Repaso I3

Grafos, grafos y más grafos

Carlos Paredes - cparedesr@uc.cl

1. Búsqueda en grafos

- Lo más importante en grafos para recordar es DFS y BFS. Luego ver como se relaciona con los otros algoritmos O(V+E =) n + n-1 = O(n)

- Para la vida (y DAA). Preferir las implementaciones iterativas stack-based

```
bfs(graph G, node start)
      Queue q
      q.push(start)
      mark start as visited
      while not q.empty()
            u = q.pop()
            for v in G.adjacent[u]
                  if v not visited
                        mark v as visited
                        q.push(v)
            mark u as Done
```

1. Búsqueda en grafos

- Sea **T** un ***grafo conexo no dirigido sin ciclos***, se dice que T es un árbol. El diámetro del árbol corresponde al largo máximo de un camino en el grafo. El problema es calcular el diámetro de un árbol.

1. Búsqueda en grafos

- Sea **T** un ***grafo conexo no dirigido sin ciclos***, se dice que T es un árbol. El diámetro del árbol corresponde al largo máximo de un camino en el grafo. El problema es calcular el diámetro de un árbol.

- **Idea: encontrar el camino más largo en T usando BFS.**

1. Búsqueda en grafos

Notar que como T es un árbol, para cada par de nodos ***u, v*** existe un único camino que los une, más aún cada camino en T corresponde a un par de nodos ***u, v***.

Notar además que dado un camino de largo máximo entre dos nodos ***u, v*** y un tercer nodo ***w***, se tiene que ***u*** está a distancia máxima de ***w*** o que ***v*** está a distancia máxima de ***w***.

1. Búsqueda en grafos

Solución:

- Correr 2 BFS, uno desde cualquier nodo, y otro desde el último nodo visitado por el BFS anterior.

- El diámetro del árbol es la distancia entre el último nodo del primer BFS y el último nodo del segundo BFS.

1. Búsqueda en grafos

Solución:

- Correr 2 BFS, uno desde cualquier nodo, y otro desde el último nodo visitado por el BFS anterior.
- El diámetro del árbol es la distancia entre el último nodo del primer BFS y el último nodo del segundo BFS.

Ejercicio: ¿Cuál es la complejidad de este algoritmo?

2. Algoritmos asociados a DFS/BFS

- **Topological Sort 65% -> Transponer grafo, y luego TopSort con BFS**
- **Strongly connected components (SCC) 35% -> Kosaraju, Tarjan**
- Todaaas las derivaciones...

2. Algoritmos asociados a DFS/BFS

Después de un estresante semestre, Batman decidió que ya no quiere seguir estudiando computación, y que lo suyo es la Ingeniería Comercial, ya que ahí sí que están las lucas.

Estuvo revisando la malla, con sus cursos y requisitos, y quiere saber en cuántos semestres como mínimo podría terminar la carrera. Batman es un genio, así que no hay límite a la cantidad de cursos que puede hacer en un mismo semestre.

2. Algoritmos asociados a DFS/BFS

Diseña un algoritmo ***O(|cursos| + |requisitos|)*** que permita resolver este problema, entregando la lista de cursos que debe tomar en cada semestre. Justifica su complejidad. Puedes suponer que todos los cursos se dictan todos los semestres.

2. Algoritmos asociados a DFS/BFS

Diseña un algoritmo ***O(|cursos| + |requisitos|)*** que permita resolver este problema, entregando la lista de cursos que debe tomar en cada semestre. Justifica su complejidad. Puedes suponer que todos los cursos se dictan todos los semestres.

Idea: Usar grafo de requisitos y topological sort

2. Algoritmos asociados a DFS/BFS

Definimos el grafo ***G(V, E)*** de la malla de la siguiente forma:

- Cada vértice v ∈ V corresponde a un curso distinto
- Si el curso a tiene como requisito al curso b, entonces existe una arista ***(a, b) ∈ E***

2. Algoritmos asociados a DFS/BFS

Definimos el grafo ***G(V, E)*** de la malla de la siguiente forma:

- Cada vértice v ∈ V corresponde a un curso distinto
- Si el curso a tiene como requisito al curso b, entonces existe una arista ***(b, a) ∈ E***

Esto se parece mucho a los ejemplos de orden topológico vistos en clases. El problema es que el algoritmo de orden topológico no tiene noción de tareas que pueden realizarse simultáneamente, por lo que no podemos usarlo directamente.

2. Algoritmos asociados a DFS/BFS

Idea: Usar BFS y coloración

2. Algoritmos asociados a DFS/BFS

Recorrer el grafo con DFS/BFS a partir de los cursos del primer semestre, entrando a un curso (vértice) solo una vez que ya se entró a todos sus requisitos (ancestros). ***El semestre de un curso será 1 más que el semestre de su requisito con el mayor semestre***.

```
mallaPorSemestres(G(V, E)):
    for v ∈ V:
        v.incoming ← 0
        v.semestre ← 0
    for (u, v) ∈ E:
        v.incoming ← v.incoming + 1
    Open ← lista vacía
    for v ∈ V:
        if v.incoming = 0:
            Agregar v a Open
            v.semestre ← 1
    while Open ≠ ∅:
        Extraer un vértice x cualquiera de Open
        foreach arista (x, v) que sale de x:
            v.incoming ← v.incoming − 1
            if v.incoming = 0:
                Insertar v en Open
            if x.semestre + 1 > v.semestre:
                v.semestre ← x.semestre + 1
```

3. Greedy

- **Djikstra**
- **Minimum Spanning Tree (Prim, Kruskal)**

```
dijkstra(graph G, node start)
     priority_queue q
     q.push((0, start))
     distance[start] = inf
     while not q.empty()
          d,u = q.pop()
          for v,w in G.adjacent[u]
               if distance[v]>d+w
                    distance[v]=d+w
                    q.push((distance[v],v))
```

3. Greedy (Problema)

- Hay un juego donde el héroe quiere llegar al último nivel con el menor costo, para ganar cada nivel ***i*** el héroe debe gastar una cantidad de energía E_i, y en algunos niveles hay unas tiendas, cada tienda ***t*** fija la energía de nuestro héroe a E_t por un costo C_t. El héroe quiere tu ayuda para lograr esto.

3. Greedy (Problema)

- Hay un juego donde el héroe quiere llegar al último nivel con el menor costo, para ganar cada nivel ***i*** el héroe debe gastar una cantidad de energía E_i, y en algunos niveles hay unas tiendas, cada tienda ***t*** fija la energía de nuestro héroe a E_t por un costo C_t. El héroe quiere tu ayuda para lograr esto.

Idea: Usar Djikstra

3. Greedy (Problema)

- Notemos que si ponemos que cada nivel es un nodo tenemos un grafo con pesos, donde cada arista corresponde a una tupla ***(u,v,w)*** donde en el nivel ***u*** hay una tienda con costo ***w*** tal que fija suficiente energía para llegar a ***v***.
- Correr Dijsktra en el grafo descrito anteriormente comenzando en el nivel “0” para llegar al último nivel.

3. Greedy (Problema 2)

```
prim(Grafo G)
     set Q //de nodos ordenados por C[u]
     //C[u] es infty en un comienzo
     forest F
     while not Q.empty()
          u = min(Q)
          F.add(u)
          if E[u]≠null
               F.add(E[u])
          for w,v in G.adjacent[u]
               C[v] = w
               E[v] = (u,v)
```

```
kruskal(Grafo G)
     UnionFind T
     E = G.edges
     sort(E) //Por el peso
     for w,u,v in E
          if not T.same_set(u,v)
               T.join(u,v)
               //añadir arista u,v
```

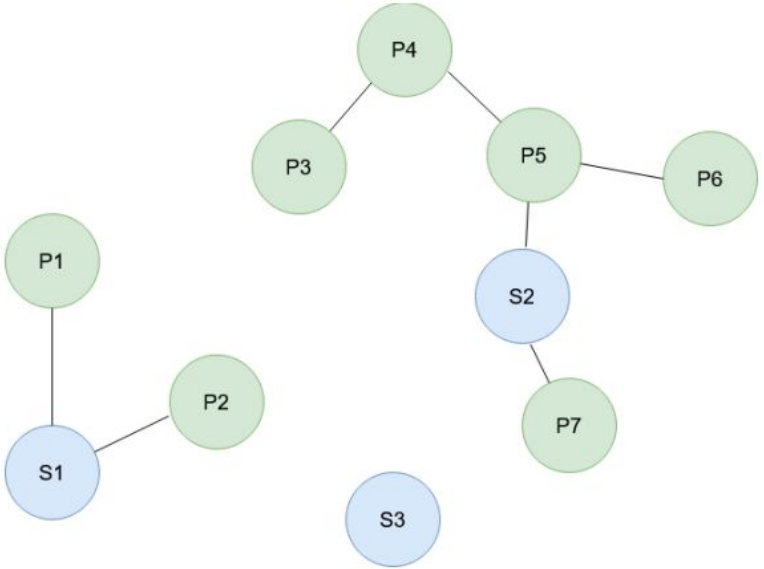
3. Greedy (Problema 2)

- Luego de una traumática experiencia en Ingeniería Comercial, Batman decidió renunciar y dedicarse a su verdadera pasión: la jardinería. Planificando el sistema de riego para los jardines del tacaño duque Rico McPato, Batman se encontró con un problema de optimización que le trajo recuerdos:

Dada una serie de puntos P que deben ser regados, y una serie de puntos S de tomas de agua, se debe disponer de cañerías entre puntos de manera que desde cada punto $p \in P$ exista una **única** ruta mediante cañerías a algún punto $s \in S$. Considerando que el costo de una cañería es proporcional a su largo, se quiere resolver este problema **minimizando** el costo total de las cañerías dispuestas.

3. Greedy (Problema 2)

Considera el siguiente ejemplo de una solución:



3. Greedy (Problema 2)

Dado S, P y el grafo no dirigido y con costos $G(V, E)$, con $V = S \cup P$ y $E = V \times V$; es decir, un grafo completo. Diseña un algoritmo que resuelva este problema en tiempo $O(E \cdot \log V)$. Dicho algoritmo puede ser en prosa o en pseudocódigo: evita lenguajes de programación.

3. Greedy (Problema 2)

Lo que se pide es, básicamente, encontrar un *bosque de cobertura mínimo*. Es decir, un conjunto de $|S|$ árboles que cubren el grafo, tal que cada árbol contiene exactamente un nodo de S, y el costo total de las aristas de todos los árboles es mínimo.

Este problema se puede resolver de varias maneras, y son todas equivalentes:

Opción 1

Conectamos todos los nodos de S a un nuevo nodo $s^\star$ mediante aristas de costo 0, y luego aplicamos el algoritmo de Prim o Kruskal tal como se vieron en clases.

Teniendo el MST del grafo, eliminamos $s^\star$ y las aristas que lo conectan con los nodos de S para separar el árbol en sub-árboles, formando así el bosque que se pide.

Agregar y quitar este nodo y sus aristas no agrega complejidad a los algoritmos, por lo que están dentro de la complejidad esperada.

3. Greedy (Problema 2)

Opción 2

En lugar de agregar un nodo extra y aristas de costo cero, podemos modificar uno de los algoritmos para poder manejar este caso:

Opción 2.1: Prim

El algoritmo de Prim parte poniendo de un nodo "de partida" en el heap y genera el árbol a partir de ahi, marcando cuales nodos ya fueron "incluidos". Como para este problema debemos generar un árbol por cada nodo de S, simplemente hay que partir desde todos al mismo tiempo y construir los árboles del bosque mínimo de manera simultánea. Para eso basta con partir poniendo **todos** los nodos de S en el heap con prioridad mínima, para que todos partan siendo incluidos.

Estos nodos eventualmente iban a ser agregados al heap: forzar que se agreguen al comienzo no modifica la complejidad del algoritmo.

Opción 2.2: Kruskal

El algoritmo de Kruskal parte diciendo que cada nodo del grafo es su propio conjunto, y va agregando aristas hasta que todos los nodos son parte del mismo conjunto. Sabemos que los nodos de S no pueden pertenecer al mismo sub-arbol, asique sus conjuntos jamás deberian unirse. Para esto, al principio del algoritmo, podemos unir todos los nodos de S en un mismo conjunto: así jamás se eligiran aristas que los conecten.

Estos nodos eventualmente iban a ser unidos al conjunto final, forzar su union al comienzo no modifica la complejidad del algoritmo.

4. DP

- Se acercan las fiestas, y el generoso Krampus te presenta ***n*** regalos, de los cuales debes escoger ***k***. Cada regalo trae la etiqueta con su precio, y como eres una persona materialista, ***quieres maximizar la suma de los precios de los regalos que elijas.***
- Los regalos están dispuestos en la forma de un árbol binario de navidad (no de búsqueda), donde cada nodo corresponde a un regalo. Hay una sola restricción para los regalos que puedes escoger: ***si escoges el regalo u, necesariamente debes escoger el padre de u en el árbol, y así sucesivamente.***

4. DP

- Escribe un algoritmo de programacion dinamica que indique los k regalos que debes escoger de manera de maximizar el precio total

Idea: Usar una tabla D[i, k], (i = nodo escogido, k= regalos restantes). Almacena el precio hasta ese momento

Idea 2: Generar la tabla usando DFS

https://youtu.be/re9rF9SqRFc -> Recomendacion

Bellman Ford

```
BellmanFord(graph G, node start)
     for u in G
          dist[u] = infty
     dist[s] = 0
     repeat |V| - 1
          for u,v,w in G.edges
               dist[u] = min(dist[u],dist[v]+w)
     cycle = false
     for u,v,w in G.edges
          if dist[u] > dist[v] + w
               cycle = true
     return dist, cycle
```

Bellman Ford (Correctitud)

Lema: Después de i repeticiones del loop se tiene que si dist[u] es el largo de un camino entre s y u, más aún, se tiene que si existe un camino entre s y u con a lo más i aristas entonces dist[u] es a lo más el largo del camino más corto entre s y u con a lo más i aristas.

Bellman Ford (DP)

Se puede refrasear el algoritmo en términos de la siguiente recursión:

DP[u][i]=min(DP[v][i-1]+w)

donde el mínimo es sobre todas las aristas de la forma (v,u,w), y donde DP[u][i] representa el largo del camino más corto con a lo más i aristas entre s y u.

Wormholes

En el futuro lejano un historiador quiere ver si puede usar agujeros de gusano para volver arbitrariamente al pasado, se consigue un mapa entre agujeros de gusano donde muestra cuanto se demora en llegar a cada uno, y muestra cómo están conectados los agujeros de gusano con cuanto tiempo pasa desde que se entra al agujero y se sale por el otro lado (posiblemente tiempo negativo).

Wormholes

Idea: Usar Bellman Ford para el solucionar el problema.

Wormholes

Si modelamos los agujeros de gusano como nodos con el tiempo como el peso de las aristas, tenemos que ver si el historiador puede ir arbitrariamente al pasado si es que hay un ciclo negativo.

Recordamos el lema y vemos que si después de |V|-1 iteraciones aún se puede mejorar entonces estamos repitiendo algún nodo en el camino, por lo que hay un ciclo negativo en el grafo.

Keep It Energized

Hay un juego donde el héroe quiere llegar al último nivel con el menor costo, para ganar cada nivel i el héroe debe gastar una cantidad de energía E_i, y en algunos niveles hay unas tiendas, cada tienda t fija la energía de nuestro héroe a E_t por un costo C_t. El héroe quiere tu ayuda para lograr esto.

Keep It Energized

Vamos a reformular el problema con DP.

DP[i]=min(c_t+min{DP[j]:i<j<=r_t})

Donde el mínimo es sobre las tiendas en el nivel i, con c_t siendo el costo de la tienda y r_t siendo lo más lejos que se puede llegar comprando en la tienda.

4. Tips

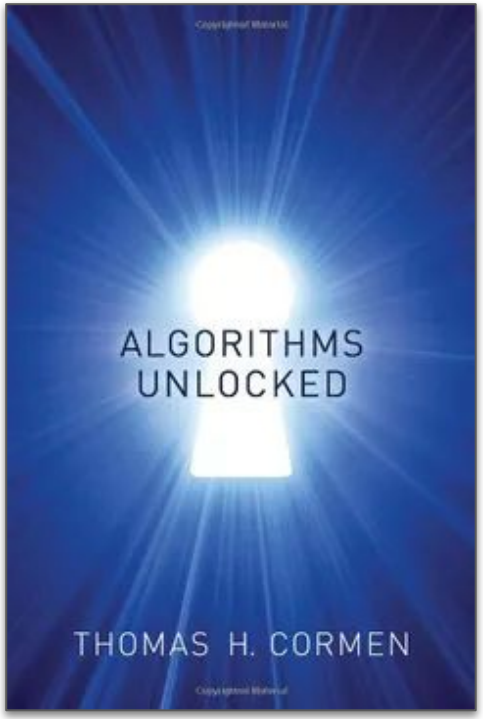
1. EDD es un curso antiguo, por lo que la variedad de preguntas posibles es limitada 👀
2. DFS y BFS vale la pena aprenderlos bien
3. Leer los ejercicios con calma y subrayar las partes clave “n elementos, se busca maximizar k...”
4. Si un algoritmo es dificil de entender, usar approach “Black-Box” para entenderlo de a poco
5. Hacer ejercicios de la guía y revisar compilado. Hacer el resumen con tiempo.

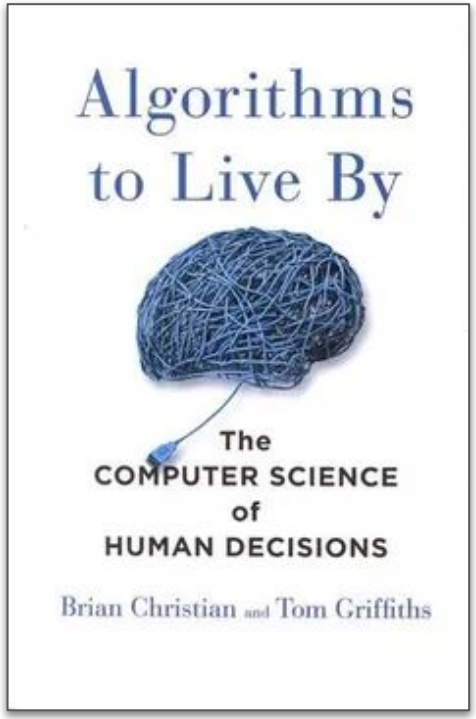
4. Tips Especificos

1. Topological Sort es más común que SCC, aunque SCC es importante de estudiar. En ese caso se recomienda estudiar Kosaraju
2. Kruskal es más simple de definir en pseudocodigo que Prim. Los ejercicios de MST son relativamente “Claros”
3. En Dijkstra puede que el problema no se vea como un grafo, lo importante de esos ejercicios será visualizar el problema como uno.
4. En programación dinámica leer el problema con calma y empezar a separarlo en problemas más pequeños

Floyd warshall: https://www.youtube.com/watch?v=mjJ88I43x4E

4. Recomendaciones asociadas al curso





4. Recomendaciones asociadas al curso

IIC2283 - Diseño y Análisis de Algoritmos

IIC3242 - Complejidad Computacional

MAT2515 - Analisis Real

¡Éxito con la I3!

Repaso I3

Otra invitación (No del curso)

Se encuentran abiertas las postulaciones al trainee de platanus!

A través de nuestro sitio web: platan.us/jobs

Hasta el 11 de julio