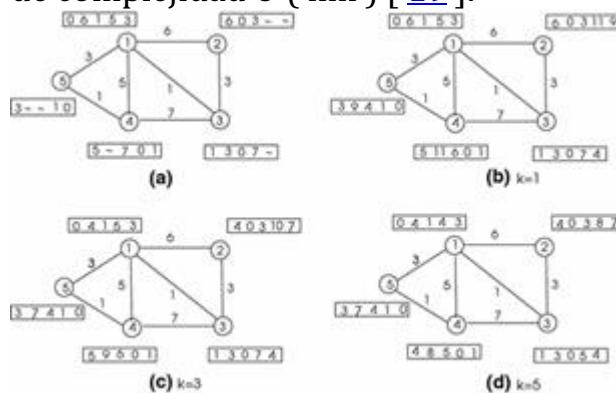


Fig. 7.13 Ejecución del algoritmo DFW_APSP en una red pequeña. Los vectores actuales en cada nodo se muestran junto a ellos. En iteraciones $k = 2$ y $k = 4$, no hay cambios en los valores de distancia anteriores y estos no se muestran. Despues de cinco iteraciones, se determinan todas las rutas más cortas.

7.4 Notas del capítulo

Observamos los algoritmos de gráficos ponderados en este capítulo, donde los bordes de un gráfico tienen pesos asociados. Revisamos un problema fundamental en tales gráficos; encontrar el MST y los métodos descritos para la construcción del MST de un gráfico ponderado, no dirigido y conectado en este capítulo. El problema MST puede resolverse mediante codiciosos algoritmos secuenciales en tiempo polinomial. Estos algoritmos se deben a Boruvka , Kruskal y Prim en orden cronológico, cada uno con un enfoque diferente. Mostramos en detalle cómo formar una versión paralela del algoritmo de Prim encontrando los MWOE en paralelo y también describimos formas de paralelizar el algoritmo de Boruvka . El estudio de los algoritmos MST se proporciona en [9] y en [17].

En una configuración de red donde cada nodo del gráfico es un nodo computacional, nuestro primer enfoque nuevamente es considerar estos algoritmos secuenciales. Describimos cómo obtener una versión de red distribuida del algoritmo de Prim y revisamos un método basado en el algoritmo de Boruvka para el procesamiento distribuido en una red. El contenido de los mensajes y los instantes que se envían son importantes en una red, tal como lo discutimos. Nos interesan las complejidades de tiempo y mensaje en los algoritmos de red.

En cuanto a las conversiones entre los tres métodos fundamentales, ya hemos realizado Seq (Prim) → Par (Prim) y Seq (Prim) → Dist (Prim), y describimos formas de lograr Seq (Boruvka) → Par (Boruvka) y Seq (Boruvka) → Dist (Boruvka). Una cosa a considerar es si podemos tener conversiones como Par (Prim) ↔ Dist (Prim). Una forma de lograr esto sería dividir el gráfico en k particiones no superpuestas y distribuir los subgrafos G_0, \dots, G_{k-1} a procesos p_0, \dots, p_{k-1} de la red. El proceso raíz p_0 ejecuta el algoritmo Prim secuencial en su partición hasta que se cumplen los bordes que cruzan las particiones. Luego puede realizar el proceso como en la versión distribuida del algoritmo de Prim solicitando MWOE de cada partición en lugar de nodos individuales. Cada nodo es una partición ahora y lo que hemos descrito es una conversión de algoritmo distribuido a algoritmo paralelo para este método.

Luego consideraremos los problemas del camino más corto; estos problemas se consideran como la ruta más corta de una sola fuente (SSSP) o todos los problemas de las rutas más cortas de pares (APSP). Un algoritmo fundamental debido a Dijkstra resuelve el problema de SSSP pero no funciona con bordes de peso negativo o ciclos negativos. El algoritmo de Bellman-Ford funciona con pesos negativos e informa ciclos negativos con mayor complejidad de tiempo. Buscamos formas de tener versiones paralelas y distribuidas de estos algoritmos. Floyd: el algoritmo Warshall encuentra rutas APSP en un gráfico con bordes de peso negativo. También presentamos una versión paralela y una versión distribuida de este algoritmo. Estos algoritmos proporcionan ejemplos significativos de conversión entre una versión secuencial, paralela y distribuida del mismo método.

Ceremonias

Encuentre el MST de la gráfica de muestra de la figura [7.14](#) usando el algoritmo MST de Prim.

1.

2. Calcule el MST del gráfico representado en la Fig. 7.15 utilizando los algoritmos MST de Kruskal y Boruvka y muestre que ambos dan como resultado el mismo MST de este gráfico, ya que los pesos de los bordes son distintos.
3. Calcule las rutas más cortas de una sola fuente del vértice a del dígrafo representado en la figura 7.16 usando *Dijkstra_SSSP* algoritmo mostrando cada iteración.
4. Escribe el pseudocódigo de paralelo. *Dijkstra_SSSP* Algoritmo y ejercicio de su eficiencia.
5. Construya las rutas más cortas de una sola fuente del vértice a del dígrafo representado en la figura 7.17 usando *BF_SSSP* algoritmo mostrando cada iteración.
6. Construya las rutas más cortas de todos los pares del dígrafo representado en la Fig. 7.18 usando *FW_APSP* algoritmo mostrando cada iteración.
7. Forme la matriz de distancia D para la gráfica de la figura 7.19 y proporcione una división bidimensional de esta matriz en 4 procesos. Muestre los datos enviados por cada proceso durante la ejecución paralela del algoritmo Floyd-Warshall para las dos primeras iteraciones. Calcula los valores finales de D después de k iteraciones.
8. Modificar el algoritmo APSP distribuido *DFW_APSP* pseudocódigo para que el nodo raíz también pueda ejecutar este código mostrando el inicio y el final de cada ronda.

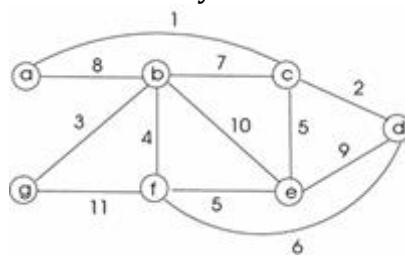


Fig. 7.14 Gráfico de muestra para el Ejercicio 1

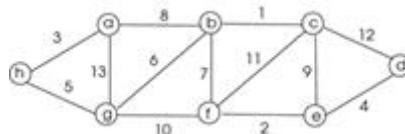


Fig. 7.15 Gráfico de muestra para el Ejercicio 2

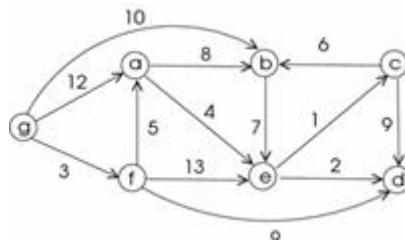


Fig. 7.16 Gráfico de muestra para el Ejercicio 3

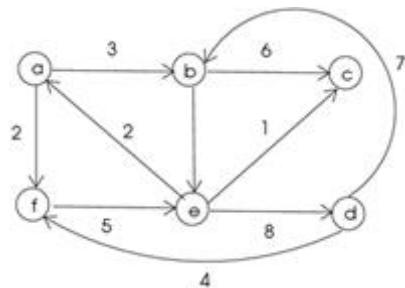


Fig. 7.17 Gráfico de muestra para el ejercicio 5

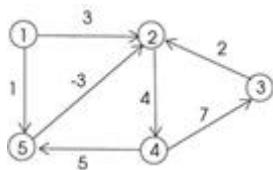


Fig. 7.18 Gráfico de muestra para el ejercicio 6

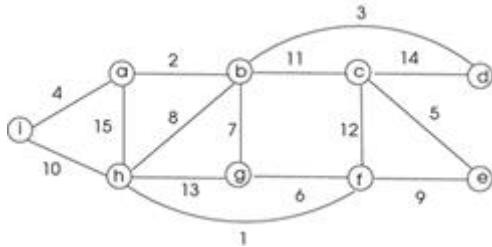


Fig. 7.19 Gráfico de muestra para el ejercicio 7

Referencias

1. Acar UA, Blelloch GE (2017) Diseño de algoritmo: paralelo y secuencial. Capítulo 18, Libro borrador, Carnegie Mellon University, Departamento de Ciencias de la Computación. <https://www.parallel-algorithms-book.com/>
2. Bellman R (1958) En un problema de enrutamiento. Q Appl Math 16: 87–90
[MathSciNet Crossref](#)
3. Boruvka O (1926) Sobre un cierto problema mínimo. Prce Mor. prrodoved . spol . v Brne III (en checo, resumen alemán), 3: 37–58
4. Chung S, Condon A (1996) Implementación paralela del algoritmo de árbol de expansión mínimo de Boruvka . Informe técnico 1297, Departamento de Ciencias de la Computación, Universidad de Wisconsin
5. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introducción a los algoritmos, 3^a ed . MIT Press, Cambridge, Capítulo 23
6. Dijkstra EW (1959) Nota sobre dos problemas en relación con gráficos. Numerische Mathematik 1: 269–271
[MathSciNet Crossref](#)
7. Erciyes K (2013) Algoritmos de grafos distribuidos para redes informáticas. Capítulo 6, Springer serie de comunicaciones y redes informáticas. Springer, Berlín
[Referencia cruzada](#)
8. Gallager RG, Humblet PA, Spira PM (1983) Algoritmos distribuidos para árboles de expansión de peso mínimo. ACM Trans Progrmm Lang Syst 5 (1): 66–77
[Referencia cruzada](#)
9. Graham RL, Hell P (1985) Sobre la historia del problema del árbol de expansión mínimo. Ann Hist Comput 7 (1): 4357
[MathSciNet Crossref](#)
10. Grama A, Gupta A, Karypis G, Kumar V (2003) Introducción a la computación paralela, 2^a ed . Capítulo 10. Addison Wesley, Boston
11. Kleinberg J, Tardos E (2005) Diseño del algoritmo. Capítulo 4. Pearson Int. Ed. ISBN-13: 978-0321295354 ISBN-10: 0321295358
12. Korte B, Vygen J (2008) Optimización combinatoria: teoría y algoritmos. Capítulo 7, 4^a ed . Springer, Berlín
13. Kruskal JB (1956) En el subárbol más corto de una gráfica y el problema del vendedor ambulante. Proc Am Math Soc 7: 4850

14. Loncar V, Skrbic S, Bala A (2013) Paralelización de algoritmos de árbol de expansión mínimo que utilizan arquitecturas de memoria distribuida. Transacción sobre tecnología de ingeniería. Volumen especial del congreso mundial de ingeniería, pp 543–554.
[Referencia cruzada](#)
15. Peleg D (1987) Computación distribuida: un enfoque sensible a la localidad Monografías de SIAM sobre matemáticas y aplicaciones discretas. Capítulo 5
- dieciséis. Prim RC (1957) Redes de conexión más cortas y algunas generalizaciones. Bell Syst Tech J 36 (6): 1389–1401
[Referencia cruzada](#)
17. Tarjan RE (1987) Estructuras de datos y algoritmos de red. SIAM, CBMS-NSF serie de conferencias regionales sobre matemáticas aplicadas (Libro 44)
18. Thorup M (2000) Conectividad de gráficos totalmente dinámica casi óptima. En: Actas del 32º simposio de la ACM sobre teoría de la computación, pp 343–350
19. Toueg S (1980) Un algoritmo distribuido de ruta más corta de todos los pares. Informe técnico RC 8327, Centro de Investigación IBM TJ Watson, Yorktown Heights, NY 10598

8. conectividad

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Un gráfico no dirigido está conectado si hay una ruta entre cualquier par de sus vértices. En un dígrafo, la conectividad implica que hay una ruta entre dos de sus vértices en ambas direcciones. Comenzamos este capítulo definiendo los parámetros de conectividad de vértice y borde. Continuamos describiendo algoritmos para encontrar vértices y puentes de gráficos no dirigidos. Luego, revisamos los algoritmos para encontrar bloques de gráficos y componentes fuertemente conectados de dígrafos. Describimos la relación entre la conectividad y los flujos de red y la comparación y revisamos los algoritmos secuenciales, paralelos y distribuidos para todos los temas mencionados.

8.1 Introducción

La conectividad es un concepto fundamental en la teoría de grafos que tiene implicaciones tanto teóricas como prácticas. Un gráfico no dirigido está conectado si hay una ruta entre cualquier par de sus vértices. En un dígrafo, la conectividad implica que hay una ruta entre dos de sus vértices en ambas direcciones. En la práctica, el estudio de la conectividad es necesario para redes de comunicación confiables, ya que la conectividad se debe proporcionar en la pérdida de bordes (enlaces) o vértices (enrutadores) en estas redes. Un corte-vértice de un gráfico G es un vértice especial en Geliminación de que desconecta G . De manera similar, la eliminación de un borde llamado puente de un gráfico conectado G desconecta G. Sería interesante detectar dichas partes de redes para mejorar la conectividad en estas regiones mediante el suministro de dispositivos y enlaces de comunicación adicionales.

Comenzamos este capítulo definiendo formalmente los parámetros de conectividad de vértice y borde. Continuamos describiendo algoritmos para encontrar vértices y puentes de gráficos no dirigidos. Los bloques son componentes conectados máximos de un gráfico sin un vértice de corte y revisamos los algoritmos para encontrar bloques de gráficos. Luego revisamos los componentes fuertemente conectados de los dígrafos junto con los algoritmos para descubrirlos. La conectividad está relacionada con los flujos de red y la comparación, como veremos. Nuestro objetivo principal en una red de flujo es encontrar el flujo máximo desde un nodo de origen a un nodo de destino y mostramos un algoritmo para encontrar el flujo máximo que se puede usar para encontrar qué tan bien conectado está un gráfico. Una coincidencia de una gráfica es un conjunto de sus bordes

disjuntos y veremos en el siguiente capítulo que se puede emplear un algoritmo de flujo para encontrar una coincidencia máxima de una gráfica bipartita.

8.2 Teoría

Definimos los parámetros básicos de conectividad en esta sección.

Definición 8.1(gráfico conectado, componente) Un gráfico (dirigido o no dirigido) está conectado si hay un recorrido entre cada par de sus vértices. Cualquier gráfico que no tenga esta propiedad está desconectado . Los subgrafos conectados máximos de una gráfica se denominan componentes .

En otras palabras, una gráfica. $G = (V, E)$ está conectado si para cada $u, v \in V$, Existe una (u, v) camino en G . El subgrafo de deleción de vértice $F = (V', E')$ de una gráfica G mostrada por $G - F$ Se obtiene eliminando todos los vértices de V' y su incidente bordes de G . Del mismo modo, el subgrafo de eliminación de bordes. $H = (V'', E'')$ de una gráfica G mostrada por $G - H$ se obtiene mediante la eliminación de todos los bordes en E'' .

Definición 8.2(gráfico biconectado) Un gráfico no dirigido conectado $G = (V, E)$ se llama biconectado si para cada vértice $v \in V$, $G - v$ está conectado.

Es decir, un gráfico conectado y no dirigido está biconectado si permanece conectado después de eliminar cualquiera de sus vértices. Un ciclo, por ejemplo, es 2-conectado. En términos prácticos, esto significa que la falla de un nodo en una red informática biconectada lo dejará aún conectado, ya que habrá rutas alternativas. Si una gráfica no está biconectada , la eliminación de al menos uno de sus vértices hará que se desconecte. Tales vértices de desconexión se llaman vértices de corte o puntos de articulación .

Definición 8.3(corte de vértice) Un corte de vértice de un gráfico G conectado es un subconjunto V' de sus vértices tales que $G - V'$ Tiene al menos dos componentes diferentes.

Cuando V' consta de un solo vértice, este vértice se llama la CUT ertex (o el punto de articulación) de G . Un gráfico completo K_n El orden n no tiene un vértice de corte, ya que no hay una eliminación de vértice único que desconecte dicho gráfico. El corte de borde de un gráfico se puede definir de manera similar a continuación.

Definición 8.4(corte de borde) Un corte de borde de un gráfico G conectado es un subconjunto E' de sus bordes tal que $G - E'$ Tiene al menos dos componentes diferentes.

Cuando E' consiste en un solo borde, este borde se llama un borde de corte o un puente . Una arista e es un puente si y sólo si no está incluido en cualquier ciclo de G . Si correo era parte de un ciclo, y luego eliminarlo de G no se desconecte G . La figura [8.1](#) muestra estos conceptos.

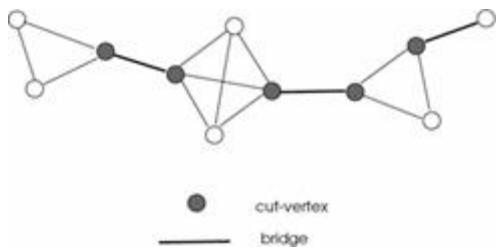


Fig. 8.1 Corte de vértices y puentes en una gráfica de muestra

8.2.1 Vertex y conectividad de borde

Definición 8.5(conectividad de vértice) La conectividad de vértice $\kappa(G)$ de un gráfico conectado G es el número mínimo de vértices que se eliminan, lo que da como resultado un gráfico desconectado o trivial.

Es la cardinalidad del corte mínimo de vértice de una gráfica. En una gráfica G de orden n , el grado máximo $\Delta(G)$ será a lo sumo $n - 1$. Por lo tanto,

$$0 \leq \kappa(G) \leq n - 1$$

(8.1)

Un gráfico G se llama k conectado si $\kappa(G) \geq k$. En otras palabras, una gráfica está conectada con k si la eliminación de k vértices desconecta la gráfica. Por lo tanto, un gráfico conectado a k también está conectado a m para cada entero m con $0 \leq m \leq k$. Para una gráfica completa K_n , $\kappa(K_n) = n - 1$ Podemos definir la conectividad de borde de manera similar a lo siguiente.

Definición 8.6(conectividad de borde) La conectividad de borde $\lambda(G)$ (o $\lambda(G)$) de un gráfico conectado G es el número mínimo de eliminación de bordes de los que se obtiene un gráfico desconectado

Un gráfico G se llama k-edge-connected si $\lambda(G) \geq k$. Por lo tanto, un gráfico conectado por el borde K también está conectado por m para cada entero m con $0 \leq m \leq k$. Para una gráfica completa K_n , $\lambda(K_n) = n - 1$. Para cada gráfica G de orden n , necesitamos eliminar como máximo $n - 1$ bordes del vértice V del grado más alto para hacer v aislado y, por lo tanto, para tener G desconectado. Por lo tanto,

$$0 \leq \lambda(G) \leq n - 1$$

(8.2)

La conectividad de vértice y la conectividad de borde de un gráfico desconectado son 0, ya que no necesitamos eliminar ningún vértice o borde para desconectarlo. Los números de conectividad de vértice y conectividad de borde del gráfico en la Fig. 8.1 son ambos la unidad.

Teorema 8.1 (Whitney [15]) Para cada gráfica G ,

$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

Prueba Un gráfico G se desconecta si se eliminan todos los bordes que inciden en un vértice v . El valor máximo de conectividad de borde será, por lo tanto, $\lambda(G)$ ya que los bordes alrededor del vértice de grado mínimo v forman un corte de borde de G y, por lo tanto, la desigualdad en el lado derecho se mantiene. Para probar el lado izquierdo de la desigualdad, consideremos, un corte de borde mínimo $C \in E$ de G que separa los vértices en G en los subconjuntos S y S' . En el peor de los casos, tendríamos todos los vértices en S conectados a todos los vértices en S' . El límite superior suelto en la conectividad para cualquier gráfico es el de un gráfico completo que es $n - 1$. Por lo tanto, $\lambda(G) = |S| \cdot |S'| \leq n - 1$. En el caso, cuando todos los vértices en S no están conectados a todos los vértices de S' , tenemos al menos una ventaja $(u, v) \notin C$ con $u \in S$ y $v \in S'$. □

8.2.2 Bloques

Definición 8.7(block) Un bloque o un componente biconectado de un gráfico G es un subgrafo conectado máximo de G sin un vértice de corte (punto de articulación)

Un bloque de un gráfico G es un conjunto máximo de bordes. Cada gráfico es una unión de sus bloques. Un bloque B de una gráfica G puede contener unos vértices de corte de G, aunque no puede tener un vértice de corte propio. Un borde es un bloque de un gráfico G si y sólo si es un puente de G . Por lo tanto, cada borde de un árbol es sus bloques y cada vértice aislado de una gráfica son sus bloques. En resumen, los bloques de una gráfica están compuestos por todos los componentes bi-conectados, todos los puentes y todos los vértices aislados. En la Fig. 8.2 se muestran los bloques de una muestra desconectada y un gráfico no dirigido .

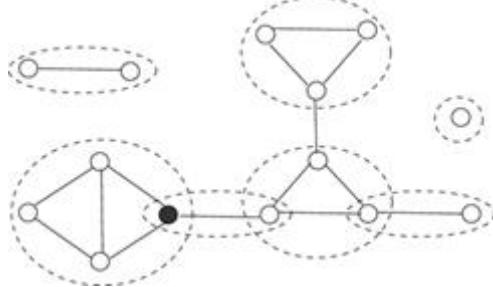


Fig. 8.2 Bloques de una gráfica no dirigida que se muestra rodeada. El vértice en negrita, por ejemplo, es un vértice de corte del gráfico pero no el vértice de corte del bloque al que pertenece

8.2.3 Teoremas de Menger

Necesitamos definir rutas disjuntas entre dos vértices de un gráfico antes de exponer los teoremas de Menger para la conectividad.

Definición 8.8(connectividad borde de dos vértices) Let u y v sea dos vértices distintos de un grafo no dirigido G . La conectividad del borde de los vértices u y v , $\lambda(u, v)$, es el menor número de bordes que se eliminarán de G para tener u y v desconectados.

Definición 8.9(la conectividad del vértice de dos vértices) Let u y v sea dos vértices distintos de un grafo no dirigido G . La conectividad de vértice de los vértices u y v , $\kappa(u, v)$, es el menor número de vértices seleccionados de $V - \{u, v\}$ que se eliminarán de G para desconectar u y v . Podemos ver inmediatamente que la conectividad de vértice del gráfico G es el mínimo de $\kappa(u, v)$ para cada par de vértices u y v .

Definición 8.10(rutas de separación de vértices) La colección de rutas entre los dos vértices u y v de un gráfico G se denomina separación de vértices(independiente) si no comparten ningún vértice distinto de u y v . El mayor número de rutas independientes entre los dos vértices u y v se denota como $\lambda(u, v)$.

Definición 8.11(rutas de separación de borde) La colección de rutas entre los dos vértices u y v de un gráfico G se denomina separación de borde (independiente del borde) si no comparten ningún borde. El mayor número de rutas independientes del borde entre los dos vértices u y v se denota como $\lambda'(u, v)$.

Ahora vamos a declarar de Menger teoremas sin probar los que proporcionan las condiciones necesarias y suficientes para un gráfico que ser kcomunicado con los o k -Edge conectado.

Teorema 8.2 (Teorema de Menger , versión de vértice) Deje $\kappa(u, v)$ sea el número máximo de rutas de separación de vértices entre los vértices u y v . Un gráfico está conectado a k si, y solo si cada par de vértices en el gráfico está conectado por al menos k rutas desunidas.

Teorema 8.3 (Teorema de Menger , versión límite) Permitir $\lambda(u, v)$ sea el número máximo de rutas de separación de borde entre los vértices u y v . Una gráfica está conectada por el borde K si, y solo si, cada par de vértices en la gráfica está conectado por al menos k trayectorias de separación de bordes.

8.2.4 Conectividad en dígrafos

La conectividad en dígrafos requiere más especificaciones ya que la conexión entre vértices no es simétrica en tales gráficos. Es decir, puede ser posible alcanzar un vértice v desde un vértice u pero no al revés. Requerimos que haya una ruta entre cada par de vértices en ambas direcciones en un dígrafo para que la conectividad se mantenga. Un dígrafo fuertemente conectado se define de la siguiente manera.

Definición 8.12(digraph fuertemente conectado) Un digraph se llama fuertemente conectado si para cada $u - v$ par de vértices, hay un camino de u a v y un camino de v a u .

Definición 8.13(componentes fuertemente conectados de un dígrafo) Un componente fuertemente conectado (SCC) de un gráfico dirigido es un subconjunto máximo de vértices que contiene una ruta dirigida desde cada vértice a todos los demás en el subconjunto.

Se pueden observar las siguientes propiedades para SCCs en dígrafos.

- Cada vértice pertenece exactamente a un SCC
- Cualquiera de los dos SCC son desunidos
- Los SCC de un gráfico G forman una partición de G

Definición 8.14(dígrafo débilmente conectado) Un dígrafo está débilmente conectado si su gráfico no dirigido subyacente está conectado.

8.3 Algoritmos de conectividad secuencial

Revisaremos los algoritmos para encontrar componentes conectados, puntos de articulación, puentes, SCC y bloques de gráficos en las siguientes secciones.

8.3.1 Encontrar componentes conectados

Siempre podemos comprobar la conectividad de un gráfico no dirigido. $G = (V, E)$ ejecutando DFS o BFS desde un vértice arbitrario v y registrando los vértices visitados en una lista V' durante la búsqueda. Si se visita vértice establecido $V' = V$, entonces G está conectado. Este algoritmo funciona ya que estas búsquedas siempre visitarán cada vértice que está conectado a través de una ruta al vértice fuente, formando un árbol de expansión arraigado en la fuente al final. El tiempo empleado será $O(n + m)$ como en el algoritmo DFS o BFS. Podemos encontrar los componentes conectados de un gráfico no dirigido utilizando el algoritmo DFS con una simple modificación; ejecuta DFS en el gráfico para obtener un bosque; Cada árbol en el bosque formado por una llamada desde el programa principal es un componente conectado como se muestra en esta versión modificada de DFS en el algoritmo 8.1. Cada vez que se realiza una devolución desde el procedimiento DFS, se han visitado todos los vértices de ese componente. Podemos realmente etiquetar los vértices con los componentes en los que se encuentran definiendo una matriz $label[1 \dots n]$, donde etiqueta [i] muestra el número del vértice componente i pertenece. El tiempo necesario para encontrar los componentes del gráfico G utilizando el algoritmo DFS es $O(n + m)$.

Algorithm 8.1 DFS_Component

```

1: Input :  $G(V, E)$ , an undirected graph
2: Output :  $C = \{C_1, C_2, \dots, C_k\}$                                  $\triangleright$  components of  $G$ 
3: boolean visited[1 ... n]
4: count  $\leftarrow 0$ 
5: for all  $u \in V$  do
6:   visited[u]  $\leftarrow false$                                           $\triangleright$  initialize
7: end for
8: for all  $u \in V$  do
9:   if visited[u] = false then
10:    count  $\leftarrow count + 1$ 
11:    DFS(u)                                                  $\triangleright$  call for each connected component
12:   end if
13: end for
14:
15: procedure DFS( $u$ )
16:   visited[u]  $\leftarrow true$                                           $\triangleright$  first visit
17:   label[u]  $\leftarrow count$ 
18:    $C_{count} \leftarrow C_{count} \cup \{u\}$ 
19:   for all  $(u, v) \in E$  do
20:     if visited[v] = false then                                      $\triangleright$  visit neighbours
21:       DFS(v)
22:     end if
23:   end for
24: end procedure

```

8.3.2 Búsqueda de puntos de articulación

Un punto de articulación o un vértice de corte de un gráfico G no dirigido es un corte de vértice que consiste en un solo vértice, por lo tanto, la eliminación de dicho vértice hará que G se desconecte. Un enrutador es el componente básico de una red de computadoras que dirige los mensajes provenientes de sus puertos de entrada a sus puertos de salida. Un enrutador que es un punto de articulación en una red de computadoras es un punto único de falla de falla que causará una desconexión y, por lo tanto, una red deficitaria. Necesitamos encontrar dichos puntos de articulación en las redes para

proporcionar enlaces adicionales a su alrededor para hacer que la red sea más robusta ante las fallas. Primero describiremos un algoritmo ingenuo para encontrar puntos de articulación de un gráfico no dirigido y luego un algoritmo basado en DFS con una mejor complejidad de tiempo.

8.3.2.1 El algoritmo ingenuo

Como un enfoque simple para encontrar el punto de articulación de un gráfico G , podemos eliminar los vértices uno por uno del gráfico G y verificar si G está conectado o no aplicando el algoritmo DFS o BFS después de cada eliminación, como se muestra en el Algoritmo 8.2. Si la eliminación de un vértice v deja G desconectado, entonces v es un punto de articulación. El bucle for se ejecuta n veces y el recorrido DFS o BFS toma $\mathcal{O}(n(n+m))$ tiempo, resultando en $\mathcal{O}(n(n+m))$ Tiempo para este algoritmo. Tendríamos que buscar algoritmos para que se usen mejores complejidades en gráficos grandes.

Algorithm 8.2 Naive_AP

```

1: Input :  $G = (V, E)$ 
2: Output : articulation points of  $G$  in  $P$ 
3: set of vertices  $L \leftarrow \emptyset$ 
4: for all  $v \in V$  do
5:    $G' \leftarrow G - \{v\}$ 
6:   run  $DFS(G', u)$  where  $u$  is any vertex in  $G'$ 
7:   record the visited vertices in  $L$ 
8:   if  $V' \neq L$  then
9:      $P \leftarrow P \cup \{v\}$ 
10:  end if
11: end for

```

8.3.2.2 Algoritmo basado en DFS

Tarjan presentó un algoritmo para encontrar puntos de articulación del gráfico G usando DFS [13]. Antes de revisar este algoritmo, recordemos la propiedad del borde posterior en un árbol DFS. Un borde posterior (u, v) de un vértice w en un árbol DFS es un borde de cualquier vértice v del subárbol arraigado en w a cualquier vértice ancestral u de w en el árbol. Desde esta definición, podemos ver que el borde (u, v) no forma parte del árbol DFS ya que forma un bucle.

Observación 4 Un vértice w con un borde posterior (u, v) en un árbol DFS de un gráfico G no puede ser un punto de articulación, ya que la eliminación de w no deja G desconectado.

Podemos ver que esto es válido ya que (u, v) aún mantiene el gráfico G conectado y, a la inversa, la eliminación de un vértice w que no tiene un borde posterior deja G desconectado y, por lo tanto, w es un punto de articulación. Ahora tenemos una propiedad para clasificar vértices; cualquier vértice que no tenga un borde posterior desde un vértice en su subárbol hasta uno de sus ancestros en un árbol DFS es un punto de articulación. La raíz rEl árbol DFS necesita un tratamiento especial ya que no tiene bordes posteriores, pero aún puede ser un punto de articulación si y solo si tiene más de un hijo. En tal caso, si dos vértices en los subárboles de los hijos de la raíz estuvieran conectados por un borde que no sea un árbol, estarían en el mismo subárbol. Por lo tanto, cuando la raíz r tiene más de un hijo, la eliminación de r dejará la gráfica G desconectada.

Observación 5 El vértice raíz de un árbol DFS de un gráfico G es un punto de articulación si y solo si tiene más de un hijo.

Podemos construir un algoritmo basado en esta propiedad solo ejecutando DFS desde cada vértice del gráfico y verificando si cada raíz tiene más de un elemento secundario. Este enfoque requiere $O(n(n + m))$ sin embargo, podemos tener un mejor rendimiento al usar la propiedad del borde posterior junto con esta propiedad raíz como se describe a continuación.

Necesitamos una forma de detectar bordes posteriores y DFS proporciona esta propiedad al momento de visitar los vértices y, por lo tanto, la razón para usar DFS. Realizaremos un DFS desde cualquier vértice en G y registraremos los tiempos de descubrimiento de los vértices a medida que se visitan. Que este número sea $\text{num}(v)$ para un vértice v . También registraremos para cada vértice v el vértice descubierto más antiguo que esté conectado a cualquier vértice en el subárbol de v y bajo (v) sea el vértice con el número más bajo que se puede alcanzar desde v usando 0 o más bordes de árbol de expansión y luego a lo sumo un borde posterior. Ahora podemos ver $\text{bajo}(v)$ es el mínimo de:

$\text{num}(v)$ (Regla 1)

- 1.
2. $\text{bajo num}(T)$ entre todos los bordes posteriores (v, u) (Regla 2)
3. el mínimo más bajo (u) entre todos los bordes del árbol (v, u) (Regla 3)

Cualquier vértice v que no sea la raíz en el árbol DFS es un punto de articulación si y solo si $\text{low}(u) \geq \text{num}(v)$ para cualquier niño u de v , lo que significa que no hay bordes posteriores de ningún vértice en el subárbol de v a ninguno de sus antepasados. La raíz es un punto de articulación si y solo si tiene más de un hijo. Ahora podemos estructurar un algoritmo basado en lo anterior como se muestra en el algoritmo 8.3. El procedimiento assign_num es básicamente un algoritmo DFS que también asigna los valores numéricos a los vértices a medida que se visitan. El segundo procedimiento check_AP encuentra los valores bajos de los vértices al verificar las reglas anteriores y prueba la condición del punto de articulación e incluye los vértices que satisfacen esta condición en V' .

Algorithm 8.3 DFS-based_AP

```

1: Input : connected and undirected graph  $G = (V, E)$ 
2: Output : articulation points  $V' \subseteq V$  of  $G$ 
3: select any vertex  $v \in V$ 
4:  $counter \leftarrow 1$ 
5:  $assign\_num(v)$ 
6:  $check\_AP(v)$ 
7:
8: procedure ASSIGN_NUM(vertex v)
9:    $num(v) \leftarrow counter + 1$ 
10:   $visited(v) \leftarrow true$ 
11:  for all  $u \in N(v)$  do
12:    if  $visited(u) = false$  then
13:       $parent(u) \leftarrow v$ 
14:       $assign\_num(u)$ 
15:    end if
16:  end for
17: end procedure
18:
19: procedure CHECK_AP(vertex v)           ▷ Rule 1
20:    $low(v) \leftarrow num(v)$ 
21:   for all  $u \in N(v)$  do
22:     if  $num(u) \geq num(v)$  then
23:        $check\_ap(u)$ 
24:       if  $low(u) \geq num(v)$  then
25:          $V' \leftarrow V' \cup \{v\}$            ▷ AP found
26:       end if
27:        $low(v) \leftarrow min(low(v), low(u))$  ▷ Rule 3
28:     else if  $parent(v) \neq u$  then
29:        $low(v) \leftarrow min(low(v), num(u))$  ▷ Rule 2
30:     end if
31:   end for
32: end procedure

```

La ejecución de este algoritmo se muestra en la Fig. 8.3. Una gráfica simple con dos puntos de articulación b y d se da en (a). Formamos un árbol DFS que se muestra en (b) para este gráfico y etiquetamos cada vértice v con $num(v)$ y $low(v)$ como se describe. Por ejemplo, el vértice g tiene 7, 1 desde que se descubrió por última vez en el DFS y el borde posterior (g, b) lo conecta al vértice b, que tiene un valor numérico de 1, por lo tanto, su valor bajo se establece en 1. Encontramos que d tiene un vértice descendente e que tiene un valor bajo de 4 que es igual al valor numérico de d, por lo tanto, el vértice d es un punto de articulación. El vértice b es un punto de articulación simplemente porque tiene más de un hijo. Otros posibles árboles DFS enraizados en los vértices c y e se muestran en (c) y (d) de la misma figura. Encontramos que los vértices b y d son nuevamente puntos de articulación en ambos con el mismo razonamiento anterior. El tiempo de ejecución de este algoritmo es simplemente el tiempo que tarda DFS, que es $\mathcal{O}(n + m)$.

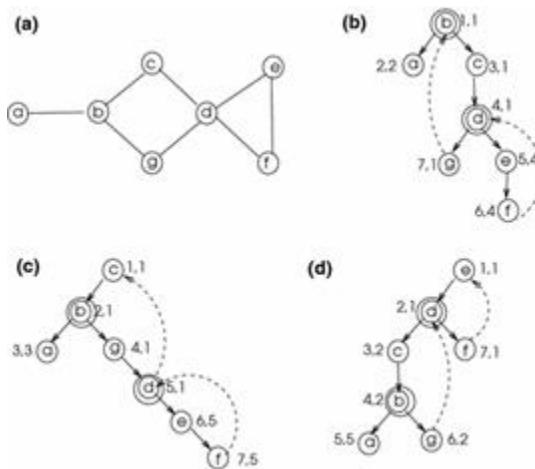


Fig. 8.3 Ejecución del algoritmo AP basado en DSF en un gráfico de muestra para tres árboles DFS diferentes formados. Los puntos de articulación encontrados son los vértices b y d en todos los casos mostrados en círculos dobles

8.3.3 Descomposición de bloques

Un bloque o una biconexas componente de un gráfico G es un máximo biconexas subgrafo de G . Notamos que cada bloque de G está conectado a uno o más bloques por puntos de articulación y un punto de articulación pertenece a más de un bloque. Con esta observación, podemos estructurar un algoritmo similar al algoritmo de punto de articulación basado en DFS como se describe a continuación.

Hopcroft- Algoritmo de Tarjan

Hopcroft y Tarjan proporcionaron un algoritmo para encontrar bloques de una gráfica usando DFS como en el algoritmo de punto de articulación [10]. La idea principal de este algoritmo es la observación clave de que los bloques están separados por los puntos de articulación del gráfico. Tenga en cuenta que un punto de articulación de un gráfico no es un punto de articulación de ningún bloque al que pertenece, ya que un bloque no contiene un punto de articulación propio. Por lo tanto, podemos descubrir puntos de articulación en el gráfico y todos los vértices entre dos puntos de articulación cualquiera serán un bloque. Este algoritmo usa este hecho y funciona de manera similar al punto de articulación basado en DFS, encontrando un algoritmo con la excepción de que empujamos los bordes visitados en una pila hasta que descubrimos tal vértice de corte y colocamos todos los vértices de bordes de la pila en un bloque. estructura de datos cuando lo hacemos. Las variables num y low. como en el punto de articulación se utilizan algoritmos y el algoritmo consta de los siguientes pasos.

Iniciar la DFS de un vértice arbitrario s de la gráfica $G = (V, E)$. Conjunto $counter \leftarrow 1$ y $num(s) \leftarrow 1$, $low(s) \leftarrow 1$.

- 1.
 2. Realice DFS como de costumbre y siempre que se encuentre un vértice vecino v del vértice u considerado, verifique el borde (u , v).
El vértice v se descubre por primera vez y, por lo tanto, (u , v) es un borde de árbol. Incremento de contador y set. $num(v) = counter$, $low(v) = num(v)$. Empuje el borde (u , v) en la pila S .
 - a. segundo. El vértice v ha sido visitado antes y $num(v) < num(u)$. Por lo tanto, el borde (u , v) es un borde posterior. Conjunto $low(u) = \min\{low(u), num(v)\}$. Empuje el borde (u , v) en la pila S .
 - do. El vértice v ha sido visitado antes con $num(v) > num(u)$. Por lo tanto, el borde (u , v) es un borde delantero. Esto es válido solo cuando G es un dígrafo. Dado que el borde (u , v) ya se ha procesado en este caso, no hacemos nada.
 3. Al retroceder desde un vértice v que se buscó usando el borde (u , v), establecer $low(u) = \min\{low(u), low(v)\}$. Si $low(u) \geq num(u)$, el vértice u es un punto de articulación como en el algoritmo 8.3. En este caso, saque todos los bordes de la pila S hasta e incluyendo el borde (u , v). El vértices incidente a estos bordes se formará un bloque de G .
 4. Cuando se realiza un retorno desde el vértice de origen s , haga estallar todos los bordes restantes de la pila S e incluya todos los vértices incidentes en estos bordes en un solo bloque.
- El algoritmo 8.4 muestra un posible pseudocódigo de este algoritmo con más detalle. Asumimos que la gráfica puede consistir en más de un componente.

Algorithm 8.4 DFS_Block

```

1: Input : An undirected graph  $G = (V, E)$ 
2: Output : Block set  $\mathcal{B} = \{\mathcal{B}_1, \dots, \mathcal{B}_k\}$ 
3:  $counter \leftarrow 1; bl\_cnt \leftarrow 1$                                  $\triangleright$  DFS and Block counters
4: for all  $u \in V$  do
5:    $visited(u) \leftarrow false$ 
6: end for
7: select an arbitrary vertex  $r$  to start DFS
8:  $num(r) \leftarrow 1; low(r) \leftarrow 1$ 
9:  $DFS\_Block(r)$ 
10: pop all remaining edges on  $S$  to  $\mathcal{B}_{bl\_cnt}$ 
11: for all  $w \in V$  do                                               $\triangleright$  check other components
12:   if  $\neg visited(w)$  then
13:      $DFS\_Block(w)$ 
14:     pop all remaining edges on  $S$  to  $\mathcal{B}_{bl\_cnt}$ 
15:   end if
16: end for
17:
18: procedure  $DFS\_BLOCK(u)$ 
19:    $visited(u) \leftarrow true$ 
20:    $counter \leftarrow counter + 1$ 
21:    $num(u) \leftarrow counter; low(u) \leftarrow num(u)$ 
22:   for all  $v \in N(u)$  do
23:     if  $visited(v) = false$  then                                          $\triangleright (u, v)$  is a tree edge
24:        $Push(S, (u, v))$ 
25:        $parent(u) \leftarrow v$ 
26:        $DFS\_Block(v)$ 
27:       if  $low(v) \geq num(u)$  then                                          $\triangleright u$  is an articulation point
28:          $Form\_Block((u, v))$                                           $\triangleright$  pop edges of the block
29:       end if
30:        $low(u) = \min[low(u), low(v)]$                                           $\triangleright$  correct low value
31:     else if  $parent(u) \neq v$  then                                          $\triangleright (u, v)$  is a back edge
32:        $Push(S, (u, v))$ 
33:        $low(u) = \min[low(u), num(v)]$                                           $\triangleright$  correct low value
34:     end if
35:   end for
36: end procedure
37:
38: procedure  $FORM\_BLOCK((u, v))$ 
39:   repeat
40:      $(x, y) \leftarrow Pop(S)$ 
41:      $\mathcal{B}_{bl\_cnt} \leftarrow \mathcal{B}_{bl\_cnt} \cup \{x, y\}$ 
42:   until  $(x, y) = (u, v)$ 
43:    $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}_{bl\_cnt}$ 
44:    $bl\_cnt \leftarrow bl\_cnt + 1$ 
45: end procedure

```

La operación de este algoritmo en un pequeño gráfico se muestra en la Fig. 8.4 . DFS se ejecuta desde el vértice de una y todas las aristas del árbol DFS se muestran en negrita que apunta a los padres se empujan en la pila S . El borde (d , b) es un borde posterior ya que $low(b) < low(d)$. El borde (d , b) se empuja sobre S y el valor bajo para el vértice d se corrige a 2. Al regresar del vértice d a c , el valor bajo del vértice c se corrige y se comprueba la condición del punto de articulación, lo cual es falso. A continuación, al regresar del vértice c a b , no se necesita corrección de low (b), sin embargo, el vértice b es un punto de articulación ya que $low(c) \geq num(b)$ y, por lo tanto, llamamos al procedimiento de formación de bloques que saca bordes de la pila S hasta e incluyendo el borde (b , c). Los vértices en el primer bloque. B_1 son b , c y d como se muestra encerrado en el gráfico y en negrita en la pila. En (b), seguimos con el DFS e incluyen bordes (b , e) y (e , f) en el árbol DFS se muestra en negrita y empuje estos bordes en la pila S . Los vértices e y f son los puntos de articulación y el bloque. B_2 con vértices e y f y bloque B_3 con vértices b y e se forman. Finalmente, al regresar del vértice fuente a significa que eliminamos todos los bordes restantes, que es solo el borde (a , b), para formar el último bloque B_4 . La complejidad del tiempo de este algoritmo es $O(n + m)$ Debido a la DFS realizada.

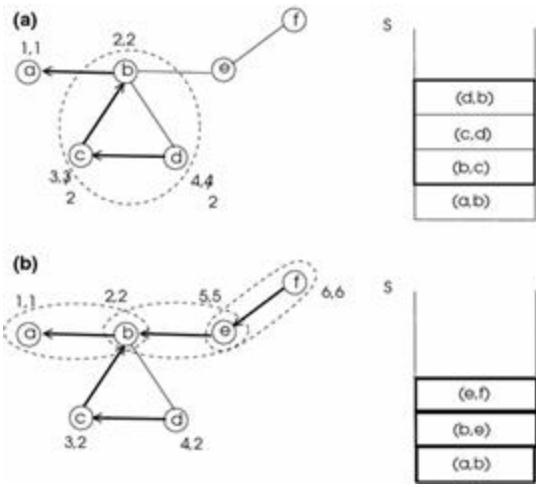


Fig. 8.4 Ejecución del algoritmo Hopcroft– Tarjan en un pequeño gráfico

8.3.4 Encontrando puentes

Un puente es un borde de un gráfico G eliminación de que aumenta el número de componentes conectados de G . Cuando se conecta un gráfico G , dicha eliminación lo deja desconectado. Una red de computadoras tiene enlaces entre sus enrutadores y estos enlaces pueden fallar y causar una interrupción en las transferencias de mensajes de la red. Necesitamos encontrar estos enlaces de déficit y proporcionar rutas alternativas a su alrededor para una red más confiable. Como otro ejemplo, un puente que conecta a dos personas A y B en un gráfico de una red social conectada muestra A y B son amigos y si esta amistad se rompe, la red social tendrá dos componentes y las personas en un componente no tendrán conexión con el otro. Revisemos algunas propiedades útiles de los puentes de una gráfica.

- $\Delta\alpha$ eliminación de un borde que forma parte de un ciclo de un gráfico G no desconecta G y, por lo tanto, un borde (u, v) es un puente si y solo si (u, v) no está contenido en ningún ciclo.
- Considere un puente (u, v) de un gráfico de G . Los vértices u o v son puntos de articulación de G si tienen un grado mayor que 1.

Podemos aplicar la misma estrategia para encontrar puentes de un gráfico no dirigido. $G = (V, E)$ como hicimos en encontrar los vértices cortados; elimine cada borde uno por uno y verifique la conectividad del gráfico utilizando DFS o BFS después de cada eliminación. Si G se desconecta después de quitar un borde e , este borde e es un puente (corte borde) de G . Necesitamos ejecutar el bucle para cada borde por un total de m veces y verificar la conectividad toma $O(n + m)$ tiempo por DFS o BFS dando como resultado un tiempo total de $O(m(n + m))$ para este algoritmo. Nuevamente, este método no es favorable para gráficos grandes y buscamos algoritmos con mejores rendimientos.

El algoritmo de encontrar el puente de Tarjan

Tarjan proporcionó un algoritmo de tiempo lineal para descubrir puentes en un gráfico utilizando el principio de borde posterior como antes [14]. Notamos que cualquier borde de árbol DFS (u, v) que tiene un borde posterior desde cualquier vértice del subárbol arraigado en v hasta u o cualquier antepasado de u forma un ciclo que contiene el borde (u, v) y, por lo tanto, (u, v) no puede ser un puente. El algoritmo propuesto por Tarjan consta de los siguientes pasos.

Realizar un DFS de la gráfica. $G = (V, E)$ desde cualquier vértice de G para obtener el árbol DFS T y etiquetar cada vértice v con num (v) con respecto a su primera visita.

- 1.
2. Para cada vértice $v \in V$ Haz lo siguiente.

Calcule el número de descendientes ND (v) de v . Este es el número de hijos de v más 1 ya que se cuenta un vértice.

a.

segundo. Calcule bajo (v), que es el valor numérico más bajo alcanzado desde v usando los bordes de los árboles y como máximo un borde posterior.

do. Calcule alto (v), que es el valor numérico más alto alcanzado desde v usando los bordes de los árboles y como máximo un borde posterior.

re. Condición del puente : Tarjan demostró que por un borde $(u, v) \in T$ con u siendo padre de v ; Si $low(v) = num(v)$ y $high(v) < num(v) + ND(v)$ entonces (u, v) es un puente [14].

Necesitamos probar esta condición para cada vértice en el árbol DFS. Para hacerlo, ejecutaremos el algoritmo DFS en el gráfico G y registraremos el tiempo de descubrimiento (num (v)) para cada vértice v . Luego, calculamos los valores de ND (v), bajo (v) y alto (v) para cada vértice y verificamos la condición del puente. La ejecución de este algoritmo en un pequeño gráfico se muestra en la Fig. 8.5 . Los bordes (b , c) y (d , e) en negrita se muestran los dos puentes de la gráfica en (a) como se puede ver. Ejecutamos el algoritmo DFS y calculamos los valores num , ND , bajo y alto para cada vértice como se muestra al lado de cada vértice en el árbol DFS en (b). A continuación, se comprueba la condición de puente para cada vértice v incidente en el borde (u , v);

$low(v) = num(v)$ y $high(v) < num(v) + ND(v)$. Solo los vértices c y e satisfacen esta condición y, por lo tanto, (b , c) y (d , e) son los puentes de esta gráfica. El tiempo de ejecución es simplemente el tiempo para el DFS que es $O(n + m)$.

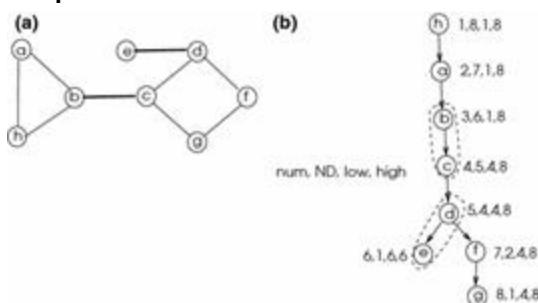


Fig. 8.5 Ejecución del algoritmo de búsqueda de puentes de Tarjan en un gráfico de muestra

8.3.5 Comprobación de conectividad fuerte

Un dígrafo puede ser utilizado para modelar una máquina de estado finito y conectividad fuerte en un dígrafo tal implica la recuperación de un estado de mal funcionamiento ya que siempre hay un camino de cada estado a otro. Podemos querer encontrar personas fuertemente conectadas en una red social que sean amigos cercanos para analizar dicha red. Podemos comprobar si un digraph $G = (V, E)$ está fuertemente conectado o no seleccionando un vértice arbitrario v , ejecutando DFS (o BFS), invirtiendo la dirección de los bordes para obtener el gráfico de transposición G^T y luego ejecutando DFS (o BFS) desde ese vértice en G^T otra vez. Si los vértices visitados en ambas direcciones son iguales a V , G está fuertemente conectado. Este método es suficiente, ya que es posible obtenerlo de cualquier vértice u a w a través de v . El dígrafo puede no estar fuertemente conectado,

en este caso, este algoritmo determina el componente fuertemente conectado que contiene el vértice de inicio v . Este componente llamado V_e tiene los vértices comunes visitados durante DFS o BFS de G y luego G^T como se muestra en el algoritmo 8.5. Como ejecutamos DFS o BFS en ambas direcciones, el tiempo requerido para este algoritmo es $O(n + m)$.

Algorithm 8.5 Strong_Conn

```

1: Input :  $G = (V, E)$ 
2: Output : Show whether  $G$  is strongly connected
3: set of vertices  $V_1 \leftarrow \emptyset$ ,  $V_2 \leftarrow \emptyset$ ,  $V_e \leftarrow \emptyset$ 
4: pick any  $v \in V$ 
5: run  $DFS(G, v)$  (or  $BFS(G, v)$ )
6: record the visited vertices in  $V_1$ 
7: reverse direction of edges to obtain  $G^T$ 
8: run  $DFS(G, v)$  (or  $BFS(G, v)$ )
9: record the visited vertices in  $V_2$ 
10: if  $V_1 \neq \emptyset \wedge V_2 \neq \emptyset$  then
11:   if  $V_1 = V_2 = V$  then
12:     Output "G is strongly connected"
13:   else
14:     Output "G is not strongly connected"
15:   end if
16: end if

```

8.3.6 Detección de componentes fuertemente conectados

La descomposición de un dígrafo en sus SCC es útil en varios algoritmos, ya que permite ejecuciones independientes del algoritmo en cada SCC, por lo tanto, permite el procesamiento paralelo. Hay dos algoritmos fundamentales para detectar SCC en un dígrafo debido a Tarjan [13] y Aho [1]. Ambos algoritmos hacen uso de DFS, el algoritmo de Tarjan funciona con una sola llamada DFS mientras que el algoritmo de Kosaraju requiere dos llamadas DFS, sin embargo, es más sencillo de implementar que el algoritmo de Tarjan .

8.3.6.1 Algoritmo SCC de Tarjan

Este algoritmo ingresa un grafo dirigido. $G = (V, E)$ y detecta SCC s en este gráfico [13]. La idea clave en este algoritmo es la observación de que un SCC de un gráfico es un subárbol de un árbol DFS. En otras palabras, no habrá un borde posterior de un subárbol arraigado en un vértice v de cualquiera de sus descendientes a cualquiera de sus ascendentes si v es la raíz de un SCC.

El uso apropiado de una pila es crucial en la operación de este algoritmo. Los vértices se empujan en una pila S en el orden en que se visitan. La propiedad invariante es que un vértice v se deja en la pila S después de que se visitó si y sólo si existe un camino en G de v que ya están en la pila algún otro vértice S . Cuando la llamada a un vértice v y sus descendientes regresa, verificamos si v tiene una ruta a un vértice que ya está en la pila. Si existe tal camino, el vértice v se deja en la pila para mantener la variante. De lo contrario, el vértice v es la raíz de un SCC y, por lo tanto, se extrae de la pila junto con el SCC que se asigna como la raíz.

A cada vértice v se le asigna $num(v)$ en el orden de la primera visita del DFS y $low(v)$ es el valor num más pequeño que se debe alcanzar desde el vértice v que se incluye a sí mismo. Si $low(v) = num(v)$, el vértice v debe eliminarse de la pila S como la raíz de un SCC. De lo contrario, si $low(v) < num(v)$, el vértice v debe permanecer en la pila. El pseudocódigo para este algoritmo se muestra en el algoritmo 8.6.

Algorithm 8.6 Tarjan_SCC

```

1: Input : directed graph  $G = (V, E)$ 
2: Output : SCC set  $\mathcal{S} = \{S_1, \dots, S_k\}$  of  $G$ 
3:  $i \leftarrow 0; scc\_cnt \leftarrow 1$ 
4: stack  $\mathcal{S} \leftarrow \emptyset$ 
5: for all  $u \in V$  do
6:    $num(u) \leftarrow 0$ 
7: end for
8: for all  $u \in V$  do
9:   if  $num(u) = 0$  then
10:     $SCC(u)$ 
11:   end if
12: end for
13:
14: procedure  $SCC(v)$ 
15:    $low(v) \leftarrow num(v) \leftarrow i; i \leftarrow i + 1$                                 initialize v
16:   push  $v$  on  $\mathcal{S}$ 
17:   for all  $w \in N(v)$  do
18:     if  $num(w) = 0$  then                                               $(v, w)$  is a tree edge
19:        $SCC(w)$ 
20:        $low(v) \leftarrow \min(low(v), low(w))$ 
21:     else if  $num(w) < num(v)$  then                                          $(v, w)$  is a friend or a cross edge
22:       if  $w \in \mathcal{S}$  then
23:          $low(v) \leftarrow \min(low(v), num(w))$ 
24:       end if
25:     end if
26:   end for
27:   if  $low(v) = num(v)$  then                                          $v$  is the root of a component
28:     while  $w$  is on top of  $\mathcal{S} \wedge num(w) \geq num(v)$  do
29:       delete  $w$  from  $\mathcal{S}$ 
30:        $S_{scc\_cnt} \leftarrow S_{scc\_cnt} \cup \{w\}$ 
31:     end while
32:      $\mathcal{S} \leftarrow \mathcal{S} \cup S_{scc\_cnt}$ 
33:      $scc\_cnt \leftarrow scc\_cnt + 1$ 
34:   end if
35: end procedure

```

Este algoritmo es un procedimiento DFS modificado que requiere $O(n + m)$ hora. Las pruebas para determinar si un vértice está en la pila se pueden realizar en un tiempo constante si se mantiene una matriz booleana para las entradas en la pila, según lo propuesto por el autor [13].

8.3.6.2 Algoritmo de Kosaraju

Podemos usar la transposición de un dígrafo para encontrar sus SCC basadas en la observación de que la gráfica G y su transposición G^T tiene exactamente los mismos SCCs. El algoritmo debido a Kosaraju se basa en la contracción de un dígrafo definido a continuación.

Definición 8.15(contracción de un dígrafo) La contracción de un dígrafo G es otro dígrafo G^{SCC} con SCCs de G como super vértices C_1, \dots, C_k . Con los bordes definidos de la siguiente manera. Si hay un borde en G desde un vértice u en SCC^{C_s} a un vértice v en SCC^{C_r} , entonces C_s y C_r están conectados por un borde en G^{SCC} .

Observamos que el dígrafo contratado. G^{SCC} , comúnmente llamado gráfico de componentes de G , no tiene ciclos, en otras palabras, es un gráfico acíclico dirigido. Si hubiera un ciclo entre los SCC, podrían ser contratados para un SCC más grande. El algoritmo de Kosaraju se basa en la idea de que existen los mismos SCC en un gráfico G y su transposición G^T . Mostramos la descripción de alto nivel de este algoritmo en el algoritmo 8.7. Consta de dos fases: primero realizamos un DFS en G para formar un bosque DFS y colocamos los vértices en una pila con respecto a sus tiempos de finalización durante el DFS. En la segunda fase, eliminamos un vértice de la pila y realizamos un DFS en la transposición del gráfico G^T . La segunda llamada a DFS es, de hecho, para visitar los vértices en G^{SCC} . Cuando la búsqueda termina, tenemos todos los vértices de un SCC visitado. Luego continuamos con el siguiente vértice de la pila hasta que todos los vértices se visitan y se colocan en los SCC.

Algorithm 8.7 Kosaraju_SCC

```

1: Input :  $G = (V, E)$ 
2: Output : SCCs of  $G$ 
3: stack  $S \leftarrow \emptyset$ 
4: while  $S \neq V$  do
5:   pick an arbitrary vertex  $v \notin S$ 
6:   run  $DFS(G, v)$  by putting finished vertices on stack  $S$ 
7: end while
8: reverse direction of edges to obtain  $G^T$ 
9: while  $S \neq \emptyset$  do
10:   $u \leftarrow pop(S)$ 
11:  run  $DFS(G^T, u)$             $\triangleright$  form a DFS tree for each vertex on  $S$ 
12: end while
13: vertices in each tree rooted at stack vertices are the SCCs of  $G$ 

```

En la Fig. 8.6 se muestra un gráfico de muestra y la operación de la primera fase de este algoritmo .

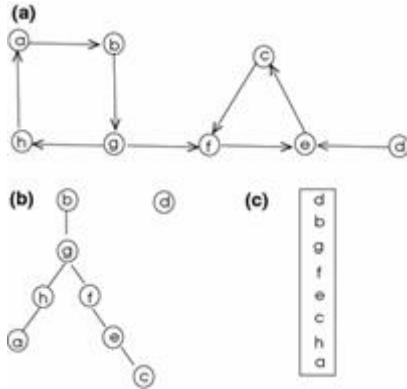


Fig. 8.6 Primera fase del algoritmo de Kosaraju. a el dígrafo, b árbol DFS formado, c contenido final de la pila

La segunda fase del algoritmo saca vértices de la pila y realiza un DFS en estos vértices para obtener SCC como se muestra en la Fig. 8.7 .

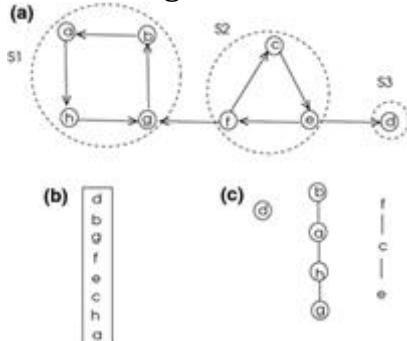


Fig. 8.7 Segunda fase del algoritmo de Kosaraju; una G^T , b la pila, c SCC extraídos por DFS en la pila

Análisis

Dado un digraph $G = (V, E)$ con dos SCC distintos C_1 y C_2 , considera una ventaja $(u, v) \in E$ con $u \in C_1$ y $v \in C_2$. Entonces tenemos la siguiente observación.

Observación 6 Se puede demostrar que $fin(C_1) > fin(C_2)$. Del mismo modo, si (u, v) es un borde en G^T con $u \in C_1$ y $v \in C_2$, entonces $fin(C_2) > fin(C_1)$ [3].

Teorema 8.4 El algoritmo de Kosaraju encuentra correctamente los SCC de un dígrafo $G = (V, E)$ en $O(n + m)$ hora.

La prueba nos vamos a demostrar la exactitud de este algoritmo por inducción. Sea k el número de árboles formados cuando se llama a DFS^{G^T} . Cuando $k = 0$, el caso base se mantiene, y asume la primera $k - 1$ árboles obtenidos de esta manera son los CE de la gráfica G . Sea u la raíz del árbol k th y un miembro del SCC C_1 de g . Para cualquier SCC no

descubierto C_i en el paso k , $\text{fin}(C_i) > \text{fin}(C_j)$ y todos los otros vértices de C_i . Serán descendientes de u en el árbol DFS descubierto. Cualquier borde que se vaya C_i en G^T . Debe dirigirse a los SCC ya descubiertos por el comentario anterior. Por lo tanto, todos los descendientes de u estarán solo en el SCC C_i y ningún otro SCC de G^T [3]. La complejidad del tiempo de este algoritmo es $O(n + m)$ ya que involucra dos llamadas DFS, la primera en G y la segunda en G^T . \square

En la Fig. 8.8 se muestra un enfoque práctico para implementar el algoritmo de Kosaraju. Atravesamos el gráfico utilizando DFS y etiquetamos los vértices con los tiempos de su primera visita en los numeradores que se muestran. Cuando hacemos un retroceso, los tiempos de finalización se registran en los denominadores, como se muestra. Una vez finalizada esta primera fase, ordenamos los denominadores, invirtimos la dirección de los bordes de G para obtener G^T y realice un bosque DFS a partir del vértice del denominador más grande. Cada componente descubierto por el DFS es un SCC.

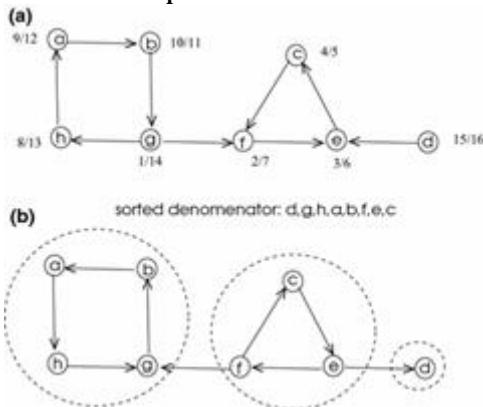


Fig. 8.8 Una implementación práctica del algoritmo de Kosaraju.

8.3.7 Búsqueda de vértices y conectividad perimetral

Encontrar los números de conectividad de vértice y borde de las redes de computadoras nos brinda la información vital sobre qué tan confiables son. Claramente, cuanto mayor es el número de conectividad, más robusta es una red. Veremos formas eficientes de encontrar la conectividad de vértice y borde de un gráfico cuando revisemos los flujos de red en la siguiente sección. Revisamos brevemente un algoritmo de fuerza bruta que encontrará conectividad de vértice y luego un algoritmo de fuerza bruta para conectividad de borde.

Como primer intento, podemos implementar la siguiente estrategia de fuerza bruta. Primero encontramos todos los subconjuntos de los vértices del gráfico de red; ordénelos en orden creciente y luego elimine cada subconjunto del gráfico comenzando desde los subconjuntos más pequeños y verifique la conectividad del gráfico utilizando el algoritmo BFS (o DFS) como se muestra en el pseudocódigo del algoritmo 8.8. El algoritmo BFS_{Conn} verifica la conectividad del gráfico utilizando el algoritmo BFS y devuelve verdadero si el gráfico está conectado y falso de lo contrario.

Algorithm 8.8 Finding k -connectivity value

```

1: Input :  $G(V, E)$ , directed or undirected graph
2: Output : value of  $k$ 
3:  $\mathcal{P} \leftarrow \text{Power Set}(V)$ 
4:  $\mathcal{P}' \leftarrow \text{Sort}(\mathcal{P})$ 
5: for all  $s \in \mathcal{P}'$  in increasing order of  $|s|$  do
6:    $G' \leftarrow G - \{s\}$ 
7:    $\text{result} \leftarrow \text{BFS\_Conn}(G')$ 
8:   if  $\text{result} = \text{false}$  then
9:     return  $|s|$ 
10:    end if
11: end for
```

Este método de fuerza bruta proporcionará el valor k exacto para el gráfico. Aunque funcionará, el principal problema en la práctica con este algoritmo es su complejidad exponencial de tiempo. Para una gráfica con n vértices, el número de subconjuntos de su conjunto de potencias es 2^n , Resultando en 2^n iteraciones del bucle for . El algoritmo BFS dentro de cada iteración de bucle también tiene $O(n + m)$ complejidad de tiempo que resulta en una complejidad de tiempo total de $O(2^n(n + m))$ lo cual es inaceptable incluso para gráficos de tamaño moderado.

El mismo método se puede aplicar para encontrar la conectividad borde de un gráfico G , esta vez mediante la formación del conjunto potencia de bordes de G . El número del conjunto de potencias es 2^m este tiempo y por lo tanto, el tiempo total necesario es $O(2^m(n + m))$ en este caso.

8.3.8 Cierre transitivo

En muchos casos, nos interesaría encontrar si se conectan dos vértices de un gráfico, es decir, hay una ruta entre estos dos vértices. El siguiente gráfico derivado del gráfico original proporciona esta información.

Definición 8.16(cierre transitivo) El cierre transitivo de una gráfica. $G = (V, E)$ es la gráfica $G' = (V, E')$ donde $(u, v) \in E'$ si hay un camino entre u y v en el gráfico G .

La matriz de conectividad de la gráfica definida a continuación proporciona una representación adecuada de una gráfica que es equivalente a su cierre transitivo.

Definición 8.17(matriz de conectividad) La matriz de conectividad de un gráfico. $G = (V, E)$ Es una matriz C con elementos. $c_{i,j}$ tal que $c_{i,j} = 1$ si hay un camino entre los vértices i y j en G y cuando $i = j$ y ∞ de otra manera.

Por lo tanto, encontrar el cierre transitivo de un gráfico se reduce a trabajar su matriz de conectividad. Podemos establecer 0 para vértices que no son vecinos en lugar de ∞ en C para obtener la matriz A y calcular los poderes de A , por ejemplo A^k utilizando la multiplicación de matrices usando la lógica y / o lógica y en lugar de la multiplicación escalar y la suma para obtener conectividad de vértices que son $k+1$ salta lejos Dado que el camino más largo en un gráfico puede ser de longitud $n-1$ a lo sumo, A^{n-1} será igual a C . Alternativamente, ejecutar el algoritmo BFS para cada vértice proporcionará C o configurará los pesos de los bordes 1 y ejecutará el algoritmo Floyd– Warshall usando la matriz de adyacencia del gráfico con bordes que tienen pesos unitarios, lo que también dará como resultado la matriz de conectividad en $O(n^3)$ hora. Warshall del algoritmo puede ser implementado para encontrar la matriz de conectividad C . Tenemos un grafo dirigido. $G = (V, E)$ con una matriz de adyacencia A [n , n], donde $A[i, j] = 1$ Si $(i, j) \in E$, y calcular la matriz C , donde $C[i, j] = 1$ si hay un camino de longitud mayor o igual a 1 desde i hasta j como

se muestra en el algoritmo 8.9. Este algoritmo tiene $\Theta(n^3)$ complejidad de tiempo debido a tres bucles anidados.

Algorithm 8.9 *Warshall's Algorithm*

```

1: Input :  $G(V, E)$ , directed or undirected graph
2: Output : connectivity matrix  $C$ 
3: for  $i = 1$  to  $n$  do                                ▷ initialize C
4:   for  $j = 1$  to  $n$  do
5:      $C[i, j] \leftarrow A[i, j]$ 
6:   end for
7: end for
8: for  $k = 1$  to  $n$  do
9:   for  $i = 1$  to  $n$  do
10:    for  $j = 1$  to  $n$  do
11:      if  $C[i, j] = 0$  then
12:         $C[i, j] \leftarrow C[i, k] \wedge C[k, j]$ 
13:      end if
14:    end for
15:  end for
16: end for

```

8.4 Conectividad basada en flujo

La búsqueda de conectividad que utiliza los algoritmos de flujo de red se basa en encontrar los valores de conectividad de borde o vértice entre cada par de vértices de un gráfico. Una vez que tenemos estos valores, la conectividad de borde o vértice se asigna al valor mínimo de los valores calculados. Primero revisaremos el método de flujo de red con dos algoritmos básicos para calcular el flujo en una red y luego describiremos los algoritmos para encontrar la conectividad de vértice y borde utilizando este método.

8.4.1 Fluxos de red

Asumamos un grafo dirigido $G = (V, E)$ en el sentido habitual. Formaremos una red de flujo de la siguiente manera. Cada borde $e \in E$ de G se le asigna un entero no negativo denominado capacidad $c(e)$ y tenemos un vértice de origen s y un vértice de sumidero t . Un flujo $f(u, v)$ a través del borde (u, v) en esta red satisface lo siguiente:

- *Restricción de capacidad* : $\forall (u, v) \in E$,

$$0 \leq f(u, v) \leq c(u, v)$$

lo que significa que un flujo a través de un borde no puede exceder la capacidad asignada a ese borde.

- *Conservación del flujo* : $\forall u \in [V - \{s, t\}]$,

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

En otras palabras, el flujo hacia el vértice u es igual al flujo que sale de u para cualquier vértice v , excepto el vértice de origen s y el vértice de sumidero t .

El valor $|f|$ de un flujo f se define como sigue.

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

(8.3)

Es decir, el valor de flujo de una red de flujo es la diferencia de la suma de flujos del vértice sumidero s a la suma de flujos en s . El problema del flujo máximo es encontrar un flujo con un valor máximo en una red de flujo. En la Fig. 8.9 se muestra un ejemplo de red de flujo.

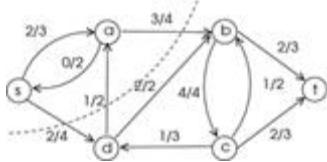


Fig. 8.9 Una red de flujo. Una etiqueta en un borde es el flujo / capacidad del borde

8.4.1.1 Cortes

Definición 8.18(corte) Un corte (S, T) de una red de flujo divide la red en dos conjuntos S y T de nodos separados de manera que $s \in S$ y $t \in T$. La capacidad de un corte, $c(S, T)$ se define como $\sum_{e \in [S, T]} c(e)$.

Un borde (u, v) con $u \in S$ y $v \in T$ se llama borde delantero del corte (S, T). Cuando $v \in S$ y $u \in T$, el borde (u, v) se dice que es un borde hacia atrás. El flujo a través de un corte (S, T) es la diferencia entre la suma de los flujos en los bordes hacia adelante y la suma de los flujos en los bordes hacia atrás. El corte mostrado por una curva discontinua en la figura 8.9 tiene un flujo de $2 + 3 - 1 = 4$ valor. Dado, una red de flujo G con cualquier corte (S, T) de G , se pueden mostrar las siguientes observaciones.

Observación 7 El valor del flujo f en G es igual al valor a través del corte (S, T).

Observación 8 El valor del flujo f a través del corte (S, T) no excede la capacidad del corte (S, T).

8.4.1.2 Redes Residuales

Definición 8.19(red residual) Dado un flujo f en un gráfico $G = (V, E)$, la red residual $G_f = (V, E_f)$ tiene el mismo conjunto de vértices establecidos como G y bordes con capacidades residuales positivas definidas de la siguiente manera.

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \text{ is a forward edge} \\ f(v, u) & \text{if } (u, v) \text{ is a backward edge} \end{cases}$$

En otras palabras, la capacidad residual de un borde (u, v) es la cantidad de flujo que se puede empujar a través de (u, v) y la capacidad residual del borde (v, u) es el flujo que se utiliza. Cuando estamos actualizando una red residual después de un cambio de flujo, siempre debemos modificar estos valores para que se obedezca la conservación del flujo en un nodo de la red. Por ejemplo, si aumentamos el valor del flujo a través de un borde (u, v) en 3 unidades, entonces deberíamos aumentar el valor del flujo a través del borde (v, u) en 3 unidades para mantener la propiedad de la red de flujo. Además, el flujo

utilizado por el borde (v, u) puede devolverse si causará un mayor flujo de red al hacerlo. En resumen, G_f Tiene bordes que pueden ser utilizados para tener más flujos a través de ellos. La red residual de la red de la figura 8.9 se muestra en la figura 8.10.

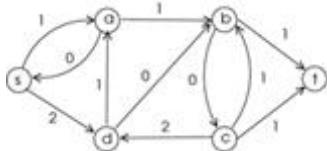


Fig. 8.10 La red residual de la red de la Fig. 8.9

Definición 8.20(ruta de aumento) Una ruta p desde la fuente s hasta el destino t en una red residual G_f Se llama un camino de aumento con respecto al flujo f . Podemos aumentar el valor del flujo a través de una ruta de aumento P por su capacidad residual definida a continuación:

$$c_f(P) = \min\{c_f(u, v)\}$$

donde la capacidad residual $c_f(P)$ de una trayectoria P es la capacidad residual mínima de sus bordes. Podemos empujar un flujo adicional máximo a través de P por el valor de $c_f(P)$ de lo contrario, estaremos violando la restricción de capacidad en la red residual.

Teorema 8.5 Un flujo es máximo si y solo si no hay rutas de aumento desde la fuente s hasta el sumidero t . El valor de este flujo. $|f| = c(S, T)$ por alguna corte (S, T) de G .

Prueba Si hay un camino de aumento desde s hasta t , entonces podemos aumentar el valor del flujo máximo f a través de este camino. Por lo tanto, f sería máximo.

Como consecuencia, puede afirmarse el siguiente teorema.

Teorema 8.6 El valor de un flujo máximo, $|f|$, En una red de flujo G es igual a la capacidad de un mínimo de corte de la red G .

8.4.1.3 Algoritmo de Ford – Fulkerson

Con estos antecedentes, podemos ver que mientras exista una ruta de aumento en una red residual, podemos aumentar el flujo a través de esa ruta. Algoritmo de Ford-Fulkerson utiliza esta idea y encuentra que el valor de flujo máximo en una red de flujo que tiene un vértice fuente s y un lavabo vértice t y capacidades de borde positivos incrementando gradualmente el flujo en el gráfico residual. La idea principal en este algoritmo es aumentar el flujo a través de los bordes de la red investigando una ruta de aumento. Cada vez que se encuentra una ruta de este tipo, el flujo aumenta por la capacidad residual de la ruta p como se muestra en el algoritmo 8.10.

Algorithm 8.10 Ford-Fulkerson Algorithm

```

1: Input :  $G(V, E)$ , directed or undirected graph
2: Output : The value of maximum flow  $f$ 
3:
4:  $f \leftarrow 0$ 
5:  $G_f \leftarrow G$ 
6: while  $\exists$  an  $(s - t)$  path  $P$  of  $G_f$  do
7:    $c_f(P) \leftarrow \infty$ 
8:   for all  $e \in P$  do                                 $\triangleright$  compute the residual capacity  $c_f(P)$  of path  $P$ 
9:     if  $c_f(e) < c_f(P)$  then
10:       $c_f(P) \leftarrow c_f(e)$ 
11:    end if
12:   end for
13:   for all  $e \in P$  do                           $\triangleright$  update flows through edges of path  $P$ 
14:     if  $c_f(e) > 0$  then
15:        $f(e) \leftarrow f(e) + c_f(P)$ 
16:     else
17:        $f(e) \leftarrow f(e) - c_f(P)$ 
18:     end if
19:   end for
20: end while

```

Tenga en cuenta que cuando un flujo f_s se empuja a través de un borde (u, v) por primera vez, necesitamos formar un nuevo borde (v, u) con etiqueta f_v . Si tal borde no existe. La Figura 8.11 muestra el funcionamiento de este algoritmo en una red pequeña. Comenzamos con un flujo de 0 y la suma de todos los flujos posibles a cualquier nodo u es igual a la suma de todos los flujos posibles de u . Tenemos un corte arbitrario en esta red con un valor de 7 como se muestra y este valor es el flujo máximo que se alcanza en esta red como lo muestra el teorema de corte mínimo de flujo máximo. Luego buscamos rutas de aumento y cada vez que se encuentra dicha ruta p , los flujos a través de todos los bordes de esta ruta disminuyen por el valor del flujo mínimo a lo largo de la ruta p y el flujo aumenta con este valor. Encontramos el borde con el valor mínimo en la ruta de aumento. $s - a - b - c - t$ is (s, a) con el valor de 3. Por lo tanto, el flujo f se establece en 3, el gráfico residual se actualiza para obtener el gráfico en (c) y, de esta manera, tenemos el gráfico residual final en (f) después de cuatro iteraciones que no tienen ningún camino de aumento y paramos con un valor f final de 7 como en el corte.

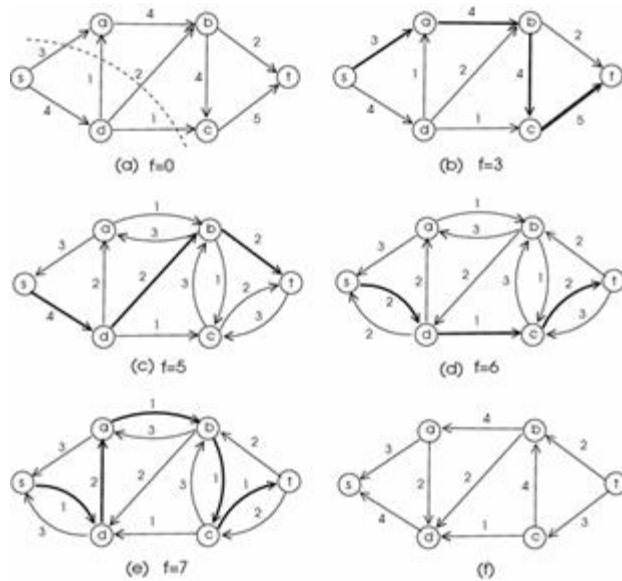


Fig. 8.11 Operación del algoritmo Ford-Fulkerson en una red pequeña

Análisis

Dado que los valores de flujo son enteros, estaremos incrementando el valor de flujo $\lfloor f^* \rfloor$ veces como máximo donde f^* es el valor de flujo máximo, ya que el valor de flujo se incrementa en uno en cada paso en el peor de los casos. Cada ruta de aumento puede ser encontrada por el algoritmo DFS o BFS en $O(n + m)$ tiempo que resulta en un tiempo total

de $O(|f^m|(n + m))$ hora. Cuando la elección de las rutas de aumento se realiza de forma arbitraria y $|f^m|$ es grande, la complejidad del tiempo de este algoritmo puede ser alta.

8.4.1.4 Algoritmo de Edmonds-Karp

El algoritmo Edmonds-Karp funciona de manera similar al algoritmo Ford-Fulkerson, con la excepción de seleccionar la ruta de aumento con la longitud mínima entre todas las rutas de aumento posibles. La selección de las rutas de aumento más cortas se realiza mediante el algoritmo BFS.

Algorithm 8.11 Edmond-Karp Algorithm

```

1: Input :  $G(V, E)$ , directed or undirected graph
2: Output : The value of maximum flow  $f$ 
3:
4:  $f \leftarrow 0$ 
5:  $G_f \leftarrow G$ 
6: while  $G_f$  contains an  $s - t$  path  $P$  do
7:    $P \leftarrow$  an  $s - t$  path in  $G_f$  with minimum length
8:   augment  $f$  using  $P$ 
9:   update  $G_f$ 
10: end while

```

El camino de longitud mínima p se puede determinar en $O(n + m)$ tiempo utilizando BFS. Habiendo encontrado la ruta más corta p , podemos aumentar f en $O(n)$ y actualizar de G_f toma también $O(n)$ tiempo resultando en $\approx O(m)$ tiempo para una iteración del tiempo de bucle. Hay $O(n)$ iteraciones que resultan en un tiempo total de $O(m^2n)$ [2].

8.4.2 Búsqueda de conectividad de borde

La conectividad del borde $\lambda(u, v)$ de los dos vértices u y v de un gráfico simple G es el menor número de bordes que se eliminan, lo que hace que u y v se desconecten. En una gráfica no dirigida, $\lambda(u, v) = \lambda(v, u)$ y en el caso de un gráfico dirigido, esta igualdad puede no ser válida. Podemos ver que cuando un gráfico G no dirigido no es trivial, la conectividad de borde de G , $\lambda(G)$, es el valor mínimo de $\lambda(u, v)$ para cada par de vértices desordenados u y v . Para un digraph G' , $\lambda(G')$ es el valor mínimo de $\lambda(u, v)$ para cada par de vértices ordenados u y v .

Con este fondo, podemos calcular el valor de $\lambda(G)$ para un gráfico no dirigido o dirigido si tenemos un método para encontrar los valores de conectividad para cada par de vértices. De hecho, este método se basa en el algoritmo de flujo máximo que hemos revisado anteriormente. Incluso proporcionó un algoritmo basado en el flujo máximo para calcular $\lambda(u, v)$ para cada par de vértices y la conectividad del borde del gráfico es simplemente el mínimo de todos los valores calculados [6]. El algoritmo para encontrar $\lambda(u, v)$ consta de los siguientes pasos [4, 6].

Entrada : Un grafo no dirigido o dirigido $G = (V, E)$ y un par de vértices u y v .

- 1.
2. Salida : El valor de $\lambda(u, v)$.
3. Formar la red $G' = (V', E')$ implementando lo siguiente.

Reemplace cada borde $(x, y) \in E$ con arcos (x , y) y (y , x).

a.

segundo. Designa u como la fuente y v como el vértice del sumidero.

do. Asigna una capacidad de 1 a cada arco.

4. Encuentra la función de flujo máximo f en \tilde{G} .

5. $\lambda(u, v) \leftarrow f$

Tenemos $\frac{n(n-1)}{2}$ pares desordenados en un gráfico no dirigido y $\frac{n(n-1)}{2}$ Ordenó pares en un digraph para comprobar. Por lo tanto, tenemos que llamar al procedimiento anterior tantas veces. En [6] se demostró que la complejidad temporal de este algoritmo es O (nm).

Consideremos la gráfica G de la figura 8.12 . El corte de borde C que se muestra en la línea discontinua separa los vértices en dos subconjuntos de G_1 y G_2 . Si C es el mínimo de corte de borde y seleccionamos un solo vértice $a \in G_1$ y comprobar la conectividad $\forall v \in G_2$, Puede observarse que $\kappa(G)$ Es el mínimo de estos valores.

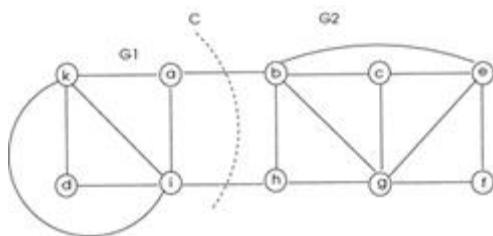


Fig. 8.12 Un gráfico de muestra.

Por lo tanto, podemos tener un algoritmo que consta de los siguientes pasos [5]:

Entrada : Un grafo no dirigido o dirigido $G = (V, E)$.

1.

2. Salida : El valor de $\lambda(G)$.

3. Encuentre el corte de borde mínimo C de G que separa G en G_1 y G_2 .

4. Seleccione un vértice arbitrario $u \in G_1$.

5. Calcular $\lambda(u, v)$, $\forall v \in G_2$.

6. $\lambda(G) \leftarrow \min\{\lambda(u, v) | v \in G_2\}$

Este algoritmo requiere $\frac{n(n-1)}{2}$ Cálculos de conectividad de borde en lugar de $\frac{n(n-1)}{2}$ de la primera algoritmo, sin embargo, un problema con este algoritmo es para determinar el separador de borde mínimo C . Incluso y Tarjan notaron que el cálculo del mínimo corte de borde no es necesario, y por lo tanto, podemos calcular $\lambda(u, v)$ para todos los vértices $v \neq u$ del gráfico G e implemente el algoritmo anterior sin calcular el límite mínimo de corte [7]. Un conjunto Y que contiene vértices tanto de G_1 y G_2 tal que $Y \cap G_1 \neq \emptyset$ y $Y \cap G_2 \neq \emptyset$ se llama un λ -cubrimiento del grafo g . La selección de vértices de un conjunto tan pequeño en la línea 3 del algoritmo anterior da como resultado un algoritmo con mejor rendimiento.

8.4.2.1 Un algoritmo basado en árbol de expansión

Esfahanian y Hakimi propusieron un algoritmo para formar un árbol de expansión con un gran número de hojas y algunos vértices intermedios [5]. Considere un árbol de expansión $T = (V, E(T))$ de un grafo G con $L \in V$ Como vértices no foliares. Entonces L es un λ -cubrimiento de G que significa ambas particiones G_1 y G_2 contener al menos un vértice que es un vértice intermedio de T . Al utilizar este razonamiento, la selección de un árbol de expansión de un gráfico G con la menor cantidad de vértices intermedios que sea posible da como resultado un algoritmo que debe seleccionar solo unos pocos de esos vértices para calcular los valores de conectividad de borde. El pseudocódigo del algoritmo de formación de árbol de expansión se muestra en el algoritmo 8.12, donde $I(u)$ se refiere al conjunto de todos los bordes incidentes en un vértice u , y $A(u)$ se refiere al conjunto de todos los vértices adyacentes a u .

Algorithm 8.12 Spanning Tree Algorithm

```

1: Input :  $G(V, E)$ 
2: Output : A spanning tree  $T$  of  $G$ 
3:  $V(T) \leftarrow \emptyset; E(T) \leftarrow \emptyset$ 
4: select  $u \in V(G)$ 
5:  $V(T) \leftarrow \{u\} \cup A(u)$ 
6:  $E(T) \leftarrow E(T) \cup I(u)$ 
7: while  $|E(T)| < |V(T)| - 1$  do
8:   select a leaf vertex  $w \in T$  such that  $|A(w)(V(G) - V(T))| \geq |A(x)(V(G) - V(T)), \forall x \in T$ 
      that is a leaf vertex
9:   for all  $v \in A(w) \cup ((V(G) - V(T))$  do
10:     $V(T) \leftarrow V(T) \cup \{v\}$ 
11:     $E(T) \leftarrow E(T) \cup \{(w, v)\}$ 
12:   end for
13: end while

```

Una vez que se construye un árbol de expansión utilizando el algoritmo 8.12, la conectividad del borde del gráfico se puede calcular mediante el siguiente algoritmo.

Entrada : Un grafo no dirigido o dirigido $G = (V, E)$.

- 1.
2. Salida : El valor de $\lambda(G)$.
3. Construya un árbol de expansión T de G usando el algoritmo 8.12.
4. $P \leftarrow$ El mínimo del orden de las hojas o los vértices interiores de T .
5. Seleccione un vértice arbitrario $u \in P$.
6. Calcular $\lambda(u, v), \forall v \in P \setminus \{u\}$.
7. $c \leftarrow \min[\lambda(u, v)]$
8. $\lambda(G) \leftarrow \min\{c, \delta(G)\}$

8.4.2.2 Un algoritmo dominante basado en conjuntos

Un conjunto dominante de una gráfica. $G = (V, E)$ es el conjunto $V' \subset V$ de sus vértices tal que cualquier vértice $v \in V'$ es un elemento de V' o un vecino de un vértice en V' . La conectividad de borde de un gráfico G se puede calcular utilizando un conjunto dominante de G como lo muestra Matula [11]. Primero se forma un conjunto D dominante de una gráfica y un vértice $u \in D$ esta seleccionado Los valores de conectividad de borde para u y v , $\forall v \in D$ luego se calculan utilizando el algoritmo de flujo máximo. La conectividad de borde de la

gráfica G , $\delta(G)$, es entonces el mínimo del mínimo valor y el grado más pequeño de la gráfica como en el algoritmo anterior. Claramente, un conjunto dominante más pequeño de G da como resultado un mejor rendimiento. Encontrar el conjunto dominador de orden mínimo de una gráfica es NP-difícil, sin embargo, encontrar un conjunto dominador mínimo que no esté contenido en ningún otro conjunto dominante de G se puede encontrar utilizando un enfoque codicioso. El algoritmo 8.13 muestra cómo calcular un pequeño conjunto dominante. Matula demostró que utilizando este algoritmo de conjunto dominante. $\delta(G)$ Se puede calcular en tiempo $O(nm)$ [11].

Algorithm 8.13 Minimal Dominating Set Algorithm

```

1: Input :  $G(V, E)$ 
2: Output : A minimal dominating set  $D$  of  $G$ 
3:
4: select  $u \in V$ 
5:  $D \leftarrow \{u\}$ 
6:  $G' \leftarrow G - u$ 
7: while  $G' \neq \emptyset$  do
8:   select a vertex  $v$  in  $G'$ 
9:    $D \leftarrow D \cup \{v\} \cup N(v)$ 
10:   $G' \leftarrow G - D$ 
11: end while

```

8.4.3 Búsqueda de conectividad de vértices

La conectividad del vértice. $\kappa(u, v)$ de dos vértices u y v de un gráfico simple $G = (V, E)$ es el menor número de vértices supresión de lo que hace que u y v desconectado. Cuando $(u, v) \in E$, $\kappa(u, v) = n - 1$. El método empleado para encontrar la conectividad de vértice es similar al cálculo de conectividad de borde. Podemos encontrar los valores de conectividad de borde de todos los pares de vértices en G utilizando el método de flujo máximo y asignar el mínimo de estos valores como la conectividad de borde de G como se muestra en [6]. Este algoritmo al que llamaremos el algoritmo de Even consiste en los siguientes pasos.

Entrada : Un grafo simple no dirigido o dirigido $G = (V, E)$ y un par de vértices u y v .

- 1.
2. Salida : El valor de $\kappa(G)$.
3. Formar la red $G' = (V', E')$ implementando lo siguiente.

Reemplaza cada vértice a con dos vértices $a_1, a_2 \in E'$ que están conectadas por un borde $e_{a_1 a_2}$. Cada borde $(a, b) \in E$ Luego es reemplazado por dos bordes $e_1 = (a_2, b_1)$ y $e_2 = (b_2, a_1)$

a.
segundo. Designa u como la fuente y v como el vértice del sumidero.

do. Asigna una capacidad de 1 a cada arco en G'

4. Encuentra la función de flujo máximo f en G' .

5. $\kappa(u, v) \leftarrow f$.

los G' El gráfico obtenido de esta manera tendrá $2n$ vértices y $2n+m$ bordes La figura 8.13 muestra el gráfico dirigido. G' obtenido de un gráfico G no dirigido utilizando este procedimiento. Las carreras en G' están etiquetadas con valores unitarios y el flujo

máximo en esta red desde el vértice v_2 a v_1 se calcula. Se mostró en [6] que la complejidad del tiempo del algoritmo anterior es $O(nr^{2/3})$

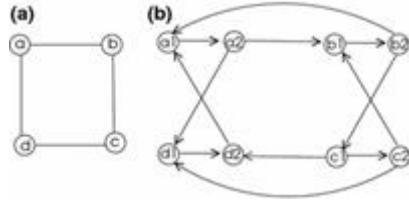


Fig. 8.13 Obtención del gráfico dirigido expandido. G' en b de la gráfica de ejemplo G en una

Incluso y Tarjan Mostró que no necesitamos encontrar $\kappa(u, v)$ para cada par de vértices u y v , solo necesitamos valores de cálculo para el conjunto $V' \subset V$ de vértices con $|V'| = \kappa + 1$ y actualizar el valor mínimo de $\kappa(u, v)$ como lo hacemos en el algoritmo 8.14 [7].

Algorithm 8.14 Even-Tarjan Algorithm

```

1: Input :  $G(V, E)$ , directed or undirected graph
2: Output : The value of  $\kappa(G)$ 
3:
4:  $min\_conn \leftarrow n - 1$ 
5:  $i \leftarrow 1$ 
6: while  $i \leq min\_conn$  do
7:   for  $j = i + 1$  to  $n$  do
8:     if  $i > min\_conn$  then
9:       break
10:      else if  $v_i$  and  $v_j$  are not adjacent then
11:        compute  $\kappa(v_i, v_j)$  using Even's algorithm
12:         $min\_conn \leftarrow \min[min\_conn, \kappa(v_i, v_j)]$ 
13:      end if
14:    end for
15:  end while
16:  $\kappa(G) \leftarrow min\_conn$ 

```

8.5 Búsqueda de conectividad paralela

Es posible que tengamos que encontrar si un gráfico que representa una red está conectado y sus parámetros de conectividad. Presentamos algoritmos para este propósito en esta sección.

8.5.1 Cálculo de la matriz de conectividad

los $n \times n$ La matriz de conectividad C de una gráfica G de orden n se define como sigue.

$$C[i, j] = \begin{cases} 1 & \text{if there is a path of length 0 or more from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Podemos obtener la matriz de conectividad C de la matriz de adyacencia A de un gráfico G multiplicando A $n-1$ veces por sí mismo, en otras palabras, tomando el $n-1$ º poder de A . Sin embargo, debemos realizar la suma y la multiplicación requeridas en la multiplicación de matrices habitual como la suma booleana (lógica o de operación) y la multiplicación booleana (lógica y de operación). Antes de realizar la multiplicación booleana, necesitamos generar la matriz B , que difiere de la matriz de adyacencia A con todos los elementos diagonales como 1 en lugar de 0. Esta matriz Bahora tiene 1 para todas las rutas en G que tienen 0 o 1 longitud. Multiplicar B por sí mismo proporciona B^k que muestra caminos de longitud 2 o menos y en general, B^k contiene 1 para trayectos de longitud k o menos entre dos vértices cualquiera.

La longitud máxima del camino en una gráfica G con n vértices puede ser $n-1$ y por lo tanto, tenemos que encontrar B^{n-1} . El número requerido de multiplicaciones booleanas es

entonces $\lceil \log(n-1) \rceil$. Por ejemplo, para encontrar B^8 , Necesitamos encontrar $B \times B$ ceder B^2 ; entonces $B^2 \times B^2$ dar B^4 y finalmente $B^4 \times B^4$ para obtener B^8 para un total de $\lceil \log 7 \rceil = 3$ multiplicaciones $C = B^8$ donde $m = 2^{\lceil \log(n-1) \rceil}$ cuando $n-1$ no es un poder de 2 desde $B^m = B^{m-1}$ cuando $m > n-1$ [2]. En la Fig. 8.14 se muestra un gráfico de muestra y su matriz de adyacencia .

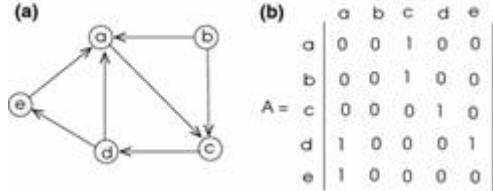


Fig. 8.14 Un ejemplo de gráfico no dirigido y su matriz de adyacencia A

B^2 y B^4 para este ejemplo de gráfica son las siguientes.

$$B = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}, B^2 = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}, C = B^4 = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Ahora, al verificar la entrada $C[i, j]$ se muestra si hay una ruta desde el vértice i hasta j . Podemos ver que el vértice b puede alcanzar todos los otros vértices, mientras que ningún otro vértice puede llegar a él como se muestra en la gráfica. Para paralelizar este algoritmo, podemos utilizar cualquiera de los procedimientos de multiplicación de matrices paralelas, como el descrito en la Secta. 4.7.1 utilizando la multiplicación booleana y la suma en lugar de la multiplicación y la suma de números reales.

8.5.2 Encontrar componentes conectados en paralelo

Hemos visto cómo encontrar componentes conectados de un gráfico no dirigido en la Sect. 8.3.1 . Ahora describiremos dos algoritmos para encontrar los componentes de un gráfico no dirigido en paralelo.

8.5.2.1 Usando la Matriz de Conectividad

Sabemos cómo trabajar la matriz de conectividad C en paralelo de la sección anterior. Definamos ahora una matriz D que tiene las siguientes entradas [2].

$$D[i, j] = \begin{cases} v_j & \text{if } C[i, j] = 1 \\ 0 & \text{otherwise} \end{cases}$$

La fila i de D tiene los nombres de los vértices, el vértice i está conectado, que de hecho están en el mismo componente que i , ya que hay una ruta entre cada uno de ellos y el vértice i . Ahora podemos asignar un vértice v_k al componente k si k es el índice más pequeño para el cual $D[i, k] \neq 0$. La formación paralela de este algoritmo tiene tres pasos de la siguiente manera.

Forma la matriz C en paralelo

- 1.
2. Construye la matriz D a partir de C en paralelo
3. Asignar un componente para cada vértice en D .

8.5.2.2 Usando la Matriz de Adyacencia

Hemos revisado cómo encontrar los componentes conectados de un gráfico no dirigido de forma secuencial utilizando el algoritmo DFS. Cada vez que llamábamos al procedimiento DFS desde el programa principal, se procesaba un nuevo componente, ya que cada llamada procesaba cada vértice en un componente. Ahora intentaremos construir un algoritmo paralelo basado en el algoritmo secuencial. Sea A la matriz de adyacencia del gráfico no dirigido que estamos investigando. Los procesos paralelos de partición A a k serán por filas , de modo que cada proceso p_i consigue $\lfloor n/k \rfloor$ filas Cada proceso luego encuentra bosques de expansión DFS para los subgrafos que incluyen los bordes de los que son responsables. El último paso consiste en la fusión de los bosques que se extienden por pares hasta que quede un bosque que se extiende. La fusión de los bosques se puede hacer de manera eficiente utilizando las operaciones de búsqueda y unión [9]. La Figura 8.15 muestra un gráfico no dirigido con dos componentes. La matriz de adyacencia tiene 8 filas que pueden dividirse en dos procesos p_0 (filas 1-4) y p_1 (filas 5-8)

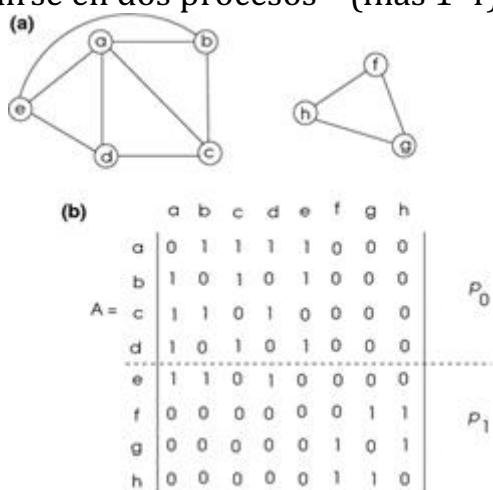


Fig. 8.15 Un ejemplo de gráfico no dirigido con dos componentes

Proceso p_0 tiene los subgraphs se muestra en la Fig. 8.16 a y p_1 tiene los subgrafos mostrados en la figura 8.16 c. Cada proceso luego construye los árboles de expansión DFS para sus subgrafos que se muestran en (c) y (d) de la misma figura.

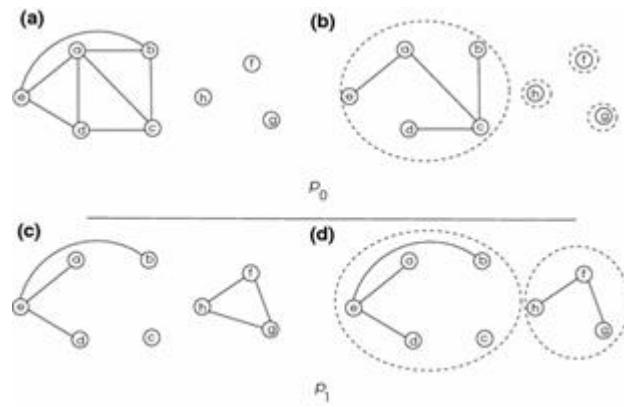


Fig. 8.16 Partición de la gráfica de muestra de la Fig. 8.15 en dos procesos

Estos dos bosques abarcadores se combinan utilizando los procedimientos de búsqueda y unión para encontrar los componentes. $C_1 = \{a, b, c, d, e\}$ y $C_2 = \{f, g, h\}$ como se muestra en la figura 8.17 .

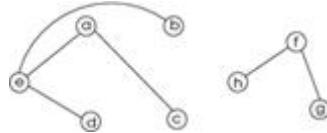


Fig. 8.17 Componentes de la gráfica de muestra de la Fig. 8.15

Análisis

Cuando usamos el mapeo de bloques 1-D con k procesos, calculamos el bosque que abarca cada proceso P_i necesariamente $\Theta(n^2/k)$ tiempo desde $n/k \times k$. El bloque de la matriz de adyacencia se asigna a cada proceso. La fusión de los bosques abarcadores toma $\Theta(n \log k)$ tiempo desde que hay $\log k$ fusionando cada toma $\Theta(n)$ hora. El costo de comunicación del envío de bosques extensos también es $\Theta(n \log k)$. El tiempo de ejecución paralelo de este algoritmo es entonces,

$$T_P = \Theta\left(\frac{n^2}{k}\right) + \Theta(n \log k)$$

(8.4)

La aceleración S y la eficiencia E considerando la complejidad secuencial de $\Theta(n^2)$ para este algoritmo son entonces los siguientes.

$$S = \frac{\Theta(n^2)}{\Theta(n^2/k) + \Theta(n \log k)}$$

(8.5)

$$E = \frac{1}{1 + \Theta((k \log k)/n)}$$

(8.6)

8.5.3 Una encuesta de algoritmo de SCC paralelo

Los algoritmos clásicos de Tarjan y más tarde Kosaraju para encontrar los SCC de un dígrafo son difíciles de paralelizar debido a la operación inherentemente secuencial del algoritmo DFS empleado en ambos. Un algoritmo paralelo denominado algoritmo de división y conquista de componentes fuertes (DCSC) por Fleischer et al. [8] utiliza un enfoque diferente dividiendo el dígrafo en tres subgrafos desunidos y procesando estos subgrafos recursivamente en paralelo. Describiremos brevemente este algoritmo paralelo, ya que puede usarse en la práctica con algunas modificaciones.

Dado un digraph $G = (V, E)$, los descendientes $\text{Desc}(G, v)$ de un vértice v son los vértices en G que son alcanzables desde v incluyéndose a sí mismo. Los predecesores $\text{Pred}(G, v)$ de un vértice v se pueden definir de manera similar al conjunto de vértices desde los cuales se puede alcanzar el vértice v . Los vértices restantes en el gráfico G se llaman el resto mostrado por $\text{Rem}(G, v) = V \setminus (\text{Desc}(G, v) \cup \text{Pred}(G, v))$. Se muestra que

$$\text{SCC}(G, v) = \text{Desc}(G, v) \cup \text{Pred}(G, v)$$

y cualquier SCC de G es un subconjunto de $\text{Desc}(G, v)$, $\text{Pred}(G, v)$ o $\text{Rem}(G, v)$. El algoritmo diseñado hace uso de esta propiedad al seleccionar primero un vértice aleatorio v , encontrar sus conjuntos predecesor y descendiente y luego encontrar el SCC que contiene este vértice. Luego recurre en los vértices restantes en paralelo como se muestra en el algoritmo 8.15. Se mostró en [8] este algoritmo tiene una complejidad de tiempo esperada de $O(n \log n)$ en el caso de serie. Más adelante, McLendon et al. extendió este algoritmo mediante una simple modificación para mejorar el rendimiento [12].

Algorithm 8.15 *Finding SCCs in parallel*

```

1: procedure DCSC(G)
2:   if  $G = \emptyset$  then
3:     return
4:   end if
5:   select a vertex  $v \in V$ 
6:    $\text{SCC}(G, v) = \text{Desc}(G, v) \cup \text{Pred}(G, v)$ 
7:   output  $\text{SCC}(G, v)$ 
8:   do in parallel
9:      $\text{DCSC}(\text{Pred}(G, v) \setminus \text{SCC}(G, v))$ 
10:     $\text{DCSC}(\text{Desc}(G, v) \setminus \text{SCC}(G, v))$ 
11:     $\text{DCSC}(\text{Rem}(G, v))$ 
12:   end do
13: end procedure

```

8.6 Algoritmos de conectividad distribuida

En un entorno de red, nuestro objetivo es que cada nodo de la red descubra los valores de conectividad del gráfico que representa la red.

8.6.1 Un algoritmo de conectividad k distribuida

Sabemos que el grado más bajo $\delta(G)$ de un gráfico G es un límite superior en el valor de la conectividad $\kappa(G)$ ya que podemos aislar este vértice eliminando todos los bordes incidentes para tener G desconectado. Este concepto se puede utilizar en una configuración distribuida por nodos que intercambian sus grados para estimar $\delta(G)$. Tres algoritmos distribuidos localizados para determinar el valor de $\kappa(G)$, digamos k , que funciona solo con el conocimiento del vecino, son propuestos por Jorgic et al. [16, 1]. En el primer algoritmo llamado descubrimiento de vecino local (LND), cada nodo descubre su grado d primero enviando mensajes de saludo a los vecinos y contando las respuestas de ellos. Cada nodo

luego intercambia información de grado con sus vecinos y envía estos datos a los vecinos. Repetir este proceso p veces resulta en grados de nodos transferidos a todos los nodos dentro de p saltos desde ellos. Los nodos pueden simplemente ordenar los grados que recibieron en total y denotar el valor más bajo como el valor de k . El pseudocódigo de una posible implementación se muestra en el Algoritmo 8.16 y aunque el algoritmo original utiliza el campo de tiempo de vida del mensaje que se inicializa a p , proporcionamos una versión del algoritmo SSI que funciona en p Rondas para lograr la misma función.

Algorithm 8.16 Local Neighborhood Detection

```

1: boolean round_over
2: message type deg
3: set of int degs, all_degs
4: degs ←  $d_i$ ; all_degs ←  $d_i$ 
5: for  $i = 1$  to  $p$  do
6:   round_over ← false
7:   while ~round_over do
8:     send degs to  $N(i)$ 
9:     receive deg(j)
10:    degs ← deg(j). $d_j$ 
11:    all_degs ← all_degs ∪ degs
12:    received ← received ∪ {j}
13:    if received =  $N(i)$  then
14:      round_over ← true
15:    end if
16:   end while
17:   degs ← {0}
18: end for
19:  $k \leftarrow \min\{d_j \in all\_degs \cup d_i\}$ 

```

Todos los bordes del gráfico se recorrerán en ambas direcciones en cada ronda, por lo que habrá un total de mensajes $O(pm)$. Aunque este tiempo lineal puede parecer favorable, se necesita un valor alto de p para estimar k más correctamente. Además, $\delta(G)$ es un límite superior en el valor de k , por lo que el valor real puede ser mucho más bajo. En el segundo algoritmo denominado detección de conectividad de subgrafo local (LSCD) propuesto por los mismos autores, se realiza una prueba adicional para encontrar un subgrafo de p -vecinos vecinos de un nodo dado que está conectado con k . Un nodo v determina que el gráfico está conectado a k cuando se cumplen las dos condiciones siguientes:

- Todos los vecinos p -hop de v tienen al menos un grado de k
- La unión de v y el subgrafo de p -hop vecinos de v está conectado a k

El tercer algoritmo busca la eliminación de nodos críticos de los cuales se desconectará el gráfico.

8.7 Notas de capítulo

La conectividad es un concepto fundamental en la teoría de gráficos que tiene aplicaciones inmediatas en las redes de computadoras. Un dígrafo está fuertemente conectado si hay una ruta en ambas direcciones entre cualquier par de vértices u, v en dicho dígrafo. Encontrar componentes fuertemente conectados que sean subgrafos de un dígrafo que estén fuertemente conectados tiene varias aplicaciones, como detectar regiones altamente conectadas en redes biológicas y sociales. Un sistema vial unidireccional en una ciudad también debe estar fuertemente conectado. Revisamos dos algoritmos de tiempo lineal debidos a Tarjan y Kosaraju para detectar dichos SCC de dígrafos.

La eliminación de un punto de articulación o un puente de un gráfico G no conectado conectado deja G desconectado. Necesitamos encontrar estos vértices y bordes de una red

de computadoras para reforzar enrutadores y enlaces adicionales en estas áreas para tener una red más robusta. Describimos un algoritmo basado en DFS de tiempo lineal que hace uso de la propiedad simple de que cualquier vértice en un árbol DFS de una gráfica que no tenga un borde posterior desde su subárbol hasta sus antepasados es un punto de articulación.

Un bloque de un gráfico es un subgrafo conectado máximo sin ningún punto de articulación. Revisamos dos algoritmos de tiempo lineal para identificar bloques en una gráfica. Podemos encontrar la conectividad y la conectividad fuerte de un gráfico en paralelo, como lo demuestran dos algoritmos. Finalmente, describimos un algoritmo heurístico distribuido que estima la conectividad de vértice de una red.

Ceremonias

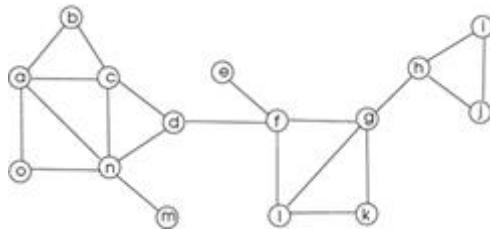


Fig. 8.18 Gráfico de muestra para el ejercicio 1

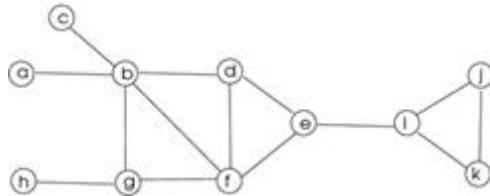


Fig. 8.19 Gráfico de muestra para los ejercicios 2 y 3

Muestre los puntos de articulación, los puentes y los bloques en el gráfico no dirigido de la figura [8.18](#).

- 1.
2. Escriba el pseudocódigo del algoritmo de búsqueda de punto de articulación basado en DFS como un procedimiento principal. Identifique los puntos de articulación en el gráfico no dirigido de muestra de la figura [8.19](#) utilizando el algoritmo basado en DFS. Muestra los valores bajos y numéricos de los vértices en cada iteración.
3. Encuentre los puentes del mismo gráfico de la figura [8.19](#) utilizando el algoritmo de Tarjan .
4. Calcule los bloques de la muestra gráfica representada en la figura [8.20](#) utilizando el algoritmo Hopcroft-Karp mostrando las iteraciones del algoritmo.
5. Encuentre los SCC del dígrafo en la figura [8.21](#) utilizando el algoritmo de Kojarasu . Mostrar el contenido de la pila y los árboles DFS formados.
6. Encuentre el flujo máximo en la red de la figura [8.22](#) utilizando el algoritmo Ford-Fulkerson mostrando todas las iteraciones del algoritmo.

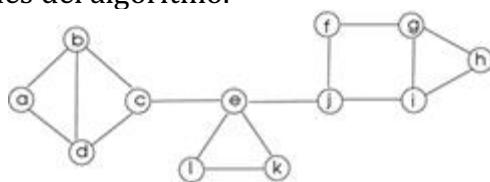


Fig. 8.20 Gráfico de muestra para el ejercicio 4

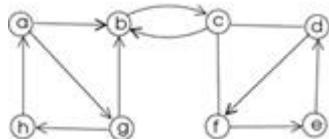


Fig. 8.21 Gráfico de muestra para el ejercicio 5

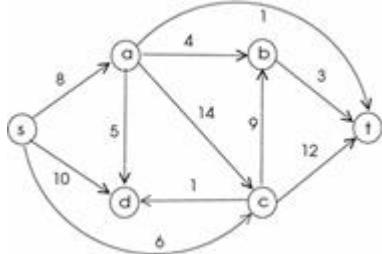


Fig. 8.22 Red de muestra para el Ejercicio 6

Referencias

1. Aho AV, Hopcroft JE, Ullman JD (1983) Estructuras de datos y algoritmos. Addison-Wesley
1. Akl SG (1989) El diseño y análisis de algoritmos paralelos. Prentice Hall, Englewood Cliffs, p. 07632
2. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introducción a los algoritmos, 3^a ed . MIT Press, Cambridge
3. Esfahanian AH (1988) Sobre la evolución de los algoritmos de conectividad. En: Wilson R, Beineke L (eds) Temas seleccionados en la teoría de grafos. Cambridge University Press, Cambridge
4. Esfahanian AH, Hakimi SL (1984) sobre el cálculo de las conectividades de gráficos y dígrafos. Redes 355–366
5. Even S (1979) Graph algorithms, Computer Science Press (serie de ingeniería de software informático). ISBN-10: 0914894218, ISBN-13: 978-0914894216
6. Even S , Tarjan RE (1975) Flujo de red y conectividad de gráficos de prueba. SIAM J Comput 4: 507–518
[MathSciNet Crossref](#)
7. Fleischer L, Hendrickson B, Pinar A (2000) En la identificación de componentes fuertemente conectados en paralelo. Procesamiento paralelo y distribuido, pp 505–511
8. Grama A, Karypis G, Kumar V, Gupta A (2003) Introducción a la computación paralela, 2^a ed . Addison-Wesley, Nueva York
9. Hopcroft J, Tarjan R (1973) Algoritmo 447: algoritmos eficientes para la manipulación de gráficos. Commun ACM 16 (6): 372–378
[Referencia cruzada](#)
10. Matula DW (1987) Determinación de conectividad de borde en $\mathcal{O}(mn)$. En: Actas, 28 simposio sobre fundamentos de la informática, pp 249–251.
11. McLendon W III, Hendrickson B, Plimpton SJ, Rauchwerger L (2005) Encontrar componentes fuertemente conectados en gráficos distribuidos. J Parallel Distrib Comput 65 (8): 901–910
[Referencia cruzada](#)
12. Tarjan RE (1972) Búsqueda en profundidad y algoritmos de grafo lineal. SIAM J Comput 1 (2): 146-160
[MathSciNet Crossref](#)
13. Tarjan RE (1974) Una nota sobre cómo encontrar los puentes de una gráfica. Inf Process Lett 2 (6): 160–161
[MathSciNet Crossref](#)
14. Whitney H (1932) Gráficos congruentes y la conectividad de los gráficos. Am J Math 54: 150–168

9. emparejando

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Una coincidencia de un gráfico es un subconjunto de bordes que no comparten ningún punto final. La coincidencia se puede utilizar en muchas aplicaciones, incluida la asignación de frecuencia de canal en redes de radio, partición de gráficos y agrupación. En una gráfica no ponderada, la coincidencia máxima de una gráfica es el conjunto de bordes que tiene la cardinalidad máxima entre todas las coincidencias en esa gráfica. En un gráfico ponderado de borde, nuestro objetivo es encontrar una coincidencia con el peso total máximo (o mínimo). Encontrar una coincidencia máxima (ponderada) en un gráfico no ponderado o ponderado es uno de los problemas de gráficos raros que se pueden resolver en tiempo polinomial. En este capítulo, revisamos los algoritmos secuenciales, paralelos y distribuidos para los gráficos generales no ponderados y ponderados y los gráficos bipartitos.

9.1 Introducción

Una M coincidente de un gráfico $G = (V, E)$ es un subconjunto de aristas de G que no comparten ningún punto final. En otras palabras, cada vértice de G es incidente a un máximo de un borde de M . También podemos ver una coincidencia de M como un conjunto de bordes independientes en G . Los emparejamientos se pueden usar en una variedad de aplicaciones, como la asignación de frecuencia de canal en redes de radio, la alineación de redes de interacción de proteínas cuando se investiga la similitud entre dos o más redes de este tipo [8]. También se utiliza en algoritmos de partición de gráficos multinivel [15].

La coincidencia máxima es el conjunto M de bordes que no se puede ampliar más mediante la adición de nuevos bordes. En un gráfico no ponderado, máxima coincidencia de un gráfico es el conjunto de bordes que tiene la cardinalidad máxima entre todos los emparejamientos en ese gráfico. En un gráfico ponderado por el borde, nuestro objetivo es encontrar una coincidencia con el peso total máximo (o mínimo). Encontrar una coincidencia máxima (ponderada) en un gráfico no ponderado o ponderado es uno de los problemas de gráficos raros que se pueden resolver en tiempo polinomial. Sin embargo, hay varios algoritmos de aproximación para mejorar el tiempo de ejecución de la coincidencia. Además, los algoritmos de aproximación resultan ser más fáciles de implementar con significativamente menos líneas de código que los algoritmos exactos. El

emparejamiento no ponderado o ponderado en gráficos bipartitos puede tratarse por separado que el emparejamiento general de gráficos, ya que la estructura de gráficos bipartitos puede explotarse para diseñar algoritmos conceptualmente diferentes a los del caso general.

En este capítulo, revisamos el problema de coincidencia en gráficos generales y gráficos bipartitos para casos ponderados y no ponderados. Describimos algoritmos secuenciales, paralelos y distribuidos para estos gráficos.

9.2 Teoría

Revisaremos la teoría, los parámetros y la notación utilizados en la comparación en gráficos generales no ponderados o ponderados y en gráficos bipartitos que comienzan con una comparación no ponderada en esta sección.

9.2.1 Emparejamiento no ponderado

Una coincidencia de una gráfica $G = (V, E)$ es un conjunto de aristas $M \subseteq E$ que no comparten pares los puntos finales. Un vértice coincide o se satura si incide en un borde de coincidencia, de lo contrario, es un vértice libre o no coincidente o insaturado. Un borde que forma parte de una coincidencia se denomina coincidente y no coincidente de lo contrario. Una coincidencia máxima (MM) de un gráfico G no puede hacerse más grande mediante la adición de un nuevo borde, lo que significa que no está contenida en ninguna coincidencia más grande. Un juego máximo (MaxM) de un gráfico de Gtiene el tamaño máximo entre todos los emparejamientos de G. Encocidencia perfecta, cada vértice del gráfico incide en un borde que se incluye en la coincidencia. En otras palabras, todos los vértices de un gráfico están saturados en una coincidencia perfecta. MM y MaxM de un gráfico de muestra se muestran en la Fig. 9.1.

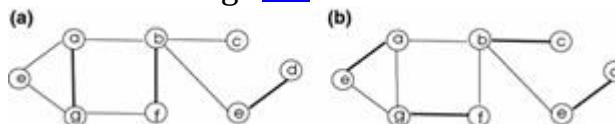


Fig. 9.1 a A MM de tamaño 3, b A MaxMM con tamaño 4 de una gráfica de muestra. La ruta (e , a , g , f , b , c) es una ruta de aumento en (a) y la ruta (g , f , b , c) es una ruta alternativa en (b). No hay ruta de aumento en (b) ya que la coincidencia es máxima. También podemos ver que la coincidencia en (b) es perfecta ya que cada vértice está saturado

Necesitamos algunas definiciones más antes de revisar los algoritmos de coincidencia secuenciales, paralelos y distribuidos.

Definición 9.1 (ruta alterna) Una M : la ruta alterna de una M coincidente es una ruta que alterna entre los bordes en M y los bordes en $E - M$ (bordes no en M).

Definición 9.2 (trayectoria de aumento) Un M - aumentar trayectoria de un juego M es un camino que es una ruta alterna que comienza y termina en vértices insaturados.

Las rutas de aumento y alternas se muestran en la Fig. 9.1 . Una ruta de aumento de una M coincidente que contiene k bordes contiene exactamente $k+1$ bordes que no están en M para un total de $2k+1$ bordes Intentamos encontrar rutas de aumento, ya que podemos aumentar el tamaño de una coincidencia que tiene una ruta de aumento alternando los bordes de la coincidencia en la ruta con bordes que no pertenecen a la coincidencia en la ruta.

Definición 9.3 (árbol alternativo) Un árbol alternativo está arraigado en un vértice libre y cada ruta de este árbol es una ruta alternativa.

Por ejemplo, el árbol arraigado en el vértice e en la figura [9.1 b](#) es un árbol alterno con ramas e , a , b , c , e , g , f y e , a , b , e , d .

Definición 9.4 (factor de gráfico) subgrafo que abarca o un factor de de un gráfico de G contiene todos los vértices de G . A k subgrafo -regular spanning de G se denomina su k -factor. A 1-factor G es una coincidencia perfecta de G que satura todos los vértices de G .

Definición 9.5 (diferencia simétrica de dos gráficos) Diferencia simétrica de dos gráficos G y G' mostrado como $G \oplus G'$ (o $G \triangle G'$) es una gráfica G'' que se induce en el conjunto de bordes que contiene bordes en G o G' pero no en ambos lo que es igual a $(G - G') \cup (G' - G)$.

Esto significa que necesitamos encontrar bordes que estén presentes en solo uno de los gráficos de entrada e incluir vértices que inciden en esos bordes. Dos gráficas y su diferencia simétrica se muestran en la figura [9.2](#) .

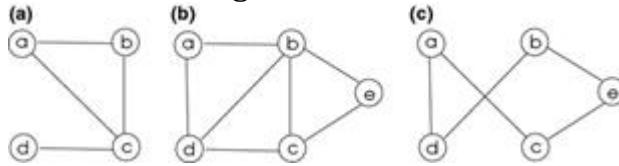


Fig. 9.2 La diferencia simétrica de los dos gráficos en a y b se muestra en c

Lema 9.1 Dado un gráfico G con una M coincidente y un camino de aumento P de M , $M \oplus P$ es de nuevo un juego con una cardinalidad uno más que la de M .

Prueba La diferencia simétrica de una M coincidente con una ruta de aumento P con respecto a M , que es una ruta de aumento que incide con los bordes de M , da como resultado una nueva coincidencia M' que tiene bordes que no formaban parte de la coincidencia en P como bordes coincidentes y los bordes coincidentes en P ahora no tienen coincidencia . En otras palabras, cambiamos los bordes de coincidencia en P con bordes que no coinciden. El nuevo juego M' está de hecho formada tomando la diferencia simétrica de una trayectoria de aumento P de un juego M con M , para dar lugar a un juego que tiene una unidad de tamaño más grande que M , como se muestra a continuación.

$$M' = M \oplus P = (M - P) \cup (P - M)$$

Ahora probar este lema, vemos que P tiene un número impar de aristas y sus bordes se alternan entre los bordes como en M y los bordes no en M . Los bordes en el complemento de la trayectoria P tienen el mismo conjunto de vecinos en M que en M' y los vértices en P tienen exactamente un vecino en M' , por lo tanto M' es también un juego de G . □
Este proceso se denomina aumento de la M coincidente y se puede usar en una cantidad de algoritmos de coincidencia máxima. La coincidencia en la ruta de aumento en la figura [9.3 a](#) se aumenta para tener una coincidencia con un tamaño incrementado en (b).

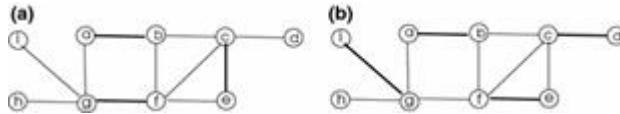


Fig. 9.3 una A MM M de tamaño 3 se muestra en el gráfico de la muestra en una . Sin embargo, hay un camino de aumento $P = \{i, g, f, e, c, d\}$ en esta coincidencia y la nueva coincidencia M' obtenido tomando la diferencia simétrica de M con P da como resultado una coincidencia que tiene un tamaño de 4 como se muestra en b

Ahora podemos establecer un importante teorema de Berge que forma la base de algunos algoritmos de emparejamiento fundamentales [3].

Teorema 9.1 (Berge) Un juego M es máximo si y sólo si no hay trayectorias de aumento con respecto a M .

Prueba Nosotros sabemos por el lema 9.1 que si juego M tiene una trayectoria de aumento, no es de máxima cardinalidad. Ahora tenemos que probar la dirección inversa de la declaración; si no hay rutas de aumento de una M coincidente, entonces M es el máximo. Supongamos que M no es máximo cuando no hay rutas de aumento. Esto significa que existe una coincidencia M' eso es máximo tal que $|M'| > |M|$. Formemos $H = M \oplus M'$. Cada vértice v de H es incidente a lo sumo un M borde y uno M' borde, por lo tanto, cada vértice de H tiene a lo sumo un grado de 2. Además, cada camino de H se alterna con los bordes en M y los bordes en M' . Como cada ruta es alterna, cada ciclo debe ser parejo. Como el tamaño de M' es mayor que el tamaño de M , H tiene una trayectoria P que tiene más bordes coincidentes máximos que M bordes. Esta trayectoria P comienza y termina con un borde máximo de coincidencia, por lo tanto es una trayectoria de aumento de M , es decir, M puede ser agrandado usando P . Esto contradice el primer supuesto de que M no tiene caminos de aumento. □

Esto nos da una plantilla de algoritmo simple para encontrar la M máxima coincidente de un gráfico que consta de los siguientes pasos:

$M \leftarrow \emptyset$

- 1.
2. mientras \exists un camino de aumento P con respecto a M
3. $M \leftarrow M \oplus P$
4. **terminar mientras**
5. volver M

Necesitamos un método para encontrar la ruta de aumento P y veremos que es más conveniente tener diferentes procedimientos para gráficos bipartitos y gráficos generales en las siguientes secciones.

9.2.2 Emparejamiento ponderado

En la comparación de un gráfico ponderado, nuestro objetivo es encontrar la coincidencia con el peso total máximo o mínimo. Una coincidencia ponderada máxima (MaxWM) de un gráfico ponderado $G = (V, E, w)$ con $w : E \rightarrow \mathbb{R}^*$ tiene el peso total máximo entre todos los

emparejamientos máximos ponderados de G , donde el peso de un M coincidente se define como $w(M) = \sum_{e \in M} w(e)$. Del mismo modo, el juego ponderado mínimo (MinWM) de G tiene el menor peso total entre todos los matchings máximos ponderados de G . Por una coincidencia ponderada máxima (MWM) de un gráfico Gponderado , nos referimos a una coincidencia ponderada de G que no puede ser ampliada por un nuevo borde. Veremos que tanto MaxWM como MinWM tienen aplicaciones prácticas. La figura 9.4 muestra MWM nd MaxWM de un gráfico de muestra.

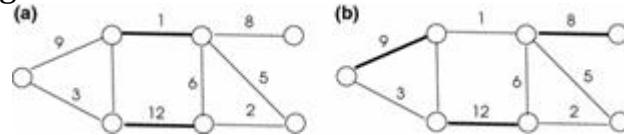


Fig. 9.4 a A MWM de peso total 13, b A MaxWM con peso total 29 de una gráfica de muestra

9.3 Igualación de gráficos bipartitos no ponderados

Una M coincidente en un gráfico bipartito $G = (A \cup B, E)$ Tiene la misma propiedad, es decir, no tiene vértice $v \in A$ o $v \in B$ Es incidente a más de un borde en m . Los emparejamientos máximos y máximos no ponderados y ponderados en gráficos bipartitos se definen de manera similar a los gráficos generales. Las comparaciones bipartitas en dos gráficos de muestra se muestran en la Fig. 9.5 . Una cubierta de vértice de una gráfica. $G = (V, E)$ es un subconjunto V' de su conjunto de vértices tal que cada borde $e \in E$ es incidente a al menos un vértice en V' .

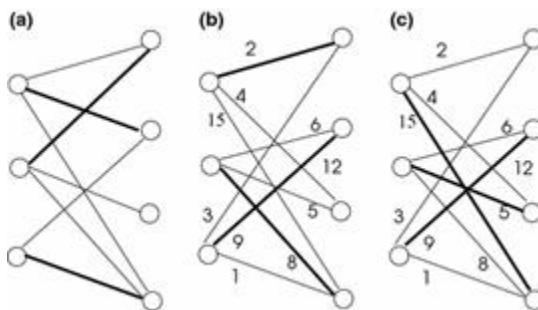


Fig. 9.5 a Un MM de una gráfica bipartita no ponderada, b Una concordancia ponderada máxima de magnitud 19 de una gráfica bipartita ponderada, c la concordancia ponderada máxima de magnitud 29 de la misma gráfica bipartita ponderada de b . Los bordes coincidentes se muestran en negrita

Dado un grafo bipartito $G = (A \cup B, E)$, el barrio $N(S)$ de un conjunto de vértices $S \subset A$ es un conjunto de vértices en B tal que $\forall u \in S$ y $(u, v) \in E, v \in N(S)$. En otras palabras, $N(S)$ es la unión de los vecinos de los vértices en S .

Teorema 9.2 (Hall) Un gráfico bipartito $G = (A \cup B, E)$ contiene una coincidencia que satura cada vértice en A si y solo si,

$$|N(S)| \geq |S|, \text{ for all } S \subset A$$

(9.1)

Prueba En la dirección hacia adelante de la prueba, cada vértice en $S \subset A$ se corresponde con un vértice en $N(S)$ debajo de la coincidencia M y dado que todos los vértices en A están

saturados, con dos vértices distintos en S que coinciden con dos vértices distintos en $N(S)$, esto sigue $|N(S)| \geq |S|$.

En la dirección contraria, tenemos que probar si $|N(S)| \geq |S|$, Entonces G tiene una coincidencia que satura cada vértice de A . Para esta condición, asuma M^* es una coincidencia máxima en G y $u \in A$ no es un vértice saturado por M^* . Consideremos el conjunto W que contiene vértices a los que se puede llegar desde u mediante una ruta alternativa. Dejar $S = W \cap A$ y $T = W \cap B$. Podemos ver que cada vértice en $S \setminus \{u\}$ está saturado y cada vértice en T también está saturado, lo que significa $|T| = |S| - 1$. Entonces podemos escribir $|N(S)| = |T|$ lo que significa $|N(S)| < |S|$ Y por lo tanto una contradicción. Ahora podemos concluir que no existe tal vértice y , por lo tanto, cada vértice en S está saturado. \square

Teorema 9.3 (König 1931) Para cualquier gráfica bipartita $G = (A \cup B, E)$, el tamaño máximo de una coincidencia $\alpha(G)$ es igual al tamaño mínimo de una cubierta de vértice $\beta(G)$ [16].

Mostraremos una prueba constructiva de este teorema que da como resultado un algoritmo eficiente para encontrar la cobertura mínima de vértices de un gráfico bipartito cuando revisamos los algoritmos de cobertura de vértices en el Cap. 10 .

9.3.1 Un algoritmo secuencial que usa rutas de aumento

Usaremos el enfoque de ampliar la coincidencia con rutas de aumento para encontrar una coincidencia máxima en un gráfico bipartito no ponderado. Para ello, consideremos una gráfica bipartita $G = (A \cup B, E)$ y construir un digraph $G' = (A \cup B, E')$ donde cada borde $e \in E'$ se dirige de A a B si $e \notin M$ y se dirige de B a A si $e \in M$. Podemos ver que hay una ruta de aumento con respecto a la coincidencia de M en G si y solo si hay una ruta dirigida desde un vértice no emparejado en A a un vértice no emparejado en B en la gráfica G' . Ahora reescribamos formalmente el algoritmo de correspondencia genérico como se muestra en el Algoritmo 9.1 con el procedimiento $Find_AP$. Para encontrar los caminos de aumento.

Algorithm 9.1 $MaxM_UBGI$

```

1: Input: An undirected bipartite graph  $G = (A \cup B, E)$ 
2: Output: Maximum matching  $M$  of  $G$ 
3:
4:  $M \leftarrow$  any edge  $e \in E$ 
5: repeat
6:    $P \leftarrow Find\_AP(G, M)$ 
7:   if  $P \neq \emptyset$  then
8:      $M \leftarrow M \oplus P$ 
9:   end if
10:  until  $P = \emptyset$ 
11:
12: procedure  $Find\_AP(G(A \cup B), M)$ 
13:    $A' \leftarrow$  a set of free vertices in  $A$ 
14:    $B' \leftarrow$  a set of free vertices in  $B$ 
15:   direct unmatched edges from  $A$  to  $B$ , matched edges from  $B$  to  $A$  to construct  $G'$ 
16:   find the shortest path  $P$  from  $A'$  to  $B'$  in  $G'$ 
17:   if  $P$  not found then
18:     return  $\emptyset$ 
19:   else
20:     return  $P$ 
21:   end if
22: end procedure

```

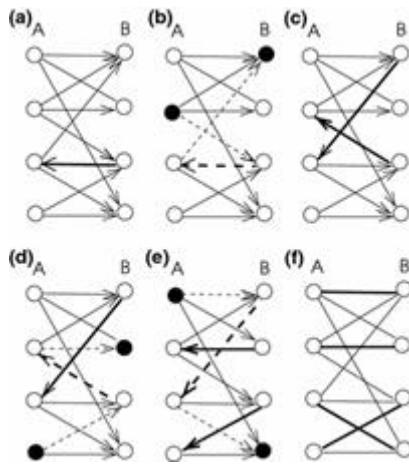


Fig. 9.6 Ejecución de MaxM_BPG1 en un pequeño gráfico bipartito $G = (A \cup B, E)$. La coincidencia seleccionada inicialmente de forma aleatoria M en a se muestra en negrita. El uso de esto coincide con una ruta de aumento que se muestra en líneas de trazo que comienzan desde un vértice en negrita en A y terminan en un vértice en negrita en B que se muestra en b, que está XOR con M para obtener la coincidencia en c . Encontramos una ruta de aumento ahora con esta coincidencia mostrada en líneas discontinuas mostradas en d para formar coincidencias en e en la que se descubre otra ruta de aumento mostrada en líneas discontinuas. El emparejamiento final obtenido por XORing. esta ruta con coincidencia actual no tiene rutas de aumento, ya que todos los vértices ahora están saturados después de 3 pasos

La corrección es evidente desde el procedimiento. Find_AP devuelve una ruta de aumento P si dicha ruta existe en G' , ya que comienza a partir de un vértice libre y termina en un vértice libre utilizando bordes coincidentes.

Teorema 9.4 Algoritmo MaxM_UBG1 tiene complejidad $O(nm)$.

Prueba El límite superior en el tamaño de la coincidencia es $n / 2$ y en cada paso solo podemos extender la coincidencia actual en 1, lo que resulta en un tiempo $O(n)$ para el bucle que llama Find_AP . Se puede encontrar una ruta de aumento en el tiempo $O(m)$ buscando todos los bordes en el peor de los casos. El tiempo total necesario es por lo tanto $O(nm)$. \square

La ejecución de este algoritmo en un gráfico de muestra se muestra en la Fig. 9.6.

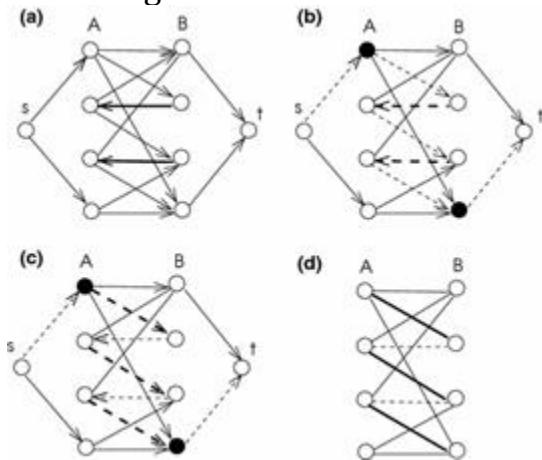


Fig. 9.7 Ejecución de FindBFS_AP en el gráfico de la Fig. 9.6 . Los dos bordes existentes emparejado se hizo dirigidos desde B a A y todos los otros bordes se dirigen desde A a B . Se agrega un vértice s a la izquierda con bordes dirigidos a todos los vértices no emparejados en A y bordes dirigidos desde vértices no emparejados en B al nuevo vértice t . Ahora podemos ejecutar BFS de s a t para encontrar la ruta más corta que se muestra como líneas discontinuas en b de la figura. Sabemos aumentar esta ruta para obtener la ruta

que se muestra en c y la coincidencia máxima final se muestra en d , que es diferente de la coincidencia máxima que se encuentra en la figura 9.6 pero tiene el mismo tamaño

Encontrar rutas de aumento utilizando BFS

Todavía no hemos mostrado cómo buscar una ruta de aumento en G' . Una forma de lograr esto es mediante la adición de una fuente de vértice s a la izquierda de la gráfica bipartita y de la conexión por bordes dirigidos a todos los vértices libres de A . También se agrega un vértice de sumidero t a todos los vértices libres en B dirigiendo todos los bordes desde los vértices de B al vértice t como se muestra en la Fig. 9.7. Luego ejecutamos BFS desde s y devolvemos la ruta más corta como se muestra en el procedimiento FindBFS_AP en el algoritmo 9.2. El camino más corto que comienza desde s y termina en t será un camino de aumento a medida que atraviesa los vértices libres en A y B y tiene que ir a través de algunos bordes coincidentes para poder volver a casa. El tiempo de ejecución para BFS es $O(n+m) \approx O(m)$ en un gráfico denso y, por lo tanto, el tiempo total del algoritmo 9.1 que utiliza el enfoque basado en BFS es $O(nm)$, ya que el tamaño de la coincidencia puede ser como máximo $n/2$.

Algorithm 9.2 MaxM_UBG2

```

1: procedure FINDBFS_AP( $G(A \cup B), M$ )
2:    $A' \leftarrow$  a set of free vertices in  $A$ 
3:    $B' \leftarrow$  a set of free vertices in  $B$ 
4:   direct unmatched edges from  $A$  to  $B$ , matched edges from  $B$  to  $A$  to construct  $G'$ 
5:   add  $s, t$  to  $G'$ 
6:   run BFS from  $s$  of  $G'$  to obtain shortest path  $P$ 
7:   if  $P$  not found then
8:     return  $\emptyset$ 
9:   else
10:    return  $P \setminus [s, t]$ 
11:   end if
12: end procedure

```

La operación de este algoritmo se muestra en la Fig. 9.7 en el mismo gráfico bipartito de la Fig. 9.6 con una coincidencia inicial diferente.

9.3.2 Un algoritmo basado en el flujo

Encontrar la máxima coincidencia en un gráfico bipartito no ponderado $G = (A \cup B, E)$. Se puede realizar utilizando el flujo máximo de la siguiente manera. Añadimos un vértice de origen s y un vértice de sumidero t , conectando s a todos los vértices de A utilizando bordes dirigidos y todos los vértices de B al vértice t de nuevo utilizando bordes dirigidos. También dirigimos los bordes de A a B y damos una capacidad de unidad a todos los bordes para obtener el gráfico G' como se muestra en la figura 9.8. Ahora resolvemos el problema de flujo de red máximo en este gráfico utilizando el algoritmo Ford-Fulkerson. Los bordes que se utilizan en el flujo máximo de red corresponden a los bordes de la coincidencia de máximo de G . Tendremos un flujo a través de un borde o no, ya que todas las capacidades son 1.

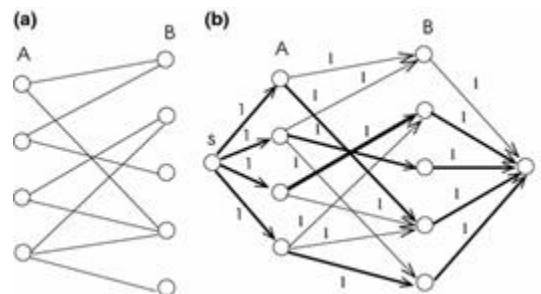


Fig. 9.8 La reconstrucción basada en el flujo de un ejemplo de gráfico bipartito en a se muestra en b. Los bordes de flujo máximo que corresponden a los bordes coincidentes máximos se muestran en negrita

Las capacidades de los bordes de G' son todos 1, por lo tanto, el flujo a través de un borde es 0 o 1. Cada $u \in A$ tiene exactamente un borde de flujo entrante desde el vértice s , entonces puede haber como máximo un borde (u, v) con $v \in B$. Eso puede tener un flujo de 1 por la ley de conservación de flujo. Cada vértice $v \in B$ tiene exactamente un borde de salida que puede pasar el flujo al vértice t y, por lo tanto, como máximo, uno de sus bordes de entrada puede llevar el flujo máximo, de nuevo por la ley de conservación del flujo. Esto quiere decir que los bordes del flujo máximo son disjuntos que resultan en una coincidencia. Por lo tanto, cada $u \in A$ será comparado con a lo sumo un vértice de B formando un juego máximo en G . Cuando tenemos un flujo de valor k , esto corresponde a un juego de tamaño k con el mismo conjunto de bordes de G . Como intentamos maximizar el flujo, estamos maximizando la coincidencia.

La complejidad del tiempo de este algoritmo es la misma de Ford – Fulkerson que es $O(kC)$, donde k es el número de bordes por los que corre el flujo y C es el flujo máximo en el gráfico que es $|A| = n$. El número de aristas en el gráfico recién formado, G' es $2n + m$ ya que agregamos 2 n nuevos bordes a los vértices s y t . La complejidad de este algoritmo es por lo tanto $O(n^2 + nm)$.

9.3.3 Algoritmo de Hopcroft – Karp

El algoritmo Hopcroft-Karp también hace uso de rutas de aumento al encontrar la máxima coincidencia en un gráfico bipartito. Sin embargo, este algoritmo busca muchos caminos simultáneamente en lugar de uno por uno como en el algoritmo anterior y reduce la complejidad del tiempo para $O(\sqrt{nm})$ [14]. El principio de funcionamiento de este algoritmo se basa en el siguiente lema.

Lema 9.2 Dada una gráfica bipartita $G = (A \cup B, E)$ con una M coincidente y una coincidencia máxima M^* para esta gráfica, vamos $|M^*| - |M| = k$ para algún entero k . Sea P la diferencia simétrica $M \oplus M^*$. Entonces P contiene al menos k separar rutas de aumento.

Prueba de todos los bordes del conjunto E obtenido por $M \oplus M^*$ tener un grado máximo de 2, lo que significa que los componentes conectados del subgrafo G' Inducido por E Son caminos y ciclos simples. Consideremos caminos y ciclos por separado en G' . Cada ciclo tiene el mismo número de aristas en M^* sin embargo, al igual que en M , cada recorrido de M tiene exactamente un borde menos en M que en M^* . En $M \oplus M^*$, tenemos exactamente k más bordes en M^* que los bordes en M . Por lo tanto, G' Contiene k vértice disjoint aumentando las rutas de m . \square

El algoritmo consta de una serie de fases y se buscan en cada fase todas las rutas de aumento disjoint de vértice posibles. La diferencia simétrica de la unión de todas estas rutas con la coincidencia existente se calcula para producir la nueva coincidencia, como se muestra en la descripción de alto nivel del algoritmo en el algoritmo 9.3.

Algorithm 9.3 HKT-UBM

```

1: Input: An undirected bipartite graph  $G = (A \cup B, E)$ 
2: Output: Maximum matching  $M$  of  $G$ 
3:  $M \leftarrow \emptyset$ 
4: while  $M$  is not maximum do
5:   find  $\mathcal{P} = \{P_1, \dots, P_k\}$  a maximal set of vertex disjoint shortest  $M$ -augmenting paths
6:    $M \leftarrow M \oplus \mathcal{P}$ 
7: end while

```

Encontrar las rutas de aumento disjointas de un gráfico bipartito $G = (A \cup B, E)$ en cada fase se puede hacer mediante un algoritmo BFS modificado de la siguiente manera. El algoritmo

BFS se ejecuta para cada vértice no emparejado $v \in A$ para formar capas que comiencen en v utilizando caminos alternos de bordes no coincidentes y emparejados para formar un árbol de borde alternativo arraigado en v . El algoritmo BFS se detiene cuando se alcanzan uno o más vértices no coincidentes en B , ya que estamos buscando rutas de aumento más cortas. Un camino que llegue a un vértice no emparejado en B será un camino de aumento, ya que comenzó desde un vértice no emparejado y atravesó bordes alternos. Todos los vértices incomparables alcanzados en B se almacenan en el conjunto F . Una vez que termina esta primera parte de la fase, se ejecuta un algoritmo DFS modificado para cada vértice en F hasta que una ruta de aumento que termina en un vértice libre en A es encontrado. El algoritmo DFS modificado debe ejecutarse a través de bordes alternativos para descubrir una ruta de aumento. Cada camino descubierto P_x se agrega al conjunto \mathcal{P} vértices en P_x se retira del árbol BFS con los vértices huérfanos y este procedimiento se repite para otros vértices libres en B . Al final de cada fase, la nueva coincidencia se forma al XORear el conjunto \mathcal{P} con el juego existente M . La versión detallada de este algoritmo se muestra en el algoritmo 9.4.

Algorithm 9.4 HK2_UBM

```

1: Input: An undirected bipartite graph  $G = (A \cup B, E)$ 
2: Output: Maximum matching  $M$  of  $G$ 
3:  $M \leftarrow \emptyset, \mathcal{P} \leftarrow \emptyset$ 
4: while  $\mathcal{P} \neq \emptyset$  do                                 $\triangleright$  continue until no augmenting paths found
5:   for all  $u \in A$  do
6:      $BFS\_Mod(u)$                                  $\triangleright$  run modified BFS
7:   end for
8:    $F \leftarrow$  all reached free vertices in  $B$ 
9:   for all  $v \in F$  do,
10:     $P_v \leftarrow DFS\_Mod(v)$                        $\triangleright$  run modified DFS
11:    remove  $P_v$  and orphan vertices from BFS graph
12:     $\mathcal{P} \leftarrow \mathcal{P} \cup P_v$ 
13:   end for
14:    $M \leftarrow M \oplus \mathcal{P}$ 
15:    $\mathcal{P} \leftarrow \emptyset$ 
16: end while

```

La corrección del algoritmo es evidente ya que los algoritmos BFS y DFS descubren rutas de aumento en función de sus operaciones. Además, dado que eliminamos la ruta de aumento que se encuentra durante el DFS de los árboles BFS, las rutas descubiertas son diferentes, por lo que es posible incluir la unión de las mismas a la vez. Ejecución de este algoritmo en un gráfico bipartito. $G = (A \cup B, E)$ se representa en la figura 9.9 con $A = \{a, b, c, d, e\}$ y $B = \{1, 2, 3, 4, 5\}$. La primera iteración del algoritmo comienza con $M = \{\emptyset\}$ y todos los vértices son incomparables. El BFS de todos los vértices en A termina en todos los vértices libres en B , lo que da como resultado un árbol BFS igual al gráfico original que no se muestra. Por lo tanto, el conjunto de vértices libres F tiene $\{1, 2, 3, 4\}$ al final de BFS y nos detenemos BFS en la primera capa ya que hemos alcanzado vértices libres en B . Ahora ejecutamos DFS desde cada uno de los vértices en F para encontrar rutas que se incluirán en \mathcal{P} . Deberíamos eliminar los bordes y vértices encontrados en la ruta junto con los vértices huérfanos restantes antes de buscar la siguiente ruta. Hemos seleccionado la coincidencia que se muestra en (a) optando siempre por el primer vértice libre en A desde la izquierda mientras se ejecuta DFS. En la segunda fase, corremos el BFS de los vértices libres d y e en A para obtener los árboles BFS mostrados en (a) con raíz en vértices d y e que terminan en los vértices libres 4 y 5 en B . Tenga en cuenta que el vértice 3 no es un vértice libre y no es necesario que ejecutemos DFS desde allí, pero tuvimos que detenernos en la capa 3 ya que alcanzamos los vértices libres en B . La ejecución de DFS desde los vértices 4 desde el primer árbol y 5 desde el segundo da como resultado la coincidencia máxima final del gráfico con el tamaño 5 en dos fases, que es una coincidencia perfecta ya que cada vértice se compara como se muestra en (b). Si hubiéramos

seleccionando la ruta de aumento del vértice 5 en el árbol BFS de la izquierda, tendríamos bordes (5, c) y (2, d) coincidentes y necesitaríamos una tercera fase que seleccionaría la ruta de aumento (e, 3), (3, b), (b, 5) (5, c), (c, 4), sin embargo, llegaríamos a la misma coincidencia máxima.

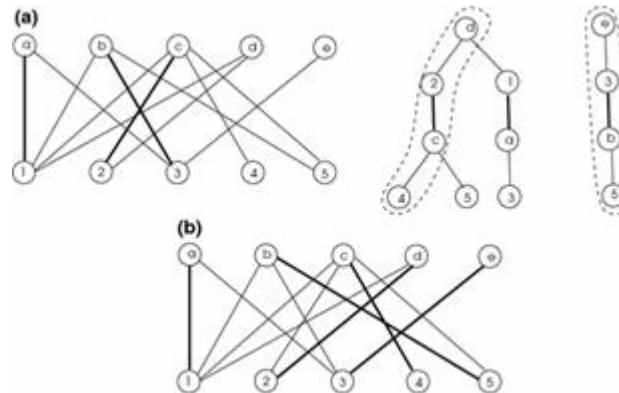


Fig. 9.9 Ejecución del algoritmo Hopcroft-Karp en un pequeño gráfico bipartito. Las rutas de aumento en los árboles BFS están encerradas en líneas discontinuas

Análisis

Primero indicaremos un lema para ayudar al análisis de la complejidad de este algoritmo.

Lema 9.3 Si el camino de aumento más corto con respecto a una M coincidente en un gráfico G tiene l bordes, entonces el tamaño de la coincidencia máxima en G tiene un tamaño máximo de,

$$|M| + \frac{|V|}{l+1}$$

Prueba de dejar M^* ser la coincidencia de máximo de G . Entonces

$M^* \oplus M$ contiene $|M^*| - |M|$ Vértice desunido aumentando rutas por Lemma 9.2. Cada camino de aumento de este tipo tiene al menos l bordes, por lo tanto, al menos $\frac{l+1}{l+1}$ vértices. Por lo tanto, podemos tener a lo sumo $\frac{|V|}{l+1}$ de tales caminos, lo que significa que M puede incrementarse como mucho. \square

Ahora podemos indicar la complejidad del tiempo de este algoritmo.

El algoritmo del teorema 9.5 Hopcroft-Karp tiene un tiempo de ejecución de $O(\sqrt{nm})$.

Prueba Cada fase del algoritmo aumenta la longitud del camino de aumento más corto en al menos uno. Por lo tanto, la longitud del camino de aumento más corto después de $\lfloor \sqrt{n} \rfloor$ las iteraciones serán al menos $\lfloor \sqrt{n} \rfloor + 1$. Habrá a lo sumo $\frac{|n|}{\lfloor \sqrt{n} \rfloor + 1} \leq \sqrt{n}$. Aumentando las rutas a la izquierda y por lo tanto, el algoritmo se ejecutará para otro \sqrt{n} iteraciones a lo sumo. Por lo tanto, el número total de ejecución de bucle será $2\sqrt{n}$ veces. Cada iteración del bucle while requiere tiempo $O(m)$ debido a que los algoritmos BFS y DFS hacen que la complejidad del tiempo de este algoritmo $O(\sqrt{nm})$. \square

Algoritmo paralelo de Hopcroft-Karp

Las operaciones de BFS disjuntas durante la primera parte de cada fase y las operaciones de DFS disjuntas en la segunda parte del algoritmo Hopcroft-Karp lo hacen adecuado para el procesamiento en paralelo. La construcción del gráfico basado en BFS se puede realizar en paralelo iniciando el algoritmo BFS modificado simultáneamente desde vértices libres en el conjunto izquierdo del gráfico bipartito. Este enfoque, entre otros métodos, como el algoritmo DFS lookahead , se experimenta en [2] mediante el uso de multiprocesos.

9.4 Correspondencia no ponderada en gráficos generales

En esta sección, revisaremos los algoritmos de coincidencia secuencial, paralela y distribuida en gráficos generales no ponderados.

9.4.1 Algoritmos secuenciales

El primer algoritmo secuencial es un codicioso que selecciona iterativamente los bordes legales y el segundo algoritmo encuentra MaxM en tiempo lineal.

9.4.1.1 Algoritmo codicioso

Para un gráfico G no ponderado , se puede diseñar un algoritmo codicioso para encontrar MM M de manera que se pueda seleccionar un borde (u, v) al azar, incluirlo en M y eliminar todos los bordes que inciden en u o v de G como se muestra en el algoritmo 9.5.

Algorithm 9.5 Seq_MM

```
1: Input :  $G = (V, E)$ 
2: Output : MM  $M$  of  $G$ 
3:  $M \leftarrow \emptyset$ 
4:  $E' \leftarrow E$ 
5: while  $E' \neq \emptyset$  do
6:   select any  $(u, v) \in E'$ 
7:    $M \leftarrow M \cup \{(u, v)\}$ 
8:    $E' \leftarrow E' \setminus \{(u, v)\} \cup \{all (u, x) \in E' \cup all (v, y) \in E'\}$ 
9: end while
```

Este proceso se repite hasta que no quedan bordes. El funcionamiento de este algoritmo se muestra en la Fig. 9.10 El algoritmo codicioso es correcto, ya que nunca seleccionamos bordes adyacentes para que se incluyan en M (regla de coincidencia), ya que estos se eliminan del gráfico y continuamos hasta que el gráfico se vuelve vacío, lo que significa que no puede haber más. Bordes agregados a M (regla MM). El número de iteraciones del mientras bucle tiene un límite superior como el número de bordes y por lo tanto la complejidad de tiempo de este algoritmo es $O (m)$.

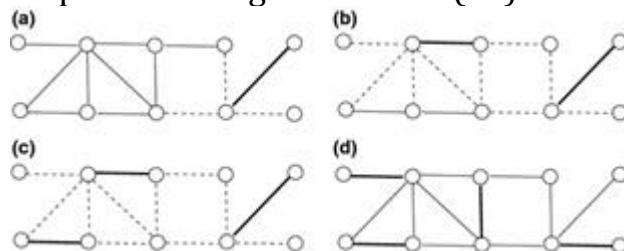


Fig. 9.10 de tres iteraciones del algoritmo de coincidencia codicioso en una muestra resultados del gráfico en MM de cardinalidad 3 como se muestra en una , b y c . Los bordes coincidentes se muestran en negrita y los bordes eliminados se muestran como discontinuos. Se muestra un MaxM del mismo gráfico en d

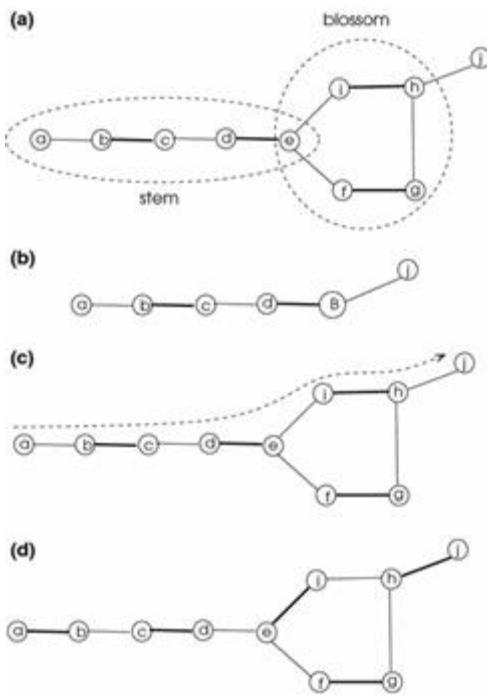


Fig. 9.11 La flor, la flor y el tallo de un gráfico de muestra y la contracción y descontracción de la flor.

9.4.1.2 Algoritmo de flor de Edmond

Hemos visto el método principal para obtener una coincidencia máxima en gráficos no ponderados ; comience con alguna coincidencia inicial M , encuentre una ruta de aumento P con respecto a M y forme $M \oplus P$ para obtener un emparejamiento M' que tiene un tamaño de una unidad más grande que M ; y repita este procedimiento hasta que no se encuentren más rutas de aumento. Describimos métodos para encontrar rutas de aumento en gráficos bipartitos, sin embargo, encontrar rutas de aumento en un gráfico general es más difícil debido a los ciclos alternos impares que no existen en los gráficos bipartitos. Veamos el gráfico de la figura 9.11 a donde se muestra la coincidencia actual con bordes en negrita y tratemos de encontrar un camino de aumento desde el vértice libre a . La búsqueda de ruta de aumento puede atravesar el ciclo de 5 y no puede encontrar la ruta de aumento que tiene a y j como puntos finales al terminar con el borde (i , e) o el borde (f , e) en su lugar. Necesitamos una forma de encontrar rutas de aumento en presencia de ciclos alternos tan extraños.

Edmond presentó un algoritmo de tiempo lineal para superar esta dificultad en gráficos generales no ponderados [7]. Este algoritmo mejora la concordancia actual al encontrar rutas de aumento en el gráfico como en otros algoritmos de concordancia, pero también cuida los ciclos alternos impares. La idea general de este algoritmo es detectar dichos ciclos y eliminarlos al reducirlos a los supernodos y luego continuar la búsqueda de rutas de aumento. Una flor en una gráfica G es un ciclo impar que consiste en $2k + 1$ bordes con exactamente k bordes pertenecientes a la coincidencia de corriente M como se muestra en la Fig. 9.11 a, donde la flor consiste en vértices e , f , g , h , i , y el vástago es una trayectoria alterna de vértices de longitud uniforme a , b , c , d y e , comenzando desde un vértice libre y terminando en la base (o la punta) de la flor. El vértice base de la flor está conectado al tallo y es a la vez parte del tallo y la flor. El tallo y la flor forman la flor . La esencia de este algoritmo se basa en el siguiente teorema que afirmamos sin pruebas.

Teorema 9.6 let $G = (V, E)$ ser un gráfico general no ponderado con una M coincidente. Dejemos que B a flor descubra en esta gráfica y con un emparejamiento la gráfica obtenida después de contraer G usando B como un solo vértice. es un juego máximo en G si y sólo si M es un juego máximo en G .

Por lo tanto podemos investigar rutas de aumento en el gráfico contratado. G' contratando flores a medida que realizamos la búsqueda y siempre que un camino de aumento P se descubre en G' , Que uncontract flores para obtener G y marcar la trayectoria de aumento correspondiente P en G . El último paso de la iteración es actualizar la coincidencia actual para obtener $M \leftarrow M \oplus P$. Un camino de apertura en G' existe si y sólo si existe un Mcamino - augmenting en G . Si encontramos un camino de inauguración P en G' , podemos formar un camino de aumento P en G después de haber retirado la flor B con un vértice base b_B como sigue:

Si P a partir de un vértice libre u en G' pasa a través de b_B y termina en un vértice libre v en G' , entonces P es reemplazado por un camino $u \rightarrow (x \rightarrow \dots \rightarrow y) v$. De tal manera que los bordes en la flor incluidos en P son alternantes.

1. Si P a partir de un vértice libre u en G' termina en b_B , el camino $u \rightarrow v$ es reemplazado por el camino $u \rightarrow (x \rightarrow \dots \rightarrow y) v$ tal camino $P' = u \rightarrow y$ es alternante y yes un vértice libre.

La contracción de la flor del gráfico G en la figura 9.11 a para obtener G' se representa en (b) donde tenemos un camino de aumento y no más flores, por lo tanto, podemos unir la flor en (c) para marcar el camino de aumento que se muestra con líneas discontinuas. Cuando esta ruta recorre la flor B , seleccionamos bordes alternos en B para completar la ruta de aumento que termina en el vértice j . Finalmente, formamos la nueva pareja. $M \leftarrow M \oplus P$ con tamaño 5 que, de hecho, es el máximo para este gráfico, ya que no hay flores o caminos de aumento. Otro ejemplo cuando la ruta alterna termina en una flor se muestra en la figura 9.12. Aplicamos la misma estrategia, encogemos la flor B para obtener G' primero en (b), busca una ruta de aumento en G' y cuando se encuentra tal camino P que termina en B como se muestra en (c), desenrolle B y marque los bordes del camino de aumento dentro de la flor en consecuencia (c). Por último, realizar $M \leftarrow M \oplus P$ para obtener la M coincidente de tamaño 5 en (d), que es máxima ya que no hay otras flores o caminos de aumento.

Como hemos visto en estos ejemplos, hay tres posibilidades al buscar una ruta de aumento en la gráfica G :

No se encontró ruta de aumento: en este caso, la coincidencia actual M es máxima y el algoritmo termina

- 1.
2. Se encuentra un camino de aumento P : $M \leftarrow M \oplus P$; continuar
3. Se encuentra una flor B : la flor B se contrae reemplazándola con su vértice base.

Por lo tanto, este algoritmo buscará una ruta de aumento o una floración o concluirá que no existen en el gráfico, en cuyo caso, la coincidencia es máxima y el algoritmo se detiene. Mostramos un pseudocódigo de alto nivel del algoritmo de Edmond en el algoritmo 9.6. El algoritmo comienza a construir un árbol BFS a partir de un vértice expuesto y etiqueta los bordes en niveles iguales como externos (0) y en niveles impares como internos(I). Cada vez que encontramos dos vértices exteriores que son vecinos, se encuentra una flor con un ciclo impar. Esta flor se contrae a un solo vértice y la búsqueda

continúa. Cuando encontramos un camino de aumento P en el gráfico contraído, entonces todas las flores encontradas hasta ahora no están contraídas y la nueva coincidencia $M \leftarrow M \oplus P$ se calcula .

Algorithm 9.6 Edmond_Blossom

```

1: Input :  $G = (V, E)$ 
2: Output : MaxM  $M$  of  $G$ 
3:  $M \leftarrow \emptyset$ 
4:  $S \leftarrow$  an arbitrary free vertex in  $V$ 
5: for all  $v \in V$  such that  $v$  is saturated do
6:   search for simple paths starting from  $v$ 
7:   shrink any blossoms found
8:   if any found path  $P$  ends at a saturated vertex then
9:      $M \leftarrow M \oplus P$                                 > P is an augmenting path
10:    else if no augmenting paths found then
11:      discard  $v$  in future searches
12:    end if
13: end for

```

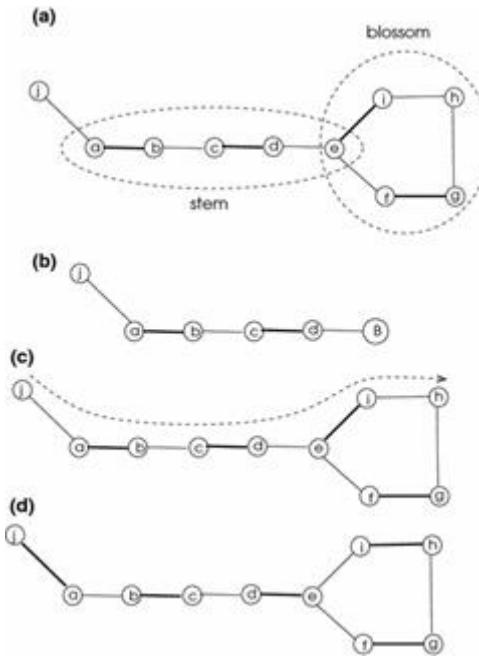


Fig. 9.12 Búsqueda de un camino de aumento cuando el camino termina en una flor

En la figura 9.13 se muestra un ejemplo más detallado con dos flores anidadas . Comenzamos un BFS desde un vértice libre a y etiquetamos los vértices como internos y externos correspondientes a niveles pares e impares respectivamente. Los vértices c y e son vértices externos, por lo tanto, se detecta una flor y esto se contrae al vértice B_1 en el nuevo grafico G' . Encontramos otra flor (B_2) en G' Y esto se contrata para dar la nueva gráfica. G'' . Tenga en cuenta que entre G' y G'' formación, no hemos encontrado un camino de aumento, de lo contrario habríamos descontado B_1 en G' para obtener una nueva coincidencia. Encontramos un camino de aumento en G'' y, por lo tanto, retire las flores y marque el camino de aumento P a través de ellas para alternar. Finalmente, la nueva coincidencia M está formada por $M \leftarrow M \oplus P$.

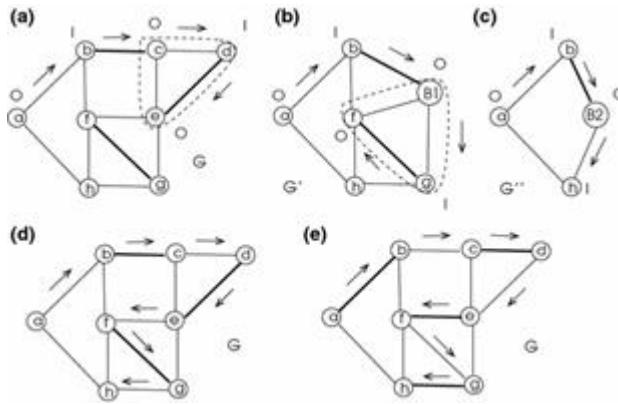


Fig. 9.13 Trabajando el algoritmo de Edmond en una gráfica simple

Análisis

El algoritmo se basa en el teorema de Berge, intenta encontrar una ruta de aumento en un gráfico general y, cuando se encuentra una ruta de este tipo, la amplía. Solo queda mostrar que la contracción y la descontracción de las flores no perturban los caminos de aumento encontrados.

Teorema 9.7 El algoritmo de Edmond tiene una complejidad de tiempo de $O(n^2m)$.

Prueba Habrá a lo sumo n aumentos y habrá como máximo $n / 2$ veces de floración encogiéndose entre dos aumentos. El árbol alternativo se puede construir en tiempo $O(m)$, por lo tanto, el tiempo total tomado es $O(n^2m)$. □

Una mejora en el tiempo de ejecución de este algoritmo para $O(\sqrt{nm})$ fue proporcionado por Micali y Vazirani [19] y se dio una prueba completa en [25].

9.4.2 Un algoritmo distribuido codicioso

En un entorno distribuido, queremos encontrar el MM de una red de modo que cada nodo de la red deba decidir si está saturado o adyacente a un nodo que está saturado al final. Describiremos un algoritmo distribuido que utiliza coloración de bordes. Una coloración de borde de un gráfico G es la asignación de colores en forma de números enteros a cada borde del gráfico, de manera que no haya dos bordes adyacentes que tengan el mismo color que veremos con más detalle en el Capítulo 11. Los bordes del mismo color constituyen una clase de color de borde y podemos ver instantáneamente que cualquier coloreado de borde de un gráfico G proporciona una coincidencia de Gya que los bordes en una clase de color no pueden ser adyacentes. Sin embargo, al intentar encontrar una coincidencia máxima con este método, debemos ser cuidadosos, ya que la unión de las clases de color de borde contiene claramente bordes adyacentes. Sin embargo, podemos comenzar con la clase de color 1, por ejemplo, incluir todos los bordes de esta clase en la coincidencia, ya que estos no son adyacentes a la definición de coloración de bordes y luego continuar con la clase de color 2. Deberíamos incluir un borde en esta clase solo si no es adyacente a ningún borde previamente emparejado.

Un algoritmo distribuido basado en esta observación se propone en [12] para encontrar una coincidencia máxima en una red que ya está coloreada con k colores. Es un algoritmo SSI que funciona en rondas bajo el control de un nodo raíz. Hay k rondas que comienzan con la ronda 1 y en la ronda r , cualquier nodo que tenga un borde incidente (u, v) coloreado con r verifica si puede incluir (u, v) en la coincidencia legal. Es decir,

no hay otros bordes adyacentes a (u, v) que se incluyen en el emparejamiento en las rondas anteriores. Esbozaremos una posible implementación de este algoritmo como en [9] pero utilizando un FSM. Hay tres estados de un nodo, como se muestra en la Fig. 9.14 .

- UNMATCHED: Inicialmente, todos los nodos están en estado UNMATCHED, lo que significa que pueden competir para ser un nodo emparejado.
- EMPAREJADO: cualquier nodo que tenga un borde de incidente que se determine como un borde coincidente entra en este estado.
- NEIGH_MATCHED: cuando un nodo tiene un vecino que está MATCHED, se asigna a este estado.

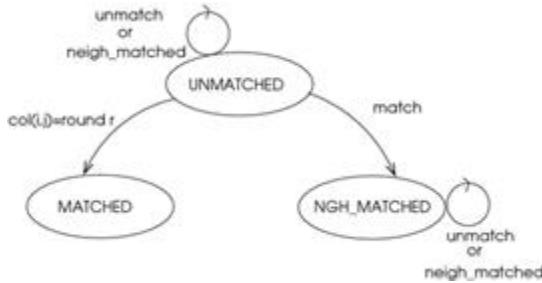


Fig. 9.14 El FSM del algoritmo coincidente para el nodo i con un nodo vecino j

Tenemos los siguientes tipos de mensajes:

- Round (r) : Enviado por la raíz i para iniciar la ronda r .
- Match (r) : enviado por un nodo i que coincide con sus vecinos.
- Unmatch : enviado por un nodo i que no tiene un borde incidente con el mismo color que el número r redondo . Esto es necesario para la sincronización.
- Neigh_match (r) : enviado por un nodo i que no tiene un borde incidente con el mismo color que el número redondo r .

El pseudocódigo para una sola ronda de este algoritmo distribuido para un nodo i se muestra en el algoritmo 9.7.

Algorithm 9.7 Edgocol_MM

```

1: int i,j                                /* i is this node, j is the sender of a message
2: message types round,match,unmatch
3: states UNMATCHED, MATCHED, NEIGH_MATCHED
4: boolean round_over, round_recv = false
5: curr_neighs ← N(); received, neighs_removed ← {0}
6: currstate ← UNMATCHED
7: for r = 1 to k do
8:   { round r for all nodes}
9:   while ¬round_over do
10:     receive msg(j)
11:     case msg(j).type of
12:       round(r):
13:         if currstate ≠ MATCHED then
14:           if (∀j ∈ curr_neighs such that col(i,j) = r) then
15:             currstate ← MATCHED
16:             send match to j
17:             send neigh_match to curr_neighs \ {j}
18:           else send unmatch to curr_neighs
19:             round_recv ← true
20:       match(r):
21:         currstate ← NEIGH_MATCHED
22:         received ← received ∪ {j}
23:         received ← received ∪ {j}
24:         neighs_removed ← neighs_removed \ {j}
25:       unmatch(r):
26:         received ← received ∪ {j}
27:     if round_recv ∧ (received = curr_neighs) then
28:       curr_neighs ← (curr_neighs \ neighs_removed); received ← ∅
29:       round_recv ← false; round_over ← true;
30:     end if
31:   end while
32: end for
  
```

La operación de este algoritmo en una red de muestra pequeña se muestra en la figura 9.15 . Este algoritmo encuentra correctamente una coincidencia máxima en una red, ya que obedecemos la regla de coincidencia en cada ronda al no considerar los bordes

adyacentes de los bordes combinados y también, continuamos hasta que se considera cada clase de color y, por lo tanto, la coincidencia es máxima. No habrá un total de k rondas para un k - de color de borde de la red y cada borde se atravesado como máximo una vez por el partido , unmatch o neigh_match mensajes y por lo tanto el número total de mensajes transferidos es $O(km)$.

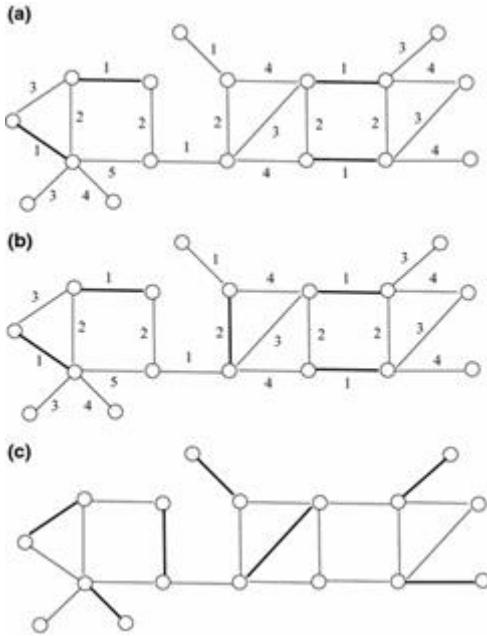


Fig. 9.15 Ejecución del algoritmo 9.7 en una red de muestra. La primera y la segunda ronda se muestran en a y b respectivamente. En solo dos rondas se obtiene una coincidencia máxima de tamaño 5. La coincidencia máxima de tamaño 7 para esta red se muestra en c

9.5 Correspondencia de gráficos bipartitos ponderados

Bordes de una gráfica bipartita ponderada $G = (A \cup B, E, w)$, $w : E \rightarrow \mathbb{R}$ tienen pesos asociados con ellos. Nuestro objetivo es buscar un peso máximo total o mínimo que coincide con el máximo en dichos gráficos.

9.5.1 Algoritmo codicioso

Podemos implementar una estrategia codiciosa en la que siempre seleccionamos el borde de mayor peso disponible de todos los bordes disponibles. Necesitamos clasificar los pesos de los bordes inicialmente y luego verificar la disponibilidad. El tiempo de ejecución de este algoritmo está dominado por la operación de clasificación y, por lo tanto, necesitamos $O(m \log m)$ y el factor de aproximación es $1/2$ [22].

Algorithm 9.8 MWM_BPG

```

1: Input: An undirected weighted bipartite graph  $G = (A \cup B, E)$ 
2: Output: Maximal weighted matching  $M$  of  $G$ 
3:
4:  $Q \leftarrow$  sorted edges of  $G$  in the order of decreasing weights
5: while  $Q \neq \emptyset$  do
6:    $(u, v) \leftarrow Q.front$ 
7:    $M \leftarrow M \cup (u, v)$ 
8:   remove all edges incident to  $u$  or  $v$  from  $Q$ 
9: end while
```

La ejecución de este algoritmo en un pequeño gráfico bipartito se muestra en la figura 9.16 .

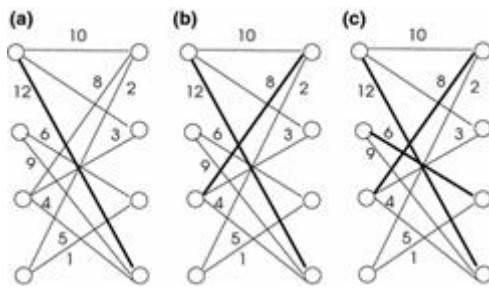


Fig. 9.16 Ejecución del algoritmo codicioso para encontrar MWM en un gráfico bipartito ponderado con los pesos que se muestran al lado de los bordes. El borde de mayor peso disponible se selecciona en cada paso para obtener la coincidencia final del peso total 26 que se muestra con líneas en negrita en c en 3 iteraciones

9.5.2 El método húngaro

El método húngaro, llamado por su desarrollador Kuhn ya que se basa en las ideas anteriores de dos matemáticos húngaros, König y Egervary , encuentra la máxima coincidencia en un gráfico bipartito completo ponderado con el mismo orden de conjuntos de vértices bipartitos en tiempo lineal [17] . Este método resuelve el problema de asignación que tiene como objetivo asignar objetos tales como máquinas, personas, procesadores a tareas al encontrar la coincidencia ponderada mínima o máxima en dicho gráfico. Supongamos que nos dan un conjunto de personas y un conjunto de tareas que estas personas pueden realizar, que son los vértices de la gráfica bipartita consecutivamente. El peso en un borde (u, v) muestra el tiempo requerido por la persona u para realizar la tarea v y nuestro objetivo es tener la cantidad mínima de tiempo para realizar todas las tareas. Podríamos tener procesadores de un sistema de multiprocesamiento en lugar de personas y las tareas serían módulos de software que se ejecutan en estas máquinas en este caso. Describiremos este método en dos enfoques equivalentes; utilizando la matriz de costos y como un método gráfico-teórico (algoritmo Kuhn- Munkres). Estos dos enfoques tienen la misma complejidad de tiempo.

9.5.2.1 Interpretación matricial

Podemos considerar el problema de asignación como un gráfico bipartito ponderado $G = (A \cup B, E)$ donde necesitamos asignar vértices en A a los vértices en B para obtener un mapeo óptimo. El gráfico puede ser representado por la matriz de costos C con elementos. c_{ij} denotando el costo de asignar $a_i \in A$ a $b_j \in B$. El orden de A y B debe ser igual y, si no se proporciona, podemos simplemente agregar filas o columnas ficticias con 0 entradas para igualarlas. Este algoritmo se basa en las siguientes dos observaciones.

- Si un número se agrega o se resta de todas las entradas de una fila o columna de una matriz de costos C_i para obtener una matriz de costos C_{i+1} , luego en asignación óptima para la matriz de costos C_{i+1} También es una asignación óptima para la matriz de costos C_i .
- Tenemos una asignación óptima de a_i a b_j Si $c_{ij} = 0$. En otras palabras, si podemos reducir a cero el costo de asignar un elemento de A a un elemento de B , esta asignación es óptima.

El enfoque de la interpretación matricial del método húngaro es, entonces, transformar la matriz de costos original. C_i a una matriz C_{i+1} para proporcionar una asignación cero en cada fila y columna mediante las operaciones de suma y resta. Podemos tener los siguientes pasos del algoritmo para lograr este objetivo.

Reducir filas : resta el menor valor de cada fila de todas las entradas en su fila.

- 1.
2. Reducir columnas : reste el valor mínimo de cada columna de todas las entradas en su columna.
3. Cubrir ceros : cubra las filas y columnas de entrada cero utilizando el número mínimo de líneas.
4. Si el número de líneas es n goto 6.
5. de lo contrario, encuentre el elemento descubierto más pequeño x . Reste x de todos los elementos descubiertos de C y agregue x a los elementos que están en la intersección de las líneas de cobertura en 3. Ir a 1.
6. Asignación : seleccione una fila o columna con solo un cero y asigne. Si no lo encuentra, seleccione arbitrariamente. Seleccione otras asignaciones para que no se asigne dos tareas a las mismas personas.

Veamos el funcionamiento de este algoritmo a través de un ejemplo que se muestra en la figura [9.17 a.](#)

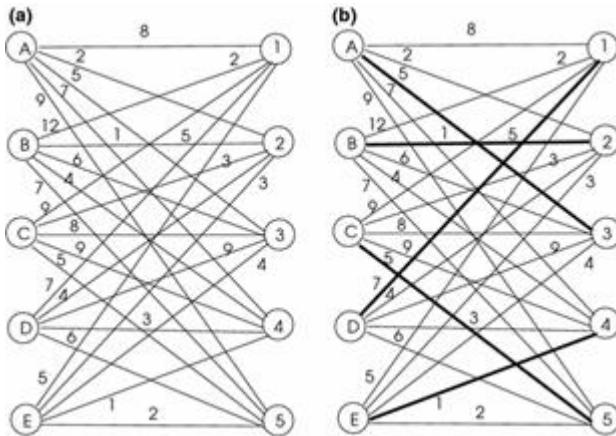


Fig. 9.17 Un ejemplo de gráfico bipartito totalmente ponderado no ponderado para probar el algoritmo húngaro

La matriz de costos C para este gráfico se proporciona a continuación y los dos primeros pasos del algoritmo que reducen las filas y luego las columnas dan como resultado las matrices que se muestran.

$$\begin{array}{c}
 \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline A & 8 & 2 & 5 & 7 & 9 \\ B & 12 & 1 & 6 & 4 & 7 \\ C & 9 & 3 & 8 & 9 & 5 \\ D & 7 & 4 & 9 & 3 & 6 \\ E & 5 & 3 & 4 & 1 & 2 \end{array} & \xrightarrow{\text{(1)}} &
 \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline A & 6 & 0 & 3 & 5 & 7 \\ B & 11 & 0 & 5 & 3 & 6 \\ C & 6 & 0 & 5 & 6 & 2 \\ D & 4 & 1 & 6 & 0 & 3 \\ E & 4 & 2 & 3 & 0 & 1 \end{array} & \xrightarrow{\text{(2)}} &
 \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline A & 2 & 0 & 0 & 5 & 7 \\ B & 7 & 0 & 2 & 3 & 6 \\ C & 2 & 0 & 2 & 6 & 2 \\ D & 2 & 1 & 3 & 0 & 3 \\ E & 2 & 2 & 0 & 0 & 1 \end{array}
 \end{array}$$

$$\begin{pmatrix} 4 & 0 & 3 & 0 & 1 \end{pmatrix}$$

Cubrir filas y columnas con ceros da como resultado la primera matriz C a la izquierda a continuación con las entradas cubiertas mostradas en negrita. Dado que el número de líneas cubiertas es 4, que es menor que 5, debemos continuar con el

algoritmo. Seleccionamos el valor más bajo descubierto que es 2 y restamos 2 de todos los valores descubiertos y lo agregamos a las entradas en la intersección de las entradas cubiertas para obtener la segunda matriz y cubrir esta matriz nuevamente esta vez con las filas y columnas cubiertas que se muestran en figuras en negrita

$$A \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & \mathbf{0} & 5 & 7 \\ 7 & 0 & 2 & 3 & 6 \\ 2 & \mathbf{0} & 2 & \mathbf{6} & 2 \\ 2 & 1 & 3 & \mathbf{0} & 3 \\ 2 & 2 & \mathbf{0} & 0 & 1 \end{pmatrix} \rightarrow C \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & \mathbf{0} & 7 & 7 \\ 5 & \mathbf{0} & 0 & 3 & 4 \\ \mathbf{0} & 0 & 0 & \mathbf{6} & \mathbf{0} \\ \mathbf{0} & 1 & 1 & \mathbf{0} & 3 \\ 2 & 4 & \mathbf{0} & 0 & 1 \end{pmatrix}$$

Encontramos que el número de filas y columnas cubiertas es 5, que es el número de vértices del gráfico bipartito, por lo tanto, nos detenemos y continuamos con el paso de asignación. Primero buscamos filas individuales, ya que esto significa que esa persona solo puede hacer la tarea que tiene 0 en su columna. La persona A tiene dicha propiedad y le asignamos la tarea 3 y eliminamos la columna de la tarea 3, ya que esta tarea no se puede asignar a otra persona. La persona F puede realizar las tareas 3 y 4, pero como la tarea 3 ya está asignada, debemos asignarle la tarea 4 y eliminar la columna 4 de la matriz. De manera similar, a la persona C se le asigna la tarea 2 y la persona E solo se puede asignar a la tarea 1 que deja a la persona D sola con la tarea 5, aunque es capaz de realizar las tareas 1, 2, 3 y 5. Las asignaciones en 0 ubicaciones se muestran en negrita en la matriz de costos final abajo a la izquierda y se muestra la asignación real en la matriz de costos original usando estos valores en la parte inferior derecha. Para el tiempo total tomado, calculamos esto como $5 + 1 + 5 + 7 + 1 = 19$ unidades de la matriz de costos original. Esta comparación se representa en el gráfico bipartito de la figura [9.17 b.](#)

$$A \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & \mathbf{0} & 7 & 7 \\ 5 & \mathbf{0} & 0 & 3 & 4 \\ 0 & 0 & 0 & 6 & \mathbf{0} \\ \mathbf{0} & 1 & 1 & 0 & 3 \\ 2 & 4 & 0 & \mathbf{0} & 1 \end{pmatrix} \leftrightarrow C \begin{pmatrix} 8 & 2 & 5 & 7 & 9 \\ 12 & 1 & 6 & 4 & 7 \\ 9 & 3 & 8 & 9 & 5 \\ 7 & 4 & 9 & 3 & 6 \\ 5 & 3 & 4 & 1 & 2 \end{pmatrix}$$

9.5.2.2 Algoritmo de Kuhn- Munkres

El algoritmo Kuhn - Munkres fue propuesto por primera vez por Kuhn [[17](#)] y luego analizado por Munkres [[20](#)] para resolver el problema de asignación de manera eficiente. Antes de describir el funcionamiento de este algoritmo, necesitamos hacer algunas definiciones.

Definición 9.6 (etiquetado de vértice) Un etiquetado de vértice de un gráfico $G = (V, E)$ es una función $l: V \rightarrow \mathbb{R}$. Un etiquetado legal permite el etiquetado dos vértices u y v de un gráfico bipartito G tal que,

$$l(u) + l(v) \geq w(u, v), \forall (u, v) \in E \text{ (labeling rule)}$$

(9.2)

Definición 9.7 (gráfico de igualdad) El gráfico de igualdad de un gráfico $G = (V, E)$ con respecto a una función de etiquetado l es un gráfico G_l tal que

$$E_l = \{l(u, v) : w(u, v) = l(u) + l(v)\}$$

(9.3)

Podemos ver inmediatamente que G_l es un subgrafo que abarca G ya que contiene todos los vértices de G y $E_l \subseteq E$. A medida que seleccionamos bordes de G en G_l que solo proporcionan equivalencia. Un posible etiquetado de una gráfica bipartita. $G = (A \cup B, E)$ se puede lograr mediante la asignación de cada vértice del conjunto B el máximo de los pesos de todos incidente bordes a la misma y 0 para cada vértice del conjunto A . Es decir,

$$\forall u \in A, l(u) = 0, \text{ and } \forall v \in B, l(v) = \max\{w(u, v)\}$$

(9.4)

De esta manera, nos aseguramos de que se cumpla la regla de etiquetado y de que se obtenga un gráfico de igualdad inicial. En la figura 9.18 se muestra un gráfico bipartito que está etiquetado en consecuencia y su gráfico de igualdad . Tenga en cuenta que cuando el gráfico bipartito no está completamente conectado, debemos agregar bordes con 0 pesos.

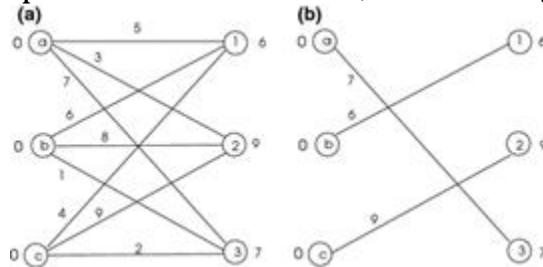


Fig. 9.18 a Un gráfico bipartito ponderado legalmente etiquetado, b Su gráfico de igualdad

El siguiente teorema de Kuhn y posterior de Munkres proporciona la base para este algoritmo de asignación teórico-gráfica.

Teorema 9.8 (Kuhn- Munkres) Dado un gráfico bipartito ponderado $G = (A \cup B, E)$ y su gráfica de igualdad $G_l = (A \cup B, E_l)$, M es una combinación perfecta en G_l si y sólo si M es un peso máximo a juego en G .

Prueba Deje que nosotros suponemos G_l contiene una coincidencia perfecta M . La coincidencia M también es una coincidencia perfecta en G ya que todos los bordes de G_l están contenidos en el conjunto de bordes E de G que significa,

$$w(M) = \sum_{e \in M} w(e) = \sum_{v \in V(G)} l(v)$$

(9.5)

Supongamos M' es cualquier coincidente en G . Entonces, la regla de etiquetado implica,

$$w(M') = \sum_{e \in M'} w(e) \leq \sum_{v \in V(G)} l(v)$$

(9.6)

Por lo tanto $w(M') \leq w(M)$ lo que significa que M es un juego máximo de la gráfica G . \square

Por lo tanto, encontrar la coincidencia máxima de peso en el gráfico original G se reduce a encontrar una combinación perfecta del gráfico de igualdad G_l . Ahora podemos formar los pasos del algoritmo basado en este teorema.

l es un etiquetado legal de G , M es una coincidencia inicial de G_l

- 1.
2. mientras que M no es una coincidencia perfecta en G_l
3. encontrar un camino de aumento P en G_l
4. $M \leftarrow M \oplus P$
5. Si $P = \emptyset$
6. $l' \leftarrow l$ tal que $E_l \subset E_{l'}$

Encontrando un nuevo etiquetado. ^r Es crucial en el funcionamiento de este algoritmo. Para un etiquetado legal del gráfico G , primero definamos las relaciones de vecindad de un vértice en G_l y el conjunto S ,

$$N_l(u) = \{v : (u, v) \in E_l\}, N_l(S) = \bigcup_{u \in S} N_l(u)$$

(9.7)

por $S \subseteq A$ y $T = N_l(S) \neq B$, definamos parámetro a_l

$$a_l = \min\{u \in S, v \notin T\} \{l(u) + l(v) - w(u, v)\}$$

(9.8)

Ahora, el etiquetado mejorado ^r para cualquier vértice de G se puede especificar en términos del etiquetado anterior l usando ^{a_l} como sigue.

$$l'(x) = \begin{cases} l(x) - a_i, & \text{if } x \in S \\ l(x) + a_i, & \text{if } x \in T \\ l(x) \text{ otherwise} \end{cases}$$

(9.9)

Ahora podemos escribir el pseudocódigo del algoritmo Kuhn- Munkres como se muestra en el algoritmo 9.1.

Algorithm 9.9 Kuhn-Munkres_WBM

```

1: Input: An undirected weighted complete bipartite graph  $G = (A \cup B, E)$ 
2: Output: Maximum weighted matching  $M$  of  $G$ 
3:
4:  $E_I \leftarrow \{(u, v) \in E(G) : w(u, v) = l(u) + l(v)\}$                                 // labeling of G according to Eqn. 9.4
5:  $M \leftarrow$  some initial matching of  $G$ 
6: while  $A$  is not  $M$ -saturated in  $G_I$  do
7:   select an unmatched vertex  $x \in A$ 
8:    $S \leftarrow \{x\}$ ,  $T \leftarrow \emptyset$ 
9:   repeat
10:    if  $N_{G_I}(S) = T$  then
11:       $a_I \leftarrow \min_{u \in S, v \in T} [l(u) + l(v) - w(u, v)]$ 
12:      for all  $u \in S$  do
13:         $l(u) \leftarrow l(u) - a_I$ 
14:      end for
15:      for all  $v \in T$  do
16:         $l(v) \leftarrow l(v) + a_I$ 
17:      end for
18:      update  $G_I$ 
19:    end if
20:    select  $v \in \{N_{G_I}(S) - T\}$ 
21:    while  $v$  is  $M$ -saturated and  $N_{G_I}(S) \neq T$  do
22:       $u \leftarrow$  a vertex in  $A$  matched to  $v$  in  $M$ 
23:       $S \leftarrow S \cup \{u\}$ ,  $T \leftarrow T \cup \{v\}$ 
24:      if  $N_{G_I}(S) \neq T$  then
25:        select  $v \in \{N_{G_I}(S) - T\}$ 
26:      end if
27:    end while
28:  until  $v$  is  $M$ -unsaturated
29:   $P \leftarrow M$ -augmenting path from  $x$  to  $v$ 
30:   $M \leftarrow M \oplus P$ 
31: end while

```

En la figura 9.19 se representa una operación de ejemplo de este algoritmo en un pequeño gráfico bipartito ponderado .

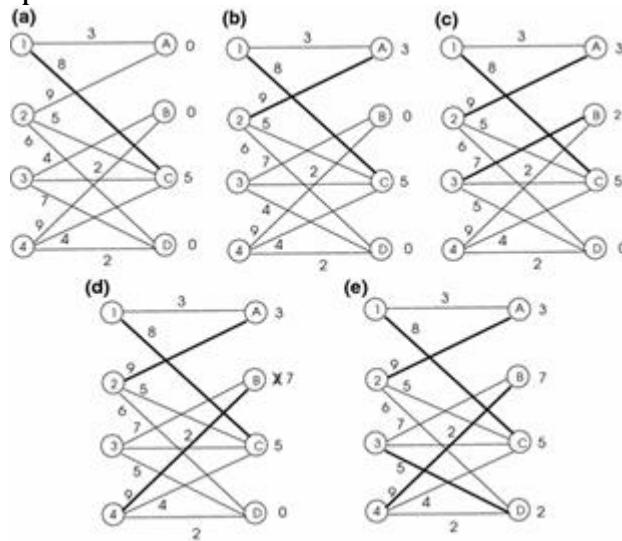


Fig. 9.19 Ejecución del algoritmo Kuhn- Munkres en un pequeño gráfico bipartito ponderado. Los bordes de la coincidencia obtenida en cada iteración se muestran en negrita

Análisis

Hay n fases del algoritmo y en cada fase el tamaño de la coincidencia se incrementa en 1. El cálculo de holgura inicial toma $O(n^2)$ hora. Cuando un vértice se mueve de S a T , calculamos los holguras para todos $y \in T$. En cada fase, la mayoría de los n vértices van de S a T , por lo tanto n relajan los recálculos, cada uno en el tiempo $O(n)$, para un total de $O(n^2)$. El algoritmo lleva $O(n^3)$ tiempo en total.

9.5.3 El algoritmo de subasta

El problema de emparejamiento en gráficos bipartitos ponderados se puede resolver de manera eficiente utilizando el método de subasta que se basa en la teoría de juegos. Las subastas en la vida diaria involucran a un subastador que abre la licitación y los licitadores que presentan las ofertas, y el objeto en cuestión es adquirido por el licitador que ofrece el precio más alto. Los algoritmos de subasta se basan en este principio en el que un gráfico bipartito $G = (A \cup B, E)$ se considera con el conjunto de vértices A como compradores y B como objetos [4]. Cada objeto i tiene un precio, p_i asociado con él y el peso de un borde entre un postor i y un objeto j , w_{ij} muestra la cantidad en que el postor i valora el objeto j , en otras palabras, es el costo del objeto como lo ve el comprador i . El algoritmo consiste en la fase de licitación y la fase de asignación. Cada objeto se puede vender a una sola persona y cada persona puede comprar solo un objeto.

Para cada objeto, un comprador tiene un beneficio y un precio para ser el propietario de ese objeto. La ganancia de un objeto por parte de un comprador es la diferencia entre el beneficio y el precio de un objeto. El algoritmo 9.10 muestra el pseudocódigo para el algoritmo de subasta secuencial adaptado de [24]. En cada iteración, el primer elemento de los compradores del conjunto B se selecciona, luego se encuentra un objeto con el beneficio máximo para ese comprador. El segundo objeto con mayor rentabilidad también se calcula y la oferta se calcula como la diferencia de las dos primeras mejores ganancias. El objeto que proporciona esta oferta se asigna al comprador en la fase de asignación. El precio nuevo para el objeto se incrementa con la oferta y un pequeño valor designado como ϵ que puede ser inicializado a $\delta \leftarrow 1/(n+1)$. Las iteraciones continúan hasta que cada comprador es asignado a un objeto.

Algorithm 9.10 Auction_Alg

```

1: Input :  $G(A \cup B, E, w)$                                  $\triangleright$  undirected weighted bipartite graph
2: Output : Matching  $M$ 
3:  $B = \{1, \dots, n\}$  : set of buyers
4:  $M \leftarrow \emptyset$ 
5: initialize  $\epsilon$ 
6: for  $j=1$  to  $n_2$  do
7:    $p_j \leftarrow 0$                                           $\triangleright$  initialize prices for objects to all zeroes
8: end for
9: while  $B \neq \emptyset$  do
10:  select  $i \in B$                                         $\triangleright$  start auction
11:   $j_b \leftarrow \max_j \{w_{ij} - p_j\}$                        $\triangleright$  select an available buyer
12:   $x_j \leftarrow w_{ij_b} - p_{j_b}$                             $\triangleright$  find the best object for this buyer
13:  if  $x_j > 0$  then                                      $\triangleright$  profit for the best object
14:     $t_i \leftarrow \max_{j \neq j_b} \{w_{ij} - p_j\}$             $\triangleright$  second best profit
15:     $p_{j_b} \leftarrow p_{j_b} + x_j - t_i + \epsilon$            $\triangleright$  update the bid for object
16:     $M \leftarrow M \cup (i, j_b)$ ;  $B \leftarrow B \setminus \{i\}$        $\triangleright$  assign buyer to object
17:     $M \leftarrow M \setminus (k, j_b)$ ;  $B \leftarrow B \cup \{k\}$        $\triangleright$  release previous owner  $k$ 
18:    update  $\epsilon$ 
19:  else
20:     $B \leftarrow B \setminus \{i\}$                                 $\triangleright$  no object with profit found for buyer  $i$  found,
21:  end if
22: end while

```

Mostraremos el funcionamiento de este algoritmo utilizando el gráfico bipartito ponderado de la figura 9.19 como ejemplo en la figura 9.20 . Tenemos cuatro compradores 1, 2, 3 y 4 y cuatro objetos A , B , C y D con las ofertas iniciales establecidas en cero y $\epsilon = 0$. Comenzamos con el comprador de índice más bajo 1 que tiene la ganancia más alta en el objeto C con costo 8 y la segunda ganancia más alta es el objeto A con ganancia 3. La oferta es, por lo tanto, $8 - 5 = 3$ para el objeto C como se muestra. El comprador 1 se asigna a este objeto y comenzamos la segunda iteración con el comprador 2. Del mismo modo, el comprador 2 se asigna al objeto A con la oferta 3, que es la diferencia de sus dos mejores ganancias como se muestra en (b) y el comprador 3 se asigna a el objeto B con la oferta 2 como se muestra en (c). Tenemos una situación diferente en (d) donde el comprador puede hacer una oferta 4 7, que es la diferencia entre sus dos mejores beneficios, para el

objeto B . Esta oferta es mayor que la oferta actual de 2 para el objeto B y, por lo tanto, liberamos al comprador 3 del objeto B y asignamos el comprador 4 a este objeto. Finalmente, el comprador 3 se reasigna esta vez al objeto D como se muestra en (d), que es la coincidencia ponderada máxima para este gráfico bipartito.

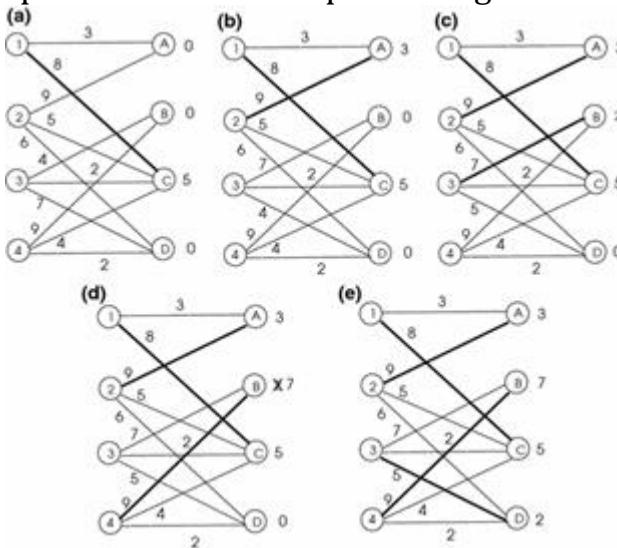


Fig. 9.20 Un ejemplo de gráfico bipartito ponderado para probar el algoritmo de subasta. Tenemos la misma coincidencia máxima que en la figura 9.19

Recientemente, se mostró en [21] que la complejidad de tiempo esperada del algoritmo de subasta para gráficos bipartitos aleatorios donde cada borde se selecciona independientemente con probabilidad $p \geq \frac{c \log n}{n}$ con $c > 1$ es $O(\frac{n \log^2 n}{\log np})$. También en este estudio, la complejidad del tiempo esperado de este algoritmo en un sistema paralelo de memoria compartida con $O(\log n)$ se muestra que los procesadores son $O(n \log n)$ (Fig. 9.20).

Algoritmos de subasta paralela

Las fases de licitación y asignación del algoritmo de subasta. `areMatching!` `weighted!` `Auction` algoritmo basado en subasta ! paralelo tanto conveniente para procesamiento paralelo. Cuando cada uno de estos pasos se realiza simultáneamente, se forma un algoritmo paralelo síncrono. Además, un comprador puede hacer ofertas en momentos arbitrarios en operaciones asíncronas. El modo de operación asíncrono en paralelo junto con el procedimiento síncrono de memoria compartida se describen y analizan en [5]. La parallelización mediante la distribución de los vértices del gráfico bipartito ponderado a un conjunto de procesos paralelos se presenta en [24]. Cada proceso realiza la fase de licitación del algoritmo para compradores libres en su conjunto y luego estas ofertas se intercambian con otros procesos para determinar la oferta global más grande. Por lo tanto, el proceso de licitación se realiza de forma independiente en cada proceso con sincronización al final. La implementación se realizó en una computadora de memoria distribuida utilizando MPI. Otras implementaciones del algoritmo de subasta paralela se realizaron en computadoras con memoria distribuida en [6 , 23].

9.6 Emparejamiento ponderado en gráficas generales

Como un primer enfoque para diseñar un algoritmo de aproximación para encontrar la coincidencia de peso máximo en un gráfico ponderado, podemos usar la misma estrategia que en el caso del gráfico no ponderado. Esta vez, necesitamos cambiar la línea 6 del algoritmo 9.5 para seleccionar el borde de mayor peso global. Este algoritmo requiere la

clasificación de los bordes con respecto a sus pesos y, por lo tanto, el tiempo dominante tomado es este paso que resulta en una complejidad de tiempo de $O(m \log m)$. Este método da como resultado el mismo MWM para el gráfico de muestra en la Fig. 9.4. Este algoritmo tiene un factor de aproximación de $1/2$ [22].

9.6.1 Algoritmo de Preis

Preis creó un algoritmo de coincidencia ponderada codicioso que tiene un mejor rendimiento que el algoritmo codicioso global [22]. La idea de este algoritmo es seleccionar los bordes localmente más pesados en lugar de un peso máximo global. Un borde localmente más pesado es el borde con mayor peso entre todos sus bordes adyacentes. La selección de un borde localmente más pesado se realiza arbitrariamente eligiendo un borde (u, v) , pero si se encuentra un borde adyacente que tiene un peso mayor, entonces se selecciona ese borde. El funcionamiento de este algoritmo se describe en el algoritmo 9.11. Podemos ver que las operaciones locales son independientes y, por esta razón, este enfoque es adecuado para la comparación distribuida y también paralela.

Algorithm 9.11 Preis_MWM

```

1: Input :  $G = (V, E)$ 
2: Output : MM  $M$  of  $G$ 
3:  $M \leftarrow \emptyset$ 
4:  $E' \leftarrow E$ 
5: while  $E' \neq \emptyset$  do
6:   select some locally heaviest weight edge  $e \in E'$ 
7:    $M \leftarrow M \cup \{e\}$ 
8:    $E' \leftarrow E' \setminus \{e \text{ and its adjacent edges}\}$ 
9: end while
```

Las iteraciones de este algoritmo se ilustran en la figura 9.21. La complejidad del tiempo de este algoritmo es $O(m \log n)$ con una relación aproximada de 2 [22].

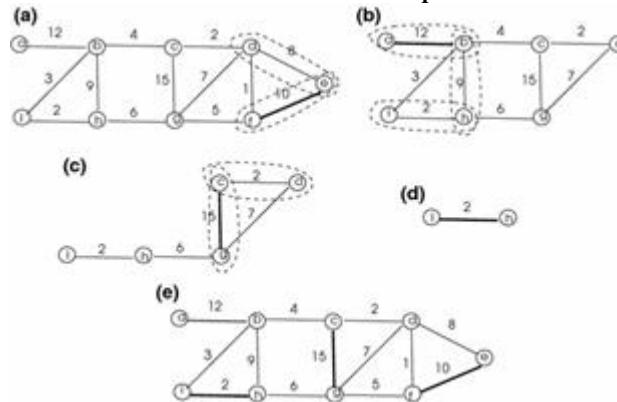


Fig. 9.21 La operación del algoritmo de Preis en una gráfica de muestra. El primer borde seleccionado es (d, e) pero un borde adyacente (e, f) tiene un peso mayor, por lo que (e, f) se comprueba y se considera que es más pesado a nivel local y se incluye en la M correspondiente en a. Todos los bordes adyacentes a (e, f) se eliminan del gráfico para obtener el subgrafo en b. Esta vez los bordes (i, h) , luego (h, b) y luego (b, a) se seleccionan en secuencia para encontrar el borde más grueso localmente (b, a) que se agrega a M en b. Los terceros selecciona de iteración (c, d) y (c, g) , a su vez para añadir (c, g) a M . El último borde para agregar es (i, h) como se muestra en d, y la coincidencia final con un peso total de 39 se muestra en e

9.6.2 Algoritmo de emparejamiento distribuido de Hopeman

En una configuración distribuida, nuestro objetivo es tener una coincidencia máxima donde los nodos de una red de computadoras participan activamente en el proceso de coincidencia. Hopeman modificó el algoritmo secuencial de Preis para que los nodos cooperen para encontrar el borde de mayor peso a nivel local [13]. La idea principal es que si los nodos u y v en los extremos de un borde (u, v) deciden que (u, v) es el borde más pesado que incide en ambos, no puede haber un borde más grueso adyacente a este

borde y, por lo tanto, se puede incluir en el MWM. Hay dos tipos de mensajes, solicitud y eliminación ; un nodo que encuentra (u, v) es el incidente de borde más pesado al que envía la solicitud al nodo vecino v . Si este nodo encuentra que (u, v) es el borde de mayor peso incidental para él, responde por un mensaje de solicitud y (u, v) se incluye en el MWM como se muestra en el algoritmo 9.12.

Algorithm 9.12 Hoepman-MWM

```

1: set of int  $R, S$ 
2: message types req, drop
3:  $R \leftarrow \emptyset$ 
4:  $N \leftarrow N(i)$ 
5:  $c \leftarrow candidate(i, S)$ 
6: if  $c \neq \perp$  then
7:   send request to  $c$ 
8: end if
9: while  $S \neq \emptyset$  do
10:   receive msg( $j$ )
11:   case msg( $j$ ).type of
12:     req:    $R \leftarrow R \cup \{u\}$ 
13:     drop:  $S \leftarrow S \setminus \{u\}$ 
14:       if  $u = c$  then  $c \leftarrow candidate(i, S)$ 
15:         if  $c \neq \perp$  then send req to  $c$ 
16:       if  $c \neq \perp \wedge c \in R$  then
17:         for all  $w \in N \setminus \{c\}$  do
18:           send drop to  $w$ 
19:         end for
20:        $S \leftarrow \emptyset$ 
21:     end if
22:   end while

```

Análisis

Un borde de la gráfica de la red puede ser atravesada por un máximo de dos mensajes, ya sea por req partir de dos nodos en sus puntos finales o un req y una gota mensaje. Por lo tanto, el número total de mensajes intercambiados será de $2 m$. Dado que este algoritmo imita el algoritmo Preisssecuencial , la salida es la misma coincidencia que se produce, es la misma que la del algoritmo de coincidencia más pesado con la misma relación de aproximación de $1/2$ [13].

9.6.3 Métodos de algoritmos paralelos

En la búsqueda de un algoritmo paralelo para el problema de coincidencia, podemos dividir el gráfico y distribuir los vértices a los procesadores. Cada proceso luego realiza lo siguiente para su partición del gráfico.

mientras que hay bordes a la izquierda

- 1.
2. Se establece un puntero desde cada vértice para que apunte al vértice más grueso vecino.
3. si dos vértices u y v apuntan entre sí, el borde (u, v) se incluye en la coincidencia.

Tenemos que tener cuidado al considerar los vértices de borde en las particiones. Esto se puede manejar mediante la introducción de vértices fantasma que son los vértices no miembros que están conectados a los vértices de borde de una partición. En este caso, cuando un vértice de borde coincide en una partición i , el proceso p_i responsable de la partición i debería informar a los procesos p_j que sostienen los vértices fantasma que son vecinos de v de la coincidencia.

Paralelizando el algoritmo de Hoepman

Manne et al. describió la similitud entre el algoritmo de Hoepman y el algoritmo paralelo de Luby para construir un conjunto independiente de una gráfica que

describiremos en el Cap. 10 . Primero se desarrolla una versión secuencial del algoritmo de Hoepman que, de hecho, es similar al algoritmo de Preis usando la notación del algoritmo 9.12. Este algoritmo busca los bordes dominantes en el gráfico. Luego se forma una versión paralela de este algoritmo que asigna un bloque de vértices a cada proceso.¹⁸ del sistema de procesamiento paralelo. El gráfico en cuestión se divide al replicar los vértices de los bordes y cada proceso ejecuta el algoritmo secuencial en su partición para dar como resultado una coincidencia máxima global. Los autores muestran que el algoritmo paralelo diseñado es eficiente hasta 32 procesadores [18].

9.7 Notas de capítulo

Revisamos problemas de emparejamiento y problemas relacionados en este capítulo. Encontrar la coincidencia máxima de un gráfico general ponderado o no ponderado se puede realizar en tiempo polinomial, como hemos visto, de hecho, este problema es uno de los pocos problemas relacionados con los gráficos que se pueden realizar en tiempo polinomial. Sin embargo, hay varios algoritmos de aproximación para reducir el tiempo lineal. En [11] se demostró que cualquier coincidencia codiciosa no ponderada es una aproximación de $1/2$ a la coincidencia máxima. Además, cualquier coincidencia ponderada codiciosa que seleccione bordes legales con pesos máximos tiene la aproximación de $1/2$ al peso máximo coincidente [1]. El algoritmo secuencial que hemos revisado en este capítulo se muestra en la Tabla 9.1 . El algoritmo codicioso y el algoritmo debido a Los Preis son algoritmos de aproximación y todos los demás algoritmos enumerados son exactos. Edmonds proporcionó el primer algoritmo de tiempo polinomial para la comparación ponderada con $\mathcal{O}(n^2m)$ complejidad del tiempo [7].

Tabla 9.1 Algoritmos de concordancia secuencial

	Gráficos bipartitos	Graficas generales
Emparejamiento no ponderado	Algoritmo de ruta de aumento: $\mathcal{O}(nm)$ Algoritmo Hopcroft – Karp: $\mathcal{O}(n^2)$	Algoritmo codicioso: $\mathcal{O}(nm)$ El algoritmo de Edmond: $\mathcal{O}(n^2m)$
	Emparejamiento ponderado	Algoritmo húngaro: $\mathcal{O}(n^3)$ Algoritmo basado en el flujo $\mathcal{O}(m^2 + mn)$
		Algoritmo de Preis : $\mathcal{O}(nm)$

Cabe señalar que hay varias mejoras a estos algoritmos básicos. Por ejemplo, Gabow mejoró el tiempo de ejecución para la comparación ponderada a $\mathcal{O}(nm + n^2 \log n)$ [10]. Hay varios algoritmos paralelos para gráficos no ponderados o ponderados, generales o bipartitos. Vimos cómo el algoritmo Hopcroft-Karp se puede paralelizar convenientemente debido al procesamiento BFS y DFS desunido en cada fase. El algoritmo de subasta para la coincidencia de gráficos bipartitos ponderados puede modificarse para tener un procesamiento paralelo en las fases de licitación y asignación. Además, podemos dividir un gráfico general y distribuir particiones a procesos paralelos que realizan la comparación y los resultados se pueden recopilar en un proceso raíz que los fusiona para encontrar la coincidencia global. Una encuesta reciente de algoritmos paralelos para la máxima coincidencia en gráficos bipartitos se proporciona en [2].

En muchos casos, los algoritmos de coincidencia de aproximación resultan ser más rápidos a costa de devolver una solución aproximada en lugar de una exacta. Para gráficos muy grandes, pueden ser preferibles ya que los tiempos involucrados pueden ser muy altos. También hemos descrito cómo una serie de conversiones de un tipo de algoritmo puede llevar a soluciones eficientes. El algoritmo que ordena los bordes y luego incluye los bordes legales para hacer coincidir tiene $O(m \log m)$. La complejidad se debe al proceso de clasificación y tiene una relación de aproximación de $1/2$. A Preis se le ocurrió la idea de seleccionar bordes más pesados locales que sean independientes entre sí para dar como resultado una mejor complejidad de tiempo de $O(m)$. Más tarde, Hoepman proporcionó una versión distribuida de este algoritmo con la misma relación de aproximación que analizamos. Finalmente Manne et al. presentó un algoritmo de coincidencia de aproximación paralela basado en Hopeman trabajo. Podemos ver la secuencia de desarrollo aquí son un algoritmo secuencial; un algoritmo secuencial mejorado; un algoritmo distribuido del algoritmo secuencial mejorado y un algoritmo paralelo de memoria distribuida que se basa en el algoritmo distribuido. Este camino, aunque en menos pasos, se sigue comúnmente en varios problemas de gráficos, como vimos.

Los algoritmos de coincidencia distribuida deben considerarse cuidadosamente, ya que la coincidencia de un incidente de borde con un nodo en una red requiere notificación a los vecinos de dos saltos, ya que se verán afectados. Matching tiene numerosas aplicaciones y, por lo tanto, se necesitan algoritmos paralelos y distribuidos con mejores desempeños.

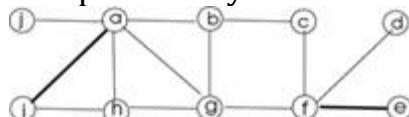


Fig. 9.22 Gráfico de muestra para el Ejercicio 1

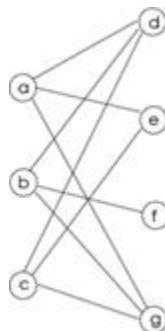


Fig. 9.23 Gráfico de muestra para el Ejercicio 2

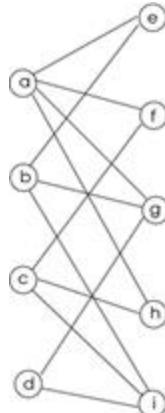


Fig. 9.24 Gráfico de muestra para los ejercicios 3 y 4

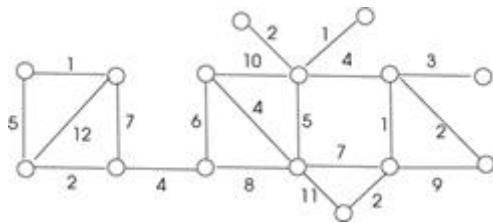


Fig. 9.25 Gráfico de muestra para el ejercicio 6

Ceremonias

Dada la gráfica de la figura 9.22 con la coincidencia inicial que se muestra en negrita, encuentre iterativamente las rutas de aumento para obtener una coincidencia máxima para esta gráfica.

- 1.
2. Calcule la coincidencia máxima en el gráfico bipartito de la figura 9.23 utilizando el algoritmo de ruta de aumento.
3. Encuentre la coincidencia máxima en el gráfico bipartito de la figura 9.24 utilizando el algoritmo Hopcroft-Karp que muestra los árboles BFS construidos en todas las iteraciones.
4. Determine la coincidencia máxima en el gráfico de la figura 9.24 esta vez utilizando el método de flujo máximo del algoritmo Ford-Fulkerson.
5. Un sistema multiprocesador tiene 5 computadoras P_1, \dots, P_5 que debe terminar 5 tareas $1, \dots, 5$. El tiempo para finalizar las tareas para cada procesador se indica en la siguiente matriz de costos. Calcule el tiempo mínimo para terminar todas las tareas con estos 5 procesadores usando el algoritmo húngaro. Mostrar cada paso del algoritmo.

$$C = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ P_1 & 8 & 1 & 4 & 3 & 2 \\ P_2 & 2 & 5 & 9 & 6 & 4 \\ P_3 & 6 & 2 & 3 & 4 & 5 \\ P_4 & 1 & 4 & 7 & 9 & 3 \\ P_5 & 5 & 0 & 8 & 1 & 2 \end{pmatrix}$$

(9.10)

6. Diseñe un algoritmo distribuido de concordancia ponderada sincrónica que encuentre la concordancia ponderada mínima en una red de computadoras. Un nodo v que tenga el menor grado entre sus vecinos tiene el privilegio de proponerle a su vecino u si (u, v) es el borde de menor peso que incide sobre él. Calcule el tiempo y las complejidades de los mensajes de este algoritmo y muestre su funcionamiento en la red que se muestra en la figura 9.25.
7. Proporcione una extensión al pseudocódigo del algoritmo de coincidencia máxima basado en la coloración de bordes (Algoritmo 9.7) para que cuando no haya más nodos que estén en el estado DESCONECTADO, el algoritmo en un nodo se detenga sin esperar a que finalicen k vueltas. La raíz también está informada de esta condición.

Referencias

1. Avis D (1983) Una encuesta de heurísticas para el problema de emparejamiento ponderado. Redes 13: 475–493
[MathSciNet Crossref](#)
2. Azad A, Halappanavar M, Rajamanickam S, Boman EG, Khan AM, Pothen A (2012) Algoritmos multiproceso para la coincidencia máxima en gráficos bipartitos. IPDPS 2012: 860–872
3. Berge C (1957) Dos teoremas en la teoría de grafos. Proc Natl Acad Sci USA 43: 842–844
[MathSciNet Crossref](#)

4. Bertsekas DP, Tsitsiklis JN (1989) Computación paralela y distribuida: métodos numéricos. Prentice-Hall, Englewood Cliffs
 5. Bertsekas DP, Castanon DA (1991) Implementaciones paralelas síncronas y asíncronas del algoritmo de subasta. *Comput paralelo* 15: 707–732
 6. Bus L, Tvardik P (2009) Hacia algoritmos de subasta para problemas de grandes asignaciones densas. *Comput Optim Appl* 43 (3): 411–436
[MathSciNet Crossref](#)
 7. Edmonds J (1965) Caminos, árboles y flores. *Can J Math* 17: 449–467
 8. Erciyes K (2015) Algoritmos distribuidos y secuenciales para bioinformática. Springer serie de biología computacional. Springer, Cham
[Referencia cruzada](#)
 9. Erciyes K (2015) Algoritmos de grafos distribuidos para redes informáticas. Springer serie de informática y comunicaciones. Springer, Londres
 10. Gabow HN (1976) Una implementación eficiente del algoritmo de Edmonds para la máxima coincidencia en los gráficos. *J Assoc Comput Mach* 23: 221–234
[MathSciNet Crossref](#)
 11. Hausmann D, Korte B (1978) K-codiciosos algoritmos para sistemas de independencia. *Z Oper Res* 22 (1): 219–228
 12. Hirvonen J, Suomela J (2012) Máxima coincidencia distribuida: la codicia es óptima. En: Kowalski D, Panconesi A (eds) PODC12. Actas del simposio ACM de 2012 sobre los principios de la computación distribuida, Madeira, Portugal, 161–8 de julio de 2012
 13. Hoepman JH (2004) Combinaciones ponderadas distribuidas simples. Informe técnico, Instituto Nijmegen de informática y ciencias de la información (NIII)
 14. Hopcraft J, Karp RM (1973) An $\mathcal{O}(n^3)$ Algoritmo para máxima coincidencia en gráficos bipartitos. *SIAM J Comput* 2: 225–231
[MathSciNet Crossref](#)
 15. Karypis G, Kumar V (1998) Un algoritmo paralelo para la partición de gráficos multinivel y el ordenamiento de matrices dispersas. *J Parallel Distrib Comput* 48 (1): 71–95
[Referencia cruzada](#)
- dieciséis. König D (1931) Graphen und matrizen . Mates. *Lapok* 38: 116–119
[MATES](#)
17. Kuhn HW (1955) El método húngaro para el problema de asignación. *Nav Res Logist Q* 2: 83–97
[MathSciNet Crossref](#)
 18. Manne F, Bisseling RH (2007) Un algoritmo de aproximación paralelo para el problema de emparejamiento máximo ponderado. En: Wyrzykowski R, Karczewski K, Dongarra J, Wasniewski J (eds) Actas de la séptima conferencia internacional sobre procesamiento paralelo y matemáticas aplicadas (PPAM 2007). LNCS, vol. 4967. Springer, Berlín, pp. 708–717
 19. Micali S, Vazirani V (1980) An $O(\sqrt{|V|} \cdot |m|)$ algoritmo para encontrar la máxima coincidencia en gráficos generales. En: Actas del 21 simposio anual sobre fundamentos de la informática, IEEE, pp 17–27.
 20. Munkres J (1957) Algoritmos para la asignación y problemas de transporte. *J Soc Ind Appl Math* 5 (1): 32–38
 21. Naparstek O, Leshem A (2014) La complejidad del tiempo esperado del algoritmo de subasta y el algoritmo de cambio de marca para la máxima coincidencia bipartita en gráficos aleatorios. *Aleatorios Struct Algoritmos* 48: 384-395

[Referencia cruzada](#)

22. Preis R (1999) Tiempo lineal 1/2 algoritmo de aproximación para la máxima coincidencia ponderada en gráficos generales. En: Meinel C, Tison S (eds) Simposio sobre aspectos teóricos de la informática (STACS) 1999. LNCS, vol. 1563, Springer, Berlín, 259–269
23. Riedy J (2010) Hacer que el giro estático sea escalable y confiable. Doctor en Filosofía. Tesis, Departamento de EECS, Universidad de California, Berkeley
24. Sathe M (2012) Algoritmos de gráficos paralelos para encontrar emparejamientos ponderados y subgrafos en ciencias computacionales. Doctor en Filosofía. Tesis, Universidad de Basilea.
25. Vazirani VV (1994) Una teoría de alternar caminos y flores para probar la corrección de la $O(\sqrt{V}E)$ Gráfica general de algoritmo de máxima coincidencia. Combinatoria 14 (1): 71-109
[MathSciNet](#) [Crossref](#)

10. Independencia, dominación y cobertura de vértices.

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Los subgrafos de una gráfica pueden tener algunas propiedades especiales y la detección de estos subgrafos puede ser útil para varias aplicaciones. En este capítulo, estudiamos la teoría y los algoritmos secuenciales, paralelos y distribuidos para tres subgrafos especiales: conjuntos independientes, conjuntos dominantes y cobertura de vértices.

10.1 Introducción

La detección de subgrafos de una gráfica con propiedades especiales puede ser útil en varias implementaciones. En este capítulo, estudiamos la detección de tres subgrafos especiales: conjuntos independientes , conjuntos dominantes y cobertura de vértices . Veremos que todos estos problemas son equivalentes en términos de complejidad y encontrar una solución a uno de ellos produce la solución a los otros.

Un conjunto independiente de una gráfica es un subconjunto de sus vértices, de manera que no hay dos vértices en este conjunto que sean vecinos. Encontrar un conjunto máximo independiente de un gráfico G, que es el conjunto independiente de máximo orden entre todos los conjuntos independientes de G es NP-duro. Un conjunto independiente de una gráfica G es una camarilla en el complemento de G. Un conjunto dominante de una gráfica es un subconjunto de vértices de tal manera que cada vértice de la gráfica está en este conjunto o un vecino a un vértice en este conjunto. Encontrar un conjunto dominante mínimo de una gráfica es nuevamente NP-duro. El último problema que estudiamos en este capítulo es la cubierta de vértices de un gráfico que consta de vértices de manera que cada borde del gráfico incide en al menos un vértice en este conjunto. Los conjuntos independientes, los conjuntos dominantes y las cubiertas de vértices se pueden usar para varias aplicaciones de redes complejas de la vida real, como la detección de clústeres en redes biológicas y el enrutamiento en redes de computadoras. Estos subgrafos especiales también pueden servir como bloques de construcción de algoritmos de gráficos más complejos. En este capítulo, investigamos las propiedades teóricas de estos conjuntos de vértices en gráficos, mostramos cómo se relacionan y estudiamos de forma secuencial, paralela,

10.2 Conjuntos independientes

Un conjunto independiente de una gráfica es un subconjunto de sus vértices, de manera que ningún vértice en este conjunto es adyacente a ningún otro vértice contenido en este conjunto. Podemos definir formalmente el conjunto independiente de la siguiente manera.

Definición 10.1 (conjunto independiente) Un conjunto independiente de una gráfica $G = (V, E)$ Es un subconjunto I de sus vértices tal que $\forall u \in I \quad \forall v \in I \quad (u, v) \notin E$.

Un conjunto independiente es máxima si no se puede ampliar aún más por vértices adicionales, es decir, un conjunto independiente máximo (MIS) no está contenida adecuadamente en cualquier otro conjunto independiente de G . Un conjunto máximo independiente (MaxIS) de un gráfico G es el conjunto G de mayor orden independiente entre todos sus conjuntos independientes. El número de vértices en un MaxIS se llama el número de independencia, $\alpha(G)$, de la gráfica G . La figura 10.1 muestra los conjuntos máximos y máximos independientes de un gráfico de muestra. Encontrar un MaxIS de un gráfico es NP-duro y la versión de decisión de este problema que determina si un gráfico tiene un conjunto independiente máximo de orden k o más es NP-completo [2]. Sin embargo, podemos encontrar el MIS de un gráfico en tiempo lineal como veremos. Cuando cada vértice en el gráfico tiene un peso asociado, nuestro objetivo es encontrar el conjunto independiente con el peso total máximo (MaxWIS). Este problema es nuevamente NP-duro y también es muy difícil de aproximar.

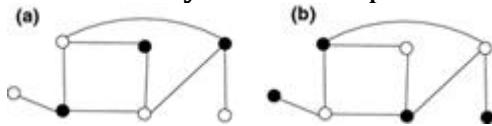


Fig. 10.1 a Un MIS con orden 3. b A MaxIS con orden 4 de una gráfica de muestra. Los vértices en los conjuntos independientes se muestran en negrita

10.2.1 Reducción a la camarilla

Podemos reducir el problema de encontrar un conjunto independiente en un gráfico para encontrar una camarilla. Una camarilla de un gráfico, $G = (V, E)$, es un subconjunto C de sus vértices de tal manera que cada vértice en C está conectado a todos los otros vértices en C . En un conjunto independiente I de G , que no es necesariamente máximo, no hay dos vértices adyacentes. Por lo tanto, cuando tomamos el complemento de una gráfica para obtener la gráfica, \bar{G} , el conjunto I será un clique desde el complemento exhibirá todas las bolas en G . La Figura 10.2 muestra un gráfico con un conjunto independiente y el complemento de este gráfico tiene el conjunto independiente como una camarilla.

Esta dualidad muestra que el problema de la camarilla es al menos tan difícil como el problema del conjunto independiente y también que el problema de la serie independiente es al menos tan difícil como el problema de la camarilla, lo que significa que pueden reducirse entre sí en el tiempo polinomial que se indica a continuación:

Independent Set Problem (IND) $\leq_{\text{pr}}^{} \text{Clique Problem (CLIQUE)}$

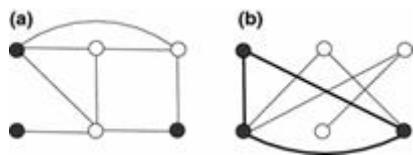


Fig. 10.2 a Un MIS con orden 3 en una gráfica de muestra. b Los mismos vértices forman una camarilla en el complemento del gráfico de muestra

10.2.2 Algoritmos secuenciales

Revisaremos cuatro algoritmos secuenciales para encontrar el MIS de un gráfico, comenzando con uno aleatorio codicioso. El segundo algoritmo usa una heurística y el tercero considera las etiquetas de los vértices al seleccionar miembros de MIS, mientras que el cuarto algoritmo es un método general que busca un conjunto independiente en cada paso.

Algorithm 10.1 Seq_MIS1

```

1: Input :  $G = (V, E)$  an undirected unweighted graph
2: Output : MIS  $I$  of  $G$ 
3:  $I \leftarrow \emptyset$ 
4:  $V' \leftarrow V$ 
5: while  $V' \neq \emptyset$  do
6:   select any  $v \in V'$ 
7:    $I \leftarrow I \cup \{v\}$ 
8:    $V' \leftarrow V' \setminus \{v\} \cup N(v)$ 
9: end while

```

10.2.2.1 El algoritmo codicioso aleatorio

Como primer intento, adoptaremos una estrategia codiciosa para formar el MIS. Intuitivamente, podemos elegir arbitrariamente un vértice v del gráfico G , incluir este vértice en el MIS y eliminar v y todos sus vecinos $N(v)$ junto con los bordes incidentes en estos vértices del gráfico, simplemente porque sus vecinos no pueden ser incluidos en el MIS. Procedemos de esta manera hasta que no queden más vértices. El algoritmo 10.1 muestra el código para este procedimiento.

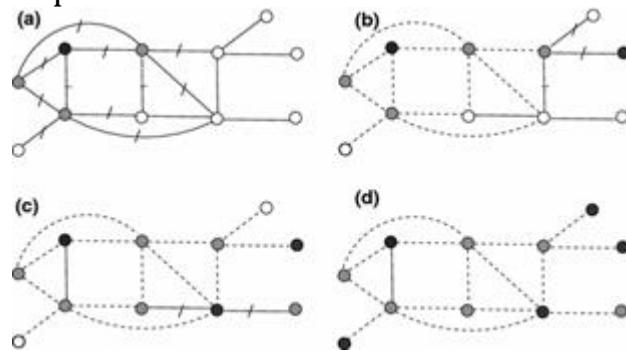


Fig. 10.3 Ejecución de Seq_MIS1 en un gráfico de muestra. Las tres primeras iteraciones se muestran en a - c ; y las dos últimas iteraciones se muestran en d . Los vértices establecidos independientes se muestran en negrita y los vértices adyacentes eliminados en gris con los bordes eliminados marcados con líneas discontinuas

El conjunto de salida I es un conjunto independiente, ya que nunca incluimos ningún vecino del vértice seleccionado v en el gráfico para obedecer la propiedad IS, es decir, no hay dos vértices en este conjunto deben ser adyacentes. También es máxima ya que se procede hasta que no haya dejado vértices y por lo tanto no se puede ampliar lo más lejos. Este algoritmo requiere pasos $O(n)$, ya que podemos terminar seleccionando un

vértice y su vecino único repetidamente, como en el caso de una red lineal. La ejecución de este algoritmo en un gráfico de muestra se muestra en la Fig. 10.3.

10.2.2.2 El primer algoritmo de grado más bajo

Como otro enfoque para formar el MIS de una gráfica, podemos seleccionar el vértice con el grado más bajo en la línea 5 del algoritmo 10.1 en lugar de un vértice aleatorio. Llamamos a este algoritmo el algoritmo de primer grado más bajo (LDFA). Esta heurística es razonable ya que nuestro objetivo es tener un conjunto independiente lo más grande posible, lo que significa que queremos eliminar la menor cantidad posible de vecinos de un vértice seleccionado. Por lo tanto, siempre seleccionamos vértices con el menor número de vecinos. La ejecución de este algoritmo se muestra en la Fig. 10.4. Tenga en cuenta que tuvimos ocho pasos en lugar de cuatro pasos del algoritmo codicioso aleatorio, pero obtuvimos un MIS de tamaño 6 que es el máximo para este gráfico.

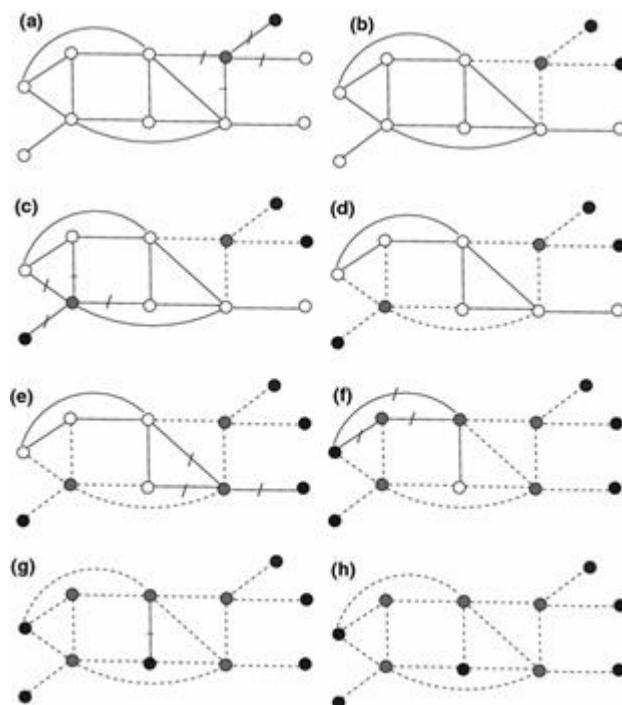


Fig. 10.4 Ejecución de LDFA en el mismo gráfico de la Fig. 10.3. El MIS final se muestra en h

El teorema 10.1 LDFA proporciona un MIS I tal que $|I| \geq n/(\Delta(G) + 1)$ donde $\Delta(G)$ Es el grado máximo del gráfico.

Prueba Un vértice u está incluido en $V \setminus I$ cuando un vértice $v \in N(u)$ se selecciona para estar en el conjunto I . Si marcamos un vértice u cada vez que se selecciona uno de sus vecinos para que esté en I , se puede marcar como máximo $\Delta(G)$ hora. Por lo tanto, $|V \setminus I| \leq \Delta(G)|I|$. Además, un vértice puede estar en el MIS o un vecino de un vértice en el MIS. Por lo tanto, $|I| + |V \setminus I| = n$. Por lo tanto, $(\Delta(G) + 1)|I| \geq n$ lo que significa $|I| \geq n/(\Delta(G) + 1)$. \square

Corolario 10.1 LDFA es una $1/(\Delta(G) + 1)$ Algoritmo de aproximación para el máximo conjunto independiente de problemas.

10.2.2.3 El primer algoritmo lexicográfico

De una manera ligeramente diferente, pero utilizando nuevamente el enfoque codicioso, podemos etiquetar n vértices de una gráfica con el orden $1, \dots, n$ y encuentre el MIS en secuencia utilizando estas etiquetas como se muestra en el algoritmo 10.2. En este caso, sabemos qué vértice seleccionar en cada iteración y este algoritmo se denomina algoritmo Lexicographically First MIS (LFA). Sin embargo, este algoritmo no mejora el tiempo de ejecución del anterior, ya que tiene un enfoque codicioso similar al primero y tiene una complejidad de tiempo de $\mathcal{O}(n + m)$. Tiempo desde que comprobamos vecinos de cada vértice.

Algorithm 10.2 Seq_MIS₃

```

1: Input :  $G = (V, E)$  an undirected unweighted graph
2: Output : MIS  $I$  of  $G$ 
3:  $I \leftarrow \emptyset$ 
4: for  $i = 1$  to  $n$  do
5:   if  $I \cap N(v_i) = \emptyset$  then
6:      $I \leftarrow I \cup \{v_i\}$ 
7:   end if
8: end for

```

10.2.2.4 El algoritmo incremental

Otro enfoque para encontrar MIS es para agrandar gradualmente el actual conjunto independiente que los nuevos conjuntos independientes formados a partir de los vértices no en el conjunto I . En lugar de seleccionar un solo vértice en la línea 4 del algoritmo 10.1, seleccionamos un conjunto independiente de la gráfica G este momento. El conjunto independiente seleccionado, I' , del gráfico de entrada G se agrega luego al MIS I y todos los vértices de I' , sus vecinos con sus bordes incidentes se eliminan de la gráfica ya que estos vecinos no pueden considerarse más. El es I' no necesita ser un MIS de G . El algoritmo termina cuando ya no quedan más vértices para considerar, como se muestra en el algoritmo 10.3.

Algorithm 10.3 Seq_MIS₄

```

1: Input :  $G = (V, E)$  an undirected unweighted graph
2: Output : MIS  $I$  of  $G$ 
3:  $I \leftarrow \emptyset$ 
4:  $G' \leftarrow G$ 
5: while  $G' \neq \emptyset$  do
6:   select any independent set  $I'$  of  $G'$ 
7:    $I \leftarrow I \cup \{I'\}$ 
8:    $G' \leftarrow G' - I'$ 
9: end while

```

Claramente, la elección del conjunto independiente en la línea 6 determina el rendimiento de este algoritmo. Veremos que la aleatorización en esta selección proporciona algoritmos con buen rendimiento (Fig. 10.5).

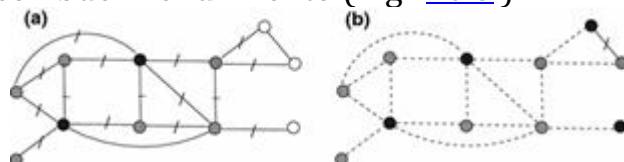


Fig. 10.5 Encontrar MIS de un gráfico de muestra seleccionando un IS en cada paso. Los vértices establecidos independientes se muestran en negrita y los vértices adyacentes eliminados en gris con los bordes eliminados marcados con líneas discontinuas

10.2.2.5 Construcción MIS usando coloración de vértices

Una coloración de vértice de un gráfico es asignar colores a sus vértices en forma de enteros $1, \dots, k$ de modo que no haya dos vértices adyacentes que reciban el mismo color que veremos en el siguiente capítulo. Podemos hacer uso de la coloración de vértices para encontrar el MIS de un gráfico.

Una clase de color $V' \in V$ en una gráfica de color vértice $G = (V, E)$ Es un conjunto de vértices de G que tienen el mismo color. Como no hay dos vértices adyacentes de un gráfico que tienen el mismo color, cada clase de color es un conjunto independiente de G . Podemos explotar esta propiedad para encontrar el MIS de un gráfico G si primero incluimos todos los vértices del color 1 en el MIS y luego verificamos iterativamente los vértices de colores crecientes para incluirlos en el MIS o no, como se muestra en el algoritmo 10.4. Si un vértice u tiene un vecino v que ya está en el MIS, no podemos incluir el vértice u en el MIS. Supondremos que el gráfico está coloreado con k colores antes de ejecutar este algoritmo.

Algorithm 10.4 Seq_MIS

```

1: Input :  $G = (V, E)$  an undirected unweighted graph
2: Output : MIS I of G
3:  $I \leftarrow$  vertices of color 1
4: for  $i = 2$  to  $k$  do
5:    $I \leftarrow I \cup$  all allowed vertices of color  $k$ 
6: end for
```

La ejecución de este algoritmo en un gráfico de muestra se muestra en la Fig. 10.6 para un gráfico de muestra coloreado con cuatro colores. La complejidad del tiempo de este algoritmo es $O(km)$ ya que necesitamos ejecutar el bucle for $O(k)$ veces y es posible que debamos verificar todos los bordes en cada ejecución. También necesitamos el tiempo C para colorear los vértices del gráfico G y, por lo tanto, el tiempo total es $C + O(km)$.

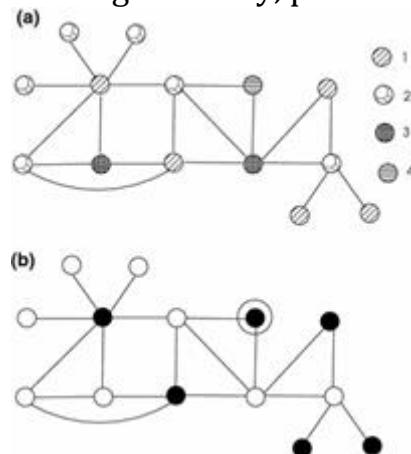


Fig. 10.6 Ejecución del algoritmo MIS basado en color de vértice en un gráfico de muestra que está coloreado con cuatro colores. El MIS mostrado por vértices en negrita está casi formado al incluir todos los vértices del color 1 en el MIS en el primer paso. El único vértice agregado al MIS después del primer paso cuando $k=4$ se muestra dentro de un círculo grande

Algorithm 10.5 Luby_MIS

```

1: Input :  $G = (V, E)$ 
2: Output : MIS I of G
3:  $I \leftarrow \emptyset$ 
4:  $G' = (V', E') \leftarrow G = (V, E)$ 
5: while  $V' \neq \emptyset$  do in parallel
6:   choose a random set of vertices  $V' \in V(G')$  by choosing each vertex with probability
     $1/(2d(v))$ 
7:   for all  $(u, v) \in E(G')$  do in parallel
8:     if  $(u \in V' \wedge v \in V')$  then
9:       remove the endpoint with lower degree to obtain  $S$  from  $V'$ 
10:      end if
11:    end for
12:     $I \leftarrow I \cup V'$ 
13:     $G' \leftarrow G' \setminus (S \cup N(S))$ 
14: end while
```

10.2.3 Algoritmo MIS paralelo de Luby

Podemos implementar la selección de un conjunto independiente en el algoritmo 10.3 utilizando varios enfoques. Luby, en 1986, propuso un algoritmo de Monte Carlo paralelo

aleatorio eficiente para encontrar MIS de una gráfica que se desarrolla de la siguiente manera [7]. Cada vértice vestá marcado con probabilidad $1 / (2 d(v))$ en paralelo, donde $d(v)$ es el grado de v , que se incluirá en el conjunto independiente o no. Esta marca puede producir bordes con ambos puntos finales marcados para estar en el MIS, ya que la asignación se realiza en paralelo de forma independiente y, por lo tanto, se necesitan correcciones. El siguiente paso identifica dichos bordes y para cada uno de esos bordes (u, v) con u y v cubiertos en el MIS, se selecciona el vértice con el grado más alto y, en el caso de un empate, los identificadores de vértice se usan para seleccionar solo uno de dichos vértices. Los vértices seleccionados y sus vecinos se eliminan del gráfico y este proceso se repite hasta que el gráfico quede vacío como se muestra en el algoritmo 10.5.

Algorithm 10.6 SMLuby_m.MIS

```

1: Input :  $G = (V, E)$ 
2: Output : MIS  $I[n]$  of  $G$ 
3: boolean  $I[n], C[n], R[n]$ 
4: for  $j = ((i - 1) * n/k) + 1$  to  $i * n/k$  do                                t> initialize my partition
5:    $C[i] \leftarrow 1$ 
6:    $I[i] \leftarrow 0$ 
7: end for
8: repeat
9:   generate a random number for each available vertex  $i$  ( $C[i] \neq 0$ ) in my partition w.c.t
    $1/(2d(v))$ 
10:  check the random numbers for neighbors of my vertices in  $R$ 
11:  for any vertex  $i$  in my partition that has highest random number than its neighbors do
12:     $C[i] \leftarrow 0$ 
13:     $I[i] \leftarrow 1$ 
14:    for all  $j \in N(i)$  do
15:       $C[j] \leftarrow 0$ 
16:    end for
17:  end for
18: until  $C[i] = 0, \forall 1 \leq i \leq n$ 

```

La selección de los nodos que se incluirán en el conjunto independiente se puede implementar en una EREW-PRAM utilizando procesadores $O(m)$ con cada ejecución tomando $O(\log n)$ hora. Se puede demostrar que la ejecución de la mientras bucle es $O(\log n)$ veces [7] resultando en un tiempo total de $O(\log^2 n)$ para este algoritmo. Usaremos este algoritmo como la base de un algoritmo distribuido como se describe en la siguiente sección.

Podemos formar una versión paralela de memoria compartida del algoritmo de Luby como se describe en [3]. Tenemos tres vectores de elementos para n vértices; C, I , y R . $I[i]$ muestra si el vértice i está incluido en el MIS o no, $C[i]$ muestra si el vértice i es un candidato para ser incluido con 0, lo que significa que está en MIS o un vecino de un vértice en el MIS y, por lo tanto, no puede ser incluido, y finalmente $R[i]$ contiene el número aleatorio generado para el vértice i en cada iteración. El vector C se inicializa en todos los 1 s, lo que significa que todos los vértices son candidatos y I se inicializa en todos los 0 s, ya que no se determina ningún miembro de MIS. Los vectores se dividen en 1-D entre los procesos k y cada proceso realiza los siguientes pasos en cada iteración hasta que se encuentra MIS, que se determina por todas las entradas del vector C que se convierten en 0 como se muestra en el algoritmo 10.6 para el proceso paralelo i para un número total de k procesos. Este algoritmo requiere sincronización en cada paso; de lo contrario, su rendimiento es similar al algoritmo 10.5 cuando no se considera la sincronización.

10.2.4 Algoritmos distribuidos

Podemos usar varias heurísticas para diseñar algoritmos distribuidos para el problema MIS en una configuración de red. En esta sección describimos tres algoritmos MIS distribuidos; el primer algoritmo usa identificadores de nodos para romper simetrías y el segundo es una versión distribuida del algoritmo paralelo de Luby, siendo el tercero otro

algoritmo distribuido aleatorio con un mejor rendimiento. Si bien los dos últimos algoritmos tienen estructuras similares, mostramos dos formas comunes de implementación, utilizando máquinas de estado finito en la primera y un enfoque más directo en la segunda al mostrar explícitamente los mensajes de control.

Algorithm 10.7 Dist_MIS1

```

1: Input: unweighted undirected graph  $G = (V, E)$ 
2: Output: MIS  $I$  of  $G$ 
3: message types:  $in\_mis, neigh\_mis$ 
4: states: {IDLE, INMIS, NONMIS}                                 $\triangleright$  states of a node
5:  $state \leftarrow IDLE$                                           $\triangleright$  initialize
6: while  $state = IDLE$  do
7:   if  $id > id_s$  of all current neighbors then            $\triangleright$  not in MIS or a neighbor to a MIS vertex
8:      $state \leftarrow INMIS$ 
9:     send  $in\_mis(i)$  to  $N(i)$                                  $\triangleright$  inform neighbors of INMIS state
10:    else if  $in\_mis(j)$  is received from neighbor  $j$  then
11:       $state \leftarrow NONMIS$ 
12:      send  $neigh\_mis(i)$  to  $N(i)$ 
13:    else if  $neigh\_mis(j)$  is received from neighbor  $j$  then
14:       $N(i) \leftarrow N(i) \setminus \{j\}$                                  $\triangleright$  remove neighbor  $j$  from neighbor list
15:    end if
16:  end while

```

10.2.4.1 Algoritmo distribuido codicioso

En nuestro primer intento de encontrar un algoritmo MIS distribuido, modificaremos el algoritmo MIS codicioso (Seq_MIS1), esta vez, las decisiones de unirse al MIS deben tomarse localmente según algunos criterios. Asumiremos que cada nodo tiene un identificador único y el nodo con el identificador más grande entre vecinos se une al MIS como se muestra en el Algoritmo 10.7 ejecutado por un nodo activo i en cada ronda sincrónica. Suponemos que cada nodo conoce inicialmente los identificadores de sus vecinos. Cada nodo de la red puede estar en uno de los siguientes estados.

- IDLE: Este es el estado inicial de un nodo.
- INMIS: Un nodo asignado al MIS ingresa a este estado e informa a los vecinos mediante el mensaje in_mis .
- NONMIS: un nodo en este estado ha dejado de estar en MIS porque uno de sus vecinos se ha convertido en miembro de MIS. Informa a sus vecinos por el mensaje $neigh_mis$.

Un nodo que se incluye en el MIS no debe participar en el algoritmo en otras rondas. Esto se logra cambiando su estado e informando a sus vecinos mediante el mensaje in_mis para que también permanezcan inactivos en otras rondas. Cualquier nodo adyacente a un nodo vecino de un nodo MIS es informado por el vecino por el $neigh_mis$ mensaje para que se elimine de la lista de vecinos activos. La operación de este algoritmo en una red de muestra se muestra en la Fig. 10.7 .

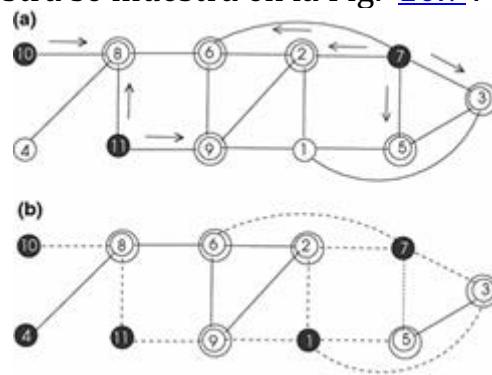


Fig. 10.7 Ejecución de Dist_MIS1 en un gráfico de muestra con identificadores de nodo únicos. Los mensajes in_mis se muestran mediante flechas, los nodos MIS están en negro y los nodos vecinos que ingresan al estado NONMIS se muestran con círculos dobles. En solo dos rondas mostradas en a y b , se forma el MIS

La complejidad del tiempo de este algoritmo es $O(n)$, ya que puede haber una operación puramente secuencial como en una red lineal con identificadores crecientes / decrecientes o una red general con vecinos que tienen identificadores crecientes o decrecientes en una secuencia. El número de mensajes transmitidos es proporcional a m . Por lo tanto, este algoritmo puede resultar lento.

10.2.4.2 El primer algoritmo distribuido aleatorizado

En la búsqueda de un algoritmo para un mejor rendimiento, intentaremos convertir el algoritmo de Luby en un algoritmo distribuido síncrono que llamaremos Dist_Luby que funciona en rondas. Cada ronda se compone de dos fases; en la primera fase, cada nodo activo i se marca con una probabilidad de $1 / 2d(i)$ para estar en el MIS. En este estado MARCADO / NO MARCADO, intercambia el estado con los vecinos para concluir la fase. En la segunda fase, si el nodo i está marcado y hay otro nodo vecino MARCADO, el grado más alto se incluye en el MIS y el otro se elimina de los nodos activos de nuevo intercambiando mensajes de estado. El pseudocódigo para el nodo i se muestra en el algoritmo 10.8. Este algoritmo se completa en el número esperado de $\tilde{O}(\log n)$ rondas [12] y como el número de mensajes es proporcional al número de bordes, el número total de mensajes es $\tilde{O}(m \log n)$.

Algorithm 10.8 Dist_MIS2

```

1: Input: unweighted undirected graph  $G = (V, E)$ 
2: Output: MIS  $I$  of  $G$ 
3: message types: info                                > sent by nodes to inform their states to neighbors
4: states = {IDLE, INMIS, NONMIS}                      > states of a node as idle, in, or neighbor to a node in MIS
5: init_states = {MARKED, UNMARKED}                    > initial states at each round
6:  $I \leftarrow \emptyset$ ;  $state \leftarrow \text{IDLE}$                          > initialize vertex cover  $C$ 
7: while state = IDLE do                                > not in MIS or a neighbor to a MIS vertex
8:   mark init_state MARKED with probability  $1/(2d(i))$           > Phase 1
9:   send info(init_state) to  $N(i)$ 
10:  receive info(init_statej) from  $\forall j \in N(i)$ 
11:  if init_state=MARKED then                                > Phase 2
12:    if  $\exists j \in N(i)$  with statej=MARKED then
13:       $state \leftarrow \text{INMIS}$ 
14:    else
15:       $state \leftarrow \text{NONMIS}$ 
16:    end if
17:    send info(state) to  $N(i)$ 
18:    receive info(statej) from  $\forall j \in N(i)$ 
19:  end if
20:  if  $\exists j \in N(i)$  with statej=INMIS then
21:     $state \leftarrow \text{NONMIS}$ 
22:  end if
23:  if  $\exists j \in N(i)$  with statej=NONMIS then
24:     $state \leftarrow \text{NONMIS}$ 
25:     $N(i) \leftarrow N(i) \setminus \{j\}$                                 > remove neighbor  $j$  from neighbor list
26:  end if
27: end while

```

10.2.4.3 El segundo algoritmo distribuido aleatorizado

Nuestro último algoritmo distribuido es uno más reciente y aleatorio que tiene mejor desempeño que la versión distribuida del algoritmo de Luby en el que cada nodo activo i elige un número aleatorio r entre 0 y 1 en cada ronda. Si el nodo i tiene la r más grande entre sus vecinos, se asigna a MIS e informa a sus vecinos de su decisión para que no participen en la selección de MIS en las próximas rondas. El pseudocódigo detallado de listo para ser codificado para un nodo se muestra en el algoritmo 10.9, donde mostramos explícitamente la lógica para terminar una ronda [1]. Este algoritmo tiene la misma complejidad de tiempo y mensaje que el primero, ya que la lógica es similar.

Algorithm 10.9 *Dist_MIS3*

```
1: set of int curr_neighs ← N(i); received, values ← Ø
2: message types round, info
3: states INMIS, NONMIS
4: boolean inflag, outflag, round_recv, round_over ← false
5: while ¬round_over do
6:   receive msg(j)
7:   case msg(j).type of
8:     round(k):
9:       if inflag then state ← INMIS
10:      inflag ← false
11:    else if outflag then state ← NONMIS
12:      outflag ← false
13:    else draw rval ∈ {0, 1}
14:    send info(k, rval, state) to curr_neighs
15:    round_recv ← true
16:  info(k, r, sta):
17:    received ← received ∪ {j}
18:    values ← values ∪ {j}
19:    if sta = INMIS then outflag ← true
20:    if sta = INMIS/NONMIS
21:      lost_neighs ← lost_neighs ∪ {j}
22:    if round_recv ∧ (received = curr_neighs) then
23:      if ∀x ∈ curr_neighs : rval > rx ∈ values then
24:        inflag ← true
25:      round_over ← true; curr_neighs ← curr_neighs \ lost_neighs
26:      round_recv ← false; received, values, lost_neighs ← {Ø}
27:    end if
28:  end while
```

10.3 Conjuntos dominantes

Un conjunto dominante de una gráfica es un subconjunto de sus vértices, de manera que cada vértice está en este conjunto o adyacente a un vértice en él. Podemos definir formalmente este conjunto de la siguiente manera:

Definición 10.2 (conjunto dominante) Un conjunto dominante de una gráfica $G = (V, E)$ Es un subconjunto D de sus vértices tal que $\forall v \in V$, ya sea $v \in D$ o $v \in N(u)$ donde $u \in D$. Igualmente, un conjunto $D \subseteq V$ es un conjunto dominante si $\bigcup_{u \in D} N[u] = V$. En otras palabras, la unión de las vecindades cerradas de cada vértice de la gráfica debe ser igual al conjunto de vértices de la gráfica.

Existen diversas aplicaciones de conjuntos dominantes en redes de comunicación y sistemas de vigilancia. Un conjunto dominante se puede utilizar de manera efectiva como una red troncal para transferir mensajes en una red inalámbrica, como veremos en el Capítulo 14.

Cada conjunto máximo independiente es un conjunto dominante, ya que cada vértice de la gráfica estará en este conjunto o adyacente a un vértice en este conjunto. Sin embargo, no todos los conjuntos dominantes son independientes, ya que los miembros de un conjunto dominante pueden ser vecinos. Un conjunto dominante mínimo (MinDS) de un gráfico es el conjunto con el orden mínimo entre todos los conjuntos dominantes de ese gráfico. La cardinalidad de MinDS de un gráfico G se denomina número de dominación ($\gamma(G)$) de G . Un conjunto dominador mínimo (MDS) de un gráfico no contiene ningún otro conjunto dominante de ese gráfico como un subconjunto adecuado como se muestra en la Fig. 10.8. En otras palabras, la eliminación de un vértice de dicho conjunto destruirá la propiedad del conjunto dominante de este conjunto. En un conjunto dominador conectado (CDS), hay una ruta entre cada par de vértices en el conjunto dominante, que consiste solo en los vértices de conjunto dominantes. Formalmente, $\forall u \in D \wedge v \in D$, hay un camino $u, x_1, \dots, x_k, v \in D$ tal que

Un conjunto D que domina k de una gráfica $G = (V, E)$ consiste en vértices tales que cada $v \in V - D$ es adyacente a al menos k elementos de D . Un conjunto dominante de distancia k que a veces se confunde con los conceptos de conjunto dominante k es un conjunto de

vértices que tienen una distancia de al menos k a al menos uno de los vértices del conjunto dominante. Tenga en cuenta que la última definición afloja la definición general del conjunto dominante.

Encontrar MinDS de una gráfica es NP-difícil [2] y estamos interesados principalmente en encontrar conjuntos de gráficos dominantes mínimos cuando revisamos algoritmos secuenciales, paralelos y distribuidos para este propósito en esta sección.

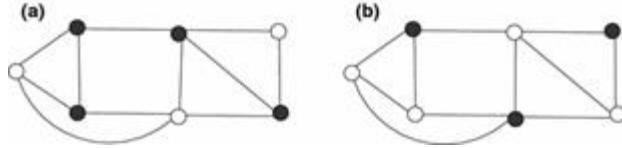


Fig. 10.8 a Un DS conectado con orden 4. b Un MDS no conectado con orden 3 de un gráfico de muestra. Los vértices en los conjuntos dominantes se muestran en negro.

10.3.1 Un algoritmo secuencial codicioso

Para el diseño de un algoritmo codicioso, usaremos la coloración de vértices de manera que los vértices en MDS se muestren en negro , sus vecinos dominados sean de color gris y cualquier otro vértice en el gráfico sea blanco con todos los vértices inicializados en blanco . El intervalo de un vértice v es el número de vecinos blancos que tiene, incluso si es blanco . La heurística que usaremos es seleccionar siempre un vértice blanco o gris con el intervalo más alto en el gráfico. Ya que nuestro objetivo es encontrar un MDS, estamos tratando de cubrir tantos blancos Vértices como sea posible en cada paso con esta heurística. El pseudocódigo para este algoritmo llamado Span_MDS se representa en el algoritmo 10.10.

Algorithm 10.10 Span_MDS

```

1: Input : An undirected unweighted graph  $G = (V, E)$ 
2: for all  $v \in V$  do ▷ all nodes are white initially
3:    $\text{color}[v] \leftarrow \text{white}$ 
4: end for
5:  $V' \leftarrow V, D \leftarrow \emptyset$ 
6: while  $\exists u \in V', \text{color}[u] = \text{white}$  do
7:   select  $v \in V'$  with the highest span
8:    $\text{color}[v] \leftarrow \text{black}$  ▷ select a MDS vertex
9:    $V' \leftarrow V' \setminus \{v\}$ 
10:  for all  $w \in (V' \cap N(v))$  do ▷ color its white neighbors grey
11:    if  $\text{color}[w] = \text{white}$  then
12:       $\text{color}[w] \leftarrow \text{grey}$ 
13:    end if
14:  end for
15: end while

```

La Figura 10.9 muestra la operación de este algoritmo en un gráfico de muestra donde el vértice con el tramo más alto es de color negro en cada iteración. Podemos ver que después de tres iteraciones, no quedan vértices blancos y el algoritmo termina. Si siempre seleccionamos un vértice gris con el intervalo más alto en cada iteración después de la primera, tenemos un DS conectado como se muestra en la Fig. 10.9 d.

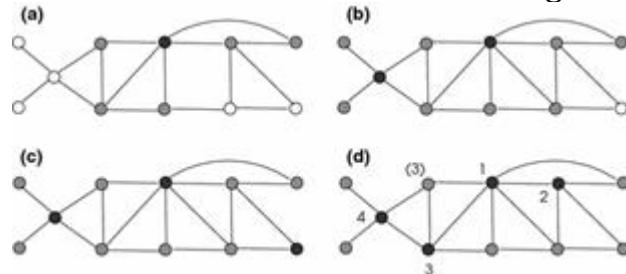


Fig. 10.9 Ejecución de MDS_Span en un gráfico de muestra. Las tres iteraciones se muestran en a - c con los vértices de conjunto dominantes en negro y los vértices dominados en gris. Un DS conectado se muestra en d para el mismo gráfico con los números de los pasos de iteración que se muestran junto a los vértices incluidos

Span_MDS El algoritmo proporciona un DS de un gráfico G , ya que no quedan vértices blancos cuando este algoritmo termina. Esto significa que todos los vértices de G terminan en color gris o negro como dominador o vértices dominados. El DS es mínimo (MDS), ya que en cada paso, uno o más vértices blancos se colorean coloreando un solo vértice negro que tiene el intervalo más alto. Eliminar este vértice del gráfico dejará uno o más vértices blancos y, por lo tanto, el DS obtenido es mínimo y no está contenido en un DS más grande. La complejidad del tiempo de este algoritmo es $O(n)$ ya que es posible que tengamos que n iteraciones como en el caso de una red lineal. La relación de aproximación de este algoritmo es $\ln d$ como se muestra en [12].

En ciertos gráficos, este algoritmo puede producir un conjunto dominante que puede estar lejos de ser óptimo. Por ejemplo, en un gráfico en el que dos vértices u y v están conectados por n vértices que usan rutas separadas, este algoritmo puede comenzar coloreando u o v en negro y luego intentar colorear todos los vértices intermedios en negro, lo que da como resultado $n-1$ Pasos donde la coloración de u y v negro es suficiente para formar un DS en dos pasos.

10.3.1.1 Guha - Algoritmos de Khuller

Guha - Los algoritmos de Khuller proporcionan mejoras a Span_MDS para que tales casos sean manejados más eficientemente [4]. El primer algoritmo presentado por estos autores comienza coloreando todos los vértices en blanco. Luego colorea el vértice de grado más alto del gráfico en negro y todos sus vecinos en gris. Luego, en cada paso, se encuentran los vecinos blancos de los nodos gris y blanco y, dependiendo del número de vecinos blancos, un vértice gris o un par de vértices gris - blanco se colorea de negro para poder colorear el mayor número posible de vértices blancos gris. Este proceso continúa hasta que no quedan más blancos vértices a la izquierda. Este algoritmo proporciona un MCDS con una relación de aproximación de $2(1 + H(d))$ donde H es la función armónica [4]. La ejecución de este algoritmo se muestra en la figura 10.11 (figura 10.10).

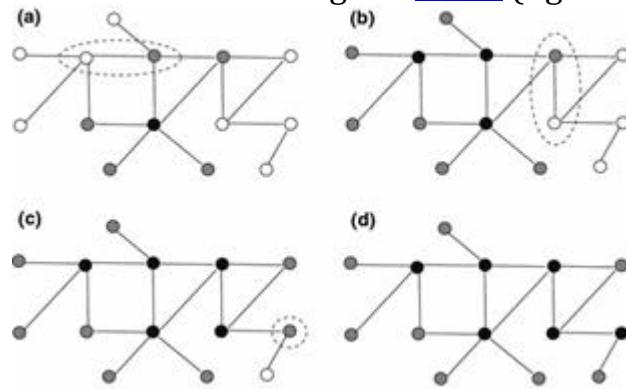


Fig. 10.10 Ejecución del primer algoritmo de Guha - Khuller en un gráfico de muestra. Los vértices MCDS seleccionados están coloreados en negro y sus vecinos se muestran en gris. Los pares de vértice o vértice seleccionados se muestran dentro de las regiones discontinuas. El MCDS se forma después de cuatro iteraciones del algoritmo

El segundo algoritmo también colorea todos los vértices con e inicialmente. Una pieza se define como un vértice blanco o un conjunto conectado de vértices negros. Este algoritmo tiene dos fases principales. En la primera fase, se selecciona el color negro del vértice u que dará la mayor disminución en el número de

piezas; El vértice u es de color negro y sus vecinos son de color gris . Esta fase continúa hasta que no hay blanco.vértices a la izquierda. Dado que la MDS obtenida puede no estar conectada, los vértices de la MDS se conectan utilizando un algoritmo basado en el árbol de Steiner en la segunda fase. La salida MCDS tiene una relación de aproximación de $3 + \ln d$ [4].

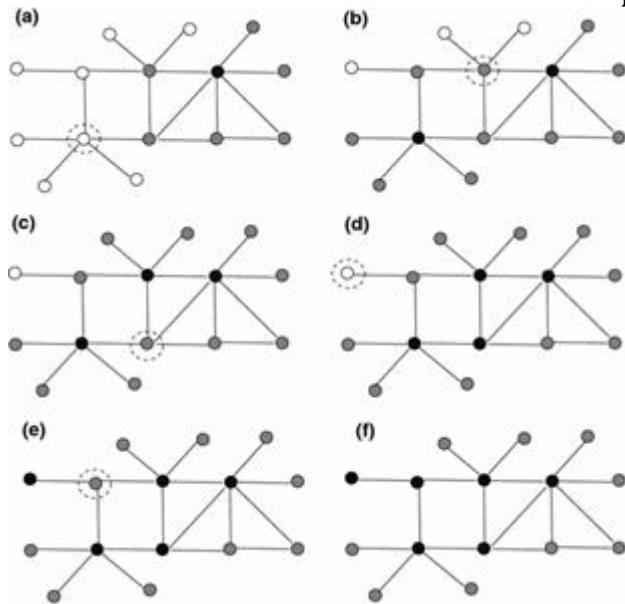


Fig. 10.11 Ejecución del segundo algoritmo de Guha - Khuller en un gráfico de muestra. El vértice seleccionado en cada iteración se muestra dentro de un círculo discontinuo que se muestra en negroen la siguiente iteración. Tenga en cuenta que podríamos haber optado por incluir el vértice blanco en MDS en c, ya que al hacerlo también se obtiene una pieza menos. Además, podríamos haber seleccionado el vértice gris junto al blanco en d para dar un paso menos. El MDS no conectado en e está conectado en f . El MCDS se forma después de seis iteraciones del algoritmo.

10.3.1.2 Algoritmos basados en MIS

Una forma alternativa de construir un MCDS es primero formar un MIS de la gráfica y luego conectar los vértices en este MIS para obtener un MCDS en el segundo paso. Podemos usar cualquier algoritmo para encontrar MIS, como el primer algoritmo codicioso de grado más bajo. La conexión de los vértices de MIS se puede realizar usando varias heurísticas, como seleccionar el vértice con el grado más alto entre los vértices de MIS o usar un algoritmo basado en el árbol de Steiner como en el segundo algoritmo de Guha - Khuller .

10.3.2 Un algoritmo distribuido para encontrar MDS

Un algoritmo distribuido simple basado en la extensión de un nodo en su vecindario de dos saltos en la red se puede formar de la siguiente manera. Cada nodo intercambia su intervalo con todos sus vecinos de dos saltos y, si tiene el mayor intervalo entre todos estos vecinos, ingresa al MDS. El algoritmo es ejecutado por un nodo i hasta que no tenga vecinos blancos que no sean MDS ni nodos dominados, como se muestra en el algoritmo 10.11. Se puede mostrar que este algoritmo calcula un MDS con $\ln d + 2$ relación de aproximación [12]. El número de rondas necesarias es $O (n)$, ya que habrá al menos un nuevo nodo que ingrese al conjunto dominante en cada ronda.

Algorithm 10.11 Dist_CDS

```
1: Input  $G = (V, E)$ 
2:  $S \leftarrow V$ , MIS  $\leftarrow \emptyset$ 
3: while  $\exists j \in N(i) : \text{color}[j] = \text{white} \wedge \text{state} \neq \text{INCDS}$  do
4:    $x \leftarrow \text{span}(i)$ 
5:   send  $x$  to nodes at distance of at most 2
6:   receive spans of nodes at distance 2
7:   if  $x > \text{all spans}$  then
8:     state  $\leftarrow \text{INCDS}$ 
9:   end if
10: end while
```

10.4 Cubierta Vertex

Una cubierta de vértice o una cubierta de un gráfico es un subconjunto de sus vértices, de manera que cada borde incide en al menos un vértice en este subconjunto. La cobertura de vértice tiene numerosas aplicaciones, como colocar tiendas en una región para que cada camino lleve a al menos una tienda, en bioinformática [10] y en química [9]. Podemos definir esta propiedad de conjunto formalmente de la siguiente manera:

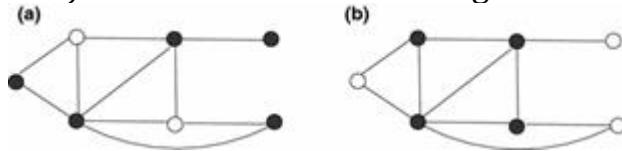


Fig. 10.12 a Un MVC desconectado con orden 5. b A MinVC conectado con orden 4 de un gráfico de muestra. Los vértices en las cubiertas de vértices se muestran en gris. La tapa del vértice en bestá conectada.

Definición 10.3 (cubierta de vértice) Una cubierta de vértice de un gráfico $G = (V, E)$ es un subconjunto V' de sus vértices tales que $\forall(u, v) \in E$, ya sea u, v o ambos en V' .

Una cubierta de vértices mínima (MinVC) de un gráfico es el conjunto con la cubierta de vértices de orden mínimo entre todas las cubiertas de vértices de esa gráfica. Una cubierta de vértice mínima (MVC) de una gráfica no contiene ninguna otra cubierta de vértice de esa gráfica como se muestra en la Fig. 10.12 . En otras palabras, eliminar un vértice de un MVC destruirá la propiedad de cubierta de vértice de este conjunto. Encontrar MinVC de una gráfica es NP-difícil [2] y, comúnmente, nuestro objetivo es encontrar una cubierta de vértice mínima de una gráfica.

Habíamos dicho esto antes en el capítulo. 3 como ejemplo de una reducción y sería apropiado reformularlo aquí. Un conjunto V' Es una cubierta de vértice de una gráfica. $G = (V, E)$ si y solo si $V' \setminus V'$ es un conjunto independiente de G . Ya que V' es una cubierta de vértice, cada borde (u, v) tiene al menos un punto final en V' . Si ambos vértices u y v están en $V' \setminus V'$, entonces $(u, v) \notin E$ como de otra manera V' no será una cubierta de vértice ya que no tiene un vértice que incide en el borde (u, v) . Por lo tanto, $V' \setminus V'$ es un conjunto independiente de G . Vimos que I es un conjunto independiente de una gráfica G si y solo si soy una camarilla en \bar{G} . Por lo tanto, los tres problemas de conjunto independiente, camarilla y cubierta de vértice son equivalentes.

Los vértices de un gráfico pueden tener pesos asociados con ellos que representan algunas propiedades físicas, como la capacidad de un enrutador en una red informática. En tal caso, nuestro objetivo es encontrar una cubierta de vértice con el peso total mínimo. La cubierta de vértice de peso mínimo (MinWVC) es la cubierta de vértice con el peso total mínimo entre todas las cubiertas de vértice ponderadas de un gráfico. Visto desde otra perspectiva, es posible que debamos buscar una cubierta de vértice conectada mínima (MCVC), lo que quiere decir que hay una ruta entre cada par de vértices en la cubierta que

consiste en un subconjunto de vértices en este conjunto solamente. Revisaremos algoritmos secuenciales, paralelos y distribuidos para encontrar coberturas de vértices mínimos ponderados y no ponderados en esta sección.

10.4.1 Cubierta de vértice no ponderada

Los algoritmos de cobertura de vértices no ponderados suponen que los vértices de la gráfica no tienen ponderaciones (o, en ocasiones, ponderaciones unitarias) asignadas.

10.4.1.1 Cobertura de vértice de gráfico general sin ponderar

Como primer acercamiento natural, podemos seleccionar el vértice con el grado más alto primero para incluirlo en la cubierta del vértice y continuar seleccionando siempre los vértices de grado más alto. En lugar de seleccionar los vértices con grados estáticos iniciales, seleccionamos el grado más alto actual en cada iteración, ya que eliminar un vértice y sus bordes incidentes causará una disminución en los grados de sus vecinos activos. Sin embargo, este enfoque no produce una relación de aproximación fija. De hecho, la relación de aproximación es $\Theta(\log n)$. Lo cual no es favorable ya que depende del número de vértices.

Ya hemos revisado cómo calcular la cobertura de vértice de una gráfica a partir de una coincidencia que produjo una relación de aproximación constante de 2 como ejemplo de un algoritmo de aproximación (consulte la Sección 3.8.2). En este algoritmo, encontramos una coincidencia máxima de un gráfico al seleccionar un borde legal arbitrario e incluir ambos extremos del borde seleccionado en el MVC. El borde seleccionado y sus bordes adyacentes se eliminan del gráfico y el proceso se repite hasta que no queden bordes. En cuanto a un algoritmo de cobertura de vértice paralelo, siempre podemos usar un algoritmo de coincidencia máxima paralelo incluyendo ambos puntos finales de bordes coincidentes en la cubierta de vértice en consecuencia.

10.4.1.2 Cobertura de vértice de gráfico bipartito no ponderado

En cualquier gráfico bipartito, el número de bordes en una coincidencia máxima es igual al número de vértices en una cobertura de vértice mínima según el teorema de König [5]. Sabemos cómo encontrar una coincidencia máxima de un gráfico bipartito por el algoritmo de ruta de aumento en tiempo $O(nm)$ o por el algoritmo Hopcroft-Karp en $O(\sqrt{nm})$ tiempo (ver sec. 9.3). Una vez que tengamos una máxima coincidencia M^* de un grafo bipartito $G = (A \cup B, E)$, sabemos el tamaño de la cubierta mínima del vértice, $|V^*|$ es igual a $|M^*|$. Pero necesitamos encontrar los elementos del conjunto VC. Primero proporcionaremos un método para hacerlo y luego demostraremos su corrección. Pero en primer lugar, observamos $V(u, v) \in M^*$, debemos incluir el vértice u , o el vértice v pero no ambos en V^* simplemente porque un punto final del borde (u, v) es suficiente para cubrirlo. Nuestro procedimiento para encontrar los vértices contenidos en la cubierta del vértice mínimo se basa en esta observación. Si marcamos todos los vértices posibles que se pueden alcanzar utilizando rutas alternas de vértices no emparejados en A en los conjuntos S y T para aquellos en A y B respectivamente, el conjunto mínimo de cobertura de vértices V^* es $(A \setminus S) \cup (B \cap T)$ como se muestra en el algoritmo 10.12.

En la Fig. 10.13 a se muestra un ejemplo de gráfico bipartito con una coincidencia máxima mostrada en líneas en negrita. Primero formamos G' . Orientando los bordes. El único vértice no emparejado en A es d , por lo tanto, ejecutamos el algoritmo DFS desde este vértice visitando los vértices. $\{r, a, p, b, s, d\}$ se muestra en gris. La cobertura mínima del

vértice es entonces $(\{a, b, c, d, e\} \setminus \{a, b, d, e\}) \cup (\{p, q, r, s, t\} \cap \{p, r, s\}) = \{c, e, p, r, s\}$ con orden 4, que es el tamaño de la coincidencia máxima como se muestra en la Fig. 10.13 b. Podemos ver que solo se incluye un extremo de los bordes coincidentes en la cubierta de vértice mínimo.

El teorema 10.2 El algoritmo 10.12 construye correctamente una cobertura de vértice mínima de un gráfico bipartito a partir de su coincidencia máxima en $O(n + m)$ hora.

Algorithm 10.12 Bipartite_MinVC

```

1: Input: an undirected, unweighted bipartite graph  $G = (A \cup B, E)$ 
2:      a maximum matching  $M^*$  of  $G$ 
3: Output: minimum vertex cover  $V^*$  of  $G$ 
4:
5: form  $G'$  by directing matched edges from  $B$  to  $A$  and unmatched edges from  $A$  to  $B$ 
6: for all  $u \in A$  of  $G'$  that is free do
7:   run  $DFS(u)$  and insert all vertices visited in  $A$  in  $S$ , and visited in  $B$  in  $T$ 
8: end for
9:  $V^* \leftarrow (A \setminus S) \cup T$ 

```

Prueba Nosotros primero demostraremos V^* es una cubierta de vértice, entonces es una mínima. Supongamos V^* . No es una cubierta de vértice. Entonces $\exists (u, v) \in E$ con $u \in A$ y $v \in B$ tal que $u \notin V^*$ y $v \notin V^*$. Ya que $V^* = ((A \setminus S) \cup T)$, Debemos tener $u \in S$ y $v \in B \setminus T$. Tenemos dos posibilidades en tal caso:

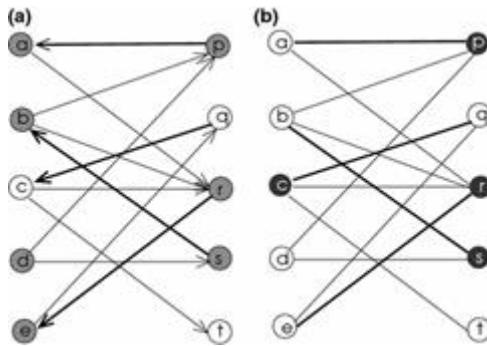


Fig. 10.13 Un ejemplo de gráfico bipartito para probar el algoritmo de cobertura de vértice mínimo desde la coincidencia máxima. Los vértices sombreados en a son visitados por DFS desde el vértice d y los vértices oscuros en b están contenidos en la cubierta mínima del vértice

Asumir ventaja $(u, v) \in M^*$. Ya que $u \in S$, debemos haberlo visitado por DFS. Sin embargo, solo podemos alcanzar u sobre un borde coincidente que significa $v \in T$ dando como resultado una contradicción.

- 1.
2. Asumir ventaja $(u, v) \notin M^*$. En este caso, ya que $u \in S$, v se incluye en T, que es una contradicción otra vez desde que asumimos $b \notin T$.

Ahora mostraremos $|V^*| \geq M^*$ y desde $|M^*| \geq V^*$ habremos probado $|M^*| = |V^*|$. Las siguientes observaciones pueden ser expresadas [11].

$\exists u \in (A \setminus S)$ que es inigualable ya que partimos cada DFS desde un vértice sin igual en una que se incluye en S .

- 1.
2. $\exists u \in T$ eso es incomparable ya que esto significaría que existe un camino de aumento de G con respecto a M^* y por lo tanto M^* no sería maximo
3. $\exists (u, v) \in M^*$ tal que $u \in (A \setminus S)$ y $v \in T$. Si existiera tal borde, u estaría en S cuando se encontró que b estaba en T, lo que resulta en una contradicción ya que $u \in (A \setminus S)$.

Podemos concluir que cada vértice del conjunto V' es incidental a un borde coincidente por las dos primeras observaciones y los dos puntos finales de un borde coincidente no están ambos incluidos en V' . Por la última observación. \square

10.4.1.3 Algoritmos distribuidos

Podemos usar cualquiera de los algoritmos de coincidencia distribuidos descritos en el Cap. 9 para encontrar bordes a juego. Cualquier nodo que tenga un borde de coincidencia de incidente puede marcarse como miembro del MVC en una configuración distribuida. Este enfoque proporcionará un MVC con una relación de aproximación de 2 como en el caso secuencial, ya que simplemente estamos imitando la operación secuencial en un entorno de red. Describiremos dos algoritmos distribuidos sincrónos; la primera es la versión distribuida del algoritmo codicioso que favorece a los nodos de alto grado y la segunda tiene un número fijo de rondas que proporcionan una relación de aproximación constante.

Algoritmo codicioso

El primer grado en el vértice heurístico proporcionó una relación de aproximación que dependía del orden de la gráfica. Sin embargo, describiremos una versión distribuida de este algoritmo para dar otro ejemplo de cómo convertir un algoritmo secuencial en uno distribuido.

El algoritmo MVC distribuido que utiliza esta heurística funciona en rondas sincrónas. Cada nodo activo compara su grado actual con sus vecinos activos y si tiene el grado más alto, se marca para estar en el MVC e informa a sus vecinos de su decisión. La ejecución de este algoritmo en un gráfico de muestra se muestra en la Fig. 10.14.

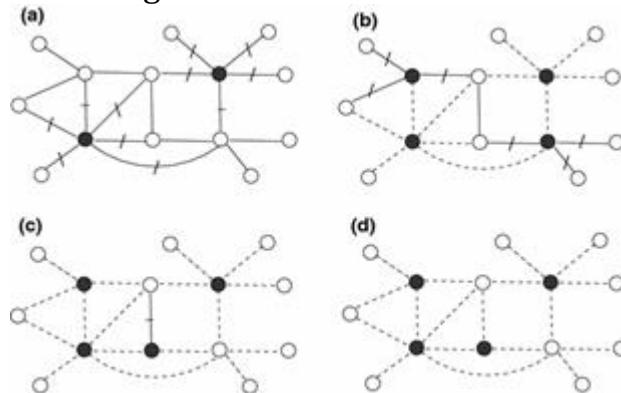


Fig. 10.14 Ejecución del codicioso algoritmo de cobertura de vértice distribuido en un ejemplo de gráfico no dirigido. Los nodos incluidos en la cubierta se muestran en negrita con los bordes eliminados como discontinuos en cada iteración

Parnas -Ron Algoritmo

Parnas y Ron proporcionaron un algoritmo de cobertura de vértice no ponderado distribuido sincrónico que funciona para un número constante de rondas dependiendo del grado más alto $d(G)/2^i$ en una gráfica [8]. Cada nodo en la red verifica si su grado es mayor que $d(G)/2^i$ en cada ronda i . Si esta comprobación devuelve un valor verdadero, se convierte en parte de la cubierta del vértice.

Algorithm 10.13 Parnas_MVC

```

1: Input: unweighted undirected graph  $G = (V, E)$ ,  $\Delta(G)$ 
2: Output: vertex cover  $C$  of  $G$ 
3: message types:  $in\_cover$                                  $\rightarrow$  sent by a node entering VC
4: states: {INV, NONVC}                                      $\triangleright$  states of a node as in or not in the vertex cover
5:  $C \leftarrow \emptyset$ ;  $state \leftarrow \text{NONVC}$                        $\triangleright$  initialize vertex cover  $C$ 
6: for  $i = 1$  to  $\log \Delta(G)$  do
7:   if  $d(i) \geq \Delta(G)/2^i$  then
8:      $state \leftarrow \text{INV}$ 
9:     send  $in\_cover(i)$  to  $N(i)$ 
10:    end if
11:   if  $in\_cover(j)$  is received from neighbor  $j$  then
12:      $N(i) \leftarrow N(i) \setminus \{j\}$                                  $\triangleright$  remove neighbor  $j$  and the edge  $(i, j)$  from graph
13:   end if
14: end for

```

Un gran inconveniente de este algoritmo es que $\Delta(G)$. El parámetro debe difundirse a todos los nodos antes de la ejecución del algoritmo. La operación de este algoritmo en la misma gráfica de muestra de la figura 10.14 se muestra en la figura 10.15.

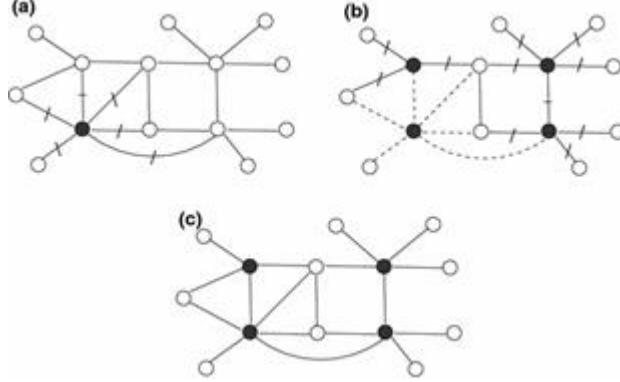


Fig. 10.15 Ejecución del algoritmo Parnas –Ron en la misma gráfica de muestra de la Fig. 10.14

Teorema 10.3 *Parnas_MVC* El algoritmo construye correctamente una cubierta de vértice mínima VC tal que $|V^*| \leq |VC| \leq (2 \log \Delta(G) + 1) \cdot |V^*|$ donde V^* Es la cobertura mínima del vértice.

Prueba En la última iteración del algoritmo, todos los vértices que tienen grado ≥ 1 se incluirán en la cubierta de vértice, por lo tanto, todos los bordes del gráfico se eliminarán con al menos uno de sus vértices incidentes incluidos en la cubierta de vértice, por lo que el algoritmo construye correctamente una cubierta de vértice.

El número de iteraciones es como máximo $\log \Delta(G)$ desde después $\log \Delta(G)$ En las iteraciones, los vértices restantes en la gráfica tendrán 0 grados. En cada iteración, hay como máximo $2|V^*|$ Nuevos vértices que se agregan desde $G - V^*$ a VC. Al comienzo de la i^{a} iteración, el grado de cada vértice es como máximo $d/2^{i-1}$. Por lo tanto, el número de bordes entre V^* y $G - V^*$ es a lo más $|V^*| \cdot d(\Delta(G)/2^{i-1})$. Vamos a asignar x_i a los vértices de números en $G - V^*$ de grado al menos $d/2^i$ al comienzo de la i^{a} iteración. Por lo tanto, $x_i \cdot d/2^i \leq |V^*| \cdot d/2^{i-1}$; por lo tanto, $x_i \leq 2|V^*|$. Como tenemos a lo sumo $\log \Delta(G)$ iteraciones, se deduce que el número total de vértices incluidos en VC es a lo sumo $2|V^*| \cdot \log \Delta(G)$ [8]. \square

10.4.2 Cubierta de vértice ponderada

Cuando los vértices tienen pesos, buscamos una cubierta de vértice ponderada mínima. Encontrar la cubierta del vértice con el peso total mínimo es NP-duro como la mayoría de los problemas que hemos estudiado. El algoritmo de fijación de precios es un algoritmo de aproximación secuencial para el problema MWVC como se describe a continuación junto con algoritmos paralelos y distribuidos para el problema MWVC.

10.4.2.1 Algoritmo de fijación de precios

La idea principal de este algoritmo es cubrir un borde con el peso mínimo del vértice que incide. Un borde $e \in E$ paga un precio $p_e \geq 0$ estar cubierto por el vértice u es un incidente y la suma de los precios asignados a los bordes que inciden en un vértice u no debe exceder el peso w_u de un vértice. Formalmente,

- Un borde $e \in E$ paga un precio $p_e \geq 0$. Para ser cubierto por un vértice que es incidente.
- $\forall u \in V \sum_{e \in E} p_e \leq w_u$

Cuando la suma de los precios de los bordes que inciden en un vértice es igual a su peso, se dice que el vértice es ajustado. Una posible implementación de este algoritmo se muestra en el algoritmo 10.14 donde cada vértice $v \in V$ tiene una capacidad c_v que se inicializa a su peso y el conjunto de borde activo S se inicializa a E . El algoritmo inspecciona cada borde. $e_{uv} \in S$ y si u o v tiene una capacidad restante, carga el borde e con el menor de las capacidades. Cuando la capacidad de u o v se convierte en 0, se etiqueta como un nodo ajustado y se incluye en el $MWVC$. El algoritmo se detiene cuando cada vértice tiene un vértice apretado, al menos incidental a uno de sus puntos finales, lo que significa que todos los bordes están cubiertos por vértices apretados en uno o ambos extremos. La ejecución de este algoritmo se muestra en la figura 10.16.

Algoritmo 10.14 Pricing_MWVC

```

1: Input  $G(V, E)$ 
2:  $S \leftarrow E$ ,  $V' \leftarrow \emptyset$ 
3: while  $S \neq \emptyset$  do
4:   pick any  $e_{uv} \in S$ 
5:   if  $c_u \neq 0 \vee c_v \neq 0$  then
6:      $q \leftarrow$  node with  $\min\{c_u, c_v\}$ 
7:      $p_e \leftarrow c_q$ ,  $q \leftarrow right$ 
8:      $V' \leftarrow V' \cup \{q\}$ ;  $S \leftarrow S \setminus \{e\} \cup$  any other edge incident at  $q$ 
9:   end if
10: end while

```

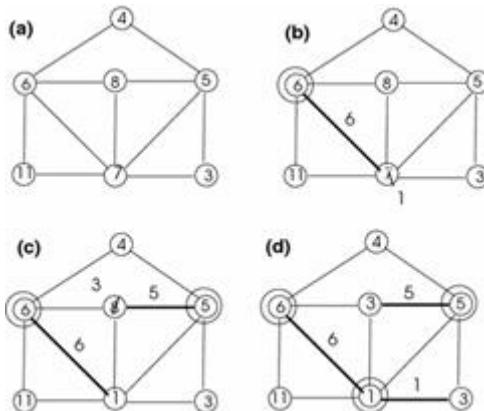


Fig. 10.16 Ejecución de **Pricing_MWVC** Algoritmo en una gráfica de muestra. Los pesos de los vértices se muestran dentro de ellos y los vértices ajustados se muestran con círculos dobles. Después de cinco iteraciones, MWVC se forma como se muestra en f con un peso total de 18

Teorema 10.4 **Pricing_MWVC** El algoritmo construye correctamente un MWVC de un gráfico en iteraciones $O(n)$ con una relación de aproximación de 2.

Prueba Este algoritmo construye correctamente una cubierta de vértice, ya que continuamos explorando todos los bordes hasta que no queden bordes con un vértice ajustado en uno o ambos de sus puntos finales, por lo que todos los bordes están

cubiertos. Habrá al menos un vértice apretado en cada iteración, lo que resultará en una complejidad de tiempo $O(n)$

Dejar V' Ser el conjunto de todos los vértices ajustados al final del algoritmo y V'' Los vértices mínimos cubren los vértices. Tenemos que mostrar $w(V') \leq 2w(V'')$. Dado que todos los vértices en V' Estamos apretados, podemos escribir la siguiente ecuación.

$$w(V') = \sum_{v \in V'} w_v = \sum_{v \in V'} \sum_{e=(u,v)} p_e \leq \sum_{v \in V} \sum_{e=(u,v)} p_e$$

(10.1)

Como cada borde se cuenta dos veces, podemos escribir lo anterior como

$$= 2 \sum_{e \in E} \leq 2w(V'')$$

(10.2)

□

10.4.2.2 Algoritmos distribuidos

Podemos usar de nuevo cualquiera de los algoritmos de coincidencia ponderada distribuida con la simple modificación de incluir ambos puntos finales de los bordes coincidentes en la cubierta de vértice mínimo. Como otro enfoque simple, podemos usar el método codicioso en un algoritmo distribuido síncrono en el que cada nodo que tiene el menor peso en su vecindario ingresa al conjunto de coberturas de vértices. El algoritmo diseñado de esta manera es básicamente muy similar al algoritmo distribuido codicioso que utiliza grados de nodos y el nodo de grado más alto localmente se asigna a la cobertura de vértice en el caso de cobertura de vértice no ponderado. Sin embargo, en ciertas topologías de red, puede producir una solución que dista mucho de ser óptima, como se muestra en la configuración en estrella de la figura 10.17 .

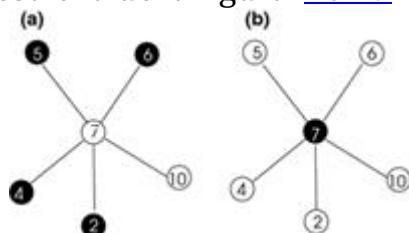


Fig. 10.17 un gráfico Una estrella con vértices MWVC muestran en negro formado por el algoritmo de MWVC distribuido codicioso que tiene un peso total de 17 para la cubierta b la óptima MWVC para el mismo gráfico

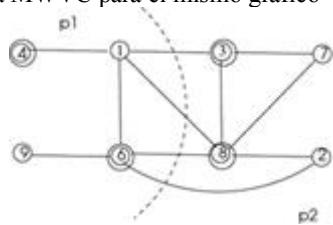


Fig. 10.18 Manejo de vértices fantasma.

10.4.3 Algoritmos paralelos

Los algoritmos paralelos, ya sea que la memoria compartida o la memoria distribuida para MVC en casos no ponderados o ponderados sean escasos. Se puede lograr un enfoque básico para realizar la búsqueda paralela de la cobertura de vértices al dividir la gráfica en un número de aproximadamente subgrafos de igual orden y hacer que cada proceso encuentre cubiertas de vértices en sus subgrafos. Cada proceso puede funcionar independientemente para formar sus cubiertas de vértices, sin embargo, se debe tener cuidado al tratar con vértices que aparecen en los bordes de las particiones. Una forma de resolver este problema es utilizar el concepto de vértice fantasma en el que los vértices de los bordes se replican en cada proceso y la elección de un vértice fantasma para incluir en qué partición se puede manejar rompiendo simetrías utilizando identificadores de vértice únicos, como hemos resumido brevemente descrito en el cap. 4.

Este concepto se ilustra en la Fig. 10.18 donde una gráfica simple con 8 vértices se divide en dos procesos paralelos P_1 y P_2 . Cada vértice tiene un identificador entero único y los vértices de borde entre las particiones son 1, 3, 6, 8 y 2. Cada proceso es responsable del vértice de borde de identificador más alto que se le asigna. A este respecto, los bordes (1, 8) y (1, 3) deben estar cubiertos por el proceso P_2 y (6, 2) está cubierto por proceso P_1 . Los vértices en la cubierta se muestran mediante círculos dobles.

El algoritmo 10.15 muestra un posible pseudocódigo de un algoritmo paralelo para realizar una cobertura de vértice paralelo en el que un proceso especial, la raíz, divide el gráfico utilizando un algoritmo adecuado, envía cada partición a los procesos y recopila los resultados de la cobertura de vértice mínimo parcial para formar el final Cobertura mínima del vértice de la gráfica. Cada proceso encuentra la cobertura de vértice mínima en su partición al incluir vecinos identificadores más altos de los vértices de borde en su partición y envía la cobertura de vértice mínima parcial a la raíz .

Algorithm 10.15 Par_MVC

```

1: Input: unweighted undirected graph  $G = (V, E)$ ,  $\Delta(G)$ 
2: Output: minimal vertex cover  $MVC$  of  $G$ 
3: if  $i = root$  then
4:   partition  $G$  into  $k$  subgraphs  $G(1), \dots, G(k)$ 
5:   for  $i = 1$  to  $k$  do
6:     send  $G(i)$  to  $p_i$  with neighbors of border vertices
7:   end for
8:   for  $i = 1$  to  $k$  do
9:     receive  $VC(i)$  from  $p_i$ 
10:     $MVC \leftarrow MVC \cup VC(i)$ 
11:   end for
12: else
13:   receive  $G(i)$  with neighbors of border vertices
14:   for all  $u \in$  border vertices do
15:     if  $\exists v \in$  neighbors of border vertices such that  $(u, v) \in E$  then
16:       if  $id(u) > id(v)$  then
17:          $G(i) \leftarrow G(i) \cup \{v\}$ 
18:       else
19:          $G(i) \leftarrow G(i) \setminus \{u\}$ 
20:       end if
21:     end if
22:   end for
23:   find vertex cover  $VC(i)$  of  $G(i)$ 
24:   send  $VC(i)$  to root
25: end if

```

10.5 Notas de capítulo

Hemos revisado tres problemas en los gráficos; El conjunto máximo independiente, el conjunto dominante mínimo y los problemas de cobertura de vértice mínimo. Todos estos

problemas son NP-duros y los algoritmos propuestos en la literatura son algoritmos de aproximación o heurísticos que encuentran soluciones máximas o mínimas en lugar de subgrafos de orden máximo o mínimo. Podemos emplear algoritmos codiciosos comúnmente mediante el uso de cierta heurística, pero con frecuencia se buscan algoritmos con mejores desempeños. Como hemos señalado en el primer algoritmo de mayor grado para encontrar una cobertura de vértice mínima, una heurística aparentemente natural puede no proporcionar una relación de aproximación constante.

Vimos que los problemas de cobertura de conjunto, camarilla y vértice independientes son computacionalmente equivalentes; un conjunto de vértices V' de un grafo $G = (V, E)$ es una cubierta de vértice de G si y solo si $V \setminus V'$ es un conjunto independiente de G . Además, el conjunto $V \setminus V'$ es una camarilla en \bar{G} si y solo si V' es un conjunto independiente de g . Se puede conectar un conjunto dominante y se puede usar un conjunto independiente como primer paso para formar un conjunto dominante conectado. También se puede conectar una cubierta de vértice y también los vértices pueden tener pesos asociados con ellos que representan algún parámetro físico atribuido a los nodos del sistema que representa el gráfico. Las versiones ponderadas de estos problemas generalmente requieren consideraciones diferentes a las no ponderadas.

Los algoritmos paralelos para estos problemas son escasos y con los avances recientes que resultan en la disponibilidad de datos para redes reales muy grandes, existe una creciente necesidad de algoritmos paralelos. En algunos casos, se han desarrollado algoritmos paralelos deterministas rápidos y eficientes para estos problemas, pero estos algoritmos pueden ser bastante complicados. Para el cálculo paralelo de MDS, podemos usar un algoritmo similar al utilizado para la cobertura de vértice al dividir la gráfica en un conjunto de procesos, cada uno de los cuales ejecuta un algoritmo MDS en su partición.

Los algoritmos de red distribuida se encuentran en un nivel más investigado para estos problemas, como hemos señalado. En muchos casos, estos algoritmos se derivan de los secuenciales, sin embargo, todavía hay una necesidad de algoritmos con mejores rendimientos.

Ceremonias

Proponga una heurística para encontrar el IS en el algoritmo 10.2 e implemente este método con el algoritmo 10.2 para encontrar el MIS de la figura [10.19](#).

- 1.
2. Para encontrar el MaxIS de un árbol arraigado, podemos incluir todas las hojas del árbol en el MaxIS y movernos hacia arriba en el árbol al no incluir un nivel en el MaxIS y el siguiente nivel en el MaxIS .

Escriba el pseudocódigo para este algoritmo secuencial.

a.

segundo. Demuestre que este algoritmo encuentra el MaxIS para un árbol enraizado.

do. Muestra cómo el código secuencial se puede convertir en un algoritmo distribuido.

re. Proponga un método para encontrar el MaxIS de un árbol en paralelo usando este método.

3. Implemente el primer algoritmo de Guha - Khuller para encontrar el MDS en el gráfico de ejemplo de la figura [10.20](#).
4. Implemente el algoritmo de rango distribuido para encontrar el MDS en el gráfico de ejemplo de la figura [10.21](#).
5. Encuentre el MVC del ejemplo de gráfico bipartito de la figura [10.22](#) usando la coincidencia. Mostrar todas las iteraciones del algoritmo.
6. Calcule el MWVC en el gráfico de muestra de la figura [10.23](#) usando el algoritmo de precios.

7. Implemente el algoritmo de cobertura de vértice ponderado distribuido codicioso para encontrar el MWVC en el gráfico que se muestra en la Fig. [10.24](#), donde se muestran los pesos de los vértices en su interior.

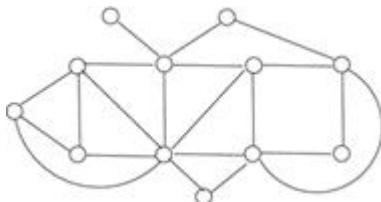


Fig. 10.19 Ejemplo de gráfica para el Ejercicio 1

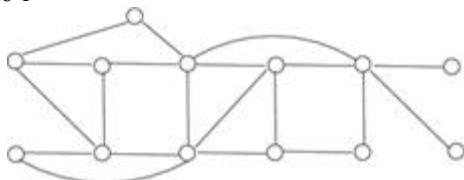


Fig. 10.20 Gráfico de muestra para el Ejercicio 3

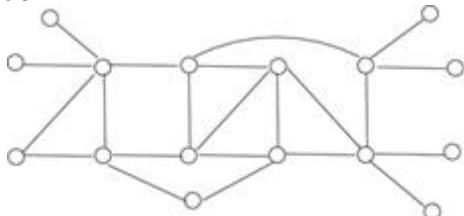


Fig. 10.21 Gráfico de muestra para el Ejercicio 4

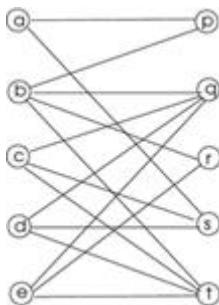


Fig. 10.22 Gráfico de muestra para el Ejercicio 5

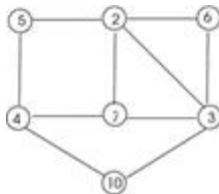


Fig. 10.23 Gráfico de muestra para el Ejercicio 6

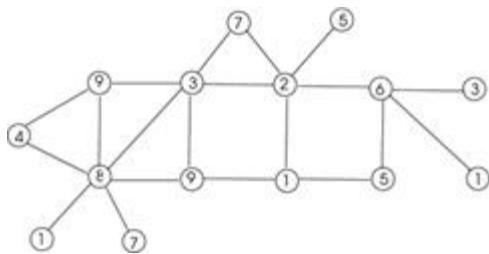


Fig. 10.24 Gráfico de muestra para el Ejercicio 7

Referencias

2. Erciyes K (2013) Algoritmos de gráficos distribuidos para redes de computadoras (Cap. 10). Serie de comunicaciones y redes informáticas, Springer, Berlín. ISBN 978-1-4471-5172-2
[Referencia cruzada](#)
3. Garey MR, Johnson DS (1978) Computadoras e intratabilidad: una guía para la teoría de la integridad de NP. Freeman, nueva york
4. Grama A, Gupta A, Karypis G, Kumar V (2003) Introducción a la computación en paralelo (Capítulo 10), 2^a ed . Addison Wesley, leyendo
6. Guha S, Khuller S (1998) Algoritmos de aproximación para conjuntos dominantes conectados. Algorithmica 20 (4): 374–387
[MathSciNet Crossref](#)
8. König D (1931) Graphen und Matrizen . Matemáticas Lapok 38: 116119
9. Koufogiannakis C, Young N (2009) Algoritmos distribuidos y paralelos para la cobertura de vértices ponderados y otros problemas de cobertura. En: El 28 ° simposio ACM SIGACT-SIGOPS sobre principios de computación distribuida (PODC 2009)
10. Luby M (1986) Un algoritmo paralelo simple para el problema del conjunto máximo independiente. SIAM J Comput 15 (4): 1036–1053
[MathSciNet Crossref](#)
11. Parnas M, Ron D (2007) Aproximando la cobertura de vértice mínima en tiempo sublineal y una conexión a algoritmos distribuidos. Theor Comput Sci 381 (1): 183–196
[MathSciNet Crossref](#)
12. Rhodes N, Willett P, Calvet A, Dunbar JB, Humblet C (2003) Clip: similitud en la búsqueda de bases de datos 3D utilizando la detección de camarillas. J Chem Inf Comput Sci 43 (2): 443448
[Referencia cruzada](#)
13. Samudrala R, Moult J (2006) Un algoritmo gráfico-teórico para el modelado comparativo de la estructura de proteínas. J Mol Biol 279 (1): 287302
14. Stein C (2012) IEOR 8100. Apuntes de clase. Universidad de Colombia
15. Wattenhofer R (2016) Principios de computación distribuida (Capítulo 7). Apuntes de clase. ETH Zurich

11. colorear

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Colorear en una gráfica se refiere a colorear vértices , colorear bordes o ambos, en cuyo caso se llama coloración total . A cada vértice se le asigna un color de un conjunto de colores, de modo que no hay dos vértices adyacentes que tengan el mismo color en la coloración de vértice. La coloración de bordes es el proceso de asignar colores a los bordes de un gráfico de manera que no se asigne el mismo color a dos bordes incidentes en el mismo vértice. En este capítulo, revisamos los algoritmos secuenciales, paralelos y distribuidos para la coloración de vértices y bordes.

11.1 Introducción

Colorear en una gráfica se refiere a colorear vértices , colorear bordes o ambos, en cuyo caso se llama coloración total. A cada vértice se le asigna un color de un conjunto de colores, de modo que no hay dos vértices adyacentes que tengan el mismo color en la coloración de vértice. Este método tiene muchas aplicaciones, incluida la asignación de frecuencia de canal y la programación de trabajos. La asignación de canales de frecuencia a estaciones de radio puede modelarse coloreando un gráfico con los vértices que representan estaciones de radio y un borde conecta dos estaciones si están dentro de una distancia de interferencia entre sí. Diferentes colores en este caso corresponden a diferentes frecuencias de emisión. Como ejemplo de programación, es posible que tengamos que asignar exámenes finales en una universidad para que ningún estudiante tome dos exámenes al mismo tiempo. Podemos representar cada examen mediante un vértice en una gráfica y un borde conecta dos vértices a y b si un estudiante está tomando ambos exámenes finales a y b. Si los colores de los vértices representan los intervalos de tiempo para los exámenes finales, nuestro objetivo es colorear cada vértice de la gráfica de manera que dos vértices adyacentes reciban un color diferente, lo que significa que un alumno que tome ambos exámenes los atenderá en diferentes intervalos de tiempo. La coloración de bordes es el proceso de asignar colores a los bordes de un gráfico de manera que no se asigne el mismo color a dos bordes incidentes en el mismo vértice. La coloración de bordes se puede usar en la planificación de un horario para que los maestros enseñen cursos en una escuela para lograr la cantidad mínima de tiempo de curso. Se puede formar un gráfico bipartito con el profesor y particiones de cursos de vértices, y buscamos una coloración de bordes mínima de este gráfico. Luego, encontramos un valor de tiempo

máximo que cualquier maestro está involucrado en la enseñanza, que es el tiempo máximo empleado en la enseñanza de todos los cursos.[\[7\]](#).

El objetivo principal de cualquier método de coloración es utilizar un número mínimo posible de colores. Dado que este es un problema de NP-duro para la coloración de vértices y bordes [\[10\]](#), comúnmente se emplean varias heurísticas. Los algoritmos de coloración de vértices paralelos intentan colorear simultáneamente diferentes regiones de la gráfica bajo consideración por una serie de procesos para lograr una aceleración. Por otro lado, un nodo del gráfico de red ejecuta un algoritmo de coloreado de gráfico distribuido, de modo que cada nodo determina su color al final.

Podemos tener algoritmos de coloración de bordes paralelos y distribuidos como en la coloración de vértices. La coloración total del gráfico se puede lograr al colorear los vértices y los bordes de una gráfica. Comenzamos con el problema de la coloración de vértices en este capítulo describiendo los algoritmos secuenciales, paralelos y distribuidos para esta tarea y continuamos con los algoritmos para la coloración de bordes.

11.2 Colorear vértice

La coloración de vértice de un gráfico es la asignación de colores a sus vértices, de modo que no haya dos vértices adyacentes que tengan el mismo color. Se puede definir formalmente de la siguiente manera.

Definición 11.1 (coloración de vértice) Coloración de vértice o coloración de una gráfica $G = (V, E)$ es una función de asignación $\phi: V \rightarrow C$ tal que $\forall (u, v) \in E, \phi(u) \neq \phi(v)$, donde C es un conjunto de colores cuyos elementos son comúnmente los elementos de \mathbb{N}^* .

Para una gráfica con n vértices, el conjunto C con n elementos proporcionará su coloración, sin embargo, nuestro objetivo es encontrar el número mínimo de colores en la versión de optimización del problema de coloración de vértice . La versión de decisión de este problema busca encontrar una respuesta a la pregunta. “¿Podemos colorear los vértices de un gráfico con un máximo de k colores?”. Consideraremos gráficos conectados y simples para este problema. La coloración k de un gráfico G es la coloración de G utilizando k colores. En un vértice adecuado para colorear. De un gráfico, los vértices adyacentes están coloreados con colores distintos. Los vértices del mismo color en un gráfico forman una clase de color .

Definición 11.2 (número cromático) El número cromático $\chi(G)$ de un gráfico G es el número mínimo de colores necesarios para colorear correctamente sus vértices.

Hallazgo $\chi(G)$ de G es un problema NP-duro [\[10\]](#). Sin embargo, podemos especificar un límite superior en el valor de este parámetro como veremos.

Observación 9 Cualquier subgrafo H de un gráfico G puede colorearse con menos colores que G , es decir, $\chi(H) \leq \chi(G)$.

Mientras coloreamos G , colorearemos todos sus subgrafos y es probable que usemos menos colores para colorear sus subgrafos.

Observación 10 El número cromático. $\chi(G)$ de una gráfica G con n vértices es n si y solo si $G = K_n$. Es decir, $\chi(K_n) = n$.

Esto es válido ya que todos los vértices están conectados a todos los otros vértices en K_n y por lo tanto, necesitamos n colores distintos para colorear tal gráfico.

Observación 11 El número cromático, $\chi(G)$, de un grafico de estrellas S_n con n vértices es 2, ya que podemos colorear todos los vértices conectados al centro con el mismo color y el centro con otro color.

Hay algunas propiedades interesantes de coloración de vértice de la siguiente manera.

- Un gráfico bipartito no tiene ciclos de longitud impar y, por lo tanto, se puede colorear con 2 colores. Podemos colorear un gráfico bipartito ejecutando el algoritmo BFS de Cap. 7 y colorear los vértices en niveles impares con el color 1 y los vértices en niveles iguales con el color 2.
- Dado que un árbol es un gráfico bipartito, podemos colorear un árbol con dos colores.
- Un gráfico de ciclo con un número par de vértices es un gráfico bipartito y, por lo tanto, podemos colorear dicho gráfico con dos colores Fig. 11.1 a.
- Un gráfico de ciclo con un número impar de vértices no es un gráfico bipartito. Podemos colorear este gráfico con n vértices usando un total de tres colores; dos colores para $\frac{n-1}{2}$ vértices y un tercer color para el n vértice como se muestra en la Fig. 11.1 b.

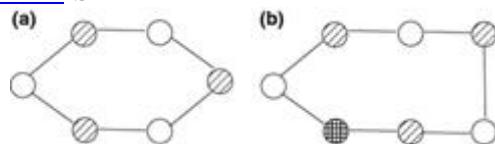


Fig. 11.1 Coloración de gráficos de ciclos pares e impares.

Teorema 11.1 (Teorema de Brook [4]) Para un gráfico G conectado que no está completamente conectado o un ciclo impar,

$$\chi(G) \leq \Delta(G) + 1$$

(11.1)

Lovasz dio una prueba algorítmica breve y simple de este teorema al considerar tres casos [15].

Observación 13 La igualdad se mantiene en sólo dos casos.

- En una gráfica completa, K_n con $n \geq 3$, $\Delta(K_n) = n - 1$ y $\chi(K_n) = n$. Por lo tanto,

$$\Delta(K_n) + 1 = n = \chi(K_n)$$

- Para una gráfica de ciclo impar G , $\chi(G) = 3$ y desde $\Delta(G) = 2$, $\chi(G) = \Delta(G) + 1$

Veremos que el algoritmo codicioso presentado en la siguiente sección también tiene una complejidad como este límite superior. Brook demostró que la igualdad se mantiene solo

para los gráficos de ciclo impar y los gráficos completos. Sin embargo, este límite superior en un número cromático de un gráfico puede resultar estar muy lejos del valor real como en el gráfico de estrellas. $\mathcal{A}(S_n) = n - 1$ y $\chi(S_n) = 2$ en tal grafo.

11.2.1 Relación con conjuntos independientes y camarillas

Un subconjunto I de vértices de una gráfica. $G = (V, E)$ se denomina conjunto independiente si no hay dos vértices en I que sean vecinos y el conjunto independiente máximo (MaxIS) es un conjunto independiente con el orden máximo que describimos en la Secta. [10.2](#). Por otra parte, un conjunto independiente máximo (MIS) es un conjunto independiente que no se puede ampliar agregando vértices. Hemos revisado un método para obtener un MIS de un gráfico de k colores en el Cap. [10](#) y la operación inversa de colorear vértices de una gráfica usando conjuntos independientes también es posible. Observamos que los vértices en un conjunto independiente pueden ser coloreados con el mismo color ya que no son adyacentes. De hecho, usaremos esta propiedad para diseñar algoritmos de coloración de vértices como veremos. Dado un gráfico K -cromático G , podemos dividir los vértices de G en k conjuntos independientes desunidos I_1, \dots, I_k que se llaman clases de color . Cada vértice en la clase de color i entonces puede ser coloreado por i . En otras palabras, podemos encontrar k conjuntos independientes de un gráfico G tal que $\bigcup_{i=1}^k I_i = V$ que no son necesariamente máximos, podemos colorear todos los elementos de cada conjunto con un nuevo color y el número cromático para este gráfico es k .

Observación 14 Si un gráfico G puede dividirse en k conjuntos independientes separados pero no menos, entonces $\chi(G) = k$.

Cada clase de colorante i de G es un conjunto independiente. Por lo tanto,

$$|I_i| \leq \alpha(G)$$

dónde $\alpha(G)$ es el número máximo independencia de G . Ya que

$$|I_1| + |I_2| + \dots + |I_k| \leq k\alpha(G) = \chi(G)\alpha(G)$$

podemos concluir,

$$\chi(G) \geq \lfloor \frac{n}{\alpha(G)} \rfloor$$

(11.2)

A clique de un gráfico G es un subgrafo completa de G . El numero de la camarilla $\omega(G)$ de un gráfico G es el orden de su camarilla más grande. Hay una relación entre una camarilla y un conjunto independiente de una gráfica G , como hemos notado en la Secta. [10.2.1](#) , un subconjunto V' de vértices de $G = (V, E)$ es una camarilla si y solo si V' es un conjunto máximo independiente en \bar{G}

Teorema 11.2 Sea $\omega(G)$ sea el número clique de gráfico G , es decir, que es el orden de la más grande clique de G . Entonces,

$$\chi(G) \geq \omega(G)$$

(11.3)

Prueba Dado que todos los vértices de una camarilla son adyacentes entre sí, cada vértice debe ser coloreado con un color diferente. Por lo tanto, el orden de la camarilla máximo en un gráfico de G establece un límite inferior en el índice cromático de G . \square

11.2.2 Algoritmos secuenciales

Dado que la coloración de los vértices de una gráfica con su número cromático de colores es un problema NP-difícil, en la literatura se proponen varias heurísticas que aproximan la cantidad de colores a $\chi(G)$. En esta sección, primero presentamos una plantilla de algoritmo de colores codiciosos y luego revisamos cuatro algoritmos utilizando diferentes heurísticas, que son el algoritmo aleatorio, el algoritmo de primer ajuste, el algoritmo de primer grado más grande y el algoritmo de ordenación basado en la saturación. Todos estos algoritmos pueden clasificarse como enfoques codiciosos, ya que seleccionan el vértice que mejor cumple los criterios requeridos en cada iteración.

11.2.2.1 Plantilla de algoritmo

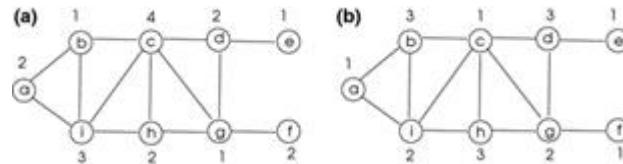
Necesitamos colorear cada vértice de la gráfica con colores que obedezcan al principio de coloración, es decir, a cada vértice se le asigna un color que no entre en conflicto con los colores ya asignados de los vecinos. Formaremos una plantilla de algoritmo sin especificar los criterios de selección como se muestra en el Algoritmo 11.1 y luego discutiremos varias heurísticas para la selección del vértice a colorear. En la forma más simple, un vértice se selecciona al azar y el color más pequeño disponible se asigna a este vértice en la selección aleatoria. La operación de este algoritmo en un gráfico simple se muestra en la Fig. 11.2 a.

Algorithm 11.1 *Coloring_Template*

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi : V \rightarrow C$  where  $C = \{1, 2, \dots, n\}$ 
3:  $V' \leftarrow V$ 
4: while  $V' \neq \emptyset$  do
5:   select a vertex  $v \in V'$  according to some heuristic
6:    $\phi(v) \leftarrow$  the smallest legal color from  $C$ 
7:    $V' \leftarrow V' \setminus \{v\}$ 
8: end while

```



La selección del vértice v en la línea 5 del algoritmo se puede realizar utilizando varias heurísticas de la siguiente manera.

- Algoritmo basado en identificador : en este caso, los vértices de la gráfica se numeran de 1 a n para obtener un conjunto de vértices $V = \{v_1, \dots, v_n\}$ y los vértices están coloreados en secuencia obedeciendo las reglas de coloración; es decir, colorear cada vértice con el mínimo color posible que no entre en conflicto con los vecinos. Este algoritmo también se denomina algoritmo de primer ajuste y se utiliza como máximo $\Delta(G)$ colores en la media [11]. Es simple y rápido en sentido general, pero puede producir una relación de aproximación de $n / 4$ en algunos gráficos especiales [12] que requieren un tiempo de ejecución de $O(m)$.
- Algoritmo de primer grado más grande (LDF) : Tiene sentido colorear los vértices de alto grado primero para tener una menor cantidad de colores, ya que los vértices de bajo grado generalmente se pueden colorear de una manera más flexible como se propone en [20]. El funcionamiento de este algoritmo se muestra en la gráfica de la Fig. 11.2 b, y podemos ver que resulta en un color menos que el enfoque codicioso. Este algoritmo se puede implementar para tener una complejidad de tiempo $O(m)$ (Fig. 11.3).
- Algoritmo de orden de grado de saturación (SDO) : se puede proporcionar un refinamiento adicional al algoritmo LDF de la siguiente manera [3]. El grado de saturación $s(v)$ de un vértice v se define como el número de colores distintos asignados actualmente a sus vecinos. Este parámetro es dinámico y se puede diseñar un algoritmo codicioso basado en los grados de saturación para seleccionar siempre el vértice con el valor más alto de este parámetro. En el caso de los lazos, se selecciona el vértice con el grado más alto, lo que significa que estamos buscando el mayor valor del par $(s(v), \text{grados}(v))$ de todos los vértices $v \in V$. A los que no se les asigna un color. Tenga en cuenta que dicho algoritmo comenzará con el vértice de mayor grado v de la gráfica y le asignará el color mínimo a v , y continuará con el vértice adyacente de mayor grado de v . La operación del algoritmo SDO se muestra en la Fig. 11.3 . Este algoritmo tiene $O(n^2)$ complejidad de tiempo [3].
- Algoritmo de orden de grado de incidente : esta heurística es una forma modificada del SDO. El grado de incidente de un vértice es el número de sus vecinos de color. Tenga en cuenta que los colores de los vecinos no tienen que ser distintos como en la heurística basada en la saturación. El vértice que tiene el grado de incidencia más alto se selecciona en cada iteración del algoritmo [6]. Los identificadores de vértices se utilizan en el caso de vínculos como en el algoritmo de grado de saturación. Es un algoritmo de tiempo lineal que se ejecuta en tiempo $O(m)$.

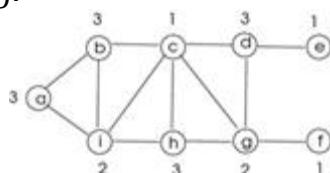


Fig. 11.3 Ordenación basada en la saturación. El orden de los vértices seleccionados es c - g - i - d - b - a - e - f

11.2.2.2 Algoritmos independientes basados en conjuntos

La idea principal de los algoritmos independientes basados en conjuntos es que a los vértices de un conjunto independiente se les puede asignar el mismo color que no están

adyacentes. Por lo tanto, podemos encontrar (máximo) conjuntos independientes de un gráfico G y colorear todos los vértices de este conjunto con un nuevo color, eliminar los vértices del gráfico G y continuar hasta que todos los vértices estén coloreados como se muestra en el algoritmo 11.2. Claramente, el rendimiento de este algoritmo genérico está influenciado por el método utilizado para encontrar los conjuntos independientes.

Algorithm 11.2 $IS_{\text{m}}.Vcolor$

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi : V \rightarrow C$  where  $C = \{1, 2, \dots, k\}$ 
3:  $V' \leftarrow V$ 
4: while  $G' \neq \emptyset$  do
5:    $I \leftarrow \text{Find}_{\text{m}}MIS(G')$ 
6:   color the vertices in  $I$  with a new color
7:    $V' \leftarrow V' \setminus I$ 
8:    $G' \leftarrow \text{graph induced by } V'$ 
9: end while

```

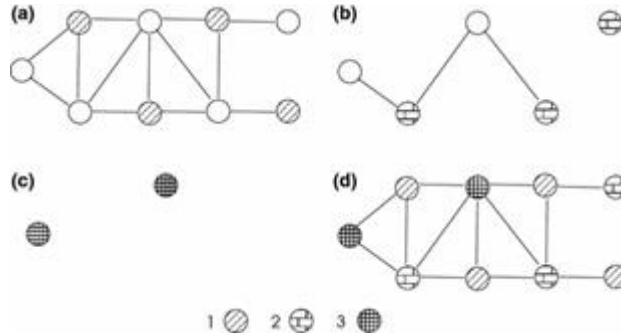


Fig. 11.4 Coloración de vértices basada en el conjunto independiente máximo de la misma gráfica de la Fig. 11.2 ; a, b , yc muestran las iteraciones del algoritmo, donde se asigna un nuevo color a cada nuevo conjunto independiente como se muestra mediante diferentes patrones; D Muestra la coloración final de la gráfica.

11.2.3 Algoritmos paralelos

Solo hay pocos algoritmos para colorear los vértices de una gráfica en paralelo. A continuación, describimos los algoritmos basados en conjuntos independientes y el algoritmo basado en identificadores para este propósito.

11.2.3.1 Algoritmos paralelos independientes basados en conjuntos

Podemos usar el algoritmo de coloreado basado en conjuntos independientes para colorear en paralelo los vértices de una gráfica, ya que podemos realizar la búsqueda de un conjunto independiente máximo en paralelo. El algoritmo 11.3 muestra el código para el algoritmo paralelo para colorear de vértices basados en conjuntos independientes. El código dentro de la mientras bucle se ejecuta en paralelo de forma sincrónica.

Algorithm 11.3 $ParIS_{\text{m}}.Vcolor$

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi : V \rightarrow C$  where  $C = \{1, 2, \dots, k\}$ 
3:  $G' \leftarrow G$ 
4: while  $G' \neq \emptyset$  in parallel do
5:    $I \leftarrow \text{Find}_{\text{m}}MIS(G')$ 
6:   color the vertices in  $I$  with a new color
7:    $V' \leftarrow V' \setminus I$ 
8:    $G' \leftarrow \text{graph induced by } V'$ 
9: end while

```

Hemos visto cómo se puede construir un conjunto independiente en paralelo utilizando el método Monte Carlo de Luby en la Sect. 10.2.3 . En este algoritmo, a los vértices se les asignaron permutaciones aleatorias. $1, \dots, n$ en cada iteración y un vértice con un valor mínimo local se colorearon estos valores se usaron para romper simetrías y decidir los colores de los vértices. Simplemente podemos implementar este algoritmo para este propósito y luego colorear los vértices de la gráfica en consecuencia.

Jones- Algoritmo de Plassmann

En otro enfoque más reciente, Jones y Plassmann presentaron un algoritmo de coloreado de gráficos paralelos basados en conjuntos independientes (JP_Color) [13]. Su enfoque es diferente del algoritmo de Luby, ya que los números aleatorios se asignan solo una vez al principio del algoritmo y no cambian. Además, la asignación de los colores a los vértices de conjuntos independientes se asigna individualmente para cada vértice, es decir, cada vértice en el conjunto se colorea con el color mínimo que no existe en sus vecinos, como se muestra en el algoritmo 11.4. A cada vértice v se le asigna un número aleatorio que es su peso $w(v)$ inicialmente. Si el peso de un vértice es mayor que todos los pesos asignados a sus vecinos, v se asigna al conjunto independiente I con lazos rotos por identificadores de vértice únicos. Este paso se realiza en paralelo y los elementos del conjunto I también se colorean en paralelo con los colores legales. A diferencia del algoritmo 11.3, el conjunto independiente que formó en cada paso no necesita ser MIS como en el método de Luby y los vértices de I pueden estar coloreados con colores diferentes. Jones y Plassmann demostraron que el tiempo de ejecución esperado de este algoritmo en gráficos de grado acotado utilizando el modelo PRAM es $O(\log n / \log \log n)$ [13].

Algoritmo 11.4 JP_Vcolor

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi : V \rightarrow C$  where  $C = \{1, 2, \dots, k\}$ 
3:
4:  $G' \leftarrow G$ 
5: while  $G' \neq \emptyset$  do
6:   for all  $v \in V'$  in parallel do
7:      $I = \{v\}$  such that  $w(v) > w(u), \forall u \in N(v)$ 
8:     for all  $u \in I$  in parallel do
9:       assign  $u$  the minimum color not used by  $N(u)$ 
10:      end for
11:    end for
12:     $V' \leftarrow V' \setminus I$ 
13:     $G' \leftarrow$  graph induced by  $V'$ 
14: end while

```

El algoritmo paralelo de mayor grado primero (PLDF) tiene una estructura similar a JP_Alg con la diferencia de que los pesos asignados son los grados de los vértices y los lazos se rompen al seleccionar un número aleatorio. En [9] se describe un método de coloración de vértices paralelos que utiliza la partición de gráficos . Un estudio experimental reportado en [1] compara el conjunto independiente paralelo, los algoritmos de Jones y Plassmann y LDF para la coloración de vértices paralelos.

11.2.3.2 Cole - Algoritmo de Vishkin

Cole y Vishkin propusieron un método para reducir los colores utilizados en un gráfico en paralelo [5], sin embargo, se puede implementar la misma técnica en un algoritmo distribuido para colorear los nodos de una red. Básicamente, dado un color K apropiado de una gráfica G con un k grande posible , apunta a encontrar una coloración G con un valor k más pequeño , por lo tanto, es una técnica de reducción de color. La idea general de este algoritmo es que inicialmente a los nodos se les asignen etiquetas únicas de $\log n$ pedacitos Luego, las nuevas etiquetas de nodo que son mucho más pequeñas que las anteriores se calculan en cada iteración del algoritmo.

Cada nodo en el gráfico tiene como máximo un sucesor, lo que significa que este algoritmo se puede usar en rutas dirigidas, ciclos dirigidos. El color k se reduce a $\log k$. La coloración en un solo paso y la coloración 6 del gráfico se determinan después de algunas iteraciones. Cada nodo v en paralelo envía su color a su sucesor. Un nodo v recibiendo el color. c_v De su predecesor u , lo compara con su color. c_u y encuentra el bit más a la derecha que es diferente. Luego establece su nuevo color como la concatenación del índice con el

valor en el índice como se muestra en el algoritmo 11.5. Las iteraciones continúan hasta que tengamos $k = 6$, es decir, colores en la gama. Necesitamos otros métodos si se necesita una reducción adicional.

```

0001 0001
0100 1001 0111
0110 1001 1011 100

```

Algorithm 11.5 CV_Vcolor

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi : V \rightarrow C$  where  $C = \{0, 1, 2, 3, 4, 5\}$ 
3:  $G' \leftarrow G$ 
4: for all  $v \in V$  do
5:    $\text{color}(v) \leftarrow v_1$ 
6: end for
7: while  $\exists v_i : \text{color}(v_i) > 5$  in parallel do
8:   assume  $\text{color}(v_j)$  and  $\text{color}(v_{j-1})$  as little-endian bit strings
9:   let  $j$  be the smallest bit index  $x$  they differ
10:   $c \leftarrow j \cup x$ 
11:   $\text{color}(v_i) \leftarrow c$ 
12: end while

```

Lema 11.1 Algoritmo 11.5 encuentra correctamente una coloración legal de un gráfico G .

Prueba La coloración inicial de los vértices es legal. Tenemos que mostrar que los colores recién formados también son legales. Supongamos dos casos.

- Caso 1 : Sucesor del vértice u , vértice v , elige el mismo índice. Dado que los valores de bits son diferentes, tienen diferentes colores asignados.
- Caso 2 : cuando tienen diferentes índices, tienen diferentes colores. □
los $\log^+ n$ Es una función de n . Se define de la siguiente manera.

Definición 11.3 comenzando con n , $\log^+ n$ es el número de veces que se aplica el logaritmo en la base 2 hasta alcanzar un número menor o igual a 2. Es decir, $\log^+ n = \min \{i : \log^i n \leq 2\}$. En otras palabras, $\forall n \leq 2, \log^+ n = 1$ y $\forall n > 2, \log^+ n = 1 + \log^+ (\log n)$. Por ejemplo, $\log^+ 2^{32} = 1 + \log^+ 32 = 2 + \log^+ 16 = 3 + \log^+ 4 = 4$.

El teorema 11.3 El algoritmo 11.5 calcula un color de 6 de una gráfica en $\log^+ n$ hora.

Prueba de dejar n_j Ser el número máximo de bits utilizados por el color. c_v del vértice v después de la iteración j y dejar $n_0 = \lceil \log n \rceil$ Ser el número de bits utilizados para la coloración inicial de los nodos. Podemos afirmar $n_{j+1} \leq \lceil \log n_j \rceil + 1 \leq \log n_j + 2$. Podemos seguir encontrando $n_1 \leq \log n_0 + 2$ y $n_2 \leq \log(\log n_0 + 2) + 2 \leq \log \log n_0 + 3$ cuando $\log n_0 \geq 2$. Podemos ver por $j = 1, 2, \dots$ con $\log^{(j)} n_0 \geq 3, n_j \leq \log^{(j)} n_0 + 3$. Por lo tanto, cuando el número de iteraciones $j = \log^+ n_0, n_j \leq 5$. Ya que $n_0 = \lceil \log n \rceil$, el número de bits para c_v es como máximo 5. El número de bits después de dos iteraciones más se reduce a 3 y, por lo tanto, el número de colores posibles se convierte en 8. Otra iteración hace que el tamaño de la paleta 6 sea la primera parte del color con 3 valores posibles [2]. □

Tenga en cuenta que este algoritmo solo puede reducir los colores a 6 colores. Por ejemplo, supongamos que el nodo u es el antecesor de vértice v y $\phi(v) = 101_B$ y $\phi(u) = 011_B$, el índice es 001_B y el nodo v establecerá 011_B que será el mismo que el valor de color de u .

11.2.4 Colorear vértices distribuidos

En un entorno distribuido, nuestro objetivo es que cada nodo de la red tenga asignado un color legal. Primero presentaremos un algoritmo de reducción de color simple utilizando identificadores de nodos como colores iniciales y luego un algoritmo síncrono que rompe

las simetrías utilizando los identificadores de nodos y un algoritmo para colorear los nodos de un árbol en esta sección.

11.2.4.1 Un algoritmo de reducción sincrónica

Utilizaremos el modelo SSI de computación distribuida con un solo iniciador que inicia rondas síncronas. En este algoritmo, asumimos que cada nodo de la red tiene un identificador único i y está coloreado inicialmente con ese identificador. Nuestro objetivo es reducir la coloración de los nodos en rondas síncronas y utilizar el número de ronda para este propósito. Las rondas están numeradas de $\Delta+2$ a m , ya que sabemos que una coloración legal requiere un número cromático menor que $\Delta+1$ por el teorema de Brook y, por lo tanto, queremos reducir cualquier número de coloreado inicial mayor que este valor. Cualquier nodo i que encuentre que tiene un color mayor que el número redondo cambia su color al color más pequeño que no utilizan sus vecinos e informa a los vecinos de esta elección como se muestra en el algoritmo 11.6. Tenga en cuenta que los identificadores únicos se utilizan para seleccionar el nodo en ejecución y, por lo tanto, solo habrá un nodo que ejecute el código en cualquier momento. Este algoritmo es simple y no requiere una técnica de ruptura de simetría, pero su funcionamiento es inherentemente secuencial.

La figura 11.5 muestra la ejecución de este algoritmo simple en una red pequeña. Tenga en cuenta que la operación es secuencial y el número de colores utilizados es exactamente $\Delta+1$ para $n > \Delta+1$ en este algoritmo.

Algorithm 11.6 Dist_Vcoll

```

1: Input: neighbor list  $N(i)$ 
2: Output: color  $my\_color$  of node  $i$  in the network
3: for  $r = \Delta + 2$  to  $n$  do
4:   if  $my\_color = k$  then
5:      $my\_color \leftarrow$  first free color not used by neighbors
6:     inform neighbors of my color by the status message
7:   else if status message received from a neighbor then
8:     update available color list
9:   end if
10: end for

```

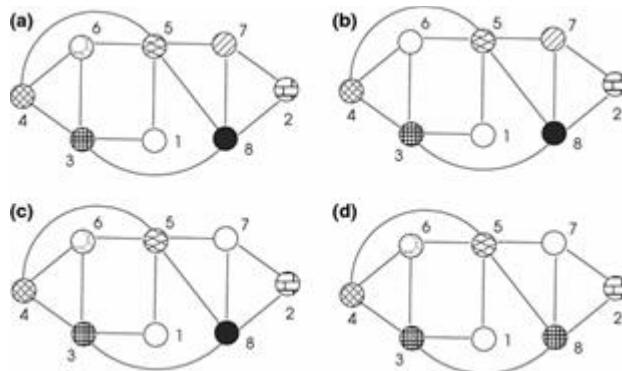


Fig. 11.5 Coloración distribuida de un gráfico de muestra utilizando el algoritmo 11.6. Las rondas 6, 7 y 8 proporcionan un colorante legal con 8 ($\Delta+1$) colores

El teorema 11.4 El algoritmo 11.6 proporciona la coloración legal de una red con $\Delta+1$ colores en $n - \Delta + 1$ tiempo usando $O(dn)$ mensajes

Prueba Dado que la coloración inicial es legal y los nodos de cambio de color realizan la coloración legal, es decir; Seleccionando un color no utilizado por vecinos, la coloración final es legal y utiliza $\Delta+1$ colores exactamente para $n > \Delta+1$ en $n - \Delta + 1$ rondas. El único mensaje enviado en una ronda es el nodo i que tiene un identificador igual al número de ronda y,

por lo tanto, esto sería $O(d)$ mensajes. El número total de mensajes intercambiados será entonces $O(dn)$. □

11.2.4.2 Un algoritmo sincrónico basado en rangos

Usamos el modelo SSI nuevamente por su simplicidad en este algoritmo. El identificador único de nodos se utiliza para asignar prioridades en este momento para romper simetrías. La idea general de este algoritmo es dar prioridad a los nodos que tienen el identificador más alto (o más bajo). En cada ronda, un nodo incoloro que encuentra que tiene el identificador más alto entre sus vecinos incoloros se asigna un color legal a sí mismo e informa su decisión a todos sus vecinos. Una forma de implementar este algoritmo para un nodo i en cada ronda se muestra en detalle con las estructuras de datos necesarias en el algoritmo 11.7.

Algorithm 11.7 Dist_V.col2

```

1: Input: neighbor list  $N(i)$ 
2: Output: color  $my\_color$  of node  $i$  in the network
3: message types:  $decided(x, col)$ ,  $undecided(x)$  → sent by node  $x$  coloring itself by  $col$  and a
   message by undecided node
4: states: {COLORED, UNCOLORED}           → states of a node as colored or uncolored
5:  $neigh\_colors \leftarrow \emptyset$ 
6:  $my\_state \leftarrow \text{UNCOLORED}$ 
7: send  $my\_id$  to  $N(i)$ 
8: receive  $ids$  of neighbors
9: while  $my\_state = \text{UNCOLORED}$  do
10:   if  $my\_id > id$  of all active neighbors then
11:      $color \leftarrow$  first free color  $c \notin neigh\_colors$ 
12:      $state \leftarrow \text{colored}$ 
13:     send  $decided(i, c)$  to  $N(i)$ 
14:   else if  $decided(j, col)$  received from neighbor  $j$  then
15:      $N(i) \leftarrow N(i) \setminus \{j\}$            → remove neighbor  $j$  from active neighbor list
16:      $neigh\_colors \leftarrow neigh\_colors \cup \{col\}$ 
17:   end if
18: end while

```

El funcionamiento de este algoritmo se representa en la Fig. 11.6 para la misma gráfica de muestra de la Fig. 11.5. Este gráfico está coloreado con tres colores legales en cuatro rondas. Podemos aplicar un criterio diferente, como los grados de vértices o un número aleatorio seleccionado entre 0 y 1 para romper simetrías en lugar de identificadores de vértices que resultan básicamente en un algoritmo estructurado de manera muy similar.

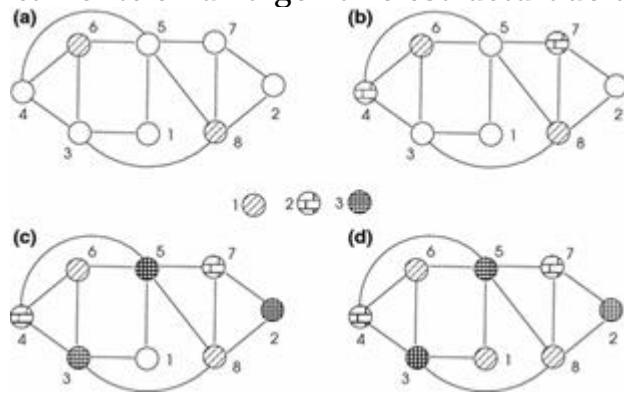


Fig. 11.6 Coloración distribuida de un gráfico de muestra. Algoritmo 11.7. En cuatro rondas, tres colores asignados a ocho nodos.

El teorema 11.5 El algoritmo 11.7 colorea correctamente los nodos de una red en tiempo $O(n)$ con $O(d + 1)$ colores.

La corrección de la prueba es evidente ya que la regla de coloración se aplica en cada paso. Como en otros algoritmos distribuidos codiciosos basados en rangos que hemos revisado, la cantidad de rondas requeridas puede ser tan alta como la cantidad de

vértices n como en el caso de una red con vecinos identificadores ordenados. El número máximo de colores utilizados será $d+1$ ya que siempre habrá un color libre en esa gama. □

11.2.4.3 Árbol para colorear

Una red que está configurada como un árbol arraigado puede colorearse usando solo dos colores. La raíz inicia el algoritmo al colorearse con el color 0 y enviando su color a sus hijos que se tiñen con el otro color. Continuando de esta manera, dos colores son suficientes para colorear todo el árbol. Los pasos de este algoritmo para el nodo i son los siguientes.

Si $i = \text{raíz}$ entonces

- 1.
2. $c_i \leftarrow 0$

3. enviar color (1) a niños
4. de lo contrario recibe color (c)
5. $c_i \leftarrow 1 - c$
6. Si $i \neq \text{leaf}$ entonces
7. enviar color (c_i) para niños

Cada nodo debe saber si es un nodo raíz , intermedio o hoja en este algoritmo. La figura 11.7 muestra un árbol construido usando el procedimiento anterior. Podemos ejecutar este algoritmo en rondas síncronas en el modelo SSI o de forma asíncrona. El tiempo de ejecución de los algoritmos es la profundidad d del árbol $O(d)$ en ambos casos. El número total de mensajes intercambiados en ambos casos es el número de bordes del árbol que es $n - 1$. El algoritmo propuesto por Cole y Vishkin se puede utilizar para colorear árboles de forma distribuida.

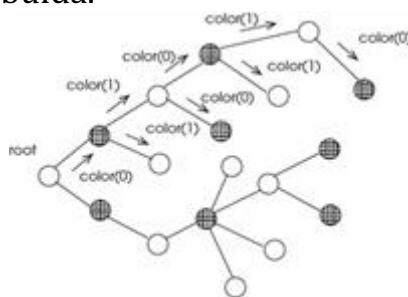


Fig. 11.7 Coloración distribuida de un árbol de muestra con dos colores.

11.3 Colorear los bordes

En la coloración de bordes de un gráfico, a los bordes se les asignan colores de manera que los bordes adyacentes tengan colores diferentes.

Definición 11.4 (coloración de bordes) Coloración de bordes de un gráfico $G = (V, E)$ es una tarea $\phi' : E \rightarrow C$ tal que cualquiera de los dos bordes e_i y e_j que son incidentes en el mismo

vértice tienen $\phi'(e_i) \neq \phi'(e_j)$, donde C es un conjunto de colores cuyos elementos son comúnmente los elementos de \mathbb{N}^* .

En la coloración adecuada de los bordes, los bordes adyacentes tienen asignados colores distintos. El problema de la coloración de bordes es encontrar el número mínimo de colores para colorear los bordes de un gráfico. En la versión de decisión de este problema, tratamos de encontrar una respuesta a si los bordes de un gráfico se pueden colorear con un máximo de k colores diferentes. Se dice que un gráfico es k -edge coloreable si hay un colorante $\phi : E \rightarrow C$ tal que $|C| = k$. El número cromático del borde o el índice cromático $\chi'(G)$ de un gráfico G es el valor mínimo de k tal que G es k -edge coloreable. En otras palabras, G es k -edge cromático si $\chi'(G) = k$.

Teorema 11.6 Para cualquier gráfica simple G ,

$$\chi'(G) \geq \Delta(G)$$

(11.4)

Prueba Let v ser el vértice con el máximo grado de G . Todos los bordes incidentes a v deben ser coloreados con un color diferente, por lo tanto $\chi'(G)$ debe ser al menos igual a $\Delta(G)$.

Existen fuertes límites inferiores y superiores para la coloración de bordes de gráficos. Vizing ha demostrado que para cada gráfico simple no vacío G sin bordes múltiples y sin bucles [19],

$$\chi'(G) \leq 1 + \Delta(G)$$

(11.5)

Basado en las ecuaciones . 11.4 y 11.5 ,

$$\Delta \leq \chi'(G) \leq 1 + \Delta(G),$$

(11.6)

lo que significa para cada gráfico simple no vacío G , ya sea $\chi'(G) = \Delta(G)$ o $\chi'(G) = 1 + \Delta(G)$. Los sencillos gráficos de G donde $\chi'(G) = \Delta(G)$ son llamados gráficos de clase 1 y los gráficos que tienen $\chi'(G) = \Delta(G) + 1$ Se llaman gráficos de clase 2 .

Observación 16 Una gráfica de estrellas. S_n tiene $n - 1$ bordes Dado que todos estos bordes son adyacentes entre sí, $\chi(S_n) = n - 1$. Por lo tanto, S_n Es un gráfico de clase 1.

Observación 17 Un gráfico de ciclo par puede ser de color de borde con dos colores. Por otro lado, un gráfico de ciclo impar necesita tres colores, ya que habrá al menos un borde

que necesita un tercer color, como se muestra en la Fig. 11.8 . Por lo tanto, los gráficos de ciclo par son gráficos de Clase 1 y los gráficos de ciclo impar son gráficos de Clase 2 .

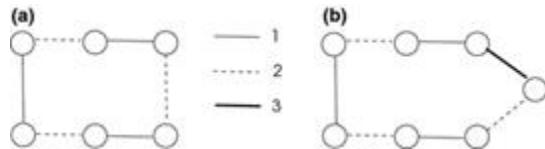


Fig. 11.8 Coloración de bordes de gráficos de ciclos pares e impares

11.3.1 Relación con la concordancia de gráficos

En una coloración de bordes adecuada de un gráfico G , los bordes del mismo color constituyen una coincidencia en G ya que estos bordes no son adyacentes entre sí. En otras palabras, dados los dos bordes (u, v) y (w, y) de una coincidencia, u o v no pueden ser vecinos de w o y . Podemos, por lo tanto, definir la coloración de bordes EC de un gráfico G como la unión de un conjunto de emparejamientos desunidos

$M_1 \cup M_2 \cup \dots \cup M_k$ de G . Para una gráfica G con m aristas y un tamaño máximo coincidente $\alpha'(G)$, cada colorante de borde de G debe usar al menos $\frac{m}{\alpha'(G)}$ colores.

Teorema 11.7 Para una gráfica $G = (V, E)$ con talla m mayor que 1,

$$\chi'(G) \geq \frac{m}{\alpha'(G)}$$

(11.7)

Prueba Deje que nosotros suponemos $\chi'(G) = k$ y E_1, E_2, \dots, E_k son las clases de color de G . $\forall E_i \in E$, $1 \leq i \leq k$; $|E_i| \leq \alpha'(G)$ ya que los bordes en cada color constituyen una coincidencia de G y $\alpha'(G)$ es el tamaño de la coincidencia de máximo de G . Por lo tanto,

$$m = |E(G)| = \sum_{i=1}^k |E_i| \leq k\alpha'(G)$$

(11.8)

por lo tanto, $\chi'(G) \geq \frac{m}{\alpha'(G)}$. \square

Un posible método para encontrar EC es encontrar una coincidencia máxima M_i del gráfico G y coloréelo con un nuevo color, elimine todos los bordes de M_i desde G y repita este proceso hasta que todos los bordes estén coloreados como se muestra en el algoritmo 11.8. Tenga en cuenta que encontramos un color no utilizado simplemente incrementando el índice i .

Algorithm 11.8 MM_Ecolor

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi' : E \rightarrow C$  where  $C = \{1, 2, \dots, k\}$ 
3:  $G' (V', E') \leftarrow G(V, E)$ 
4:  $i \leftarrow 1$ 
5: while  $E' \neq \emptyset$  do
6:    $M_i \leftarrow \text{Find\_MM}(G')$ 
7:   color the edges in  $M_i$  with  $i$ 
8:    $E' \leftarrow E' \setminus M_i$ 
9:    $i \leftarrow i + 1$ 
10: end while

```

El rendimiento de este algoritmo depende claramente de cómo se realiza la búsqueda de la coincidencia máxima. La operación de este algoritmo se representa en la Fig. 11.9, donde los bordes MM de un gráfico de muestra encontrado en cada paso se colorean con un nuevo color.

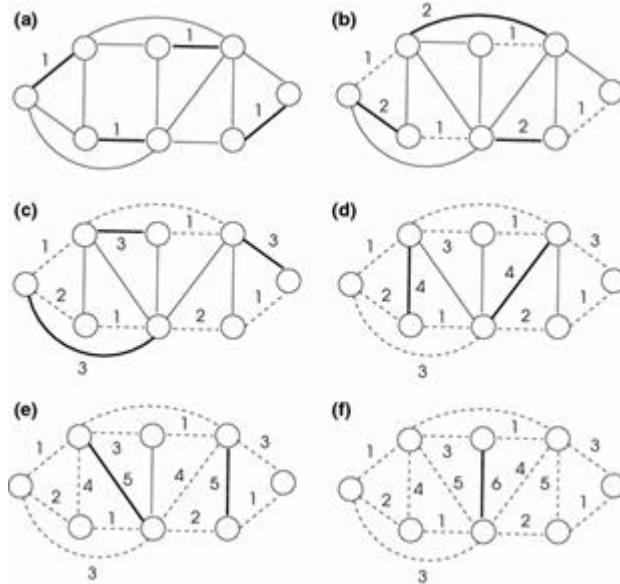


Fig. 11.9 Operación del algoritmo 11.9 en una pequeña muestra gráfica. Los bordes seleccionados en cada iteración se muestran en negrita con un número de color asignado al lado de los bordes. Hemos coloreado los bordes de este gráfico con $d+1=6$ colores

11.3.2 Un algoritmo secuencial codicioso

El codicioso algoritmo para colorear bordes de un gráfico puede dibujarse de forma similar a los codiciosos algoritmos que hemos visto. Un borde eincoloro se selecciona aleatoriamente y se colorea con el color legal mínimo que no entre en conflicto con los colores asignados de los bordes adyacentes del borde e . Este borde se elimina del gráfico y el proceso se repite hasta que no queden más bordes sin color. El algoritmo 11.9 muestra el pseudocódigo para este algoritmo.

Algorithm 11.9 Greedy_Ecolor

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi' : E \rightarrow C$  where  $C = \{1, 2, \dots, 2d - 1\}$ 
3:  $E' \leftarrow E$ 
4: while  $E' \neq \emptyset$  do
5:   select an edge  $(u, v) \in E'$ 
6:    $\phi'((u, v)) \leftarrow$  the smallest legal color from  $C$ 
7:    $E' \leftarrow E' \setminus \{(u, v)\}$ 
8: end while

```

Los colores requeridos para este algoritmo pueden ser tan altos como $2d-1$ como se muestra en la figura 11.10 , donde dos vértices con d Los grados están conectados por un borde que queremos colorear. El único color disponible en este caso, es $2d-1$.

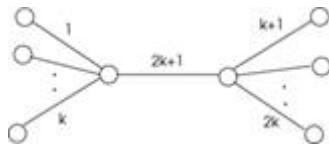


Fig. 11.10 Dos vértices con los grados están coloreados con todos los colores disponibles y el único color restante para el borde entre ellos es $2d - 1$

La operación de este algoritmo en un gráfico de muestra simple se muestra en la figura 11.11. El mientas bucle se ejecuta m veces para colorear todos los bordes. También hay tiempo necesario para encontrar un color disponible para cada vértice, lo que resulta en complejidad del tiempo $O(dm)$.

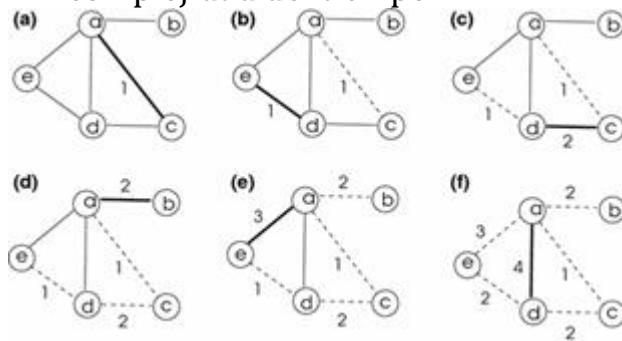


Fig. 11.11 Operación del algoritmo 11.9 en una pequeña muestra gráfica. Los bordes seleccionados en cada iteración se muestran en negrita con un número de color asignado al lado de los bordes. Podríamos colorear este gráfico con 4 colores

El borde-coloración de un gráfico G es equivalente al vértice-coloración del gráfico de línea $L(G)$ de G . Dado que para el grado máximo de G , el grado máximo de $L(G)$ es $2d-2$ lo que significa que podemos colorear los vértices de $L(G)$ usando $2d-1$ colores utilizando el teorema de Brook. Por lo tanto, G puede ser de color borde con $2d-1$ colores ya que cada borde de G corresponde a un vértice en $L(G)$.

11.3.3 Colorear Gráficos Bipartitos

Los bordes de los gráficos bipartitos se pueden colorear con d colores probados por König [14] que significa,

$$\chi'(G) = \Delta(G)$$

(11.9)

Cada gráfico bipartito G está contenido en una $\Delta(G)$ -gráfico regular. Podemos añadir vértices y aristas a un gráfico bipartito. $G = (V_1, V_2, E)$ para hacerlo un d -gráfico bipartito regular G' con $V_1 = V_2$. Tal gráfico formado siempre tiene una coincidencia perfecta. Entonces, podemos encontrar una combinación perfecta. M_1 (1-factor) tal que cada vértice de G' es incidente a un borde de M_1 y colorea esta combinación perfecta con el primer color disponible. Removiendo M_1 de G y encontrando otra pareja perfecta M_2 del nuevo gráfico y continuando de esta manera, podemos tener d Colores para los bordes de G . Este método motiva un algoritmo para colorear los bordes de un gráfico bipartito como se muestra en el algoritmo 11.10.

Algorithm 11.10 Bipartite_Ecolor

```

1: Input :  $G = (V_1, V_2, E)$ 
2: Output :  $\phi' : E \rightarrow C$  where  $C = \{1, 2, \dots, d\}$ 
3:  $E' \leftarrow E$ 
4:  $G' \leftarrow G \cup \{\text{added edges}\} \cup \{\text{added vertices}\}$ 
5: for  $i = 1$  to  $d(G)$  do
6:   find a perfect matching  $M$  of  $G'$ 
7:    $\forall (u, v) \in M: \phi'((u, v)) \leftarrow i$ 
8:    $G' \leftarrow G' - \{M\}$ 
9: end for
10: remove added edges from  $G'$  to get edge colored  $G$ 

```

Encontrar una coincidencia perfecta en un gráfico bipartito regular k lleva tiempo $O(km)$ (vea el Capítulo 9), por lo tanto, el algoritmo 11.10 proporciona la coloración de bordes de un gráfico bipartito con un grado máximo d en $O(dm)$ hora. La ejecución de este algoritmo simple en un pequeño gráfico bipartito se muestra en la figura 11.12 .

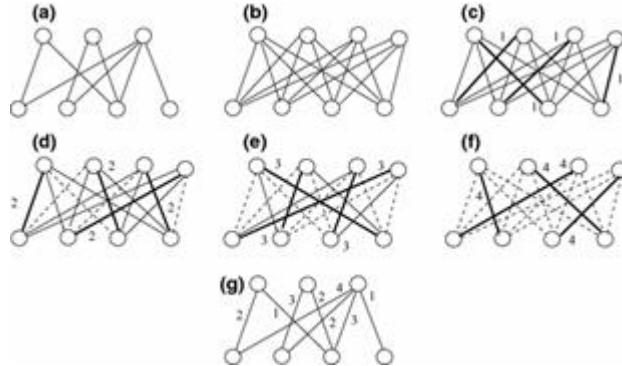


Fig. 11.12 Colorear los bordes de un gráfico bipartito. La gráfica en a se hace completamente bipartita agregando bordes para obtener la gráfica en b . Después de tres iteraciones de combinaciones desunidas perfectas, el gráfico original es de color de borde con $d=4$ colores como se muestra en g

11.3.4 Coloración de bordes de gráficos completos

Un gráfico completo K_n se puede colorear con $n-1$ colores si n es par, de lo contrario, n son necesarios cuando n es impar. Podemos encontrar la coloración de los bordes de un gráfico completo, K_n mediante la selección iterativa de los emparejamientos máximos desunidos de la misma como se muestra en la figura 11.13 . La coloración de los bordes de gráficos completos pares e impares se muestra en la Fig. 11.14 .

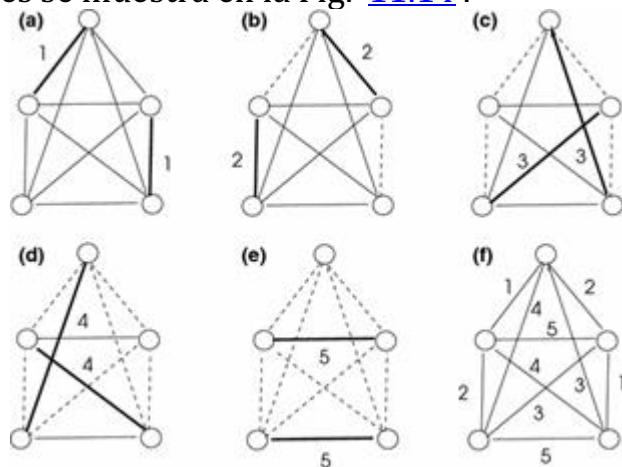


Fig. 11.13 Coloración del borde de K_5 . Las coincidencias máximas disjuntas se seleccionan en cada iteración con cada coincidencia asignada un nuevo color. El gráfico de color final que se muestra en e tiene 5 colores distintos

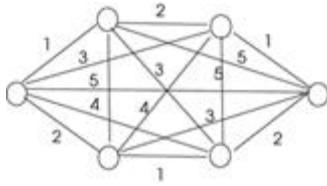


Fig. 11.14 Coloración de K_6 . 5 colores son suficientes para colorear este grafico.

11.3.5 Un algoritmo paralelo

Los bordes del mismo color en un gráfico G correctamente coloreado establecen una coincidencia máxima de G como hemos visto. Hemos revisado en el cap. 9 cómo realizar una comparación paralela de un gráfico y, por lo tanto, podemos usar estos algoritmos para realizar la coloración de bordes paralelos. El algoritmo 11.11 muestra el pseudocódigo de dicho algoritmo que realiza la búsqueda de MM y elimina estos bordes de la gráfica además de asignar colores a los bordes de la coincidencia máxima.

Algorithm 11.11 Par_Ecolor

```

1: Input :  $G = (V, E)$ 
2: Output :  $\phi' : E \rightarrow C$  where  $C = \{1, 2, \dots, k\}$ 
3:  $G' (V', E') \leftarrow G(V, E)$ 
4: while  $E' \neq \emptyset$  in parallel do
5:    $M_i \leftarrow \text{Find\_MM}(G')$ 
6:   color the edges in  $M_i$  with a new color
7:    $E' \leftarrow E' \setminus M_i$ 
8: end while

```

11.3.6 Un algoritmo distribuido

En una configuración de red, cada nodo debe actuar independientemente para colorear los bordes adyacentes a él. Necesitamos una forma de romper las simetrías en un entorno distribuido como implementamos comúnmente. Definamos una heurística para que el nodo con grado máximo decida qué color se asignará a su borde adyacente. A diferencia del problema de coloración de vértices distribuidos, debemos considerar el caso cuando un nodo con un grado menor recibe solicitudes de dos vecinos de grado máximo. En tal caso, asumiremos que selecciona el que tiene el grado mayor. Los mensajes que se utilizarán en el algoritmo que proponemos son los siguientes.

- Grado (i, deg (i)) : Se envía por el nodo i para informar a los vecinos de su grado grados (i).
- Proponer (i, col) : enviado por el nodo i para solicitar la coloración del borde (i, j) adyacente a su vecino j .
- Ack (i), nack (i) : enviado por el nodo i para aceptar el mensaje y rechazar el mensaje de solicitud entrante
- Decidir (i, col) : enviado por el nodo i al nodo j confirma el color del borde del borde (i, j) con el color col .

Suponemos que cada nodo es consciente de sus vecinos y, por lo tanto, su grado inicialmente. La lista de variables $curr_neighs$ contiene los identificadores de los vecinos actualmente activos de un nodo i tal que $\forall j \in curr_neighs$, el borde (i, j) aún no está coloreado. El algoritmo comienza cuando cada nodo cambia su grado con los vecinos. A partir de entonces, siempre que haya bordes incolores adyacentes a un nodo i , verifica su grado con los vecinos actuales. Si encuentra que tiene el grado máximo entre ellos, entonces el nodo i transmite un mensaje de propuesta a todos sus vecinos activos. Un vecino que recibe más de un mensaje de solicitud responde al remitente con el mayor grado por

el mensaje de confirmación , como se muestra en un esbozo de este algoritmo en el algoritmo 11.12.

Algorithm 11.12 Dist_Ecol

```

1: Input: neighbor list  $N(i)$ 
2: Output: set of edge colors  $edge\_cols$  of node  $i$  in the network
3: message types:  $degree(x), deg(x), propose(x, col), ack(x), nack(x), decide(x, col(x))$ 
4:  $curr\_neighs \leftarrow N(i)$  ▷ active neighbors initialized
5:  $neigh\_colors \leftarrow \emptyset$ 
6: send  $degree(deg(i))$  to  $N(i)$  ▷ exchange degree values with neighbors
7: receive  $degree(deg(j))$  from all  $j \in N(i)$ 
8: while  $curr\_neighs \neq \emptyset$  do
9:   if  $deg(i) > deg(j) \forall j \in curr\_neighs$  then
10:    for all  $j \in curr\_neighs$  do
11:      send minimum available color  $c$  in  $neigh\_cols$  to  $j$  in  $propose(i, c)$ 
12:    end for
13:    receive  $ack/nack$  messages from all nodes in  $curr\_neighs$ 
14:    update  $neigh\_cols$ 
15:    update  $curr\_deg$ 
16:    select one of the senders ( $j$ ) of  $ack$  messages arbitrarily
17:    select first available color  $c \notin edge\_cols$ 
18:    send  $decide(i, c)$  to  $j$ 
19:   else if  $propose$  received from neighbor  $j$  or more than one neighbor then
20:     select the highest degree requesting neighbor  $j$ 
21:     send  $ack(i)$  to  $j$ 
22:     send  $nack(i)$  to all others
23:     receive  $decide(j, c)$ 
24:   end if
25:    $neigh\_colors \leftarrow neigh\_cols \cup \{c\}$ 
26:    $curr\_neighs \leftarrow curr\_neighs \setminus \{j\}$ 
27:   broadcast  $info(curr\_deg, neigh\_cols)$  to  $curr\_neighs$ 
28: end while

```

La operación de este algoritmo en una pequeña red de muestra se representa en la figura 11.15 , donde los nodos con grados más altos en sus vecindarios deciden colorear los bordes incidentes para ellos al proponerlos a sus vecinos. Los colores máximos utilizados serán $\frac{2d(G) - 1}{1 + \varepsilon}$ ya que siempre habrá un color en este rango propuesto por los nodos de mayor grado. Grable et al. propuso un algoritmo de coloreado de borde distribuido aleatorio sincrónico en el que cada borde (u, v) elige un color aleatorio c de su paleta de $(1 + \varepsilon) \max \{deg(u), deg(v)\}$ Colores en cada ronda [10]. Si el color c no está en conflicto con alguno de los colores seleccionados de los vecinos, se determina que c es el color del borde (u, v). Se muestra este algoritmo colorea los bordes de un gráfico con $(1 + \varepsilon)d(G)$ colores para cualquier $\varepsilon > 0$ en $O(\log \log n)$ Rondas para gráficos con grados suficientemente grandes.

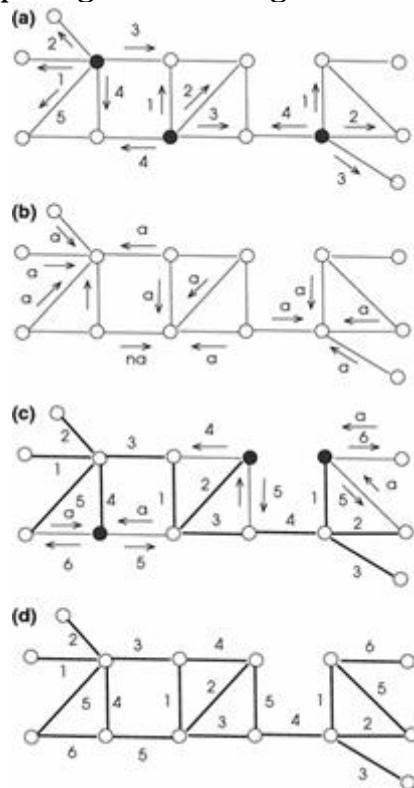


Fig. 11.15 Coloración de bordes de una red pequeña por Dist_Ecol . Los nodos negros tienen el máximo grado de corriente entre sus vecinos. Proponen colores de borde para sus bordes incoloros de sus vecinos que responden mediante mensajes ack (a) o nack (na). Los bordes de colores se muestran en negrita. No se muestra la transmisión de mensajes de información . La coloración final requiere 6 colores.

11.4 Notas de capítulo

La coloración de vértices es el proceso de colorear los vértices de un gráfico de manera que no haya dos vértices adyacentes que tengan el mismo color. Puede formar la base de algoritmos gráficos más complicados y también tiene varias implementaciones prácticas, como la asignación de frecuencias de canal en redes inalámbricas. Revisamos los aspectos teóricos de la coloración de vértices y luego describimos los algoritmos secuenciales, paralelos y distribuidos para este propósito. Colorear los vértices de una gráfica con su número cromático es NP-duro pero por lo demás se colorea con $\Delta + 1$ colores es sencillo.

Un algoritmo paralelo simple hace uso de la propiedad de conjunto independiente donde no hay dos vértices en dicho conjunto adyacentes. Por lo tanto, podemos asignar el mismo color a los vértices de un conjunto independiente. Podemos encontrar un conjunto independiente en paralelo, como se muestra en el capítulo. [10](#) , por lo tanto, este algoritmo puede usarse para asignar colores en paralelo. Hay varios algoritmos de coloración de vértices distribuidos. Describimos un algoritmo básico basado en rangos que es lento, ya que su tiempo de ejecución en rondas depende de la cantidad de nodos en la red y otro algoritmo simple para colorear árboles.

La coloración de bordes se refiere a asignar colores a los bordes de un gráfico, de manera que no haya dos bordes con puntos finales comunes que reciban el mismo color. Describimos un algoritmo secuencial simple y algoritmos paralelos y distribuidos para la coloración de bordes. La coloración de bordes puede hacer uso de algoritmos de coincidencia, ya que los bordes de la misma clase en un gráfico, es decir, bordes con el mismo color, constituyen una coincidencia en ese gráfico. De esta manera, podemos realizar la coloración de bordes en paralelo utilizando un algoritmo de coincidencia máxima paralela mientras revisamos. Los algoritmos distribuidos para colorear bordes requieren una cuidadosa consideración, ya que necesitamos verificar los colores de bordes incidentes a los vecinos de un nodo v cuando se colorea un borde incidental a v . La coloración total del gráfico requiere que se coloren los vértices y los bordes de una gráfica.

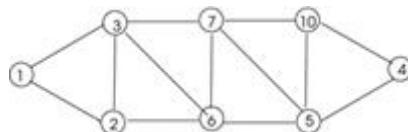


Fig. 11.16 Gráfico de muestra para el Ejercicio 1

Ceremonias

Colorea los vértices de la gráfica en la figura [11.16](#) usando el algoritmo codicioso que selecciona el primer índice más bajo incoloro.

- 1.
2. Particione el gráfico de muestra de la figura [11.17](#) en todos los conjuntos independientes máximos separados posibles y coloree cada conjunto con un nuevo color para encontrar la coloración de vértice de este gráfico.

3. Encuentre el color del vértice de la gráfica que se muestra en la Fig. [11.18](#) utilizando el método de algoritmo distribuido de codicia 11.6. Asignar identificadores a los nodos arbitrariamente.
4. Colorea los bordes del gráfico bipartito que se muestra en la Fig. [11.19](#) formando primero un gráfico bipartito regular y luego encontrar emparejamientos máximos desunidos de este gráfico y colorear los bordes de cada emparejamiento con un nuevo color.
5. Utilice el método de emparejamiento máximo desunido para colorear los bordes del gráfico representado en la Fig. [11.20](#).
6. Calcule la coloración de los bordes de la gráfica que se muestra en la figura [11.21](#) utilizando el método distribuido codicioso del algoritmo 11.12.

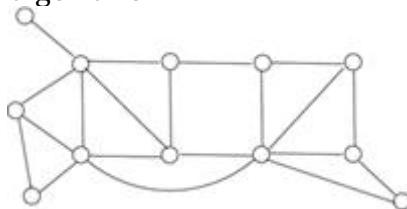


Fig. 11.17 Gráfico de muestra para el ejercicio 2

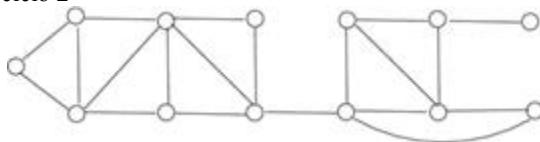


Fig. 11.18 Gráfico de muestra para el Ejercicio 3

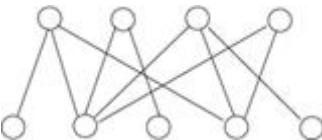


Fig. 11.19 Ejemplo de gráfica para el Ejercicio 4



Fig. 11.20 Gráfico de muestra para el Ejercicio 5

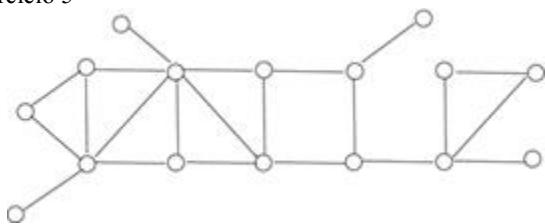


Fig. 11.21 Gráfico de muestra para el Ejercicio 6

Referencias

1. Allwright JR, Bordawekar R, Coddington PD, Dincer K, Martin CL (1995) Una comparación de algoritmos de coloración de gráficos paralelos. Informe técnico SCCS-666, Northeast Parallel Architecture Center, Syracuse University
2. Barenboim L, Elkin M (2013) Coloración gráfica distribuida. Monografía, Universidad Ben Gurion del Negev
3. Brelaz D (1979) Nuevos métodos para colorear los vértices de una gráfica. Commun ACM 22 (4): 251–256
[MathSciNet Crossref](#)

4. Brooks RL (1941) Sobre la coloración de los nodos de una red. Proc Camb Philos Soc Math Phys Sci 37: 194–197
[MathSciNet Crossref](#)
5. Cole R, Vishkin U (1986) Lanzamiento de moneda determinista con aplicaciones para una clasificación óptima de listas paralelas. Inf Control 70 (1): 32–53
[MathSciNet Crossref](#)
6. Coleman TF, More JJ (1983) Estimación de matrices jacobianas dispersas y problemas de coloración gráfica. SIAM J Numer Anal 20 (1): 187–209
[MathSciNet Crossref](#)
7. Gandham S, Dawande M, Prakash R (2005) Programación de enlaces en redes de sensores: colores de bordes distribuidos revisados. En: Actas de la 24^a INFOCOM, vol. 4, pp 2492–2501
8. Garey MR, Johnson DS (1979) Computadoras e intratabilidad. WH Freeman, Nueva York
[MATES](#)
9. Gebremedhin AH (1999) Coloración gráfica paralela. Tesis de maestría, Departamento de Informática de la Universidad de Bergen Noruega
10. Grable D, Panconesi A (1997) Coloración de bordes distribuida casi óptima en $O(\log \log n)$ rondas. Azar Struct Algoritmos 10 (3): 385–405
[MathSciNet Crossref](#)
11. Grimmett GR, McDiarmid CJH (1975) Sobre la coloración de gráficos aleatorios. Matemáticas Proc Camb Philos Soc 77: 313–324
[Referencia cruzada](#)
12. Halldorsson MM (1991) Métodos frugales para los problemas de coloración de conjuntos independientes y gráficos. Doctor en Filosofía. tesis, Universidad Estatal de Nueva Jersey, New Brunswick, Nueva Jersey, octubre de 1991
13. Jones MT, Plassmann PE (1993) Un gráfico paralelo para colorear heurística. SIAM J Sci Comput 14 (3): 654–669
[MathSciNet Crossref](#)
14. König D (1931) Graphen und Matrizen . Matemáticas Lapok 38: 116–119
[MATES](#)
15. Lovasz L (1975) Tres pruebas cortas en teoría de grafos. J Comb Theory Ser B 19: 269-271
[MathSciNet Crossref](#)
- dieciséis. Luby M (1986) Un algoritmo paralelo simple para el problema del conjunto máximo independiente. SIAM J Comput 15 (4): 1036-1055
[MathSciNet Crossref](#)
17. Matula DW, Marble G, Isaacson JD (1972) Algoritmos de coloreado de gráficos. Prensa Académica, Nueva York
[MATES](#)
18. Nishizeki T, Terada O, Leven D (1983) Algoritmos para la coloración de bordes de gráficos. Universidad de Tohoku, Departamento de Comunicaciones Eléctricas, informe técnico, TRECIS 83001
19. Vizing VG (1964) Sobre una estimación de la clase cromática de un p-gráfico. Diskret Anal 3: 25–30 (en ruso)
[MathSciNet](#)
20. Welsh DJA, Powell MB (1967) Un límite superior para el número cromático de una gráfica y su aplicación a problemas de tiempo. Comput J 10: 85–86
[Referencia cruzada](#)

Parte III

Temas avanzados

12. Algoritmos de gráficos algebraicos y dinámicos

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

La teoría de gráficos algebraicos es el estudio de métodos algebraicos para resolver problemas de gráficos. Revisamos soluciones algebraicas a los principales problemas de gráficos en la primera parte de este capítulo. Muchas redes de la vida real están representadas por gráficos dinámicos en los que se pueden insertar nuevos vértices / bordes y algunos vértices / bordes se pueden eliminar a medida que avanza el tiempo. Describimos algunos problemas de gráficos dinámicos que pueden resolverse mediante algoritmos de gráficos dinámicos, y finalmente damos una breve descripción de los métodos utilizados en los algoritmos de gráficos algebraicos dinámicos, que se utilizan para los gráficos dinámicos que utilizan técnicas algebraicas lineales.

12.1 Introducción

La teoría de gráficos algebraicos es el estudio de métodos algebraicos para resolver problemas de gráficos. El álgebra lineal y la teoría de grupos son las dos áreas de álgebra referidas en su mayoría al tratar con gráficos. Los algoritmos de gráficos algebraicos que usan álgebra lineal comúnmente hacen uso de las matrices asociadas con un gráfico para resolver varios problemas en los gráficos. Al utilizar este enfoque para formar algoritmos de gráficos para muchos de los problemas, hemos visto que tiene varios beneficios. En primer lugar, podemos usar varias operaciones de matriz existentes para esta tarea, lo que resulta en algoritmos más simples que se pueden convertir a códigos ejecutables con facilidad en general. Como otra ventaja, las operaciones de matriz paralela y dichos entornos de software están disponibles para hacer más sencilla la formación paralela de estas tareas.

Nuestro propósito en la primera parte de este capítulo es presentar este paradigma y dar ejemplos de cómo resolver algunos de los problemas gráficos que hemos investigado en la Parte II de este libro. Comenzamos con una breve revisión de las matrices que se utilizan para representar gráficos. Luego revisamos los algoritmos de gráficos algebraicos para algunos problemas de gráficos, que incluyen recorridos de gráficos, rutas más cortas desde una sola fuente, rutas más cortas de todos los pares, conectividad y comparación. También discutimos implementaciones paralelas de varios algoritmos

presentados. Un tratamiento exhaustivo de este tema, a saber, algoritmos de gráficos algebraicos se puede encontrar en [13].

En la segunda parte de este capítulo, nos fijamos en los algoritmos de gráficos dinámicos. Muchas redes de la vida real están representadas por gráficos dinámicos en los que se pueden insertar nuevos vértices / bordes y algunos vértices / bordes se pueden eliminar a medida que pasa el tiempo. Por ejemplo, son posibles nuevas interacciones en redes de interacción de proteínas y dos amigos de una persona en una red social pueden conocerse para formar una ventaja entre ellos en dicha red. En lugar de ejecutar los algoritmos, hemos visto en la Parte II desde cero en estos gráficos dinámicos que es beneficioso usar nuevos algoritmos con mejores rendimientos, como lo investigaremos. Por último, revisamos los algoritmos de gráficos algebraicos dinámicos que se utilizan para los gráficos dinámicos utilizando técnicas algebraicas lineales.

12.2 Matrices de grafos

Un $m \times n$ matriz contiene mn números reales organizados como m filas y n columnas. Mostramos el elemento en la fila i y la columna j th de una matriz A como a_{ij} o i veces A [i , j] en algoritmos. Cuando el número de filas y el número de columnas de una matriz A son iguales, A se llama matriz cuadrada . Las operaciones básicas como la suma, la multiplicación por un escalar y la multiplicación de matrices se describen en el Apéndice B.3. Un $n \times 1$ matriz se llama un vector de columna y $1 \times n$ matriz se llama un vector de fila . Las tres matrices principales asociadas con una gráfica son su matriz de incidencia, su matriz de adyacencia y su matriz laplaciana, como vimos en el cap. 2 , que ahora vamos a revisar brevemente.

Matriz de incidencia

Vamos a suponer un grafo no dirigido o dirigido G con n vértices

v_1, \dots, v_n y $E = \{e_1, \dots, e_m\}$ bordes La matriz de incidencia $Q(G)$ de un gráfico G se define como $q_{ij} = 0$ Si e_j no es incidente al vértice v_i ; $q_{ij} = 1$ Si e_j comienza desde v_i y $q_{ij} = -1$ Si e_j termina en el vértice v_i . Las siguientes propiedades de $Q(G)$ se pueden indicar [1].

- Las sumas de columna de $Q(G)$ son cero, por lo tanto, las filas de $Q(G)$ son linealmente independientes.
- El rango de $Q(G)$ es $n-1$ para un gráfico conectado G .
- Para una gráfica con k componentes. $Q(G) = n - k$

Matriz de adyacencia

Nosotros hemos utilizado la matriz de adyacencia $A(G)$ de un gráfico de G en diferentes algoritmos. Revisemos brevemente las propiedades algebraicas básicas de $A(G)$ con respecto a los gráficos. La entrada a_{ij} de $A(G)$ es igual a 0, si los vértices v_i y v_j No son adyacentes y es 1 si son vecinos. Una entrada a_{ii} es 0 en $A(G)$ y, por lo tanto, $A(G)$ tiene ceros en su diagonal. Observamos las siguientes propiedades de $A(G)$.

- Es el número de trayectos de longitud k desde el vértice i al vértice j . $A[i, j]^k$
- Formemos A^{n-1} utilizando la multiplicación booleana y la suma. La entrada (i , j) en esta matriz es 1 si el vértice i está conectado al vértice j . Hemos usado esta propiedad para formar la matriz de conectividad del gráfico G .
- Si la distancia $d(i , j)$ entre los dos vértices i y j es k , entonces las matrices I, A, \dots, A^k son linealmente independientes [1].

Valores propios

Vamos a considerar la ecuación $Ax = \lambda x$ donde A es una matriz cuadrada no singular, x es un vector, y λ es una constante. Cuando un vector se multiplica por una matriz, su dirección cambia. Algunos vectores como x son diferentes ya que no cambian de dirección. Estos vectores se llaman vectores propios de la matriz A y el número λ se llama un valor propio de A. Cuando A es la matriz de identidad, todos los vectores son los vectores propios de A y todos los valores propios son 1. Reescribamos la ecuación $Ax = \lambda x$.

$$Ax - \lambda x = 0$$

(12.1)

$$(A - \lambda I)x = 0,$$

por lo tanto,

$$\det(A - \lambda I)x = 0.$$

(12.2)

La ecuación $\det(A - \lambda I) = 0$ se llama el polinomio característico de A. Esto nos proporciona el método para encontrar los valores propios y los vectores propios de una gráfica mediante la implementación de los siguientes pasos.

Calcular el determinante de $A - \lambda I$. Esto da un polinomio de grado n en λ .

- 1.
2. Resolver $\det(A - \lambda I) = 0$ para encontrar n raíces $\lambda_1, \dots, \lambda_n$ que son los n valores propios de A.
3. Resolver $(A - \lambda_i I)x = 0$ para $i = 1, \dots, n$ para encontrar los vectores propios correspondientes a cada valor propio.

Matriz laplaciana

La matriz Laplaciana L (G) de un gráfico G no ponderado no dirigido, sin bordes múltiples, se define como

$$L(G) = D(G) - A(G),$$

(12.3)

donde D es la matriz de grado con una entrada d_{ii} como el grado de vértice i con todos los demás elementos iguales a 0 y A (G) es la matriz de adyacencia de G. Las otras entradas en L son -1 si el vértice i es adyacente al vértice j y 0 de lo contrario. El L laplaciano (G) de un gráfico G está relacionado con su matriz de incidencia Q (G) de la siguiente manera:

$$L(G) = Q(G)Q^T(G),$$

(12.4)

dónde $Q^T(G)$ Es la transposición de la matriz de incidencia. El laplaciano normalizado \mathcal{L} de G se define como

$$\mathcal{L} = D^{-1/2} L D^{-1/2} = D^{-1/2}(D - A)D^{-1/2} = I - D^{-1/2}AD^{-1/2},$$

(12.5)

El laplaciano normalizado \mathcal{L} Luego tiene las siguientes entradas:

$$l_{ij} = \begin{cases} d_i & \text{if } i = j \\ -1 & \text{if } i \text{ and } j \text{ are neighbors} \\ 0 & \text{otherwise.} \end{cases}$$

(12.6)

El conjunto de todos los valores propios de la matriz laplaciana de un gráfico de G se denomina espectro laplaciano o simplemente el espectro de G . Veremos en breve los valores propios de la matriz laplaciana, que proporcionarán información vital sobre la conectividad de un gráfico.

12.3 Algoritmos de grafos algebraicos

Gráfico algebraicas algoritmos emplean diversas operaciones utilizando las tres matrices principales asociados con un gráfico, su matriz de adyacencia A que es escasa en general, su matriz de incidencia I , y el gráfico Laplaciano L . En esta sección proporcionamos algoritmos algebraicos para problemas de gráficos de muestra.

12.3.1 Conectividad

El segundo valor propio más pequeño de la matriz laplaciana de un gráfico G , llamado el valor de Fiedler , proporciona información sobre qué tan bien está conectado G. Este valor es mayor que 0 si y solo si G está conectado. Por otra parte, cuanto mayor es este valor, el más conectado G es y el número de 0s en los valores propios de Laplace de una gráfica G es el número de componentes conectados de G . El valor de Fiedler de un gráfico G , mostrado por $\alpha(G)$, se denomina conectividad algebraica de G y se ha utilizado en numerosas aplicaciones relacionadas con la teoría de grafos espectrales y problemas de optimización combinatoria. Dejar $\kappa(G)$ denota la conectividad de vértice de g . Fiedler demostró que [9]

$$\kappa(G) \geq \alpha(G).$$

(12.7)

La matriz Laplaciana $L(G)$ de un gráfico G también se puede usar para enumerar los árboles de expansión del gráfico G de acuerdo con el teorema de abajo.

Teorema 18 (teorema del árbol de matrices) Sean u y v los vértices de un gráfico $G = (V, E)$ y sea $L(G)(u, v)$ la submatriz resultante de la eliminación de la fila u y la columna v de su matriz laplaciana $L(G)$. Entonces $\det L(G)(u, v)$ es igual al número de árboles de expansión de G .

Componentes fuertemente conectados

Un digraph fuertemente conectado tiene caminos entre cualquiera de sus vértices en ambas direcciones. Un componente fuertemente conectado (SCC) de un gráfico dirigido G es un subgrafo G' de G , tal que cada vértice en G' está conectado a todos los otros vértices en G' . Revisamos dos algoritmos principales para detectar SCC de un gráfico dirigido debido a Tarjan y Kosaraju en el Cap. 8. El primero usó bordes posteriores en un árbol DFS de un gráfico G para detectar SCC, y el último realizó un DFS en el gráfico colocando los vértices visitados en una pila de acuerdo con sus últimos tiempos de visita, luego obtuvo la transposición de G y los subárboles DFS formados. de G por reventar vértices de G . Cada subárbol con raíces en vértices hechos estallar es entonces un SCC de G .

Podemos estructurar un algoritmo simple utilizando la matriz de conectividad C de un gráfico dirigido para encontrar sus SCC. La matriz de conectividad C tiene entradas $c_{ij} = 1$ Si hay un camino desde un vértice i hasta j y $c_{ij} = 0$ de otra manera. La observación clave aquí es que si formamos la transposición de C , entonces C^T debe tener una ruta desde i hasta j que muestre la ruta de j a i en C para que i y j estén en el mismo SCC. Entonces podemos formar una nueva matriz, C' el cual se forma tomando lógicamente y de cada elemento de C con C^T . Esta matriz C' tiene entradas $c'_{ij} = 1$ si y solo si hay una ruta desde i hasta j y desde j hasta i , es decir, los vértices i y j están en el mismo SCC. Ilustremos este método con la gráfica de la figura 12.1.

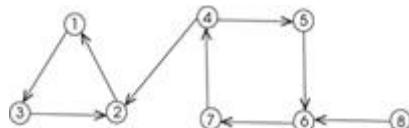


Fig. 12.1 Gráfico de muestra para el ejercicio 6

La matriz de conectividad C y su transposición. C^T Se puede estructurar de la siguiente manera:

$$C = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 5 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 6 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 7 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 8 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, C^T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 4 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 5 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 6 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 7 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Realizar lógicas y de estas dos matrices nos da la nueva matriz. C' . Cualquier elemento i en esta matriz está en el mismo SCC con una entrada j si y solo si $C'[i, j] = 1$. Para este ejemplo de gráfico dirigido, podemos ver que los SCC son $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ y $\{8\}$.

$$C' = C \wedge C^T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 5 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 6 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 7 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

La matriz de conectividad C se puede formar multiplicando sucesivamente la matriz de adyacencia A del gráfico dirigido para obtener A^{n-1} , ya que la ruta más larga en un gráfico no puede ser más larga que $n-1$. Por ejemplo, para una gráfica con 12 vértices, necesitamos formar $C = A^{11}$ que se puede obtener por $A^2 = A \times A$; $A^4 = A^2 \times A^2$; $A^8 = A^4 \times A^4$ y $A^{11} = A^8 \times A^3$ para un total de 4 multiplicaciones de matrices usando lógica y y lógico o las operaciones en lugar de la multiplicación escalar y adición. Necesitaríamos $\lceil \log n \rceil$ multiplicaciones de matriz y desde una $n \times n$ la multiplicación de matrices requiere $O(n^\omega)$ operaciones, la complejidad de este paso es $O(n^\omega \log n)$. Tomando la transposición de C y formando. $C \wedge C^T$ ambos toman $O(n^2)$ tiempo, por lo tanto la complejidad del tiempo total es $O(n^\omega \log n)$ con $\omega < 2.376$. Los algoritmos de detección SCC de Tarjan o Kosaraju usan DFS y, por lo tanto, tienen mejores rendimientos de $O(n+m)$. Sin embargo, paralizar el DFS es difícil, como se explica en el capítulo. 6, pero la multiplicación de matrices se puede paralelizar simplemente distribuyendo las filas o columnas de matrices a un conjunto de procesos como vimos en el Cap. 4.

12.3.2 Búsqueda de amplitud

En lo ancho primera búsqueda (BFS), exploramos vértices que son k saltos de distancia desde un vértice fuente dada antes de explorar los que están $k+1$ salta lejos como se discute en el cap. 6. Esto dio lugar a recorridos de distancia más cortos en un gráfico no ponderado no dirigido. Consideremos la dispersa matriz de adyacencia A de un gráfico no ponderado no dirigido $G = (V, E)$. Tenemos un vector disperso X para mostrar la posición del vértice de origen, por ejemplo, $X[3] = 1$ con todos los demás elementos, 0 significa que el vértice 3 es la fuente. Multiplicando A^T por X nos da el vector que tiene 1 en todos los vecinos del vértice 3. Multiplicando el producto nuevamente por A^T proporciona vecinos que están a dos saltos de distancia y así sucesivamente. Ahora podemos dibujar un algoritmo algebraico usando esta propiedad como se muestra en el Algoritmo 12.4. Tenemos la matriz A y el vector X como entrada y queremos formar la $n \times n$ matriz N , que muestra los vértices que son i distancia en su i^{a} fila. Necesitamos proporcionar una modificación simple ya que el resultado de la multiplicación muestra todos los vértices que están a lo sumo y que se alejan.

Algoritmo 12.1 BFS_Algeb

```

1: Input :  $G(V, E)$ ,  $A$ ,  $X$             $\triangleright$  connected, directed or undirected graph  $G$ , its adjacency matrix  $A$ 
   and source vertex vector  $X$ 
2: Output :  $N[n, n]$                     $\triangleright$  vertices that are 1 to  $n$  hops away
3:
4: form  $A^T$ 
5: temporary matrix  $T$ 
6:  $N \leftarrow 0$ 
7: for  $i = 1$  to  $\text{diam}(G)$  do
8:    $Y^{(i)} \leftarrow A^T \times X$             $\triangleright$  find neighbors  $i$  hop away
9:    $T \leftarrow Y^{(i)} - Y^{(i-1)}$ 
10:   $Y^{(i-1)} \leftarrow Y^{(i)}$ 
11:   $N[i, :] \leftarrow T$                     $\triangleright$  store neighbor identifiers
12:   $X \leftarrow Y$                           $\triangleright$  update
13: end for

```

Investigaremos cómo funciona este algoritmo para el gráfico de muestra de la figura 12.2. La transposición de la matriz de adyacencia A es en sí misma ya que el gráfico G no está dirigido.

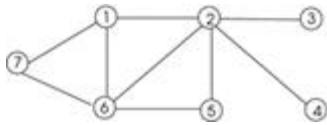


Fig. 12.2 Gráfico de muestra para probar el algoritmo BFS algebraico

Las matrices A y X formadas para el vértice fuente 7 en este gráfico y la matriz adyacente resultante $N^{[1, *]}$ son como sigue. Mostramos la matriz completa para la comparación, sino sólo su i^a fila se muestra en negrita se modifica en i^a iteración.

$$A^T \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 6 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 7 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \times X^{(1)} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \rightarrow N^{(1)} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La segunda iteración del bucle for da como resultado lo siguiente. Tenga en cuenta que el resultado de la multiplicación es el vector (1 1 0 0 1 1 1) y restamos el valor anterior (1 0 0 0 1 0) de este producto para obtener (0 1 0 0 1 0 0), que se convierte en la segunda fila de n .

$$A^T \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 6 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 7 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \times X^{(2)} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow N^{(2)} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ 2 & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

El valor final de N en la tercera iteración se muestra a continuación. La primera fila tiene 1s en los vecinos inmediatos del vértice 7, la segunda fila tiene 1s en los vecinos de dos saltos y la tercera fila muestra los vecinos de tres saltos. Como el diámetro del gráfico es 3, podemos detenernos en la tercera iteración.

$$N^{(3)} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ 2 & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \\ 3 & \mathbf{0} & \mathbf{0} & 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Necesitamos las iteraciones diam (G) del bucle for y también necesitamos realizar $\Theta(n^2)$ multiplicaciones en cada iteración resultando en $\Theta(n^2 \text{diam}(G))$ complejidad de tiempo para este algoritmo. Podemos ver inmediatamente que este enfoque algebraico se puede paralelizar convenientemente mediante la partición 1-D de A^T y X y distribuyéndolos a una serie de procesos.

12.3.3 Rutas más cortas

Analizamos las versiones algebraicas de dos algoritmos principales para las rutas más cortas: el algoritmo SSSP de Bellman-Ford y el algoritmo APSP de Floyd-Warshall en esta sección.

12.3.3.1 Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford se basa en la programación dinámica y construye las rutas más cortas desde un vértice de origen en un gráfico ponderado dirigido o no dirigido progresivamente, tal como lo analizamos en el Capítulo 7. Puede funcionar en presencia de bordes de peso negativo de la gráfica, sin embargo, solo informará los ciclos negativos si existen. Dado un gráfico ponderado $G = (V, E, w)$ con $w: E \rightarrow \mathbb{R}$ y un vértice de origen s , comienza con una estimación de la distancia $d(v)$, $\forall v \in V$ utilizando la matriz de adyacencia A y realiza la relajación en cada paso hasta encontrar las distancias más cortas. La relajación de un borde (u, v) se declara como que proporciona $d(v) = \min[d(v), d(u) + w(u, v)]$. Una formulación algebraica de este algoritmo utilizará la matriz de adyacencia dispersa $A[n, n]$ y un vector $D[n]$ que muestra la distancia más corta $d(i)$ desde s , $\forall i \in V$ al final como se muestra en el algoritmo 12.2.

Algorithm 12.2 BF_Algeb

```

1: Input:  $G(V, E)$ ,  $A[n, n]$ ,  $D[n]$   $\triangleright$  connected, directed or undirected weighted graph  $G$ , its
   adjacency matrix  $A$  and distance vector  $D$ 
2: Output:  $D[n]$   $\triangleright$  shortest distances
3:
4:  $D \leftarrow \infty$ 
5: for  $k = 1$  to  $n$  do
6:    $D \leftarrow D \text{ min.} + A$   $\triangleright$  find neighbors  $i$  hop away
7:   if  $D \neq d \text{ min.} + A$  then
8:     return "negative cycle found"
9:   end if
10: end for

```

12.3.3.2 Floyd - Algoritmo Warshall

Este algoritmo dinámico proporcionó APSP entre todos los vértices de un gráfico en $O(n^3)$ tiempo como hemos revisado en el cap. 7. Utilizó el método de relajación para actualizar las distancias trabajando en la matriz de adyacencia del gráfico. Por lo tanto, podemos decir que es un algoritmo de grafo algebraico. El pseudocódigo de este algoritmo usando notación matricial se muestra en el algoritmo 12.5.

Algorithm 12.3 FW_Algeb

```

1: Input:  $G(V, E)$ ,  $A[n, n]$ ,  $D[n, n]$   $\triangleright$  connected, directed or undirected weighted graph  $G$ , its
   adjacency matrix  $A$  and distance vector  $D$ 
2: Output:  $D[n]$   $\triangleright$  shortest distances
3:
4:  $D \leftarrow A$ 
5: for  $k = 1$  to  $n$  do
6:    $D \leftarrow D \text{ min}[D(:, k) \text{ min.} + D(k, :)]$   $\triangleright$  find neighbors  $i$  hop away
7: end for

```

12.3.4 Árboles de expansión mínimos

Un árbol de expansión mínimo (MST) de un gráfico conectado ponderado no dirigido $G = (V, E, w)$, $w: E \rightarrow \mathbb{R}$ es un árbol de expansión del MST con la suma de los pesos mínimos de los bordes entre todos los árboles de expansión de G . Un MST de un gráfico es único si todos los pesos de los bordes son distintos. Un bosque de expansión mínimo de un gráfico no dirigido ponderado sin conectar G consta de MSTs en cada componente de G . Ahora reconsideraremos el algoritmo de Prim que revisamos en el cap. 7, pero esta vez utilizando la versión algebraica.

Algoritmo de Prim

Este algoritmo comienza con un conjunto S que consiste en cualquier vértice del gráfico $G = (V, E, w)$ inicialmente y iterativamente busca el peso mínimo borde saliente (MWOE) e de S . El borde e está incluido en el árbol MST T y los puntos finales del borde e se añaden a S . El algoritmo termina cuando $S = V$ y tiene una complejidad de tiempo de $O(m \log n)$ o $O(m + n \log n)$ cuando se utiliza la estructura de datos de la cola de prioridad.

En la versión algebraica de este algoritmo, asumiremos que los vértices están numerados de 1 a n para mayor comodidad y describiremos un algoritmo como en [13]. Tenemos la matriz de distancia escasa D con entradas $d_{ij} = 0$ cuando $i = j$ es igual a w_{ij} cuando $(i, j) \in E$ y es asignado ∞ cuando $(i, j) \notin E$. UNA $1 \times n$ vector M con entradas m_i muestra si vértice $i \in S$ o no con $m_i = 0$ Si $i \notin S$ y $m_i = \infty$ Si $i \in S$. UNA $1 \times n$ vector Q con entradas q_i muestra el peso de borde más ligero que conecta i al conjunto S y $q_i = 0$ Si $i \in S$.

Comenzamos el algoritmo incluyendo el primer vértice i en el conjunto M haciendo su valor ∞ en el sentido de que es un elemento del conjunto S y asignar Q a la primera fila de la matriz de distancia D. La variable wt almacena la suma de los pesos de los bordes del MST en cualquier momento. A continuación, entrar en el tiempo de bucle y continuar hasta que todos los elementos de M son procesados y tienen ∞ valores. El argmin operación en la línea 8 se utiliza para encontrar el vértice u , que tiene el borde más ligero a un vértice en S . Esto es realmente buscar el MWOE en el algoritmo clásico. Este vértice se incluye en S al establecer su valor en ∞ en m . También necesitamos otro vector. π que contiene el padre de cada vértice para almacenar información MST. Una tupla $<weight(u, v), u>$ se puede definir para encontrar el parente del vértice u recién agregado con $v \in S$. El parente de la ventaja cuando se está incluido en el MST se mantiene en la variable x en la línea 10 y el parente del vértice en el extremo de la nueva ventaja cuando su se introduce en el MST se guarda para asignar al valor de los padres en π .

Algorithm 12.4 Prim_Algeb

```

1: Input :  $G(V, E)$ ,  $A[n, n]$ ,  $D[n]$   $\triangleright$  connected, directed or undirected weighted graph G, its
   adjacency matrix A and distance vector D
2: Output:  $D[n]$   $\triangleright$  shortest distances
3:
4:  $M \leftarrow 0$ ,  $wt \leftarrow 0$ 
5:  $\Pi \leftarrow \emptyset$ ,  $M[1] \leftarrow \infty$ 
6:  $Q \leftarrow D[1, :]$ 
7: while  $M \neq \infty$  do
8:    $u \leftarrow \text{argmin}(M + Q)$ 
9:    $M[u] \leftarrow \infty$ 
10:   $<w, x> \leftarrow Q[u]$ 
11:   $wt \leftarrow wt + Q[u]$ 
12:   $\pi[u] \leftarrow x$ 
13:   $Q \leftarrow Q, \min D[u, :]$ 
14: end while

```

Necesitamos $\Theta(n)$ para la operación argmin y el tiempo total, por lo tanto, es $O(n^2)$ que es peor que el algoritmo Prim original para gráficos dispersos pero comparable para gráficos muy densos. Sin embargo, el enfoque algebraico es adecuado para el procesamiento paralelo como en otros algoritmos de gráficos algebraicos.

12.3.5 Emparejamiento algebraico

Una M coincidente de un gráfico no dirigido $G = (V, E)$ es un subconjunto de sus bordes, de manera que ningún borde en M comparte puntos finales. En una coincidencia perfecta M , cada $v \in V$ es incidente a un borde $e \in M$. Una coincidencia máxima M' de G tiene el tamaño más grande entre todos los emparejamientos de G y un acoplamiento máximo M no está contenido en cualquier otro juego de G .

Definición 12.1 (matriz de Tutte) La matriz de Tutte T (G) de un gráfico simple no dirigido $G = (V, E)$ es un $n \times n$ matriz con elementos;

$$T_{ij} = \begin{cases} x_{ij} & \text{if } (i, j) \in E \text{ and } i < j \\ -x_{ij} & \text{if } (i, j) \in E \text{ and } i > j \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

dónde x_{ij} Son variables formales. Tutte demostró una relación importante entre la matriz de Tutte y la combinación perfecta de una gráfica [24].

Teorema 12.1 (Tutte) Let $G = (V, E)$ ser un simple gráfico no dirigido con un Tutte matriz T . Entonces, G tiene una coincidencia perfecta si y solo si $\det(T) \neq 0$.

La matriz de Tutte consiste en variables y su determinante es un polinomio de sus variables. El determinante debe ser un polinomio con todos los parámetros cero para tener una coincidencia perfecta. Por lo tanto, podemos calcular la matriz T de Tutte , calcular su determinante y verificar si esto es cero. Sin embargo, computar T puede tomar tiempo exponencial. Lovasz proporcionado un algoritmo aleatorio para probar la condición de coincidencia perfecta de un gráfico mediante la sustitución para cada variable de Tutte matriz T a partir de una polinomialmente gran conjunto de números enteros y luego comprobar si T es no singular [14]. Lovasz También mostró que el rango de la matriz de Tutte de un gráfico G proporciona el tamaño de la coincidencia máxima de G [15] que se muestra en el siguiente teorema.

Teorema 12.2 (Lovasz) Let $G = (V, E)$ ser un simple gráfico no dirigido con un Tutte matriz T y k sea el tamaño de la coincidencia de máximo de G . Entonces rango $(T) = 2k$

Rabin– Algoritmo Vazirani

Una vez que sabemos que el gráfico G tiene una coincidencia perfecta, necesitamos un algoritmo para encontrar esta coincidencia. Supongamos un borde (u , v) que pertenece a una coincidencia perfecta en G . El subgrafo obtenido al eliminar (u , v) y todos sus bordes adyacentes de G también tiene una coincidencia perfecta. Si sabemos cómo encontrar una ventaja correo de un juego perfecto, podemos construir de forma recursiva una coincidencia perfecta de G . Rabin y Vazirani encontraron que lo inverso T^{-1} La matriz de Tutte proporciona esta información como se muestra en el siguiente teorema.

Teorema 12.3 (Rabin– Vazirani) Veamos $G = (V, E)$ ser un simple gráfico no dirigido con un Tutte matriz T . Entonces, $(T(G')^{-1})_{i,j \neq 0}$ si y solo si $G - \{i, j\}$ tiene una coincidencia perfecta.

Rabin y Vazirani desarrollaron un algoritmo aleatorio basado en este teorema, que calcula la matriz de Tutte y su inverso y encuentra un borde (i , j) que satisface la propiedad en el teorema 12.3 . Esta ventaja pertenece a la coincidencia perfecta y, por lo tanto, se agrega a la coincidencia actual. Luego se elimina de la gráfica con todos sus bordes adyacentes. El algoritmo continúa con el gráfico restante hasta que se vacía.

Algorithm 12.5 RV_Alg

```

1: Input :  $G = (V, E)$                                  $\triangleright$  undirected simple graph  $G$ 
2: Output:  $M$                                       $\triangleright$  perfect matching of  $G$ 
3:
4:  $M \leftarrow \emptyset$ 
5:  $G' = (V', E') \leftarrow G = (V, E)$ 
6: while  $G' \neq \emptyset$  do
7:   compute  $T(G')$  and instantiate each variable with a random value from  $\{1, \dots, n^2\}$ 
8:   compute  $T(G')^{-1}$ 
9:   find  $i$  and  $j$  such that  $(v_i, v_j) \in E'$  and  $(T(G')^{-1})_{i,j} \neq 0$ 
10:   $M \leftarrow M \cup \{(v_i, v_j)\}$ 
11:   $G' \leftarrow G' - \{v_i, v_j\}$ 
12: end while

```

Rabin y Vazirani demostraron que las inversiones de matriz $n / 2$ son suficientes ya que cada inversión proporciona un borde de coincidencia. Matriz de inversión que lleva $\mathcal{O}(n^{\omega})$ el tiempo domina el tiempo que toma el algoritmo y cada prueba para encontrar las coincidencias perfectas $\mathcal{O}(n^{(\omega+1)})$ tiempo [19].

12.4 Algoritmos de gráficos dinámicos

Hemos visto algoritmos de gráficos estáticos que proporcionan una salida de alguna función en la estructura de datos de gráficos hasta ahora. Sin embargo, los gráficos que representan muchas redes de la vida real no son estáticos a través de modificaciones en el tiempo. Un gráfico dinámico puede evolucionar con el tiempo debido a cambios en G , como la inserción o eliminación de bordes. Los gráficos dinámicos representan muchas redes de la vida real, por ejemplo, Internet, redes de interacción de proteínas y redes sociales en las que estos cambios ocurren con frecuencia. Un algoritmo de gráfico dinámico permite las siguientes operaciones en gráficos dinámicos:

- Consulta : Evaluamos una cierta propiedad de la gráfica G . Por ejemplo; “¿Está conectada la gráfica?”
- Insertar : se inserta un borde o un vértice aislado en el gráfico.
- Eliminar : se elimina de la gráfica un borde o un vértice aislado.

Las dos últimas operaciones son comúnmente llamadas procedimientos de actualización . Podemos realizar estas operaciones utilizando los algoritmos de gráficos estáticos que hemos visto hasta ahora desde cero para el gráfico modificado. Sin embargo, el objetivo principal de un algoritmo de gráfico dinámico es proporcionar soluciones más eficientes para estas operaciones que los algoritmos estáticos. Estos algoritmos se clasifican de la siguiente manera:

- Totalmente dinámico : se permiten las inserciones y la eliminación de bordes y vértices.
- Incremental : solo se permiten inserciones de aristas y vértices.
- Decremental : solo se permite la eliminación de bordes y vértices.

Los dos últimos tipos de algoritmos se denominan algoritmos de gráficos dinámicos parciales . Se permiten consultas en todos los algoritmos descritos. Intuitivamente, responder a una consulta en un gráfico dinámico en general es más sencillo que realizar una operación de actualización. Otra distinción es entre gráficos no dirigidos y dirigidos. Una operación de gráfico dinámico si una consulta o una actualización es generalmente más difícil en un gráfico dirigido que en un gráfico no dirigido. La tarea de un algoritmo de gráfico dinámico sigue siendo proporcionar un mejor rendimiento que su contraparte estática. Veremos que el diseño de estructuras de datos inteligentes es crucial cuando se forman algoritmos dinámicos.

El algoritmo de conectividad completamente dinámico en un gráfico no dirigido permite la inserción y eliminación de bordes, y permite consultas como "¿está conectado el gráfico" o "están los vértices u y v en el mismo componente?". En el problema del árbol de expansión mínima totalmente dinámico , mantenemos un bosque de árboles de expansión mínima cuando se insertan, eliminan y cambian los pesos de los bordes. Los principales problemas en los gráficos dirigidos son el cierre transitivo dinámico y las rutas más cortas dinámicas . En el primer problema, mantenemos información para evaluar si un vértice v es accesible desde un vértice u cuando los bordes se eliminan y se insertan. El problema de la ruta más corta implica proporcionar y mantener información sobre las rutas más cortas cuando los bordes se insertan y eliminan en un entorno dinámico.

Comenzamos esta sección definiendo primero algunos métodos que se utilizarán en el diseño de algoritmos de gráficos dinámicos eficientes para gráficos no dirigidos y dirigidos y luego proporcionamos una breve encuesta de algoritmos para dos problemas representativos de gráficos dinámicos; Conectividad y emparejamiento.

12.4.1 Métodos

Los métodos para gráficos no dirigidos y gráficos dirigidos difieren significativamente. Clasificaremos estos métodos como se describe en [7] para gráficos no dirigidos y dirigidos.

12.4.1.1 Gráficos no dirigidos

Los principales métodos de diseño de algoritmos para el gráfico dinámico no dirigido son el agrupamiento, la dispersión y la aleatorización.

Agrupamiento

El método de agrupación en clústeres para los gráficos dinámicos se propuso en [10], donde la gráfica se divide en una serie de agrupaciones y la operación de actualización se realiza solo en la agrupación relacionada, por ejemplo, un árbol de expansión puede dividirse en varios subárboles. Se mostró en [10] que el bosque de expansión mínimo de un gráfico se puede mantener en $\mathcal{O}(\sqrt{m})$ por actualización utilizando este método.

Sparsification

Sparsification una técnica que se utiliza como una caja negra en el diseño de algoritmos de gráficos dinámicos como se describe en [8]. Este es un método de dividir y conquistar en el que el gráfico G se divide en un conjunto de subgrafos dispersos $O(m/n)$. Cada información de subgrafo se mantiene en un certificado que se fusiona en pares con otros subgraphs dando como resultado un árbol equilibrado para el gráfico con cada vértice que tiene un certificado disperso. Luego se realiza una operación de actualización en $\mathcal{O}(\log m/n)$ Gráficos con cada uno con $O(n)$ bordes [7 , 8].

Aleatorización

La aleatorización es un método poderoso para el diseño de algoritmos. También demostró ser eficiente en el diseño de algoritmos de gráficos dinámicos para gráficos no dirigidos en [11]. Utilizaron el muestreo aleatorio para seleccionar un borde que no es de árbol cuando se elimina un borde de árbol de la gráfica cuando se mantiene un árbol de expansión. También se puede combinar con la descomposición del gráfico.

12.4.1.2 Gráficos dirigidos

Las herramientas para gráficos dirigidos se describen en [7] en términos de estructuras de datos utilizadas y métodos empleados. Se puede usar una estructura de árbol especial

llamada árbol de accesibilidad para resolver problemas de gráficos dinámicos. El objetivo de utilizar esta estructura de datos es mantener un árbol BFS durante las eliminaciones de bordes. Los dos tipos de operaciones admitidas son Eliminar (u, v), que elimina el borde (u, v) del gráfico y Nivel (u), que devuelve el nivel de un vértice u en el árbol BFS. Las estructuras de datos matriciales también se pueden usar para mantener información de gráficos dinámicos con operaciones dinámicas en filas y columnas.

Demetrescu y Italiano propusieron rutas localmente más cortas definidas como rutas que tienen todas las subrutas adecuadas como rutas más cortas [4]. Dado un grafo $G = (V, E)$, la propiedad de rutas largas significa seleccionar $S \subset V$ en resultados aleatorios en caminos suficientemente largos de G que tienen vértices comunes en S con alta probabilidad. Usando esta propiedad, podemos encontrar un camino largo usando búsquedas cortas para diseñar algoritmos para el cierre transitivo y los caminos más cortos [5, 7].

12.4.2 Conectividad

En el problema de conectividad dinámica, necesitamos probar la conectividad del gráfico cuando hay consultas y actualizaciones. Las consultas típicas estarían probando si el gráfico G está conectado (conectado (G)) y si los vértices u y v están conectados (conectado (u, v)). Para el problema de la actualización, deberíamos realizar estas consultas cuando se inserta o elimina un borde (u, v) del gráfico. Consideraremos dos casos por separado ya que sus implementaciones son muy diferentes; La conectividad dinámica incremental y decremental .

12.4.2.1 Algoritmos incrementales

Queremos mantener la información sobre los componentes conectados de un gráfico G no dirigido de forma dinámica con el uso de las siguientes operaciones.

- Conectado (u, v) : informe si u y v están en el mismo componente conectado de G
- Insertar (u, v) : Añadir borde (u, v) a G .

Recordemos la estructura de datos de búsqueda sindical que hemos revisado en el cap. 7 . Esta estructura mantiene grupos separados de elementos de datos y cada grupo tiene un representante. Soporta dos operaciones; find (x) devuelve el representante del conjunto al que pertenece x y union (x, y) fusiona los grupos de x e y . Podemos verificar si dos elementos de datos están en el mismo grupo probando a sus representantes con find para ver si tienen el mismo representante. Si lo hacen, están en el mismo grupo. También podemos unir dos grupos por el sindicato.operación. La estructura de datos de union-find puede implementarse en $\mathcal{O}(\alpha(n))$ tiempo donde $\alpha(n)$ es la inversa de la función de Ackermann de crecimiento muy rápido [22].

Podemos ver que esta estructura de datos es adecuada para tener un algoritmo de conectividad incremental dinámico. Podemos realizar un algoritmo DFS en el gráfico y almacenar cada árbol del bosque en un grupo con la raíz del árbol como representante del árbol. Cada consulta se puede realizar mediante operaciones de encontrar (x), que generan la raíz del árbol que contiene x y una operación de inserción (u, v) se realiza mediante la operación de unión que combina dos árboles si u y v están en árboles diferentes .

12.4.2.2 Algoritmos decrementales

En el problema de conectividad por decremento, queremos eliminar un borde mediante la operación de eliminar (u, v) y verificar si el gráfico todavía está conectado o no. La estructura de búsqueda de la unión no funciona en este caso y varios estudios apuntaron a encontrar una solución a este problema. Un tiempo de actualización de $O(\sqrt{m})$ utilizando grupos se presentó en [10] que se mejoró a $O(\sqrt{n})$. Utilizando el método de dispersión en [8]. Un algoritmo aleatorio con amortizado $O(\log^2 n)$ el tiempo esperado por operación fue propuesto en [11]. Un algoritmo determinista con amortizado $O(\log^2 n)$ por operación se presentó en [12] y se seleccionó un algoritmo aleatorio con $O(\log n(\log \log n)^3)$. El tiempo amortizado por operación se describe en [23]. Examinaremos de cerca una estructura de datos novedosa llamada árbol de visitas de Euler que se puede usar para problemas de conectividad dinámica.

Euler Tour Trees

El árbol de giras de Euler (ETT) se presentó en [11] para almacenar información sobre gráficos dinámicos. Un recorrido de Euler de un gráfico G es un camino que atraviesa cada borde de G exactamente una vez. Un árbol no tiene un recorrido de Euler, para realizar un recorrido de Euler de un árbol T , cada borde se considera bidireccional y, por lo tanto, cada borde se cruza dos veces y el recorrido comienza y termina en el vértice de la raíz [11]. Un árbol de recorrido de Euler (ETT) asocia una clave ponderada o no ponderada para cada vértice. Una ETT es básicamente un árbol binario equilibrado de una gira de Euler del árbol T . Podemos pensar en un recorrido de Euler de un árbol T como un recorrido de profundidad primero T . Un recorrido de Euler de un árbol muestra se muestra en la Fig. 12.3, donde el BST almacena los vértices en el orden de sus tiempos de visita y cada vértice en el árbol tiene punteros a los vértices en el BST que muestran su primer y último tiempo visitado.

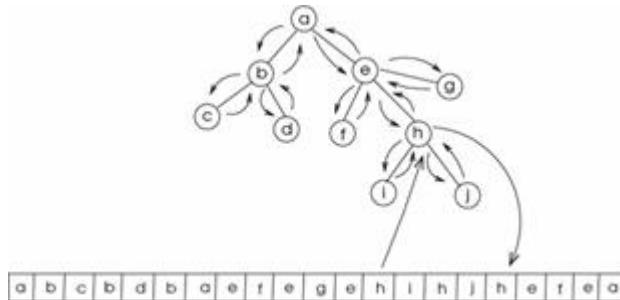


Fig. 12.3 Un recorrido de Euler de un árbol. Se muestran los tiempos de visita para el vértice h .

La idea principal del algoritmo de conectividad basado en ETT es almacenar el recorrido de Euler de un árbol en lugar de almacenar el árbol. Las inserciones o eliminaciones de bordes se pueden realizar modificando los árboles de Euler del bosque. La prueba de si dos vértices están conectados se puede hacer verificando si estos vértices están en el mismo ETT. Las siguientes operaciones principales se proporcionan en un ETT:

- FindRoot (v) : encuentra la raíz del árbol que contiene el vértice v . Como la raíz se visita como el primer elemento y el último elemento del árbol, se devuelve el elemento mínimo o máximo del árbol.
- Cortar (v) : el subárbol arraigado en el vértice v se corta del árbol que contiene. Esto se puede implementar dividiendo el BST en tres segmentos; segmento antes de la primera visita a v , segmento entre la primera y última visita a v , y el segmento después de la última visita a v . El segundo

segmento se muestra como el cuadro en negrita en la Fig. 12.4 . El primer segmento contiene el recorrido de Euler del árbol antes de alcanzar el vértice v , el recorrido de Euler del subárbol arraigado en v y el recorrido de Euler del árbol después de v . Se visita la última vez. Ahora podemos fusionar el primer y tercer segmento para realizar la operación de corte. Este procedimiento se ilustra en la figura 12.4 para el vértice h . Tenga en cuenta que una aparición del vértice e debe eliminarse de la BST.

- Enlace (u, v) : el subárbol arraigado en el vértice u está conectado como un hijo del vértice v . Dividimos el BST en dos segmentos; segmento izquierdo S_1 Es desde el principio hasta antes de la última visita a v , y el segundo segmento. S_2 Es el resto de la BST. El ETT del bosque es entonces. $ETT_F = \{S_1 \cup v \cup ETT_u \cup S_2\}$, donde ETT_u es el ETT del árbol arraigado en el vértice u .

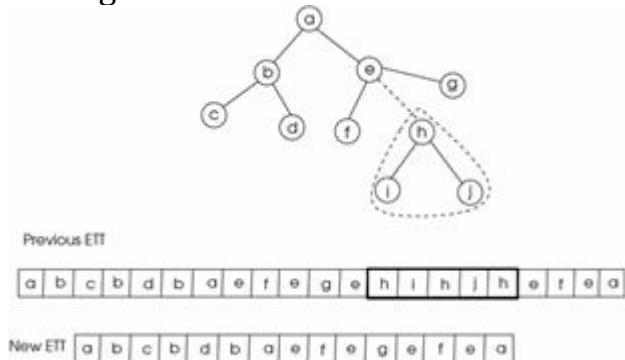


Fig. 12.4 Operación del procedimiento Corte (h) en el gráfico de muestra de la Fig. 12.3

Hay varios otros procedimientos para modificar los ETT como se describe en [11]. Para responder a la consulta conectada (u, v), el procedimiento FindRoot encuentra las raíces de los ETT que contienen estos vértices y verifica si son iguales. La inserción y la eliminación se realizan mediante la reconstrucción de los recorridos de Euler con cambios en solo $O(\log n)$ Vértices de la BST balanceada. Las consultas pueden ser respondidas en $O(\log n / \log \log n)$ tiempo e inserciones toman $O(\log^2 n / \log \log n)$ tiempo utilizando este método [11].

12.4.3 Emparejamiento dinámico

La coincidencia dinámica de un gráfico G debe mantener la coincidencia máxima de G en presencia de inserciones de borde y eliminaciones. Para hacerlo, debemos asegurarnos de que no haya un borde (u , v) de G con ambos puntos finales libres, ya que dicho borde debe incluirse en la coincidencia. Necesitamos verificar lo siguiente al insertar y eliminar bordes en un gráfico con una M máxima coincidente .

- Insertar (u, v) : siempre que se inserte un borde (u , v) en G , podemos verificar sus puntos finales e incluir este borde en la coincidencia máxima M de G si tanto u como v están libres. Cuando se elimina un borde no coincidente (u , v), no tenemos que hacer nada, ya que este borde no es parte de la coincidencia máxima. Para otros casos, necesitamos verificar los vecinos de los vértices u y v y actualizar la coincidencia máxima según corresponda.
- Eliminar (u, v) : debemos considerar dos casos cuando se elimina un borde (u , v) del gráfico; cuando se elimina un borde coincidente (u , v), debemos verificar $\deg(u) + \deg(v)$ vecinos de estos vértices que está bien cuando los grados de vértices son pequeños. Para una gráfica con grandes grados, se puede usar la

aleatorización en la que un vértice se empareja con un vértice vecino seleccionado al azar en el apareamiento aleatorio . El enfoque simple descrito se combina con el apareamiento aleatorio que se presenta en [2] para dar como resultado la amortización esperada $\mathcal{O}(\log n)$ Tiempo por actualización.

Analizaremos más de cerca un algoritmo determinista que funciona en la lógica descrita para actualizar la coincidencia máxima de un gráfico.

Algoritmo de Neiman y Salomón

Neiman y Solomon presentaron un algoritmo determinista para encontrar la coincidencia máxima de un gráfico que se ejecuta en $\mathcal{O}(\sqrt{m})$ Tiempo de actualización con una aproximación de $3/2$ [17]. La idea principal de este algoritmo es considerar tres casos al agregar un borde (u, v) a la gráfica G . Si ambos puntos finales de (u, v) están libres, entonces (u, v) se agregará a la coincidencia existente. Si tanto u como v coinciden, entonces la coincidencia no se cambia. Cuando un punto extremo del borde coincide y otro no, se buscan los vecinos de los vértices u y v . Cuando se elimina un borde coincidente (u, v) del gráfico, los vecinos de u y v se comprueban para ver si se puede agregar un borde (u, w) o (u, y) a la coincidencia. El algoritmo funciona en rondas y los tres invariantes que se mantendrán en la i quinta ronda del algoritmo son los siguientes.

El grado $\deg(v)$ de un vértice libre v que puede ser igualada en todo momento es $\leq \sqrt{2(m_i + n)}$

- 1.
2. Para un vértice libre v , $\deg(v) \leq \sqrt{2m_i}$. Cuando un vértice de alto grado u se libera y todos sus vecinos coinciden, un sustituto v' Se busca en lugar de u . El vértice v' se hace coincidir con un vecino v de u tal que $\deg(v') \leq \sqrt{2m}$. Entonces u y v se pueden emparejar y el vértice de bajo grado v' se vuelve libre.
3. M es maximo

El algoritmo 12.6 muestra el pseudocódigo del procedimiento de inserción en este algoritmo. Tenemos un procedimiento llamado Sustituto que se llama cuando un punto final del borde que se va a insertar coincide y el otro está libre. En este caso, agregar el borde (u, v) puede resultar en un camino de aumento, lo que significa que la coincidencia máxima M se puede ampliar.

Algorithm 12.6 Inserting an edge

```

1: procedure INSERT( $(u, v)$ )
2:    $E \leftarrow E \cup \{(u, v)\}$ 
3:    $m_i \leftarrow m_i + 1$ 
4:   if both  $u$  and  $v$  are free then
5:      $M \leftarrow M \cup \{(u, v)\}$ 
6:     if  $u$  is free and  $v$  is matched (or other way) then
7:       surrogate( $u$ ) (or  $v$ )
8:     end if
9:   end if
10: end procedure

```

El procedimiento Insertar llama al procedimiento sustituto cuando un extremo del borde agregado (u, v) está libre y el otro no. En el primer caso, si $(u, v) \notin M$, simplemente eliminamos el borde (u, v) del gráfico sin cambiar la M coincidente . En el segundo caso, $(u, v) \in M$, y el borde (u, v) se borra de la M coincidente . Esto puede resultar en la formación de nuevos caminos de aumento de longitud menor o igual a 3 que comienzan en u o v . En este caso, verificamos si hay un vértice libre w que sea un vecino del vértice u o v , en cuyo caso el borde (u, w) se agrega a la M correspondiente . Además, se comprueba el grado de vértice u considerado; dos casos son cuando $\deg(u) \leq \sqrt{2m}$ o $\deg(u) > \sqrt{2m}$. En el primer caso, u puede

volverse libre, pero se realiza una búsqueda de una ruta de aumento. Cuando $\deg(u) > \sqrt{2m}$, u no puede ser libre ya que su grado es alto o no tiene vecinos libres. En este caso, se llama al procedimiento Sustituto para encontrar un vértice sustituto que puede quedar libre en lugar del vértice u .

Un trabajo reciente sobre la coincidencia máxima aproximada determinista dinámica con el tiempo de actualización de peor caso de $O(\log^3 n)$ el tiempo se presenta en [3] y los algoritmos de coincidencia aproximados de 2 aproximados se informan en [18 , 21].

12.5 Algoritmos de gráficos algebraicos dinámicos

Un algoritmo de gráfico algebraico dinámico tiene como objetivo resolver un problema en un gráfico dinámico utilizando métodos algebraicos. Nuestro enfoque principal aquí es nuevamente implementar operaciones de álgebra lineal para el problema del gráfico dado. Se puede formar una biblioteca de matriz dinámica para este propósito y primero listaremos las posibles operaciones en dicha biblioteca. Luego examinaremos los algoritmos de gráficos algebraicos dinámicos para detectar dos problemas principales que hemos estado investigando en este capítulo; Conectividad y emparejamiento.

12.5.1 Una biblioteca de matrices dinámicas

Una operación de matriz dinámica se puede definir como el procedimiento que realiza una función de matriz como encontrar determinante o inverso de una matriz cuando se produce un cambio como el contenido de una fila o columna. Este procedimiento debe implementar la función requerida sin tener que ejecutar la contraparte estática desde cero y, por lo tanto, debe proporcionar un mejor rendimiento. Las siguientes operaciones de matriz se pueden realizar dinámicamente como se describe en [20].

- Determinante de una matriz
- Adjunto de una matriz
- Inversa de una matriz
- Rango de matriz
- Polinomio característico de una matriz.
- Sistema lineal de ecuaciones.

Se demostró en [20] que encontrar el determinante dinámico de una matriz, el cálculo de la matriz inversa, la matriz adjunta y la resolución de un sistema lineal de ecuaciones con actualizaciones de fila y columna no singulares, se puede resolver con los siguientes costos:

- Inicialización : $O(n^{\omega})$ operaciones aritméticas.
- Actualización : $O(n^2)$ operaciones aritméticas.
- Consulta : O (1) operaciones aritméticas.

12.5.2 Conectividad

Hemos buscado una solución al problema de conectividad utilizando primero métodos algebraicos, luego revisamos el problema de conectividad dinámica. Ahora queremos investigar los algoritmos de conectividad dinámica utilizando álgebra lineal. Recordemos los dos principales métodos algebraicos para la conectividad:

- Matriz laplaciana : el segundo valor propio más pequeño λ_2 de la matriz laplaciana llamada conectividad algebraica o el valor de Fiedler proporciona información sobre la conectividad del gráfico. El laplaciano de un gráfico no

ponderado es una matriz semidefinita positiva y simétrica que tiene un positivo

λ_2 Si y solo si el gráfico está conectado. Para encontrar este valor, necesitamos resolver la ecuación $\det(A - \lambda I)x = 0$.

- Matriz de adyacencia : la suma de los poderes. $C = \sum_{k=0}^{\infty} A^k$ de la matriz de adyacencia A para $k \leq n-1$. Proporcionar información sobre la conectividad de un gráfico. Los c_{ij} la entrada de C muestra el número de caminos de longitud k o menos entre los vértices i y j. Si todas las entradas de A^k son positivos, entonces la gráfica está conectada.

Encontrar el determinante de una matriz se puede realizar dinámicamente [20] y, por lo tanto, tenemos un método dinámico para encontrar la conectividad utilizando la matriz laplaciana. De manera similar, la multiplicación de matrices de matrices dinámicas se puede realizar para obtener un método de conectividad dinámica utilizando la matriz de adyacencia. El mantenimiento de la conectividad en un sistema distribuido que consta de muchos elementos informáticos autónomos tiene varias aplicaciones. Por ejemplo, es necesario proporcionar conectividad en una red de robot móvil para la coordinación de los robots y mantener una definición positiva de entradas positivas de C es suficiente para proporcionar conectividad en dicha red [25].

12.5.3 Combinación perfecta usando eliminación gaussiana

Nosotros describimos Rabin y Vazirani algoritmo [19] para una perfecta coincidencia en la Sección. 12.3.5 . La matriz de adyacencia aleatoria A (G) del gráfico G , llamada matriz Tutte T , se crea primero en este algoritmo. Su inverso $A^{-1}(G)$ luego se calcula y se encuentra un borde e permitido , este borde y sus puntos finales se eliminan de G y se crea una nueva matriz T de Tutte para el nuevo gráfico. Este bucle continúa hasta que G se vacía. El tiempo para calcular la matriz. $T^{-1}(G)$ es $O(n^{\omega})$ Resultando en $O(n^{2\omega+1})$ tiempo en total.

Mucha y Sankowski descubrieron que el cálculo de la inversa de la matriz de Tutte en cada iteración no es necesario, ya que la matriz de Tutte en la iteración r , T_{r+1} , es T_r con dos filas y columnas correspondientes a i y j eliminadas [16]. El siguiente teorema fue usado para formar una relación entre T_{r+1}^{-1} y T_r^{-1} .

Teorema 12.4 (Eliminación) Sea A un no singular $n \times n$ matriz. Entonces, A y su inverso se pueden escribir como

$$A = \begin{pmatrix} a_{11} & v^T \\ u & B \end{pmatrix}, A^{-1} = \begin{pmatrix} \hat{a}_{11} & \hat{v}^T \\ \hat{u} & \hat{B} \end{pmatrix}$$

dónde a_{11} , \hat{a}_{11} son numeros, u , v , \hat{u} , \hat{v} son $n-1 \times 1$ vectores y $\hat{a}_{11} \neq 0$. Entonces, $B^{-1} = \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{11}$. Usando este teorema, podemos calcular el inverso de una matriz dinámicamente después de eliminar una fila y una columna sin tener que calcular el inverso desde cero. Consideramos las columnas como variables y las filas como las ecuaciones, y el procedimiento descrito elimina la primera variable utilizando la primera ecuación. Mucha y Sankowski proporcionaron una $O(n^3)$ algoritmo mostrado en el algoritmo 12.7 que encuentra la combinación perfecta de un gráfico simple no dirigido, basado en el algoritmo Rabin- Vazirani , utilizando el método que hemos descrito [16].

Algorithm 12.7 MS_AlgMatch

```
1: Input :  $G(V, E)$ 
2: Output:  $M$ 
3:
4:  $M \leftarrow \emptyset$ 
5:  $A \leftarrow T_G^{-1}$ 
6:  $G' = (V', E') \leftarrow G = (V, E)$ 
7: while  $G' \neq \emptyset$  do
8:   find  $i$  such that  $(v_i, v_j) \in E'$  and  $a_{ij} \neq 0$ 
9:    $M \leftarrow M \cup \{(v_i, v_j)\}$ 
10:   $G' \leftarrow G' - \{v_i, v_j\}$ 
11:  eliminate  $i$ -th row and  $j$ -th column of  $A$ 
12:  eliminate  $j$ -th row and  $i$ -th column of  $A$ 
13: end while
```

12.6 Notas del capítulo

Hemos revisado primero los algoritmos algebraicos, luego los gráficos dinámicos y, finalmente, los algoritmos dinámicos de gráficos algebraicos en este capítulo. Una clase de algoritmos de gráficos algebraicos se basa en gran medida en matrices relacionadas con un gráfico y operaciones en ellas para resolver un problema de gráfico. Podemos ver que las funciones de biblioteca de matriz ya disponibles pueden usarse para tales problemas, ya sea en operaciones secuenciales o paralelas. Los algoritmos de grafos algebraicos proporcionaron soluciones que pueden tener peores rendimientos que las contrapartes clásicas, pero son mucho más fáciles de paralelizar que las clásicas. La multiplicación de matrices se usa frecuentemente para resolver varios problemas de grafos como hemos visto. Como esta operación de matriz básica se puede paralelizar convenientemente, podemos deducir que estos problemas se pueden paralelizar más fácilmente que los algoritmos de grafos tradicionales descritos hasta ahora. Los algoritmos de gráficos dinámicos brindan soluciones más eficientes que las estáticas cuando hay un cambio en la estructura del gráfico. El diseño de estructuras de datos sofisticadas es crucial para los algoritmos de gráficos dinámicos, ya que las actualizaciones y consultas dependen en gran medida de las estructuras de datos utilizadas. Revisamos métodos básicos para gráficos no dirigidos que incluyen esparsificación, aleatorización y agrupamiento; y gráficos dirigidos con árboles de accesibilidad, estructuras de datos matriciales, rutas locales más cortas y rutas largas. Luego revisamos los algoritmos para la conectividad dinámica y la correspondencia dinámica. Nuestro último tema de revisión fueron los algoritmos de gráficos algebraicos dinámicos que trabajan en gráficos dinámicos usando métodos de álgebra lineal. Volvimos a ver dos problemas principales; Conectividad y emparejamiento. Vimos cómo una simple modificación del algoritmo de concordancia algebraica Rabin- Vazirani usando un método básico nuevamente del álgebra lineal condujo a una solución más eficiente.

Estos temas son áreas de estudio relativamente más recientes que los algoritmos de gráficos estáticos que hemos visto hasta ahora y han sido el foco de muchos estudios recientes. Nuestro objetivo principal en el análisis de estos temas es proporcionar una encuesta general con ejemplos para dar una idea de los conceptos relacionados en lugar de ser exhaustivos. Una buena revisión de algoritmos de gráficos algebraicos se proporciona en [13]. En [6 , 7] se proporciona un estudio detallado de gráficos dinámicos y algoritmos de gráficos dinámicos , y la teoría algebraica relacionada con los gráficos se presenta en [1]. En [20] se proporciona un análisis exhaustivo de las operaciones de matriz dinámica para algunos problemas de gráficos .

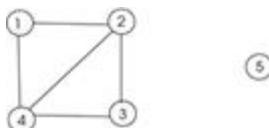


Fig. 12.5 Gráfico de muestra para el ejercicio 1

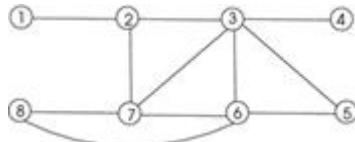


Fig. 12.6 Gráfico de muestra para el Ejercicio 2

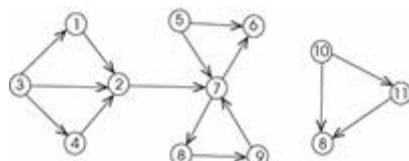


Fig. 12.7 Gráfico de muestra para el ejercicio 3

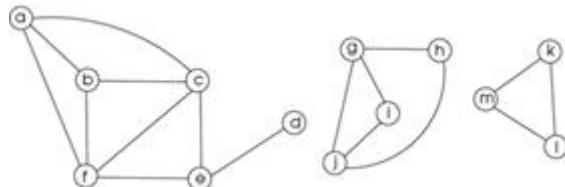


Fig. 12.8 Gráfico de muestra para el Ejercicio 4

Ceremonias

Encuentre la matriz laplaciana para el gráfico de muestra que se muestra en la Fig. [12.5](#) y calcule los valores propios y los vectores propios de este gráfico.

- 1.
2. Calcule el algoritmo BFS algebraico en el gráfico de muestra representado en la Fig. [12.6](#) para el vértice fuente 2. Muestre los contenidos de la matriz de vecindad N en cada iteración. Describe cómo ejecutar este algoritmo en paralelo usando dos procesos p_0 y p_1 usando esta gráfica como ejemplo.
3. Descubra los SCC del gráfico dirigido representado en la Fig. [12.7](#) utilizando su matriz de conectividad y su transposición.
4. Forme las estructuras de datos de búsqueda de unión como árboles para la gráfica de la figura [12.8](#). Muestre la operación de buscar (c), conectar (d , f) e insertar (e , k) usando esta estructura de datos en este gráfico.
5. Dibuje una versión paralela del algoritmo Rabin– Vazirani para una coincidencia perfecta para ejecutar usando k procesos p_0, \dots, p_{k-1} en ordenadores de memoria distribuida. Escriba el pseudocódigo para un proceso mostrando explícitamente la comunicación entre procesos . Puede asumir un modelo maestro / esclavo o totalmente distribuido de un sistema de procesamiento paralelo.

Referencias

1. Bapat RB (2014) Gráficos y matrices (Universitext), 2^a ed . Springer, Berlín (Capítulos 3 y 4)
2. Baswana S, Gupta M, Sen S (2011) Coincidencia máxima totalmente dinámica en el tiempo de actualización O ($\log n$). En: 52.o simposio anual de IEEE sobre fundamentos de la informática FOCS 2011, pp 383–392

3. Bhattacharya S, Henzinger M, Nanongkai D (2017) Coincidencia máxima aproximada completamente dinámica y cobertura de vértice mínima en $\mathcal{O}(\log^3 n)$ peor tiempo de actualización de caso. En: 28º simposio ACM SIAM sobre algoritmos discretos (SODA17), pp 470–489
5. Demetrescu C, Italiano GF (2004) Un nuevo enfoque para las rutas más cortas dinámicas de todos los pares. *J Assoc Comput Mach (JACM)* 51 (6): 968–992
[MathSciNet Crossref](#)
6. Demetrescu C, Italiano GF (2006) Totalmente dinámico todos los pares de rutas más cortas con pesos de borde reales. *J Comput Syst Sci* 72 (5): 813–837
[MathSciNet Crossref](#)
7. Demetrescu C, Finocchi I, Italiano GF (2004) Gráficos dinámicos. Manual de estructuras de datos y aplicaciones. Serie de informática y ciencias de la información. Chapman y Hall / CRC, Boca Raton (Sect. 36)
8. Demetrescu C, Finocchi I, Italiano GF (2013) Algoritmos de gráficos dinámicos, 2ª ed . Manual de teoría de grafos. Chapman y Hall / CRC, Boca Raton (Sect. 10-2)
9. Eppstein D, Galil Z, Italiano GF, Nissenzweig A (1997) Sparsification es una técnica para acelerar algoritmos de gráficos dinámicos. *J Assoc Comput Mach* 44: 669–696
[MathSciNet Crossref](#)
10. Fiedler M (1973) Conectividad algebraica de gráficos. *Matemáticas checoslovacas J* 23: 298-305
[Matemáticas Matemáticas](#)
11. Frederickson GN (1985) Estructuras de datos para la actualización en línea de árboles de expansión mínima, con aplicaciones. *SIAM J Comput* 14 (4): 781–798
[MathSciNet Crossref](#)
12. Henzinger MR, King V (1999) Algoritmos de gráficos aleatorios completamente aleatorios con tiempo polilogarítmico por operación. *J ACM* 46 (4): 502–516
[MathSciNet Crossref](#)
13. Holm J, de Lichtenberg K, Thorup M (2001) Algoritmos totalmente dinámicos determinista polilogarítmicos para conectividad, árbol de expansión mínimo, 2 bordes y biconnectividad . *J Assoc Comput Mach* 48 (4): 723–760
[MathSciNet Crossref](#)
14. Kepner J, Gilbert J (eds) (2011) Algoritmos gráficos en el lenguaje del álgebra lineal. SIAM
[MATES](#)
15. Lovasz L (1979) Sobre determinantes, emparejamientos y algoritmos aleatorios. En: Budach L (ed) Fundamentos de la teoría de la computación. Akademia-Verlag , Berlín
- dieciséis. Lovasz L, Plummer M (1986) Teoría del emparejamiento. Prensa Académica, Budapest
18. Mucha M, Sankowski P (2006) Máximos emparejamientos en gráficos planos mediante eliminación gaussiana. *Algorithmica* 45 (1): 3–20
[MathSciNet Crossref](#)
19. Neiman O, Solomon S (2013) Algoritmos deterministas simples para una coincidencia máxima totalmente dinámica. En: Actas del simposio ACM sobre teoría de la computación (STOC'13), pp 745-754
20. Onak K, Rubinfeld R (2010) Manteniendo una cubierta grande y una cubierta de vértice pequeña. En: Actas del simposio ACM sobre teoría de la computación (STOC'10), pp 457–464.

21. Rabin MO, Vazirani VV (1989) Máximos emparejamientos en gráficos generales a través de la aleatorización. *J Algoritmos* 10 (4): 557–567
[MathSciNet Crossref](#)
22. Sankowski P (2005) Algoritmos de grafos algebraicos. Doctor en Filosofía. Tesis, Universidad de Varsovia, Facultad de Matemáticas, Información y Mecánica.
26. Solomon S (2016) Máxima coincidencia máxima dinámica en tiempo de actualización constante. En: Actas de FOCS, pp 325–334
27. Tarjan R (1975) Eficiencia de un buen algoritmo de unión de conjuntos no lineales. *J ACM* 22 (2): 215-225
[MathSciNet Crossref](#)
28. Thorup M (2000) Conectividad de gráficos totalmente dinámica casi óptima. En: Actas del trigésimo segundo simposio anual de la ACM sobre Teoría de la computación. ACM Press, pp 343–350
29. Tutte WT (1947) La factorización de gráficos lineales. *J Lond Math Soc* s1–22 (2): 107–111
[MathSciNet Crossref](#)
30. Zavlanos MM, Pappas GJ (2005) Controlando la conectividad de gráficos dinámicos. En: Actas de la 44^a conferencia del IEEE sobre decisión y control y conferencia de control europeo, Sevilla, España, diciembre de 2005, pp 6388–6393

13. Análisis de gráficos grandes

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

El análisis de estos gráficos requiere la introducción de nuevos parámetros y métodos conceptualmente diferentes a los utilizados para gráficos relativamente más pequeños. Describimos nuevos parámetros y métodos para el análisis de estos gráficos y también describimos varios modelos para representarlos en este capítulo. Dos modelos ampliamente utilizados para los gráficos grandes que representan redes reales son modelos de escala pequeña y de mundo pequeño. El primero significa que la distancia promedio entre dos nodos cualquiera en los gráficos grandes es pequeña y solo existen pocos nodos con grados altos, con la mayoría de los nodos que tienen grados bajos en el segundo.

13.1 Introducción

Los gráficos grandes constan de miles de vértices y decenas de miles de bordes. El análisis de estos gráficos requiere la introducción de nuevos parámetros y métodos conceptualmente diferentes a los que hemos revisado hasta este punto. La descripción global de gráficos grandes es muy difícil debido a los grandes tamaños involucrados. Una forma de abordar este problema es seleccionar una muestra y un subgrafo representativo de un gráfico dado, analizarlo y extender los resultados obtenidos a todo el gráfico. Sin embargo, la selección de la muestra es un problema en sí mismo y la confiabilidad de extrapolar los resultados del análisis es otro tema a considerar. Alternativamente, y más comúnmente, podemos analizar las propiedades locales de los vértices y bordes en estos gráficos y usar los resultados obtenidos para tener una idea de la estructura general del gráfico.

Comenzamos este capítulo definiendo algunos parámetros nuevos para el análisis de grandes gráficos. Las grandes redes reales representadas por grandes gráficos tienen algunas propiedades interesantes. Estas redes, comúnmente llamadas redes complejas , exhiben un mundo pequeño y no tienen escala. estructuras El primero significa que la distancia entre cualquiera de los dos nodos en estas redes es pequeña en comparación con el número de nodos que tienen y la propiedad sin escala se representa por la existencia de pocos nodos de muy alto grado y muchos nodos de bajo grado. Estos atributos no se encuentran en redes aleatorias y proporcionamos una breve revisión de estos modelos de redes de la vida real. Los principales tipos de redes complejas son las redes tecnológicas,

las redes biológicas y las redes sociales, como analizaremos en el siguiente capítulo. Describimos el concepto de centralidad que proporciona la asignación de valores de importancia a vértices y bordes según su uso en una red y revisamos los algoritmos principales para evaluar las centralidades en las redes. La agrupación en clúster proporciona agrupar objetos similares utilizando alguna medida de similitud. El agrupamiento de gráficos es el proceso de detección de regiones densas de un gráfico determinado. Definimos parámetros para evaluar la calidad de la salida de cualquier método de agrupación y revisamos algunos algoritmos básicos para encontrar agrupaciones y estructuras similares a agrupaciones en los gráficos de la última parte de este capítulo.

13.2 Parámetros principales

El análisis de grandes gráficos es difícil debido a la cantidad de datos que los representan. Sin embargo, podemos analizar las propiedades locales en estos gráficos con el objetivo de deducir sus propiedades globales. Necesitamos definir algunos parámetros nuevos para la evaluación de propiedades globales de gráficos grandes como se describe en las siguientes secciones.

13.2.1 Distribución de Grados

Definición 13.1 (distribución en grados) La distribución en grados de una gráfica G es la relación entre el número de vértices con grado k y el número total de vértices.

Básicamente, este parámetro muestra la probabilidad de que un vértice seleccionado al azar tenga un grado k . Se puede formular de la siguiente manera:

$$P(k) = \frac{n_k}{n},$$

(13.1)

dónde n_k es el número de vértices con grado k y n es el número de vértices. El trazado de P (k) en función de los grados proporciona un análisis visual de la distribución de los grados de vértice de la gráfica. Para un gráfico aleatorio, esperamos que la distribución de grados sea binomial con un pico alrededor del grado promedio del gráfico. Para gráficos que representan muchas redes de la vida real, vemos distribuciones bastante interesantes que son radicalmente diferentes a las del binomio. En la Fig. 13.1 se muestra una distribución de grados de una gráfica simple.

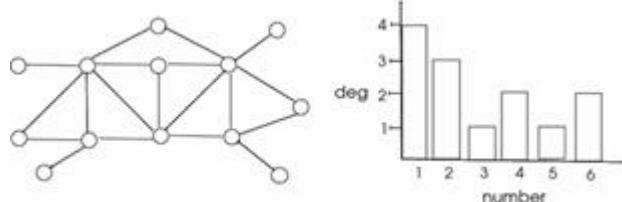


Fig. 13.1 Distribución en grados de un gráfico simple.

En una red surtida , un nodo tiene una alta probabilidad de ser vecino de un nodo con un grado similar. Por ejemplo, los nodos de una red social son personas y esta propiedad se exhibe en dichas redes, ya que una persona con muchos amigos tiene una alta

probabilidad de tener a otra persona con muchos amigos como amigo, como en el caso de las celebridades que se conocen entre sí. En las redes desasortadas , un vértice de alto grado se une comúnmente al vértice con un grado bajo como en el caso de redes biológicas como las redes de interacción de proteínas.

13.2.2 Densidad de gráfico

La densidad de una gráfica G , $\rho(G)$, se define como la relación entre el tamaño de sus bordes existentes y el máximo posible de tamaño de bordes que puede existir de la siguiente manera:

$$\rho(G) = \frac{2m}{n(n-1)}$$

(13.2)

En una gráfica no dirigida, la suma de los grados es igual a $2m$ por el teorema de apretón de manos (vea el Capítulo 2) y el grado promedio de una gráfica, grados (G), es la suma de todos los grados dividida por el número de sus Los vértices son $2m / n$. Sustitución en ec. 13.2 rendimientos

$$\rho(G) = \frac{\deg(G)}{n-1} \approx \frac{\deg(G)}{n} \text{ when } n \text{ is large}$$

(13.3)

En una gráfica densa , $\rho(G)$ es estable cuando n se incrementa a valores muy grandes y $\rho(G)$ se acerca a 0 con grandes valores de n en gráficos dispersos . El grado promedio de la gráfica en la figura 13.1 es $38/13 = 2.9$.

13.2.3 Coeficiente de agrupamiento

El coeficiente de agrupamiento es una propiedad local definida para un vértice en un gráfico de la siguiente manera.

Definición 13.2 (coeficiente de agrupamiento) El coeficiente de agrupamiento $CC(v)$ de un vértice v en un gráfico G es la relación entre el número de bordes entre los vecinos de v y el número máximo posible de conexiones entre estos vecinos.

Dado que el número máximo posible de conexiones entre k vértices en un gráfico es $k(k-1)/2$, el coeficiente de agrupamiento $CC(v)$ de un vértice v se define de la siguiente manera.

$$CC(v) = \frac{2n_v}{|N(v)|(|N(v)|-1)},$$

(13.4)

dónde n_v es el número de aristas entre los vecinos del vértice v . Este parámetro muestra básicamente qué tan bien están conectados los vecinos de un vértice y, por lo tanto, su

tendencia a formar una camarilla. En una red social, por ejemplo, el coeficiente de agrupamiento de una persona proporciona una evaluación de la cantidad de amigos de esa persona que son amigos entre sí. El coeficiente de agrupamiento promedio de una gráfica G , $CC(G)$, es el valor medio de todos los coeficientes de agrupamiento de vértices de la siguiente manera:

$$CC(G) = \frac{1}{n^2} \sum_{v \in V} CC(v),$$

(13.5)

dónde n^2 es el número de nodos que tienen un grado de dos o más. Para un vértice v con un grado menor que dos, $CC(v)$ a veces se considera uno o cero y, en este caso, el denominador en la ecuación anterior se puede tomar como n . Si este parámetro es alto en un gráfico G , podemos deducir que los vértices de G están bien conectados y, por lo tanto, G es un gráfico denso. Los coeficientes de agrupación de vértices de un gráfico de muestra se representan en la Fig. 13.2.

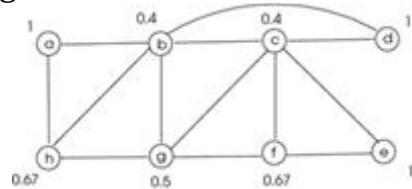


Fig. 13.2 Valores de coeficiente de agrupamiento de los vértices de un gráfico de muestra

La transitividad $T(G)$ de una gráfica. $G = (V, E)$ como se propone en [12] evalúa qué tan bien están conectados los vecinos de los vértices de una gráfica. Deja un subgrafo triángulo de un gráfico de G sea $G_t = (V_t, E_t)$ con $V_t = \{v_1, v_2, v_3\}$ y $E_t = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$. Un triplete es un subgrafo de tres vértices $G_r = (V_r, E_r)$ con $V_r = \{v_1, v_2, v_3\}$ y $E_r = \{(v_1, v_2), (v_2, v_3)\}$ con v_2 en el medio. Cada triángulo contiene tres tripletes; La transitividad de un gráfico ahora se puede definir de la siguiente manera:

$$T(G) = \frac{3 \times \text{number of triangles}}{\text{number of connected triplets}}$$

(13.6)

En la Fig. 13.3 se muestra un gráfico simple de cuatro vértices. Los coeficientes de agrupamiento se dan junto a los vértices. El coeficiente de agrupamiento de gráficos es el promedio de estos valores, que produce $\frac{(2 + \frac{4}{3})/4 = 5/6}{3} = 0.75$. Hay dos triángulos en este gráfico y ocho tripletes, contando tres tripletes por triángulo e incluyendo $\{d, a, b\}$ y $\{a, b, c\}$. Trillizos dando un total de ocho trillizos. Por lo tanto, la transitividad de este gráfico es $\frac{3 \times 2/8 = 0.75}{3} = 0.75$.

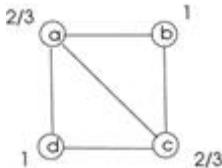


Fig. 13.3 Comparación de la transitividad y el coeficiente de agrupamiento en un gráfico de muestra

13.2.4 Índice de juego

Una forma de evaluar la similitud entre los dos vértices de una gráfica es encontrar el número de sus vecinos comunes. El índice de coincidencia definido a continuación se utiliza para cuantificar esta similitud.

Definición 13.3(índice coincidente) El índice coincidente de dos vértices u y v en un gráfico G se define como la relación del número de sus vecinos comunes al número de la unión de sus vecinos.

En la figura [13.2](#), el índice coincidente de los vértices b y f , m_{bf} , es 0.33 ya que la unión de su conjunto vecino es $N(b) \cup N(f) = \{a, c, d, e, g, h\}$ con un tamaño de 6 y tienen dos vecinos comunes, c y g . Los vértices en una gráfica pueden estar muy separados, especialmente en el caso de redes muy grandes, en cuyo caso los vecinos comunes pueden no existir. En tal caso, proponemos el parámetro de índice de coincidencia de k -hop que es básicamente la proporción de vecinos comunes de dos vértices en el vecindario k -hop a todos sus vecinos en dicho vecindario. Podemos evaluar este parámetro de forma secuencial, y también en una configuración distribuida. Los siguientes pasos de algoritmo distribuido para un nodo i de un sistema distribuido se pueden emplear para este propósito. El algoritmo puede ser implementado en k rondas sincronas bajo el control de un supervisor utilizando el modelo SSI.

```
degs[1] ← deg(i)
```

- 1.
2. para $i = 1$ a k
3. enviar grados [k] a $N(i)$
4. recibir $degs[k + 1]$ de $N(i)$
5. **final para**
6. $comun \leftarrow$ vecinos comunes en grados [1 .. k]
7. $all \leftarrow$ vecinos en grados [1 .. k]
8. $m_i \leftarrow \frac{|comun|}{|all|}$

13.3 Modelos de red

Los modelos de red se pueden clasificar como redes aleatorias, redes de mundo pequeño, redes sin escala y redes jerárquicas.

13.3.1 Redes aleatorias

Un modelo de red aleatorio asume que los bordes entre nodos se insertan aleatoriamente. En el modelo gráfico aleatorio básico, $G(n, p)$, propuesto por Erdos - Renyi, hay n vértices y cada borde se agrega sucesivamente entre dos vértices

con probabilidad p independientemente [7 , 8]. Para generar una red aleatoria basada en este modelo $G(n, p)$, se aplican los siguientes pasos:

Hay n nodos aislados inicialmente.

- 1.
2. Seleccione un par de nodos u y v de estos y genere un número aleatorio $0 \leq x \leq 1$. Si $x \geq p$ luego conecte u y v para formar el borde (u, v) , de lo contrario, déjelos desconectados.
3. Repita el paso 2 para cada uno de los $\frac{n(n-1)}{2}$ pares de vértices.

Tendremos una red aleatoria diferente con los mismos valores de n y p para cada generación. La distribución de grados en redes aleatorias es binomial centrada alrededor del grado promedio y estas redes también muestran un coeficiente de agrupamiento promedio pequeño con un diámetro bajo con respecto al número de nodos.

13.3.2 Redes del mundo pequeño

Una red de mundo pequeño tiene un diámetro pequeño en comparación con su tamaño, lo que significa que el alcance desde cualquier nodo a cualquier otro en una red de este tipo se puede realizar con pocos saltos. Esta propiedad se observa en muchas redes grandes, como las redes sociales y las redes biológicas. Esta es una propiedad útil en redes grandes, ya que es posible la comunicación rápida entre dos nodos. La propiedad del mundo pequeño se caracteriza por un bajo valor de la longitud promedio de la trayectoria l definida como la media de las distancias entre todos los pares de n nodos en una gráfica $G = (V, E)$ como a continuación:

$$l = \frac{1}{n(n-1)} \sum_{u,v \in V, u \neq v} d(u, v)$$

(13.7)

En una red de mundo pequeño, el valor de l debe estar limitado por $\log n$. Esta propiedad en redes complejas se asoció comúnmente con un alto coeficiente de agrupamiento promedio que indica la presencia de agrupamientos, como lo muestran Watts y Strogatz [4 , 18].

13.3.3 Redes libres de escala

Una propiedad interesante observada en grandes gráficos que representan redes de la vida real es la distribución de grados mostrada por $P(k) \approx Ak^{-\gamma}$ con $2 < \gamma < 3$. En lugar de una distribución binomial de una red aleatoria. En términos prácticos, esto significa que hay pocos nodos llamados concentradores con grados muy altos, con la mayoría de los nodos que tienen grados pequeños en estas redes. Esta distribución de los vértices en grados de ley de potencia se puede generar siguiendo las reglas de crecimiento y vinculación preferencial como en el siguiente algoritmo:

Crecimiento : se genera un nuevo vértice v y se conecta a uno de los vértices existentes con la siguiente regla.

- 1.
2. Acoplamiento preferencial : el vértice v se adjunta a uno de los vértices existentes, digamos u , con una probabilidad relacionada con el grado del vértice u .

Comenzando con un pequeño número de vértices, se obtiene un gráfico sin escala cuando estos dos pasos se repiten el número suficiente de veces como lo demuestran Barabasi y Albert [1].

Definamos la función de agrupamiento C (k) de un gráfico G como el coeficiente de agrupamiento promedio de los nodos con grado k . En diversas redes biológicas,

$C(k) \approx k^{-1}$ mostrar que el parámetro del coeficiente de agrupamiento es mayor en los nodos de grados más bajos que en los nodos de grados más altos. Básicamente, esto significa que los nodos de grado superior tienen vecinos que están menos conectados que los vecinos de los nodos de grado inferior. Este nuevo modelo se denominó redes jerárquicas en las que los nodos de bajo grado se agrupan densamente y estas regiones están conectadas por nodos de alto grado [6 , 15].

13.4 centralidad

La centralidad de un vértice o una arista en un gráfico denota un valor que muestra su importancia en el gráfico. La importancia puede tener diferentes significados; por ejemplo, la centralidad de un borde puede reflejar el porcentaje de caminos más cortos a través de él. Revisamos los principales parámetros de centralidad para los gráficos en las siguientes secciones.

13.4.1 Grado de centralidad

El indexDegree El grado de centralidad de un vértice se denota como su centralidad de grado . Un vértice con un valor más alto que el grado promedio de una gráfica da una idea de su importancia en la red. En general, es difícil predecir la estructura general de una gráfica a partir del valor de su grado promedio. Sin embargo, la gráfica de su secuencia de grados proporciona información sobre la organización general de la gráfica. Dada la matriz de adyacencia A de una gráfica $G = (V, E)$, podemos formar el vector de centralidad DC que tiene el valor de centralidad DC (i) para el vértice i de la siguiente manera:

$$DC = A \times [1]$$

(13.8)

En un entorno distribuido, podemos tener un proceso de raíz. p_i reuniendo todos los valores de grado de los nodos sobre un árbol de expansión T construido previamente utilizando la operación de convergecast con el modelo SSI. Este proceso puede entonces calcular el promedio grado gráfica \bar{G} y la centralidad del vector C . Al recibir un mensaje de inicio desde la raíz sobre el árbol T , la función de los nodos es simplemente hacer una convergencia de sus grados sobre los bordes de T hasta la raíz.

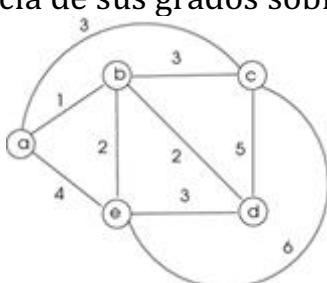


Fig. 13.4 Los valores de centralidad de proximidad para los vértices a , b , c , d y e en este gráfico son 1/10, 1/9, 1716, 1/16 y 1 / 13 respectivamente

13.4.2 Centralidad de proximidad

En el atributo de centralidad de proximidad , evaluamos la distancia de un vértice v a todos los otros vértices en el gráfico, asumiendo que un vértice que está más cerca de todos los otros vértices en el gráfico es más importante que un vértice que no está tan cerca de todos los demás. El valor de proximidad de un vértice se define de la siguiente manera:

$$CC(v) = \frac{1}{\sum_{u \in V} d(u, v)}$$

(13.9)

Sumamos la distancia de cada vértice al vértice v y tomamos el recíproco de este valor. Una posible forma de evaluar las centralidades de proximidad de todos los vértices en un gráfico es calcular las rutas APSP en un gráfico utilizando una versión modificada del algoritmo de Dijkstra que vimos en el Cap. [7](#) para este fin. Necesitamos agregar el cálculo de las sumas de distancias al asignar un vértice al conjunto de rutas decididas (Ver Ejercicio 4). En una configuración paralela o distribuida, podemos usar los algoritmos descritos en el Capítulo. [7](#) con la modificación descrita (Ver Ejercicio 5). Si el gráfico no está ponderado, el algoritmo BFS se puede utilizar con una modificación similar. Figura [13.4](#) muestra un ejemplo de gráfico ponderado no dirigido desde el cual se pueden calcular los valores de centralidad de proximidad utilizando las rutas más cortas.

13.4.3 Centralidad de intermediación

En otro intento de evaluar la importancia de un vértice o un borde en un gráfico, podemos evaluar el uso de un vértice o un borde cuando se necesitan comunicaciones entre vértices de un gráfico. El valor de centralidad de vértice de un vértice es el porcentaje de rutas más cortas entre todos los pares de vértices que se ejecutan a través de ese vértice. De manera similar, el valor de intermediación de borde de un borde se refiere al porcentaje de rutas más cortas que pasan por ese borde. Describiremos definiciones formales y algoritmos para encontrar estos parámetros en gráficos en las siguientes secciones.

13.4.3.1 Centralidad de la intermediación del vértice

El intervalo entre vértices de un vértice v en un gráfico G es la relación de las rutas más cortas que van a través de v al número total de rutas más cortas en G de la siguiente manera:

$$BC(v) = \sum_{s, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}},$$

(13.10)

con $\sigma_{st}(v)$ como el número total de rutas más cortas entre los vértices s y t que se ejecutan a través de vértice v , y σ_{st} como el número total de rutas más cortas entre los vértices s y t . Para un gráfico no ponderado, simplemente podemos ejecutar el algoritmo

BFS para cada vértice, contar el número de la ruta más corta a través de cada vértice y dividir este número por el número total de rutas más cortas en el gráfico. Para un gráfico ponderado, necesitamos ejecutar un algoritmo APSP. Los valores de intermediación del vértice de un gráfico de muestra se muestran en la Fig. 13.5 .

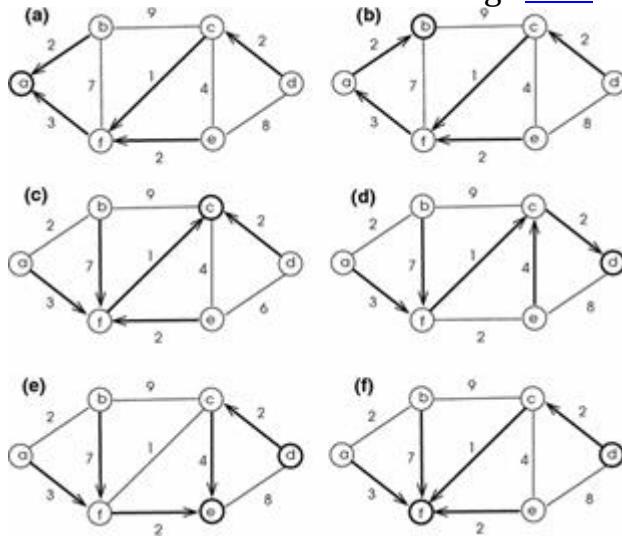


Fig. 13.5 Cálculos de valores de intermediación de vértices de una gráfica de muestra

Podemos contar el número de rutas más cortas a través de cada vértice excluyendo los vértices inicial y final para encontrar 3, 0, 7, 0, 0 y 9 para los vértices a , b , c , d y e , respectivamente. Hay un total de 15 rutas más cortas entre cada par de vértices y los valores de intermediación de vértice para estos vértices son entonces 0.2, 0, 0.47, 0, 0 y 0.6, respectivamente, en orden lexicográfico. Podemos concluir que el vértice f es el vértice más influyente, ya que el mayor número de caminos lo atraviesan, lo que de hecho puede detectarse visualmente.

13.4.3.2 Centralidad de intermediación de borde

El límite entre el borde de un borde e en un gráfico G es la relación de las rutas más cortas que se ejecutan a través de e al número total de rutas más cortas en G de la siguiente manera:

$$BC(e) = \sum_{x \in \text{borde}} \frac{\sigma_{xt}(e)}{\sigma_{xt}}$$

(13.11)

Ahora vamos a describir un algoritmo debido a Newman y Girvan [11] para calcular borde intermediación valores de centralidad de bordes en un gráfico. Este algoritmo tiene dos partes: una asignación de peso de vértice y fases de asignación de peso de borde. Los vértices están etiquetados con el número de rutas más cortas que los atraviesan primero y luego esta información se usa para encontrar ponderaciones de borde para obtener valores de centralidad de borde más adelante. La primera parte del algoritmo se representa en el algoritmo 13.1 que funciona para un nodo fuente que tiene una distancia de 0 y un peso de 1 inicialmente [6]. El procedimiento de asignación de peso de vértice luego asigna iterativamente valores de distancia a los vértices como en un algoritmo BFS con valores de peso de vértice adicionales. Sin embargo, si un vértice u se visita antes y

tiene un peso uno más que el peso de su antepasado v , su peso se hace igual a la suma de su peso anterior y el peso de su antepasado. Esto es necesario ya que un camino más corto alternativa a la fuente de vértice s es descubierto a través de la ancestro vértice v . Necesitamos repetir este procedimiento para todos los vértices con cada uno como el vértice de origen en una iteración y el peso del vértice de un vértice v es la suma de todos los valores asignados a él en cada ejecución del procedimiento.

Algorithm 13.1 Label_Vertex

```

1: Input :  $G(V, E)$ 
2: Output :  $\forall v \in V : w_v$ , weight of vertex  $v$ 
3:  $d_s \leftarrow 0; w_s \leftarrow 1$                                 > initialize source vertex  $s$ 
4: for all  $v \in N(s)$  do
5:    $d_v \leftarrow d_s + 1; w_v \leftarrow 1$                       > initialize neighbors of vertex  $s$ 
6: end for
7: repeat
8:   for all  $u \in N(v)$  do
9:     if  $u$  is not assigned a distance then
10:     $d_u \leftarrow d_v + 1; w_u \leftarrow w_v$                   > test multiple shortest paths
11:   else if  $d_u = d_v + 1$  then
12:      $w_u \leftarrow w_v + w_u$ 
13:   end if
14:   end for
15: until all vertices have assigned distances

```

El segundo procedimiento del algoritmo utiliza los pesos de vértice asignados para denotar pesos de borde. Comienza asignando pesos a los bordes que terminan en vértices de hojas como la relación entre el peso del vértice de su antepasado y el peso de sí mismo. Básicamente, estamos denotando pesos que muestran el porcentaje de rutas más cortas hacia los bordes de las hojas. Luego, a medida que avanzamos hacia arriba en el árbol hacia el vértice de origen s , a cada borde (u, v) se le asigna un peso que es la suma de todos los pesos de los bordes debajo (u, v) multiplicado por la relación del peso del vértice u al peso del vértice v . De nuevo, la suma de todos los caminos más cortos a través del borde (u, v) se escala para dar el porcentaje de rutas más cortas a través de ese borde como se muestra en el algoritmo 13.2. Este proceso debe repetirse para todos los vértices teniendo en cuenta que cada uno de ellos como fuente y el valor del intervalo entre aristas de un borde se determina como la suma de todos los valores encontrados para ese borde. El tiempo total necesario es $O(nm)$ considerando n vértices. En la Fig. 13.6 se muestran los valores de separación entre bordes de los bordes de un gráfico de muestra para un vértice de una sola fuente .

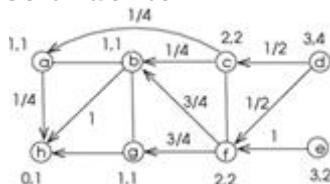


Fig. 13.6 Edge intermediación valores de los bordes de un gráfico que muestra para vértice fuente h . La distancia del vértice, los valores de peso obtenidos del primer procedimiento se muestran junto a los vértices con los valores de separación entre bordes obtenidos por el segundo procedimiento mostrado en los bordes

Algorithm 13.2 Edge_Betweenness

```

1: Input :  $G(V, E)$ 
2: Output :  $\forall v \in V : w_v$ , weight of vertex  $v$ 
3: for all  $v \in V$  which is a leaf vertex do
4:   for all  $u \in N(v)$  do  $w_{uv} \leftarrow w_u/w_v$           > initialize leaf edges
5:   end for
6: end for
7: repeat
8:   move upwards to the source vertex such that  $u$  is closer to  $s$ ;
9:    $w_{uv} \leftarrow (1 + \text{sum of the edge weights below } v) \times w_u/w_v$ 
10: until vertex  $s$  is reached

```

13.4.4 Centralidad de valores propios

La idea general de la centralidad del valor propio es atribuir importancia a un nodo si tiene vecinos importantes, como ocurre con frecuencia en una red social. Las puntuaciones de importancia de los vecinos de un nodo i afectan su puntuación de la siguiente manera [6]:

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j = \frac{1}{\lambda} \sum_{j \in V} a_{ij} x_j,$$

(13.12)

donde $N(i)$ es el conjunto de vecinos de nodo i y a_{ij} Es la entrada ij -th de la matriz de adyacencia A del gráfico. $G = (V, E)$ y λ es una constante Podemos reescribir esta ecuación en notación matricial de la siguiente manera:

$$Ax = \lambda x,$$

(13.13)

que resulta ser la ecuación de valor propio de la matriz A . Habrá n valores propios y los vectores propios correspondientes asociados con la matriz de adyacencia A . Los valores de centralidad de valores propios de los vértices están determinados por el vector propio correspondiente al valor propio más grande de A, como lo muestra

el teorema de Perron - Probenius [14]. Ahora podemos indicar los pasos de un algoritmo para encontrar las centralidades de valores propios de los vértices en un gráfico $G = (V, E)$ como sigue:

Construir la matriz de adyacencia A de G .

- 1.
2. Calcular valores propios $\lambda_1, \dots, \lambda_n$ de A usando la ecuación $\det(A - \lambda I) = 0$ y seleccione el mayor valor propio λ_{\max} a partir de estos valores propios.
3. Calcular el vector propio x_{\max} asociado con λ_{\max} .

Cada vértice $v_i \in V$ tiene un valor de centralidad eigenvalor $x_i \in X_m$. La centralidad de valores propios se puede utilizar para la clasificación de páginas en la Web y también para investigar asociaciones de genes y enfermedades [13].

13.5 subgrafos densos

Ahora veremos formas de encontrar subgrafos densos de una gráfica dada. Estos subgrafos, a menudo denominados agrupamientos , indican una región de actividad densa en la red representada por el gráfico. En el caso extremo, una camarilla es un subgrafo que está completamente conectado. Sin embargo, las estructuras tipo clique o cliquish son más comunes en la práctica. Revisaremos los métodos para encontrar camarillas y estas estructuras en esta sección.

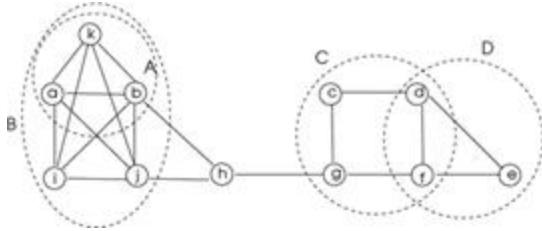


Fig. 13.7 Camarillas de un gráfico de muestra; $A = \{a, b, k\}$ es una camarilla pero no máxima, ya que se incluye en la camarilla más grande $B = \{a, b, k, i, j\}$ que es también la camarilla máxima de la gráfica. $C = \{c, d, f, g\}$ y $D = \{d, e, f\}$ son camarillas máximas ya que no son un subconjunto de camarillas más grandes

13.5.1 camarillas

Un subgrafo en el que cada vértice es un vecino de todos los demás en este subgrafo se denomina camarilla . Una pandilla de un gráfico G es una región densamente conectada en G que puede indicar un tipo especial de actividad en la red que está representada por G , por ejemplo, una pandilla de amigos en una red social. Por lo tanto, la detección de camarillas es una tarea comúnmente requerida en los gráficos que representan fenómenos de la vida real. En las redes de la vida real, a menudo se encuentran estructuras similares a camarillas que a camarillas completas en gráficos que representan redes de la vida real como las redes de interacción de proteínas de la célula y las redes sociales y, por lo tanto, presentamos algoritmos para descubrir dichas estructuras en gráficos.

Definición 13.4(camarilla) una camarilla de un gráfico $G = (V, E)$ es un subconjunto C de sus vértices de tal manera que cada vértice en C es adyacente a todos los otros vértices en C . En otras palabras, una camarilla es un subgrafo completo inducido de G .

Una camarilla máxima de un gráfico G es la camarilla que no es un subconjunto de una camarilla más grande. La camarilla máxima de un gráfico G es la pandilla de G con el orden más grande entre todas las camarillas de G como se muestra en la Fig. 13.7 . El orden de la camarilla máxima de G se denota por $\omega(G)$ y se llama el número clique de G . Encontrar la camarilla máxima en un gráfico se denomina problema de camarilla máxima y es NP-duro [9], y por lo tanto, se utilizan comúnmente varias heurísticas para calcular una aproximación a este parámetro. La versión de decisión de este problema, el problema de k -clique , es determinar si una gráfica tiene un clique con al menos k vértices. Encontrar todas las camarillas máximas de una gráfica también es NP-Duro ya que el número de tales camarillas crece exponencialmente con el orden de las gráficas.

La detección de camarillas en un gráfico tiene varias implicaciones, ya que representan una interacción intensa entre las entidades representadas por los nodos del gráfico. Una camarilla en una red representaría un grupo íntimo de detección que puede ayudar a comprender el comportamiento general de dicha red. Sin embargo, es posible que nos interese encontrar estructuras similares a camarillas en lugar de pandillas en los gráficos, ya que las camarillas ocurren con menos frecuencia y también encontrarlas no es una tarea trivial. Las estructuras tipo camarilla que podemos buscar son las siguientes [6]:

Definición 13.5(k -clique) A k -clique de un gráfico G se muestra mediante un subgrafo G_e , donde el camino más corto entre dos vértices en G_e es a lo mas k . Las rutas pueden consistir en vértices y bordes externos a G_e .

Algunos ejemplos de k - clique se muestran en la figura 13.8 .

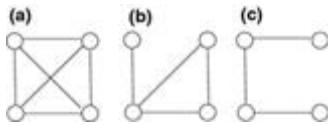


Fig. 13.8 Ejemplos de k -clique , a 1-clique b 2-clique c 3-clique

Definición 13.6(k -club) A k -Club de un gráfico G es un subgrafo G_k en el cual $\text{diam}(G_k) \leq k$.

Cada k -club de una gráfica es también su k -clique. Sin embargo, no todos los k -clique de un gráfico es su k -Club desde un k -clique puede contener vértices fuera de G_k . El problema máxima k -club es encontrar el más grande k -Club de un gráfico y es NP-duro.

Definición 13.7(k -plex) En un subgrafo k -plex $G_k \subseteq G$, cada vértice está conectado al menos a $n-k$ otros vértices en G .

Definición 13.8(k -core) A k -core subgraph G_k de G consiste en vértices que tienen al menos un grado de k . Una camarilla es un núcleo (k -1).

Podemos encontrar algunas estructuras parecidas a una camarilla, como las puntuaciones k en el tiempo polinomial. Primero describimos un algoritmo recursivo que tiene una complejidad de tiempo exponencial que se ha utilizado como base para varios algoritmos paralelos, como veremos. Luego presentamos un algoritmo para encontrar k -puntuaciones de un gráfico que funciona en tiempo polinomial.

13.5.1.1 Algoritmo de Bron y Kerbosch

Bron y Kerbosch proporcionaron un sencillo procedimiento de seguimiento que encuentra todas las camarillas máximas de un gráfico no dirigido [5]. El algoritmo utiliza tres conjuntos de vértices, R es el conjunto de vértices que se incluirán en la camarilla, X es el conjunto que se excluirá de la camarilla y P es el conjunto que se decidirá. El conjunto R es inicialmente vacío y se expande usando vértices en P y sin utilizar vértices en X . Los siguientes pasos comprenden el algoritmo:

Seleccione un vértice candidato $v \in P$.

- 1.
2. Añadir V a R .
3. Crear nueva conjuntos P y X de los viejos conjuntos mediante la eliminación de todos los vértices no conectados a R .
4. Llame al operador de la extensión en los conjuntos recién formados.
5. A su regreso, retire v de R , añadirlo a la X .

El pseudocódigo para este algoritmo se muestra en el algoritmo 13.3 [6]. Nos basta con exponer P a estar vacío tan R no puede extenderse y X a estar vacío para asegurar que la camarilla no está incluido en otra camarilla tener una camarilla máxima en R . Esta condición se verifica en cada llamada recursiva al procedimiento primero. De lo contrario, una llamada recursiva se hace que añade un vértice en P a R y sus vecinos en P y X . Experimentalmente, se encontró que la complejidad del tiempo era $O(4^n)$ y una segunda versión que utiliza pivote resultó en la complejidad del tiempo

de $O(3^{14^k})$ [5]. Existen varias implementaciones paralelas de este algoritmo; usando MPI en [10], usando grupos de subprocesos en Java en [3] y en una supercomputadora Cray XT en [17].

Algorithm 13.3 Bron Kerbosch Algorithm

```

1: procedure BronKerbosch( $P, R, X$ )
2:    $P \leftarrow V$  includes all of the vertices and  $R, X \leftarrow \emptyset$ 
3:   if  $P = \emptyset \wedge X = \emptyset$  then
4:     return  $R$  as a maximal clique
5:   else
6:     for all  $v \in P$  do
7:       BronKerbosch( $R \cup \{v\}, P \cap N(v), X \cap N(v)$ )
8:        $P \leftarrow P \setminus \{v\}$ 
9:        $X \leftarrow P \cup \{v\}$ 
10:    end for
11:   end if
12: end procedure

```

13.5.2 k- puntuaciones

Vamos a considerar un gráfico $G = (V, E)$; un subgrafo $H = (V', E')$ de G inducida por el conjunto de vértices V' es un k -core de G si y solo si $\forall v \in V', \deg(v) \geq k$ y H es el gráfico máximo con esta propiedad [6]. El núcleo principal de un gráfico G es un núcleo de G que tiene el orden máximo y el valor de concordancia o el valor central de un vértice v es el orden más alto de un núcleo que incluye el vértice v [2]. Los núcleos de una gráfica se anidan naturalmente y un núcleo de subgrafo con un valor de núcleo más grande se anida dentro de los núcleos con valores más pequeños como se muestra en la Fig. 13.9.

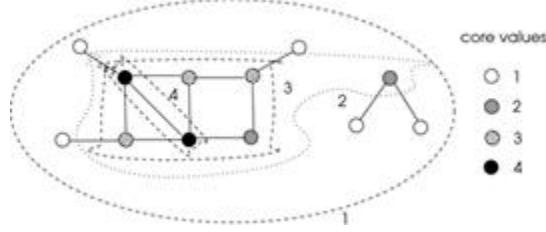


Fig. 13.9 Núcleos de muestra gráfica con dos componentes. Los núcleos 1, 2, 3 y 4 están rodeados y los valores centrales de los vértices se muestran con colores rellenos

Encontrar núcleos de una gráfica es útil para detectar regiones densas de una gráfica, ya que cada vértice en un k -core tiene al menos un grado de k . Por lo tanto, en lugar de tratar de encontrar camarillas o estructuras similares a una camarilla de un gráfico para descubrir regiones densas en ese gráfico, podemos encontrar núcleos de un gráfico que se pueden lograr en tiempo polinomial debido a un algoritmo descrito en la siguiente sección.

13.5.2.1 Algoritmo de Batagelj y Zaversnik

En el algoritmo propuesto por Batagelj y Zaversnik [2], los valores centrales para cada vértice se determinan en tiempo lineal. La idea general de este algoritmo es eliminar todos los vértices que tengan un grado menor que k de la gráfica en consideración para obtener el k -core de la gráfica. El algoritmo comienza ordenando todos los vértices en el gráfico con respecto a sus grados y los vértices se colocan en una cola en grados crecientes. El vértice v de la parte delantera de la cola se elimina, se etiqueta con el valor central que es igual a su grado, y los grados de todos los vecinos de v que tienen grado más que v se decrementan en uno. El grado de cualquier vecino u con el mismo grado de v no disminuye, ya que esto resultaría en que se incluya u en una clase de vértices con un valor k más bajo que el de sí mismo. El pseudocódigo de este algoritmo se representa en el algoritmo 13.4 [2 , 6]. Los autores demostraron que la complejidad del tiempo de este

algoritmo en un gráfico general es $O(\max(m, n))$, y esto se convierte en $O(m)$ en una red conectada como $m \geq n - 1$ en tal grafo.

Algorithm 13.4 BatallejAlg

```

1: Input :  $G = (V, E)$ 
2: Output : core values of vertices
3:  $Q \leftarrow$  sorted vertices of  $G$  in increasing degrees
4: while  $Q \neq \emptyset$  do
5:    $v \leftarrow$  front of  $Q$ 
6:    $\text{core}(v) \leftarrow \deg(v)$ 
7:   for all  $u \in N(v)$  do
8:     if  $\deg(u) > \deg(v)$  then
9:        $\deg(u) \leftarrow \deg(u) - 1$ 
10:      end if
11:    end for
12:   update  $Q$ 
13: end while

```

Veamos el funcionamiento paso a paso de este algoritmo en el gráfico de muestra de la figura 13.10. El contenido de la cola ordenada junto con los vértices eliminados y etiquetados se muestran en la Tabla 13.1 para ejecutar el algoritmo en este gráfico simple.

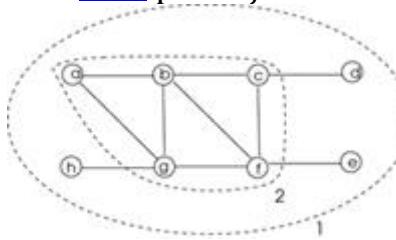


Fig. 13.10 Un ejemplo de gráfica con dos núcleos.

Tabla 13.1 Ejecución del algoritmo BZ

Iteraciones Cola ordenada en grado ascendente Valores de cortesía

1	2	3	4	$k = 1$	$k = 2$
1	h, d, e	una	do	b, f, g	{h}
2	d, e	una	c, g	b, f	{h, d}
3	mi	a, c	sol	b, f	{h, d, e}
4	-	c, g	g, f	segundo	{h, d, e}
5	-	g, b, f	b, f	-	{a}
6	-	b, f	-	-	{h, d, e}
7	-	F	-	-	{a, c}
8	-	-	-	-	{h, d, e}
				-	{a, c, g}
				-	{b, a, c, g}
				-	{b, f, a, c, g}

13.5.3 Agrupación

La agrupación en clúster es el proceso de agrupación de objetos similares en función de alguna métrica. Cuando los nodos de un gráfico se utilizan para representar objetos y los bordes representan sus relaciones, este proceso es equivalente a encontrar subgrafos densos del gráfico que representa la red. En este caso, el objetivo de un algoritmo de agrupamiento es dividir el gráfico en subgrafos de tal manera que la densidad dentro de un subgrafo se maximice con el mínimo número de bordes entre grupos. Cuando se ponderan los bordes, debemos maximizar el número total de pesos de bordes dentro de un grupo y minimizar el peso total de los bordes entre ellos.

Podemos tener un vértice que pertenezca a más de un clúster en la agrupación de gráficos y un vértice puede pertenecer a un solo clúster en la partición de gráficos. Necesitamos evaluar la calidad de los clusters obtenidos después de usar un método de clustering. Una forma conveniente de lograr esto es comparar las densidades de los grupos y la densidad promedio de la gráfica. La densidad $\rho(G)$ de un gráfico simple no

ponderado, no dirigido, G es la relación entre el tamaño de sus bordes y el tamaño de los bordes máximos posibles en G como $\rho(G) = \frac{2m}{n(n-1)}$.

Los bordes dentro de un grupo se llaman bordes internos del grupo y los bordes que conectan los vértices del grupo a los otros vértices del gráfico se llaman bordes externos. Podemos examinar si un vértice v se coloca adecuadamente en un grupo examinando la relación del número de bordes internos incidentes a v al número de bordes externos en él. Ahora podemos definir la densidad intracluster de un cluster C_i como la relación de todos los bordes internos en C_i a todo número posible de bordes en C_i como sigue [16]:

$$\deg_{int}(C_i) = \frac{2 \sum_{v \in C_i} \deg_{int}(v)}{|C_i||C_i| - 1|}$$

(13.14)

La densidad intracluster de todo el gráfico como el promedio de todas las densidades intracluster de la siguiente manera [6]:

$$\deg_{int}(G) = \frac{1}{k} \sum_{i=1}^k \deg_{int}(C_i),$$

(13.15)

donde k es el número de agrupamientos obtenidos. Claramente, necesitaríamos $\deg_{int}(G)$ lo más alto posible en comparación con la densidad del gráfico para una agrupación adecuada. En la Fig. 13.11 se muestra un gráfico de muestra dividido en tres grupos con densidades intracluster calculadas.

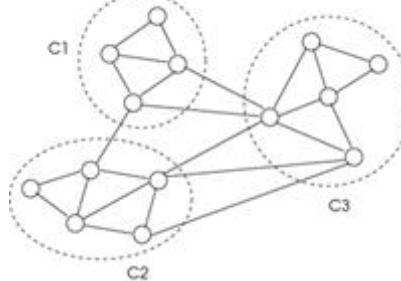


Fig. 13.11 Una gráfica de muestra dividida en tres grupos C_1 , C_2 y C_3 . Las densidades intracluster son $\delta_{int}(C_1) = 0.83$, $\delta_{int}(C_2) = 0.7$ y $\delta_{int}(C_3) = 0.6$ con la densidad media de la gráfica intracluster de 0.71. La densidad del gráfico es 0.28

Ahora podemos definir la densidad intercluster. $\deg_{ext}(G)$ como la relación entre el tamaño de los bordes del interclúster y el número máximo permitido de bordes entre los grupos, como se muestra a continuación [16], y necesitamos que este parámetro sea significativamente más bajo que la densidad del gráfico para un agrupamiento de buena calidad.

$$deg_{ew}(G) = \frac{2 \times \text{sum of inter cluster edges}}{n(n-1) - \sum_{i=1}^k (|C_i||C_i|-1)}$$

(13.16)

La densidad intercluster del gráfico de muestra en la figura [13.11](#) es 0.092, que es significativamente menor que la densidad del gráfico 0.28 y, por lo tanto, podemos considerar que la agrupación en este gráfico es favorable. Se pueden definir parámetros de cluster similares para gráficos ponderados por bordes. Consideremos primero la densidad de un gráfico ponderado por bordes, que es la relación entre la suma de los pesos de bordes y el número máximo posible de bordes como se muestra a continuación:

$$\rho(G(V, E, w)) = \frac{2 \sum_{(u,v) \in E} W_{(u,v)}}{n(n-1)}$$

(13.17)

Las densidades intracluster ahora se forman como la relación entre la suma de los pesos de borde en un grupo y el número máximo posible de bordes en ese grupo y la densidad de intracluster del gráfico es el valor promedio de todos los grupos contenidos en el gráfico. Tanto en los gráficos no ponderados como en los ponderados, un buen agrupamiento debería dar como resultado valores promedio entre los valores de la empresa, que son significativamente más altos que las densidades del gráfico. El intercluster densidad en el caso de un gráfico ponderado es la relación de la suma de pesos de todos los bordes entre cada par de grupos para el número máximo posible de bordes entre clusters.

13.6 Notas del capítulo

El análisis de grandes gráficos en su conjunto es difícil debido a sus enormes tamaños. Como un enfoque para superar este problema en cierta medida, las propiedades locales alrededor de los vértices pueden evaluarse y las propiedades globales pueden luego aproximarse utilizando estas propiedades locales. Por ejemplo, el grado de un vértice y su coeficiente de agrupamiento son propiedades locales; La distribución en grados y el coeficiente de agrupamiento promedio son propiedades globales de un gráfico que dan una idea de su estructura general. Revisamos estos parámetros junto con el índice coincidente de dos vértices y la densidad de un gráfico en la primera parte de este capítulo.

Las redes del mundo real, como las redes tecnológicas, las redes biológicas y las redes sociales, se denominan redes complejas y los gráficos se utilizan comúnmente para representarlas. Luego describimos los principales modelos de estas redes, que son redes de mundo pequeño y sin escala junto con redes aleatorias. La distancia entre cualquiera de los dos vértices es pequeña en comparación con el tamaño de la red en las redes del mundo pequeño, que también tienen altos coeficientes de agrupamiento promedio de lo esperado. Las redes sin escala tienen pocos nodos de alto grado y el resto de los nodos tienen solo unos pocos grados con altos coeficientes de agrupamiento. Las redes complejas

suelen exhibir estas dos propiedades, pero en el caso de las redes biológicas, pueden ser necesarios otros modelos, como las redes jerárquicas.

La centralidad es un concepto clave en el análisis de gráficos grandes y, nuevamente, podemos estimar las propiedades globales de un gráfico a partir de las diversas formas de este parámetro local. La centralidad de grado de un vértice es simplemente su grado y la centralidad de proximidad muestra la importancia de un vértice en función de su distancia a todos los otros vértices en la gráfica. La centralidad de la interrelación de un vértice o un borde manifiesta los porcentajes de las rutas más cortas que se ejecutan a través de ellos y la centralidad del valor propio se basa en los espectros del gráfico. La separación entre bordes y las centralidades de valores propios se utilizan en la práctica más para analizar y también para descubrir agrupamientos en gráficos que otras medidas de centralidad.

La detección de subgrafos densos en gráficos grandes es necesaria ya que estas regiones pueden indicar una gran actividad allí. Revisamos los algoritmos de descubrimiento de camarillas y estructuras similares y describimos algunos métodos de agrupamiento en grandes gráficos.

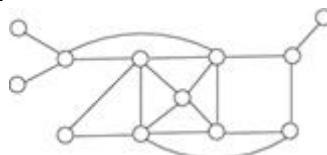


Fig. 13.12 La gráfica de muestra para el Ejercicio 1

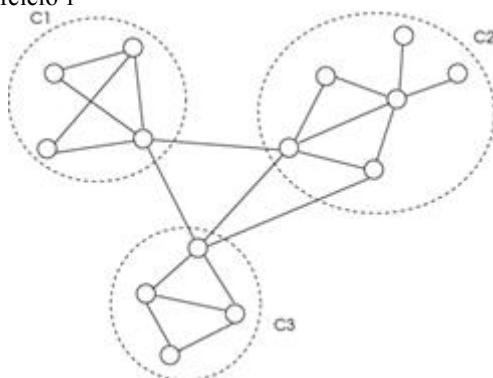


Fig. 13.13 La gráfica de muestra para el Ejercicio 2

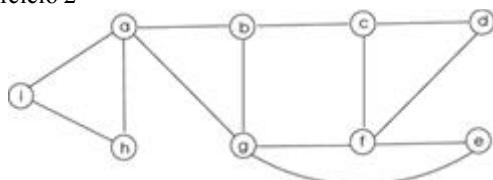


Fig. 13.14 La gráfica de muestra para el Ejercicio 3

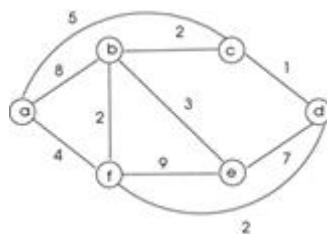


Fig. 13.15 La gráfica de muestra para el Ejercicio 6
Ceremonias

Grafique la distribución en grados de la gráfica representada en la Fig. [13.12](#).

- 1.
2. Calcula la densidad gráfica, intracluster densidades para tres grupos, y la intercluster densidad y la densidad de clúster promedio para el gráfico de la Fig. [13.13](#).
3. Encuentre el coeficiente de agrupamiento para cada vértice en el gráfico que se muestra en la Fig. [13.14](#) y también calcule el coeficiente de agrupamiento promedio.
4. Se calcularán los valores de centralidad de proximidad de los vértices de un gráfico. Modifique el algoritmo APSP de Dijkstra para encontrar los valores de centralidad de proximidad.
5. Modificar distribuido Bellman – Ford Algoritmo de Cap. [7](#) para encontrar centralidades de proximidad en una red informática.
6. Calcule los valores de intermediación del vértice para cada vértice en el gráfico de ejemplo que se muestra en la figura [13.15](#).

Referencias

1. Barabasi AL, Albert R (1999) Aparición de escalado en redes aleatorias. Ciencia 286: 509–512
[MathSciNet](#) [Crossref](#)
2. Batagelj V, Zaversnik M (2003) Un algoritmo O (m) para la descomposición de los núcleos de las redes. CoRR (repositorio de investigación en computación), [arXiv : 0310049](#)
3. Blaar H, Karnstedt M, Lange T, Winter R (2005) Posibilidades para resolver el problema de la camarilla mediante el paralelismo de hilos utilizando grupos de tareas. En: Actas del XIX simposio internacional de procesamiento distribuido y paralelo IEEE (IPDPS05) Taller 5 Volumen 06 en Alemania
4. Boccaletti S, Latorab V, Morenod Y, Chávez M, Hwang DU (2006) Redes complejas: estructura y dinámica. Phys Rep 424: 175–308
[MathSciNet](#) [Crossref](#)
5. Bron C, Kerbosch J (1973) Algoritmo 457: encontrar todas las camarillas de un gráfico no dirigido. Commun ACM 16 (9): 575–577
[Referencia cruzada](#)
6. Erciyes K (2015) Algoritmos distribuidos y secuenciales para bioinformática. Springer, Berlín (capítulos 10–11)
7. Erdos P, Renyi A (1959) En gráficos al azar. Publicationes Mathematicae 6: 290–297
[Matemáticas](#) [Matemáticas](#)
8. Erdos P, Renyi A (1960) Sobre la evolución de gráficos aleatorios. Publ Math Inst Hung Acad Sci 5: 17–61
[Matemáticas](#) [Matemáticas](#)
9. Garey MR, Johnson DS (1979) Computadoras e intratabilidad: una guía para la teoría de NP-completa. WH Freeman y compañía, Nueva York
[MATES](#)
10. Jaber K, Rashid NA, Abdullah R (2009) El algoritmo de cliques máximos paralelos para el agrupamiento de secuencias de proteínas. Am J Appl Sci 6: 13681372
13. Newman MEJ, Girvan M (2004) Búsqueda y evaluación de la estructura de la comunidad en redes. Phys Rev E 69: 026113
[Referencia cruzada](#)
14. Newman MEJ, Strogatz SH, Watts DJ (2002) Modelos de gráficos aleatorios de redes sociales. Proc Natl Acad Sci USA 99: 25662572

[Referencia cruzada](#)

15. Özgr A, Vu T, Erkan G, Radev DR (2008) Identificación de asociaciones de genes y enfermedades utilizando la centralidad en una red de interacción de genes minada en la literatura . Bioinformática 24 (13): 277–285
[Referencia cruzada](#)
- dieciséis. Perron O (1907) Mathematische Annalen . Zur Theorie der Matrices 64 (2): 248–263
17. Ravasz E, Somera AL, Mongru DA, Oltvai ZN, Barabsi AL (2002) Organización jerárquica de la modularidad en redes metabólicas. Ciencia 297: 15511555
[Referencia cruzada](#)
18. Schaeffer SE (2007) Agrupación de gráficos. Comput Sci Rev 1: 2764
[Referencia cruzada](#)
19. Schmidt MC, Samatova NF, Thomas K, Park BH (2009) Un algoritmo escalable y paralelo para la enumeración máxima de camarillas. J Parallel Distrib Comput 69: 417428
[Referencia cruzada](#)
21. Watts DJ, Strogatz SH (1998) Dinámicas colectivas de redes de mundo pequeño. Naturaleza 393: 440442
[Referencia cruzada](#)

14. Redes complejas

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

Resumen

Las redes complejas constan de decenas de miles de nodos y cientos de miles de bordes que conectan estos nodos. Los gráficos utilizados para modelar estas redes son grandes y, por lo general, se necesitan métodos especiales para el análisis de estas redes. Las principales redes complejas que son redes biológicas, redes sociales, redes tecnológicas y de información se revisan con una breve descripción de los algoritmos necesarios para resolver algunos problemas en estas redes en este capítulo.

14.1 Introducción

Las redes complejas constan de decenas de miles de nodos y cientos de miles de bordes que conectan estos nodos. Los gráficos utilizados para modelar estas redes son grandes y veremos que generalmente se necesitan métodos especiales para el análisis de estas redes. En teoría, podemos usar los algoritmos que hemos revisado en la Parte II para algunos de los problemas encontrados en redes complejas, pero incluso los algoritmos de tiempo lineal pueden requerir un tiempo de cálculo sustancial y, por lo tanto, los algoritmos heurísticos con complejidades más bajas son generalmente los preferidos. Sin embargo, muchos problemas en estas redes son difíciles de usar, lo que hace que el uso de las heurísticas sea la única opción.

Revisaremos las principales redes complejas de la vida real que son redes biológicas, redes sociales, redes tecnológicas y redes de información con una breve descripción de los algoritmos necesarios para resolver algunos problemas en estas redes en este capítulo. La agrupación o la detección de la comunidad es un problema en todas estas redes y describiremos algoritmos secuenciales, paralelos y distribuidos fundamentales para este propósito, con énfasis en la agrupación en paralelo en redes biológicas y la agrupación distribuida en redes informáticas. En la Web se necesitan algoritmos de búsqueda eficientes y los revisaremos en la última parte de este capítulo.

14.2 Redes biológicas

Las redes biológicas son las redes de organismos con nodos que representan entidades biológicas y bordes que muestran las interacciones entre ellos. La célula es la unidad biológica básica de todos los organismos. Las redes biológicas se pueden clasificar como redes dentro de la célula y redes fuera de la célula a un nivel más macroscópico. Las células

son principalmente de dos tipos: células eucariotas que tienen núcleos que llevan la información genética en los cromosomas y células procarióticas que no tienen núcleos. Un gen es la unidad básica de la herencia y consiste en una cadena de nucleótidos que son pequeñas moléculas que producen ácido desoxirribonucleico (ADN) en una estructura de doble hélice.

Los genes se decodifican para producir aminoácidos que se encadenan para producir proteínas que son moléculas grandes fuera del núcleo de la célula que llevan a cabo todas las funciones vitales relacionadas con la vida. Este proceso se llama el dogma central de la vida . Las proteínas son moléculas grandes y su funcionalidad principal depende de sus secuencias de aminoácidos, su forma tridimensional y también la interacción con otras proteínas. Estas interacciones se pueden representar mediante bordes de gráficos para obtener redes de interacción proteína-proteína (PPI) con proteínas como los nodos de la red. La figura 14.1 muestra la red PPI de *T. pallidum* . Otras redes principales en la célula son las redes de regulación de genes.(GRN) formados por proteínas y genes interactivos, y las redes metabólicas representan reacciones bioquímicas en la célula para generar metabolismo [8]. Otras redes biológicas fuera de la célula incluyen redes cerebrales, redes neuronales, redes filogenéticas y la red alimentaria [8]. Los principales problemas encontrados en las redes biológicas son la agrupación, la búsqueda de motivos de red y la alineación de la red.

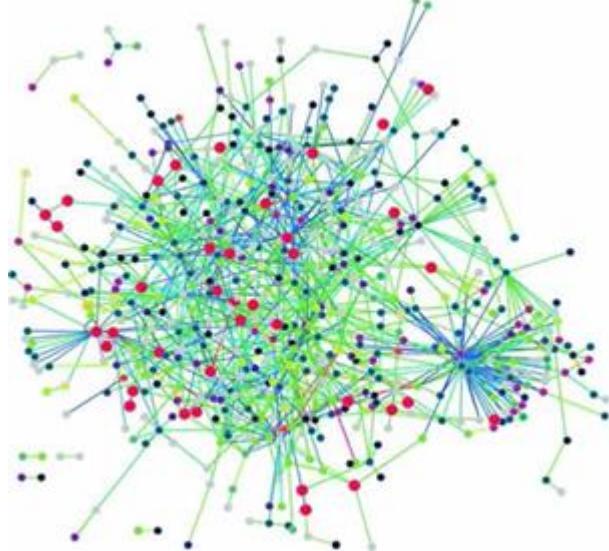


Fig. 14.1 La red PPI de *T. pallidum* tomada de [22]

14.2.1 Agrupación

Es necesario encontrar agrupamientos en redes biológicas, ya que estas regiones densas de actividad biológica tienen cierta importancia, ya que a veces muestran estados de enfermedad de un organismo. Agrupación de los algoritmos en redes biológicas se pueden clasificar ampliamente como agrupación jerárquica , basada en la agrupación densidad , basado en la agrupación de flujo y agrupamiento espectral [8]. Ya hemos revisado la agrupación basada en densidad como la búsqueda de estructuras tipo camarilla en el capítulo. 10 y revisaremos los métodos de muestra en las siguientes secciones.

14.2.1.1 Agrupación basada en MST

El agrupamiento jerárquico es un método clásico, simple y popular de agrupamiento. un ejemplo de este método es el algoritmo de agrupamiento basado en MST que se basa en la

idea de que dos nodos que están más alejados en el MST pueden asignarse a dos clústeres diferentes. Dado que MST es acíclico, la eliminación de un borde de MST lo divide en dos grupos. Este proceso puede repetirse hasta que se cumpla con un cierto criterio de calidad de cluster. El algoritmo se implementa siguiendo los siguientes pasos:

calcular el MST T de la gráfica $G = (V, E)$

- 1.
2. repetir
3. $e_{uv} \leftarrow \max\{v(u, v) \in T, w_{uv}\}$,
4. $G \leftarrow G - \{e_{uv}\}$
5. etiquetar vértices de nuevos clusters
6. Hasta que se cumpla un criterio de calidad.

Tenga en cuenta que MST se calcula solo una vez y los bordes más pesados se eliminan de forma iterativa. El etiquetado de los clústeres recién formados se puede hacer simplemente mediante el algoritmo BFS en tiempos lineales para los dos clústeres. Podemos eliminar una cantidad de bordes en cada paso del MST que están a más de una distancia de umbral. El cálculo de MST se puede realizar en paralelo utilizando cualquiera de los algoritmos descritos en el Cap. 7. El agrupamiento a través de MST en paralelo (CLUMP) es un método propuesto de agrupamiento para datos biológicos [20] en el que los subgrafos bipartitos se construyen en paralelo.

14.2.1.2 Agrupación espectral

Vimos la representación de la matriz de adyacencia de un gráfico G y su matriz laplaciana no normalizada es $L = D - A$ donde A es la matriz de adyacencia y D es la matriz de grados con grados de vértices en la diagonal. El laplaciano normalizado es $L = I - D^{1/2}AD^{-1/2}$. L en ambas formas es real y simétrica y, por lo tanto, sus valores propios son reales. El segundo valor propio más pequeño de L contiene información sobre su conectividad, como vimos en el Capítulo. 12. También podemos usar este valor llamado el valor de Fiedler y su vector propio llamado vector de Fiedler [9] para dividir la gráfica G en dos grupos

C_1 y C_2 como abajo [12]:

$G = (V, E)$ con $V = \{v_1, \dots, v_n\}$

- 1.
2. $C_1 \leftarrow \emptyset, C_2 \leftarrow \emptyset$
3. $L \leftarrow D - A$
4. calcular Fiedler vector F de L
5. **para todo** $F[i]$ de F **entonces**
6. Si $F[i] \leq \tau$

7. $C_1 \leftarrow C_1 \cup \{v_i\}$
8. más $C_2 \leftarrow C_2 \cup \{v_i\}$
9. **terminara si**
10. **final para**

El valor umbral τ comúnmente se toma como 0 y una implementación recursiva de este algoritmo producirá k clusters. Otros métodos de agrupación en redes biológicas incluyen un enfoque basado en el flujo llamado algoritmo de agrupación de Markov que utiliza la idea de que las caminatas aleatorias en un gráfico terminarán en una agrupación [6]. Hay varios otros algoritmos para agrupar en redes biológicas, la mayoría de los cuales utilizan heurística.

14.2.2 Búsqueda de motivos de red

Un motivo de red es un subgrafo conectado recurrente en una red biológica. Estos se consideran como bloques de construcción de tales redes y comúnmente se asume que tienen alguna función básica asociada a ellos. El descubrimiento de motivos ayuda a comprender la estructura de las redes biológicas y también se pueden comparar varias redes que representan organismos para buscar si tienen antepasados comunes. La figura 14.2 muestra algunos ejemplos de motivos.

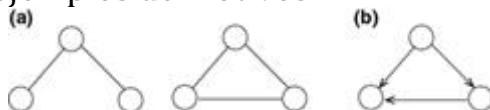


Fig. 14.2 Ejemplos de motivos de red de orden 3

El descubrimiento de motivos es una de las áreas de investigación fundamentales en las redes biológicas. Es un proceso complicado en el que se utilizan comúnmente las heurísticas y consta de los siguientes pasos:

- Búsqueda de motivos : se buscan todas las instancias de un motivo o se utiliza el muestreo donde solo se busca un subgrafo de muestra y los resultados se proyectan a todo el gráfico. Para gráficos muy grandes, este último es comúnmente empleado.
- 1.
 2. Clases isomorfas: los subgrafos que parecen diferentes pueden ser isomorfos, por lo tanto, deben agruparse para un procesamiento correcto.
 3. Significación estadística : necesitamos determinar la significación estadística de los subgrafos descubiertos. Este proceso generalmente se realiza generando un conjunto de gráficos aleatorios de estructura similar al gráfico objetivo y realizando la búsqueda en estos gráficos. Los resultados obtenidos en dos casos se pueden comparar para determinar si los subgrafos encontrados son motivos reales.

14.2.3 Alineación de red

A menudo se requiere comparar dos redes biológicas en su totalidad o en parte para determinar su relación, lo que puede ayudar a comprender mejor el proceso evolutivo. Podemos comparar dos redes biológicas en alineación por pares o varias redes biológicas en alineación múltiple para buscar sus similitudes. En la alineación global , se comparan redes completas para especies similares, mientras que las subredes similares se buscan en alineación local para diversas especies.

Para evaluar la similitud de dos redes, podemos evaluar la similitud de nodos o la similitud topológica de las redes, sin embargo, en muchos casos, ambas métricas se utilizan con diferentes ponderaciones. El primero puede reflejar la estructura interna de los nodos, como las secuencias de aminoácidos de las proteínas en las redes PPI. Supongamos que tenemos que hacer gráficas, G_1 y G_2 . Representando dos redes biológicas. Se puede formar una matriz de similitud R que tiene entrada, r_{ij} , con un peso que representa la similitud entre el nodo i de G_1 y el nodo j de G_2 . Sobre la base de las entradas de R, se puede construir un gráfico bipartito ponderado completo y el problema de la alineación de la red se reduce al problema máximo de emparejamiento bipartito que hemos estudiado en el Cap. 9.

14.3 Redes sociales

Las redes sociales consisten en individuos o grupos de personas con relaciones entre ellos. Una gráfica que representa una red social tiene sus nodos como personas o grupos y los bordes representan las relaciones. Las redes sociales tienen propiedades de pequeño mundo y de escala libre como otras redes complejas.

14.3.1 Relaciones y equivalencia

Algunos términos relacionados con las relaciones en una red social son los siguientes.

- Cierre triádico : Supongamos que A y B son dos amigos en una red social. Hay una buena posibilidad de que un amigo de A (o B) que no sabe B (o A) se haga amigo de esa persona en el futuro. Esta propiedad se denomina cierre triádico, ya que en el futuro se formará un triángulo entre las tres personas por la composición de un borde entre las dos personas que no se han encontrado antes.
- Homophily : El homophily propiedad observada en las redes sociales dinámicos es que los individuos o grupos tienen tendencia a organizar las relaciones con otras personas o grupos como ellos. La similitud podría ser la edad o la filosofía o algo más en común.
- Relaciones : en una red social de amistad, podemos etiquetar los bordes entre dos personas como positivos (+) lo que significa que se gustan o negativos (-) que muestran que no se gustan, asumiendo que estas relaciones son simétricas. En una pequeña red social de estructura triangular con tres personas A, B y C ; Podemos tener cuatro casos:

A , B y C son amigos mutuos.

- 1.
2. A , B y C, todos se disgustan.
3. Uno de ellos es amigo de otros dos, pero estos dos no se gustan.
4. Dos de los tres son amigos entre sí, pero ninguno de ellos es un amigo con el tercero.

El primer caso y el último caso son relaciones equilibradas, mientras que el segundo y el tercero no lo son. Con tres personas que se disgustan, hay una tendencia a que dos se conviertan en amigos contra el tercero y, por lo tanto, este caso es desequilibrado. Además, el tercer caso implica a dos personas que no quieren estar juntas, sino que quieren estar

juntas con una tercera persona, causando nuevamente una situación desequilibrada. El último caso es equilibrado ya que no hay conflicto. En términos teóricos de la gráfica, esto significa que cualquier triángulo con una o tres relaciones positivas está equilibrado y los triángulos con cero o dos bordes positivos están desequilibrados. Ahora podemos encontrar todos los triángulos en una red social dada con relaciones asignadas y si todos estos son equilibrados, toda la red social es un equilibrio, de lo contrario, incluso si existe un triángulo desequilibrado, se dice que la red está desequilibrada.

- Equilibrio estructural : un método general para determinar si una red social está equilibrada o no fue propuesta por Harary [13] en el teorema del equilibrio.

Teorema 14.1 Una red social completa se equilibra cuando todos los pares de sus nodos tienen relaciones positivas entre sí o cuando sus nodos se pueden dividir en dos conjuntos V_1 y V_2 de tal manera que todos los nodos dentro de estos grupos son amigos con todos los otros nodos en sus grupos y cada nodo de un grupo tiene relaciones negativas con todos los otros nodos en el otro grupo.

Una red social equilibrada según este teorema se representa en la figura 14.3 . Se puede ver que cualquier triángulo que tiene dos nodos en un grupo y uno en el otro tiene solo uno + etiqueta que significa que estos triángulos están equilibrados. Los triángulos restantes están todos incrustados en cada grupo y tienen + Las etiquetas en sus bordes, por lo tanto, también se equilibran dando como resultado una red equilibrada.

- Equivalencia : el concepto de equivalencia en una red social se relaciona con las posiciones o roles de las personas o una persona en una red social. Si se cambian las posiciones de dos personas equivalentes en una red social, el funcionamiento de la red no debería cambiar.

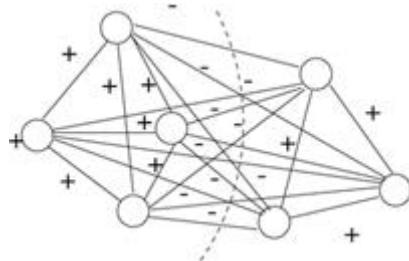


Fig. 14.3 Una gráfica completa que muestra una red social balanceada separada en dos conjuntos de nodos por la línea discontinua. Los tres triángulos en los dos conjuntos tienen todos + Las etiquetas en sus bordes y en todos los demás bordes intergrupales tienen señales que indican que esta red está equilibrada de acuerdo con el teorema del equilibrio de la estructura.

14.3.2 Detección de la comunidad

La detección de comunidades que son regiones densas de actividad en las redes sociales tiene muchas implicaciones; por ejemplo, podemos analizar estos grupos de personas o grupos para comprender su comportamiento. Revisaremos dos algoritmos para este propósito que se implementan en las redes sociales.

Edge betweenness -Based Algoritmo

El EDGE- intermediación valor de un borde e en un gráfico de G era la fracción de todo, pares-más corta caminos que pasan a través de e . Intuitivamente, los bordes con valores altos tienen una mayor probabilidad de unir regiones densas del gráfico que los bordes con valores más bajos. En el caso extremo, la eliminación de un puente que desconecta un

gráfico tiene un valor muy alto entre el borde . Con base en esta observación, Girvan et al. propuso un algoritmo para detectar agrupamientos en una red compleja representada por un gráfico $G = (V, E)$ que consta de los siguientes pasos con una estructura similar a la agrupación basada en MST [11]:

repetir

- 1.
2. calcular EDGE- intermediación valor σ_{xy} para cada borde (x , y) de la gráfica G .
3. $e_{uv} \leftarrow$ el borde con maximo σ valor
4. $G \leftarrow G - \{e_{uv}\}$
5. Hasta que se cumpla un criterio de calidad.

El paso más lento para este algoritmo es el cálculo de los valores de límite entre bordes y, por lo tanto, tiene un bajo rendimiento para los gráficos que tienen más de unos pocos miles de nodos. Este método también se utiliza para detectar agrupamientos en redes biológicas.

14.3.2.1 Algoritmo basado en la modularidad

La calidad de las agrupaciones formadas por un algoritmo de agrupación puede evaluarse mediante el concepto de modularidad . Supongamos que una gráfica G se divide en k grupos y e_{ij} es la fracción de bordes en el grupo i y a_i^2 es la fracción de bordes con al menos un extremo en el grupo i . La modularidad Q de este gráfico se puede determinar de la siguiente manera [19]:

$$Q = \sum_{i=1}^k (e_{ii} - a_i^2).$$

(14.1)

Usando esta ecuación, de hecho estamos evaluando la diferencia en las probabilidades de que un borde esté en el módulo i y que un borde aleatorio caiga en el módulo i y sume estos valores para todos los grupos. Cuando el porcentaje de bordes dentro de los grupos es mucho mayor que los que tienen un extremo en un grupo (bordes inter-grupo), esperamos un alto valor de Q , de hecho, el valor de Q se acerca a la unidad cuando solo hay algunos bordes entre grupos . Un algoritmo de agrupamiento basado en el concepto de modularidad se puede formar de tal manera que se combinen dos agrupamientos para aumentar la modularidad de la siguiente manera [19].

Cada nodo de la gráfica se considera inicialmente un cluster.

- 1.
2. repetir

3. Combine los dos grupos que aumentarán la modularidad más
4. Hasta la fusión de los clusters se incrementa la modularidad.

Este algoritmo proporciona clusters como un dendograma que puede usarse para obtener el número requerido de clusters. La complejidad del tiempo de este algoritmo es $O((m + n)n)$ o $O(n^2)$ en escasos gráficos [[19](#)].

14.4 Redes Inalámbricas Ad Hoc

Las redes de computadoras consisten en nodos computacionales tales como computadoras personales, servidores, enrutadores y teléfonos conectados por una red de interconexión. El medio de comunicación entre los nodos puede ser un cable coaxial, un enlace de fibra, un medio inalámbrico o, más frecuentemente, una combinación de todos estos. Los datos que se transfieren entre los nodos de una red de computadoras se dividen comúnmente en paquetes . Estas unidades de datos se entregan al destino mediante una red de conmutación de paquetes.donde cada paquete se enruta comúnmente de forma independiente al destino y luego todos los datos que consisten en un número de paquetes se vuelven a ensamblar en el destino antes de ser entregados al usuario o la aplicación. De esta manera, muchos usuarios / aplicaciones pueden compartir el mismo medio de comunicación. En contraste, una parte del medio de comunicación está dedicada a la aplicación durante la transferencia de datos en la conmutación de circuitos . Los paquetes de red pueden llevar información de control o información de datos. Los paquetes de control son utilizados por los protocolos de comunicación para realizar tareas como el enrutamiento y la sincronización de las partes comunicantes.

El medio cableado puede ser un par trenzado, un cable coaxial o enlaces de fibra óptica en el orden de rendimiento y costo de bajo a alto. Los enlaces de comunicación inalámbricos emplean comúnmente tecnologías de comunicación por microondas, satélite y también ópticas para transferir datos. Los nodos de red constan de repetidores, puentes, conmutadores, enrutadores y MODEM. Debido a que una señal eléctrica se debilita y se distorsiona a través de la distancia, se necesita un repetidor para eliminar el ruido de una señal eléctrica y transferirlo habilitándolo a distancias regulares. Estos dispositivos operan en la capa física del modelo ISO OSI de 7 capas [[14](#)]. Los puentes y conmutadores funcionan en la capa de enlace de datos del modelo OSI para filtrar el tráfico y reenviar paquetes. Los enrutadores que trabajan en la capa de red son los dispositivos clave para determinar la ruta a la que se transferirá el paquete.

Se forma una red ad hoc sin ninguna administración central con nodos que actúen como hosts y enrutadores al mismo tiempo. Alámbrica o inalámbrica, una red de computadoras puede modelarse convenientemente como un gráfico con los vértices del gráfico que representan los nodos y los bordes que muestran los enlaces de comunicación entre los nodos. En las redes inalámbricas, un enlace entre dos nodos sale si están dentro del rango de comunicación entre sí. Tenga en cuenta que es posible que necesitemos un dígrafo para representar dicha red si las capacidades de comunicación de dos nodos no son las mismas. Dos tipos de redes inalámbricas ad hoc ampliamente utilizadas son las redes móviles ad hoc (MANET) donde los nodos se mueven dinámicamente y las redes de sensores inalámbricos (WSN) con nodos computacionales de detección pequeña. Una WSN también puede ser móvil, pero en su mayoría, estas redes tienen topologías inalámbricas fijas.

Una red inalámbrica ad hoc no necesita una infraestructura de red fija que utilice dispositivos como enrutadores y puntos de acceso. Dicha estructura tiene muchas ventajas, ya que no tenemos que instalar y mantener equipos de infraestructura costosos como enrutadores, puntos de acceso y el medio de comunicación por cable. Sin embargo, nos enfrentamos a nuevos problemas, como la gestión de la movilidad en un MANET y la necesidad de algoritmos de eficiencia energética en una WSN, ya que la carga de la batería es escasa. Una red vehicular es un ejemplo de MANET donde los vehículos se comunican y coordinan en la carretera para prevenir accidentes e intercambiar información. En la Fig. 14.4 se muestra un ejemplo de MANET .

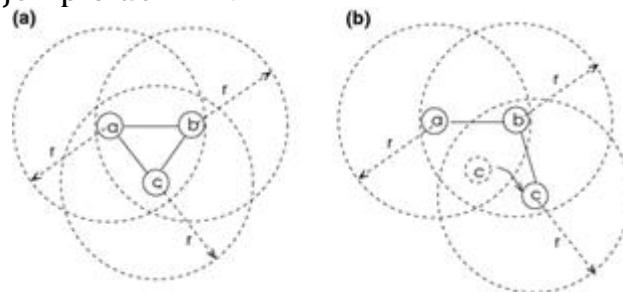


Fig. 14.4 A MANET con tres nodos de un , b y c . Todos están dentro de los rangos r entre sí en a y el nodo c se mueve a una nueva posición que está fuera del rango del nodo a en b ; haciendo que el borde (a , c) se elimine del gráfico y que se modifique el borde (b , c)

Una WSN consiste en pequeños nodos sensores que detectan un fenómeno físico como el calor, la vibración o el movimiento y se comunican entre sí para transmitir esta información a un nodo especial con capacidades más altas, llamado sumidero . Las WSN se emplean comúnmente para monitoreo ambiental, domótica y cibersalud, donde los estados de salud de las personas se pueden monitorear de forma remota. La figura 14.5 muestra una WSN con un nodo receptor. Los problemas teóricos de grafos fundamentales que enfrentan los MANET y las WSN son la conectividad, la construcción de la red troncal, el agrupamiento y el enrutamiento, tal como se describe en la siguiente sección.

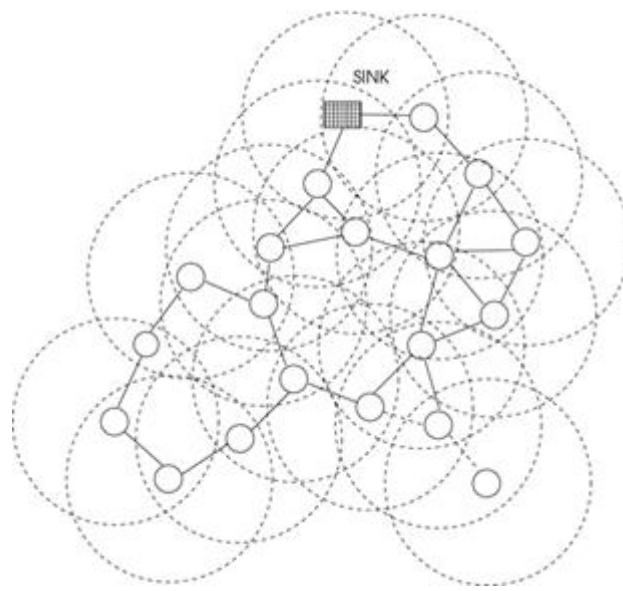


Fig. 14.5 Un WSN con un nodo sumidero. Se muestran los rangos de transmisión de todos los nodos.

Comenzamos este capítulo con las soluciones teóricas gráficas para los problemas encontrados en las redes inalámbricas ad hoc que son: monitoreo de conectividad, agrupamiento, formación de red troncal y enrutamiento.

14.4.1 k - Conectividad

Conectividad en una gráfica $G = (V, E)$ significa que hay una ruta entre cualquiera de los dos nodos u y v en G como revisamos en el Capítulo. 8 . La conectividad es necesaria en cualquier red de computadoras para la transferencia de información entre cada par de nodos, pero este problema es más importante en una red inalámbrica. Por ejemplo, mover los nodos en un MANET puede interrumpir la conectividad fácilmente y un nodo sensor que se queda sin energía de la batería puede causar la desconexión en dicha red. Las probabilidades de estos eventos son mucho mayores que la falla de un enrutador en una red cableada.

Recordemos el problema de la conectividad k : una red está conectada a la k si hay al menos k rutas disjuntas entre cualquiera de sus dos nodos. Podemos deducir que la falla de un mínimo de nodos $k-1$ da como resultado una red desconectada y, por lo tanto, no funcional en una red conectada a k . Claramente, cuanto mayor sea el valor de k , más fuertemente conectada estará la red. Por lo tanto, podemos decir que una red con un valor de conectividad k más alto es más confiable y tolerante a fallas contra fallas de nodo que una red con un valor k más bajo. En general, tenemos tres problemas principales relacionados con k -Conectividad en una red inalámbrica ad hoc [1]:

Coloca los nodos de un WSN o un MANET en cualquier momento para que se logre la conectividad k .

- 1.
2. Detección del valor de k en un MANET dado o una red WSN.
3. Restauración del valor de k .

En la búsqueda de una solución al primer problema, los vecinos de un nodo o la potencia de radio de los nodos se incrementan de forma iterativa en diversos estudios de investigación. Incluso cuando se lleva a cabo este primer paso, debemos monitorear y determinar el valor de k para tomar medidas correctivas cuando este caiga por debajo de un valor deseado. Cuando esto sucede, podemos colocar nuevos nodos en una WSN o mover los nodos móviles a nuevas ubicaciones en MANET para aumentar el valor de k . Ya hemos revisado los algoritmos para determinar el valor de k en el capítulo. 8 .

14.4.2 La agrupación en Hoc Inalámbricas Ad Redes

La agrupación en una red inalámbrica ad hoc se realiza agrupando los nodos que están dentro de sus rangos de transmisión de radio. Un nodo específico en un clúster se suele asignar como cabeza de grupo (CH) y este nodo funciona como el administrador del clúster. La agrupación con CHs es ventajosa por varias razones. En primer lugar, se obtiene una estructura jerárquica en la que se pueden asignar varias tareas a los CH que pueden trabajar en paralelo para implementar las tareas requeridas en sus grupos, como las funciones de la capa MAC, que incluyen el acceso al canal, las mediciones de potencia y mantener la sincronización de cuadros por división de tiempo. Los CHs también se pueden formar conectados directa o indirectamente entre sí y se pueden usar para construir una red troncal para transferencias de mensajes. Este tipo de enrutamiento de un mensaje

a través de la red troncal hasta que se alcanza el CH más cercano al nodo de destino en la red troncal elimina el paso excesivo de mensajes como se muestra en la Fig. 14.6. No hay necesidad de almacenar y administrar datos globales, ya que cada CH conoce los identificadores de los nodos en su grupo y cada vez que se recibe un mensaje en la red troncal, se realiza una comprobación con el campo de destino en el mensaje y los identificadores de nodo del grupo. Si hay una coincidencia, el mensaje se transmite al nodo en el clúster, de lo contrario, el mensaje se transfiere al nodo principal vecino.

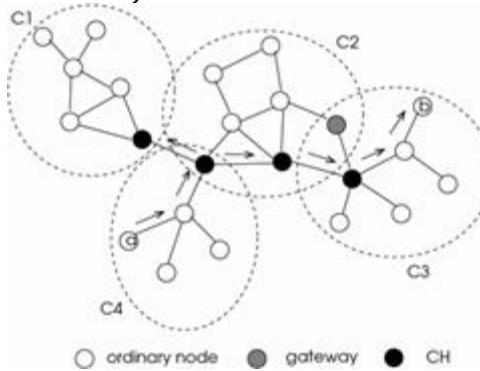


Fig. 14.6 Un ejemplo de red troncal en una red inalámbrica ad hoc. Agrupaciones C_1 , C_2 , C_3 y C_4 se muestran dentro de círculos discontinuos y los CH se muestran en negrita. Nodo a en C_4 envía un mensaje a su CH que reenvía el mensaje a lo largo de la red troncal hasta que el grupo de b que es C_3 se encuentra

Los nodos de un clúster k se pueden alcanzar mutuamente por un camino de longitud como máximo k. Un k- cluster con $k=1$ es una camarilla. Un clúster k -hop consta de nodos que están a una distancia máxima de k -hop de su CH. Los nodos en una red ad hoc agrupada pueden clasificarse como CHs, nodos de puerta de enlace que conectan dos clústeres y nodos ordinarios como se muestra en la Fig. 14.6. El agrupamiento óptimo y la selección del número óptimo de CH son a la vez NP-hard y se utilizan comúnmente varias heurísticas para realizar estas funciones. Una vez que se forma un grupo, se puede seleccionar un CH por ruptura de simetría entre los nodos utilizando sus identificadores, sus grados, movilidad de nodo, energía de la batería del nodo o varias combinaciones de estos parámetros. Después de la construcción de los grupos y la selección de los CHs, los grupos deben mantenerse. El mantenimiento de la estructura del clúster en un MANET es difícil debido a la dinámica de los nodos y es necesario que los nodos asignen nuevos CH cuando el CH se aleja. Un nodo móvil, por otro lado, puede unirse a un nuevo clúster al que se acerca.

La energía consumida por un CH es más grande que un nodo ordinario debido a su función de retransmisión de mensajes en la red. Por esta razón, la función CH puede rotarse entre los nodos ordinarios para proporcionar equilibrio de carga. La movilidad es la principal preocupación en un MANET y la potencia limitada de las baterías se debe considerar cuando se agrupan en WSN. Revisaremos un algoritmo simple para formar clusters y CHs simultáneamente en una red inalámbrica ad hoc y revisaremos la construcción de una red troncal usando enfoques teóricos de grafos.

14.4.2.1 Algoritmos

Gerla y Tsai propusieron un algoritmo de agrupamiento usando los identificadores de nodos en redes inalámbricas ad hoc [10]. Un nodo en dicha red difunde periódicamente los identificadores de los vecinos en su rango de transmisión en su gráfico de unidad de

disco. Después de la transmisión, escucha el medio por un tiempo y realiza una de las siguientes acciones:

- Un nodo que no escucha un nodo con un identificador más alto que él mismo después de un tiempo de espera decide ser un CH y transmite esta condición.
- El vecino identificador más bajo que escucha un nodo se asigna como su CH, a menos que ese nodo renuncie voluntariamente a su posición como un CH.
- Un nodo que escucha la declaración de dos o más CHs se asigna a sí mismo como una puerta de enlace que une dos clústeres.

Como puede verse, la condición de ruptura de la simetría es la elección del nodo con el identificador más bajo en el rango de transmisión y, por lo tanto, el nombre del algoritmo. El nodo que escucha todos los nodos de identificador superiores se convierte en el CH y se transmite a sí mismo como el CH. Los nodos que escuchan el mensaje de declaración de CH se convierten en parte del clúster administrado por ese CH. Una corrección final en el algoritmo implica seleccionar un nodo como puerta de enlace cuando escucha dos nodos como CHs. Este algoritmo simple crea agrupaciones en tiempo lineal, sin embargo, un nodo de identificador bajo que se une a una agrupación da como resultado la reorganización de una agrupación que puede ser costosa.

Los autores propusieron otro algoritmo para formar agrupaciones llamado algoritmo de agrupación de conectividad más alto que apunta a seleccionar el nodo con el grado más alto como CH [10]. Las siguientes reglas se aplican en este algoritmo después de que un nodo difunde la lista de nodos que puede escuchar, incluso a sí mismo:

- Un nodo que ha elegido un CH está cubierto , de lo contrario, está descubierto .
- El nodo con la conectividad (grado) más alta entre los vecinos descubiertos se elige como CH. Los identificadores se utilizan para romper los lazos.
- Un nodo que elige un CH deja de ser un CH.

Esta vez, se seleccionan los nodos con grados más altos asumiendo que pueden acceder a los nodos en sus agrupaciones fácilmente. En ambos algoritmos, no hay dos CHs adyacentes entre sí y en un clúster, la distancia entre dos nodos cualquiera es, como máximo, dos saltos. La figura 14.7 muestra agrupamientos formados por ambos algoritmos. El número total de mensajes comunicados en el primer algoritmo es $2 n$, ya que cada nodo transmitirá un mensaje (actualización) a sus vecinos y otro mensaje (*i_am_chhead* u ordinario) para informar si es un CH o un nodo ordinario.

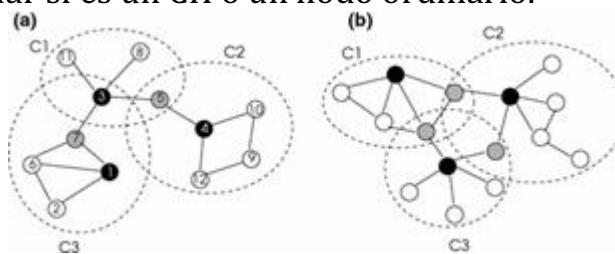


Fig. 14.7 a Algoritmo de identificador más bajo, b Implementaciones del algoritmo de conectividad más alto. Los CH se muestran en negrita y los nodos de la puerta de enlace están en gris

14.4.3 Construcción de la espina dorsal con conjuntos dominantes

En lugar de agrupar la red inalámbrica ad hoc en grupos con CHs y luego formar una red troncal con estos CHs, podemos usar conjuntos dominantes conectados como la red troncal. Un conjunto dominante D de una gráfica. $G = (V, E)$ Es un subconjunto de sus vértices tal que $\forall v \in V$, ya sea $v \in D$ o v es adyacente a un vértice $u \in D$. En un conjunto dominante

conectado (CDS), hay una ruta entre dos vértices en este conjunto. Recuerde que encontrar el orden mínimo conectado o desconectado del conjunto dominante es NP-duro (vea el Capítulo 3 y necesitamos formar un CDS para que el backbone funcione correctamente. De lo contrario, necesitamos insertar vértices entre los elementos del conjunto dominante. también forme agrupaciones de esta manera al denotar cada elemento de D como CH y cualquier vecino conectado a tal CH se convierte en el miembro del agrupamiento de esta CH. Por ejemplo, los CHs en la Fig. 14.6 forma un conjunto dominante de 3 saltos con grupos mostrados alrededor de ellos. Podemos construir un conjunto independiente máximo (MIS) del gráfico y luego conectar los nodos en el MIS para obtener un CDS. Describiremos un algoritmo directo que encuentra el CDS en tiempo lineal en la siguiente sección. Un requisito evidente es que los nodos de red troncal deben estar conectados y que cada nodo debe estar en la red troncal o un vecino a un nodo de red troncal.

14.4.3.1 Algoritmo basado en la poda

El algoritmo propuesto por Wu y Lin [23] encuentra el MCDS de una red con nodos que tienen identificadores únicos que utilizan la información vecina de los nodos y, por lo tanto, es un algoritmo distribuido local que consta de dos pasos. Los identificadores de vecinos de todos los nodos se intercambian en el primer paso y cualquier nodo que encuentre que tiene al menos dos vecinos desconectados, se denomina a sí mismo en MCDS cambiando su color a negro . Cada nodo luego envía su estado a todos sus vecinos en el segundo paso después del cual se aplican las siguientes reglas de eliminación para eliminar los nodos redundantes del MCDS:

- Si $\exists u \in N[v] (color_u = black) \wedge (N[v] \subseteq N[u]) \wedge (id_v < id_u)$ entonces $color_v \leftarrow white$
- Si $\exists u, w \in N(v) (color_u = color_w = black) \wedge (N(v) \subseteq (N(u) \cup N(w))) \wedge (id_v = \min(id_u, id_w, id_v))$ entonces $color \leftarrow white$

La primera regla significa que un nodo que encuentra todo su conjunto de vecinos está cubierto por un vecino con un identificador inferior que se elimina del CDS. Tenemos dos nodos que cubren los mismos vecinos en este caso y uno de ellos puede eliminarse y los identificadores se utilizan para romper la simetría. La segunda regla elimina un nodo de CDS si sus vecinos están cubiertos por la unión de los vecinos de dos nodos de identidad superiores en el CDS. En este caso, estamos considerando la unión de nodos que pueden estar parcialmente cubiertos por dos nodos vecinos. Como cada nodo envía exactamente dos mensajes de difusión en una red inalámbrica ad hoc en esta implementación, el número total de mensajes transmitidos es 2 y se puede utilizar convenientemente en un MANET debido a sus bajos requisitos de mantenimiento. Cuando un nodo se aleja, solo sus vecinos necesitan actualizar sus estados. La Figura 14.8 muestra un ejemplo de red en la que se forma un CDS mínimo en dos pasos.

Cokuslu y Erciyes modificaron este algoritmo incorporando los grados de los nodos, así como sus identificadores, al tiempo que recortan en el segundo paso [4]. Compararon su algoritmo con el algoritmo de Wu y demostraron experimentalmente que proporciona MCDS significativamente más pequeños. Das et al. [5] proporcionó dos algoritmos que son las versiones distribuidas de los algoritmos Guha - Khuller que hemos visto en el capítulo. 10 . En el primer algoritmo, a los nodos se les asignan pesos como sus grados efectivos, que es el número de sus vecinos que no son CDS. Inicialmente un pequeño conjunto dominador C Se forma que puede tener varios componentes desconectados. El bosque formado por los bordes. $\{v_1, v_2\}$ donde $v_1 \in C$ y $v_2 \in N(v_1)$ luego se conecta en la segunda etapa utilizando un algoritmo de árbol de expansión mínimo distribuido (MST). El CDS

consiste en los nodos no hoja del MST formado. Este algoritmo proporciona una relación de aproximación de $2Hd + 1$ en tiempo usando $O(n|C|d + m + n \log n)$ mensajes [5].

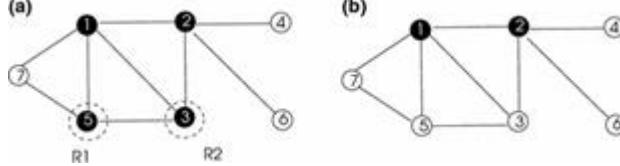


Fig. 14.8 Implementación del algoritmo basado en la poda en un pequeño gráfico. Los nodos que se marcan como negros ya que tienen dos vecinos directamente desconectados se muestran en a, y la implementación de la Regla 1 y la Regla 2 en los nodos 5 y 3, respectivamente, da como resultado el CDS más pequeño que se muestra en b

14.4.3.2 Algoritmos distribuidos codiciosos

En el segundo algoritmo, se investigan las rutas de uno o dos pasos que emanan del CDS actual para encontrar el nodo con el mayor número de nodos blancos en cada ronda. Un nodo o un par de nodos con el mayor número de tramas se agrega a los CDS existentes como en [5]. Este algoritmo logra una relación aproximada de $2Hd$ en tiempo usando $O(n|C|)$ mensajes [5].

14.4.3.3 Construcción de CDS distribuidos basados en MIS

Alzoubi et al. propuso un algoritmo de construcción de CDS basado en UDG que consta de tres fases de la siguiente manera [2]:

Elección del líder : un árbol de expansión se construye enraizado en el líder y los nodos notifican al líder que esta fase ha terminado.

- 1.
2. Cálculo de nivel : el líder comienza esta fase enviando un mensaje de nivel 0 y luego cada nodo aumenta el nivel recibido del padre y transfiere el mensaje de nivel a los niños, si existen. Una operación de convergecast por mensajes completos a la raíz concluye esta fase.
3. Marca de color : los nodos en el MIS son de color negro y todos los demás nodos son de color gris al final de esta fase. El mensaje del dominadores enviado por un nodo que se marca a sí mismo en negro y el dominado es enviado por un nodo que se marca a sí mismo en gris .

Inicialmente, todos los nodos son blancos y el algoritmo se ejecuta de acuerdo con las siguientes reglas [2]:

Un nodo blanco que recibe un mensaje de dominador por primera vez se marca en gris y emite un mensaje de dominio para informar que se ha dominado.

- 1.
2. Un nodo blanco que ha recibido mensajes dominados de todos los vecinos con rangos más bajos se marca a sí mismo en negro , envía un mensaje dominador a todos sus vecinos y asigna a su parent en T como su dominador.
3. Un nodo gris que recibe un mensaje dominador de un nodo hijo en T por primera vez y que nunca ha enviado un mensaje dominado , cambia su color a negro y envía un mensaje dominador a todos sus vecinos.
4. Cada vez que un nodo negro encuentra que todos sus vecinos son negros y tienen rangos más bajos que él mismo, cambia su color a gris y envía un mensaje de dominio a todos sus vecinos.

La regla 1 garantiza que si el vecino de un nodo blanco se incluye en el CDS, se colorea de gris para ser un nodo vecino de un nodo de CDS. En la Regla 2, si todos los vecinos de rango inferior son de color gris, se puede asignar un nodo como un nodo CDS. La segunda fase finaliza cuando las hojas del árbol están marcadas. Al final de las dos primeras fases, se forma un MIS y los nodos de este conjunto se conectan mediante mensajes

de invitación y de unión . Este algoritmo tiene una complejidad de tiempo de $O(n)$, la complejidad del mensaje de $\mathcal{O}(n \log n)$ y el CDS resultante tiene un tamaño de como máximo $8^{\lfloor \frac{n}{2} \rfloor}$ MinCDS^[+] [2].

14.5 el internet

Internet es la red de computadoras más grande del mundo y consta de miles de millones de dispositivos que incluyen computadoras personales, servidores, teléfonos móviles conectados por varias redes. Internet es una red compleja que exhibe propiedades de pequeño mundo y sin escala como se descubrió experimentalmente [3]. La distancia promedio entre cualquiera de los dos nodos es entre 3 y 9 y una pequeña fracción de los nodos en Internet tienen un número muy alto de conexiones con la mayoría de los nodos que tienen grados bajos. De hecho, el grado promedio de nodos en Internet está entre 2 y 8 [3].

El enrutamiento en Internet es necesario para encontrar rutas eficientes desde una fuente a muchos destinos en la red. Un protocolo de enrutamiento especifica un conjunto de reglas para la transferencia eficiente de datos entre fuentes y destinos. Hay varias opciones para los protocolos de enrutamiento de Internet; La información puede ser almacenada globalmente o descentralizada . Tenemos todos los enrutadores que tienen información de topología completa en el primero y un enrutador es consciente de sus vecinos y los costos de enlace a estos vecinos en el segundo. La red puede ser estática con rutas que cambian lentamente o dinámicasCon frecuentes cambios de ruta. Además, el enrutamiento puede ser sensible a la carga en la red o no. Dos algoritmos de enrutamiento representativos en Internet son el vector de distancia y los algoritmos de estado de enlace. En ambos algoritmos, se asume que el enrutador es consciente de la dirección de sus vecinos y del costo que llega a esos vecinos. Veremos que el problema de enrutamiento en Internet se puede resolver de manera eficiente con algoritmos gráficos.

14.5.1 Protocolo de vector de distancia

El protocolo de enrutamiento por vector de distancia (DVR) utiliza un algoritmo distribuido local y dinámico que se adapta a los cambios y fallas de enlaces. Se basa en el algoritmo de camino más corto dinámico de Bellman-Ford que revisamos en el cap. 3 . La idea principal de este algoritmo es la difusión de las rutas más cortas hacia los vecinos. Cada nodo i envía periódicamente sus distancias más cortas a todos los demás nodos en una actualización de mensaje que incluye la longitud del vector $[1 .. n]$ con la longitud de entrada $[j]$ que muestra su distancia al nodo j . Cuando un nodo recibe estos vectores de los vecinos, actualiza sus rutas más cortas a todos los demás nodos de la red basándose en los valores de longitud . Cuando un nodo i recibe un mensaje de actualización , comprueba las entradas en el vector de longitud y si hay una ruta más corta a un destino j de longitud que la suya, actualiza su tabla de enrutamiento local con la de la de longitud. El problema de la cuenta hasta el infinito se encuentra en este protocolo cuando un nodo se aísla debido a una falla del enlace o se rompe y todos los nodos comienzan a aumentar sus distancias a este nodo.

14.5.2 Algoritmo de estado de enlace

El protocolo de estado de enlace utiliza un algoritmo distribuido global en el que cada enrutador conoce la topología de la red completa y calcula las rutas más cortas a todos los demás nodos de forma individual utilizando el algoritmo de ruta más corta de fuente única

(SSSP) de Dijkstra que vimos en el Capítulo. [7](#). La información de la red se transfiere mediante paquetes de estado de enlace periódico (LSP) que incluyen los cots de alcanzar vecinos, un número de secuencia y un campo de tiempo de vida que disminuye en cada salto. Los nodos reúnen la información que se inunda a través de la red y la utilizan para calcular rutas utilizando el algoritmo SSSP. Las tablas de enrutamiento locales deben ser grandes, ya que toda la información de la red debe almacenarse.

14.5.3 Enrutamiento jerárquico

Hasta este punto, hemos asumido que todos los enrutadores están en una estructura de red plana que no es realista. El enrutamiento jerárquico en Internet se basa en la ubicación jerárquica de los enrutadores, que es más razonable que almacenar información de enrutamiento para millones de destinos en un solo nodo. Los enrutadores se agrupan en sistemas autónomos (AS) y los enrutadores en el mismo AS utilizan el mismo protocolo de enrutamiento, mientras que los enrutadores en diferentes AS pueden ejecutar diferentes protocolos de enrutamiento. Hay un protocolo de enrutamiento inter-AS para la transferencia de datos entre los AS. Si un paquete recibido por un enrutador está destinado a un destino en el mismo AS, se utiliza la ruta más corta calculada por el algoritmo de enrutamiento inter-AS. De lo contrario, el paquete se transfiere a uno de los enrutadores de la puerta de enlace que se entregará al AS de destino. El AS de destino requerido se calcula mediante el protocolo de enrutamiento inter-AS. El protocolo estándar inter-AS de Internet es el protocolo Border Gateway Protocol (BGP) [\[21\]](#).

14.6 La web como red de información

La World Wide Web (WWW), o Web, es una red de información formada por las referencias en los documentos Web. Podemos pensar en la Web como una estructura de nivel superior a través de Internet que consiste en páginas web con enlaces entre sí. Dicha organización de páginas web puede ser convenientemente modelada por un dígrafo, comúnmente denominado gráfico web. Luego podemos buscar una solución a los problemas, como encontrar la página más relevante para una consulta encontrada allí usando este dígrafo. El hyper text transfer protocol (http) se utiliza para la comunicación entre los clientes y servidores web y los enlaces entre páginas Web son llamados hipervínculos .

El gráfico web es muy dinámico, con numerosos nodos (páginas) que se agregan y eliminan en cualquier momento. Es una red compleja que consta de millones de nodos que tienen las propiedades de red complejas que se encuentran comúnmente, como las redes de pequeño mundo y sin escala. En otras palabras, solo hay unos pocos saltos entre dos documentos en la Web y solo un pequeño porcentaje de los nodos tienen grados muy altos, y la mayoría de los nodos tienen grados pequeños. Se comprobó experimentalmente que el gráfico web tiene un componente muy grande fuertemente conectado llamado componente gigante (GC) con otros nodos agrupados de la siguiente manera:

- IN : Este es el conjunto de nodos que han dirigido enlaces al GC.
- OUT : los nodos que han dirigido enlaces desde el GC forman este componente.
- Zarcillos : un zarcillo tiene páginas web conectadas a IN o OUT, pero no son parte de IN, OUT o el GC.
- Nodos desconectados : páginas a las que no se puede acceder desde ningún componente.

Estos componentes en la secuencia IN-GC-OUT forman una estructura de lazo. Nuestro objetivo principal desde el punto de vista gráfico-teórico en la Web es diseñar algoritmos eficientes para un acceso conveniente a las páginas en el gráfico Web. Hay dos algoritmos fundamentales para este propósito: los algoritmos HITS y PageRank descritos en las siguientes secciones.

14.6.1 algoritmo de HITS

Kleinberg presentó un algoritmo llamado selección de tema inducido por hipertexto (HITS) para asignar valores a las páginas en la Web para una búsqueda eficiente durante una consulta [16]. La idea principal de este algoritmo es dar importancia a algunos nodos en el gráfico en función de sus votos al señalar un documento durante una consulta web. Las páginas web relacionadas con una consulta se dividen en los centros que emiten los votos y las páginas apuntadas por los centros se denominan autoridades . Ambas páginas tienen puntuaciones asociadas con ellos para reflejar su importancia. Estos puntajes se pueden asignar según las siguientes reglas [16]:

- Regla de actualización de la autoridad : la puntuación de autoridad de una página es la suma de las puntuaciones centrales de todas las páginas que la apuntan.
- Regla de actualización del concentrador : El puntaje central de una página es la suma de los puntajes de autoridad de todas las páginas a las que apunta.

Con estas reglas, damos más importancia a las autoridades que son señaladas por más centros que otros. Además, si un centro ha señalado a las autoridades que han sido señaladas por muchos centros, su importancia también aumenta. Esta estructura de retroalimentación puede repetirse en un bucle para determinar la importancia de las autoridades que luego se pueden presentar al usuario con respecto a sus prioridades. En la Fig. 14.9 se muestra una implementación de este algoritmo en un solo paso .

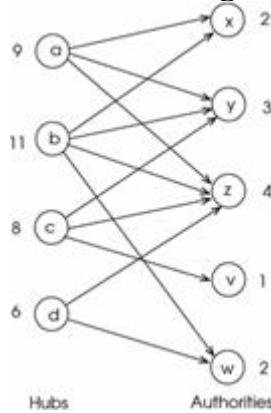


Fig. 14.9 Implementación del algoritmo HITS en un pequeño gráfico web para la primera iteración. Las puntuaciones junto a las autoridades muestran la cantidad de centros que apuntan a ellos. Los centros tienen una puntuación que refleja la puntuación total de las autoridades que señalan

Un posible pseudocódigo de este algoritmo se representa en el Algoritmo 14.1, en el que las puntuaciones de autoridad y centro de las páginas se inicializan en 1 y las reglas anteriores se aplican de forma iterativa [7]. Las puntuaciones finales en cada iteración se calculan dividiendo el valor de la puntuación con la suma de las puntuaciones. Se mostró en [16] las puntuaciones de las páginas de centro y autoridad convergen a medida que el número de iteraciones llega a infinito.

Algorithm 14.1 HITS

```

1: Input :  $P = \{p_1, \dots, p_n\}$  ▷  $P$  is set of  $n$  pages
2:    $k$  steps
3: Output : authority and hub values for all pages
4: for all  $p \in P$  do ▷ initialize authority and hub values
5:    $hub_p \leftarrow 1$ ;  $auth_p \leftarrow 1$ 
6: end for
7: for  $j \leftarrow 1$  to  $k$  do ▷ apply rules for  $k$  steps
8:   for all  $p \in P$  do
9:     apply Authority Update Rule to  $p$  to assign  $auth_p$  values
10:    end for
11:   for all  $p \in P$  do
12:     apply Hub Update Rule to  $p$  to assign  $hub_p$  values
13:   end for
14:    $auth\_total \leftarrow \sum_{p \in P} auth_p$  ▷ calculate total values
15:    $hub\_total \leftarrow \sum_{p \in P} hub_p$ 
16:    $auth_p \leftarrow auth_p / auth\_total$  ▷ calculate normalized values
17:    $hub_p \leftarrow hub_p / hub\_total$ 
18: end for

```

14.6.2 Algoritmo de PageRank

Hemos analizado un gráfico web formado dinámicamente en respuesta a una consulta. Este gráfico bipartito contenía dos tipos diferentes de páginas como centros y autoridades. El gráfico web en general no es bipartito y puede verse como un gráfico dirigido donde la estructura bipartita puede existir solo localmente. Rango de página es un atributo de importancia de una página en el gráfico web en función del número de páginas que lo hacen referencia. Es básicamente una puntuación para una página basada en los votos que recibe de otras páginas. Esta es una métrica sensible para la importancia de una página, ya que la importancia relativa y la popularidad de una página aumentan en la cantidad de páginas que la referencian. El rango de la página se puede considerar como un fluido que corre a través de la red acumulándose en nodos importantes. El algoritmo de rango de página para encontrar la importancia de las páginas en un gráfico web asigna rangos de las páginas en el gráfico web de modo que el valor total del rango de página en la red permanezca constante. Inicialmente asigna valores de rango de $1/n$ a cada página en un n Red de nodos como se muestra en el algoritmo 14.1. El valor de rango actual de una página se distribuye uniformemente a sus enlaces salientes y, a continuación, los nuevos valores de rango de página se calculan como la suma de los pesos de los enlaces entrantes de las páginas. La ejecución de este algoritmo para k pasos da como resultado resultados más refinados para los valores de rango de página como en el algoritmo de Autoridad y Hubs y los valores de rango de página convergen como $\xrightarrow{k \rightarrow \infty}$.

Algorithm 14.2 Page Rank Algorithm

```

1: Input :  $P = \{p_1, \dots, p_n\}$  ▷ set of  $n$  pages
2:    $k$  steps
3: Output : page rank values  $rank_p$ ,  $\forall p \in P$ 
4:  $E_p(in) \leftarrow$  incoming edges to page  $p$ 
5:  $E_p(out) \leftarrow$  outgoing edges from page  $p$ 
6: for all  $p \in P$  do ▷ initialize page rank values
7:    $rank_p \leftarrow 1/n$ 
8: end for
9: for  $r \leftarrow 1$  to  $k$  do ▷ apply for  $k$  steps
10:   for all  $p \in P$  do
11:     for all  $e \in E_p(out)$  do
12:        $w_e \leftarrow rank_{p_e} / |E_p(out)|$ 
13:     end for
14:      $rank_p \leftarrow \sum_{e \in E_p(out)} w_e$  ▷ find sum of the weights of all links pointing to  $p$ 
15:   end for
16: end for

```

La figura 14.10 muestra los pesos iniciales de los bordes utilizando este algoritmo en un pequeño gráfico web con cinco nodos .

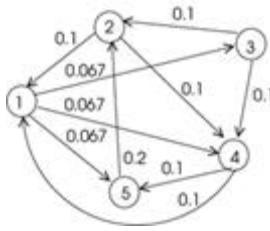


Fig. 14.10 Implementación del algoritmo PageRank en un pequeño gráfico web para encontrar valores de peso por borde

La ejecución del algoritmo de Page Rank para la gráfica de la Fig. 14.10 para las tres primeras iteraciones se muestra en la Tabla 14.1. Podemos ver que la página 3 tiene el rango más bajo ya que está señalado por una sola página y la página 4 tiene un rango ligeramente más alto que los demás, ya que es la única página apuntada por 3 páginas. Una página que no apunta a muchas páginas como otras páginas pero tiene muchos bordes de entrada puede tener puntuaciones altas después de un número significativo de iteraciones. Esta situación se corrige con la introducción del factor de amortiguación d , que se utiliza para reducir los valores de rango de página en $(1 - d)/n$ [16].

Tabla 14.1 Valores de Page Rank de los nodos web de la figura 14.10

Vértices	1	2	3	4	5
n_{out}	3	2	2	2	1
$k = 1$: peso / borde	0.067	0.1	0.1	0.1	0.2
rango	0.2	0.3	0.067	0.267	0.167
$k = 2$: peso / borde	0.067	0.15	0.034	0.134	0.167
rango	0.284	0.201	0.034	0.251	0.201
$k = 3$: peso / borde	0.095	0.101	0.017	0.067	0.167
rango	0.168	0.184	0.095	0.213	0.162

14.7 Notas del capítulo

Hemos descrito y revisado redes complejas fundamentales que son redes biológicas, redes sociales, redes tecnológicas y redes de información. Todas estas redes exhiben propiedades de redes complejas de pequeño mundo y sin escala. Luego, analizamos algunos de los ejemplos representativos de estas redes.

Las redes PPI son redes biológicas que existen fuera de los núcleos en la célula y tres problemas principales encontrados en estas estructuras son la detección de agrupaciones, la búsqueda de motivos de red y la alineación de dos redes. Los clusters o subgrafos densos en estas redes pueden indicar una actividad densa en esta región y revisamos dos algoritmos con el propósito de descubrir clusters. Los motivos de la red se repiten subgrafos y encontrarlos es otra tarea importante y un área de investigación en las redes PPI y otras redes biológicas. Se asume que estas estructuras tienen alguna funcionalidad básica y se consideran los bloques de construcción básicos de los organismos. Además, encontrar motivos de red similares en dos o más organismos puede indicar una ascendencia común. La alineación de dos o más redes muestra sus similitudes y se usa con frecuencia para comparar varias redes.

Las redes sociales están formadas por individuos o grupos de individuos, y la búsqueda de comunidades que son grupos estrechamente relacionados proporciona información sobre la estructura de una red social. Revisamos dos algoritmos principales para este propósito y también describimos las relaciones y las redes sociales equilibradas y

desequilibradas. Las redes inalámbricas ad hoc son redes tecnológicas como Internet. Dos tipos principales de estas redes son las redes móviles ad hoc y las redes de sensores inalámbricas. Describimos en detalle varios algoritmos de agrupamiento en estas redes que son algoritmos distribuidos ejecutados por nodos individuales de la red. También revisamos los principales algoritmos de enruteamiento en Internet, que son extensiones de los algoritmos de enruteamiento descritos en el Capítulo 7. Por último, revisamos la Web, que es una red de información, y analizamos dos algoritmos para atribuir importancia a las páginas Web para consultas web eficientes. El primer algoritmo llamado HITS divide los nodos en concentradores y autorizaciones durante una consulta y asigna puntuaciones a estos nodos en función de las puntuaciones de los nodos a los que apuntan y apuntan. El algoritmo de PageRank es más general y tiene un amplio uso en la Web.

Podemos decir que la agrupación en clústeres es un área de investigación fundamental en todas estas redes y las heurísticas se utilizan ampliamente para encontrar las regiones densas en los gráficos grandes que representan estas redes. No existe un solo algoritmo que se ajuste a todas las necesidades de la aplicación y los resultados experimentales obtenidos junto con su complejidad se aceptan comúnmente como la bondad de un algoritmo. Se necesitan algoritmos paralelos y distribuidos en todas las áreas de investigación en estas redes complejas. Una encuesta de algoritmos de agrupación en paralelo con los nuevos propuestos se da en [8]. Los algoritmos de agrupación en clústeres distribuidos en redes inalámbricas ad hoc se han estudiado ampliamente, pero no son válidos para la agrupación en otras redes complejas. Solo hay pocos estudios para la búsqueda de motivos de red paralela y la alineación de redes paralelas que son áreas potenciales de investigación.

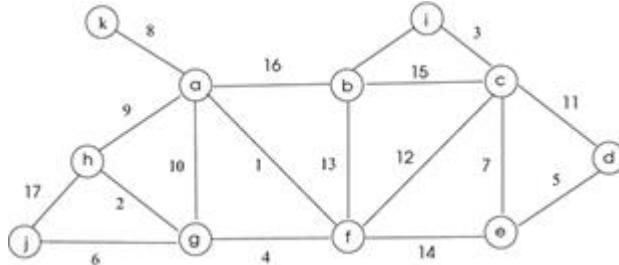


Fig. 14.11 La gráfica de muestra para el Ejercicio 1

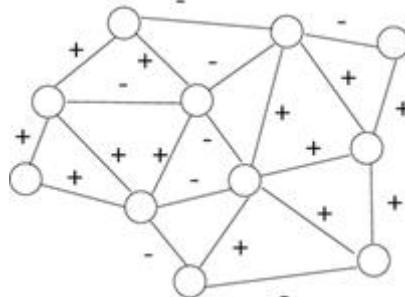


Fig. 14.12 La gráfica de muestra para el Ejercicio 2

Ceremonias

- Calcule el MST de la gráfica que se muestra en la figura 14.11 usando cualquier algoritmo. Luego, divida este gráfico en tres grupos eliminando los dos bordes más pesados del MST. Encuentre el costo total de los bordes entre clústeres y el costo de los bordes dentro de los clústeres y calcule
1. la calidad del clúster.
 2. Averigüe si la red social representada en el gráfico etiquetado de la figura 14.12 está balanceada o no al verificar cada triángulo. Sugiere qué hacer para equilibrar esta red.

3. Calcule los puntajes de las páginas del centro y de autoridad del gráfico de consulta web de la figura [14.13](#) para tres iteraciones del algoritmo HITS. Determine si existe una convergencia de valores de puntaje o no.
4. En la figura [14.14 se](#) muestra un gráfico web . Implemente el algoritmo de PageRank en este gráfico para cuatro iteraciones mostrando los puntajes de rango de página para cada página.

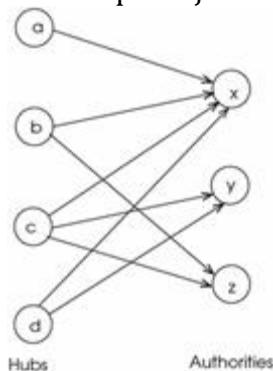


Fig. 14.13 La gráfica de muestra para el Ejercicio 3

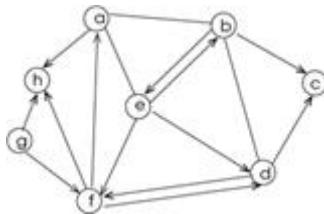


Fig. 14.14 La gráfica de muestra para el Ejercicio 4

Referencias

1. Akram VK, Orhan Dagdeviren O (2015) Sobre problemas de conectividad k en sistemas distribuidos. Métodos avanzados para el análisis de redes complejas. IGI Global
2. Alzoubi KM, Wan PJ, Frieder O (2002) Nuevo algoritmo distribuido para conjuntos dominantes conectados en redes inalámbricas ad hoc. En: Actas de la 35a conferencia internacional de Hawaii sobre ciencias del sistema, Big Island, Hawaii
3. Caldarelli G, Vespignani A (2007) Estructura a gran escala y dinámica de redes complejas: de tecnología de la información a finanzas y ciencias naturales. Sistemas complejos y ciencia interdisciplinaria. Empresa de publicación científica mundial. Capítulo 8, ISBN-13: 978-9812706645
[Referencia cruzada](#)
4. Cokusu D, Erciyes K, Dagdeviren O (2006) Un algoritmo de agrupamiento basado en conjuntos dominantes para redes móviles ad hoc. Int Conf Comput Sci 1: 571–578
[MATES](#)
5. Das B, Bharghavan V (1997) Enrutamiento en redes ad-hoc utilizando conjuntos dominantes conectados mínimos. En: Conferencia internacional IEEE sobre comunicaciones (ICC97), vol.1, pp. 376380
6. Dongen SV (2000) Agrupación de gráficos por simulación de flujo. Doctor en Filosofía. Tesis, Universidad de Utrecht, Países Bajos
7. Erciyes K (2014) Internet y la web. En: Redes complejas: una perspectiva algorítmica. Prensa CRC. ISBN-10: 1466571667, ISBN-13: 978-1466571662
[Referencia cruzada](#)
8. Erciyes K (2015) Algoritmos distribuidos y secuenciales para Bioinformática, Springer, Berlín (Chaps. 10 y 11)

9. Fiedler M (1973) Conectividad algebraica de gráficos. Czechoslov Math J 23: 298–305
[Matemáticas](#) [Matemáticas](#)
10. Gerla M, Tsai JTC (1995) Red de radio multimedia, móvil, multimedia. Wirel Netw 1: 255–265
[Referencia cruzada](#)
11. Girvan M, Newman MEJ (2002) Estructura comunitaria en redes sociales y biológicas. PNAS 99: 7821–7826
[MathSciNet](#) [Crossref](#)
12. Hagen L, Kahng AB (1992) Nuevos métodos espetrales para la partición y agrupamiento de corte de relación. IEEE Trans Comput Aided Des Integr Circuits Syst 11 (9): 1074-1085
[Referencia cruzada](#)
13. Harary F (1953) Sobre la noción de equilibrio de un gráfico firmado. Mich Math J 2 (2): 143–146
[MathSciNet](#) [Crossref](#)
14. Organización Internacional de Normalización (1989-11-15) ISO / IEC 7498-4: 1989 - Tecnología de la información - Interconexión de sistemas abiertos - Modelo de referencia básico: denominación y direccionamiento. Portal de mantenimiento de normas ISO. Secretaría Central de la ISO. Consultado el 17 de agosto de 2015
15. Jorgic M, Goel N, Kalaichevan K, Nayak A, Stojmenovic I (2007) Detección localizada de conectividad k en redes inalámbricas ad hoc, actuadores y sensores. En: Actas de la 16^a conferencia internacional sobre comunicaciones y redes de computadoras (ICCCN 2007), pp 33–38
- dieciséis. Kleinberg J (1999) Fuentes autorizadas en un entorno con hipervínculos. J ACM 46 (5): 604–632
[MathSciNet](#) [Crossref](#)
17. Lin CR, Gerla M (1997) Clustering adaptativo para redes inalámbricas móviles. IEEE J Sel Areas Commun 15 (1): 1265–1275
[Referencia cruzada](#)
18. Mount DM (2004) Bioinformática: secuencia y análisis del genoma, 2^a ed . Prensa de laboratorio de Cold Spring Harbor, NY. ISBN 0-87969-608-7
19. Newman M (2003) Rápido algoritmo para detectar la estructura de la comunidad en redes. Phys Rev E 69: 066133
[Referencia cruzada](#)
20. Olman V, Mao F, Wu H, Xu Y (2009) Algoritmo de agrupamiento paralelo para grandes conjuntos de datos con aplicaciones en bioinformática. IEEE / ACM Trans Comput Biol Bioinform6: 344–352
[Referencia cruzada](#)
21. RFC 4271 - Un protocolo de puerta de enlace 4 (BGP-4). www.ietf.org
22. Titz B, Rajagopala SV, Goll J, Hauser R, McKevitt MT, Palzkill T, Uetz P (2008) La proteína binaria interactome de *Treponema pallidum*, la sífilis espiroqueta. PLOS UNO 3 (5): e2292
[Referencia cruzada](#)
23. Wu J, Li H (1999) en el cálculo conectado conjunto dominante para eficiente de encaminamiento en redes inalámbricas ad hoc. En: Actas del tercer taller internacional sobre algoritmos y métodos discretos para computación móvil y comunicaciones, págs. 7–14.

15. epílogo

K. Erciyes¹

(1) Instituto Internacional de Computación, Universidad Ege , Izmir, Turquía

K. Erciyes

Correo electrónico: kayhan.erciyes@izmir.edu.tr

15.1 Introducción

Hemos descrito los fundamentos de los algoritmos de grafos secuenciales, paralelos y distribuidos en la Parte I. Luego, revisamos los algoritmos básicos de grafos que se pueden usar como bloques de construcción para resolver problemas más complicados relacionados con los gráficos en la Parte II que formaron el núcleo del libro. Analizamos soluciones algorítmicas secuenciales, paralelas y distribuidas a los problemas estudiados en esta parte. En la parte final, primero revisamos brevemente los algoritmos de gráficos algebraicos y dinámicos. Luego, nuestro énfasis estaba en los grandes gráficos que representan grandes redes de la vida real, comúnmente llamadas redes complejas. Los algoritmos de grafos para redes tan grandes requerían nuevos enfoques de análisis y algoritmos eficientes como señalamos.

Nuestro objetivo en este capítulo final es proporcionar brevemente una guía cuando se trata de un nuevo problema que se puede modelar mediante un gráfico. Intentamos trazar un mapa de ruta cuando se necesita un algoritmo gráfico para una tarea que tenemos y es posible que no tengamos un algoritmo gráfico existente para resolver el problema. Una vez que tengamos un método viable para el problema en cuestión, debemos investigar las opciones de uso de algoritmos secuenciales, paralelos o distribuidos para este propósito. Además, pueden usarse convenientemente métodos relativamente nuevos de algoritmos de grafos algebraicos y dinámicos. Los algoritmos de gráficos algebraicos permiten el uso de funciones de biblioteca de matrices existentes en forma secuencial o paralela, y se necesitan algoritmos de gráficos dinámicos cuando se producen eliminaciones de bordes e inserciones como en muchas redes de la vida real. Describimos dónde usar estos métodos y concluimos el capítulo con un estudio de caso.

15.2 Hoja de ruta para problemas difíciles

Ya sabemos que la mayoría de los problemas encontrados en el mundo de los gráficos son NP-Hard, excepto algunos, como el problema de coincidencia en el que buscamos los bordes desunidos con el tamaño máximo. Cualquier problema nuevo que enfrentemos

probablemente no tendrá una solución en el tiempo polinomial. Intentaremos especificar los pasos a seguir en tal caso de la siguiente manera.

- La comprensión profunda de los algoritmos de grafos básicos es muy útil. Estos algoritmos, como los algoritmos DFS y BFS, pueden usarse como bloques de construcción para resolver un problema más complicado. En muchos casos, se puede utilizar una forma modificada del algoritmo básico. Vimos cómo un algoritmo DFS simple con algunas modificaciones se puede usar para varios problemas, como encontrar puntos de articulación, puentes, componentes fuertemente conectados y los bloques de una gráfica.
- Cuando se trata de un problema de gráfica NP-dura, podemos buscar un algoritmo de aproximación si existe. Sin embargo, en muchos casos, los algoritmos de aproximación son raros y tratar de diseñar uno nuevo no es una tarea trivial. Después de todo, si se puede llegar a un nuevo algoritmo de aproximación que tenga una relación de aproximación mejor probada que las existentes, esto se puede publicar en un artículo. En algunos casos, podemos optar por utilizar un algoritmo de aproximación que tenga una relación de aproximación ligeramente peor que la mejor disponible, debido a la complejidad de implementar un mejor algoritmo. Por ejemplo, encontrar la cobertura de vértice mínima de un gráfico utilizando la coincidencia es un algoritmo simple con una relación de aproximación de 2.
- En el caso más común, el uso de heurísticas es inevitable. La elección de una heurística depende en gran medida de la naturaleza del problema en cuestión. Cuando vamos a diseñar un nuevo algoritmo de gráfico o modificar uno existente, solo tenemos unas pocas propiedades para empezar. Especialmente en el caso de alterar un algoritmo existente para nuestro propósito, se puede incorporar el grado de nodos para romper simetrías o para seleccionar directamente un nodo para trabajar. También se puede utilizar grado de vecinos o grado de k -vecinos de tienda. Podemos definir nuevos parámetros que utilizan grados y vecinos y su relación. El coeficiente de agrupamiento de un vértice, por ejemplo, muestra cuán bien conectados están los vecinos de ese vértice.
- Para problemas relacionados con gráficos grandes, podemos optar por un algoritmo de aproximación con mejor rendimiento que un algoritmo polinómico para resolver el problema debido a los altos tiempos de ejecución involucrados. Además, la paralelización siempre es útil para mejorar el rendimiento.
- Una red de computadoras consiste en nodos autónomos que funcionan independientemente. Para problemas de red, necesitamos algoritmos distribuidos eficientes que son ejecutados por los nodos de la red. Estos algoritmos comúnmente usan la información del vecino para encontrar soluciones locales que luego se usan para encontrar una solución global a un problema.

15.3 ¿Son diferentes los algoritmos de grafos grandes?

Podemos usar cualquiera de los algoritmos que hemos desarrollado en la Parte II para gráficos grandes. Sin embargo, incluso un algoritmo de tiempo lineal puede ser

problemático debido al tamaño de estos gráficos grandes. El empleo de las siguientes técnicas es frecuentemente necesario en gráficos tan grandes.

- Uso de heurísticas : las heurísticas se usan comúnmente en la resolución de problemas de NP-hard graph, como señalamos. En algunos casos, uno puede optar por un algoritmo heurístico que tiene, por ejemplo, $O(n)$ complejidad utilizando un algoritmo heurístico que un algoritmo determinista que tiene $\mathcal{O}(n^2)$ complejidad. Esta mejora en el rendimiento puede ser significativa para que un gráfico grande decida utilizar la solución heurística.
- Algoritmos paralelos escalables : se necesitan algoritmos paralelos en el análisis de gráficos grandes que representan redes complejas debido a la magnitud del gráfico. Este método es una necesidad más que una elección en tales implementaciones.
- Algoritmos distribuidos: en el caso de una gran red de computadoras, como Internet o una red de sensores inalámbricos, se necesitan algoritmos distribuidos.

15.4 Conversiones: ¿Cuándo son útiles?

Tenemos tres modos para algoritmos de gráficos: secuencial, paralelo y distribuidos como se enfatiza en todo el libro. Entonces podremos tener las siguientes conversiones posibles:

- Secuencial a paralelo : podemos diseñar un algoritmo paralelo desde cero o convertir un algoritmo secuencial existente en uno paralelo. Dos métodos comúnmente utilizados en este último son la distribución de datos (paralelismo de datos) y el código de distribución (paralelismo funcional) cuando tenemos un sistema de procesamiento paralelo de memoria distribuida, como hemos revisado en el Capítulo. 4 . Al utilizar el modelo de memoria compartida del procesamiento paralelo, debemos proporcionar control del espacio de direcciones compartido mediante bloqueos, semáforos u otros mecanismos. El paso de mensajes se usa ampliamente para el procesamiento paralelo en computadoras con memoria distribuida.
- Secuencial a distribuido: Los algoritmos distribuidos se ejecutan en los nodos de una red de computadoras para resolver un problema relacionado con la red. Cada nodo coopera comunicándose con sus vecinos para encontrar el resultado, y un nodo especial suele llegar a una solución global que luego puede transferir el resultado a nodos individuales. Podemos diseñar un algoritmo distribuido desde cero o convertir un algoritmo secuencial existente en uno distribuido como en el caso paralelo. La conversión de un algoritmo secuencial a uno distribuido requiere un análisis detallado de qué comunicar y cuándo. Un problema bien conocido en una red de computadoras es el enrutamiento donde buscamos las rutas más cortas entre cada par de nodos. Describimos cómo convertir el algoritmo de Bellman-Ford secuencial que usa la programación dinámica para encontrar rutas más cortas desde un nodo fuente. La principal diferencia entre los algoritmos secuenciales y distribuidos es que cada nodo calcula su distancia más corta a la raíz en función de su información actual en cada ronda. Necesitamos la sincronización en cada ronda que puede ser realizada por un nodo especial. En resumen, debemos asegurarnos de que la sincronización sea impecable al utilizar un algoritmo distribuido síncrono. Los

algoritmos asíncronos distribuidos son más flexibles, pero en general son más difíciles de diseñar que los síncronos.

- Distribuido a paralelo : Supongamos que tenemos una red y un algoritmo distribuido A para resolver un problema B relacionado con la red y cada vértice del gráfico representa un nodo informático de la red en el sentido general. Nuestro objetivo es proporcionar una solución a P en un entorno de computación paralelo y luego transferir los resultados a cada nodo. Una posible conversión del algoritmo A a un algoritmo paralelo B puede incluir los siguientes pasos. Rompemos el gráfico de forma iterativa utilizando un método adecuado, como la contracción del borde o la contracción de la estrella para obtener supernodos, cada uno de los cuales representa un número de nodos con bordes incidentes del gráfico original. Luego podemos asignar cada supernodo a un sistema de procesamiento paralelo donde cada proceso p resuelve el problema para el supernodo al que está asignado. Puede comunicarse con los vecinos del supernodo en el gráfico engrosado utilizando el algoritmo A distribuido . Los resultados se pueden recopilar en un nodo raíz que calcula el resultado global mediante la fusión de los resultados de cada proceso p y transfiere los resultados a cada nodo. Necesitamos considerar los vértices de borde entre cada partición con cuidado, ya que la ruptura de simetría, como el uso de identificadores únicos, puede ser necesaria para asignar un vértice de borde a un vecinoSupernodo para mayor comodidad. Consideraremos el problema de coincidencia en una red de computadoras y queremos encontrar la solución en una computadora paralela con k procesos. Deberíamos tener cada nodo de red para saber si alguno de sus bordes incidentes está incluido en la M final coincidente . Podemos ampliar la gráfica a k subgrafos de supernodos G_1, \dots, G_k y asigna cada G_i a un proceso p_i tener cada uno p_i resolver el emparejamiento M_i en su supernodo G_i . Los procesos ahora pueden comunicarse con supernodos vecinos como si fueran los nodos de una red para encontrar la solución global utilizando un supervisor o de manera totalmente distribuida.
- Paralelo a distribuido : dado un algoritmo paralelo B para resolver un problema gráfico P , queremos saber si convertir B a un algoritmo distribuido A es más conveniente que diseñar un algoritmo distribuido desde cero o convertir un algoritmo secuencial existente en uno distribuido para problema p . Nuestra estrategia en esta conversión puede ser trabajar en la dirección inversa del método utilizado en el algoritmo distribuido a la conversión de algoritmo paralelo esta vez. En lugar de engrosar, intentamos refinar el algoritmo paralelo al nivel donde cada proceso paralelo p_i es responsable de un solo vértice del gráfico y, por lo tanto, podemos dividir cada fila de la adyacencia o la matriz de distancia en un proceso distinto. La restricción que tenemos es que el proceso solo puede comunicarse con sus vecinos. Si esto es posible, tenemos un algoritmo distribuido que puede ejecutarse en nodos autónomos de la red.

15.5 Implementación

Una vez que tengamos alguna forma de resolver el problema, debemos considerar las opciones de implementación. Vimos tres formas de implementaciones que consideran el flujo de datos en un algoritmo para un problema dado; secuencial, paralela o distribuida. Desde otra perspectiva, las estructuras de datos utilizadas y el entorno en el que se practica el algoritmo juegan un papel importante para clasificar el algoritmo como clásico, algebraico, dinámico o, a veces, todo.

15.5.1 Secuencial, paralelo o distribuido

Hemos dedicado la mayor parte del libro a algoritmos de grafos secuenciales con algoritmos paralelos de muestra y distribuidos a problemas específicos de grafos. El tamaño del problema y el entorno en el que se implementa es crucial para decidir si debemos buscar un algoritmo paralelo o distribuido. Para un gráfico grande que representa una red compleja, se requieren comúnmente algoritmos paralelos para proporcionar eficiencia. Por otro lado, si queremos resolver un problema de red en el que los nodos de red participan en la búsqueda de la solución, debemos buscar un algoritmo distribuido para la tarea en cuestión. En muchos casos, estos límites no son tan claros. Por ejemplo, es posible que tengamos una red de sensores inalámbricos con un grupo de nodos informáticos utilizados como sumidero y cientos de nodos de detección. La solución de un problema como el enrutamiento o problemas más complicados pueden ser manejados por nodos que realizan alguna operación local y envían sus datos al sumidero usando un algoritmo distribuido; el sumidero encuentra la solución de manera eficiente utilizando computación paralela y luego envía el resultado a los nodos individuales utilizando el algoritmo distribuido.

Como regla general, podemos decir que la computación paralela es necesaria para resolver problemas relacionados con redes complejas representadas por grandes gráficos. Estos problemas, como la búsqueda de motivos de red o la alineación de redes, son duros para NP en muchos casos, lo que requiere el uso de heurísticas, e incluso dichas implementaciones llevan un tiempo considerable debido al gran tamaño del gráfico. La aceleración obtenida por tal algoritmo paralelo es la relación del tiempo secuencial al tiempo de computación paralelo y la eficiencia se define como la aceleración dividida por el número de elementos de procesamiento utilizados.

Cuando se trata de un problema en el que los nodos de la red representan vértices del gráfico, debemos buscar algoritmos distribuidos eficientes. Vimos que los algoritmos distribuidos síncronos de un solo iniciador se utilizan con frecuencia en estos casos debido a su relativa facilidad de implementación. El número de rondas y el número total de mensajes intercambiados para terminar el algoritmo nos proporcionan una buena indicación de su desempeño.

15.5.2 ¿Clásico, algebraico, dinámico o todo?

La parte principal de este libro, que incluye las partes I y II y la mayor parte de la Parte III, está dedicada a los algoritmos gráficos que se pueden considerar como algoritmos clásicos en el sentido de que se emplean técnicas algorítmicas tradicionales como los métodos codiciosos, dinámicos y de dividir y conquistar. Revisamos el método alternativo y relativamente más reciente de diseño de algoritmos de gráficos algebraicos en el Cap. [12](#). Este método hace uso de matrices principales asociadas con una gráfica: matriz de

adyacencia, matriz de incidencia y matriz laplaciana. La revisión de soluciones a pocos problemas gráficos mostró que los rendimientos de tales algoritmos son comúnmente inferiores a sus contrapartes clásicas. Sin embargo, el método algebraico proporciona algoritmos más simples y, lo que es más importante, una amplia biblioteca de operaciones matriciales en forma secuencial y paralela están disponibles para su uso en tales algoritmos. Por ejemplo, si el algoritmo algebraico involucra la multiplicación de matrices, ya tenemos un método para realizar esta operación en paralelo por fila, columna o partición de bloque. Por lo tanto, no necesitamos gastar mucho tiempo para encontrar un algoritmo paralelo para el problema estudiado si podemos formar la solución algebraica utilizando operaciones de matriz básicas.

Los algoritmos de gráficos dinámicos son una necesidad más que una elección, ya que las redes reales casi siempre son dinámicas con la adición y eliminación frecuentes de bordes. Ejemplos de estas redes son Internet, la Web, las redes sociales y las redes biológicas de la célula. En lugar de ejecutar un algoritmo estático (clásico o algebraico) conocido para la red modificada desde cero, es sensato diseñar algoritmos que hagan uso de la información de red existente y resuelvan el problema más rápido. Notamos que el principal desafío en el diseño de algoritmos de gráficos dinámicos reside en el diseño de estructuras de datos inteligentes para que las modificaciones puedan manejarse rápidamente.

Por último, investigamos los algoritmos de gráficos algebraicos dinámicos que son incluso un área menos investigada que los algoritmos de gráficos algebraicos y dinámicos. Cuando tenemos un método algebraico para resolver un problema gráfico, existe la posibilidad de tener una versión dinámica de esta técnica mediante el uso de algún resultado conocido del álgebra lineal y también mediante el uso de una operación de biblioteca de matriz dinámica. Hemos visto un procedimiento de este tipo en la comparación algebraica dinámica cuando se combinó una solución algebraica a este problema con operaciones de matriz dinámica y un teorema del álgebra lineal.

En conclusión, el diseño del algoritmo de grafos utilizando los enfoques tradicionales se utilizará para problemas de tamaño pequeño a moderado. Además, comúnmente proporcionan los métodos de diseño básicos para ser utilizados en algoritmos algebraicos o dinámicos para gráficos. Sin embargo, cuando buscamos una solución en un gráfico grande, se necesita un procesamiento paralelo y dicha operación puede manejarse más convenientemente mediante algoritmos de gráficos algebraicos. Los algoritmos de gráficos dinámicos son necesarios para las redes dinámicas para un mejor rendimiento. En muchos casos, las redes dinámicas como las redes de interacción de proteínas, las redes sociales e Internet son grandes. Por lo tanto, podemos concluir que los algoritmos de gráficos algebraicos dinámicos continuarán siendo un área de investigación efectiva en el futuro.

15.6 Un estudio de caso: Construcción de la red troncal en WSNs

Vamos a elaborar un estudio de caso para ilustrar las pautas que hemos expresado hasta ahora. Necesitamos agrupar los nodos de una WSN para obtener los beneficios generales que se obtienen de dicho proceso. La elección de un clusterhead (CH) facilita varias tareas, como el enrutamiento, ya que el CH puede realizar estas tareas en nombre de los nodos de su cluster. Esta estructura jerárquica es claramente útil en la gestión de cualquier establecimiento, incluidos los países. Sin embargo, necesitamos un árbol de expansión en

la WSN para transmitir varios comandos desde la raíz y también agregar los datos de los sensores a la raíz. Podemos usar dos algoritmos distintos para nuestro propósito, pero una mirada más cercana revela que dos tareas pueden ser realizadas por un algoritmo de una manera más eficiente. Llamaremos a este proceso que abarca la agrupación en clústeres basada en árboles.. La idea general del nuevo algoritmo es construir grupos y el árbol de expansión simultáneamente. Cada grupo será un subárbol en el árbol de expansión.

Ahora buscamos un algoritmo de árbol de expansión y vemos si podemos modificar este algoritmo para nuestro propósito. Vimos cómo construir un árbol de expansión utilizando inundaciones en el Cap. 5 . Erciyes et al. presentó un algoritmo que construye un árbol de expansión y forma grupos de manera simultánea utilizando la inundación [3]. La idea principal de este algoritmo es mantener un registro de la profundidad del árbol de expansión obtenida durante las iteraciones y asignar los nodos del árbol dentro de cada salto a un grupo.

El nodo raíz inicia el algoritmo enviando el primer mensaje de sondeo y los nodos que reciben este mensaje por primera vez marcan al remitente como su padre y devuelven un mensaje de confirmación , de lo contrario, el remitente responde con un mensaje nack como en el algoritmo de inundación original. Además, la profundidad del grupo de subárbol se determina antes de la ejecución en la variable \max_depth y en cada recepción de mensajes por parte de los nodos, la cuenta variable se incrementa y se compara con \max_depth . El final del grupo se marca cuando se alcanza esto y se inicia otro grupo como se muestra en el algoritmo 15.1 [2]. Un nodo no visitado que recibe una sonda de mensaje con un campo de conteo de 0 se convierte en un CH de su subárbol. Todos los nodos distintos de la raíz se clasifican como CH, ordinarios o de hoja al final del algoritmo. Los grupos formados con este algoritmo en una muestra WSN se muestran en la Fig. 15.1 .

Algorithm 15.1 ST_Clust

```

1: int parent ← ⊥
2: set of int childs ← { } , others ← { }
3: message types probe, ack, reject
4: states CH, ordinary, leaf
5: if i = root then                                ▷ root initiates tree construction
6:   send probe(0, 0) to N(i)
7:   parent ← i
8: end if
9:
10: while childs ∪ others ≠ (N(i)\{parent}) do
11:   receive msg(j)
12:   case msg(j).type of
13:     probe(cid, n_hops) : if parent = ⊥ then      ▷ probe received first time
14:       parent ← j
15:       send ack to j
16:       if count = 0 then                          ▷ I am the clusterhead
17:         state ← CH
18:         cid ← i
19:       else if count = ds then
20:         state ← leaf
21:       else state ← ordinary
22:       count ← (count + 1) MOD max_depth
23:       send probe(cid, count) to N(i)\{j}
24:     else send reject to j                      ▷ probe received before
25:     ack : childs ← childs ∪ {j}                  ▷ include j in children
26:     reject : others ← others ∪ {j}                ▷ include j in unrelated
27:   end while

```

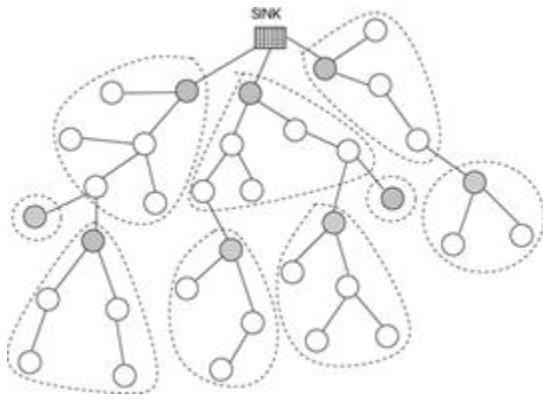


Fig. 15.1 Formación de agrupamiento en una muestra WSN con un nodo sumidero. El conteo máximo de saltos es 2 y los CH se muestran en gris con grupos en regiones discontinuas

Teorema 15.1. Complejidad del tiempo de ST_{Clust} es $O(d)$, donde d es el diámetro de la red y su complejidad de mensaje es $O(m)$.

Prueba El diámetro de la gráfica es el límite superior del tiempo requerido para el algoritmo, ya que es la distancia más lejana entre dos vértices. Dado que cada borde será atravesado dos veces por pares de mensajes sonda / ack o sonda / rechazo , la complejidad del mensaje del algoritmo es $O(m)$.

Banerjee y Khuller [1] también propusieron un protocolo basado en un árbol de expansión agrupando las ramas de un árbol de expansión en grupos de un tamaño objetivo aproximado.

15.7 Conclusiones

Podemos resumir brevemente los pasos a seguir cuando necesitamos decidir sobre un algoritmo para el problema gráfico que tenemos:

- Investigue cuidadosamente si el problema puede resolverse mediante una combinación de algoritmos de grafos básicos como BFS directo o modificado y DFS. Vimos que BFS se usa de manera efectiva para encontrar los valores de intermediación de borde en un gráfico y DFS para
1. varios problemas de conectividad. Si esto no es posible, podemos buscar un algoritmo de aproximación que funcione en tiempo lineal. Si no podemos encontrar un algoritmo adecuado, nuestra mejor opción será utilizar algunas heurísticas que den buenos resultados la mayor parte del tiempo.
 2. Si el gráfico es grande, siempre vale la pena intentar parallelizar el algoritmo. En este caso, vimos la partición de la matriz de adyacencia de la gráfica para obtener soluciones factibles en muchos casos. También podemos dividir el gráfico contratando primero sus vértices para obtener un gráfico más simple y luego dividir el gráfico simple. Para tales gráficos, el uso de algoritmos de gráficos algebraicos proporciona facilidad en la parallelización, ya que las operaciones de matriz paralelas ya están disponibles.
 3. Si necesitamos diseñar un algoritmo distribuido para una red de computadoras como una WSN, podemos intentar convertir el algoritmo secuencial a uno distribuido como un enfoque general. Es posible que esto no sea una tarea trivial, especialmente si el algoritmo secuencial se basa en gran medida en datos globales, ya que los algoritmos distribuidos generalmente funcionan utilizando datos locales alrededor de los nodos.

Referencias

1. Banerjee S, Khuller S (2000) Un esquema de agrupamiento para el enrutamiento jerárquico en redes inalámbricas. Informe técnico CS-TR-4103, Universidad de Maryland, College Park
2. Erciyes K (2013) Algoritmos de grafos distribuidos para redes informáticas. Serie de comunicaciones y redes informáticas. Springer, Berlín, pp 247–248. ISBN 978-1-4471-5172-2
[Referencia cruzada](#)
3. Erciyes K, Ozsoyeller D, Dagdeviren O (2008) Algoritmos distribuidos para formar árboles de expansión basados en clústeres en redes de sensores inalámbricos. ICCS 2008. LNCS. Springer, Berlín, pp 519–528
[Referencia cruzada](#)

Apéndice

Convenciones de pseudocódigo

A.1 Introducción

Mostramos las convenciones de escritura de pseudocódigo utilizadas a lo largo del libro aquí. Seguimos las principales adaptaciones, como en [1, 2]. Los principales puntos a destacar son los siguientes:

- Cada algoritmo comienza con la declaración de su entrada y la salida producida por él.
- Las líneas del algoritmo están numeradas para referencia.
- Utilizamos sangrías para mostrar bloques que se ejecutan dentro de las estructuras de control.
- Un procedimiento que se usa desde el cuerpo principal del algoritmo se muestra explícitamente.

Las estructuras de datos, las estructuras de control y las estructuras de algoritmos distribuidos se describen en las siguientes secciones.

A.2 Estructuras de datos

Cada línea de un algoritmo es una declaración que comúnmente evalúa una expresión o realiza una función específica, como llamar a un procedimiento. Una expresión consta de constantes, variables y operadores. La declaración de una variable se realiza como se muestra en los siguientes ejemplos:

```
boolean visited ← false
```

Aquí una variable booleana visitada se declara y se inicializa a valor falso .

```
setofint vertices ← [Ø]
```

Un conjunto de vértices que contendrá valores enteros se declara y se inicializa como vacío.

La asignación en los ejemplos anteriores se realiza utilizando el `←` operador. El valor a la derecha de este operador se evalúa y asigna a la variable de la izquierda en el sentido habitual. A veces, tenemos dos o más expresiones cortas que se colocan en la misma línea del algoritmo separadas por punto y coma de la siguiente manera. Tenga en cuenta que una línea de instrucción no termina con un punto y coma.

```
i ← 5; j ← 8
```

Las convenciones de pseudocódigo utilizadas en el libro se muestran en la Tabla [A.1](#).
Tabla 1 Convenciones generales de algoritmos.

Notación	Sentido
----------	---------

$a \leftarrow b$	Asignación
\equiv	Comparacion de igualdad
\neq	Comparación de la desigualdad

verdadera , falsa Lógico verdadero y falso

nulo	No existencia
\triangleright	Comentario

Los operadores aritméticos y lógicos utilizados se muestran en la Tabla [A.2](#).

Tabla 2 Operadores aritméticos y lógicos.

Notación Sentido

\neg	Negacion logica
\wedge	Lógico y
\vee	Lógico o
\oplus	Lógica exclusiva o
a / b	a dividido por b
una . b o ab	Multiplicación

Usamos conjuntos en lugar de matrices para mostrar una colección de variables. Un elemento x está contenido en un conjunto A es realizado por la unión (\cup) operador de la siguiente manera:

$$A \leftarrow A \cup \{x\}$$

y la eliminación de un elemento y del conjunto A se realiza utilizando el setminus (\setminus) operador de la siguiente manera:

$$A \leftarrow A \setminus \{v\}$$

La tabla [A.3](#) muestra las operaciones de conjunto utilizadas en el texto con sus significados.

Tabla 3 operaciones de ajuste

Notación	Sentido
$ S $	Cardinalidad de s
\emptyset	Conjunto vacio
$u \in S$	eres miembro de S
$S \cup R$	Unión de S y R
$S \cap R$	Intersección de S y R
$S \setminus R$	Establecer resta
$S \subseteq R$	S es un subconjunto de R
$S \subset R$	S es un subconjunto adecuado de R
máximo minimo $\{\dots\}$	Valor máximo / mínimo de un conjunto de valores

A.3 Estructuras de control

Las estructuras de control se utilizan para alterar el flujo de ejecución. La selección y la repetición son dos modos principales de control, tal como se describe a continuación.

Selección

La selección de uno de los pocos flujos alternativos se realiza comúnmente mediante la construcción if-then-else . La expresión booleana después de que el si la declaración se evalúa y después de la rama y luego se toma si esta expresión se obtiene

un cierto valor. Podemos especificar un bloque else para especificar el flujo alternativo cuando la expresión produce un valor falso . Se muestra un ejemplo en el algoritmo A.7 donde queremos probar cuál de los dos enteros dados a y b es mayor que el otro o si son iguales entre sí. Vemos que la línea 7 se ejecuta en este ejemplo.

Algorithm A.1 if-then-else structure

```

1: a ← 3; b ← 5
2: if a > b then
3:   output "a is greater"
4: else if a = b then
5:   output "a = b"
6: else
7:   output "b is greater"
8: end if

```

i= test condition
▷ else if of if statement
▷ end of if statement

Podemos seleccionar un flujo específico de ejecución de una serie de flujos alternativos nuevamente mediante la evaluación de una cantidad de expresiones usando la estructura de caso de caso . En el algoritmo A.3 se muestra una calculadora simple que puede realizar sumas, restas, multiplicaciones y divisiones.

Algorithm A.2 A Simple Calculator

```

1: input operator
2: input a and b
3: case operator of
4:   "+": c ← a + b
5:   "-": c ← a - b
6:   "*": c ← a * b
7: default c ← a/b
8: output c

```

Repetición

Usamos las construcciones de bucle para , while , y repetimos ... hasta que implementamos una declaración varias veces. El bucle for-do se usa comúnmente cuando el número de iteraciones se conoce de antemano. El ejemplo que se muestra en el algoritmo A.3 encuentra la suma de los elementos de una matriz con elementos enteros.

Algorithm A.3 Sum of a Matrix

```

1: Input: int A[n] = [...]
2: Output: sum of the elements of A
3: for i = 1 to n do
4:   xsum ← xsum + A[i]
5: end for

```

Cuando se trata de conjuntos y no sabemos el tamaño del conjunto, se puede utilizar convenientemente el bucle para todos . Comúnmente, seleccionamos arbitrariamente un elemento del conjunto y realizamos una operación en este elemento como se muestra en el Algoritmo A.4, donde simplemente generamos cada elemento del conjunto S, que consiste en números enteros.

Algorithm A.4 Output Elements of a Set

```

1: Input: set of int S = [...]
2: Output: elements of S
3: for all u ∈ S do
4:   output u
5: end for

```

Hay casos en los que queremos introducir un bucle basado en una condición. El tiempo de bucle se puede utilizar para tales implementaciones, y este tipo de bucle se puede introducir 0 o más veces sobre la base de la evaluación de una expresión booleana, como se muestra en el algoritmo A.5 donde se introduce la suma de los números introducidos se calcula hasta 99. Tenga en cuenta que 99 puede ingresarse como la primera entrada sin causar la ejecución del bloque dentro del bucle.

Algorithm A.5 Sum of Integers

```

1: Input: set of int  $S$ 
2: Output: elements of  $S$ 
3: input  $a$ 
4: while  $a \neq 99$  do
5:    $sum \leftarrow sum + a$ 
6:   input  $a$ 
7: end while

```

Nuestra última estructura de bucle que usamos en los algoritmos es la repetición. Hasta el bucle, donde la decisión de ejecutar el bucle se toma después de ejecutarse el bucle. Este tipo de bucle se usa cuando sabemos que el bucle debe ejecutarse al menos una vez, como se muestra en el algoritmo A.6, donde implementamos el ejemplo anterior de agregar números ingresados. Tenga en cuenta que no necesitamos la instrucción de entrada antes del bucle esta vez, ya que sabemos que el bucle se ejecutará al menos una vez.

Algorithm A.6 Sum of Integers

```

1: Input: set of int  $S$ 
2: Output: elements of  $S$ 
3: input  $a$ 
4: repeat
5:   input  $a$ 
6:    $sum \leftarrow sum + a$ 
7: until  $a \neq 99$ 

```

A.4 Estructura de algoritmo distribuido

Un nodo de una red ejecuta un algoritmo distribuido, y la acción a realizar se decide generalmente por el tipo y el contenido del mensaje recibido. Tenemos una estructura de casos después de que se recibe el mensaje y luego cada acción necesaria se decide por el tipo y luego el contenido del mensaje recibido. El código es ejecutado por el nodo i y frecuentemente omitimos escribir la identidad del nodo i en varias operaciones realizadas por simplicidad. Por ejemplo, enviar msg_i a j significa envío de mensaje msg_{ij} del nodo i al nodo j . Necesitamos verificar la recepción de la condición del mensaje, por ejemplo, cuando se reciben mensajes de todos los vecinos o solo se recibió un mensaje del padre en un árbol, etc. En algoritmos distribuidos síncronos, necesitamos una variable booleana para mostrarnos que la ronda es una y comúnmente, tenemos un tiempo de bucle que comprueba esta condición. Cuando esta condición se cumple, no esperamos más mensajes como se muestra en el algoritmo A.6.

Algorithm A.7 Distributed Algorithm Structure I

```

1: int  $i, j$                                  $\triangleright i$  is this node;  $j$  is the sender of a message to  $i$ 
2: while  $\neg flag$  do                       $\triangleright$  all nodes execute the same code
3:   receive  $msg_{ij}$ 
4:   case  $msg_{ij}.type$  of
5:      $\frac{a_1}{...} : action_1$ 
6:      $\dots : ...$ 
7:      $\frac{a_n}{...} : action_n$ 
8:     if messages received from all neighbors then
9:        $flag \leftarrow true$ 
10:    end if
11: end while

```

Referencias

Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introducción a los algoritmos. MIT Press, Cambridge

- 1.
2. Erciyes K (2013) Algoritmos de grafos distribuidos para redes informáticas. Springer, Berlín

apéndice B

Revisión de álgebra lineal

B.1 Introducción

Un gráfico puede representarse por su matriz de adyacencia o matriz de incidencia. La matriz laplaciana de un gráfico proporciona información sobre las propiedades espectrales de un gráfico. La teoría del gráfico algebraico se basa en la aplicación de métodos algebraicos para graficar problemas y, comúnmente, las matrices asociadas con un gráfico se usan para este propósito. El álgebra lineal es una rama de las matemáticas que trata con matrices. Proporcionamos una revisión muy breve y parcial del álgebra lineal suficiente como fondo de las propiedades del gráfico espectral y los algoritmos de gráfico algebraico descritos en el libro.

B.2 Tipos de matrices básicas

Una matriz es un conjunto de elementos organizados en filas y columnas.

- Una matriz general con m filas y n columnas se puede escribir como

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

- Transposición de una matriz : esta matriz se obtiene escribiendo filas como columnas y columnas como filas. La transposición de la matriz anterior es

$$A^T_{n,m} = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,m} & a_{2,m} & \cdots & a_{n,m} \end{pmatrix}$$

- Matriz diagonal : todas las entradas excepto los valores diagonales de esta matriz son 0. Para una 4×4 matriz

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 \\ 0 & a_{2,2} & 0 & 0 \\ 0 & 0 & a_{3,3} & 0 \\ 0 & 0 & 0 & a_{4,4} \end{pmatrix}$$

- Matriz de identidad : la matriz de identidad I es una matriz diagonal con todos los valores diagonales de unidad. I_4 se muestra a continuación:

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Matriz simétrica : los valores simétricos a la diagonal son iguales en esta matriz, lo que significa que esta matriz es igual a su transposición.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{1,2} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{1,3} & a_{2,3} & a_{3,3} & a_{3,4} \\ a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} \end{pmatrix}$$

- Matriz triangular superior : la matriz A es triangular superior si $a_{ij} = 0$ cuando $i > j$ Como se muestra abajo:

$$A = \begin{pmatrix} 3 & 1 & 5 \\ 0 & 7 & 1 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix}$$

- Matriz triangular inferior : la matriz A es triangular inferior si $a_{ij} = 0$ cuando $i < j$.

$$A = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 \\ 3 & 6 & 4 & 0 \end{pmatrix}$$

- Submatriz : una submatriz de una matriz se forma al eliminar un conjunto de filas y / o un conjunto de columnas. Eliminar la fila 1 y la columna 2 de una matriz da como resultado la submatriz mostrada.

$$A = \begin{pmatrix} 2 & 1 & 0 & 4 \\ 5 & 3 & 2 & 1 \\ 0 & 2 & 4 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 5 & 2 & 1 \\ 0 & 4 & 3 \end{pmatrix}$$

- Vector: un vector es un $n \times 1$ matriz.

B.3 Operaciones matriciales

- Adición : se pueden agregar dos matrices si tienen la misma dimensión. Los elementos correspondientes de las matrices se pueden agregar para formar la matriz de suma. La suma de dos 3×3 Las matrices A y B para obtener la matriz C son las siguientes:

$$C = A + B$$

$$\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} + \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

$$= \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & a_{1,3} + b_{1,3} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & a_{2,3} + b_{2,3} \\ a_{3,1} + b_{3,1} & a_{3,2} + b_{3,2} & a_{3,3} + b_{3,3} \end{pmatrix}$$

- Multiplicación con un escalar : se pueden agregar dos matrices si tienen la misma dimensión. Los elementos correspondientes de las matrices se pueden agregar para formar la matriz de suma. La suma de dos 3×3 Las matrices A y B para obtener la matriz C son las siguientes:

$$C = k \cdot A$$

$$C = k \cdot \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = \begin{pmatrix} k \cdot a_{1,1} & k \cdot a_{1,2} & k \cdot a_{1,3} \\ k \cdot a_{2,1} & k \cdot a_{2,2} & k \cdot a_{2,3} \\ k \cdot a_{3,1} & k \cdot a_{3,2} & k \cdot a_{3,3} \end{pmatrix}$$

- Multiplicación de matrices : la multiplicación de una matriz por la matriz de identidad no la cambia $AI = A$

$$C = A \times B$$

$$= \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix}$$

$$= \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}$$

Por ejemplo,

$$\begin{pmatrix} 5 & 4 \\ -3 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} \times \begin{pmatrix} 3 & 2 \\ 1 & 1 \end{pmatrix}$$

Multiplicando un $m \times n$ matriz por la $n \times n$ matriz de identidad I_n no lo cambia

$$AI_n = A = I_m A$$

Por un $n \times n$ matriz A , si existe una $n \times n$ matriz B tal que

$$AB = A = BA = I$$

Entonces, la matriz B se llama la inversa de A escrita como A^{-1} , y A se llama matriz no singular . Cuando dicha matriz inversa no se puede determinar, la matriz A se llama singular . Una matriz cuadrada es singular si y solo si su determinante es 0.

Determinante de una matriz

El determinante de una matriz cuadrada A , mostrado por $\det(A)$ o $|A|$, Se utiliza para diversas operaciones, incluyendo en la búsqueda de la inversa de la matriz A . Determinante de un 2×2 La matriz A se calcula de la siguiente manera:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Determinante de un 3×3 la matriz A se puede encontrar seleccionando una fila o una columna y multiplicando cada elemento de la fila / columna seleccionada por el determinante del subgrafo obtenido al eliminar la fila / columna y columna de ese elemento de la matriz como se muestra a continuación:

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei - bfg + cdh - ceg - bdi - afh$$

El menor M_{ij} de un elemento a_{ij} de una matriz cuadrada A es el determinante de la submatriz obtenida mediante la supresión de i th fila y j ésimocolumna de A . El cofactor de a_{ij} , C_{ij} , se obtiene multiplicando su menor por $(-1)^{i+j}$. En el ejemplo anterior, $\det(A)$ se calcula como la suma de los cofactores multiplicado por elementos de fila de la primera fila de A . El inverso de una matriz se puede calcular utilizando varios métodos. Cuando la matriz A no es grande, su inverso se puede determinar de la siguiente manera:

$$A^{-1} = \frac{1}{\det(A)} C^T$$

donde C es la matriz cofactor de A . Para 2×2 matriz A , su inverso se calcula de la siguiente manera:

$$A^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

B.4 Propiedades de las operaciones de matriz

Las siguientes propiedades de las operaciones matriciales son válidas suponiendo que las matrices son de tamaños apropiados:

- $A + B = B + A$
- $A(B + C) = AB + AC$
- $(A^T)^T = A$
- $(A + B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$
- $(AB)^{-1} = B^{-1}A^{-1}$
- Para un escalar k , $k(A + B) = kA + kB$

B.5 Ecuaciones lineales

Un sistema de ecuaciones lineales de la forma que se muestra a continuación se puede resolver utilizando operaciones matriciales.

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n} = b_1$$

...

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn} = b_m$$

Deja que A sea un $m \times n$ matriz de coeficientes, y x es una $n \times 1$ vector que representa m variables. Podemos escribir estas ecuaciones en forma de matriz de la siguiente manera:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

El vector de solución x es entonces la solución a la ecuación,

$$x = A^{-1}b$$

Consideremos la siguiente ecuación lineal con dos variables x_1 y x_2 :

$$2x_1 + 3x_2 = 4$$

$$x_1 + 2x_2 = 3$$

Podemos escribir esta ecuación usando la notación matricial como $Ax = b$ de la siguiente manera:

$$\begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \end{pmatrix}$$

Ahora podemos calcular A^{-1} y entonces $x = A^{-1}b$ como a continuación:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 & -3 \\ -1 & 2 \end{pmatrix} \begin{pmatrix} 4 \\ 3 \end{pmatrix} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

rendir valores $x_1 = -1$ y $x_2 = 2$.

Eliminación gaussiana

Otro método para resolver un sistema de ecuaciones lineales es primero formar la ecuación matricial $Ax = b$ como antes. Luego formamos la ecuación de matriz aumentada de la siguiente manera:

$$A_{m,n}^G = \left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & |b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & |b_2 \\ \vdots & \vdots & \ddots & \vdots & | \cdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} & |b_m \end{array} \right)$$

A continuación, la matriz aumentada. A^G Se transforma en una matriz triangular superior. A^U utilizando operaciones de fila elementales. Entonces resolvemos para x_m y luego usar el valor de x_m para obtener el valor de x_{m-1} , etc., utilizando la sustitución hacia atrás. Consideremos el siguiente sistema de ecuaciones con tres variables x_1 , x_2 y x_3 :

$$x_1 + x_2 - x_3 = -2$$

$$3x_1 - 2x_2 + x_3 = 7$$

$$2x_1 - x_2 - 3x_3 = 9$$

La matriz aumentada es entonces como sigue:

$$A_{3,3}^G = \begin{pmatrix} 1 & 1 & -1 & | & -2 \\ 3 & -2 & 1 & | & 7 \\ 2 & -1 & 3 & | & 9 \end{pmatrix}$$

Multiplicando la primera fila por -2 y agregándola a la tercera fila; luego, al multiplicar la primera fila por -3 y agregarla a la segunda fila, se obtiene la primera matriz a continuación. La última matriz triangular superior obtenida al multiplicar la segunda fila por $-3/5$ y agregarla a la tercera fila es la siguiente:

$$\begin{pmatrix} 1 & 1 & -1 & | & -2 \\ 3 & -2 & 1 & | & 7 \\ 0 & -3 & 5 & | & 13 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & -1 & | & -2 \\ 0 & -5 & 4 & | & 13 \\ 0 & -3 & 5 & | & 13 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & -1 & | & -2 \\ 3 & -2 & 1 & | & 7 \\ 0 & 0 & 13/5 & | & 26/5 \end{pmatrix}$$

Ahora podemos determinar $x_3 = 2$ evaluando la última fila. Sustitución hacia atrás de x_2 en la segunda fila proporciona $x_2 = -1$, y luego sustituyendo los valores de x_1 y x_2 en la primera fila los rendimientos $x_1 = 2$.

Índice

A

Algoritmo de grafo algebraico

BFS

conectividad

pareo

 Algoritmo Rabin-Vazirani

matrices

árbol de expansión mínima

trayectoria más corta

Algoritmo

 algebraico

 aproximación

 análisis asintótico

 clase de complejidad

 notario público

 NP-completo

 NP-duro

 PAG

 divide y conquistarás

 programación dinámica

 grafico

 árbol de expansión mínima

 GOLPES

 conjunto dominador mínimo

 árbol de expansión mínima

 NP-completo

 prueba

 contradicción

 contrapositivo

 directo

 inducción

 bucle invariante

 fuerte inducción

 aleatorizado

 Algoritmo de Karger

 recursivo

 reducciones

 estructuras

Camino alternativo

Algoritmo de aproximacion

Punto de articulación
Algoritmo de subasta
Algoritmo basado en subastas
 paralela
Camino de aumento
 segundo
Retroceso
Batagelj y el algoritmo Zaversnik
Algoritmo de Bellman-Ford
Teorema de Berge
BFS
Gráfico biconectado
Red biologica
Cubierta bipartita vértice
Emparejamiento ponderado bipartito
Bloquear
 descomposición
 Algoritmo Hopcroft – Tarjan
Algoritmo de Boruvka
Rama y límite
Puente
 Algoritmo de puente de Tarjan
Algoritmo de Bron y Kerbosch
do
Centralidad
 centralidad de proximidad
 grado de centralidad
 borde centralidad
 centralidad de valores propios
 centralidad entre vértice
Camarilla
 Algoritmo de Bron y Kerbosch
Centralidad de proximidad
Agrupamiento
 Basado en MST
 red de sensores
 espectral
Coeficiente de agrupamiento
Colorante
Clase de complejidad
 notario público
 NP-completo
 NP-duro
 PAG
Red compleja

- red biologica
- agrupación
- Internet
 - enrutamiento
 - alineación de red
 - motivo de red
 - red de sensores
 - agrupación
 - conectividad
 - red social
 - web
 - Algoritmo HITS
 - Algoritmo de PageRank
- Conectividad
 - punto de articulación
 - bloquear
 - puente
 - dígrafo
 - repartido
 - conectividad de borde
 - corte de borde
 - basado en el flujo
 - Teoremas de menger
 - paralela
 - secuencial
 - conectividad fuerte
 - algoritmo
 - clausura transitiva
 - conectividad de vértice
 - corte de vértice
- Contradicción
- Opresivo
- Corte ertex
- Ciclo
- re
- Dígrafo
 - fuertemente conectado
 - componente fuertemente conectado
- Algoritmo de Dijkstra
- Distancia
- Algoritmo distribuido
 - ACI
 - ASI
 - emisión
 - convergecast

- inundación
- elección del líder
- SCI
- árbol de expansión
- SSI

- Programación distribuida

- Sistemas distribuidos

- Divide y conquista los algoritmos.

- Conjunto dominante

- algoritmo

- codicioso

- Algoritmo de Guha-Khuller

- Basado en MIS

- Algoritmos de gráficos algebraicos dinámicos

- conectividad

- coincidencia perfecta

- Algoritmo gráfico dinámico

- conectividad

- Árbol de la gira de Euler

- pareo

- Algoritmo de Neiman y Solomon

- metodos

- agrupación

- aleatorización

- sparsification

- Programación dinámica

- mi

- Excentricidad

- Borde entre la centralidad

- Conectividad de borde

- Corte del borde

- Bordes desunidos

- Algoritmo de intermediación de borde

- Coloración del borde

- gráfica bipartita

- grafica completa

- repartido

- paralela

- Algoritmo de flor de edmond

- Algoritmo de Edmonds-Karp

- Valor propio

- Centralidad eigenvalor

- Árbol de la gira de Euler

- Incluso: algoritmo de Tarjan

- F

Máquina de estados finitos
Emparejamiento basado en flujo
Algoritmo Floyd – Warshall
Algoritmo Ford – Fulkerson
sol
Grafico
 lista de adyacencia
 matriz de adyacencia
 algoritmo
 repartido
 gráfico grande
 paralela
 secuencial
 bipartito
 producto cartesiano
 completar
 secuencia de grado
 dirigido
 valor propio
 matriz de incidencia
 intersección
 isomorfismo
 Matriz laplaciana
 gráfico grande
 línea
 regular
 subgrafo
 tipos
 Unión
 grado de vértice
 ponderado
Algoritmo gráfico
 dinámica
Matrices graficas
 proximidad
 incidencia
 Laplaciano
Traversal del gráfico
 búsqueda de amplitud
 búsqueda en profundidad
Algoritmo de Guha-Khuller
H
Algoritmo HITS
Algoritmo Hopcroft – Karp
Algoritmo Hopcroft – Tarjan

Método húngaro

yo

Conjunto independiente

Algoritmo de luby

Inducción

Internet

enrutamiento

jerárquico

estado de enlace

K

Teorema de König

k -conectividad

Algoritmo SCC de Kosaraju

Algoritmo de Kruskal

Algoritmo de Kuhn-Munkres

L

Gráfico grande

k -cores

análisis

Batagelj y el algoritmo Zaversnik

centralidad

centralidad de proximidad

grado de centralidad

centralidad de valores propios

camarilla

camarillas

Algoritmo de Bron y Kerbosch

agrupación

distribución de títulos

densidad

índice coincidente

modelos de red

redes aleatorias

redes sin escala

mundo pequeño

Gráficos grandes

centralidad

borde centralidad

centralidad entre vértice

Bucle invariante

Algoritmo de luby

METRO

Manet

Pareo

- algebraico
- Algoritmo Rabin-Vazirani
- no ponderado
 - bipartito
 - repartido
- Algoritmo de flor de edmond
 - basado en el flujo
- Algoritmo Hopcroft – Karp
- ponderado
 - Algoritmo de subasta
 - bipartito
 - Método húngaro
 - Algoritmo de Kuhn-Munkres
- Índice de coincidencia
- Conjunto independiente maximal
- Teoremas de menger
- Árbol de expansión mínima
 - algebraico
 - Algoritmo de Boruvka
 - agrupación
 - Algoritmo de Kruskal
 - paralela
 - Algoritmo de prim
 - algoritmo de borrado inverso
- Algoritmo basado en la modularidad
- Algoritmo Mucha – Sankowski
- norte
- Algoritmo de Neiman y Solomon
- Flujo de red
 - cortar
 - Algoritmo de Edmonds-Karp
 - Algoritmo Ford – Fulkerson
 - red residual
- Modelos de red
 - redes aleatorias
 - redes sin escala
 - redes del mundo pequeño
- Motivo de red
- NP-completo
- NP-duro
 - satisfacibilidad
- PAG
- Algoritmo de PageRank
- Algoritmo paralelo
 - paralelismo de datos

diseño
paralelismo funcional
grafico

- descomposición del oído
- contracción gráfica
- partición gráfica
- puntero saltando
- aleatorización

Procesamiento en paralelo

- análisis
- arquitecturas
- comunicación
- modelos
- paso de mensajes
- COCHECITO

asignación de procesador

- dinámica
- estático

Programación paralela

- MPI
- trapos

Algoritmo de Parnas-Ron

Camino

Juego perfecto

Algoritmo de precios

Algoritmo de prim

Asignación de procesador

- estático

Prueba

- contradicción
- contrapositivo
- directo
- inducción
- bucle invariante
- fuerte inducción

R

Algoritmo Rabin-Vazirani

Redes aleatorias

Aleatorización

Algoritmos aleatorios

- Algoritmo de Karger

Reaparición

Reducciones

Red residual

Eliminación de borde inverso

Enrutamiento
jerárquico
estado de enlace

S

Satisfiabilidad
Redes sin escala
Red de sensores

agrupación
conectividad

Trayectoria más corta

Algoritmo de Bellman-Ford
Algoritmo de Dijkstra
Algoritmo Floyd – Warshall
única fuente

Redes del mundo pequeño

Red social
detección de la comunidad
filo
modularidad
equivalencia
relación

Conectividad fuerte

Fuerte inducción

Componente fuertemente conectado

Algoritmo SCC de Kosaraju
paralela
Algoritmo SCC de Tarjan

Digraph fuertemente conectado

Diferencia simétrica

T

Algoritmo de puente de Tarjan

Algoritmo SCC de Tarjan

Trapos

POSIX

Sendero

Clausura transitiva

Árbol

binario
puntero saltando
árbol de expansión
atravesar
en orden
orden de publicación
hacer un pedido

V

Centralidad entre vértice
Colorante vértice
Algoritmo Cole-Vishkin
algoritmos distribuidos
Algoritmo de Jones-Plassman
coloración de árboles
Conecividad vértice
Incluso: algoritmo de Tarjan
Cubierta vértice
Vértices caminos desunidos
Conecividad vértice
Corte de vértice
Cubierta vértice
no ponderado
bipartito
ponderado
Algoritmo de precios

W

Caminar
El algoritmo de Warshall
Web
Algoritmo HITS
Algoritmo de PageRank
Grafico ponderado
árbol de expansión mínima
Algoritmo de Boruvka
Algoritmo de Kruskal
paralela
Algoritmo de prim
algoritmo de borrado inverso
trayectoria más corta
todos los pares
Algoritmo de Bellman-Ford
Algoritmo de Dijkstra
Algoritmo Floyd – Warshall
única fuente

WSN

