TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

BÁO CÁO MÔN HỌC
TÍNH TOÁN HIỆU NĂNG CAO

GRAPH THEORY

Giảng viên hướng dẫn: **TS. Nguyễn Hữu Đức**

Học viên thực hiện: **Nguyễn Đức Thắng - MSHV: 20166769**

Lê Hoàng Ngân - MSHV: 20162886

HÀ NỘI, 09/2020

Mục lục

Lời mở đầu

Lý thuyết đồ thị là một lĩnh vực nghiên cứu đã có từ lâu và có nhiều ứng dụng hiện đại. Những tư tưởng cơ bản của lý thuyết đồ thị được đề xuất vào những năm đầu của thế kỷ 18 bởi nhà toán học người Thụy Sĩ Leonhard Euler. Đồ thị được sử dụng để giải các bài toán trong nhiều lĩnh vực khác nhau.

Khoa học kỹ thuật ngày càng phát triển, đặt ra nhiều bài toán với khối lượng tính toán lớn. Trong số đó có những bài toán mà kết quả chỉ có ý nghĩa nếu được hoàn thành trong khoảng thời gian cho phép. Để giải quyết bài toán này, người ta đã nghiên cứu tăng tốc tố độ tính toán bằng hai phương pháp hoặc kết hợp cả hai:

- Phương pháp 1: Cải tiến công nghệ, tăng tốc độ xử lý của máy tính.
- Phương pháp 2: Chia bài toán ra thành những bài toán nhỏ để có thể chạy song song trên nhiều bộ xử lý

Việc phát triển công nghệ tính toán theo phương pháp 2 đã cho ra đời công nghệ tính toán song song, đó là việc sử dụng đồng thời nhiều tài nguyên tính toán để giải quyết một bài toán. Với mục đích tìm hiểu và nghiên cứu về thuật toán song song, nhóm chọn chủ đề "Thuật toán song song cho một số bài toán về lý thuyết đồ thị" nhằm tìm hiểu, nghiên cứu và tìm giải pháp song song cho một số bài toán về lý thuyết đồ thị trên cơ sở những thuật toán tuần tự truyền thống. Tất cả những thực nghiệm trong báo cáo này được so sánh trên máy MSI GV62 7RD với Ram 8GB và CPU i7 7700HQ.
Bài báo cáo được gồm 3 phần:

1. **Chương 1 :** Cơ sở lý thuyết, trình bày một số khái niệm cơ bản của lý thuyết đồ thị
2. **Chương 2 :** Trình bày thuật toán tuần tự và cách xây dựng thuật toán song song đối với đồ thị dày
3. **Chương 3 :** Trình bày thuật toán tuần tự và cách xây dựng thuật toán song song đối với đồ thị thưa

Để hoàn thành bài báo cáo môn học Tính toán hiệu năng cao, nhóm em xin gửi lời cảm ơn tới thầy giáo TS.Nguyễn Hữu Đức đã trang bị kiến thức, cùng những nhận

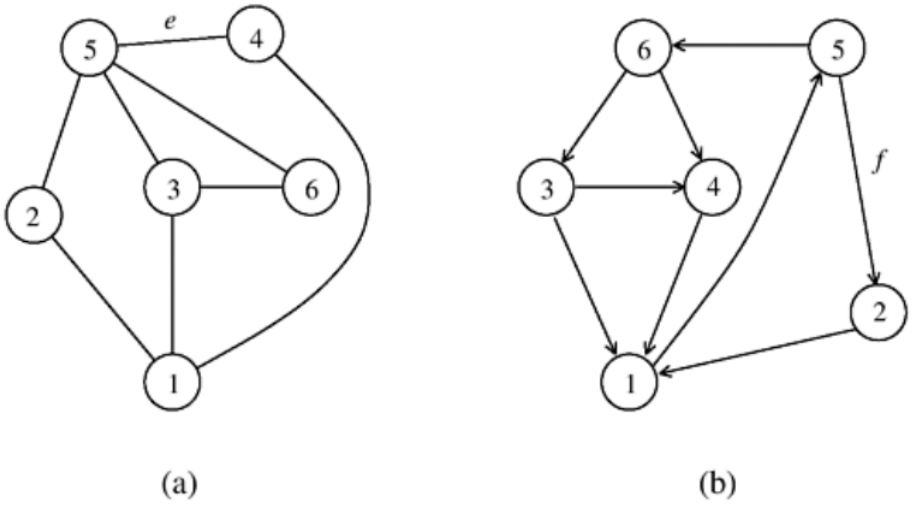
xét, góp ý trong phần trình bày của nhóm để bài báo cáo của nhóm được hoàn thiện hơn. Trong quá trình thực hiện do kiến thức còn nhiều hạn chế và thời gian có giới hạn, nên bài báo cáo chắc chắn vẫn còn những thiếu xót, nhóm rất mong những nhận xét và góp ý của thầy cô và các bạn học viên để bài báo cáo được hoàn thiện hơn.

Hà Nội, ngày 12 tháng 10 năm 2020

Chương 1

Cơ sở lý thuyết

Đồ thị vô hướng $G = (V, E)$ bao gồm V là tập các đỉnh, E là tập các cặp không phân biệt thứ tự gồm hai phần tử của V gọi là các cạnh. Hai đỉnh (u, v) của đồ thị vô hướng G được gọi là kề nhau nếu (u, v) là cạnh của đồ thị. $e = (u, v)$ là cạnh của đồ thị thì ta nói cạnh này là liên thuộc với hai đỉnh u và v, đồng thời các đỉnh u và v sẽ được gọi là các đỉnh đầu của cạnh (u, v).



Hình 1.1: (a) Đồ thị vô hướng (b) đồ thị có hướng

Đường đi độ dài n, kí hiệu P_n, đi từ đỉnh u đến đỉnh v (trong đó n là số nguyên dương) trên đồ thị vô hướng $G = (V, E)$ là dãy $x_0, x_1, ..., x_{n-1}, x_n$, trong đó $u = x_0$, $v = x_n$, $(x_i, x_{i+1}) \in E$, $i = 0, 1, 2, ..., n-1$. Đường đi nói trên còn có thể biểu diễn dưới dạng dãy các cạnh $(x_0, x_1), (x_1, x_2), ..., (x_{n-1}, x_n)$. Đường đi có đỉnh đầu trùng với đỉnh cuối ($u \equiv v$) được gọi là chu trình. Đồ thị không chứa chu trình được gọi là *acyclic*

Đồ thị vô hướng $G = (V, E)$ được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó. Đồ thị liên thông G được gọi là đồ thị k-liên thông (k-connected), với k là số nguyên dương, nếu xóa đi $k - 1$ đỉnh bất kì thuộc G, ta thu

được một đồ thị mới liên thông.

Đồ thị con của đồ thị $G = (V, E)$ là đồ thị $H = (V', E')$, trong đó $V' \subseteq V$ và $E' \subseteq E$. Trong trường hợp đồ thị không liên thông, khi đó đồ thị có các đồ thị con liên thông không có đỉnh chung. Những đồ thị con liên thông như vậy ta sẽ gọi là thành phần liên thông của đồ thị.

Đồ thị đầy đủ n đỉnh, kí hiệu K_n, là đơn đồ thị vô hướng mà giữa hai đỉnh bất kì của nó luôn có cạnh nối. Đồ thị đầy đủ K_n có tất cả $n(n-1)/2$ cạnh, là đơn đồ thị có nhiều cạnh nhất.

Cây là đồ thị vô hướng liên thông không chứa chu trình. Đồ thị không chứa chu trình được gọi là rừng. $G = (V, E)$ là đồ thị vô hướng liên thông, cây $T = (V, F)$ với $F \subset E$ được gọi là cây khung của đồ thị G. Như vậy, rừng là đồ thị mà mỗi thành phần liên thông của nó là 1 cây. Rừng tuyến tính (*linear forest*) là 1 rừng thỏa mãn bậc của mọi đỉnh đều nhỏ hơn bằng 2.Lưu ý rằng nếu $G = (V, E)$ là một cây, khi đó: $|E| = |V| - 1$.

Trong một số trường hợp, mỗi cạnh trong đồ thị được liên kết với một trọng số. Trọng số thường là số thực đại diện cho chi phí khi đi qua các cạnh có trọng số tương ứng. Đồ thị có trọng số liên kết với mỗi cạnh được gọi là đồ thị trọng số, được biểu thị bằng $G = (V, E, w)$, trong đó V là tập các đỉnh, E là tập các cạnh, và hàm $w : E \to \mathbb{R}$. Trọng số của đồ thị được định nghĩa là tổng trọng số các cạnh của đồ thị. Tương tự, trọng số của đường đi được định nghĩa là tổng trọng số của cạnh của đường đi.

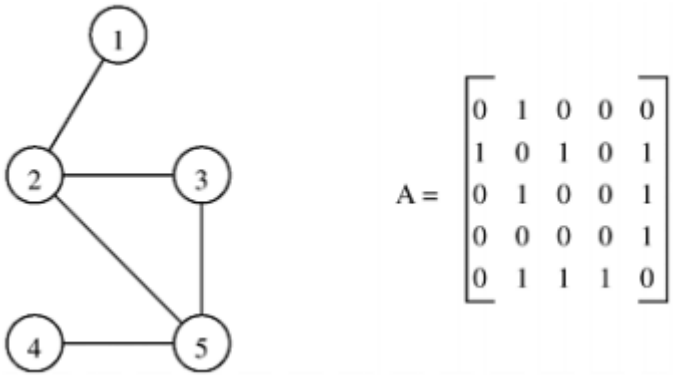
Có 2 phương pháp để biểu diễn một đồ thị trong máy tính. Phương pháp đầu tiên là sử dụng ma trận kề, phương pháp thứ 2 là sử dụng một tập hợp danh sách kề. Xét đồ thị $G = (V, E)$, trong đó $V = 1, 2, 3, \ldots, n$, ma trận biểu diễn đồ thị này là một mảng $n \times n$ $A = (a_{i,j})$, được định nghĩa như sau:

$$a_{i,j} = \begin{cases} 1 & \text{nếu}(v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

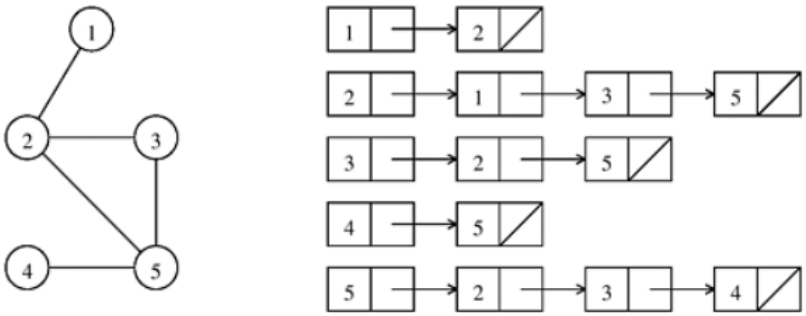
Hình 1.2 minh họa một cách biểu diễn ma trận kề (*adjacency matrix*) của một đồ thị vô hướng. Lưu ý rằng, ma trận kề biểu diễn của đồ thị vô hướng là ma trận đối xứng. Độ phức tạp bộ nhớ để lưu trữ ma trận kề của một đồ thị n đỉnh là $O(n^2)$.

Trong lý thuyết đồ thị, danh sách kề (*adjacency list*) là danh sách biểu diễn tất cả các cạnh hoặc cung trong một đồ thị. Trong đồ thị vô hướng, mỗi phần tử của danh sách là một cặp 2 đỉnh - 2 đầu nút của cạnh tương ứng. Trong đồ thị có hướng, mỗi phần tử là một cặp có thứ tự gồm 2 đỉnh là đỉnh đầu và đỉnh cuối trong cung tương ứng. Như vậy, với đồ thị $G = (V, E)$ bao gồm một mảng $Adj[1..|V|]$ gồm $|V|$ danh sách. Với mỗi $u \in V$, $Adj[u]$ là một danh sách kề chứa trỏ đến tất cả các đỉnh v sao cho tồn tạo một cạnh $(u, v) \in E$, nghĩa là $Adj[u]$ bao gồm tất cả các đỉnh kề với u trong đồ thị G. Hình 1.3 là một ví dụ về biểu diễn danh sách kề. Độ phức tạp bộ nhớ để lưu trữ danh sách kề của một đồ thị là $O(|E|)$, trong đó $|E|$ là tổng số cạnh của đồ thị.

Đồ thị $G = (V, E)$ được gọi là thưa (*sparse graph*) nếu $|E|$ nhỏ hơn nhiều so với $O(|V|^2)$, ngược lại đồ thị dày (*dense graph*) nếu $|E|$ sát với tập đỉnh bình phương $|V|^2$.



Hình 1.2: Đồ thị vô hướng và ma trận kề của nó



Hình 1.3: Đồ thị vô hướng và danh sách kề của nó

Phép biểu diễn danh sách kề thường được ưa dùng, bởi nó cung cấp một cách nén gọn để biểu diễn các đồ thị thưa. Hầu hết các thuật toán về đồ thị được thực hiện nhập dữ liệu và xử lý theo dạng danh sách kề. Tuy nhiên, phép biểu diễn ma trận kề thường được sử dụng khi đồ thị dày hoặc khi cần nhanh chóng xác định có một cạnh nối giữa hai đỉnh đã cho hay không. Ví dụ bài toán tìm đường đi ngắn nhất với tất cả các cặp đỉnh thì có thể dùng ma trận kề biểu diễn đồ thị. Tuy nhiên, trong bài toán tìm cây khung nhỏ nhất theo thuật toán PRIM thì danh sách kề sẽ được sử dụng tốt hơn.

Phần tiếp theo của báo cáo sẽ trình bày các thuật toán đồ thị, 4 phần đầu tiên trình các thuật toán đối với đồ thị dày, và phần cuối cùng sẽ thảo luận về các thuật toán cho đồ thị thưa. Tác giả giả định rằng, đồ thị dày sẽ được biểu diễn bởi một ma trận kề, đồ thị thưa thớt được biểu diễn bởi một tập hợp các danh sách kề. Trong đó, n biểu thị số đỉnh của đồ thị.

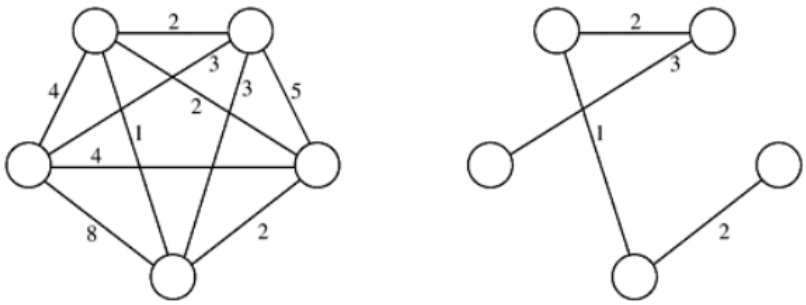
Chương 2

Các thuật toán với đồ thị dày

2.1 Cây khung nhỏ nhất: Thuật toán Prim

Cây khung(*spanning tree*), là cây con của đồ thị G, chứa tất cả các đỉnh của G.

Nói cách khác, cây khung của một đồ thị G là một đồ thị con của G, liên thông và không có chu trình. Cây khung của đồ thị liên thông G cũng có thể định nghĩa như một đồ thị con không chứa chu trình lớn nhất, hay một đồ thị con liên thông nhỏ nhất của G. Mọi đồ thị liên thông đều có cây khung. Cây khung nhỏ nhất (*Minimum Spanning Tree* - MST) của đồ thị vô hướng có trọng số là cây khung có tổng trọng số nhỏ nhất. Với đồ thị G không liên thông, không thể tồn tại một cây khung, thay vào đó, tồn tại một rừng khung (*spanning forest*)



Hình 2.1: MST của một đồ thị vô hướng

Bài toán cây khung nhỏ nhất của đồ thị là một trong những bài toán tối ưu trên đồ thị có nhiều ứng dụng khác nhau trong thực tế, một số mô hình thực tế tiêu biểu của bài toán như:

- Bài toán xây dựng đường giao thông: giả sử cần xây dựng một hệ thống đường nối n thành phố sao cho giữa các thành phố bất kỳ luôn có đường đi. Bài toán đặt ra là xác định cây khung nhỏ nhất trên đồ thị, mỗi thanh phố ứng với một đỉnh sao cho tổng chi phí xây dựng đường đi là nhỏ nhất.

- Bài toán nối mạng máy tính: cần nối mạng một hệ thống gồm n máy tính đánh số từ 1 đến n. Biết chi phí nối máy i với máy j là $c(i,j), i, j = 1, 2, \ldots, n$. Bài toán đặt ra là xác định cách nối mạng sao cho tổng chi phí nối mạng là nhỏ nhất.

Thuật toán Prim là một thuật toán tham lam để tìm cây khung nhỏ nhất của một đồ thị vô hướng liên thông có trọng số. Thuật toán tìm các tập hợp các cạnh của đồ thị tạo thành một cây chứa tất cả các đỉnh, sao cho tổng trọng số các cạnh của cây là nhỏ nhất. Thuật toán được tìm ra năm 1930 bởi nhà toán học người Séc Vojtech Jarnik và sau đó bởi nhà nghiên cứu khoa học máy tính Robert C. Prim nằm 1957 và một lần nữa độc lập bởi Edsger Dijkstra năm 1959. Do đó nó còn được gọi là thuật toán DJP, thuật toán Jarnik hay thuật toán Prim-Janik.

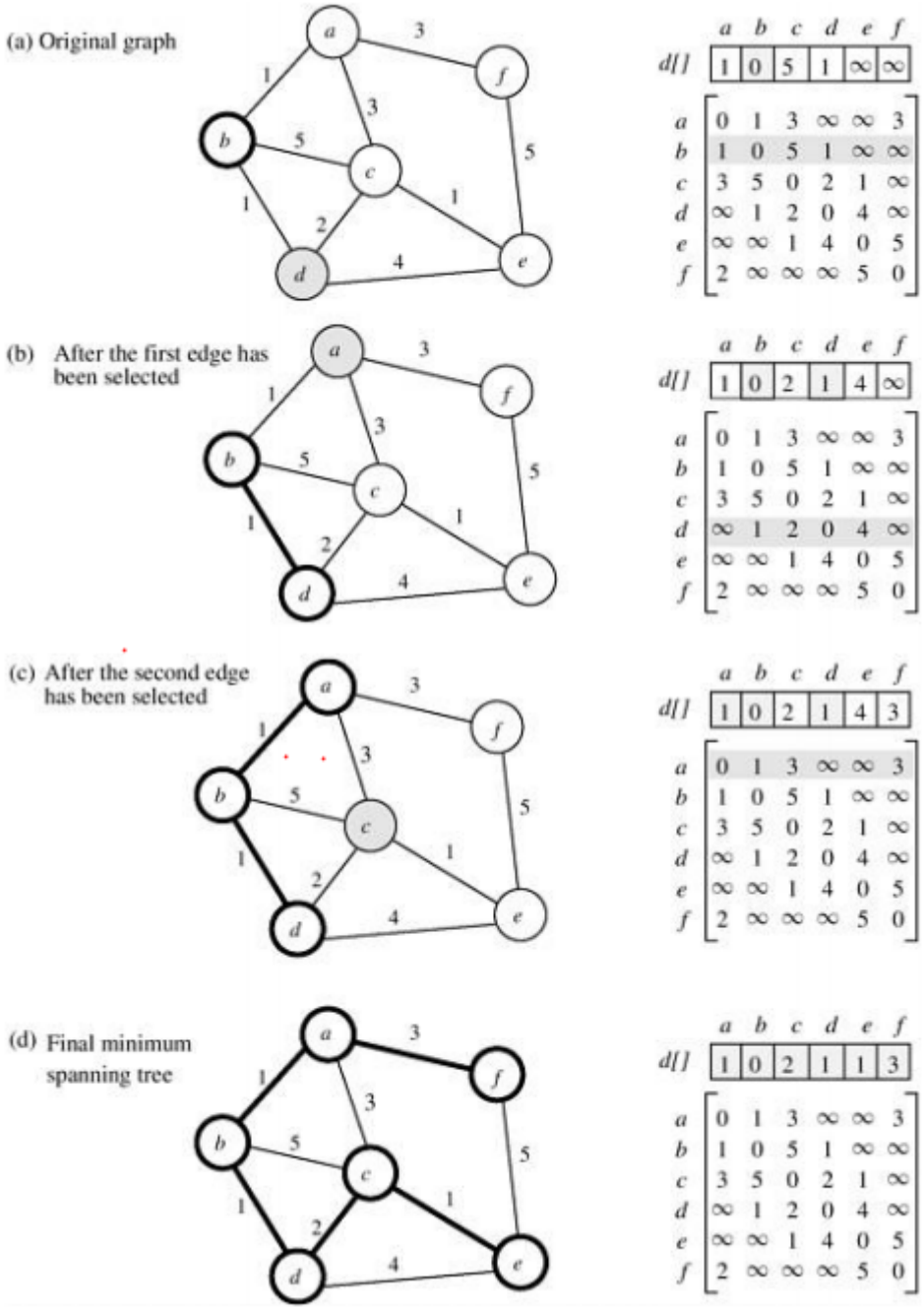
Thuật toán xuất phát từ một cây chỉ chứa đúng một đỉnh tùy ý, và mở rộng từng bước một, mỗi bước thêm một đỉnh mới vào cây và cạnh được đảm bảo là có chi phí tối thiểu, cho tới khi khung được tất cả các đỉnh của đồ thị.

Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số, $A = (a_i, j)$ là ma trận trọng số tương ứng của nó. Tập V_T là tập các đỉnh của MST, mảng $d[1 \ldots n]$, trong đó, với mỗi đỉnh $v \in (V - V_T)$, $d[v]$ lưu trọng số tối thiểu của cạnh (u, v) với $u \in V_T$ là đỉnh kề của v. Dừng thuật toán khi $V_T = V$.

```
1.   procedure PRIM_MST(V, E, w, r)
2.   begin
3.      VT := {r};
4.      d[r] := 0;
5.      for all v ∈ (V - VT ) do
6.         if edge (r, v) exists set d[v] := w(r, v);
7.         else set d[v] := ∞ ;
8.      while VT ≠ V do
9.      begin
10.        find a vertex u such that d[u] :=min{d[v]|v ∈ (V - VT )};
11.        VT := VT ∪ {u};
12.        for all v ∈ (V - VT ) do
13.            d[v] := min{d[v], w(u, v)};
14.     endwhile
15.  end PRIM_MST
```

Hình 2.2: Thuật toán Prim tìm MST

Sau khi dừng thuật toán, trọng số của MST là $\sum_{v \in V} d[v]$. Hình 2.3 cho một ví dụ minh họa thuật toán Prim tìm cây khung nhỏ nhất, MST xuất phát từ đỉnh b, với mỗi lần lặp, các đỉnh thuộc V_T và các cạnh được chọn được in đậm.



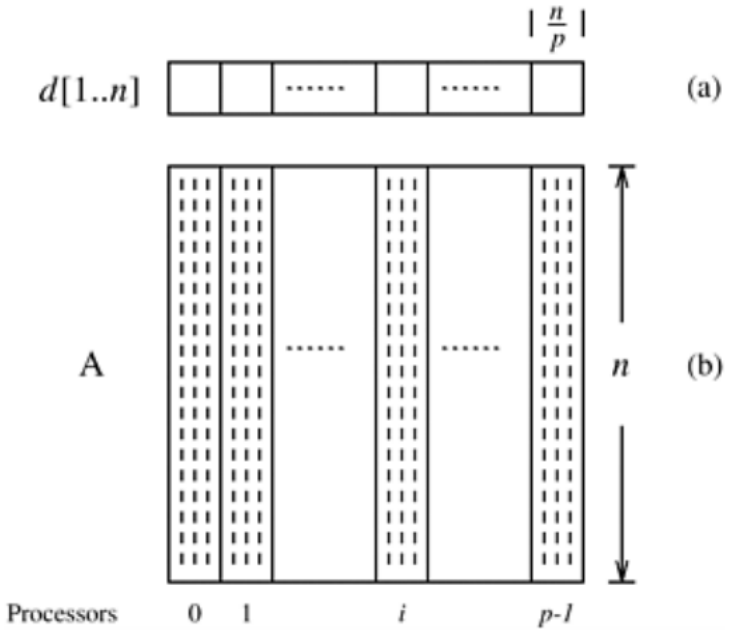
Hình 2.3: Ví dụ minh họa thuật toán Prim tìm MST

Thuật toán sử dụng 2 vòng lặp, mỗi vòng có $(n-1)$ lần lặp, do đó độ phức tạp của thuật toán Prim là $O(n^2)$.

Xây dựng thuật toán song song

Thuật toán Prim sử dụng 2 vòng lặp, mỗi lần lặp ta thêm 1 đỉnh mới vào MST. Có thể thấy, giá trị của $d[v]$ có thể thay đổi mỗi khi có một đỉnh u được thêm vào V_T, điều này rất khó để chọn nhiều hơn 1 đỉnh để thêm vào MST trong một lần lặp. Xét ví dụ 2.3, sau khi chọn đỉnh b, nếu cả 2 đỉnh d, c được chọn thì sẽ không xác định được MST, do sau khi chọn đỉnh d, giá trị của $d[c]$ được cập nhật bằng 2 thay vì là 5. Do vậy, khó có thể xây dựng song song các lần lặp khác nhau của của vòng lặp `while`. Tuy nhiên, mỗi lần lặp có thể được thực hiện song song như sau:

Cho đồ thị vô hướng liên thông $G = (V, E)$. Đặt p là số các bộ xử lý, n là số đỉnh của đồ thị. Tập V được tách thành p tập con sử dụng 1-D block mapping. Mỗi tập con có n/p đỉnh liên tiếp (*consecutive vertices*), là các cột liên tiếp của ma trận kề, mỗi tập con sẽ được xử lý bằng một bộ xử lý khác nhau. Đặt V_i là các tập con được xử lý bằng bộ xử lý P_i với $i = 1, 2, \ldots p-1$. Mỗi bộ xử lý P_i lưu trữ một mảng $d[v]$ với $v \in V_i$. Hình 2.4(a) minh họa sự phân vùng. Với mỗi bộ xử lý P_i, xác định $d_i[u] = min\{d_i[v]|v \in (V - V_T) \cap V_i\}$ trong mỗi lần lặp của vòng lặp `while`. Trọng số của cây khung nhỏ nhất của đồ thị thu được từ các $d_i[u]$ bằng cách tổng hợp kết quả từ các bộ xử lý sử dụng add-to-one reduction operation, lấy $d_i min[u]$ và được lưu trữ trong P_0. bộ xử lý P_0 lưu trữ các đỉnh u mới, sẽ được thêm vào V_T. bộ xử lý P_0 phát đỉnh u đến tất cả các bộ xử lý khác. bộ xử lý P_i đánh dấu u thuộc tập V_T, mỗi bộ xử lý cập nhật các giá trị cho $d[v]$.



Hình 2.4: Xây dựng thuật toán song song cho MST

Khi một đỉnh u được thêm vào V_T, các giá trị của $d[v]$ với $v \in (V - V_T)$ được cập

nhật. bộ xử lý đảm nhiệm đỉnh v tính trọng số của các cạnh (u, v). Do đó, mỗi bộ xử lý P_i cần để lưu trữ các cột của ma trận kề có trọng số tương ứng với tập con V_i của các đỉnh được gán cho nó. Điều này tương ứng với 1-D block mapping của ma trận. Vì vậy, độ phức tạp của mỗi bộ xử lý là $O(n^2/p)$. Hình 2.4(b) minh họa sự phân vùng của ma trận kề.

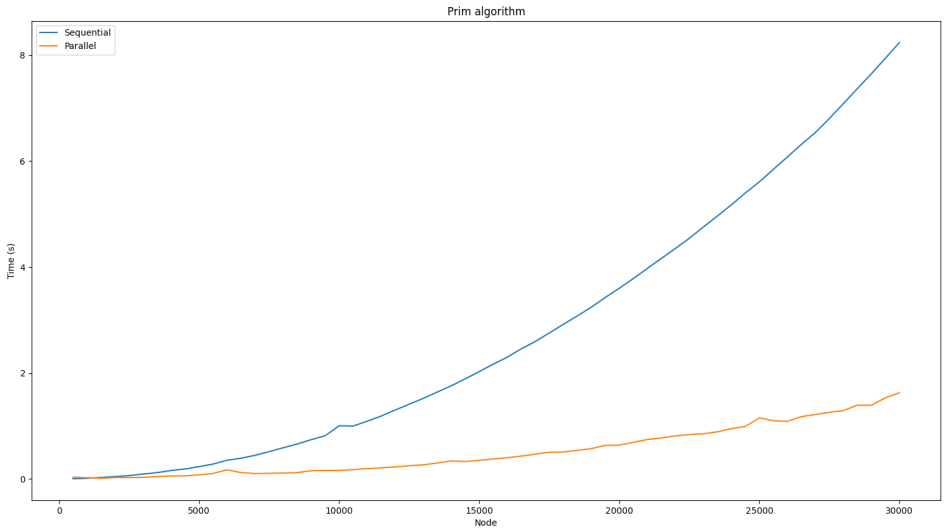
Các bộ xử lý tìm cạnh có trọng số nhỏ nhất và cập nhất các giá trị $d[v]$ trong mỗi lần lặp là $O(n/p)$. Trong một bộ xử lý, thời gian thực hiện tìm giá trị nhỏ nhất trong mỗi lần lặp là $O(\log p)$. Do vậy, thời gian thực hiện thuật toán song song là:

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log p).}^{\text{communication}}$$

So với thuật toán tuần tự, độ phức tạp của thuật toán song song tương đương nhau. Kết quả chỉ thể hiện được qua thực nghiệm chương trình. Vì thời gian chạy tuần từ là $W = O(n^2)$, tốc độ và hiệu suất được xác định theo công thức:

$$\begin{aligned} S &= \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p)} \\ E &= \frac{1}{1 + \Theta((p \log p)/n)} \end{aligned}$$

Kết quả thực nghiệm so sánh giữa thuật toán Prim tuần tự và song song trong hình 2.5.



Hình 2.5: Thực nghiệm thuật toán Prim tuần tự và song song

2.2 Đường đi ngắn nhất xuất phát từ một đỉnh: Thuật toán Dijkstra

Trong các ứng dụng thực tế, bài toán tìm đường đi ngắn nhất giữa hai đỉnh của một đồ thị liên thông có một ý nghĩa to lớn. Có thể dẫn bài toán như vậy về nhiều bài toán thực tế quan trọng. Ví dụ, bài toán chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn khoảng cách hoặc thời gian hoặc chi phí) trên một mạng giao thông đường bộ, đường thủy hoặc đường không; bài toán chọn một phương pháp tiết kiệm nhất để đưa một hệ động lực từ trạng thái xuất phát đến một trạng thái đích; bài toán lập lịch thi công các công đoạn trong một công trình thi công lớn, ... Hiện nay có rất nhiều phương pháp để giải các bài toán như vậy, thế nhưng, thông thường, các thuật toán được xây dựng dựa trên lý thuyết đồ thị tỏ ra là các thuật toán có hiệu quả cao nhất.

Cho đồ thị trọng số $G = (V, E, w)$, bài toán tìm đường đi ngắn nhất từ đỉnh $v \in V$ đến tất cả các đỉnh còn lại của đồ thị. Đường đi ngắn nhất từ u đến v là đường đi có trọng số nhỏ nhất. Trong phần này, nhóm trình bày thuật toán Dijkstra, mang tên của nhà khoa học máy tính người Hà Lan Edsger Dijkstra vào năm 1956, giải quyết bài toán tìm đường đi ngắn nhất xuất phát từ một đỉnh trên cả đồ thị có hướng và vô hướng với trọng số trên các cung là không âm.

Thuật toán dựa trên cơ sở gán cho các đỉnh các nhãn tạm thời. Nhãn của mỗi đỉnh cho biết cận trên của độ dài đường đi ngắn nhất từ v đến nó. Các nhãn này sẽ được biến đổi theo một thủ tục lặp, mà ở mỗi bước lặp có một nhãn tạm thời trở thành nhãn cố định. Nếu nhãn của một đỉnh nào đó trở thành cố định thì thuật toán sẽ trả về độ dài của đường đi ngắn nhất từ đỉnh v đến nó. Thuật toán được mô tả như sau:

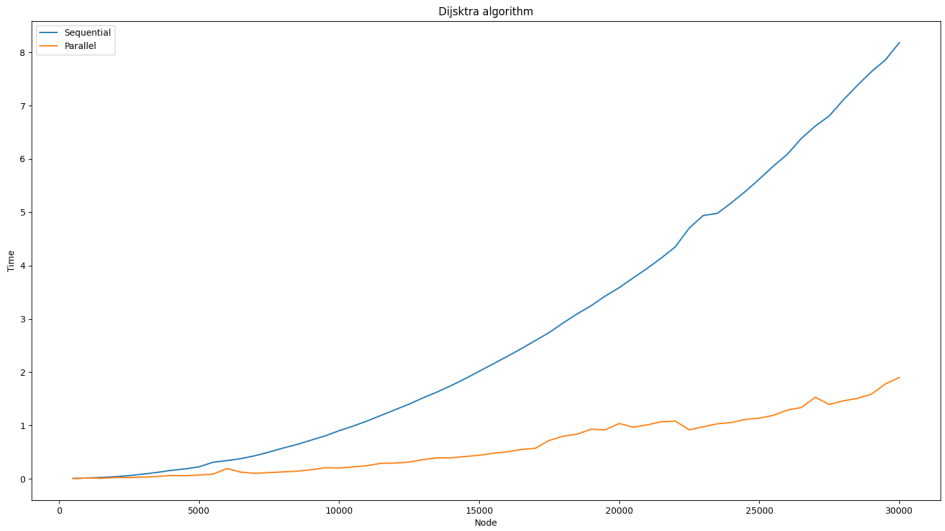
```
1.   procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.   begin
3.      VT := {s};
4.      for all v ∈ (V - VT ) do
5.         if (s, v) exists set l[v] := w(s, v);
6.         else set l[v] := ∞ ;
7.      while VT ≠ V do
8.      begin
9.         find a vertex u such that l[u] := min{l[v]|v ∈ (V - VT )};
10.        VT := VT ⋃ {u};
11.        for all v ∈ (V - VT ) do
12.            l[v] := min{l[v], l[u] + w(u, v)};
13.     endwhile
14.  end DIJKSTRA_SINGLE_SOURCE_SP
```

Hình 2.6: Thuật toán Dijkstra tìm đường đi ngắn nhất xuất phát từ một đỉnh

Tương tự thuật toán Prim tìm cây khung nhỏ nhất, thuật toán Dijkstra từng bước xác định các đường đi ngắn nhất từ đỉnh s đến các đỉnh khác của G. Thuật toán cũng là một thuật toán tham lam, nó luôn chọn cạnh của đỉnh xuất hiện gần nhất. So sánh thuật toán này với thuật toán Prim, chúng ta có thể thấy rằng hai thuật toán gần như tương tự nhau. Sự khác biệt chính là, với mỗi đỉnh $u \in (V - V_T)$, thuật toán Dijkstra lưu trữ $l[u]$, chi phí tối thiểu của đường đi từ đỉnh s đến đỉnh u đi qua các đỉnh trong V_T, thuật toán Prim lưu trữ $d[u]$, chi phí của cạnh có trọng số tối thiểu nối một đỉnh trong V_T với u. Ở mỗi bước lặp, để tìm ra đỉnh $u \in (V - V_T)$ cần thực hiện n phép toán, thuật toán phải thực hiện $(n-1)$ bước lặp. Do vậy, độ phức tạp của thuật toán Dijkstra là $O(n^2)$.

Xây dưng thuật toán song song

Xây dựng thuật toán song song cho thuật toán Dijkstra tương tự công thức song song của thuật toán Prim. Ma trận kề trọng số được phân vùng bằng cách sử dụng 1-D block mapping. Đồ thị có n đỉnh, thuật toán song song thực hiện trên p bộ xử lý, tập các đỉnh V của đồ thị được chia thành p tập con, mỗi tập con gồm n/p đỉnh liền kề và được gán để xử lý trên một tiện trình. Bộ xử lý P_i quản lý một tập con V_i là các cột liên tiếp của ma trận kề. Tại bước gán nhãn cho đỉnh u, bộ xử lý P_i sẽ tính $d_i[u] = \min d[v]|v \in V_i$. Trọng số đường đi ngắn nhất của đồ thị thu được trên tất cả các $d_i[u]$ bằng cách tổng hợp kết quả từ các bộ xử lý và được lưu trong P_0. bộ xử lý P_0 nhận đỉnh u mới và chèn vào V_T và phát đỉnh u đến các bộ xử lý khác.
Thực nghiệm so sánh giữa thuật toán Dijkstra song song và tuần tự được cho bởi hình 2.7.



Hình 2.7: Thực nghiệm so sánh thuật toán Dijkstra tuần tự và song song

2.3 Đường đi ngắn nhất giữa tất cả các cặp đỉnh

Thay vì tìm đường đi ngắn nhất từ một đỉnh đến các đỉnh còn lại, đôi khi ta có thể quan tâm đến bài toán tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị.

Cho đồ thị có trọng số $G = (V, E, w)$, bài toán tìm đường đi nhắn nhất giữa các cặp đỉnh $v_i, v_j \in V, i \neq j$. Đối với đồ thị n đỉnh, đầu ra của bài toán là một ma trận $D = (d_{i,j})$ kích thước $n \times n$, trong đó $d_{i,j}$ là chi phí đường đi ngắn nhất từ đỉnh v_i đến đỉnh v_j. Các phần tiếp theo của báo cáo trình bày hai thuật toán để giải quyết bài toán này: thuật toán thứ nhất sử dụng thuật toán Dijkstra, thuật toán thứ hai sử dụng thuật toán Floyd. Thuật toán Dijkstra chỉ áp dụng với đồ thị có trọng số không âm, trong khi thuật toán Floyd hoạt động với cả đồ thị có cạnh có trọng số âm không có chu trình.

2.3.1 Thuật toán Dijkstra

Phần 2.2, nhóm đã trình bày thuật toán Dijkstra để tìm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh còn lại. Thuật toán này cũng được áp dụng giải quyết bài toán tìm đường đi ngắn nhất giữa tất cả các đỉnh bằng cách sử dụng n lần thuật toán Dijkstra, trong đó, ta sẽ chọn s lần lượt là các đỉnh của đồ thị. Rõ ràng, khi đó ta thu được thuật toán với độ phức tạp $O(n^3)$.

Xây dựng thuật toán song song

Có hai cách để xây dựng thuật toán song song cho bài toán tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh: một cách tiếp cận đầu tiên là phân chia các đỉnh giữa các bộ xử lý, mỗi bộ xử lý tính toán đường đi ngắn nhất nguồn đơn cho tất cả các đỉnh được

gán với nó, một cách tiếp cận khác là một đỉnh được xử lý bởi một tập các bộ xử lý và sử dụng công thức song song của thuật toán (phần 2.2).

Cách tiếp cận đầu tiên sử dụng n bộ xử lý, mỗi bộ xử lý P_i tìm các đường đi ngắn nhất từ đỉnh v_i đến tất cả các đỉnh con lại bằng cách sử dụng thuật toán Dijkstra tuần tự. Thời gian chạy song song được biểu diễn bởi công thức:

$$T_P = \Theta(n^2).$$

Do độ phức tạp của thuật toán tuần tự là $W = O(n^3)$, tốc độ và hiệu suất được tính bằng công thức:

$$\begin{aligned} S &= \frac{\Theta(n^3)}{\Theta(n^2)} \\ E &= \Theta(1) \end{aligned}$$

Thuật toán có thể sử dụng nhiều nhất n quy trình. Hàm đẳng tích do đồng thời (*isoefficiency function* cho biết tốc độ tăng trưởng của W cần thiết để giữa cho hiệu suất cố định khi số bộ xử lý tăng) của hệ thống song song này là $O(p^3)$, cũng là hàm đẳng tích tổng thể. Nếu số lượng các bộ xử lý có sẵn để xử lý bài toán nhỏ ($n = O(p)$), khi đó thuật toán sẽ có hiệu suất tốt. Tuy nhiên, nếu số lượng bộ xử lý lớn hơn n, thuật toán sẽ bị giảm hiệu suất.

Với cách tiếp cận thứ hai, p bộ xử lý được chia thành n phân vùng. Nói cách khác, trước tiên ta song song bài toán đường đi ngắn nhất giữa tất cả các cặp đỉnh bằng cách gán mỗi đỉnh cho một tập hợp các bộ xử lý riêng biệt, sau đó song song thuật toán Dijkstra tuần tự bằng các sử dụng tập p/n bộ xử lý để giải quyết bài toán. Tổng số bộ xử lý có thể được sử dụng hiệu quả là $O(n^2)$. Khi đó, thời gian chạy song song được biểu diễn bởi công thức:

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log p).}^{\text{communication}}$$

Do độ phức tạp của thuật toán tuần tự là $W = O(n^3)$, tốc độ và hiệu suất được tính bằng công thức:

$$
\begin{aligned}
S &= \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta(n \log p)} \\
E &= \frac{1}{1 + \Theta((p \log p)/n^2)}
\end{aligned}
$$

Từ công thức trên, ta thấy rằng đối với công thức tối ưu chi phí $(p \log p)/n^2 = O(1)$. Do đó, hệ thống song song hiệu quả có thể sử dụng lên đến $O(n^2/\log n)$ bộ xử lý. Phương trình trên cũng chỉ ra rằng hàm đẳng tích do truyền thông tin (*communication*) là $O((p \log p)^{1.5})$. Hàm đẳng tích do tương tranh (*concurrency*) là $O(p^{1.5})$. Do vậy, hàm đẳng tích tổng thể là $O((p \log p)^{1.5})$.

So sánh hai cách tiếp cận song song với bài toán tìm đường đi ngắn nhất giữa các cặp đỉnh, ta thấy rằng với cách tiếp cận thứ nhất có thể sử dụng không quá n quy trình và thời gian giải quyết bài toán là $O(n^2)$. Ngược lại, với cách tiếp cận thứ hai, sử dụng lên đến $n^2/\log n$ bộ xử lý, thời gian giải quyết bài toán là $O(n \log n)$. Do đó, có thể thấy cách tiếp cận thứ hai tốt hơn.

2.3.2 Thuật toán Floyd

Thuật toán Floyd được Robert Floyd công bố vào năm 1962, nó cũng được tìm ra một cách độc lập bởi Bernard Roy vào năm 1959 và Stephen Warshall vào năm 1962. Thuật toán được sử dụng để giải quyết bài toán tìm đường đi ngắn nhất trên đồ thị có hướng, có trọng số, không có chu trình âm. Ý tưởng của thuật toán dựa trên phương pháp quy hoạch động: tối thiểu hóa đường đi giữa hai đỉnh bất kỳ bằng cách tối thiểu hóa đường đi giữa các đỉnh trung gian.

Cho độ thị có trọng số $G = (V, E, w)$, $V = \{v_1, v_2, \ldots, v_n\}$ là tập đỉnh của G. Xét tập con $\{v_1, v_2, \ldots, v_k\}, k \leq n$. Với mỗi cặp đỉnh $v_i, v_j \in V$, xét tất cả các đường đi từ v_i đến v_j đi qua các đỉnh thuộc tập $\{v_1, v_2, \ldots, v_k\}$. Đặt $P_{i,j}^{(k)}$ là đường đi có trọng số nhỏ nhất giữa hai đỉnh v_i, v_j mà chỉ sử dụng các đỉnh thuộc $\{v_1, v_2, \ldots, v_k\}$ làm đỉnh trung gian, $d_{i,j}^{(k)}$ là trọng số của $P_{i,j}^{(k)}$. Nếu đỉnh v_k không thuộc đường đi ngắn nhất từ v_i đến v_j, khi đó $P_{i,j}^{(k)} \equiv P_{i,j}^{(k)}$. Tuy nhiên, nếu $v_k \in P_{i,j}^{(k)}$, ta có thể chia $P_{i,j}^{(k)}$ thành 2 đường đi con, một từ v_i đến v_k, và một từ v_k đến v_j, mỗi đường đi đều sử dụng các đỉnh thuộc $v_1, v_2, \ldots, v_k$ làm trung gian. Do vậy,

$$d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$$

Khi đó, ta xác định công thức truy hồi:

$$
d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{Nếu } k = 0 \\ \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} & \text{Nếu } k \geq 1 \end{cases}
$$

Trọng số của đường đi ngắn nhất từ v_i đến v_j được cho bởi $d_{i,j}^{(n)}$. Như vậy, kết quả của bài toán được cho bởi ma trận $D^{(n)} = (d_{i,j}^{(n)})$.

```
1.   procedure FLOYD_ALL_PAIRS_SP( A)
2.   begin
3.      D(0) = A;
4.      for k := 1 to n do
5.          for i := 1 to n do
6.              for j := 1 to n do
                    d(k)i,j := min(d(k-1)i,j, d(k-1)i,k + d(k-1)k,j);
7.
8.   end FLOYD_ALL_PAIRS_SP
```

Hình 2.8: Thuật toán Floyd

Thuật toán sử dụng 3 vòng lặp `for` lồng nhau, mỗi vòng lặp có $n = |V|$ bước lặp, do vậy độ phức tạp thời gian của thuật toán là $O(n^3)$. Thuật toán sử dụng một mảng 2 chiều kích thước $n \times n$ để lưu ma trận độ dài đường đi ngắn nhất giữa hai đỉnh bất kỳ, do vậy độ phức tạp không gian của thuật toán là $O(n^2)$.

Xây dựng thuật toán song song

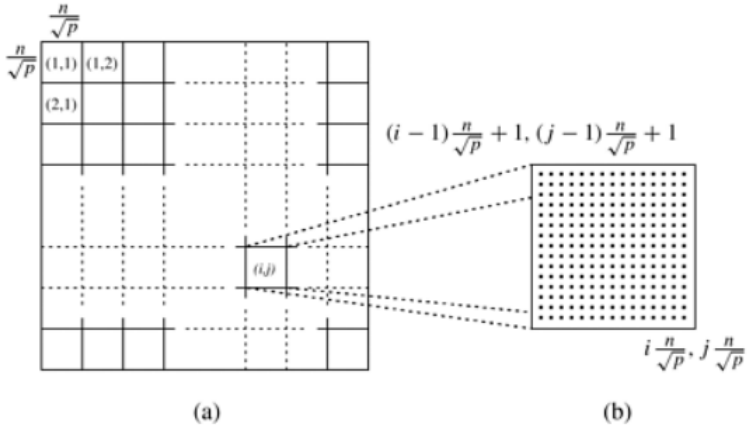
Tại mỗi vòng lặp k, thuật toán không cần sử dụng toàn bộ các phần tử trong ma trận. Do vậy,để xây dựng công thức song song cho thuật toán Floyd ta gán việc xử lý tính toán ma trận $D^{(k)}$ bằng một tập hợp các bộ xử lý, tức là ta chia ma trận độ dài đường đi thành các block, cung cấp đủ dữ liệu cần thiết để các block tính toán, và chạy song song mỗi block trên một bộ xử lý. Đặt p là số bộ xử lý. Ma trận $D^{(k)}$ được chia thành p phần, mỗi phần gán cho một bộ xử lý, mỗi bộ xử lý tính giá trị $D^{(k)}$ của phần đó. Để thực hiện việc này, bộ xử lý phải tuy cập vào các phần tương ứng với hàng và cột thứ k của ma trận $D^{(k-1)}$. Tiếp theo, nhóm sẽ mô tả một kỹ thuật để phân vùng ma trận $D^{(k)}$.

2-D Block Mapping là một cách để phân vùng ma trận $D^{(k)}$. Ta có ma trận độ dài đường đi kích thước $n \times n$, máy tính có p bộ xử lý, ta chia ma trận thành các block có kích thước $[\frac{n}{\sqrt{p}}] \times [\frac{n}{\sqrt{p}}]$, phần dư nếu có gộp vào một block. Như vậy, p bộ xử lý được sắp xếp trong một lưới có kích thước $\sqrt{p} \times \sqrt{p}$, bộ xử lý tại vị trí hàng i cột j ký hiệu $P_{i,j}$. Bộ xử lý $P_{i,j}$ gán cho một khối con của $D^{(k)}$ có góc trên bên trái là $((i-1)\frac{n}{\sqrt{p}}+1, (j-1)\frac{n}{\sqrt{p}}+1)$ và góc dưới bên phải là $(i\frac{n}{\sqrt{p}}, j\frac{n}{\sqrt{p}})$. Mỗi bộ xử lý cập nhật một phần tương ứng của ma trận trong mỗi lần lặp.

Sự phân chia nhiệm vụ của bộ xử lý được phân chia như sau:

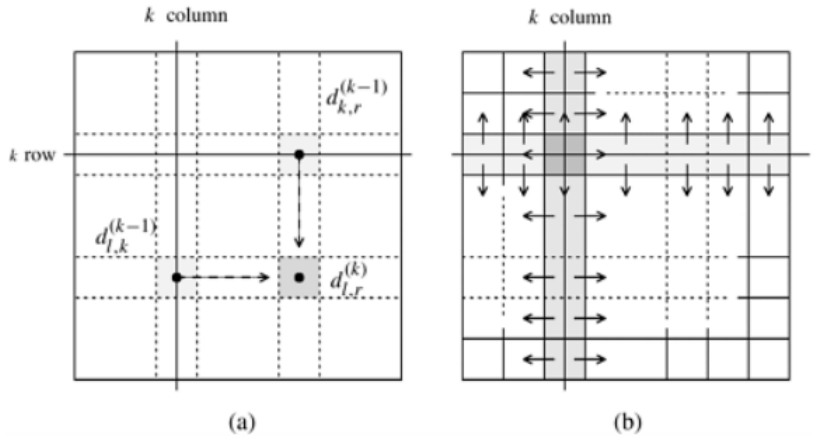
- Các bộ xử lý phụ: chạy thuật toán trên mỗi block khi nhận đủ dữ liệu.
- Bộ xử lý chính: đảm bảo truyền đủ dữ liệu đến mỗi bộ xử lý phụ, xác định khí nào có một bộ xử lý phụ được phép chạy.

Do sau khi kết thúc một vòng lặp, giá trị mỗi phần tử ma trận trong mỗi block là nhỏ nhất tính đến thời điểm hiện tại, nên bộ xử lý chính không cần thực hiện bước tổng hợp thông tin từ các bộ xử lý phụ. Tại mỗi vòng lặp, thuật toán chỉ sử dụng dữ liệu ma trận của vòng lặp trước, bộ xử lý chính chỉ thực hiện truyền dữ liệu một lần vào đầu mỗi vòng lặp đến các bộ xử lý phụ.



Hình 2.9: (a) Ma trận $D^{(k)}$ được phân phối bởi 2-D block mapping thành các khối con (b) Khối con của ma trận $D^{(k)}$ được gán cho bộ xử lý $P_{i,j}$

Trong k lần lặp đầu tiên của thuật toán, mỗi bộ xử lý $P_{i,j}$ cần thông tin nhất định tại vị trí hàng thứ k, cột thứ k của ma trận $D^{(k-1)}$. Ví dụ, để xác định $d_{l,r}^{(k)}$, ta cần có thông tin về $d_{l,k}^{(k-1)}$ và $d_{k,r}^{(k-1)}$. Hình 2.10 minh họa, bộ xử lý chứa $d_{l,k}^{(k-1)}$ nằm cùng một hàng và bộ xử lý chứa $d_{k,r}^{(k-1)}$ nằm trên cùng một cột với bộ xử lý $d_{l,r}^{(k)}$ cần tính toán. Những bộ xử lý đang tính toán những phần tử nằm trên hàng (hay cột) k, sẽ truyền thông tin những phần tử này đến những bộ xử lý nằm cùng cột (hay hàng) với nó. Với mỗi bộ xử lý như vậy có $\frac{n}{\sqrt{p}}$ phần tử nằm trên hàng hoặc cột k, như vậy nó cần truyền $\frac{n}{\sqrt{p}}$ phần tử.



Hình 2.10: (a) Cách mẫu giao tiếp được sử dụng trong 2-D Block Mapping (b) Hàng và cột của $\sqrt{p}$ bộ xử lý chứa hàng và cột thứ k tương ứng

Độ phức tạp thời gian được xác định bằng tổng độ phức tạp tính toán và độ phức tạp truyền dẫn. Do dữ liệu được chia đều có p block, độ phức tạp tính toán mỗi block là $O(n^3/p)$. Truyền dữ liệu của tất cả các phần tử đến từng block, độ phức tạp truyền dẫn của mỗi lần lặp khi đó là $O(n \log p)/\sqrt{p}$, có n lần lặp do vậy độ phức tạp truyền dẫn của thuật toán là $O(\frac{n^2}{\sqrt{p} \log p})$. Do bộ xử lý chính chỉ cho phép thực thi các bộ xử lý phụ sau khi truyền xong dữ liệu, nên thời gian tính toans và thời gian truyền dẫn không chồng lấn, nên độ phức tạp của thuật toán được xác định bởi công thức:

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta\left(\frac{n^2}{\sqrt{p}} \log p\right)}^{\text{communication}}.$$

Do độ phức tạp của thuật toán Floyd tuần tự là $W = O(n^3)$, tốc độ và hiệu suất được xác định:

$$\begin{aligned} S &= \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta((n^2 \log p)/\sqrt{p})} \\ E &= \frac{1}{1 + \Theta((\sqrt{p} \log p)/n)} \end{aligned}$$

Từ công thức trên, ta thấy rằng công thức tối ưu chi phí là $(\sqrt{p} \log p)/n = O(1)$, do vậy, chỉ nên sử dụng tối đa $O(n^2/\log^2 n)$ bộ xử lý. Cân đối kích thước của block so với ma trận và số bộ xử lý của máy tính, nếu chia quá nhiều block ($\approx n^2$) thời gian

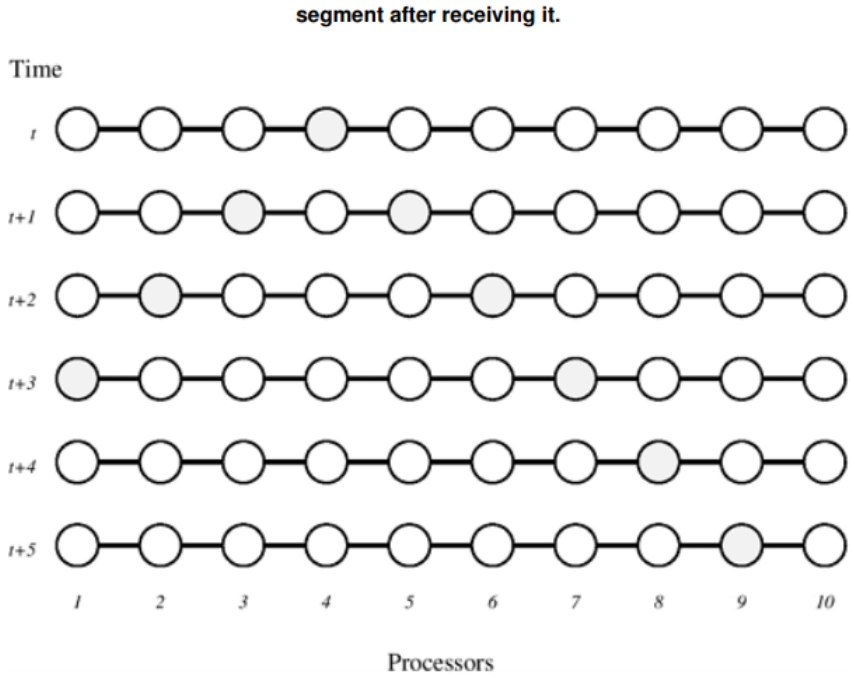
lãng phí do truyền dữ liệu giữa các block lấn át thời gian tiết kiệm được nhờ tính toán song song, nếu chia ra quá ít block (≈ 1) thì không tận dụng triệt để năng lực tính toán song song của máy tính. Từ phương trình trên, ta xác định được hàm đẳng tích do truyền thông tin là $O(p^{1.5}\log^3 p)$, hàm đẳng tích do tương tranh là $O(p^{1.5})$. Do vậy, hàm đẳng tích tổng quan là $O(p^{1.5}\log^3 p)$.

```
1.   procedure FLOYD_2DBLOCK(D(0))
2.   begin
3.      for k := 1 to n do
4.      begin
5.         each process Pi,j that has a segment of the kth row of D(k-1);
              broadcasts it to the P*,j processes;
6.         each process Pi,j that has a segment of the kth column of D(k-1);
              broadcasts it to the Pi,* processes;
7.         each process waits to receive the needed segments;
8.         each process Pi,j computes its part of the D(k) matrix;
9.      end
10.  end FLOYD_2DBLOCK
```

Hình 2.11: Công thức song song của thuật toán Floyd sử dụng 2-D Block Mapping

Trong công thức 2-D Block Mapping của thuật toán Floyd, lần lặp thứ k chỉ bắt đầu khi lần lặp thứ $k-1$ đã hoàn thành, kết quả của ma trận $D^{(k-1)}$ được truyền đến các bộ xử lý. Bước đồng bộ hóa có thể được gỡ bỏ mà không ảnh hướng đến tính chính xác của thuật toán. Để thực hiện việc này, một bộ xử lý sẽ bắt đầu làm việc tại lần lặp thứ k ngay sau khi đã tính toán xong lần lặp thứ $k-1$ và kết quả của ma trận $D^{(k-1)}$. Công thức này được gọi là pipeline 2-D Block Mapping.

Xét một hệ thống có p bộ xử lý được sắp xếp trong một To-po 2 chiều. Giả sử rằng, bộ xử lý $P_{i,j}$ bắt đầu hoạt động tại lần lặp thứ k ngay sau khi kết thúc lần lặp thứ $k-1$ và đã nhận được kết quả của ma trận $D^{(k-1)}$. Khi bộ xử lý $P_{i,j}$ có các phần tử tại hàng k và đã hoàn thành lần lặp thứ $k-1$, bộ xử lý sẽ gửi kết quả của ma trận $D^{(k-1)}$ được lưu trữ cục bộ đến các bộ xử lý $P_{i,j-1}$, $P_{i,j+1}$. Tương tự như vậy, khi bộ xử lý $P_{i,j}$ có các phần tử tại cột k và đã hoàn thành lần lặp thứ $k-1$, bộ xử lý sẽ gửi kết quả của ma trận $D^{(k-1)}$ được lưu trữ cục bộ đến các bộ xử lý $P_{i,j-1}$, $P_{i,j+1}$. Khi bộ xử lý $P_{i,j}$ nhận các phần tử của ma trận $D^{(k)}$ từ một bộ xử lý dọc theo hàng của nó trong lưới logic, nó sẽ lưu trữ cục bộ ma trận và chuyển tiếp đến bộ xử lý có vị trí ở phía đối diện với bộ xử lý nơi nó nhận ma trận. Các cột cũng tuân theo giao thức truyền thông tin tương tự. Các phần tử của ma trận $D^{(k)}$ không được chuyển tiếp khi chúng đang ở vị trí biên của lưới.



Hình 2.12: Minh hoạ giao thức truyền thông tin cho các bộ xử lý trong một hàng (hoặc một cột)

Hình 2.12 minh họa giao thức truyền thông tin trong pipeline 2-D Block Mapping của thuật toán Floyd. Giả sử rằng bộ xử lý 4 tại thời điểm t vừa tính toán một đoạn của cột thứ k trong ma trận $D^{(k-1)}$. Nọ gửi các phân đoạn đến bộ xử lý 3 và 5. Các bộ xử lý này nhận phân đoạn tại thời điểm $t+1$ (đơn vị thời gian là thời gian để một phân đoạn ma trận di chuyển bằng giao thức truyền thông tin đến các quy trình liền kề). Tương tự, các bộ xử lý xa hơn so với bộ xử lý 4 sẽ nhận được phân đoạn sau. Bộ xử lý 1 (ở biên) sẽ không chuyển tiếp phân đoạn sau khi nhận được.

Tại mỗi bước, các phần tử của hàng đầu tiên được gửi từ bộ xử lý $P_{i,j}$ đến $P_{i+1,j}$. Tương tự như vậy, các phần tử của cột đầu tiên được gửi từ bộ xử lý $P_{i,j}$ đến $P_{i,j+1}$. Mỗi bước như vậy cần thời gian là $O(n/\sqrt{p})$. Sau $O(\sqrt{p})$ bước, bộ xử lý $P_{\sqrt{p},\sqrt{p}}$ nhận được các phần tử của hàng và cột đầu tiên trong thời gian $O(n)$. Các giá trị của hàng và cột kế tiếp theo sau thời gian $O(n^2/p)$. Do đó, quá trình kết thúc phần tính toán đường đi ngắn nhất trong khoảng thời gian $O(n^3/p) + O(n)$. Khi bộ xử lý $P_{\sqrt{p},\sqrt{p}}$ kết thúc lần lặp thứ $n-1$, nó sẽ gửi các giá trị liên quan của hàng và cột thứ n cho các bộ xử lý còn lại. Các giá trị này được gửi đến bộ xử lý $P_{1,1}$ trong thời gian $O(n)$. Do vậy, thời gian chạy song song là:

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n).}^{\text{communication}}$$

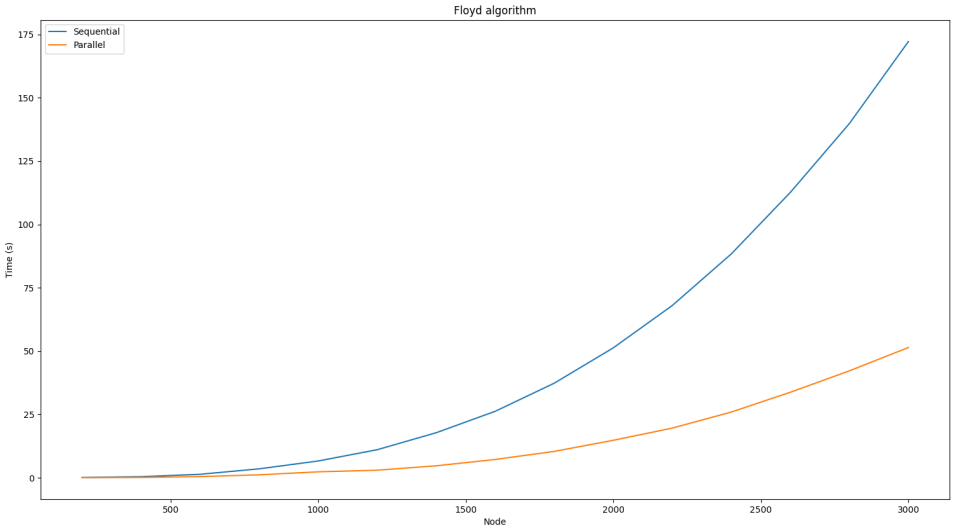
Do thời gian chạy tuần tự là $W = O(n^3)$, tốc độ và hiệu suất được cho theo công thức:

$$S = \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta(n)}$$

$$E = \frac{1}{1 + \Theta(p/n^2)}$$

Từ công thức trên, ta thấy rằng hàm tối ưu chi phí là $p/n^2 = O(1)$. Do vậy, công thức Pipeline của thuật toán Floys sử dụng lên đến $O(n^2)$ bộ xử lý để đảm bảo hiệu quả. Hàm đẳng tích do truyền thông tin là $O(p^{1.5})$, đây dùng là hàm đẳng tích tổng thể. So sánh công thức Pipeline và công thức đồng bộ 2-D Block Mapping, có thể thấy rằng công thức trước nhanh hơn đáng kể.
Thực nghiệm thuật toán Floyd tuần tự và song song được cho bởi hình 2.13

.



Hình 2.13: Thực nghiệm so sánh giữa thuật toán Floyd tuần tự và song song

2.4 Transitive Closure - Bao đóng truyền ứng

Trong một số bài toán thực tế, ta muốn xác định xem hai đỉnh có được kết nối với nhau hay không. Điều này thường được thực hiện xác định transitive closure của đồ thị. Xét đồ thị $G = (V, E)$, Transitive Closure của G được định nghĩa là đồ thị $G^* = (V, E^*)$, trong đó $E^* = \{(v_i, v_j) |$ tồn tại đường đi từ v_i đến v_j trong $G\}$. Ta xác định transitive closure của đồ thị bằng các xác định ma trận liên thông $A*$. Ma trận liên thông (*connectivity matrix*) của đồ thị G là ma trận $A^* = (a^*_{i,j})$ sao cho:

$$a^*_{i,j} = \begin{cases} 1 & \text{Nếu tồn tại đường đi giữa} v_i, v_j \\ \infty & \text{Otherwise} \end{cases}$$

Để xác định $A*$, ta chỉ định trọng số là 1 cho mỗi cạnh thuộc E, sử dụng các thuật toán xác định đường đi ngắn nhất giữa tất cả các cặp đỉnh trên đồ thị có trọng số này. Ma trận $A*$ có thể thu được từ ma trận D, trong đó D là ma trận kết quả thu được từ việc xác định đường đi ngắn nhất giữa các cặp đỉnh:

$$a^*_{i,j} = \begin{cases} 1 & \text{Nếu } d_{i,j} > 0 \text{ hoặc } i = j \\ \infty & \text{Nếu } d_{i,j} = \infty \end{cases}$$

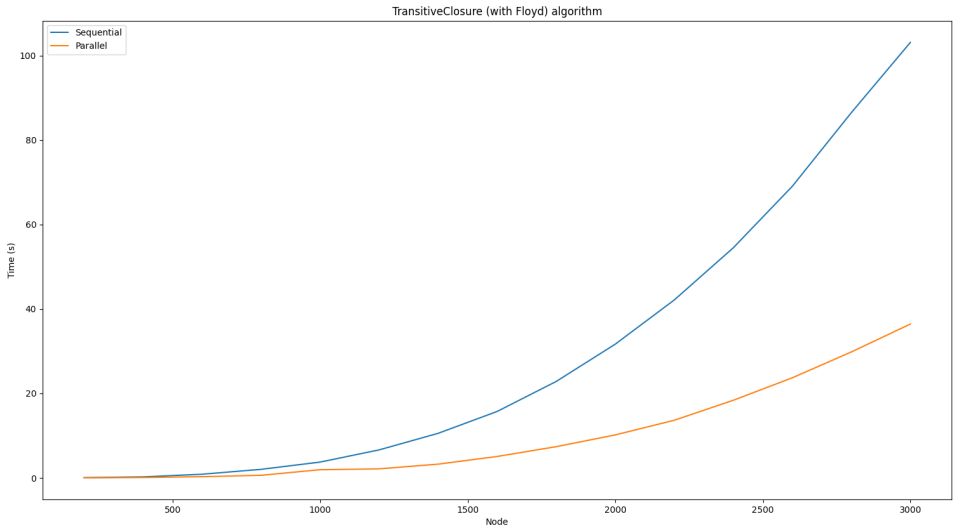
Một phương pháp khác để xác định $A*$ là sử dụng thuật toán Floyd trên ma trận kề $A = (a_{i,j})$ của G, trong đó:

$$a_{i,j} = \begin{cases} 1 & \text{Nếu } (v_i, v_j) \in E \text{ hoặc } i = j \\ 0 & \text{Otherwise} \end{cases}$$

Khi đó, ma trận A^* thu được bằng cách:

$$a^*_{i,j} = \begin{cases} \infty & \text{Nếu } d_{i,j} = 0 \\ 1 & \text{Otherwise} \end{cases}$$

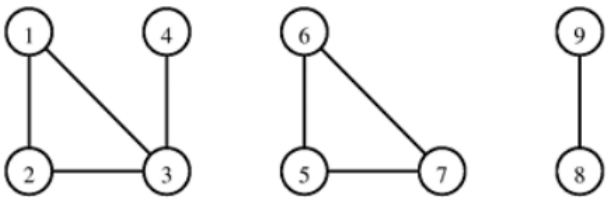
Độ phức tạp của việc xác định Transitive Closure là $O(n^3)$.
Kết quả thực nghiệm được cho bởi hình 2.14.



Hình 2.14: Thực nghiệm tuần tự và song song Transitive Closure

2.5 Các thành phần liên thông của đồ thị

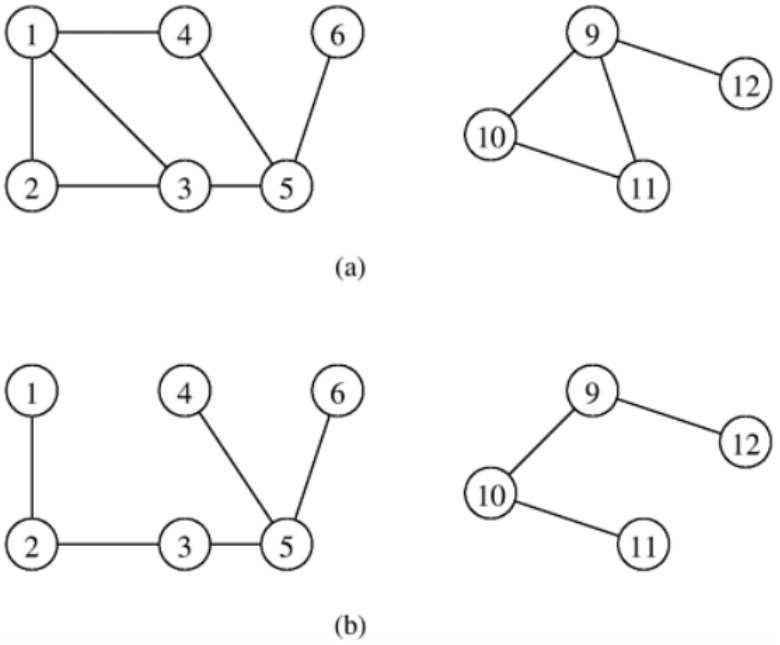
Thành phần liên thông của đồ thị vô hướng $G = (V, E)$ là các đồ thị con liên thông đôi một không có đỉnh chung $C_1, C_2, ..., C_k$ sao cho, $V = C_1 \cup C_2 \cup \cdots \cup C_k$, và $u, v \in C_i$ khi và chỉ khi có đường đi từ u đến v và ngược lại.



Hình 2.15: Đồ thị có 3 thành phần liên thông

2.5.1 Thuật toán tìm kiếm theo chiều sâu trên đồ thị

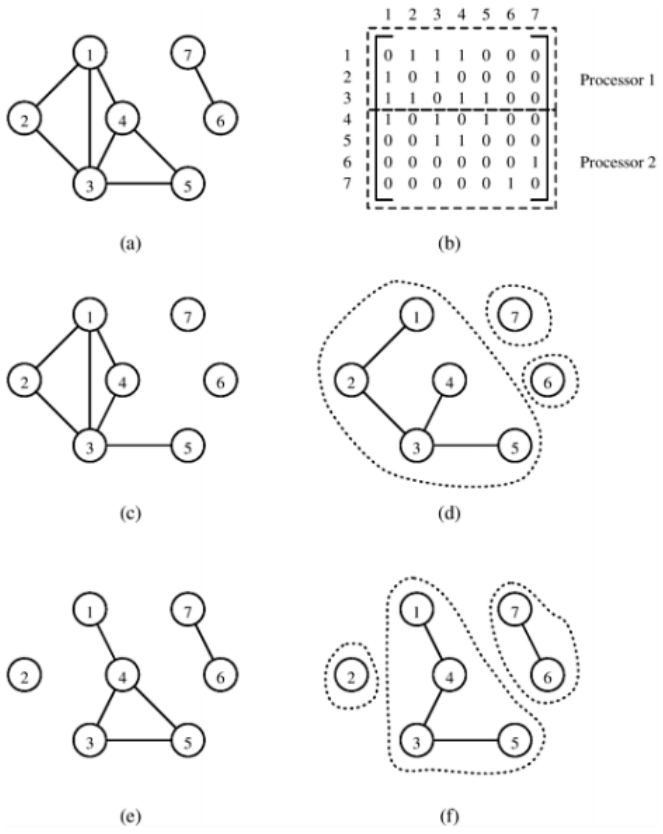
Bài toán tìm các thành phần liên thông của đồ thị là việc chúng ta cần xác định đồ thị gồm bao nhiêu thành phần liên thông và từng thành phần liên thông của nó là gồm những đỉnh nào. Do thủ tục DFS cho phép thăm tất cả các đỉnh thuộc một thành phần liên thông với s, nên số thành phần liên thông của đồ thị chính bằng số lần gọi thủ tục này. Kết quả của một lần duyệt này là một rừng gồm các cây, mỗi cây chứa các đỉnh thuộc một thành phần liên thông khác nhau. Hình 2.16 minh họa thuật toán này. Tính đúng đắn của thuật toán này dựa trên định nghĩa của cây khung. Độ phức tạp của thuật toán là $O(|E|)$.



Hình 2.16: Minh họa thuật toán DFS tìm các thành phần liên thông của đồ thị

Xây dựng thuật toán song song

Thuật toán xác định các thành phần liên thông của đồ thị có thể song song bằng cách phân vùng ma trận kề của G thành p phần và gán từng phần cho một trong p bộ xử lý. Mỗi bộ xử lý P_i có một đồ thị con G_i của G, trong đó $G_i = (V, E_i)$, E_i là các cạnh tương ứng với phần của ma trận kề được gán cho bộ xử lý này. Đầu tiên, mỗi bộ xử lý P_i xác định rừng khung của đồ thị G_i. Sau khi kết thúc ta thu được p rừng khung. Bước thứ hai, các rừng khung được hợp nhất theo từng cặp cho đến khi ta thu được 1 rừng khung. Khi đó, ta thu được rùng có đặc tính là hai đỉnh nằm trong cùng một thành phần liên thông G nếu chúng nằm trên cùng một cây. Hình 2.17 minh họa thuật toán này.



Hình 2.17: Tính toán song song xác định các thành phần liên thông. Ma trận kề của đồ thị G (a) được chia thành 2 phần (b). Mỗi bộ xử lý tương ứng với một đồ thị còn (c), (e). Mỗi bộ xử lý xác định rừng khung của đồ thị con, (d), (f).

Để hợp nhất các cặp rừng khung một cách hiệu quả, thuật toán sử dụng tập các cạnh không chung đỉnh (rời rạc). Giả sử rằng mỗi cây trong rừng khung của đồ thị con của G được đại diện bởi một tập hợp. Các tập cho các cây khác nhau là rời rạc từng cặp. Một số thao tác được định nghĩa trên các tập rời rạc:

- `find(x)` trả về con trỏ đến phần tử đại diện của tập hợp chứa x. Mỗi tập có phần tử đại diện.
- `union(x, y)` kết hợp các tập có chứa 2 phần tử x, y. Hai tập được cho là rời rạc, phân biệt trước khi kết hợp.

Các rừng khung được hợp nhất như sau: Gọi A, B là 2 rừng khung được hợp nhất. Đối với mỗi cạnh $(u, v) \in A$, một phép toán `find` được thực hiện cho mỗi đỉnh để xác định xem hai đỉnh u, v có nằm trên cùng một cây trong B chứa. Nếu không, 2 cây tương ứng của B chứa u, v được hợp nhất bởi phép toán `union`. Do vậy, việc hợp nhất A, B yêu cầu tối đa $2(n-1)$ phép toán `find`, $(n-1)$ phép toán `union`.

Câu hỏi đặt ra là làm thế nào để phân vùng ma trận kề của G và phân phối nó vào p bộ xử lý. Ta đề cập đến phương pháp 1-D Block Mapping. Ma trận kề kích thước $n \times n$ được chia thành p cột. Mỗi cột bao gồm n/p hàng liên tiếp. Để tính toán các thành phần liên thông, trước tiên mỗi bộ xử lý cần xác định rừng bao rùm cho đồ thị n đỉnh được biểu diễn bằng n/p hàng của ma trận kề được gán cho nó.

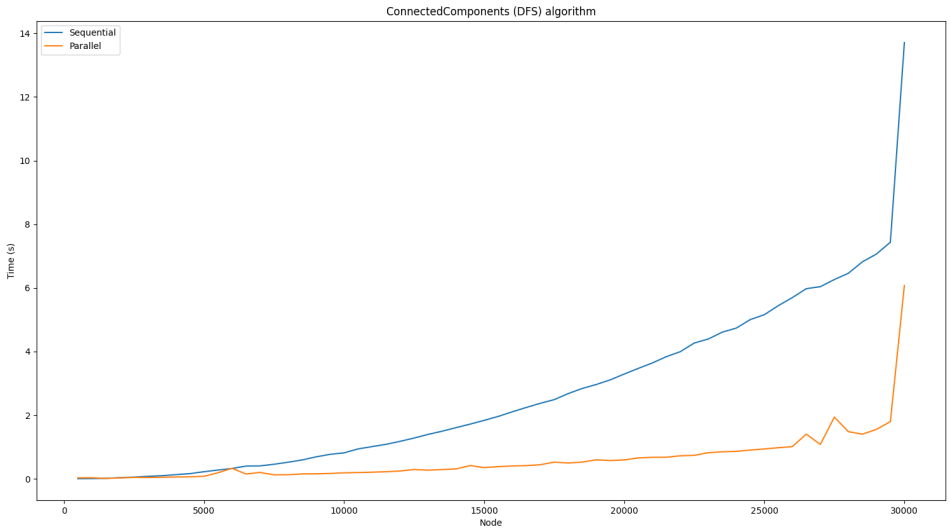
Xác định rừng khung dựa trên ma trận kề kích thước $(n/p) \times n$ được gán cho mỗi bộ xử lý cần thời gian là $O(n^2/p)$. Bước thứ 2 của thuật toán, hợp nhất từng cặp rừng khung. Chi phí hợp nhất là $O(n \log p)$. Chi phí truyền thông tin là $O(n \log p)$. Do vậy, độ phức tạp thời gian của thuật toán song song là:

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{local computation}} + \overbrace{\Theta(n \log p).}^{\text{forest merging}}$$

Do độ phức tạp thuật toán tuần tự là $W = O(n^2)$, tốc độ và hiệu suất được cho bởi công thức:

$$S = \frac{\Theta(n^2)}{\Theta(n^2/p) + \Theta(n \log p)}$$
$$E = \frac{1}{1 + \Theta((p \log p)/n)}$$

Từ công thức trên, ta thấy rằng hàm tối ưu chi phí là $p = O(n \log n)$. Hàm đẳng tích do truyền thông tin và tính toán hợp nhất là $O(p^2 \log^2 p)$. Hàm đẳng tích do tương tranh là $O(p^2)$. Do vậy, hàm đẳng tích tổng thể là $O(p^2 \log^2 p)$.
Thực nghiệm sử dụng DFS để tìm các thành phần liên thông của đồ thị được cho bởi hình 2.18.

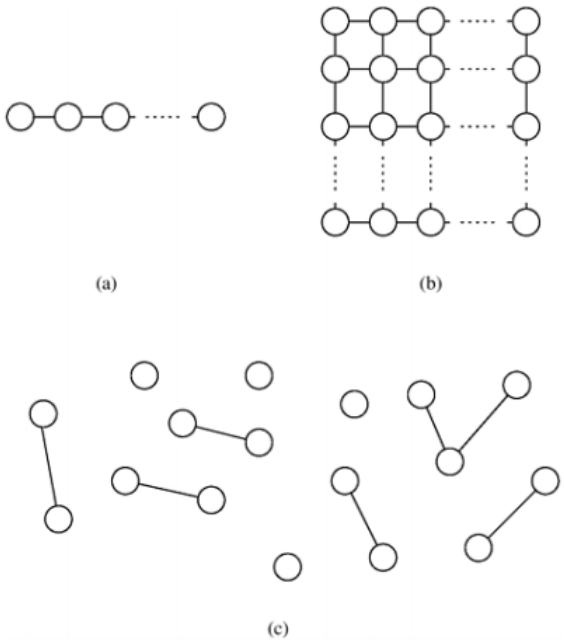


Hình 2.18: Thực nghiệm tuần tự và song song Transitive Closure

Chương 3

Các thuật toán với đồ thị thưa

Đồ thị $G = (V, E)$ được gọi là đồ thị thưa thớt (*sparse graph*) nếu $|E|$ nhỏ hơn nhiều so với $|V|^2$.

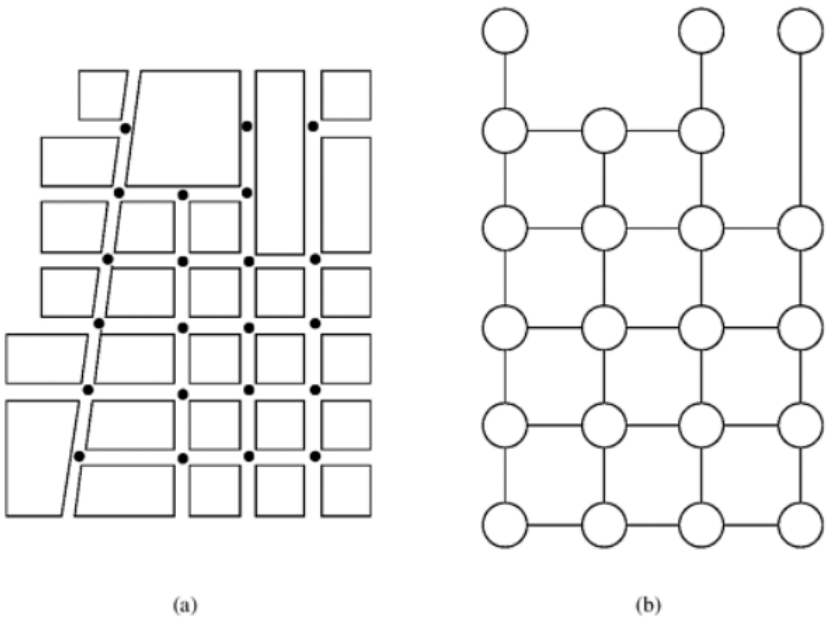


Hình 3.1: Một số ví dụ về đồ thị thưa thớt

Bất kỳ thuật toán cho đồ thị dày đặc đều có thể hoạt động tốt trên đồ thị thưa thớt, tuy nhiên, do tính thưa thớt của đồ thị, các thuật toán có hiệu suất tốt hơn đáng kể. Ví dụ, độ phức tạp thời gian của thuật toán Prim xác định cây bao trùm tối thiểu là $O(n^2)$ không phụ thuộc vào số cạnh. Nếu thuật toán Prim sử dụng danh sách kề và cấu trúc binary heap, độ phức tạp của thuật toán sẽ giảm xuống $O(|E| \log n)$. Có thể thấy, việc thay đổi khiến thuật toán vượt trội hơn thuật toán ban đầu miễn là $|E| = O(n^2/\log n)$. Một bước quan trọng trong việc triển khai các thuật toán của đồ thị thưa là sử dụng danh sách kề thay vì sử dụng ma trận kề. Có thể thấy, độ phức

tạp của các thuật toán sử dụng ma trận kề thường là $\Omega(n^2)$, độc lập với sô cạnh của đồ thị. Độ phức tạp của các thuật toán sử dựng danh sách kề thường là $\Omega(n + |E|)$, phụ thuộc vào sự thưa thớt của đồ thị.

Trong các công thức song song của các thuật toán cho đồ thị dày đặc, ta thu được hiệu suất tốt bằng cách phân vùng ma trần kề của đồ thị để mỗi bộ xử lý thực hiện một lượng tính toán và truyền thông tin bằng nhau. Tuy nhiên, rất khó để phân phối lượng tính toán và chi phí truyền thông tin cho đồ thị thưa thớt. Ta phải xem xét vấn đề phân vùng danh sách kề của đồ thị. Mỗi bộ xử lý được gán cho một phân vùng được chỉ định bởi một tập các đỉnh có số lượng bằng nhau và danh sách kề của chúng. Tuy nhiên, bậc của đỉnh trong đồ thị khác có thể khác nhau, do đó, một số bộ xử lý có thể được gán cho nhiều cạnh hơn so với các bộ xử lý khác. Điều đó dẫn đến sự mất cân bằng lượng tính toán giữa các bộ xử lý. Do vậy, rất khó để xác định các công thức song song hiệu quả cho các bài toán của đồ thị thưa thớt. Tuy nhiên, nếu đồ thị thưa thớt có cấu trúc nhất đỉnh, chúng ta thường có thể xác định được công thức song song hiệu quả. Ví dụ, xem xét bản đồ đường phố được biểu diễn trong hình 3.2. Dồ thị tương ứng với bản đồ là một đồ thị thưa thớt: bậc của đồ thị là 4.

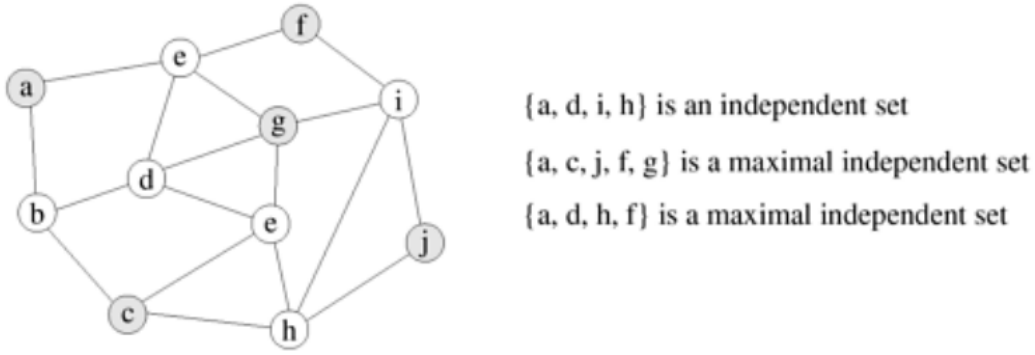


Hình 3.2: Bản đồ (a) được biểu diễn bằng đồ thị (b)

Hai phần tiếp theo trình bày các thuật toán hiệu quả để xác định tập độc lập cực đại và tìm đường đi ngắn nhất xuất phát từ một đỉnh.

3.1 Tìm tập độc lập cực đại

Xét đồ thị thưa vô hướng $G = (V, E)$, một tập đỉnh $I \subset V$ được gọi là độc lập nếu không tồn tại cạnh giữa hai cặp đỉnh nào trong I, hay hai đỉnh bất kỳ trong I là không kề nhau trên G. Một tập độc lập được gọi là cực đại nếu khi thêm một đỉnh mới bất kỳ vào I, tính độc lập không được bảo toàn. Chú ý rằng, trong một đồ thị, tập độc lập cực đại không phải là duy nhất.



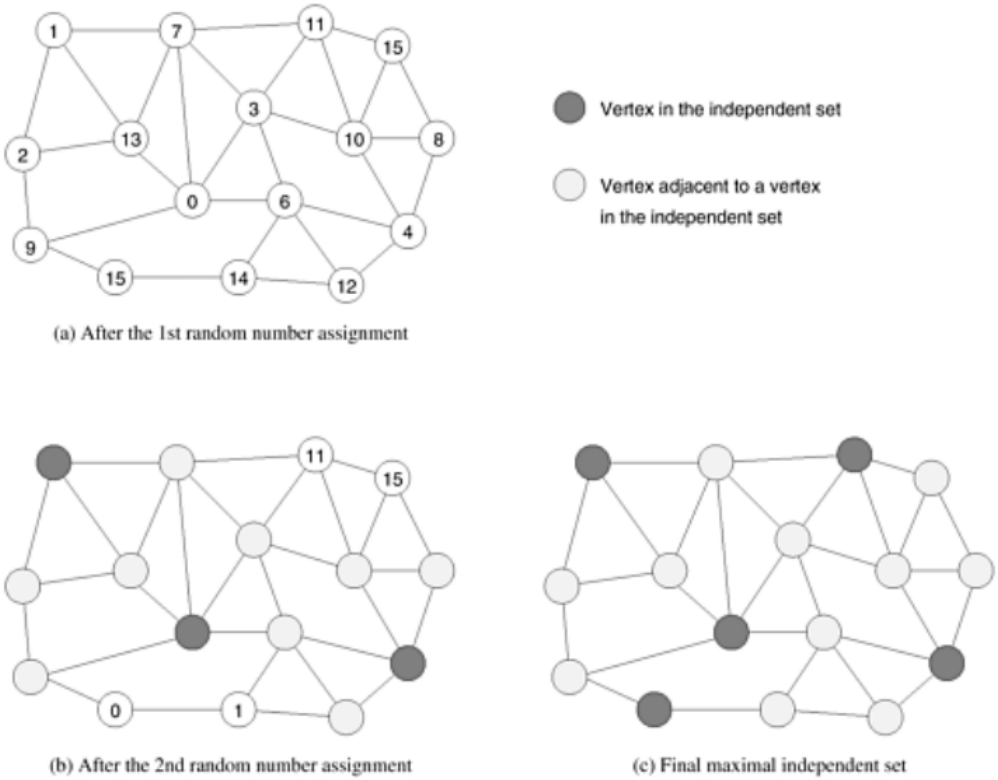
Hình 3.3: Ví dụ về một tập độc lập và tập độc lập cực đại của đồ thị

Nhiều thuật toán được đề xuất để xác định tập đỉnh của tập độc lập cực đại. Lớp các thuật toán đơn giản nhất bắt đầu bằng cách thiết lập I bằng rỗng, gán tất cả các đỉnh vào tập C - tập các đỉnh có thể đưa vào I. Sau đó, tiến hành thuật toán bằng cách liên tục thêm đỉnh v từ C vào I, sau đó xóa đỉnh kề v trong C. Thuật toán dừng khi C thành rỗng, trong trường hợp này, I là một tập độc lập cực đại. Khi đó, I chứa một tập các đỉnh độc lập, vì mỗi khi ta thêm một đỉnh vào I, ta xóa khỏi C tất cả các đỉnh kề với nó, như vậy, ta loại bỏ các đỉnh mà vi phạm điều kiện độc lập. Ngoài ra, kết quả thu được là cực đại, bởi vì khi thêm bất kỳ một đinh không thuộc I vào I thì đỉnh đó là một đỉnh kề với ít nhất một đỉnh thuộc I.

Mặc dù thuật toán trên có tư tưởng rất đơn gian, nhưng nó không phù hợp để xử lý song song vì các bước trong thuật toán là tuần tự. Vì lý do này, các thuật toán tìm tập độc lập cực đại song song thường dựa trên thuật toán ngẫu nhiên được triển khai bởi Luby cho bài toán tô màu đồ thị. Sử dụng thuật toán Luby, tập độc lập cực đại I của đồ thị $G = (V, E)$ được xác định như sau:

- Tập I được khởi tạo bằng rỗng, tập các đỉnh ứng cử viên $C = V$. Mỗi đỉnh thuộc C được gán cho một số ngẫu nhiên duy nhất.
- Nếu đỉnh v có giá trị nhỏ hơn tất cả giá trị của các đỉnh liền kề, v được thêm vào tập I. Cập nhật lại tập C bằng cách loại bỏ các đỉnh kề với đỉnh được thêm vào I.

Lưu ý rằng, các đỉnh được thêm vào I thực sự độc lập (nghĩa là không được kết nối trực tiếp qua một cạnh). Điều này là bởi vì, nếu v được thêm vào I, giá trị được gán cho v là giá trị nhỏ nhất trong các giá trị của các đỉnh liền kề v, do vậy, không có đỉnh u liền kề với v được chọn để thêm vào I. Thuật toán dừng khi $C = \emptyset$. Thuật toán hội tụ sau khoảng thời gian là $O(\log |V|)$.



Hình 3.4: Minh hoạ thuật toán Luby

Xây dựng thuật toán song song

Cho đồ thị vô hướng $G = (V, E)$, trong đó $V = \{v_1, v_2, \ldots, v_n\}$. Đặt I là một mảng có kích thước $|V| = n$, nếu đỉnh v_i thuộc tập độc lập cực đại thì $I[i] = 1$, ngược lại, $I[i] = 0$. Ban đầu, các phần tử của I được khởi tạo bằng 0. Đặt C là một mảng có kích thước $|V|$, nếu v_i là một phần của tập đỉnh ứng cử viên thì $C[i] = 1$, ngược lại, $C[i] = 0$. Ban đầu, các phần tử của C được khởi tạo bằng 1. Đặt R là mảng kích thước $|V|$, lưu các giá trị ngẫu nhiên của các đỉnh của đồ thị.

Trong mỗi lần lặp, tập C được phân chia thành các block ứng với p bộ xử lý. Mỗi bộ xử lý tạo ra một tập giá trị ngẫu nhiên gán cho các đỉnh thuộc block tương ứng, sau đó, bộ xử lý xác định đỉnh có thể được thêm vào tập độc lập. Bộ xử lý kiểm tra giá trị được gán cho đỉnh có nhỏ hơn so với giá trị của các đỉnh kề với nó hay không, nếu có, ta thiết lập các chỉ mục tương ứng trong các mảng I, C. Do mảng R được chia sẻ và có thể được truy cập bởi tất cả các bộ xử lý nên việc xác định có hay không một

đỉnh có thể được thêm vào tập độc lập là khá đơn giản.

Mảng C cũng có thể được cập nhật trong quá trình chạy song song. Mỗi bộ xử lý, sau khi xác định tồn tại một đỉnh v được thêm vào tập độc lập, ta sẽ thiết lập chỉ mục tương ứng với các đỉnh kề với v trong mảng C có giá trị bằng 0. Lưu ý rằng, việc cập nhật đồng thơi không ảnh hưởng đến tính đúng đắn của kết quả, do giá trị được ghi đồng thời là như nhau.

3.2 Đường đi ngắn nhất xuất phát từ một đỉnh

Một biến thể của thuật toán Dijkstra là thuật toán Johnson, hiệu quả với các đồ thị thưa thớt. Nhắc lại, trong mỗi lần lặp, thuật toán Dijkstra thực hiện hai bước sau: đầu tiên, chọn đỉnh $u \in (V - V_T)$ sao cho $l[u] = \min\{l[v]|v \in (V - V_T)\}$ và thêm nó vào tập V_T; thứ hai, với mỗi đỉnh $v \in (V - V_T)$, tính $l[v] = \min\{l[v], l[u] + w(u, v)\}$. Chú ý rằng, trong bước thứ hai, chỉ các đỉnh thuộc danh sách kề với đỉnh u được tính toán. Đối với đồ thị thưa thớt, số đỉnh kề đỉnh u nhỏ hơn đáng kể so với $O(n)$, do đó, bằng cách sử dụng danh sách kề sẽ cải thiện đáng kể hiệu suất.

Thuật toán Johnson sử dụng hàng đợi ưu tiên Q để lưu trữ giá trị $l[v]$ cho mỗi đỉnh $v \in (V - V_T)$. Hàng đợi ưu tiên được xây dựng sao cho đỉnh có giá trị nhỏ nhất trong l luôn nằm ở phía trước của hàng đợi. Một cách phổ biến để triển khai hàng đợi ưu tiên là phương pháp binary min-heap. Binary min-heap cho phép ta cập nhật giá trị $l[v] = \infty$ trong hàng đợi ưu tiên. Đối với đỉnh nguồn s, ta gán $l[s] = 0$. Ở mỗi bước của thuật toán, đỉnh $u \in (V - V_T)$ có giá trị tối thiểu trong l được xóa khỏi hàng đợi. Danh sách kề của đỉnh u được cập nhật, với mỗi cạnh (u, v), khoảng cách $l[v]$ của đỉnh v được cập nhật trên heap. Độ phức tạp thời gian của thuật toán phụ thuộc vào thời gian cập nhật đỉnh của hàng đợi. Tổng số lần cập nhật bằng số cạnh của đồ thị, do đó, độ phức tạp thời gian của thuật toán Johnson là $O(|E| \log n)$.

```
1.   procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.   begin
3.      Q := V ;
4.      for all v ∈ Q do
5.          l[v] := ∞ ;
6.      l[s] := 0;
7.      while Q ≠ 0 do
8.      begin
9.         u := extract min( Q);
10.        for each v ∈ Adj [u] do
11.           if v ∈ Q and l[u] + w(u, v) < l[v] then
12.               l[v] := l[u] + w(u, v);
13.     endwhile
14.  end JOHNSON_SINGLE_SOURCE_SP
```

Hình 3.5: Minh hoạ thuật toán Johnson tuần tự

Xây dựng thuật toán song song

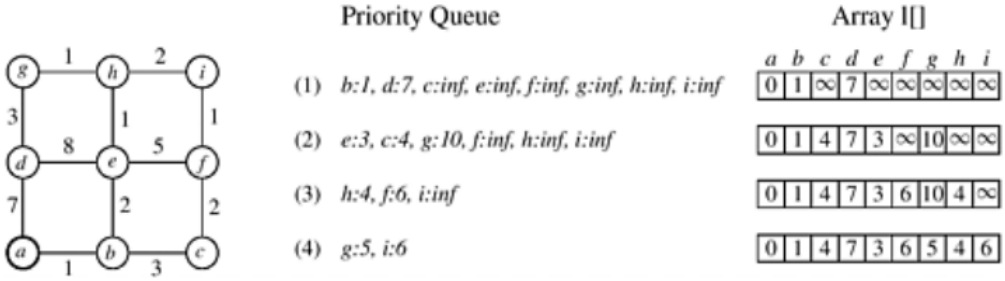
Một công thức song song hiệu quả cho thuật toán Johnson phải duy trì hàng đợi ưu tiên Q một cách hiệu quả. Một cách đơn giản là sử dụng 1 bộ xử lý, ví dụ, P_0 để duy trì Q, tất cả các bộ xử lý còn lại sẽ tính toán các giá trị mới của $l[v]$ cho $v \in (V - V_T)$, và cung cấp giá trị cho P_0 để cập nhật Q. Tuy nhiên, cách này tồn tại hai hạn chế. Trước tiên, do chỉ có 1 bộ xử lý có trách nhiệm duy trì mức độ ưu tiên của hàng đợi, nên độ phức tạp thời gian chạy song song là $O(|E| \log n)$ (có $O(|E|)$ số lần cập nhật hàng đợi, mỗi lần cập nhật cần $O(\log n)$). Thời gian chạy tuần tự của thuật toán là $O(|E| \log n)$, do vậy tốc độ của thuật toán không thay đổi. Thứ hai, trong mỗi lần lặp, thuật toán cập nhật khoảng $\frac{|E|}{|V|}$ đỉnh. Do vậy, không quá $\frac{|E|}{|V|}$ bộ xử lý cùng hoạt động trong cùng một thời điểm, rất nhỏ so với các lớp của đồ thị thưa thớt.

Hạn chế đầu tiên có thể được giảm bớt bằng cách phân phối việc duy trì hàng đợi ưu tiên cho nhiều bộ xử lý. Đây là một nhiệm vụ khó, và chỉ có thể được thực hiện một cách hiệu quả trên mô hình có độ trễ thấp, chẳng hạn như shared-address-space của máy tính. Tuy nhiên, ngay cả trong trường hợp tốt nhất, mỗi khi cập nhật hàng đợi ưu tiên chỉ mất thời gian $O(1)$, tốc độ tối đa có thể đạt được là $O(\log n)$, là một số khá nhỏ. Hạn chế thứ hai có thể được giảm bớt bằng cách tùy thuộc vào giá trị l của các đỉnh ở đầu hàng đợi ưu tiên, nhiều hơn một đỉnh được xóa khỏi hàng đợi cùng một lúc. Đặc biệt, nếu v là đỉnh ở đầu hàng đợi ưu tiên, xét tất cả các đỉnh u sao cho $l[u] = l[v]$ cũng có thể được xóa và danh sách kề tương ứng của chúng được xử lý đồng thời. Điều này là do các đỉnh có cùng khoảng cách tối thiểu đối với đỉnh nguồn có thể được xử lý theo bất kỳ thứ tự nào. Lưu ý rằng để phương pháp hoạt động, tất cả các đỉnh có cùng một khoảng cách tối thiểu với đỉnh gốc phải được xử lý trong lock-step. Nếu biết trọng số tối thiểu trên tất cả các cạnh của đồ thị là m, ta có thể xác định được mức độ bổ sung tương tranh. Trong trường hợp đó, tất cả các đỉnh u sao cho $l[u] \neq l[v] + m$ có thể được xử lý đồng thời, ta gọi các đỉnh như vậy là các đỉnh an toàn (*safe vertice*). Tuy nhiên, phương pháp này có thể dẫn đến tốc độ tốt hơn so với $O(\log n)$ chỉ khi có nhiều hơn một cập nhật hoạt động đồng thời trên hàng đợi ưu tiên.

Ở trên là những thảo luận tập trung vào việc phát triển công thức song song của thuật toán Johnson tìm đường đi ngắn nhất xuất phát từ một đỉnh. Tuy nhiên, như chúng ta đã thấy, các cách tiếp cận như vậy dẫn đến các thuật toán phức tạp và hạn chế tương tranh. Một cách tiếp cận khác là phát triển một thuật toán song song xử lý đồng thời cả hai đỉnh an toàn và không an toàn. Đặc biệt, trong thuật toán này, mỗi bộ xử lý trích xuất một trong p đỉnh đầu tiên và tiến hành cập nhật giá trị l của các đỉnh kề nó. Tất nhiên, vấn đề của cách tiếp cận này là nó không đảm bảo rằng giá trị l của các đỉnh được trích xuất từ hàng đợi ưu tiên tương ứng với chi phí của đường đi ngắn nhất. Ví dụ, xét hai đỉnh u, v nằm trên cùng của hàng đợi ưu tiên, với $l[v] < l[u]$. Theo thuật toán Johnson, tại thời điểm một đỉnh được trích xuất từ hàng đợi, giá trị l của nó là trọng số của đường đi ngắn nhất từ đỉnh nguồn đến đỉnh đó. Giả sử có một cạnh nối u và v, sao cho $l[v] + w(v, u) < l[u]$, thì giá trị chính xác của trọng số đường đi ngắn nhất đến u là $l[v] + w(v, u)$, không phải là $l[u]$. Tuy nhiên, tính chính xác của kết

quả có thể được đảm bảo bằng cách phát hiện khi ta đã tính sai đường đi ngắn nhất đến một đỉnh cụ thể và chèn nó trở lại hàng đợi ưu tiên với giá trị l được cập nhật. Có một số trường hợp cần được xem cét. Xét đỉnh v vừa được trích xuất từ hàng đợi và cho u là một đỉnh kề với v đã được trích xuất từ hàng đợi. Nếu $l[v] + w(v, u) < l[u]$, thì đường đi ngắn nhất đến u là không chính xác, khi đó, u được chèn trở lại hàng đợi ưu tiên với giá trị $l[u] = l[v] + w(v, u)$.

Để rõ cách tiếp cận của phương pháp này, ta xem xét ví dụ biểu đồ lưới được minh họa trong hình 3.6. Trong ví dụ này, có 3 bộ xử lý và yêu cầu là tìm đường đi ngắn nhất xuất phát từ đỉnh a. Sau khi khởi tạo hàng đợi ưu tiên, đỉnh b và đỉnh d là hai đỉnh kề với a, có thể được truy cập từ đỉnh nguồn. Trong bước đầu tiên, bộ xử lý P_0, P_1 và P_2 trích xuất các đỉnh e, c, g và tiến hành cập nhật giá trị l của các đỉnh liền kề với chúng. Lưu ý rằng, khi xử lý đỉnh e, bộ xử lý P_0 sẽ kiểm tra $l[e] + w(e, d)$ nhỏ hơn hay lớn hơn so với $l[d]$. Trong ví dụ cụ thể này, $l[e] + w(e, d) > l[d]$ thì chỉ ra rằng giá trị tính toán trước đó của đường đi ngắn nhất đến d không thay đổi khi e được xem xét và tất cả các tính toán cho đến hiện tại là chính xác. Trong bước thứ 3. bộ xử lý P_0 và P_1 làm việc tương ứng trên h và f. Bây giờ, khi bộ xử lý P_0 so sánh $l[h] + w(h, g) = 5$, trong khi $l[g] = 10$ được trích xuất trong lần cập nhật trước đó, có thể thấy rằng $l[h] + w(h, g) < l[g]$, khi đó g sẽ chèn trở lại hàng đợi và cập nhật giá trị $l[g] = l[h] + w(h, g)$. Cuối cùng, trong bước thứ 4, hai đỉnh còn lại được trích xuất từ hàng đợi ưu tiên và tất cả các đường đi ngắn nhất từ a được xác định.



Hình 3.6: Biểu đồ lưới

Công thức bộ nhớ phân tán - Distributed Memory Formulation

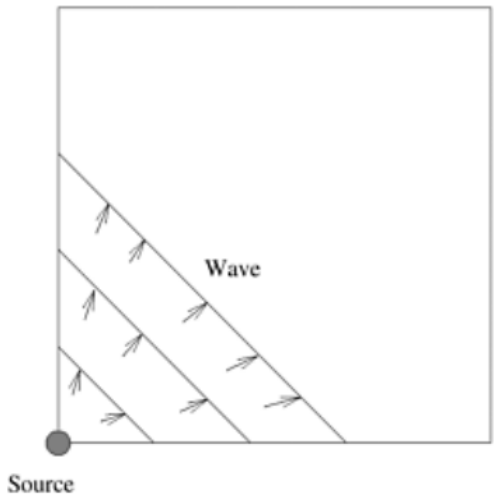
Đặt p là số bộ xử lý, $G = (V, E)$ là đồ thị thưa thớt. Ta chia tập các đỉnh V của đồ thị thành các tập rời rạc $V_1, V_2, \ldots, V_p$, gán mỗi tập các đỉnh và danh sách kề liên quan cho một trong p bộ xử lý. Mỗi bộ xử lý duy trì một hàng đợi ưu tiên cho các đỉnh được gán với nó, và tính đường đi ngắn nhất từ đỉnh nguồn đến các đỉnh này. Vì vậy, hàng đợi ưu tiên Q được chia thành p hàng đợi rời tạc $Q_1, Q_2, \ldots, Q_p$, mỗi hàng đợi được gán với một bộ xử lý. Ngoài ra, mỗi bộ xử lý P_i cũng duy trì một mảng $sp[v]$ lưu trữ chi phí của đường đi ngắn nhất từ đỉnh nguồn đến $v \in V_i$. Chi phí $sp[v]$ được cập nhật cho $l[v]$ mỗi khi đỉnh v được trích xuất từ hàng đợi. Ban đầu, ta khởi tạo $sp[v] = \infty$ cho mỗi đỉnh v khác đỉnh nguồn, sau đó chèn $l[s]$ vào hàng đợi ưu tiên (s

là đỉnh nguồn). Mỗi bộ xử lý thực hiện thuật toán Johnson trên hàng đợi của nó. Kết thúc thuật toán, $sp[v]$ lưu trữ chi phí đường đi nhấn nhất từ đỉnh nguồn s đến đỉnh v.

Khi bộ xử lý P_i trích xuất đỉnh $u \in V_i$ có giá trị $l[u]$ nhỏ nhất từ Q_i, giá trị l của các đỉnh được gán cho các bộ xử lý khác P_i có thể cần được cập nhật. Bộ xử lý P_i tuyền thông tin đến các bộ xử lý lưu trữ các đỉnh kề với u, thông báo với chúng về các giá trị mới. Sau khi nhận được giá trị này, bộ xử lý cập nhật các giá trị của l. Ví dụ, giả sử rằng có một cạnh (u, v) sao cho $u \in V_i$, $v \in V_j$, bộ xử lý P_i vừa trích xuất đỉnh u từ hàng đợi, sau đó P_i truyền thông tin đến P_j chứa giá trị mới tiềm năng $l[v]$, là $l[u] + w(u, v)$. Bộ xử lý P_j nhận được thông báo, đặt giá trị của $l[v]$ được lưu trữ trong hàng đợi thành $\min\{l[v], l[u] + w(u, v)\}$.

Do cả hai bộ xử lý P_i, P_j đều thực hiện thuật toán Johnson, nên có thể P_j đã trích xuất đỉnh v từ hàng đợi của nó. Điều này có nghĩa là bộ xử lý P_j có thể đã xác định chi phí đường đi ngắn nhất $sp[v]$ từ đỉnh nguồn đến v. Khi đó có hai trường hợp có thể xảy ra: hoặc $sp[v] \leq l[u] + w(u, v)$, hoặc $sp[v] > l[u] + w(u, v)$. Với trường hợp đầu tiên, khi đó tồn tại đường đi dài hơn đến đỉnh v đi qua u, và trường hợp thứ hai là tồn tài đường đi ngắn hơn đến đỉnh v đi qua u. Đối với trường hợp đầu tiên, bộ xử lý P_j không cần phải làm gì, vì đường đi ngắn nhất đến v không thay đổi. Đối với trường hợp thứ hai, bộ xử lý P_j phải cập nhật chi phí đường đi ngắn nhất đến đỉnh v. Điều này được thực hiện bằng cách chèn lại đỉnh v vào hàng đợi và cập nhật giá trị $l[v] = l[u] + w(u, v)$ và bỏ qua giá trị $sp[v]$. Vì đỉnh v có thể được chèn lại hàng đợi, nên thuật toán chỉ dừng khi tất cả hàng đợi bằng rỗng.

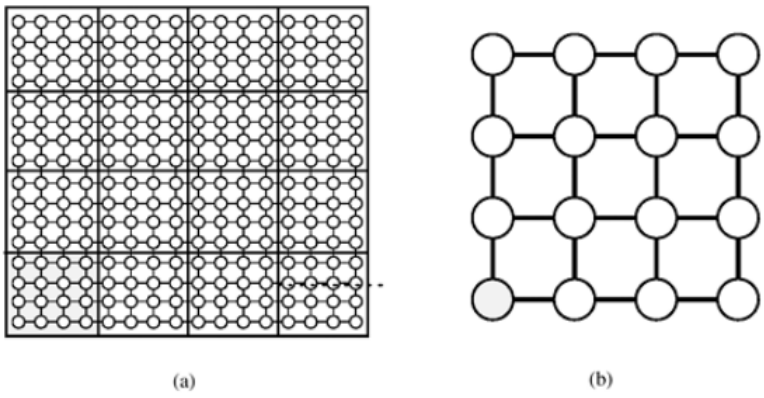
Ban đầu, chỉ có hàng đợi ưu tiên của bộ xử lý với đỉnh nguồn là không rỗng. Sau đó, hàng đợi ưu tiên của các bộ xử lý khác trở nên phổ biến khi các thông báo chứa các giá trị l mới được tạo và gửi đến bộ xử lý liền kề. Khi các bộ xử lý nhận giá trị l mới, chúng sẽ chèn vào hàng đợi ưu tiên của chúng và thực hiện các tính toán. Ta xem xét vấn đề tính toán các đường đi ngắn nhất xuất phát từ một đỉnh trong biểu đồ lưới nơi nguồn ở góc dưới bên trái. Khi đó, các phép tính toán truyền trên biểu đồ lưới dưới dạng sóng. Bộ xử lý không hoạt động trước khi sóng đến và hoạt động trở lại sau khi sóng qua. Quá trình này được minh họa trong hình 3.7. Tại bất kỳ thời điểm nào trong quá trình thực thi thuật toán, chỉ có các bộ xử lý dọc theo sóng là hoạt động. Các bộ xử lý khác đã hoàn thành tính toán hoặc trong trạng thái chưa khởi động.



Hình 3.7: Sóng hoạt động trong hàng đợi ưu tiên

Trong phần tiếp theo, nhóm sẽ tháo luận về 3 ánh xạ của biểu đồ lưới vào p-process mesh.

Một cách để ánh xạ đồ thị dạng lưới vào p-process mesh là sử dụng **2-D Block Mapping**. Cụ thể, chúng ta có thể xem các bộ xử lý p như một lưới logic và gán một block các đỉnh khác nhau cho mỗi bộ xử lý.



Hình 3.8: Ánh xạ đồ thị dạng lưới (a) và (b) sử dụng 2-D Block Mapping. Trong trường hợp này, $n = 16, \sqrt{p} = 4$

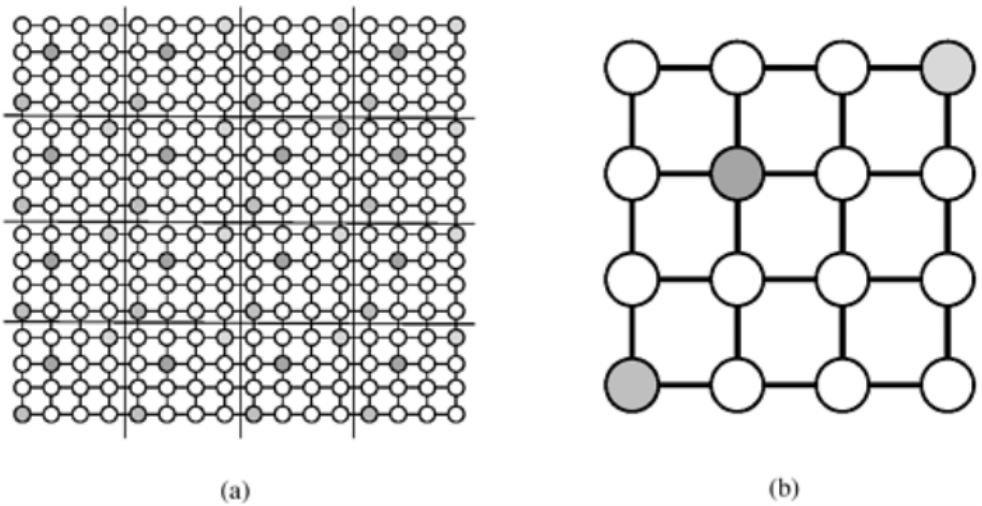
Tại mọi thời điểm, số bộ xử lý hoạt động bằng với số bộ xử lý giao nhau bởi sóng. Do sóng di chuyển theo đường chéo, nên không có nhiều hơn $O(\sqrt{p})$ bộ xử lý hoạt động cùng một thời điểm. Đặt W là độ phức tạp thuật toán tuần tự. Nếu ta giả sử rằng, tại bất kỳ thời điểm nào, các bộ xử lý đang thực hiện tính toán, nếu ta bỏ qua chi phí do truyền thông tin giữa các bộ xử lý và cộng việc bổ sung, thì tốc độ và hiểu suất tối đa

được biểu diễn bằng công thức:

$$S = \frac{W}{W/\sqrt{p}} = \sqrt{p}$$
$$E = \frac{1}{\sqrt{p}}$$

Hiệu quả của ánh xạ này kém, và trở nên kém hơn khi số bộ xử lý tăng lên.

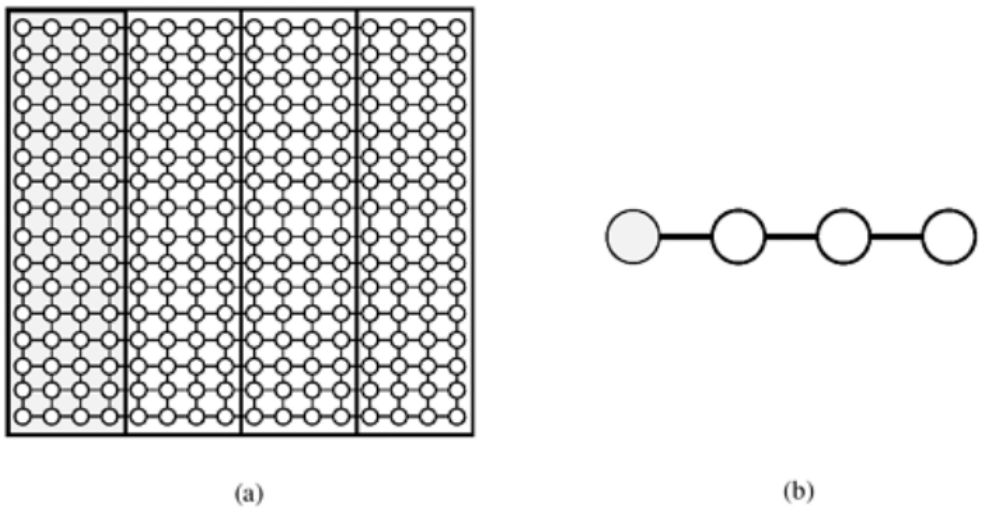
Giới hạn chính của 2-D Block Mapping là mỗi bộ xử lý chỉ chịu trách nhiệm cho một diện tích nhỏ, hạn chế của lưới. Ngoài ra, ta có thể thực hiện từng bộ xử lý chịu trách nhiệm về các khu vực rải rác của lưới bằng cách sử dụng **2-D Cyclic Mapping**. Điều này làm tăng thời gian khi một bộ xử lý đang hoạt động. Trong 2-D Cyclic Mapping, đồ thị dạng lưới kích thước $n \times n$ được chia thành n^2/p block, mỗi block có kích thước $\sqrt{p} \times \sqrt{p}$. Mỗi block ánh xạ vào $\sqrt{p} \times \sqrt{p}$ lưới bộ xử lý. Mỗi bộ xử lý chứa một block có n^2/p đỉnh. Các đỉnh này thuộc các đường chéo của đồ thị. Mỗi bộ xử lý được chỉ định các đường chéo như vậy.



Hình 3.9: Ánh xạ đồ thị dạng lưới (a) và (b) sử dụng 2-D Cyclic Mapping. Trong trường hợp này, $n = 16$, $\sqrt{p} = 4$

Mỗi bộ xử lý chịu trách nhiệm về các đỉnh thuộc các phần khác nhau của đồ thị. Khi sóng truyền qua đồ thị, sóng cắt một số đỉnh trên mỗi bộ xử lý. Do vậy, các bộ xử lý vẫn hoạt động đối với hầu hết các thuật toán. Tuy nhiên, 2-D Cyclic Mapping chịu chi phí truyền thông tin cao hơn so với 2-D Block Mapping, do các đỉnh kề nằm trên các bộ xử lý khác nhau, nên mỗi khi bộ xử lý trích xuất đỉnh u từ hàng đợi, bộ xử lý đó phải truyền thông tin đến các bộ xử lý khác về giá trị mới của $l[u]$.

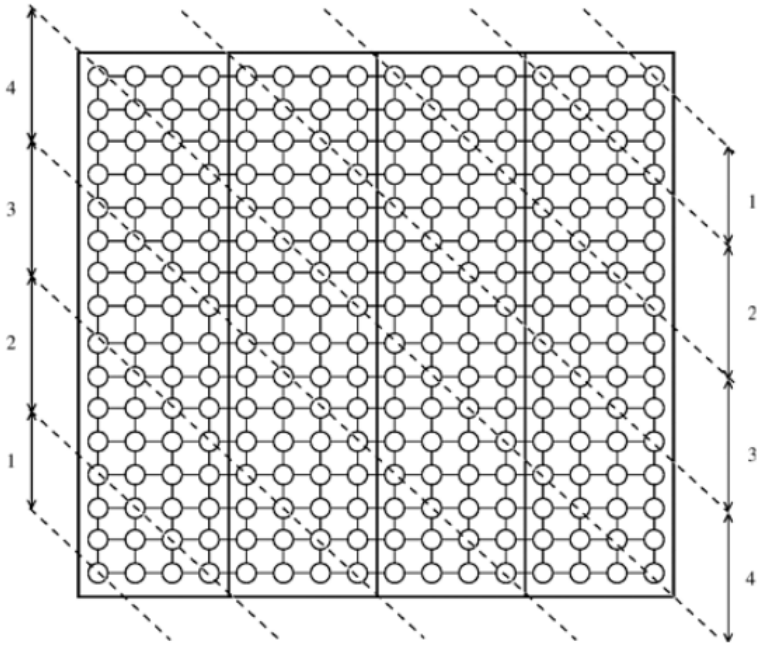
Hai ánh xạ trên đều có những hạn chế. 2-D Block Mapping không thể hoạt động cùng lúc nhiều hơn $O(\sqrt{p})$ bộ xử lý, 2-D Cyclic Mapping có chi phí truyền thông tin cao. Một ánh xạ khác coi các bộ xử lý như một mảng tuyến tính, gán n/p sọc của đồ thị dạng lưới cho mỗi bộ xử lý bằng cách sử dụng **1-D Block Mapping**.



Hình 3.10: 1-D Block Mapping

Ban đầu, sóng chỉ giao nhau trong một bộ xử lý. Trong quá trình tính toán, sóng tràn sang bộ xử lý thứ 2 để có 2 bộ xử lý hoạt động. Khi thuật toán tiếp tục, sóng sẽ giao nhau với nhiều bộ xử lý hơn, các bộ xử lý đều hoạt động. Quá trình này tiếp tục cho đến khi tất cả p bộ xử lý đều bận (nghĩa là cho đến khi tất cả bộ xử lý được giao nhau bởi sóng). Sau thời điểm này, số bộ xử lý hoạt động sẽ giảm dần. Hình 3.11 minh họa sự lan truyền của sóng. Nếu giả định rằng sóng truyền với tốc độ không đổi, thì trung bình $p/2$ bộ xử lý đang hoạt động. Bỏ qua bất kỳ chi phí phát sinh, tốc độ và hiệu suất của ánh xạ này được biểu diễn bằng công thức:

$$S = \frac{W}{W/(p/2)} = \frac{p}{2}$$
$$E = \frac{1}{2}$$



Hình 3.11: Số bộ xử lý hoạt động khi sóng truyền trên biểu đồ lưới

Do đó, hiệu suất của ánh xạ này tối đa là 50%. 1-D Block Mapping tốt hơn đáng kể so với 2-D Block Mapping nhưng không thể sử dụng nhiều hơn $O(n)$ bộ xử lý.

Kết luận

Như vậy, bài báo cáo đã giới thiệu tổng quan về các thuật toán song song và áp dụng cho một số bài toán về đò thị. Báo cáo cũng đã khái quát các khái niệm về lý thuyết đồ thị, các mô hình song song, cách biểu diễn đồ thị trên máy trính, các vấn đề về đường đi, cây bao trùm và thành phần liên thông trên đồ thị.

Từ các hiểu biết trên báo cáo đã tìm tòi, nghiên cứu để song song hóa các thuật toán tìm đường đi ngắn nhất (Dijsktra, Floyd), cây bao trùm nhỏ nhất (Prim), tập độc lập cực đại (Luby).

Kiến thức còn rất mở, do vậy, dù đã cố gắng nhưng bài báo cáo còn nhiều thiếu sót, nhóm sẽ bổ sung thêm trong tương lai. Một lần nữa, nhóm xin cảm ơn thầy giáo đã giúp đỡ, cũng như có những nhận xét sâu sắc để bài báo cáo của nhóm được hoàn thiện hơn.

Mã nguồn: https://github.com/nducthang/Parallel-algorithms-on-graphs

Tài liệu tham khảo

[1] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing (2003)

[2] N. D. Nghĩa, N. T. Thành, Toán rời rạc (2003)