

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN

=====*



BÁO CÁO BTL THUỘC HỌC PHẦN:
THỰC TẬP CƠ SỞ NGÀNH

NGHIÊN CỨU CƠ SỞ LÝ THUYẾT, ỨNG DỤNG VÀ CÀI ĐẶT ÍT NHẤT 2
THUẬT TOÁN ĐỂ GIẢI BÀI TOÁN PHÂN CÔNG CÔNG VIỆC (JOB
ASSINGMENT PROBLEM)

GVHD:	TS Nguyễn Thị Mỹ Bình	
Nhóm - Lớp:	11 - 20241IT6040003	
Thành viên:	Nguyễn Đình Hội	2022603030
	Lê Thị Ngọc Lan	2022602329
	Trần Song Hà	2022602338
	Nguyễn Tiến Hiền	2022602039
	Bùi Duy Khánh	2022601170

MỤC LỤC

LỜI MỞ ĐẦU.....	5
CHƯƠNG 1. CƠ SỞ LÝ THUYẾT	7
1.1. Cơ sở lý thuyết trong bài toán phân công công việc	7
1.1.1. Lý thuyết về Cây, Cây nhị phân và Cây tìm kiếm nhị phân.....	7
1.1.2. Phát biểu bài toán	11
1.1.3. Ứng dụng bài toán	11
1.2. Cơ sở lý thuyết của các thuật toán.....	14
1.2.1. Các khái niệm xung quanh thuật toán và đánh giá hiệu năng thuật toán	14
1.2.2. Cơ sở lý thuyết của thuật toán tham lam	21
1.2.3. Cơ sở lý thuyết của thuật toán nhánh cận.....	23
CHƯƠNG 2. THIẾT KẾ THUẬT TOÁN ĐỂ GIẢI BÀI TOÁN PHÂN CÔNG CÔNG VIỆC.....	28
2.1. Thiết kế thuật toán tham lam (Greedy).....	28
2.1.1. Thuật toán tham lam trong bài toán phân công công việc.....	28
2.1.2. Các bước giải thuật.....	28
2.2. Thiết kế thuật toán nhánh cận (Branch and Bound).....	31
2.2.1. Thiết kế thuật toán nhánh cận.....	31
2.3. Đánh giá hiệu quả các phương pháp.....	34
2.3.1. Phương pháp sử dụng thuật toán tham lam (Greedy).....	34
2.3.2. Phương pháp sử dụng thuật toán nhánh cận (Branch and Bound)	35
CHƯƠNG 3. CÀI ĐẶT VÀ KIỂM THỬ	37
3.1. Cài đặt thuật toán.....	37
3.1.1. Môi trường cài đặt	37
3.1.2. Sơ lược về chương trình	37
3.1.1. Cài đặt thuật toán tham lam.....	44

3.1.2. Cài đặt thuật toán nhánh cận.....	46
3.2. Kiểm thử	49
3.2.1. Kiểm thử thuật toán tham lam	51
3.2.2. Kiểm thử thuật toán nhánh cận.....	52
TỔNG KẾT	58
TÀI LIỆU THAM KHẢO	59

MỤC LỤC ẢNH

Hình 1. Ví dụ về các thành phần của 1 cây	7
Hình 2. Ví dụ biểu diễn cây	8
Hình 3. Ví dụ về cây con trái và cây con phải.....	9
Hình 4. Ví dụ về cây nhị phân	10
Hình 5. Nhân viên trong phòng ban phân công làm việc	12
Hình 6. Máy móc hoạt động trong nhà máy sản xuất.....	12
Hình 7. Công nhân làm việc trong các khu công nghiệp.....	13
Hình 8. Xe tải được phân công chở hàng	13
Hình 9. Minh hoạ về việc biểu diễn thuật toán bằng ngôn ngữ.....	15
Hình 10. Các kí hiệu của 1 lưu đồ thuật toán	16
Hình 11. Đây là đoạn mã giả miêu tả cách thức hoạt động của thuật toán tham lam (Greedy Algorithm) trong bài toán phân công công việc.	17
Hình 12. Ví dụ minh hoạ về độ phức tạp của thuật toán	19
Hình 13. Lưu đồ thuật toán tham lam.....	22
Hình 14. Lưu đồ thuật toán của thuật toán nhánh cận	26
Hình 15. Ví dụ về bài toán người du lịch	27
Hình 16. Lưu đồ thuật toán tham lam.....	30
Hình 17. Mã giả thuật toán tham lam	30
Hình 18. Lưu đồ thuật toán nhánh cận	33
Hình 19. Mã giả thuật toán nhánh cận.....	34
Hình 20. Minh hoạ việc thêm thư viện trong dự án Java	37
Hình 21. Minh hoạ khi chưa phân chia công việc	53
Hình 22. Minh hoạ khi công nhân 1 chọn việc.....	54
Hình 23. Minh hoạ khi công nhân 2 chọn việc.....	54
Hình 24. Minh hoạ khi công nhân 3 chọn việc.....	55

Hình 25. Minh hoạ khi công nhân 4 chọn việc.....	55
Hình 26. Minh hoạ khi công nhân 3 chọn công việc khác	56
Hình 27. Minh hoạ kết thúc quá trình chọn công việc	56

LỜI MỞ ĐẦU

Hiện nay, trong bối cảnh nền công nghiệp hoá hiện đại hoá ngày càng phát triển, việc quản lý lực lượng lao động là một vấn đề cần phải quan tâm. Tỷ lệ thất nghiệp của thanh niên năm 2023 tại Việt Nam là 13%, tương đương với 64,9 triệu người, là mức thấp nhất trong 15 năm qua. Để giảm được tỷ lệ thất nghiệp, chúng ta phải giải quyết vấn đề cốt lõi là phân công công việc. Vậy nên, việc quản lý và phân công lực lượng, công việc là một phần không thể thiếu trong lao động sản xuất.

Bài toán phân công công việc (Job Assignment Problem) là một dạng bài toán tối ưu hoá. Bài toán thường được sử dụng để phân công một nhóm người thực hiện một số công việc nhất định sao cho tổng chi phí thực hiện các công việc là nhỏ nhất (thường chi phí đó có thể là thời gian, năng suất, ...).

Chúng em mong muốn đáp ứng được nhu cầu của người dùng: tối ưu hoá lực lượng lao động, trang thiết bị, nguyên vật liệu, ngoài ra còn giảm chi phí, thời gian sản xuất dẫn tới tăng hiệu suất công việc. Và đề tài này ứng dụng rất nhiều trong đời sống, quản lý và dịch vụ. Nó khả thi trong việc áp dụng vào mọi lĩnh vực, ngành nghề và có thể nhanh chóng đưa ra cách giải quyết vấn đề. Ví dụ như phân công công việc cho các công nhân trong dây chuyền sản xuất để tối ưu hoá năng suất, hay là phân công nhân viên vào các nhiệm vụ khác nhau trong một dự án để hoàn thành dự án nhanh nhất có thể, ...

Hiểu được tầm quan trọng của việc phân công công việc, chúng em đã quyết định chọn đề tài nghiên cứu là: “Nghiên cứu cơ sở lý thuyết, ứng dụng và cài đặt ít nhất 2 thuật toán để giải bài toán Phân công công việc” để nghiên cứu và phát triển. Hai thuật toán chúng em lựa chọn là thuật toán Tham lam và thuật toán Nhánh cận. Cả 2 thuật toán đều là những thuật toán điển hình để giải những bài toán tối ưu hoá, giúp chúng em có thể giải được bài toán đã đề ra.

Mục tiêu được đề ra khi nghiên cứu đề tài là các thành viên trong nhóm là hiểu rõ được cơ sở lý thuyết của bài toán phân công công việc, thuật toán Tham lam và thuật toán Nhánh cận. Từ đó các thành viên có thể cài đặt chương trình để giải bài toán đó bằng ngôn ngữ lập trình phù hợp. Phạm vi nghiên cứu đề tài là giải quyết các bài toán phân công công việc được áp dụng trong thực tế và cuộc sống hằng ngày.

Chúng em xin gửi lời cảm ơn chân thành đến cô Nguyễn Thị Mỹ Bình đã tâm huyết giúp đỡ, hướng dẫn chúng em trong quá trình học tập học phần Thực tập cơ sở ngành. Cô đã giúp chúng em tích lũy được nhiều kiến thức để có thể hoàn thành được bài báo cáo đề tài này.

Trong quá trình thực hiện bài báo cáo, do hiểu biết của chúng em còn hạn chế, khó tránh khỏi những thiếu sót. Chúng em rất mong nhận được những lời góp ý của thầy cô để bài báo cáo ngày càng hoàn thiện hơn.

Chúng em xin chân thành cảm ơn!

CHƯƠNG 1. CƠ SỞ LÝ THUYẾT

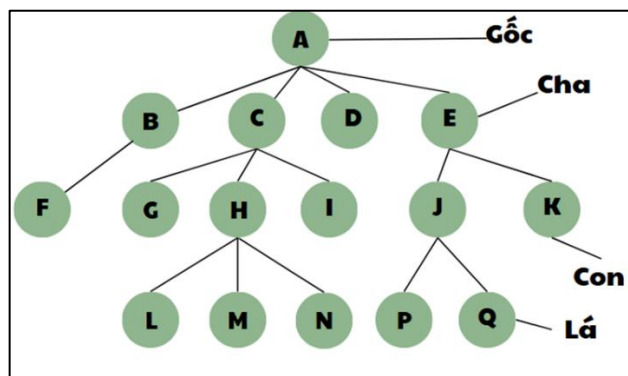
1.1. Cơ sở lý thuyết trong bài toán phân công công việc

1.1.1. Lý thuyết về Cây, Cây nhị phân và Cây tìm kiếm nhị phân

1.1.1.1. Cây và các khái niệm liên quan

Định nghĩa cây

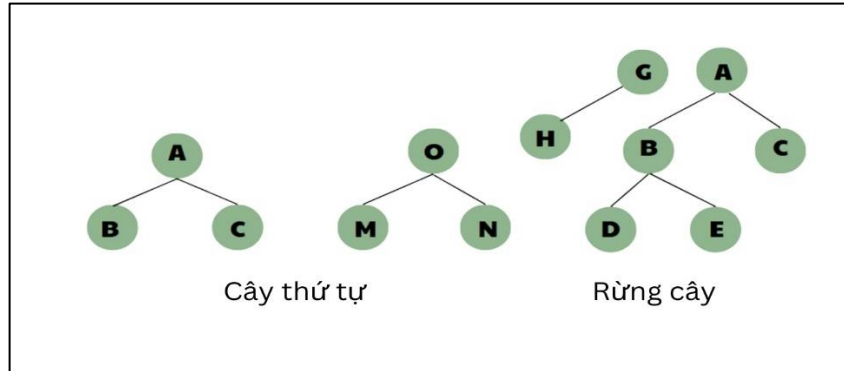
- Định nghĩa 1: Cây là một đồ thị liên thông không có chu trình.
- Định nghĩa 2: Một cây là tập hợp hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (Root). Giữa các nút có mối quan hệ phân cấp gọi là quan hệ cha- con.



Hình 1. Ví dụ về các thành phần của 1 cây

- Bậc của một nút: Là số nút con của nút đó.
- Bậc của một cây: Là bậc của nút có bậc lớn nhất trên cây đó. Cây có bậc n thì gọi là cây n – phân.
- Nút gốc: Là nút đặc biệt, không có nút cha –
- Nút lá: Là nút có bậc bằng 0 (không có nút con). –
- Nút nhánh: Là nút có bậc khác 0 và không phải là nút gốc.
- Mức của một nút .
 - + Gốc có mức 1. .
 - + Nếu nút cha có mức i thì các nút con có mức i+1.
- Chiều cao của cây: Là mức của nút có mức lớn nhất có trên cây.
- Đường đi: Dãy các nút N_1, N_2, \dots, N_k , được gọi là đường đi nếu N_i là cha của N_{i+1} ($1 \leq i \leq k-1$). .
- Độ dài của đường đi: Là số nút trên đường đi trừ đi 1.
- Cây con: Là cây có gốc là một nút nhánh, lá.

- Cây được sắp thứ tự: Các nút được sắp theo một thứ tự nhất định.
- Rừng: Là tập hợp hữu hạn các cây phân biệt.
- Cây rỗng: Cây không có nút nào.

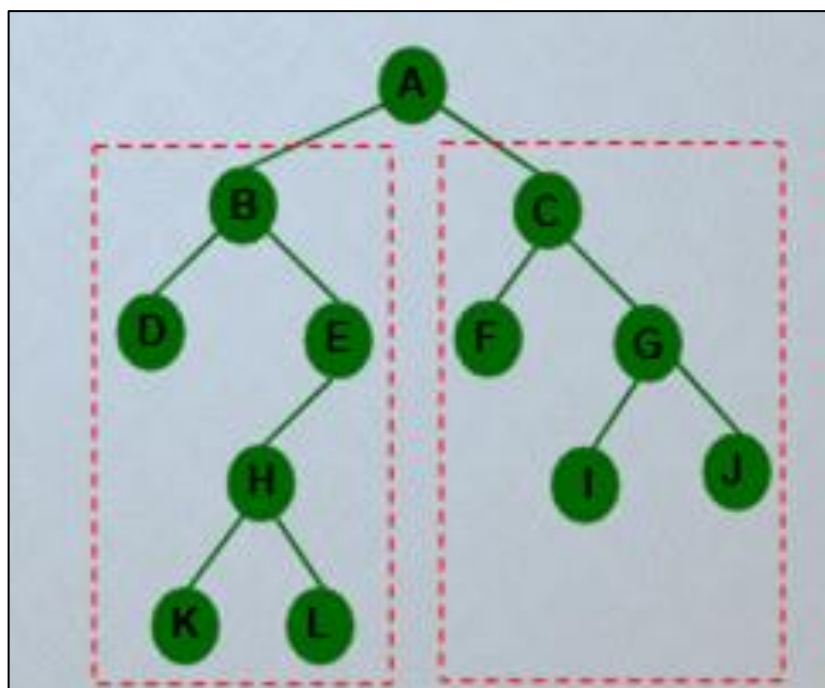


Hình 2. Ví dụ biểu diễn cây

- Ứng dụng: Cây thường được sử dụng trong nhiều ứng dụng như tổ chức dữ liệu (hệ thống tệp), biểu diễn cấu trúc phân cấp (như cây thư mục), và trong các thuật toán tìm kiếm.

1.1.1.2. Cây nhị phân

- Định nghĩa: Là cây mà mỗi nút không có quá 2 nút con, hai nút con (nếu có) được gọi là con trái và con phải.
- Cây con trái: Là cây có gốc là nút con trái.
- Cây con phải: Là cây có gốc là nút con phải.



Hình 3. Ví dụ về cây con trái và cây con phải

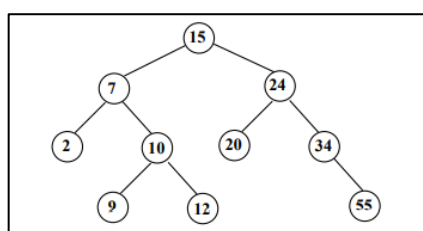
- Tính chất:
 - + Số nút tối đa ở mức i trên cây nhị phân là $2^{(i-1)}$ ($i \geq 1$).
 - + Số nút tối đa trên cây nhị phân chiều cao h là $2^h - 1$ ($h \geq 1$)
- Cấu trúc:
 - + Nút: Mỗi nút chứa một giá trị và hai tham chiếu đến nút con trái và nút con phải.
 - + Nút con trái (Left Child): Nút con bên trái của một nút.
 - + Nút con phải (Right Child): Nút con bên phải của một nút.
 - + Cây nhị phân đầy (Full Binary Tree): Tất cả các nút đều có 0 hoặc 2 nút con.
 - + Cây nhị phân hoàn chỉnh (Complete Binary Tree): Tất cả các cấp trừ cấp cuối cùng đều đầy và các nút ở cấp cuối cùng được đặt từ trái sang phải.
 - + Cây nhị phân cân bằng (Balanced Binary Tree): Chiều cao của hai cây con trái và phải không chênh lệch quá 1.
- Ứng dụng: Cây nhị phân thường được sử dụng để tổ chức dữ liệu, trong các thuật toán tìm kiếm và trong biểu diễn các biểu thức toán học.

1.1.1.3. Cây nhị phân tìm kiếm

a) Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân hoặc rỗng hoặc không rỗng thì phải thoả mãn đồng thời các điều kiện sau:

- Khoá của các nút thuộc cây con trái nhỏ hơn khoá nút gốc
- Khoá của nút gốc nhỏ hơn khoá của các nút thuộc cây con phải của nút gốc
- Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm



Hình 4. Ví dụ về cây nhị phân

- Nút: Mỗi nút trong BST chứa một giá trị, một tham chiếu đến nút con trái và một tham chiếu đến nút con phải.
- Hoạt động:
 - +Tìm kiếm (Search): Để tìm một giá trị trong BST, bắt đầu từ nút gốc, so sánh giá trị cần tìm với giá trị của nút hiện tại và tiếp tục tìm kiếm trong cây con trái hoặc cây con phải tùy thuộc vào kết quả so sánh.
 - +Chèn (Insert): Tương tự như tìm kiếm, bắt đầu từ nút gốc và tìm vị trí thích hợp để chèn nút mới theo quy tắc BST.
 - +Xóa (Delete): Có ba trường hợp: xóa nút lá (không có con), xóa nút có một con, và xóa nút có hai con (thay thế bằng giá trị lớn nhất từ cây con trái hoặc nhỏ nhất từ cây con phải).
- Ưu điểm: Cây tìm kiếm nhị phân cho phép tìm kiếm, chèn và xóa với độ phức tạp trung bình là $O(\log n)$ nếu cây được cân bằng.
- Nhược điểm: Nếu không duy trì tính cân bằng, độ phức tạp có thể trở thành $O(n)$ trong trường hợp xấu nhất (ví dụ, nếu cây trở thành một danh sách liên kết).

1.1.2. Phát biểu bài toán

Bài toán phân công công việc là bài toán tìm ra cách phân công chi phí tối thiểu cho các tác nhân sao cho mỗi tác nhân được phân công đúng một lần và các tác nhân không bị quá tải. Tất cả các cách tiếp cận đều dựa trên phân nhánh và ràng buộc với các ràng buộc được cung cấp thông qua các phương pháp tìm kiếm và nói lỏng công thức bài toán nguyên thủy.

Bài toán tổng quát được phát biểu như sau: Cho n công việc và n công nhân, trong đó công việc thứ i cần được thực hiện với một chi phí C_{ij} khi được phân công cho công nhân j . Nhiệm vụ của bạn là phân công mỗi công nhân một công việc sao cho tổng chi phí là nhỏ nhất.

- Input:
 - + Dòng đầu tiên chứa một số nguyên dương n .
 - + n dòng tiếp theo, mỗi dòng chứa n số nguyên là ma trận chi phí C_{ij} .
- Output:
 - + Dòng đầu tiên in ra tổng chi phí nhỏ nhất.
 - + Dòng thứ hai in ra thứ tự phân công công việc theo chỉ số của các công nhân.

1.1.3. Ứng dụng bài toán

Bài toán phân công công việc được ứng dụng rất nhiều trong đời sống hiện nay:



Hình 5. Nhân viên trong phòng ban phân công làm việc

Trong môi trường văn phòng ngày nay, để đạt được hiệu suất làm việc cao, nhân viên sẽ được phân công mỗi công việc khác nhau. Người quản lý sẽ theo dõi công việc của từng nhân viên và từ đó nhắc nhở nhân viên để hoàn thành công việc sao cho đạt hiệu quả nhất.



Hình 6. Máy móc hoạt động trong nhà máy sản xuất

Hiện nay, công nghệ vô cùng phát triển, các nhà máy sản xuất sử dụng robot và máy tự động trong quá trình sản xuất, các giám sát viên sẽ quan sát và phân công cho các máy móc đó. Vì vậy, để đạt được hiệu quả và sự trơn tru trong dây chuyền sản xuất bằng máy móc thì phân công công việc là vấn đề vô cùng quan trọng.



Hình 7. Công nhân làm việc trong các khu công nghiệp

Hiện nay, trong các khu công nghiệp, mỗi công nhân sẽ được phân công vào một giai đoạn nhất định trong quá trình sản xuất, từ đây có thể tiết kiệm thời gian mà vẫn có thể nâng cao năng suất lao động.



Hình 8. Xe tải được phân công chở hàng

Ngành logistic đang là một ngành rất phát triển trong thời kì hiện đại, vậy nên việc phân công các xe chở hàng sao cho hợp lí, tiết kiệm thời gian, tiết kiệm chi phí cũng là một vấn đề phải giải quyết bằng bài toán phân công công việc.

1.2. Cơ sở lý thuyết của các thuật toán

1.2.1. Các khái niệm xung quanh thuật toán và đánh giá hiệu năng thuật toán

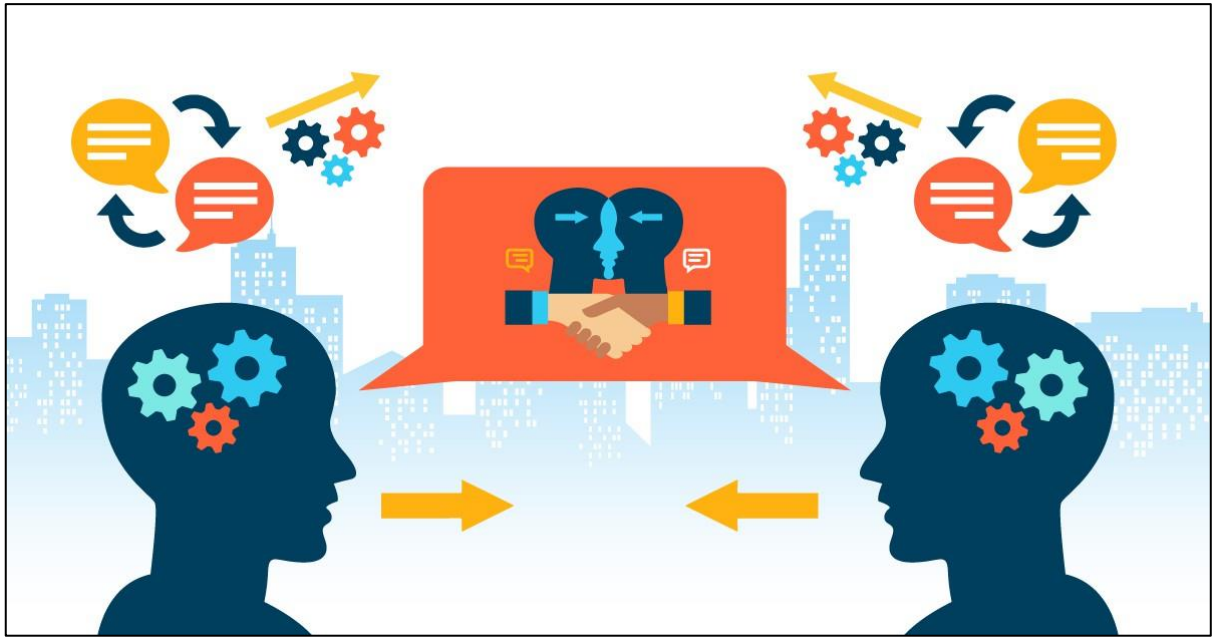
1.2.1.1. Thuật toán và phương pháp biểu diễn thuật toán

a) Các khái niệm xung quanh thuật toán

- Khái niệm thuật toán: Thuật toán là các quy tắc, chỉ thị hay phương thức nhằm hoàn thành trạng thái ban đầu được đưa ra. Chỉ khi các yêu cầu được sắp xếp một cách triệt để thì khi ấy thuật toán sẽ đem lại một kết quả chính xác.
- Thuật toán có 5 tính chất bao gồm: tính chính xác, tính khác quan, tính phổ dụng, tính rõ ràng và tính kết thúc.
 - + Tính chính xác: là yếu tố quan trọng nhất, mang tính chất khả dụng và khách quan của một thuật toán
 - + Tính khác quan: một thuật toán dù giải theo cách nào cũng chỉ có thể có một đáp án duy nhất, điều đó khẳng định sự tuyệt đối với kết quả bài toán
 - + Tính phổ dụng: mỗi một thuật toán không chỉ được ứng dụng trong một bài toán mà còn có thể áp dụng để giải các bài toán với nhiều dạng tương tự.
 - + Tính rõ ràng: trong một thuật toán, các lệnh được sắp xếp theo một trình tự vô cùng quy tắc, khi các lệnh được sắp xếp hợp lý sẽ giúp các thao tác trở nên trơn chu và nhanh gọn hơn nhiều.
 - + Tính kết thúc: là kết quả của một thuật toán

b) Phương pháp biểu diễn thuật toán bằng ngôn ngữ tự nhiên

- Là sử dụng một loại ngôn ngữ tự nhiên để liệt kê các bước của thuật toán.
- Ưu điểm:
 - + Đơn giản
 - + Không yêu cầu người viết và người đọc phải có kiến thức nền tảng



Hình 9. Minh họa về việc biểu diễn thuật toán bằng ngôn ngữ

- Nhược điểm:
 - + Dài dòng
 - + Không làm nổi bật cấu trúc của thuật toán
 - + Khó biểu diễn với những bài toán phức tạp
- Ví dụ: Tính chu vi hình chữ nhật với chiều dài a và chiều rộng b.
 Đầu vào: Hai số a và b
 Đầu ra: Chu vi hình chữ nhật chiều dài a và chiều rộng b.
 Ý tưởng: Tính tổng a và b rồi nhân 2

⇒ Mô tả bằng ngôn ngữ tự nhiên:

Bước 1: Bắt đầu

Bước 2: Nhập chiều dài a và chiều rộng b

Bước 3: Chuvi $\leftarrow (a+b)*2$

Bước 4: Hiển thị chu vi ra màn hình







Bước 5: Kết thúc

c) Phương pháp biểu diễn thuật toán sử dụng lưu đồ thuật toán

Khái niệm: Lưu đồ thuật toán (flowchart) là một biểu diễn trực quan của các bước và quá trình trong một thuật toán, được thể hiện qua các hình khối như hình chữ nhật, hình thoi, hình tròn... và các mũi tên nối để chỉ thứ tự thực hiện các bước.

Mỗi khối thường đại diện cho một loại hành động hoặc quyết định khác nhau, và các khối này kết hợp với nhau để mô tả logic của thuật toán.

- Phương pháp duyệt: có 2 phương pháp duyệt bao gồm
 - + Duyệt từ trên xuống
 - + Duyệt từ trái sang phải
- Một số quy ước ký hiệu lưu đồ:

Ký hiệu	Mô tả
	Điểm bắt đầu và kết thúc một thuật toán.
	Thao tác nhập hay xuất dữ liệu.
	Khối xử lý công việc.
	Khối quyết định chọn lựa.
	Dòng tính toán, thao tác của chương trình.
	Khối lệnh gọi hàm (chương trình con)

Hình 10. Các kí hiệu của 1 lưu đồ thuật toán

- Vai trò của lưu đồ thuật toán:
 - + Dễ hiểu: Giúp người dùng và lập trình viên dễ dàng theo dõi và nắm bắt được quy trình thực hiện của một thuật toán.
 - + Trực quan hóa logic: Giúp làm rõ và kiểm tra logic của thuật toán trước khi thực hiện mã hóa (coding).
 - + Giảm thiểu sai sót: Giúp nhận ra những điểm bất hợp lý trong thuật toán.

d) Phương pháp biểu diễn thuật toán sử dụng mã giả

- **Khái niệm:** Mã giả (pseudocode) là một cách biểu diễn thuật toán bằng cách dùng ngôn ngữ gần với ngôn ngữ tự nhiên, kết hợp với cấu trúc của ngôn ngữ lập trình. Mã giả giúp lập trình viên và người phân tích thuật toán dễ dàng hiểu và mô tả được logic của chương trình mà không cần quan tâm đến cú pháp chính xác của bất kỳ ngôn ngữ lập trình nào.
- Đặc điểm của mã giả:
 - + Không có cú pháp chuẩn: Mã giả không bị ràng buộc bởi cú pháp của bất kỳ ngôn ngữ lập trình nào. Thay vào đó, nó sử dụng các từ ngữ dễ hiểu để mô tả các bước.
 - + Cấu trúc logic rõ ràng: Sử dụng các cấu trúc điều khiển như If...Then...Else, For, While, Repeat...Until, để chỉ rõ luồng điều khiển.
 - + Dễ hiểu: Mã giả gần với ngôn ngữ tự nhiên và dễ hiểu với nhiều người, kể cả khi họ không quen thuộc với ngôn ngữ lập trình cụ thể.
- Lợi ích của mã giả:

- + Trực quan: Giúp phân tích và hiểu rõ thuật toán trước khi triển khai vào mã thực tế.
- + Giảm lỗi cú pháp: Vì không cần tuân thủ cú pháp của ngôn ngữ lập trình, mã giả giảm thiểu lỗi cú pháp, tập trung vào logic.
- + Dễ trao đổi: Mã giả rất hữu ích khi trao đổi ý tưởng với đồng nghiệp, nhà phân tích, hoặc những người không phải lập trình viên.
- Nhược điểm của pseudocode
 - + Bên cạnh những ưu điểm kể trên, pseudocode vẫn tồn tại một số khuyết điểm như sau:
 - + Pseudocode không cung cấp một đồ thị biểu diễn trực quan về logic của chương trình.
 - + Không có định dạng cố định cho mã giả.
- Ví dụ

```
# Đầu vào:
# - cost: Ma trận thời gian (cost[i][j]) chứa thời gian người lao động i thực hiện công việc j
# - n: Số lượng người lao động và công việc (n x n ma trận)

function findMinIndex(arr, assigned):
    # Khởi tạo giá trị ban đầu
    minValue = infinity
    minIndex = -1
    # Duyệt qua tất cả các công việc
    for i = 0 to length(arr) - 1:
        # Nếu công việc i chưa được gán và có thời gian nhỏ hơn minValue
        if not assigned[i] and arr[i] < minValue:
            minValue = arr[i]
            minIndex = i
    # Trả về chỉ số của công việc có thời gian nhỏ nhất
    return minIndex

function greedyAssignment(costMatrix):
    n = length(cost) # Số lượng công việc (và công nhân)
    assigned = array of size n with all values False # Đánh dấu công việc đã được gán
    assignment = array of size n with all values -1 # Ghi lại phân công công việc
    totalCost = 0 # Biến lưu tổng thời gian

    # Duyệt qua từng công nhân
    for worker = 0 to n - 1:
        # Tìm công việc có thời gian ngắn nhất chưa được gán cho công nhân hiện tại
        job = findMinIndex(cost[worker], assignedJobs)
        # Gán công việc này cho công nhân hiện tại
        assignment[worker] = job
        assigned[job] = True # Đánh dấu công việc đã được gán
        # Cộng dồn thời gian công việc vào tổng thời gian
        totalCost = totalCost + cost[worker][job]

    # Trả về tổng thời gian và danh sách phân công công việc
    return totalCost, assignment

# Đầu ra:
# - totalCost: Tổng thời gian thực hiện tất cả các công việc
# - assignment: Mảng lưu trữ kết quả phân công công việc cho mỗi công nhân
```

Hình 11. Đây là đoạn mã giả miêu tả cách thức hoạt động của thuật toán tham lam (Greedy Algorithm) trong bài toán phân công công việc.

1.2.1.2. Đánh giá hiệu năng thuật toán

- **Khái niệm:** Thời gian mà máy tính khi thực hiện một thuật toán không chỉ phụ thuộc vào bản thân thuật toán đó, ngoài ra còn tùy thuộc từng máy tính. Để đánh giá hiệu quả của một thuật toán, có thể xét số các phép tính phải thực hiện khi thực hiện thuật toán này. Thông thường số các phép tính được thực hiện phụ

thuộc vào cỡ của bài toán, tức là độ lớn của đầu vào. Vì thế độ phức tạp thuật toán là một hàm phụ thuộc đầu vào

- Xác định độ phức tạp của thuật toán
 - + Độ phức tạp tính toán của giải thuật: $O(f(n))$
 - + Việc xác định độ phức tạp tính toán của giải thuật trong thực tế có thể tính bằng một số quy tắc đơn giản sau:
- Quy tắc bỏ hằng số:

$$T(n) = O(c \cdot f(n)) = O(f(n)) \text{ với } c \text{ là một hằng số dương}$$

- Quy tắc lấy max:

$$T(n) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

- Quy tắc cộng:

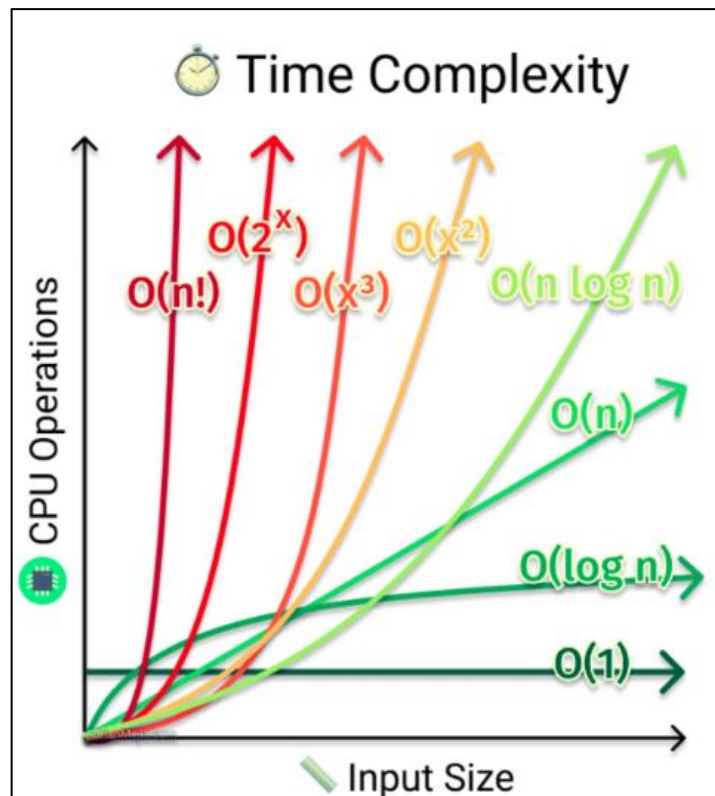
$$T1(n) = O(f(n)) \qquad T2(n) = O(g(n))$$

$$T1(n) + T2(n) = O(f(n) + g(n))$$

- Quy tắc nhân:

$$\text{Đoạn chương trình có thời gian thực hiện } T(n) = O(f(n))$$

Nếu thực hiện $k(n)$ lần đoạn chương trình với $k(n) = O(g(n))$ thì độ phức tạp sẽ là $O(g(n) \cdot f(n))$



Hình 12. Ví dụ minh họa về độ phức tạp của thuật toán

– Độ phức tạp thời gian

Định nghĩa: Độ phức tạp thời gian của thuật toán là thước đo số lượng phép tính tối đa mà thuật toán cần thực hiện để giải quyết một bài toán với bộ dữ liệu đầu vào có kích thước n .

Độ phức tạp thời gian thường được biểu diễn bằng ký hiệu $O(f(n))$, trong đó $f(n)$ là một hàm số thực không âm, nghĩa là thời gian thực hiện của thuật toán sẽ tỉ lệ thuận với $f(n)$ khi kích thước dữ liệu đầu vào n tăng lên.

Để đánh giá thời gian thực hiện thuật toán, ta xuất phát từ các lệnh đơn trong chương trình, rồi tới các câu lệnh có cấu trúc, các khối lệnh phức tạp hơn, sau đó hợp lại thành thời gian thực hiện cả chương trình. Cụ thể ta có các quy tắc:

Các lệnh đơn (lệnh khai báo, gán, nhập xuất dữ liệu, phép toán số học,...): Thời gian $O(1)$.

Các khối lệnh: Giả sử một khối lệnh gồm các câu lệnh S_1, S_2, \dots, S_m có thời gian thực hiện lần lượt là $O(f_1(n)), O(f_2(n)), \dots, O(f_m(n))$ thì thời gian thực hiện của cả khối lệnh là: $O(\max(f_1(n), f_2(n), \dots, f_m(n)))$.

Câu lệnh rẽ nhánh: Ta có cú pháp lệnh rẽ nhánh là:

Giả sử thời gian thực hiện của câu lệnh 1 và câu lệnh 2 lần lượt là $O(f_1(n))$ và $O(f_2(n))$ thì thời gian thực hiện lệnh rẽ nhánh là: $O(\max(f_1(n), f_2(n)))$.

Câu lệnh lặp: Giả sử thời gian thực hiện phần thân của lệnh lặp là $O(f_1(n))$ và số lần lặp tối đa của vòng lặp là $f_2(n)$ thì thời gian thực hiện của cả vòng lặp là $O(f_1(n).f_2(n))$. Điều này áp dụng cho tất cả các vòng lặp for, while và do...while.

Sau khi đánh giá được thời gian thực hiện của tất cả các câu lệnh trong chương trình, thời gian thực hiện của toàn bộ chương trình sẽ là thời gian thực hiện của câu lệnh có thời gian thực hiện lớn nhất. Ngoài ra, nếu như độ phức tạp tính toán là $O(c \times f(n))$ với c là một hằng số nhỏ, ta có thể bỏ qua c và coi như thuật toán có độ phức tạp là $O(f(n))$ - chẳng hạn như $O(3n), O(4n)$ có thể coi như $O(n)$.

– Độ phức tạp không gian

Định nghĩa: Độ phức tạp không gian của thuật toán là thước đo dung lượng bộ nhớ tối đa mà thuật toán cần sử dụng để lưu trữ dữ liệu và thực hiện phép tính với bộ dữ liệu đầu vào có kích thước n .

Độ phức tạp không gian cũng được biểu diễn bằng ký hiệu $O(g(n))$, trong đó $g(n)$ là một hàm số thực không âm, nghĩa là dung lượng bộ nhớ mà thuật toán sử dụng sẽ tỉ lệ thuận với $g(n)$ khi kích thước dữ liệu đầu vào n tăng lên.

Độ phức tạp không gian của một thuật toán sẽ được tính toán thông qua hàm $O(g(n))$ trước, rồi mới đổi ra giá trị dung lượng cụ thể. Nó là tổng của tất cả bộ nhớ sử dụng trong việc nhập dữ liệu đầu vào và bộ nhớ phụ sử dụng khi thực hiện thuật toán. Các quy tắc tính toán cơ bản như sau:

Các biến đơn khi khai báo (một hoặc nhiều biến): $O(1)$.

Khai báo mảng một chiều kích thước n : $O(n)$.

Khai báo mảng nhiều chiều có kích thước các chiều lần lượt là n_1, n_2, \dots, n_k : $O(n_1 \times n_2 \times \dots \times n_k)$.

Lời gọi đệ quy: Phụ thuộc vào số lượng lời gọi đệ quy lưu đồng thời trong phân vùng bộ nhớ call stack (sẽ học ở bài Hàm đệ quy).

Tổng bộ nhớ sử dụng trong toàn bộ chương trình sẽ hợp thành độ phức tạp không gian của chương trình là $O(g(n))$. Sau khi tính được, $O(g(n))$, ta sẽ quy đổi nó ra dung lượng bộ nhớ tương ứng với kiểu dữ liệu của input để tính ra được bộ nhớ sử dụng một cách tương đối chính xác.

1.2.2. Cơ sở lý thuyết của thuật toán tham lam

1.2.2.1. Lý thuyết chung về giải thuật tham lam

- Thuật toán tham lam (greedy algorithm) là một chiến lược thiết kế thuật toán, trong đó thuật toán đưa ra quyết định tối ưu tại từng bước mà không cần quan tâm đến ảnh hưởng của quyết định đó về sau. Mục tiêu là chọn các bước sao cho chi phí (hoặc giá trị) hiện tại là nhỏ nhất (hoặc lớn nhất) trong số các lựa chọn khả dĩ, với hy vọng rằng việc tích lũy các quyết định tối ưu cục bộ này sẽ dẫn đến một lời giải tốt (có thể tối ưu) cho toàn bộ bài toán.
- Thuật toán tham lam là một cách tiếp cận để giải quyết vấn đề, đưa ra một loạt các lựa chọn, từng lựa chọn một, với mục tiêu đạt được giải pháp tối ưu. Ở mỗi bước, thuật toán tham lam sẽ chọn tùy chọn khả dụng tốt nhất dựa trên một số tiêu chí được xác định trước, mà không xem xét bối cảnh toàn cục hoặc hậu quả tiềm ẩn của lựa chọn trong các bước tiếp theo. Nguyên tắc chính là luôn đưa ra lựa chọn tối ưu cục bộ, hy vọng rằng hiệu ứng tích lũy của những lựa chọn này sẽ dẫn đến giải pháp tổng thể tốt nhất.
- Trong phương pháp tham lam, việc lựa chọn quyết định tối ưu được thực hiện dựa trên thông tin hiện có mà không lo lắng về tác động mà những quyết định này có thể gây ra trong tương lai. Các thuật toán tham lam dễ phát minh, dễ triển khai và hầu hết thời gian đều khá hiệu quả

1.2.2.2. Nguyên lý hoạt động của thuật toán tham lam

Giải thuật tham lam xây dựng các giải pháp bằng cách lựa chọn hành động tốt nhất tại mỗi bước, mà không xem xét tác động của hành động đó đến các bước sau. Ý tưởng này dựa trên quan sát rằng, nếu mỗi lần chọn hành động tốt nhất tại thời điểm hiện tại, ta có thể đạt được kết quả tối ưu toàn cục.

1.2.2.3. Các thành phần của giải thuật tham lam

- Một tập hợp các ứng viên (candidate), để từ đó tạo ra lời giải
- Một hàm lựa chọn, để theo đó lựa chọn ứng viên tốt nhất để bổ sung vào lời giải

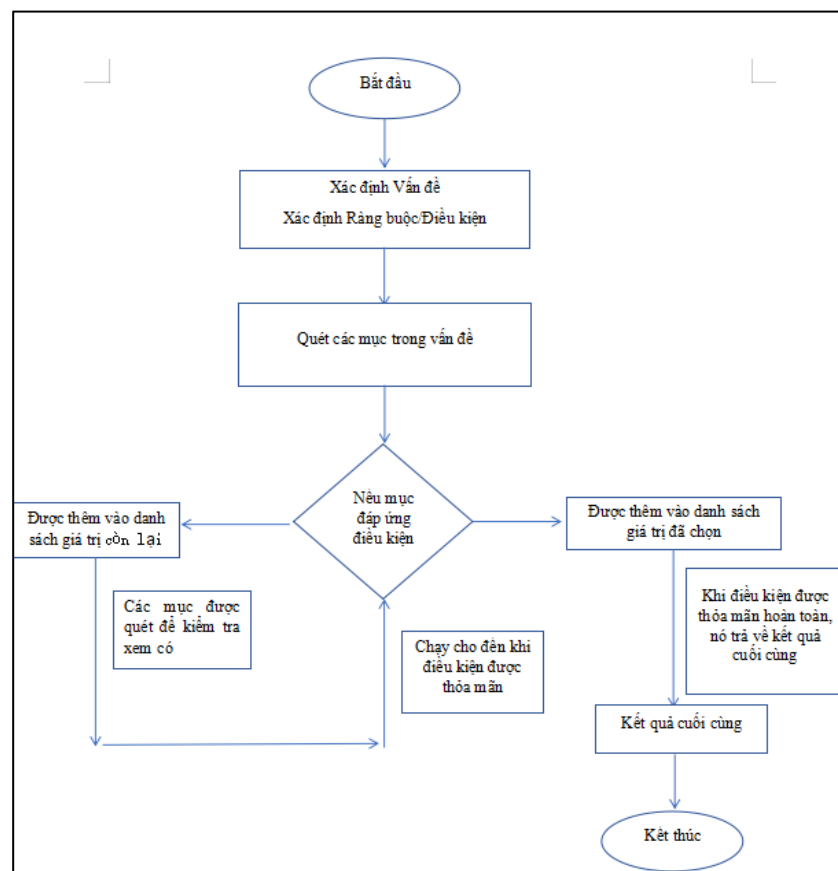
- Một hàm khả thi (feasibility), dùng để quyết định nếu một ứng viên có thể được dùng để xây dựng lời giải
- Một hàm mục tiêu, ấn định giá trị của lời giải hoặc một lời giải chưa hoàn chỉnh
- Một hàm đánh giá, chỉ ra khi nào ta tìm ra một lời giải hoàn chỉnh.

1.2.2.4. Mô hình chung

– Ý tưởng

Giả sử các bạn có thể biểu diễn nghiệm của bài toán dưới dạng một vector $X=(x_1, x_2, \dots, x_n)$ và mỗi thành phần x_i chọn ra từ một tập S_i các ứng cử viên. Vẫn tương tự như trong bài toán tối ưu, các nghiệm sẽ được xác định độ tốt bằng một hàm $f(X)$, và mục tiêu là cần đi tìm nghiệm có $f(X)$ tốt nhất (theo nghĩa lớn nhất hoặc nhỏ nhất).

Ở chiến lược Tham lam, chúng ta sẽ tìm cách tối ưu lựa chọn ở từng thành phần nghiệm. Giả sử đã xây dựng được i thành phần của nghiệm là x_1, x_2, \dots, x_i , thì khi xây dựng thành phần x_{i+1} , ta hãy cố gắng chọn nó là ứng cử viên "tốt nhất" trong tập ứng cử viên S_{i+1} . Để đánh giá được độ tốt của các ứng cử viên thì các bạn cần xây dựng một hàm chọn để làm điều đó. Tiếp tục xây dựng như vậy cho tới khi tạo ra đủ n thành phần của nghiệm.



Hình 13. Lưu đồ thuật toán tham lam

– **Ví dụ trong bài toán chọn hoạt động (Activity Selection)**

Bài toán: Bạn được cung cấp n hoạt động với thời gian bắt đầu và kết thúc. Chọn số lượng hoạt động tối đa mà một người có thể thực hiện, giả sử rằng một người chỉ có thể làm một hoạt động tại một thời điểm.

Đầu vào: $\text{start}[] = \{1, 3, 0, 5, 8, 5\}$, $\text{finish}[] = \{2, 4, 6, 7, 9, 9\}$;

Đầu ra: 0 1 3 4

Giải thích: Một người có thể thực hiện tối đa bốn hoạt động.

Tập hợp tối đa các hoạt động có thể thực hiện được

là $\{0, 1, 3, 4\}$ [Đây là các chỉ mục trong $\text{start}[]$ và $\text{finish}[]$]

1.2.3. Cơ sở lý thuyết của thuật toán nhánh cận

1.2.3.1. Lý thuyết chung về thuật toán nhánh cận

- Thuật toán nhánh cận (Branch and Bound) là một phương pháp giải quyết các bài toán tối ưu tổ hợp, trong đó tìm kiếm được tổ chức dưới dạng cây. Mỗi nhánh của cây đại diện cho một tập các quyết định và được cắt bớt khi phát hiện ra không thể tìm được lời giải tốt hơn trong nhánh đó. Ý tưởng chính của thuật toán nhánh cận là phân chia bài toán lớn thành các bài toán con nhỏ hơn và lần lượt khám phá các lời giải tiềm năng trong các nhánh này.
- Ứng dụng của Thuật Toán Nhánh Cận:
 - + Tối ưu hóa tổ hợp: Thuật toán nhánh cận được sử dụng rộng rãi trong việc giải quyết các bài toán tối ưu hóa tổ hợp như bài toán người bán hàng, bài toán cái túi và phân công công việc.
 - + Vấn đề thỏa mãn ràng buộc: Thuật toán có khả năng xử lý hiệu quả các vấn đề thỏa mãn ràng buộc bằng cách khám phá có hệ thống không gian tìm kiếm và cắt tỉa các nhánh dựa trên các ràng buộc.
 - + Phân bổ tài nguyên: Thuật toán được áp dụng trong các kịch bản như phân bổ tài nguyên, nơi cần phân phối tài nguyên một cách tối ưu giữa các nhu cầu cạnh tranh.
- Ưu điểm của Thuật Toán Nhánh Cận:
 - + Tính tối ưu: Thuật toán đảm bảo tính tối ưu trong các giải pháp cho những bài toán thỏa mãn các điều kiện nhất định, đảm bảo rằng giải pháp tốt nhất có thể được tìm thấy.
 - + Hiệu quả bộ nhớ: Thuật toán thường yêu cầu ít bộ nhớ hơn so với các phương pháp tìm kiếm toàn diện khác như brute force, đặc biệt đối với các bài toán có không gian tìm kiếm lớn.
 - + Tính linh hoạt: Thuật toán có thể điều chỉnh cho nhiều lĩnh vực vấn đề khác nhau và có thể đáp ứng các biểu diễn và ràng buộc vấn đề khác nhau.

- + Song song hóa: Các thuật toán nhánh cận có thể được song song hóa hiệu quả, cho phép khám phá nhanh chóng không gian tìm kiếm bằng cách sử dụng nhiều bộ xử lý hoặc tài nguyên tính toán.
- Nhược điểm của Thuật Toán Nhánh Cận:
 - + Độ phức tạp Việc triển khai các thuật toán nhánh cận có thể rất phức tạp, đặc biệt đối với các bài toán có ràng buộc phức tạp và không gian tìm kiếm lớn.
 - + Phụ thuộc vào Heuristic: Hiệu quả của thuật toán nhánh cận phụ thuộc nặng nề vào chất lượng của hàm giới hạn (bounding function) và các heuristics được sử dụng để hướng dẫn tìm kiếm, điều này không phải lúc nào cũng có sẵn hoặc dễ dàng thiết kế.
 - + Khó khăn trong môi trường động: Thuật toán không phù hợp cho các môi trường động hoặc thay đổi, nơi mà các ràng buộc hoặc mục tiêu của bài toán có thể thay đổi thường xuyên, vì nó dựa vào một phiên bản bài toán cố định để khám phá không gian tìm kiếm.

1.2.3.2. Nguyên lý hoạt động của thuật toán nhánh cận

Thuật toán sử dụng cây tìm kiếm, mỗi nút đại diện cho một trạng thái phân công công việc, mỗi nhánh đại diện cho việc lựa chọn một công việc được giao cho một người cụ thể.

- **Nhánh (Branch):** Tại mỗi cấp độ của cây, bạn chọn một công việc và thử phân công nó cho một người khác nhau, tạo ra các nhánh con. Các nhánh con sẽ tiếp tục phân công những công việc còn lại cho đến khi tất cả các công việc đều được gán.
- **Cận (Bound):** Để giảm thiểu số lượng nhánh phải xét, tại mỗi nút, ta tính một giới hạn dưới cho tổng thời gian tối thiểu có thể đạt được. Nếu giới hạn này lớn hơn hoặc bằng tổng thời gian của nghiệm tốt nhất đã biết, ta có thể cắt bỏ nhánh đó mà không cần tiếp tục xét.

1.2.3.3. Các thành phần của thuật toán nhánh cận

Phương pháp nhánh và cận là một dạng cải tiến của phương pháp quay lui, được áp dụng để tìm nghiệm của bài toán tối ưu. Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng một vector (x_1, x_2, \dots, x_n) mỗi thành phần x_i ($i = 1, 2, \dots, n$) được chọn ra từ tập S_i . Mỗi nghiệm của bài toán $X = (x_1, x_2, \dots, x_n)$ được xác định “độ tốt” bằng một hàm $f(X)$ và mục tiêu cần tìm nghiệm có giá trị $f(X)$ đạt giá trị nhỏ nhất (hoặc đạt giá trị lớn nhất). Tư tưởng của phương pháp nhánh và cận như sau: Giả sử, đã xây dựng được k thành phần (x_1, x_2, \dots, x_k) của nghiệm và khi mở rộng nghiệm $(x_1, x_2, \dots, x_{k+1})$, nếu biết rằng tất cả các nghiệm mở rộng của nó $(x_1, x_2, \dots, x_{k+1}, \dots)$ nếu không tốt bằng nghiệm tốt nhất đã biết ở

thời điểm đó, thì ta không cần mở rộng từ (x_1, x_2, \dots, x_k) nữa. Như vậy, với phương pháp nhánh và cận, ta không phải duyệt toàn bộ các phương án để tìm ra nghiệm tốt nhất mà bằng cách đánh giá các nghiệm mở rộng, ta có thể cắt bỏ đi những phương án (nhánh) không cần thiết, do đó việc tìm nghiệm tối ưu sẽ nhanh hơn. Cái khó nhất trong việc áp dụng phương pháp nhánh và cận là đánh giá được các nghiệm mở rộng, nếu đánh giá được tốt sẽ giúp bỏ qua được nhiều phương án không cần thiết, khi đó thuật toán nhánh cận sẽ chạy nhanh hơn nhiều so với thuật toán vét cạn.

1.2.3.4. Mô hình chung

Ý tưởng chính của thuật toán nhánh cận là giảm thiểu không gian tìm kiếm bằng cách loại bỏ sớm các nhánh không có khả năng dẫn đến nghiệm tốt hơn. Dưới đây là các bước chính của ý tưởng hoạt động của thuật toán nhánh cận:

a) Xây dựng cây tìm kiếm:

- Bài toán được thể hiện dưới dạng một cây tìm kiếm, trong đó mỗi nhánh của cây biểu diễn một trạng thái hoặc một lựa chọn. Gốc của cây là trạng thái ban đầu của bài toán, và mỗi nút của cây đại diện cho một trạng thái trung gian hoặc một lời giải gần đúng của bài toán.

b) Ràng buộc giới hạn (Bounding):

- Tại mỗi nút, thuật toán tính toán một giá trị giới hạn (bound), thể hiện tiềm năng tốt nhất có thể đạt được từ nhánh đó.
- Nếu giá trị giới hạn của nhánh hiện tại thấp hơn nghiệm tốt nhất tìm được (BestSolution), nhánh đó sẽ bị loại bỏ và thuật toán không tiếp tục mở rộng nhánh này nữa. Điều này giúp giảm không gian tìm kiếm và tiết kiệm thời gian tính toán.

c) Nhánh (Branching):

- Nếu giá trị giới hạn cho thấy nhánh này vẫn có thể có nghiệm tốt hơn, thuật toán sẽ mở rộng nhánh (branch) này để tiếp tục tìm kiếm nghiệm tối ưu. Quá trình này thực hiện bằng cách tạo ra các nút con cho các trạng thái tiếp theo của lời giải.

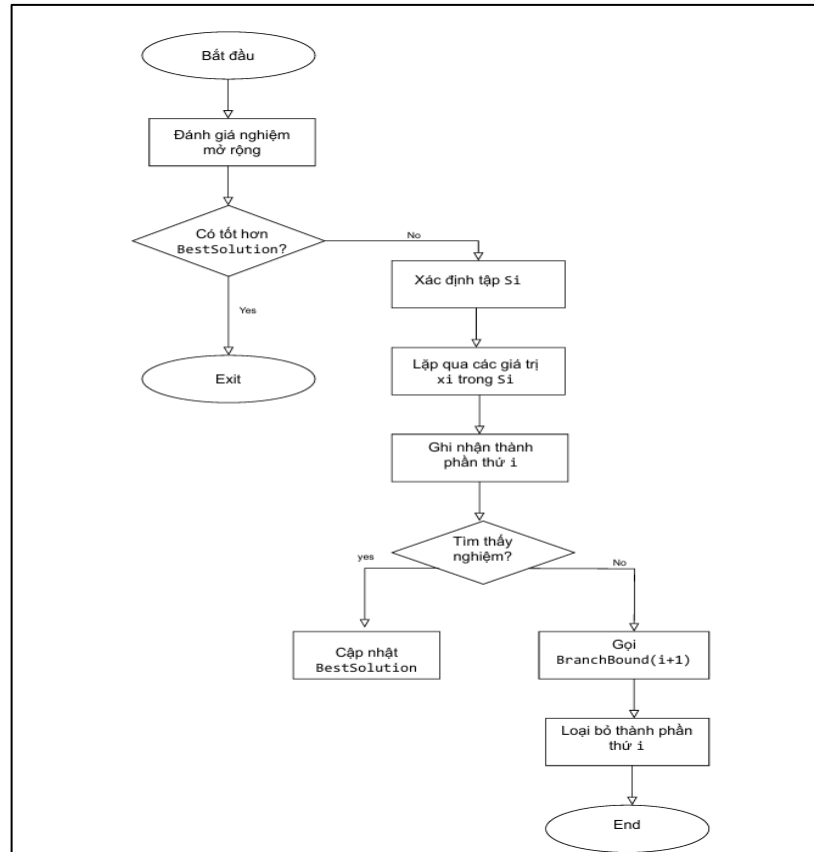
d) So sánh và cập nhật nghiệm tốt nhất:

- Khi tìm thấy một nghiệm hoàn chỉnh (còn gọi là nghiệm lá, tức là đến một lá của cây tìm kiếm) có giá trị tốt hơn nghiệm tốt nhất hiện tại, thuật toán sẽ cập nhật BestSolution bằng nghiệm này.
- Sau đó, tất cả các nhánh có giới hạn kém hơn nghiệm này đều sẽ bị loại bỏ.

e) Tiếp tục lặp lại:

- Thuật toán tiếp tục các bước trên cho đến khi không còn nhánh nào cần mở rộng. Khi đó, nghiệm tốt nhất hiện tại sẽ là nghiệm tối ưu của bài toán.

Thuật toán nhánh cận có thể mô tả bằng mô hình đệ quy sau:



Hình 14. Lưu đồ thuật toán của thuật toán nhánh cận

Ví dụ trong giải bài toán người du lịch

Bài toán: Cho n thành phố đánh số từ 1 đến n và các tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi mảng $C[1..n, 1..n]$, ở đây $C_{ij} = C_{ji}$ là chi phí phí đoạn đường trực tiếp từ thành phố i đến thành phố j . Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra cho người đó hành trình với chi phí ít nhất. Bài toán được gọi là bài toán người du lịch hay bài toán người chào hàng (Travelling Salesman Problem - TSP)

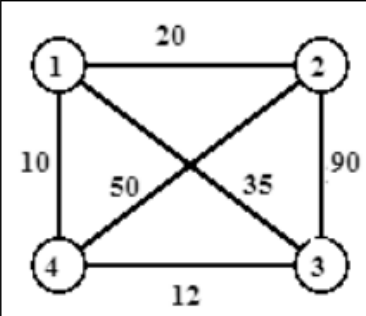
Dữ liệu vào trong file “TSP.INP” có dạng:

- Dòng đầu chứa số n ($1 < n \leq 20$), là số thành phố

- n dòng tiếp theo, mỗi dòng n số mô tả mảng C

Kết quả ra file “TSP.OUT” có dạng:

- Dòng đầu là chi phí ít nhất
- Dòng thứ hai mô tả hành trình

TSP.INP	TSP.OUT	Hình minh họa
<pre> 4 0 20 35 10 20 0 90 50 35 90 0 12 10 50 12 0 </pre>	<pre> 117 1->2->4->3->1 </pre>	

Hình 15. Ví dụ về bài toán người du lịch

- Hành trình cần tìm có dạng $(x_1 = 1, x_2, \dots, x_n, x_{n+1} = 1)$, ở đây giữa x_i và x_{i+1} : hai thành phố liên tiếp trong hành trình phải có đường đi trực tiếp; trừ thành phố 1, không thành phố nào được lặp lại hai lần, có nghĩa là dãy (x_1, x_2, \dots, x_n) lập thành một hoán vị của $(1, 2, \dots, n)$.
- Duyệt quay lui: x_2 có thể chọn một trong các thành phố mà x_1 có đường đi trực tiếp tới, với mỗi cách thử chọn x_2 như vậy thì x_3 có thể chọn một trong các thành phố mà x_2 có đường đi tới (ngoài x_1). Tổng quát: x_i có thể chọn 1 trong các thành phố chưa đi qua mà từ x_{i-1} có đường đi trực tiếp tới. ($2 \leq i \leq n$).
- Nhánh cận: Khởi tạo cấu hình BestSolution có chi phí $= +\infty$. Với mỗi bước thử chọn x_i xem chi phí đường đi cho tới lúc đó có nhỏ hơn chi phí của cấu hình BestSolution không? nếu không nhỏ hơn thì thử giá trị khác ngay bởi có đi tiếp cũng chỉ tốn thêm. Khi thử được một giá trị x_n ta kiểm tra xem x_n có đường đi trực tiếp về 1 không? Nếu có đánh giá chi phí đi từ thành phố 1 đến thành phố x_n cộng với chi phí từ x_n đi trực tiếp về 1, nếu nhỏ hơn chi phí của đường đi BestSolution thì cập nhật lại BestSolution bằng cách đi mới.

CHƯƠNG 2. THIẾT KẾ THUẬT TOÁN ĐỂ GIẢI BÀI TOÁN PHÂN CÔNG CÔNG VIỆC

2.1. Thiết kế thuật toán tham lam (Greedy)

2.1.1. Thuật toán tham lam trong bài toán phân công công việc

- Mục tiêu của bài toán là phân công n công việc cho n công nhân sao cho mỗi công nhân chỉ làm một công việc và mỗi công việc chỉ do một công nhân thực hiện. Tổng thời gian thực hiện các công việc là nhỏ nhất.
- Phương pháp tham lam đưa ra các lựa chọn tốt nhất ở mỗi bước (theo thời gian nhỏ nhất), mà không xét đến tác động lâu dài của những lựa chọn đó.
- Tư tưởng tham lam trong bài toán này là:

Tại mỗi bước, chọn công việc có thời gian ngắn nhất cho một công nhân mà công việc đó chưa được thực hiện bởi bất kỳ công nhân nào khác.

2.1.2. Các bước giải thuật

2.1.2.1. Bài toán và phân tích yêu cầu

- Input:
 - + Một ma trận $cost[n][n]$, trong đó:
 - $cost[i][j]$ đại diện cho thời gian mà công nhân thứ i cần để thực hiện công việc thứ j .
 - Số lượng công nhân là n .
- Output:
 - + Một phương án phân công công việc cho các công nhân sao cho mỗi công nhân được gán một công việc, mỗi công việc chỉ được thực hiện bởi một công nhân, và tổng thời gian là nhỏ nhất.
 - + In ra tổng thời gian tối thiểu.
- **Ràng buộc** : Mỗi công nhân chỉ làm một công việc và mỗi công việc chỉ do một công nhân thực hiện.

2.1.2.2. Giải thuật

Bước 1: Khởi tạo bài toán

- Đầu vào: Chúng ta có một tập hợp các lựa chọn (ở đây là tập hợp các công việc) và một tập các đối tượng (công nhân) cần gán.
- Biến khởi tạo: Mảng để lưu các công việc đã được gán, tất cả ban đầu đều chưa được gán (giá trị false).
- Một mảng để ghi nhận công việc nào được gán cho từng đối tượng.
- Một biến để lưu giá trị tối ưu cần tính

Bước 2: Xác định lựa chọn cục bộ

- Lựa chọn cục bộ tối ưu là lựa chọn tại mỗi bước, công việc nào cần được gán cho công nhân hiện tại.
- Đối với bài toán này, lựa chọn cục bộ là tìm công việc có thời gian ngắn nhất chưa được gán cho công nhân hiện tại.
- Điều kiện tham lam: Mỗi bước chọn công việc tốt nhất mà không quan tâm đến các bước sau.

Bước 3: Cập nhật trạng thái bài toán

- Sau khi chọn công việc tốt nhất, công việc đó được gán cho công nhân hiện tại, và được đánh dấu là "đã gán".
- Cập nhật chi phí hoặc thời gian vào tổng chi phí.

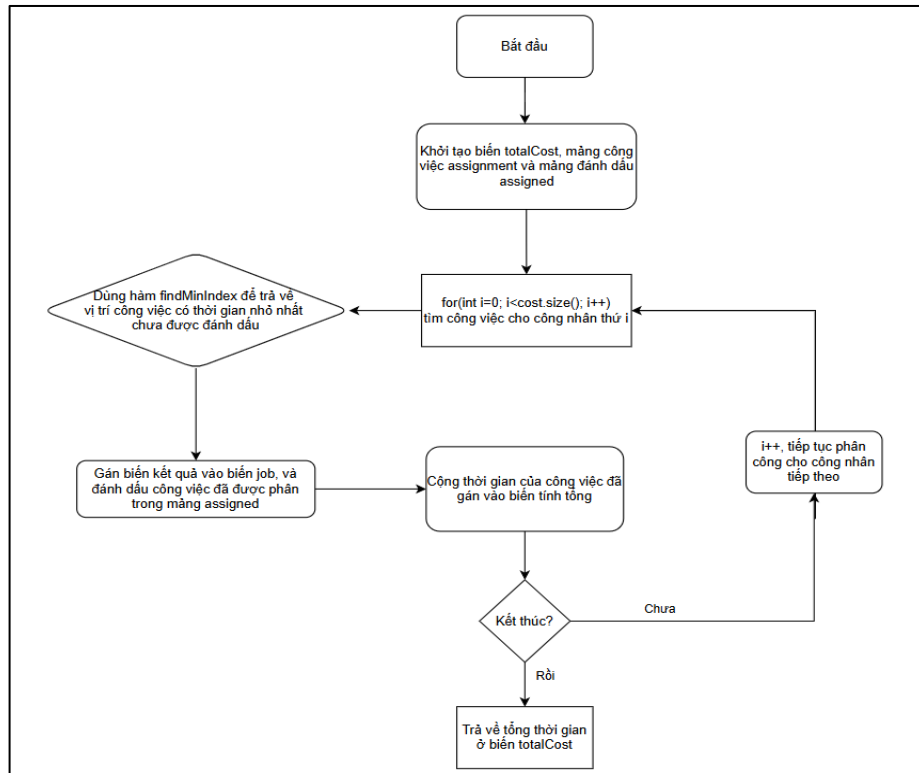
Bước 4: Lặp lại quá trình

- Tiếp tục thực hiện lựa chọn cục bộ (chọn công việc tốt nhất) cho công nhân tiếp theo và cập nhật trạng thái cho đến khi tất cả công nhân đều được gán công việc.

Bước 5: Kết thúc và xuất kết quả:

- Sau khi tất cả các công nhân đều được gán công việc, kết thúc quá trình lặp và xuất ra kết quả (công việc đã gán cho mỗi công nhân và tổng thời gian thực hiện).

2.1.2.3. Lưu đồ thuật toán



Hình 16. Lưu đồ thuật toán tham lam

2.1.2.4. Mã giả

```
# Đầu vào:
# - cost: Ma trận thời gian (cost[i][j]) chứa thời gian người lao động i thực hiện công việc j
# - n: Số lượng người lao động và công việc (n x n ma trận)

function findMinIndex(arr, assigned):
    # Khởi tạo giá trị ban đầu
    minValue = infinity
    minIndex = -1
    # Duyệt qua tất cả các công việc
    for i = 0 to length(arr) - 1:
        # Nếu công việc i chưa được gán và có thời gian nhỏ hơn minValue
        if not assigned[i] and arr[i] < minValue:
            minValue = arr[i]
            minIndex = i
    # Trả về chỉ số của công việc có thời gian nhỏ nhất
    return minIndex

function greedyAssignment(costMatrix):
    n = length(cost) # Số lượng công việc (và công nhân)
    assigned = array of size n with all values False # Đánh dấu công việc đã được gán
    assignment = array of size n with all values -1 # Ghi lại phân công công việc
    totalCost = 0 # Biến lưu tổng thời gian

    # Duyệt qua từng công nhân
    for worker = 0 to n - 1:
        # Tìm công việc có thời gian ngắn nhất chưa được gán cho công nhân hiện tại
        job = findMinIndex(cost[worker], assignedJobs)
        # Gán công việc này cho công nhân hiện tại
        assignment[worker] = job
        assigned[job] = True # Đánh dấu công việc đã được gán
        # Cộng dồn thời gian công việc vào tổng thời gian
        totalCost = totalCost + cost[worker][job]

    # Trả về tổng thời gian và danh sách phân công công việc
    return totalCost, assignment

# Đầu ra:
# - totalCost: Tổng thời gian thực hiện tất cả các công việc
# - assignment: Mảng lưu trữ kết quả phân công công việc cho mỗi công nhân
```

Hình 17. Mã giả thuật toán tham lam

2.2. Thiết kế thuật toán nhánh cận (Branch and Bound)

2.2.1. Thiết kế thuật toán nhánh cận

2.2.1.1. Bài toán và phân tích yêu cầu

- Input:
 - + n (số công nhân, cũng là số công việc): Số nguyên dương $n \geq 1$
 - + Ma trận thời gian t_{ij} Ma trận kích thước $n \times n$, trong đó phần tử t_{ij} là thời gian công nhân thứ i (với $i = 1, 2, \dots, n$) cần để hoàn thành công việc j (với $j = 1, 2, \dots, n$).
- Output:
 - + Phân công tối ưu: Danh sách các cặp công nhân và công việc sao cho mỗi công nhân chỉ làm một công việc và mỗi công việc chỉ do một công nhân thực hiện.
 - + Tổng thời gian nhỏ nhất (TTG): Tổng thời gian tối thiểu để hoàn thành tất cả các công việc theo phương án phân công tối ưu.
- Ràng buộc:
 - + Số lượng công nhân và công việc phải bằng nhau: Đảm bảo n công nhân và n công việc.
 - + Mỗi công nhân chỉ có thể làm một công việc: Không có công nhân nào có thể thực hiện nhiều công việc cùng lúc.
 - + Mỗi công việc chỉ được làm bởi một công nhân: Không được có công việc nào bị trùng lặp.
 - + Giới hạn kích thước ma trận: Khi n lớn, bài toán sẽ có độ phức tạp cao (thuộc lớp bài toán NP-Complete), gây khó khăn trong việc tìm ra giải pháp tối ưu một cách nhanh chóng.
 - + Thời gian thực hiện từng công việc t_{ij} phải là số dương: Không thể có thời gian âm.

2.2.1.2. Các bước giải thuật

Bước 1. Khởi tạo

- **Tạo trạng thái gốc:** Ở trạng thái này, chưa có công việc nào được phân công. Đây là gốc của cây tìm kiếm.

Bước 2. Phát triển nhánh

- Tại mỗi cấp độ của cây tìm kiếm, chọn một công việc và thử phân công công việc đó cho một người.
- Sau khi phân công một công việc cho một người, ta tạo ra các nhánh con tương ứng, trong đó mỗi nhánh đại diện cho việc phân công công việc tiếp theo cho các người còn lại.

Bước 3. Tính toán tổng thời gian và cận dưới

- **Tính tổng thời gian tạm thời:** Sau mỗi bước phân công, tính tổng thời gian của các công việc đã được phân công.
- **Tính cận dưới**
 - + Để tối ưu hóa quá trình tìm kiếm, tính giới hạn cận dưới cho thời gian tối thiểu có thể đạt được.
 - + Cận dưới có thể được tính bằng cách cộng tổng thời gian tạm thời với thời gian nhỏ nhất có thể cho các công việc chưa được phân công.
 - + Cận dưới giúp loại bỏ những nhánh không khả thi.

Bước 4. Cắt nhánh

- Sau khi tính được tổng thời gian và cận dưới cho một nhánh:
 - + So sánh cận dưới với giá trị của nghiệm tốt nhất hiện tại (nếu có).
 - + Nếu cận dưới của nhánh đó lớn hơn hoặc bằng tổng thời gian của nghiệm tốt nhất hiện tại, cắt nhánh đó và không mở rộng thêm nhánh con từ nhánh này.
 - + Nếu tổng thời gian tạm thời nhỏ hơn nghiệm tốt nhất, tiếp tục phân nhánh để tìm các phương án tốt hơn.

Bước 5. Cập nhật nghiệm tối ưu

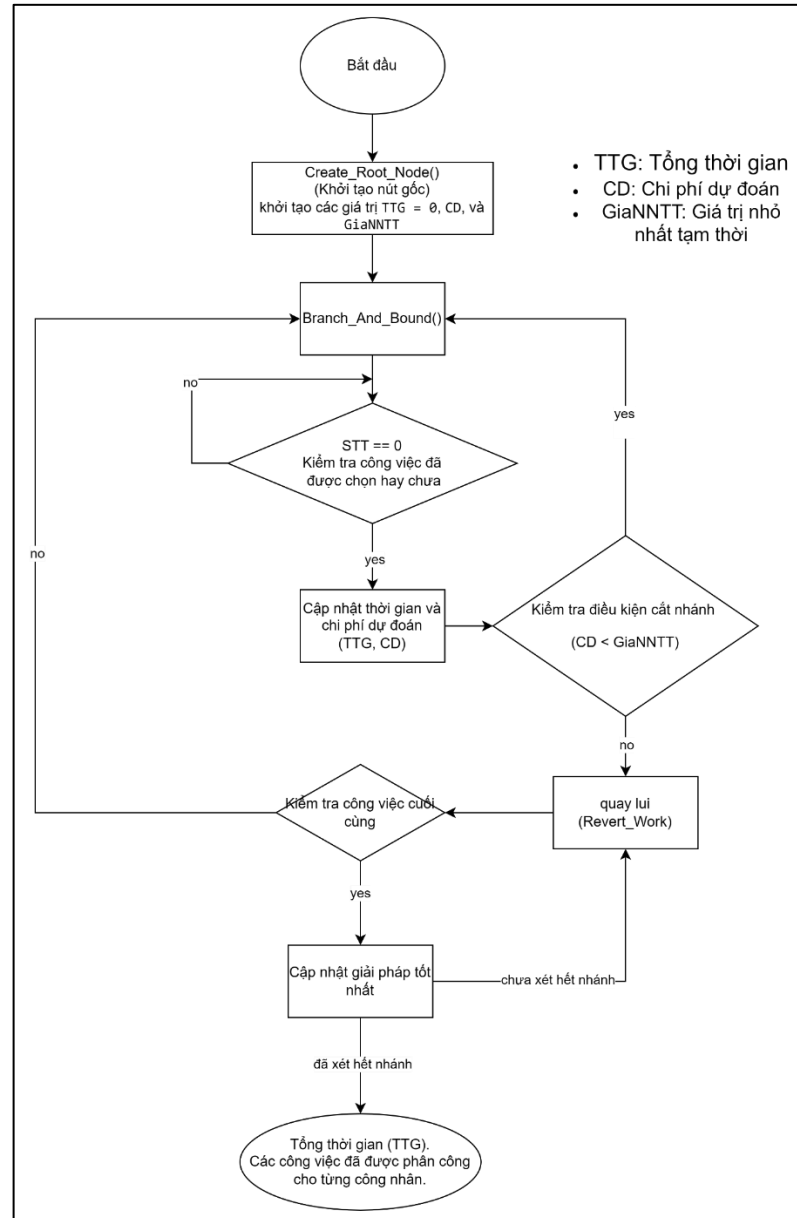
Khi một trạng thái phân công đầy đủ được tìm thấy (tất cả công việc đều được phân cho một người):

- So sánh tổng thời gian của trạng thái này với nghiệm tốt nhất hiện tại.
- Nếu tổng thời gian của trạng thái này nhỏ hơn, cập nhật nghiệm tốt nhất bằng trạng thái hiện tại.

Bước 6. Tiếp tục tìm kiếm hoặc dừng lại

- Lặp lại quá trình phát triển nhánh, tính cận dưới và cắt nhánh cho đến khi: Tất cả các nhánh khả thi đều được xét hoặc cắt bỏ.
- Khi không còn nhánh khả thi nào để mở rộng, nghiệm tối ưu sẽ là nghiệm cuối cùng được cập nhật.

2.2.1.3. Thiết kế giải thuật



Hình 18. Lưu đồ thuật toán nhánh cận

2.2.1.4. Mã giả

```
# Đầu vào:
# - matrix: Ma trận thời gian (matrix[i][j] chứa thời gian người lao động i thực hiện công việc j)
# - worker: Chỉ số của người lao động hiện tại (bắt đầu từ 0)
# - currentTime: Tổng thời gian hiện tại của các công việc đã phân
# - lowerBound: Giới hạn dưới của tổng thời gian cho những công việc tiếp theo
# - bestTime: Thời gian tốt nhất đã tìm được
# - solution: Mảng lưu trữ giải pháp tạm thời (công việc được phân cho từng người)
# - bestSolution: Mảng lưu trữ giải pháp tốt nhất (công việc được phân tốt nhất cho từng người)
# - n: Số lượng người lao động và công việc

# Đầu ra:
# - bestSolution: Giải pháp tốt nhất với tổng thời gian nhỏ nhất
# - bestTime: Tổng thời gian nhỏ nhất để hoàn thành tất cả công việc

function Branch_And_Bound(matrix, worker, currentTime, lowerBound, bestTime, solution, bestSolution, n):
    # Duyệt qua tất cả các công việc cho người thứ 'worker'
    for each job in matrix[worker]:
        # Kiểm tra nếu công việc chưa được giao
        if job is not assigned:
            # Cộng thời gian của công việc hiện tại vào tổng thời gian
            currentTime += job.time
            # Tính giới hạn dưới (lower bound) cho các công việc tiếp theo
            lowerBound = calculateLowerBound(matrix, currentTime, worker, n)
            # Nếu giới hạn dưới này nhỏ hơn thời gian tốt nhất hiện tại
            if lowerBound < bestTime:
                # Ghi nhận công việc hiện tại vào giải pháp tạm thời
                solution[worker] = job
                # Đánh dấu công việc là đã giao
                assignJob(job)
                # Nếu đã phân công hết công việc cho tất cả người (đến người cuối cùng)
                if worker == n - 1:
                    # Cập nhật giải pháp tốt nhất nếu cần thiết
                    updateBestSolution(currentTime, bestTime, solution, bestSolution)
                else:
                    # Định quy để phân công công việc cho người tiếp theo
                    Branch_And_Bound(matrix, worker + 1, currentTime, lowerBound, bestTime, solution, bestSolution, n)
                # Quay lui (backtrack), hủy việc giao công việc cho người hiện tại
                unassignJob(job)
            # Trừ thời gian của công việc hiện tại khỏi tổng thời gian (phục vụ quay lui)
            currentTime -= job.time

# Các hàm phụ trợ (giả sử đã được định nghĩa):
# - calculateLowerBound(matrix, currentTime, worker, n): Tính giới hạn dưới của tổng thời gian.
# - assignJob(job): Đánh dấu công việc là đã được giao cho người lao động.
# - unassignJob(job): Đánh dấu công việc là chưa được giao (quay lui).
# - updateBestSolution(currentTime, bestTime, solution, bestSolution):
# Cập nhật giải pháp tốt nhất nếu tổng thời gian hiện tại tốt hơn.
```

Hình 19. Mã giả thuật toán nhánh cận

2.3. Đánh giá hiệu quả các phương pháp

2.3.1. Phương pháp sử dụng thuật toán tham lam (Greedy)

2.3.1.1. Ưu điểm:

- **Đơn giản và dễ hiểu:** Thuật toán dựa trên cách tiếp cận tham lam (greedy), luôn chọn công việc có thời gian nhỏ nhất cho mỗi công nhân tại mỗi bước. Điều này giúp thuật toán dễ viết và dễ triển khai.
- **Thời gian chạy chấp nhận được:** Với độ phức tạp là $O(n^2)$ do vòng lặp tìm công việc có thời gian nhỏ nhất và gán nó cho công nhân, thuật toán có thể hoạt động tốt với số lượng công nhân và công việc không quá lớn.

2.3.1.2. Nhược điểm:

- **Không tối ưu toàn cục:** Mặc dù thuật toán này luôn tìm công việc nhanh nhất cho mỗi công nhân tại từng bước, nhưng nó có thể không cho ra giải pháp tối

ưu toàn cục. Trong một số trường hợp, việc chọn một công việc khác (không phải công việc có thời gian nhỏ nhất) có thể giúp giảm tổng thời gian. Đó là hạn chế của các thuật toán tham lam.

- **Phụ thuộc vào cách chọn:** Thuật toán này không thử nhiều lựa chọn khác nhau mà chỉ theo một chiến lược duy nhất. Điều này có thể bỏ lỡ các trường hợp tối ưu hơn.

2.3.1.3. Tính khả thi:

- Với bài toán này, nếu số công nhân và công việc bằng nhau và không có công việc nào bị bỏ trống, thuật toán sẽ luôn hoàn thành với một phương án hợp lệ. Tuy nhiên, trong trường hợp số lượng công nhân hoặc công việc thay đổi, bạn sẽ cần điều chỉnh thuật toán để đảm bảo tính hợp lệ của bài toán.

2.3.2. Phương pháp sử dụng thuật toán nhánh cận (Branch and Bound)

2.3.2.1. Ưu điểm:

- **Tìm ra được lời giải tối ưu toàn cục:** Thuật toán nhánh cận có thể khắc phục được nhược điểm của thuật toán tham lam, đó là có thể đưa ra lời giải tối ưu toàn cục dù cho bài toán có nhiều ràng buộc.
- **Giảm không gian tìm kiếm:** Thuật toán nhánh cận có thể bỏ qua các nhánh không có tiềm năng, từ đó giảm bớt thời gian so với duyệt toàn bộ không gian.

2.3.2.2. Nhược điểm:

- **Độ phức tạp cao:** Độ phức tạp của thuật toán nhánh cận trong trường hợp phải duyệt toàn bộ không gian là $O(n!)$. Vì mặc dù có thể giảm bớt không gian tìm kiếm bằng cách loại bỏ các nhánh không có tiềm năng nhưng nếu đầu vào của bài toán có kích thước lớn thì thuật toán sẽ không phù hợp.
- **Thời gian chạy phụ thuộc vào đầu vào bài toán:** Mặc dù không gian tìm kiếm đã giảm bớt nhưng nếu đầu vào bài toán có kích thước lớn thì thời gian chạy thuật toán sẽ khá tốn thời gian.

2.3.2.3. Tính khả thi:

- Thuật toán nhánh cận (Branch and Bound) là một lựa chọn tốt và phù hợp để giải những bài toán phân công công việc cần độ chính xác cao, đặc biệt là

cần lời giải tối ưu toàn cục. Tuy nhiên thì thuật toán chỉ phù hợp với những bài toán mà đầu vào của bài toán có kích thước vừa phải, mặc dù đã giảm không gian tìm kiếm nhờ cách nhánh không có tiềm năng và cận trên, cận dưới.

CHƯƠNG 3. CÀI ĐẶT VÀ KIỂM THỬ

3.1. Cài đặt thuật toán

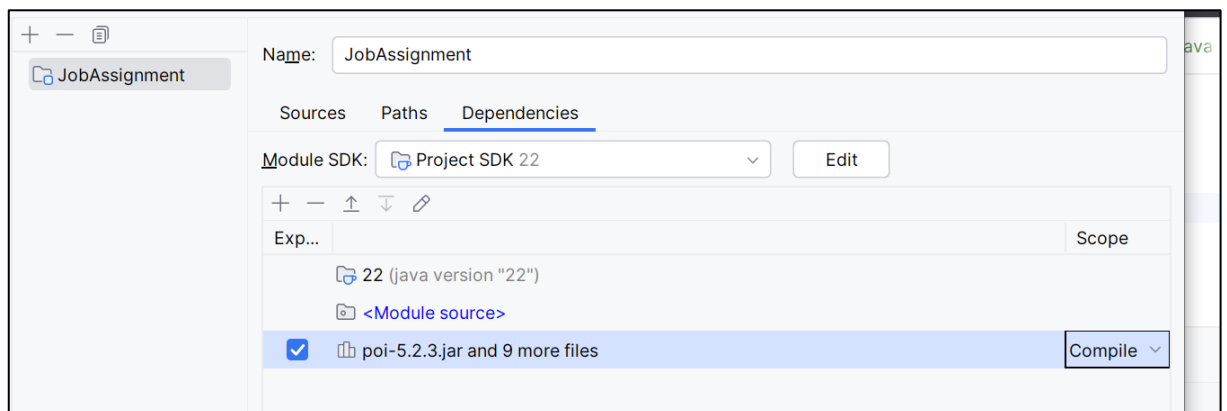
3.1.1. Môi trường cài đặt

- Loại máy: Máy tính xách tay MSI GV 75
- Thông số kỹ thuật:
- Bộ vi xử lý: Intel Core i7 7th Gen.
- Bộ nhớ RAM: 16 GB.
- Dung lượng ổ cứng: 512GB SSD.
- Hệ điều hành: Windows 10
- Ngôn ngữ lập trình: Java 22 với JDK 22.
- IDE sử dụng: IntelliJ Idea Ultimate.
- Công cụ biên dịch: JDK 22.

3.1.2. Sơ lược về chương trình

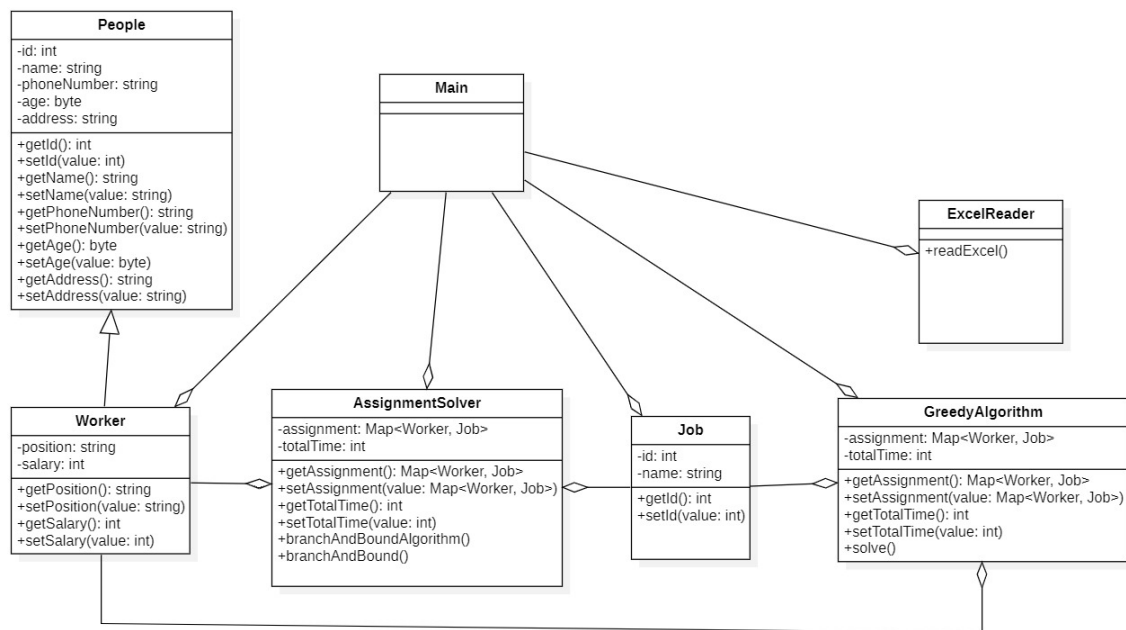
3.1.2.1. Thư viện sử dụng

Để có thể chuyển dữ liệu file Excel vào mô hình code, ta sử dụng thêm thư viện Apache Poi



Hình 20. Minh họa việc thêm thư viện trong dự án Java

3.1.2.2. Cài đặt chương trình



- Lớp People là lớp cha, đại diện cho đối tượng con người trong thực tế, có các thuộc tính cơ bản như id, name, phoneNumber, age, address trong tương lai có thể sử dụng cho các đối tượng cụ thể bằng cách extends.
- Lớp Worker là lớp con của lớp People, ở đây cụ thể hoá là 1 công nhân trong thực tế, có đủ các thuộc tính của 1 con người kèm theo thông tin của 1 công nhân như position, salary.
- Lớp Job là lớp đại diện cho công việc cụ thể, có các thuộc tính như id và name.
- Lớp ExcelReader là lớp phụ trợ, tạo hàm hỗ trợ cho việc phân tích file Excel để lấy dữ liệu cho bài toán.
- Lớp AssignmentSolver dùng để mô hình thuật toán BranchAndBound.
- Lớp GreedyAlgorithm dùng để mô hình thuật toán GreedyAlgorithm.
- Lớp Main đóng vai trò như Entry Point khởi tạo tất cả các thành phần để chạy.
- Ngoài các thuộc tính cơ bản thì còn có các phương thức hỗ trợ như Get, Set, hay các phương thức của các lớp thuật toán, chi tiết hơn ở trong giải thích đoạn code dưới đây.

3.1.2.3. Class People

```
package models;

public class People {
    private int id;

    private String name;

    private String phoneNumber;

    private byte age;

    private String address;

    public People() {}

    public People(int id, String name, String phoneNumber,
byte age, String address) {
        this.id = id;
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.age = age;
        this.address = address;
    }

    public People(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }
}
```



```

    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public byte getAge() {
        return age;
    }

    public void setAge(byte age) {
        this.age = age;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "People{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", phoneNumber='" + phoneNumber + '\'' +
            ", age=" + age +
            ", address='" + address + '\'' +
            '}';
    }
}

```

3.1.2.4. Class Worker

```

package models;

public class Worker extends People{
    private String position;
    private int salary;

    public Worker() {
    }

    public Worker(int id, String name){
        super(id, name);
    }
}

```

```

    public Worker(int id, String name, String
phoneNumber, byte age, String address, String position,
int salary) {
        super(id, name, phoneNumber, age, address);
        this.position = position;
        this.salary = salary;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Worker{" +
            "id=" + getId() +
            ", name='" + getName() + '\\\'' +
            ", phoneNumber='" + getPhoneNumber() +
            '\\\'' +
            ", age=" + getAge() +
            ", address='" + getAddress() + '\\\'' +
            ", position='" + position + '\\\'' +
            ", salary=" + salary +
            '}';
    }
}

```

3.1.2.5. Class Job

```

package models;

public class Job {
    private int id;

    private String name;

```

```

public Job() {
}

public Job(int id, String name) {
    this.id = id;
    this.name = name;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}

```

3.1.2.6. Class ExcelReader

```

package models;
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.*;

public class ExcelReader {
    public static Map<Integer, Map<Integer, Integer>>
readExcel(String filePath,

List<Worker> workers, List<Job> jobs) throws IOException
{
    Map<Integer, Map<Integer, Integer>> timeMatrix =
new HashMap<>();

    try (FileInputStream fis = new
FileInputStream(filePath);

```

```

        Workbook workbook = new XSSFWorkbook(fis))
    {

        Sheet sheet = workbook.getSheetAt(0); // Lấy
        sheet đầu tiên

        // Lấy danh sách Worker IDs (cột A)
        for (int i = 1; i <= sheet.getLastRowNum();
i++) {

            Row row = sheet.getRow(i);
            if (row != null) {
                int workerId = (int)
row.getCell(0).getNumericCellValue();
                if (workers.stream().noneMatch(w ->
w.getId() == workerId)) {
                    throw new
IllegalArgumentException("Worker ID " + workerId + " is
missing in List<Worker>");
                }
                timeMatrix.put(workerId, new
HashMap<>());
            }
        }

        // Lấy danh sách Job IDs (hàng 1)
        Row jobRow = sheet.getRow(0);
        if (jobRow != null) {
            for (int j = 1; j <
jobRow.getLastCellNum(); j++) {
                int jobId = (int)
jobRow.getCell(j).getNumericCellValue();
                if (jobs.stream().noneMatch(job ->
job.getId() == jobId)) {
                    throw new
IllegalArgumentException("Job ID " + jobId + " is
missing in List<Job>");
                }
            }
        }

        // Đọc ma trận thời gian làm việc (từ B2)
        for (int i = 1; i <= sheet.getLastRowNum();
i++) {

            Row row = sheet.getRow(i);
            if (row != null) {
                int workerId = (int)
row.getCell(0).getNumericCellValue();

```

```

        for (int j = 1; j <
row.getLastCellNum(); j++) {
            int jobId = (int)
jobRow.getCell(j).getNumericCellValue();
            int time = (int)
row.getCell(j).getNumericCellValue();
timeMatrix.get(workerId).put(jobId, time);
        }
    }
}

return timeMatrix;
}
}

```

3.1.1. Cài đặt thuật toán tham lam

3.1.1.1. Class GreedyAlgorithm

```

package models;

import java.util.*;

public class GreedyAlgorithm {

    // Lớp kết quả (Result) chứa thông tin về kết quả phân
    công công việc
    public static class Result {
        private final Map < Worker, Job > assignment; // Bản
        đồ lưu trữ phân công công việc (Worker -> Job)
        private final int totalTime; // Tổng thời gian hoàn
        thành các công việc

        public Result(Map < Worker, Job > assignment, int
totalTime) {
            this.assignment = assignment;
            this.totalTime = totalTime;
        }

        // Trả về bản đồ phân công công việc
        public Map < Worker, Job > getAssignment() {
            return assignment;
        }
    }
}

```

```

// Trả về tổng thời gian
public int getTotalTime() {
    return totalTime;
}

/
* Phương thức giải bài toán phân công công việc bằng
thuật toán tham lam (Greedy Algorithm).
* @param timeMatrix Ma trận thời gian, trong đó
timeMatrix[workerId][jobId] là thời gian mà worker thực
hiện job.
* @param workers Danh sách công nhân (Worker).
* @param jobs Danh sách công việc (Job).
* @return Kết quả phân công công việc và tổng thời
gian hoàn thành.
*/
public static Result solve(Map < Integer, Map <
Integer, Integer >> timeMatrix, List < Worker > workers,
List < Job > jobs) {
    Map < Worker, Job > assignment = new HashMap < > ();
// Lưu trữ kết quả phân công công việc
    Set < Integer > assignedJobs = new HashSet < > ();
// Tập hợp các công việc đã được gán
    int totalTime = 0; // Biến lưu tổng thời gian thực
hiện các công việc

    // Bước 1: Duyệt qua từng công nhân để phân công
công việc
    for (Worker worker: workers) {
        int minTime = Integer.MAX_VALUE; // Lưu thời gian
nhỏ nhất cho công việc phù hợp nhất
        Job chosenJob = null; // Công việc được chọn cho
công nhân này

        // Bước 2: Duyệt qua tất cả các công việc để tìm
công việc phù hợp nhất
        for (Job job: jobs) {
            // Kiểm tra nếu công việc chưa được gán cho bất
kỳ công nhân nào
            if (!assignedJobs.contains(job.getId())) {
                int time =
timeMatrix.get(worker.getId()).get(job.getId()); // Lấy
thời gian thực hiện công việc
                if (time < minTime) { // Nếu thời gian nhỏ hơn
thời gian hiện tại, cập nhật lựa chọn
                    minTime = time;

```

```

        chosenJob = job;
    }
}

// Bước 3: Gán công việc được chọn cho công nhân
nếu tìm thấy
    if (chosenJob != null) {
        assignment.put(worker, chosenJob); // Gán công
việc cho công nhân
        assignedJobs.add(chosenJob.getId()); // Đánh dấu
công việc này đã được gán
        totalTime += minTime; // Cộng thời gian thực
hiện công việc vào tổng thời gian
    }
}

// Bước 4: Trả về kết quả gồm bản đồ phân công công
việc và tổng thời gian
return new Result(assignment, totalTime);
}
}

```

3.1.2. Cài đặt thuật toán nhánh cận

3.1.2.1. Class AssignmentSolver

```

package models;

import java.util.*;

public class AssignmentSolver {
    // Lớp Result chứa kết quả của bài toán
    public static class Result {
        private Map<Worker, Job> assignment; // Kết quả
ánh xạ Worker -> Job
        private int totalTime; // Tổng thời gian tối ưu
cho tất cả Worker

        // Constructor: Lưu kết quả ánh xạ và tổng thời
gian
        public Result(Map<Worker, Job> assignment, int
totalTime) {
            this.assignment = assignment;
            this.totalTime = totalTime;
        }

        // Getter để lấy ánh xạ Worker -> Job

```

```

        public Map<Worker, Job> getAssignment() {
            return assignment;
        }

        // Getter để lấy tổng thời gian tối ưu
        public int getTotalTime() {
            return totalTime;
        }
    }

    /
    * Phương thức giải quyết bài toán phân công công
    việc (Assignment Problem)
    * bằng cách sử dụng thuật toán Branch and Bound.
    *
    * @param timeMatrix Ma trận thời gian công việc
    (Worker ID -> Job ID -> Time)
    * @param workers Danh sách Worker
    * @param jobs Danh sách Job
    * @return Kết quả chứa ánh xạ Worker -> Job và tổng
    thời gian tối ưu
    */
    public static Result branchAndBound(
        Map<Integer, Map<Integer, Integer>>
timeMatrix,
        List<Worker> workers,
        List<Job> jobs
    ) {
        int n = workers.size(); // Số lượng Worker
        int m = jobs.size();    // Số lượng Job

        // Kiểm tra nếu số Worker và Job không bằng nhau
        if (n != m) {
            throw new IllegalArgumentException("Số lượng
Worker và Job phải bằng nhau");
        }

        // Chuyển đổi ma trận thời gian thành ma trận 2D
        `costMatrix`
        int[][] costMatrix = new int[n][m];
        for (int i = 0; i < n; i++) {
            int workerId = workers.get(i).getId(); //
Lấy ID của Worker
            Map<Integer, Integer> jobTimes =
timeMatrix.get(workerId); // Lấy thời gian tương ứng
Worker -> Job
            for (int j = 0; j < m; j++) {

```



```

        int jobId = jobs.get(j).getId(); // Lấy
ID của Job
        // Lấy thời gian từ `timeMatrix` hoặc
gán giá trị vô cực nếu không tìm thấy
        costMatrix[i][j] =
jobTimes.getDefault(jobId, Integer.MAX_VALUE);
    }
}

// Gọi thuật toán Branch and Bound để tìm ánh xạ
tối ưu
Map<Integer, Integer> assignment =
branchAndBoundAlgorithm(costMatrix);

// Tính tổng thời gian tối ưu và tạo ánh xạ
Worker -> Job
int totalTime = 0;
Map<Worker, Job> resultAssignment = new
HashMap<>();
for (Map.Entry<Integer, Integer> entry :
assignment.entrySet()) {
    int workerIdx = entry.getKey(); // Chỉ số
Worker trong mảng
    int jobIdx = entry.getValue(); // Chỉ số Job
trong mảng

    // Cộng thời gian tối ưu
    totalTime += costMatrix[workerIdx][jobIdx];

    // Ánh xạ Worker -> Job
    Worker worker = workers.get(workerIdx);
    Job job = jobs.get(jobIdx);
    resultAssignment.put(worker, job);
}

// Trả về kết quả gồm ánh xạ và tổng thời gian
return new Result(resultAssignment, totalTime);
}

/
* Thuật toán Branch and Bound đơn giản để phân công
công việc tối ưu.
* Chọn công việc có chi phí nhỏ nhất cho mỗi
Worker.
*
* @param costMatrix Ma trận chi phí (n x n)
* @return Ánh xạ Worker Index -> Job Index

```

```

        */
        private static Map<Integer, Integer>
branchAndBoundAlgorithm(int[][] costMatrix) {
            int n = costMatrix.length; // Số Worker/Job
            Map<Integer, Integer> assignment = new
HashMap<>(); // Lưu ánh xạ Worker -> Job
            boolean[] assignedJobs = new boolean[n]; // Đánh
dấu công việc đã được phân công

            // Lặp qua từng Worker để gán công việc tối ưu
            for (int workerIdx = 0; workerIdx < n;
workerIdx++) {
                int minCost = Integer.MAX_VALUE; // Chi phí
nhỏ nhất cho Worker hiện tại
                int bestJobIdx = -1; // Chỉ số Job có chi
phí nhỏ nhất

                // Tìm Job với chi phí nhỏ nhất chưa được
gán
                for (int jobIdx = 0; jobIdx < n; jobIdx++) {
                    if (!assignedJobs[jobIdx] &&
costMatrix[workerIdx][jobIdx] < minCost) {
                        minCost =
costMatrix[workerIdx][jobIdx];
                        bestJobIdx = jobIdx;
                    }
                }

                // Gán Job cho Worker
                if (bestJobIdx != -1) {
                    assignedJobs[bestJobIdx] = true; // Đánh
dấu Job đã được gán
                    assignment.put(workerIdx, bestJobIdx);
                }
            }

            return assignment; // Trả về ánh xạ tối ưu
        }
    }
}

```

3.1.2.2. Class Main

```

import models.*;

import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

```

```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) throws IOException {
        // String filePath =
        "D:\\Downloads\\2024HAUI\\TTCN\\TTCN_GROUP_11\\Book1.xlsx";
        String filePath = "D:\\Project\\Book1.xlsx";
        // Danh sách worker và job
        List<Worker> workers = Arrays.asList(
            new Worker(1, "Worker A"),
            new Worker(2, "Worker B"),
            new Worker(3, "Worker C"),
            new Worker(4, "Worker D")
        );

        List<Job> jobs = Arrays.asList(
            new Job(101, "A"),
            new Job(102, "B"),
            new Job(103, "C"),
            new Job(104, "D")
        );

        // Đọc dữ liệu từ file Excel
        Map<Integer, Map<Integer, Integer>> timeMatrix =
        ExcelReader.readExcel(filePath, workers, jobs);

        // Lựa chọn thuật toán để giải bài toán
        Scanner scanner = new Scanner(System.in);
        System.out.println("Chọn thuật toán để giải bài toán phân công
        công việc:");
        System.out.println("1. Branch and Bound");
        System.out.println("2. Thuật toán Tham lam (Greedy Algorithm)");
        System.out.print("Nhập lựa chọn của bạn: ");
        int choice = scanner.nextInt();

        if (choice == 1) {
            // Giải bằng Branch and Bound
            AssignmentSolver.Result result =
            AssignmentSolver.branchAndBound(timeMatrix, workers, jobs);

            // Hiển thị kết quả
            System.out.println("\nPhân công tối ưu sử dụng Branch and
            Bound:");
            result.getAssignment().forEach((worker, job) ->
                System.out.println(worker.getName() + " -> " +
            job.getName()));
            System.out.println("Tổng thời gian tối ưu: " +
            result.getTotalTime());
        } else if (choice == 2) {
            // Giải bằng thuật toán tham lam
            GreedyAlgorithm.Result result =
            GreedyAlgorithm.solve(timeMatrix, workers, jobs);

            // Hiển thị kết quả
            System.out.println("\nPhân công công việc sử dụng Thuật toán
            Tham lam:");
        }
    }
}

```

```

        result.getAssignment().forEach((worker, job) ->
            System.out.println(worker.getName() + " -> " +
job.getName())
        );
        System.out.println("Tổng thời gian sử dụng Greedy: " +
result.getTotalTime());
    } else {
        System.out.println("Lựa chọn không hợp lệ. Thoát chương
trình.");
    }
}
}

```

3.2. Kiểm thử

3.2.1. Kiểm thử thuật toán tham lam

3.2.1.1. Ví dụ.

Công nhân/Công việc	Công việc 1	Công việc 2	Công việc 3	Công việc 4
Công nhân 1	10	19	8	15
Công nhân 2	10	18	7	17
Công nhân 3	13	16	9	14
Công nhân 4	12	19	13	19

Bước 1: Công nhân 1 chọn công việc 3 vì có thời gian ngắn nhất (8).

Bước 2: Công nhân 2 chọn công việc 1 vì thời gian là 10 (sau khi công việc 3 đã được gán).

Bước 3: Công nhân 3 chọn công việc 4 vì có thời gian ngắn nhất (14) trong các công việc chưa gán.

Bước 4: Công nhân 4 chỉ còn lại công việc 2 để chọn.

Kết quả:

- Công nhân 1 làm công việc 3 với thời gian 8.
- Công nhân 2 làm công việc 1 với thời gian 10.
- Công nhân 3 làm công việc 4 với thời gian 14.
- Công nhân 4 làm công việc 2 với thời gian 19.

Tổng thời gian: $8 + 10 + 14 + 19 = 51$.

3.2.1.2. Nhận xét về thuật toán:

1) Ưu điểm:

- **Đơn giản và dễ hiểu:** Thuật toán dựa trên cách tiếp cận tham lam (greedy), luôn chọn công việc có thời gian nhỏ nhất cho mỗi công nhân tại mỗi bước. Điều này giúp thuật toán dễ viết và dễ triển khai.
- **Thời gian chạy chấp nhận được:** Với độ phức tạp là $O(n^2)$ do vòng lặp tìm công việc có thời gian nhỏ nhất và gán nó cho công nhân, thuật toán có thể hoạt động tốt với số lượng công nhân và công việc không quá lớn.

2) Nhược điểm:

- **Không tối ưu toàn cục:** Mặc dù thuật toán này luôn tìm công việc nhanh nhất cho mỗi công nhân tại từng bước, nhưng nó có thể không cho ra giải pháp tối ưu toàn cục. Trong một số trường hợp, việc chọn một công việc khác (không phải công việc có thời gian nhỏ nhất) có thể giúp giảm tổng thời gian. Đó là hạn chế của các thuật toán tham lam.
- **Phụ thuộc vào cách chọn:** Thuật toán này không thử nhiều lựa chọn khác nhau mà chỉ theo một chiến lược duy nhất. Điều này có thể bỏ lỡ các trường hợp tối ưu hơn.

3) Tính khả thi:

- Với bài toán này, nếu số công nhân và công việc bằng nhau và không có công việc nào bị bỏ trống, thuật toán sẽ luôn hoàn thành với một phương án hợp lệ. Tuy nhiên, trong trường hợp số lượng công nhân hoặc công việc thay đổi, bạn sẽ cần điều chỉnh thuật toán để đảm bảo tính hợp lệ của bài toán.

3.2.2. Kiểm thử thuật toán nhánh cận

3.2.2.1. Ví dụ

Công nhân/Công việc	Công việc 1	Công việc 2	Công việc 3	Công việc 4
Công nhân 1	2	6	4	7
Công nhân 2	5	4	2	5
Công nhân 3	4	5	4	6

Công nhân/Công việc	Công việc 1	Công việc 2	Công việc 3	Công việc 4
Công nhân 4	5	5	3	4

Bước 1: Khởi tạo thuật toán, khi chưa có công nhân nào thực hiện công việc , ta có

TTG (Tổng thời gian) = 0,

CD (Cận dưới) = 11

CV CN	1	2	3	4
1	2	6	4	7
2	5	4	2	5
3	4	5	4	6
4	5	5	3	4

TTG=0
CD=11

Hình 21. Minh họa khi chưa phân chia công việc

Bước 2: Chọn công việc cho công nhân 1, ta có các trường hợp

- Công nhân 1 chọn CV 1

TTG = 2

CD = 11

- Công nhân 1 chọn CV 2

TTG = 6

CD = 15

- Công nhân 1 chọn CV 3

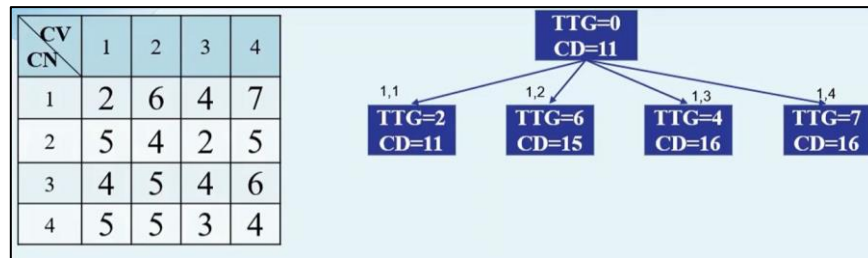
TTG = 4

CD = 16

- Công nhân 1 chọn CV 4

TTG = 7

CD = 16



Hình 22. Minh họa khi công nhân 1 chọn việc

Bước 3: Từ nhánh 1, 1 chọn công việc cho Công nhân 2, còn 3 công việc, ta có các trường hợp:

- Công nhân 2 chọn CV 2

TTG = 6

CD = 13

- Công nhân 2 chọn CV 3

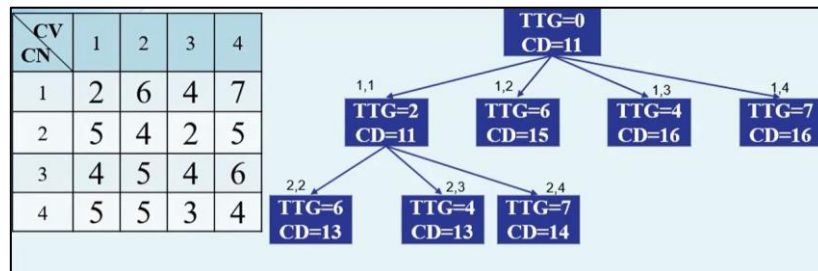
TTG = 4

CD = 13

- Công nhân 2 chọn CV 4

TTG = 7

CD = 15



Hình 23. Minh họa khi công nhân 2 chọn việc

Bước 4: Từ nhánh 2, 2 chọn công việc cho Công nhân 3, còn 2 công việc, ta có các trường hợp:

- Công nhân 3 chọn CV 3

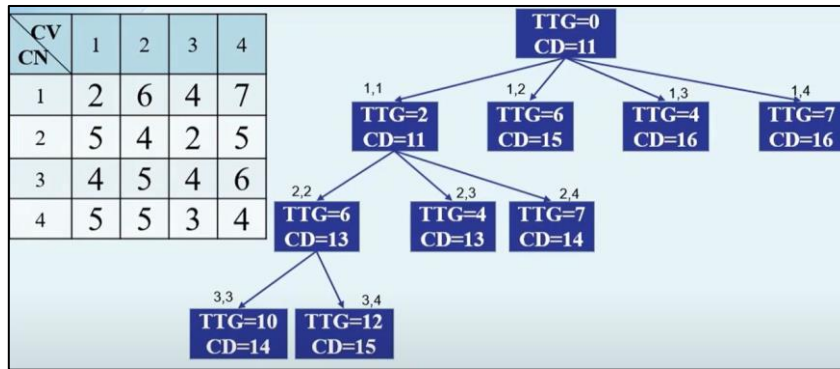
TTG = 10

CD = 14

- Công nhân 3 chọn CV 4

TTG = 12

CD = 15



Hình 24. Minh họa khi công nhân 3 chọn việc

Bước 5: Ở nhánh 3, 3 chọn công việc cho công nhân 4, còn 1 công việc, ta có:

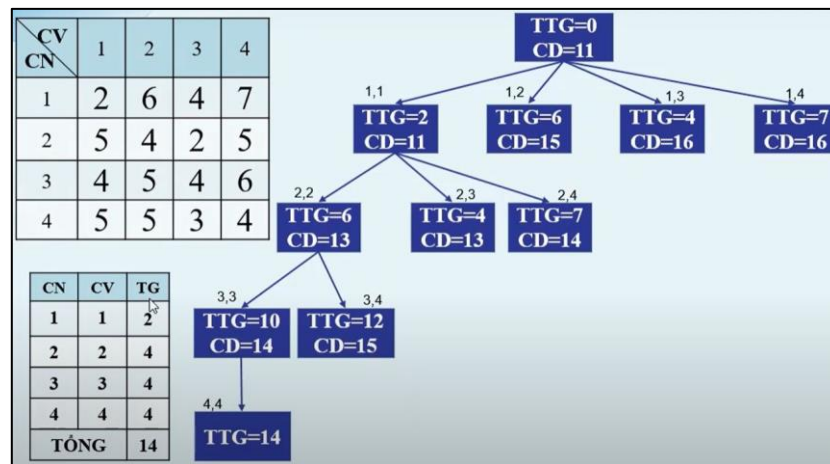
- Công nhân 4 chọn CV 4

TTG = 14

Không còn cận dưới vì không còn công nhân nhận công việc để xét

Ta có kết luận cho 1 nhánh lựa chọn công việc với tổng thời gian là 14 với

CN 1 làm CV 1, CN 2 làm CV 2, CN 3 làm CV 3, CN 4 làm CV 4



Hình 25. Minh họa khi công nhân 4 chọn việc

Bước 6: Ta nhận thấy rằng, tổng thời gian ở nhánh này vẫn còn lớn hơn ở nhánh 2, 3 vậy nên ta tiếp tục xét xem liệu rằng ở nhánh đó còn có thể lựa chọn công việc tối ưu hơn không

Ở nhánh 2,3 chọn công việc cho công nhân 3, còn 2 công việc, ta có các trường hợp:

- Công nhân 3 chọn CV 3

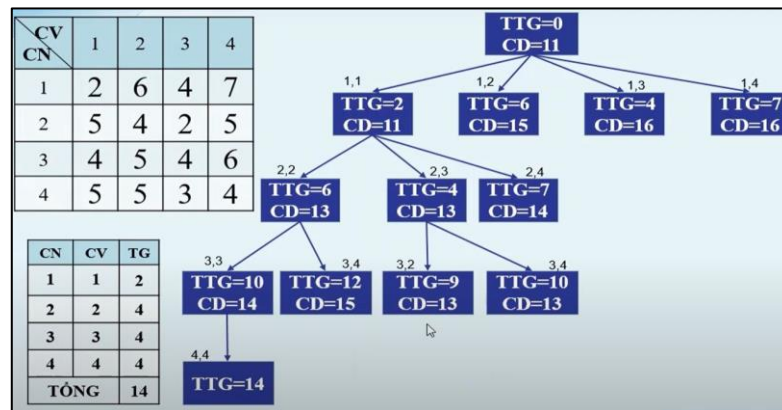
TTG = 9

CD = 13

- Công nhân 3 chọn CV 4

TTG = 10

CD = 13



Hình 26. Minh họa khi công nhân 3 chọn công việc khác

Bước 7: Ở nhánh 3,2 chọn công việc cho công nhân 4, còn 1 công việc, ta có trường hợp:

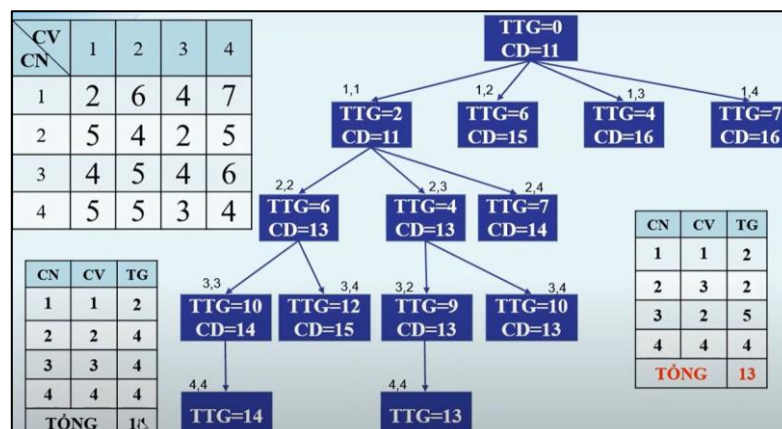
- Công nhân 4 chọn CV 4

TTG = 13

Không còn cần dưới vì không còn công nhân nhận công việc để xét

Ta có kết luận cho 1 nhánh lựa chọn công việc với tổng thời gian là 13 với

CN 1 làm CV 1, CN 2 làm CV 3, CN 3 làm CV 2, CN 4 làm CV 4



Hình 27. Minh họa kết thúc quá trình chọn công việc

Bước 8: Ta nhận thấy rằng không còn nhánh nào có cận dưới nhỏ hơn 13, vậy từ đây ta có thể kết luận rằng đây là lựa chọn tối ưu nhất cho bài toán này.

Ta đưa ra kết luận:

CN 1 làm CV 1, CN 2 làm CV 3, CN 3 làm CV 2, CN 4 làm CV 4

Với tổng thời gian là 13.

3.2.2.2. Nhận xét về thuật toán:

1) Ưu điểm:

- Đảm bảo tìm được lời giải tối ưu: Thuật toán Branch and Bound thực hiện kiểm tra tất cả các khả năng có thể xảy ra nhưng sử dụng cận để loại bỏ các nhánh không cần thiết. Điều này giúp thuật toán có khả năng tìm ra lời giải tối ưu toàn cục cho bài toán.
- Hiệu quả hơn so với thử tất cả các phương án: Nhờ vào việc cắt tỉa các nhánh không có tiềm năng, thuật toán giảm đáng kể số lượng trạng thái cần xét so với việc duyệt vét cạn hoàn toàn.
- Linh hoạt và có thể áp dụng cho nhiều bài toán: Thuật toán không bị giới hạn trong một cách chọn duy nhất, mà thay vào đó thử nhiều phương án khác nhau kết hợp với đánh giá cận, giúp nó có thể áp dụng vào các bài toán có cấu trúc phức tạp hơn.

2) Nhược điểm:

- Tốn thời gian và bộ nhớ trong trường hợp xấu: Mặc dù thuật toán có khả năng cắt tỉa nhánh, nhưng với bài toán có không gian tìm kiếm lớn, thời gian chạy và bộ nhớ cần sử dụng có thể tăng nhanh, đặc biệt khi không có cận tốt để loại bỏ các nhánh sớm.
- Độ phức tạp trong triển khai: So với thuật toán tham lam, thuật toán Branch and Bound yêu cầu việc tính toán cận trên và cận dưới, cũng như quản lý danh sách trạng thái và nhánh cần xét. Điều này làm cho thuật toán khó viết và triển khai hơn.

3) Tính khả thi:

- Thuật toán Branch and Bound đảm bảo sẽ tìm ra lời giải tối ưu nếu không gian trạng thái được duyệt hết và có cận phù hợp. Trong trường hợp số lượng công việc và công nhân lớn, thuật toán vẫn có thể hoạt động nhưng cần thêm các kỹ thuật tối ưu để giảm thiểu thời gian chạy.
- Với các bài toán dạng phân công công việc, thuật toán này có thể giải quyết được cả khi số lượng công nhân và công việc thay đổi, miễn là cấu trúc bài toán được xây dựng hợp lệ.

TỔNG KẾT

Trong bài nghiên cứu này, sau khi có những sự tìm hiểu kĩ càng và đi sâu từ các khái niệm đến các ví dụ và ứng dụng trong thực tiễn để giải quyết bài toán Phân Công Công Việc. Bằng cách sử dụng các thuật toán như Greedy, hay Brand and Bound, ở đây chúng em không chỉ đi vào áp dụng mà còn đưa ra những đánh giá, từ ưu đến nhược điểm trong từng bài toán, trường hợp khác nhau.

Thông qua quá trình nghiên cứu và triển khai, chúng em nhận thấy bài toán Phân công công việc có ý nghĩa quan trọng cả trong lý thuyết và thực tiễn, với ứng dụng trong tối ưu hóa nguồn lực, lập lịch sản xuất, quản lý nhân sự, và nhiều bài toán khác. Việc cài đặt và kiểm chứng các thuật toán không chỉ giúp hiểu rõ hơn về cách thức hoạt động mà còn cho phép đánh giá hiệu suất và tính hiệu quả dựa trên đặc điểm dữ liệu đầu vào và cấu trúc bài toán.

Kết quả nghiên cứu này đã tạo dựng một nền tảng vững chắc để áp dụng trong các tình huống thực tế, đồng thời mở ra những hướng phát triển mới nhằm cải tiến thuật toán hiện có hoặc tìm kiếm các phương pháp mới để giải quyết bài toán phân công công việc một cách hiệu quả hơn trong các hệ thống lớn và phức tạp hơn.

TÀI LIỆU THAM KHẢO

Linh, N. V (2020). *Cấu trúc dữ liệu và giải thuật*. Truy cập 25/10/2024, từ [Giải bài toán Phân công lao động bằng kỹ thuật Nhánh cận](#)

Linh, N. V (2020). *Cấu trúc dữ liệu và giải thuật*. Truy cập 25/10/2024, từ [Kỹ thuật Nhánh Cận \(Branch and Bound\) - YouTube](#)

Viblo Algorithm (2022). *Nhánh và Cận (Branch and Bound)*. Truy cập 25/10/2024, từ [Nhánh và Cận \(Branch and Bound\)](#)

David, M., Sheldon, H.J., Jason, S., , & Edward, S. *Branch-and-Bound Algorithms: Recent Advances in Searching, Branching, and Pruning*. Truy cập 23/10/2024, từ [Just a moment...](#)

Huy, P. Q (2018). *Cấu trúc dữ liệu và giải thuật – Cây nhị phân tìm kiếm*. Truy cập 24/10/2024, từ [Cấu trúc dữ liệu và giải thuật - Cây nhị phân tìm kiếm](#)

Phong, T. T (2016). *Thuật toán tham lam - Greedy Algorithm*. Truy cập 24/10/2024, từ [Thuật Toán Tham Lam - Greedy Algorithm - Viblo](#)

W3Schools. *DSA Greedy Algorithms*. Truy cập 24/10/2024, từ [DSA Greedy Algorithms](#)

Geeksforgeeks (2024). *Greedy Algorithm Tutorial*. Truy cập 26/10/2024, từ [Greedy Algorithm Tutorial - GeeksforGeeks](#)