ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

LÊ NHỰT NAM - PHẠM THỪA TIỂU THÀNH

BÀI TOÁN XẾP BA-LÔ VÀ ỨNG DỤNG

TIỂU LUẬN MÔ HÌNH TOÁN KINH TẾ

TP. Hồ Chí Minh - Năm 2024

ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

LÊ NHỰT NAM - PHẠM THỪA TIỂU THÀNH

BÀI TOÁN XẾP BA-LÔ VÀ ỨNG DỤNG

Ngành: Toán ứng dụng
Mã số: 84 60 112

NGƯỜI HƯỚNG DẪN KHOA HỌC
1. HDC: PGS. TS. Nguyễn Lê Hoàng Anh

TP. Hồ Chí Minh - Năm 2024

LỜI CAM ĐOAN

Chúng tôi cam đoan tiểu luận môn học Mô hình Toán trong Kinh Tế, với đề tài Bài toán xếp ba-lô và ứng dụng là báo cáo do Chúng tôi thực hiện dưới sự hướng dẫn của PGS. TS. Nguyễn Lê Hoàng Anh.

Những kết quả nghiên cứu và thực nghiệm của tiểu luận hoàn toàn trung thực và chính xác.

Học viên cao học
(Ký tên, ghi họ tên)

Lê Nhựt Nam

Phạm Thừa Tiểu Thành

LỜI CẢM ƠN

Lời đầu tiên, tôi xin phép gửi lời cảm ơn chân thành đến Thầy hướng dẫn của chúng tôi, PGS. TS. Nguyễn Lê Hoàng Anh - Giảng viên khoa Toán - Tin học, Trưởng bộ môn Tối ưu và Hệ thống, Trường Đại học Khoa học Tự nhiên, Đại học Quốc gia TP. HCM đã trực tiếp giảng dạy và giúp đỡ tận tình trong suốt quá trình học tập môn học và thực hiện tiểu luận. Nhờ vào những định hướng, và góp ý quý giá của thầy, chúng tôi đã hoàn thành trọn vẹn đề tài tiểu luận của mình.

Thành phố Hồ Chí Minh, những ngày mùa Thu năm 2024

Lê Nhựt Nam Phạm Thừa Tiểu Thành

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, CHỮ VIẾT TẮT

NP	Non-polynomial
P	Polynomial
NP-hard	Non-polynomial hard
NP-Complete	Non-polynomial complete

DANH MỤC CÁC THUẬT NGỮ

Additional constrains	Các ràng buộc bổ sung
Binary variable	Biến nhị phân
Bound	Chặn
Critical item	Phần tử chủ chốt
Dominated	Bị thống trị
Dynamic Programming	Quy hoạch động
Greedy	Tham lam
Knapsack	Ba-lô
Lower bound	Chặn dưới
Partial enumeration	Liệt kê riêng phần
Relaxtion	Nới lỏng
Space complexity	Độ phức tạp không gian
Time complexity	Độ phức tạp thời gian
Upper bound	Chặn trên
Weights	Trọng lượng

DANH MỤC CÁC BẢNG

DANH MỤC CÁC HÌNH VẼ, ĐỒ THỊ

CHƯƠNG 1

MỞ ĐẦU

1.1. Giới thiệu

Thuật ngữ "knapsack" có nguồn gốc từ thế kỷ 17, bắt nguồn từ từ tiếng Đức "knapzak", có nghĩa là "túi đựng thức ăn". Vào thời điểm đó, những người lính thường mang khẩu phần ăn trong một chiếc túi đeo trên lưng và phải đối mặt với tình huống khó xử khi lựa chọn những vật dụng thiết yếu nhất mà không vượt quá sức chứa của túi. Đây là lý do dẫn đến sự hình thành của bài toán này hay còn gọi là bài toán ba-lô.

Bài toán ba-lô được công nhận và nghiên cứu trong lĩnh vực Khoa học Máy tính vào giữa thế kỷ 20, khi các nhà nghiên cứu bắt đầu tập trung hơn trong việc nghiên cứu các bài toán tối ưu hóa và thuật toán để giải quyết chúng. Việc xây dựng và nghiên cứu bài toán này đã trở nên phổ biến như một phần của lĩnh vực tối ưu tổ hợp.

Vào cuối những năm 50, George Dantzig đã công khai công trình tiên phong trong việc nghiên cứu bài toán ba-lô. Sau đó, bài toán này được nghiên cứu chuyên sâu hơn và đã có nhiều khám phá trong việc ứng dụng nó trong công nghiệp và quản lý tài chính. Nhìn về mặt bản chất lý thuyết, dạng bài toán ba-lô thường xuất hiện khi sử dụng phép nớ lỏng cho nhiều biến thể của bài toán quy hoạch nguyên.

Bài toán knapsack đã được nghiên cứu chuyên sâu và rộng rãi bởi các nhà lý thuyết lẫn ứng dụng trong những thập kỷ vừa qua. Các nhà nghiên cứu lý thuyết quan tấm đến chúng chủ yếu bởi cấu trúc đơn giản của bài toán mà cho phép khám phá một số tính chất thú vị của tổ hợp và các ứng dụng của nó trong

việc giải quyết các bài toán tối ưu phức tạp thông qua việc mô hình thành một chuỗi các loại bài toán knapsack liên hoàn. Bên cạnh đó, dưới góc nhìn thực hành, các bài toán này có thể mô hình nhiều vấn đề trong công nghiệp: ngân sách vốn, xếp hàng, và cắt hàng tồn kho.

Mặc dù thoạt nhìn bài toán Knapsack có vẻ đơn giản, nhưng nó đã được chứng minh là thuộc nhóm các bài toán NP-đầy đủ, tức là không có thuật toán đa thức nào có thể giải quyết bài toán này một cách tối ưu trong mọi trường hợp. Điều này khiến bài toán Knapsack trở nên quan trọng và thú vị đối với cả những người nghiên cứu lý thuyết và những nhà thực hành tìm kiếm giải pháp tối ưu trong các tình huống thực tế.

Trong tiểu luận này, chúng ta sẽ đi sâu tìm hiểu về bản chất của bài toán Knapsack, các phương pháp giải quyết phổ biến, và những ứng dụng thực tiễn của nó. Đồng thời, chúng ta cũng sẽ xem xét các bài toán mở rộng và phức tạp hơn xuất phát từ bài toán Knapsack gốc.

1.2. Lý do chọn đề tài

Lý do lựa chọn bài toán Knapsack làm đề tài cho tiểu luận xuất phát từ sự quan trọng và tính ứng dụng rộng rãi của nó. Bài toán không chỉ giúp hiểu rõ hơn về các vấn đề tối ưu hóa, mà còn có những ứng dụng trong nhiều lĩnh vực đời sống. Trong công nghệ thông tin, bài toán Knapsack có thể được áp dụng trong quản lý tài nguyên máy tính, tối ưu hóa việc lưu trữ dữ liệu và bảo mật thông tin. Trong kinh tế và quản lý, bài toán này hỗ trợ việc ra quyết định liên quan đến phân bổ nguồn lực sao cho hiệu quả.

Ngoài ra, với sự phát triển của các thuật toán máy tính, việc tìm kiếm các giải pháp gần đúng hoặc giải pháp tối ưu hóa hiệu quả cho bài toán Knapsack ngày càng trở nên quan trọng. Nghiên cứu và hiểu rõ hơn về các phương pháp này sẽ mang lại nhiều giá trị trong việc phát triển các ứng dụng thực tiễn.

1.3. Mục tiêu của đề tài

Tiểu luận này hướng đến việc trả lời những câu hỏi chính sau:

- Bản chất của bài toán Knapsack là gì? Tại sao nó lại được coi là một bài toán ?
- Các phương pháp giải bài toán Knapsack phổ biến là gì, và mỗi phương pháp có những ưu điểm và nhược điểm nào?
- Bài toán Knapsack có những biến thể nào và các biến thể này được áp dụng ra sao trong thực tế?
- Có những ứng dụng thực tiễn nào của bài toán Knapsack trong các lĩnh vực của đời sống?

Mục tiêu của tiểu luận là cung cấp một cái nhìn tổng quan và chi tiết về bài toán Knapsack, từ lý thuyết đến ứng dụng, cũng như các phương pháp giải quyết hiệu quả cho các bài toán thực tế.

1.4. Phương pháp nghiên cứu

Tiểu luận sẽ sử dụng các phương pháp nghiên cứu sau để phân tích bài toán Knapsack:

- **Nghiên cứu lý thuyết:** Thu thập và phân tích các tài liệu, sách, bài báo khoa học liên quan đến bài toán Knapsack, các thuật toán giải và các ứng dụng thực tiễn.
- **Phân tích thuật toán:** Trình bày và đánh giá các thuật toán phổ biến như quét cạn (duyệt toàn bộ), (dynamic programming), và (greedy algorithm) trong việc giải bài toán Knapsack.
- **Ví dụ minh họa:** Sử dụng các ví dụ cụ thể để giải bài toán Knapsack bằng các phương pháp khác nhau, từ đó so sánh và đánh giá hiệu quả của từng phương pháp.

- **Thực nghiệm:** Hiện thực hoá thuật toán bằng ngôn ngữ lập trình.

1.5. Phân công công việc

STT	Công việc	Người thực hiện	Kết quả
1	Khởi tạo báo cáo	Nam	Hoàn thành
2	Soạn cơ sở lý thuyết	Thành	Hoàn thành
3	Mô hình bài toán	Thành	Hoàn thành
4	Mối liên hệ giữa dạng cực tiểu và dạng cực đại của bài toán	Nam	Hoàn thành
4	Nới lỏng Larange	Nam	Hoàn thành
5	Cải thiện các chặn cho bài toán	Thành	Hoàn thành
5	Thuật toán tham lam	Nam	Hoàn thành
5	Thuật toán quy hoạch động - khử các trạng thái bị thống trị	Thành	Hoàn thành
5	Thuật toán quy hoạch động - Thuật toán Horowitz-Sahni	Nam	Hoàn thành
5	Thuật toán quy hoạch động - Thuật toán Toth	Thành	Hoàn thành
5	Cài đặt các thuật toán bằng python	Thành	Hoàn thành
5	Kiểm thử và đánh giá thuật toán	Nam	Hoàn thành

Bảng 1.1: *Bảng phân công đồ án*

CHƯƠNG 2

CƠ SỞ LÝ THUYẾT

Chương này cung cấp nền tảng lý thuyết cho bài toán Knapsack, bao gồm các định nghĩa, phân loại và các ứng dụng lý thuyết của bài toán. Nó giúp người đọc nắm bắt được các khái niệm cơ bản trước khi đi sâu vào các phương pháp giải và ứng dụng thực tiễn.

2.1. Phát biểu bài toán

Bài toán Knapsack (còn gọi là bài toán cái ba-lô) là một trong những bài toán cơ bản trong lý thuyết tối ưu hóa tổ hợp. Đặt bài toán trong bối cảnh chọn các vật phẩm có giá trị và trọng lượng khác nhau để bỏ vào một cái túi có giới hạn dung lượng sao cho tổng giá trị của các vật phẩm trong túi là lớn nhất mà không vượt quá trọng lượng giới hạn. Bài toán được phát biểu như sau:

Cho một tập hợp gồm n vật phẩm, mỗi vật phẩm chứa hai thông số như sau:

- Trọng lượng của vật phẩm, ký hiệu là w_i.
- Giá trị của vật phẩm, ký hiệu là p_i.

Mục tiêu của chúng ta là chọn một số vật phẩm sao cho tổng trọng lượng của các vật phẩm được chọn không vượt quá dung lượng của túi W, và tổng giá trị của chúng là lớn nhất.

Bài toán này có thể được biểu diễn dưới dạng phương trình tối ưu hóa:

$$\max \sum_{i}^{n} p_i x_i$$

thoả mãn ràng buộc:

$$\sum_{i}^{n} w_i x_i \quad \text{sao cho} \quad x_i \in \{0, 1\}$$

Ở đây x_i là biến nhị phân, cho biết liệu vật phẩm i có được chọn (1) hay không (0).

2.2. Phân loại bài toán Knapsack

Bài toán Knapsack có nhiều biến thể dựa trên cách chọn vật phẩm và đặc điểm của các vật phẩm. Dưới đây là một số biến thể dựa trên bài toán gốc:

a. **Bài toán Knapsack 0/1 (0/1 Knapsack Problem):** Đây là biến thể phổ biến nhất của bài toán Knapsack. Mỗi vật phẩm chỉ có thể được chọn hoặc không chọn, tức là $x_i \in \{0, 1\}$. Không thể chọn một phần của vật phẩm. Đây là bài toán NP-đầy đủ.

b. **Bài toán Knapsack chia nhỏ (Fractional Knapsack Problem):** Trong bài toán này, ta có thể chọn một phần của vật phẩm, tức là $x_i \in [0, 1]$.

c. **Bài toán Knapsack nhiều chiều (Multi-dimensional Knapsack Problem - MKP)** bài toán Knapsack nhiều chiều mở rộng bài toán Knapsack cơ bản sang trường hợp có nhiều ràng buộc về tài nguyên (ví dụ: thời gian, dung lượng, ngân sách), thay vì chỉ giới hạn một dung lượng như trong bài toán Knapsack thông thường.

d. **Bài toán Knapsack đa mục tiêu (Multi-objective Knapsack Problem):** Trong bài toán này, ta không chỉ tối ưu hóa một giá trị (như tổng giá trị của các vật phẩm) mà còn phải tối ưu hóa nhiều mục tiêu khác nhau, chẳng hạn như cả giá trị và chi phí bảo trì.

Tuy có rất nhiều biến thể nhưng do bị giới hạn về thời gian nghiên cứu, do đó trong khuôn khổ của tiểu luận này, chúng tôi chỉ chọn bài toán 0/1 Knapsack

(bài toán gốc) để thực hiện nghiên cứu, phân tích và đánh giá, những biến thể khác sẽ là những bài toán sẽ được nghiên cứu trong tương lại.

2.3. Các khái niệm và định nghĩa cơ bản

- **Dung lượng túi (c):** Là giới hạn về tổng trọng lượng mà túi có thể chứa. Đây là ràng buộc chính trong bài toán Knapsack.
- **Trọng lượng (w_i):** Trọng lượng của mỗi vật phẩm i. Nếu tổng trọng lượng của các vật phẩm được chọn vượt quá dung lượng W, giải pháp sẽ không hợp lệ.
- **Giá trị (p_i):** Giá trị của mỗi vật phẩm i. Mục tiêu là tối đa hóa tổng giá trị của các vật phẩm được chọn.
- **Biến nhị phân (x_i):** Trong bài toán Knapsack 0/1, biến này thể hiện việc chọn hay không chọn vật phẩm i.
- **Bài toán P:** là lớp bài toán quyết định có thể được giải quyết trong thời gian đa thức.
- **Bài toán NP:** NP là lớp bài toán quyết định mà để xác nhận câu trả lời là "yes" của nó, có thể đưa ra bằng chứng ngắn gọn dễ kiểm tra.
- **Bài toán NP-đầy đủ:** Bài toán được gọi là NP-đầy đủ nếu nó thỏa mãn cả hai điều kiện sau:
  - Nó thuộc NP.
  - Nó là : Một bài toán A được gọi là NP-khó nếu như sự tồn tại thuật toán đa thức để giải nó kéo theo sự tồn tại thuật toán đa thức để giải mọi bài toán trong NP.

2.4. Các ứng dụng của bài toán Knapsack

Bài toán Knapsack xuất hiện trong nhiều tình huống thực tế đòi hỏi tối ưu hóa tài nguyên, chi phí, hoặc các nguồn lực khác. Dưới đây là một số ứng dụng điển hình:

- **Quản lý dự án:** Khi có nhiều nhiệm vụ với mức độ quan trọng và yêu cầu tài nguyên khác nhau, bài toán Knapsack giúp xác định các nhiệm vụ nào nên được thực hiện để tối đa hóa giá trị dự án trong khi không vượt quá tài nguyên sẵn có.
- **Tối ưu hóa kho hàng và vận tải:** Trong lĩnh vực vận tải, bài toán Knapsack giúp lựa chọn hàng hóa để vận chuyển với giá trị cao nhất mà không vượt quá trọng tải của xe.
- **Lập lịch công việc:** Trong lập lịch, bài toán Knapsack có thể được sử dụng để quyết định các công việc nào nên được thực hiện với mục tiêu tối đa hóa giá trị đầu ra trong thời gian giới hạn.
- **Quản lý tài chính cá nhân:** Khi cần phân bổ nguồn lực (tiền, thời gian, tài nguyên) vào nhiều khoản đầu tư khác nhau, bài toán Knapsack giúp tối đa hóa lợi nhuận mà không vượt quá ngân sách.
- **Bảo mật và mã hóa thông tin:** Bài toán Knapsack cũng được áp dụng trong các thuật toán bảo mật như , nơi các tập hợp trọng số và giá trị được sử dụng để mã hóa và giải mã thông tin.

CHƯƠNG 3

MÔ HÌNH BÀI TOÁN 0-1 KNAPSACK

3.1. Mô hình bài toán

Bài toán 0-1 Knapsack hay Knapsack nhị phân hay ba-lô nhị phân (KP): cho trước một tập n *mặt hàng* và một chiếc *ba-lô* với:

$$p_j = \text{lợi nhuận (profit) của mặt hàng } j, \tag{3.1}$$

$$w_j = \text{trọng số (weight) của mặt hàng } j, \tag{3.2}$$

$$W = \text{sức chứa (capcity) của ba-lô}, \tag{3.3}$$

cần lựa chọn một tập con của các mặt hàng sao cho

$$\text{maximize} \quad z = \sum_{j=1}^{n} p_j x_j \tag{3.4}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{3.5}$$

$$x_j = 0 \quad \text{hoặc} \quad 1, \quad j \in N = \{1, \ldots, n\} \tag{3.6}$$

trong đó:

$$x_j = \begin{cases} 1 & \text{nếu vật phẩm } j \text{ được chọn;} \\ 0 & \text{ngược lại.} \end{cases} \tag{3.7}$$

Knapsack là bài toán NP-đầy đủ. Để chứng minh điều này, chúng ta sẽ đi chứng minh thông qua hai bước: Chứng minh Knapsack thuộc lớp NP và là NP-khó.

- **Knapsack thuộc NP:** Giả sử chúng ta đã có một tập con các vật phẩm S được chọn. Chúng ta cần kiểm tra hai điều kiện:

  1. Tổng trọng lượng của các vật phẩm trong "S không vượt quá giới hạn W Tính tổng trọng lượng của các vật phẩm $\sum_{i \in S} w_i$ và kiểm tra xem tổng này có nhỏ hơn hoặc bằng W không.
  2. Tính tổng giá trị của tập con: Tính tổng giá trị $\sum_{i \in S} w_i$ và xác nhận đây là tổng giá trị được cung cấp.

  Cả hai bước trên đều có thể thực hiện trong thời gian tuyến tính theo số lượng vật phẩm n (bằng cách cộng dồn các trọng lượng và giá trị), do đó quá trình xác minh có thể thực hiện trong thời gian đa thức. Vì vậy, Knapsack thuộc lớp NP.

- **Knapsack là NP-khó:** Để chứng minh Knapsack là NP-khó, chúng ta phải chứng minh rằng mọi bài toán trong NP có thể được quy giảm về bài toán Knapsack trong thời gian đa thức, hoặc ít nhất là một bài toán NP-đầy đủ có thể quy về Knapsack. Một cách để chứng minh điều này là thực hiện quy giảm từ bài toán Subset Sum – một bài toán đã được chứng minh là NP-đầy đủ – về bài toán Knapsack.

  Bài toán Subset Sum được phát biểu như sau:

  – Đầu vào: Một tập hợp các số nguyên dương $S = a_1, a_2, \ldots, a_n$ và một số nguyên dương T.

  – Đầu ra: Tìm một tập con của S sao cho tổng của các phần tử trong tập con bằng chính xác T.

  Bài toán Subset Sum là một trường hợp đặc biệt của bài toán Knapsack khi tất cả các giá trị p_i của các vật phẩm bằng chính trọng lượng w_i. Do đó, bài toán Subset Sum có thể được chuyển đổi thành bài toán Knapsack bằng cách đặt giá trị của mỗi vật phẩm bằng trọng lượng của nó và đặt giới hạn trọng lượng W bằng T. Nói cách khác:

– Đặt giá trị $w_i = w_i$ cho mỗi vật phẩm.

– Đặt $c = T$.

Bây giờ, mục tiêu của bài toán Knapsack là tìm tập con các vật phẩm có tổng trọng lượng không vượt quá T, với tổng giá trị cũng tối đa. Trong trường hợp này, vì giá trị bằng trọng lượng, bài toán trở thành tìm tập con các trọng lượng mà tổng của chúng bằng đúng T, chính là bài toán Subset Sum. Vì Subset Sum đã được chứng minh là NP-đầy đủ, và Subset Sum có thể được quy giảm về Knapsack trong thời gian đa thức, điều này chứng tỏ rằng Knapsack là NP-khó.

3.2. Mối liên hệ giữa dạng cực tiểu và dạng cực đại của bài toán

Không mất tính tổng quát, ta giả sử rằng:

$$p_j, w_j \text{ và } c \text{ là các số nguyên dương,} \tag{3.8}$$

$$\sum_{j=1}^{n} w_j > c, \tag{3.9}$$

$$w_j \leq c \text{ với } j \in N. \tag{3.10}$$

Nếu giả sử trong Biểu thức (3.8) bị vi phạm, các phân số có thể được xử lý bằng cách nhân với một hệ số thích hợp, trong khi các giá trị không dương có thể được xử lý như sau (Glover, 1965):

1. Với mỗi $j \in N^0 = \{j \in N : p_j \leq 0, w_j \geq 0\}$, gán $x_j := 0$;

2. Với mỗi $j \in N^1 = \{j \in N : p_j \geq 0, w_j \leq 0\}$, gán $x_j := 1$;

3. Đặt $N^- = \{j \in N : p_j < 0, w_j < 0\}$, $N^+ = N \setminus (N^0 \cup N^1 \cup N^-)$ và

$$\begin{cases} y_j = 1 - x_j, \bar{p}_j = -p_j, \bar{w}_j = -w_j \quad \text{với} \quad j \in N^-, \\ y_i = x_j, \bar{p}_j = p_j, \bar{w}_j = w_j \quad \text{với} \quad j \in N^+; \end{cases} \tag{3.11}$$

4. Giải bài toán

$$\text{maximize} \quad z = \sum_{j \in N^- \cup N^+}^{n} \bar{p}_j y_j + \sum_{j \in N^1 \cup N^-}^{n} p_j \tag{3.12}$$

$$\text{subject to} \quad \sum_{j \in N^- \cup N^+}^{n} \bar{w}_j x_j \leq c - \sum_{j \in N^1 \cup N^-}^{n} w_j, \tag{3.13}$$

$$y_j = 0 \quad \text{hoặc1}, \quad j \in N^- \cup N^+. \tag{3.14}$$

Nếu dữ liệu đầu vào vi phạm giả sử (3.9), thì hiển nhiên, $x_j = 1$ với mọi $j \in N$; nếu chúng vi phạm giả sử (3.10), thì $x_j = 0$ với mỗi j mà thỏa $w_j > c$.

Trừ khi có quy định khác, ta luôn giả định rằng các mặt hàng được sắp thứ tự theo chiều tăng dần giá trị của lợi nhuận trên mỗi đơn vị trong số, tức là:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}. \tag{3.15}$$

Nếu không phải trường hợp này, lợi nhuận và trọng số có thể được tái chỉ số trong thời gian $O(n \log n)$ thông qua bất kỳ thủ tục sắp xếp nào.

Cho trước bất kỳ thể hiện bài toán I nào, ta đặt giá trị của nghiệm tối ưu là $z(I)$ hoặc nếu không có nhập nhằng gì xảy ra, ta có thể đặt đơn giản là z.

Bài toán KP luôn được xem xét trong dạng cực đại. Phiên bản cực tiểu của bài toán này,

$$\text{minimize} \quad \sum_{j=1}^{n} p_j y_j \tag{3.16}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j y_j \geq q, \tag{3.17}$$

$$y_j = 0 \quad \text{hoặc} \quad 1, \quad j \in N \tag{3.18}$$

có thể dễ dàng biến đổi thành dạng cực đại tương đương bằng cách đặt $y_j = 1 - x_j$ và giải (3.4), (3.5), (3.6) với $c = \sum_{j=1}^{n} w_j - q$. Đặt $z\,\text{max}$ là giá trị nghiệm của

bài toán, thì bài toán cực tiểu có giá trị nghiệm $z\min = \sum_{j=1}^{n} p_j - z\max$.

Trong thực tế, việc áp dụng các thuật toán để tìm nghiệm tối ưu cho bài toán Knapsack 0/1 thường rất khó khăn, hoặc nếu có thể giải được thì độ phức tạp của thuật toán rất lớn. Vì vậy, bài toán Knapsack 0/1 thường được quy về bài toán tìm nghiệm tối ưu cho bài toán Fractional Knapsack, một biến thể cho phép chia nhỏ các vật phẩm. Thay vì phải chọn toàn bộ hoặc không chọn một vật phẩm (như trong Knapsack 0/1), ta có thể chọn một phần của vật phẩm.

Việc áp dụng bài toán mang lại nhiều lợi ích trong tối ưu hóa, đặc biệt trong những trường hợp mà việc chia nhỏ vật phẩm là hợp lý. Những lợi ích chính bao gồm:

- Giải pháp tối ưu có thể đạt được thông qua thuật toán tham lam.
- Độ phức tạp tính toán thấp hơn so với bài toán Knapsack 0/1.
- Tính linh hoạt cao hơn trong việc tối ưu hóa, vì có thể chọn từng phần của vật phẩm thay vì toàn bộ.
- Giải pháp xấp xỉ tốt cho bài toán Knapsack 0/1, vì bài toán Fractional Knapsack cung cấp một giá trị gần với nghiệm tối ưu của Knapsack 0/1. Nghiệm này có thể làm cơ sở cho các phương pháp tìm kiếm nhánh cận hoặc quy hoạch động.
- Fractional Knapsack có nhiều ứng dụng thực tế, chẳng hạn như trong vận chuyển hàng hóa, phân bổ tài nguyên, và đầu tư tài chính, khi việc chia nhỏ các nguồn lực là hợp lý.

Chính vì nghiệm của phương pháp này xấp xỉ với nghiệm của bài toán 0/1 Knapsack, do đó, trong chương này chúng ta sẽ tập trung vào việc phân tích các phương pháp tìm chặn trên cho bài toán Fractional Knapsack. Sau khi có kết quả của chặn trên, ta sẽ dùng nó như là điều kiện để loại bớt các tổ hợp không phù hợp và áp dụng thuật giải nhánh cận hoặc quy hoạch động để tìm lời giải tối ưu cho bài toán Knapsack 0/1.

3.3. Phương pháp nới lỏng và chặn trên bài toán

3.3.1. Quy hoạch tuyến tính nới lỏng và chặn Dantzig

Một trong những nới lỏng tự nhiên đầu tiên nhất của KP là (linear programming relaxation), tức là bài toán knapsack liên tục $C(KP)$ có được từ Biểu thức (3.4), (3.5), (3.6) bằng cách loại bỏ các ràng buộc toàn vẹn trên x_j:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{3.19}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c, \tag{3.20}$$

$$0 \leq x_j \leq 1, \quad j = 1, \ldots, n. \tag{3.21}$$

Giả sử rằng các mặt hàng được sắp thứ tự theo như Biểu thức (3.15), được thêm vào ba-lô liên tiếp nhau cho đến khi mặt hàng đầu tiên s được tìm thấy mà không khớp. Ta gọi nó là *mặt hàng chủ chốt* hay *phần tử chủ chốt (critical item)*, tức là:

$$s = \min \left\{ j : \sum_{i=1}^{j} w_i > c \right\}, \tag{3.22}$$

và để ý rằng, bởi vì giả định (3.8), (3.9), (3.10), ta có $1 < s \leq n$. Thì bài toán $C(KP)$ có thể được giải thông qua một tính chất được thiết lập bởi Dantzig (1957) mà được phát biểu như sau đây.

Định lý 1. Nghiệm tối ưu $\bar{x}$ của $C(KP)$ là

$$\bar{x}_j = 1 \quad \text{với} \quad j = 1, \ldots, s-1, \tag{3.23}$$

$$\bar{x}_j = 0 \quad \text{với} \quad j = s+1, \ldots, n, \tag{3.24}$$

$$\bar{x}_s = \frac{\bar{c}}{w_s}, \tag{3.25}$$

trong đó:

$$\bar{c} = c - \sum_{j=1}^{s-1} w_j. \tag{3.26}$$

Chứng minh. Để ý rằng bất kỳ nghiệm tối ưu x nào của $C(KP)$ phải cực đại, có nghĩa là

$$\sum_{j=1}^{n} w_j x_j = c$$

Không mất tính tổng quát, giả định rằng $p_j/w_j > p_{j+1}/w_{j+1}$ với tất cả j. Đặt x^* là nghiệm tối ưu của $C(KP)$. Giả sử rằng, $x_k^* < 1$ với một số $k < s$ nào đó, thì ta phải có $q_q^* > \bar{x}_q$ tại ít nhất một item $q \geq s$. Cho trước một $\epsilon > 0$ đủ lớn, ta có thể tăng giá trị x_k^* bởi lượng ϵ này và giảm x_q^* bởi $\epsilon w_k/w_q$, thì giá trị của hàm mục tiêu của $\epsilon(p_k - p_q w_k/w_q)$ phải dương bởi vì $p_k/w_k > p_q/w_q$. Điều này là mâu thuẫn với giả thiết.

Lập luận tương tự, ta có thể chứng minh $x_k^* > 0$ với một số $k > s$ là hoàn toàn không thể. Do đó, $\bar{x}_s = \bar{c}/w_s$ phải là cực đại.

Chứng minh hoàn tất. □

Giá trị nghiệm tối ưu của $C(KP)$ như sau:

$$z(C(KP)) = \sum_{j=1}^{s-1} p_j + \bar{c}\frac{p_s}{w_s} \tag{3.27}$$

Bởi vì tính toàn vẹn của p_j và x_j, do đó một chặn trên hợp lý $z(KP)$ là:

$$U_1 = \lfloor z(C(KP)) \rfloor = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c}\frac{p_s}{w_s} \right\rfloor, \tag{3.28}$$

trong đó $\lfloor a \rfloor$ đại diện cho số nguyên lớn nhất không vượt quá a.

Tỷ lệ hiệu suất trong trường hợp xấu nhất của U_1 là $\rho(U_1) = 2$. Ta có thể dễ dàng chứng minh được thông qua việc đánh giá Biểu thức (3.28), $U_1 < \sum_{j=1}^{s-1} p_j +$

p_s. Cả $\sum_{j=1}^{s-1} p_j$ và p_s là các giá trị khả thi cho KP, do đó không thể lớn hơn giá trị tối ưu z, do đó mà với bất kỳ thể hiện nào, $U_1/z < 2$. Để thấy rằng đánh giá $\rho(U_1)$ là chặt, ta xem xét chuỗi các bài toán với $n = 2, p_1 = w_1 = p_2 = w_2 = k$ và $c = 2k - 1$ với $U_1 = 2k - 1$ và $z = k$, nên U_1/z có thể nhận giá trị bất kỳ gần với 2 với giá trị k đủ lớn.

Rõ ràng tính toán của $z(C(KP))$ dựa trên chặn U_1 của Dantzig tốn thời gian $O(n)$ nếu các mặt hàng đã được sắp thứ tự. Nếu không, tính toán vẫn có thể là $O(n)$ nếu ta sử dụng thủ tục tìm kiếm phần tử chủ chốt sau đây.

3.3.2. Tìm kiếm phần tử chủ chốt trong thời gian $O(n)$

Với mỗi $j \in N$, định nghĩa $r_j = p_j/w_j$. (critical ratio) r_s có thể được xác định bởi việc quyết định một phân hoạch của N thành $J1 \cup JC \cup J0$ sao cho

$$r_j > r_s, \quad \text{với} \quad j \in J1, \tag{3.29}$$

$$r_j = r_s, \quad \text{với} \quad j \in JC, \tag{3.30}$$

$$r_j < r_s, \quad \text{với} \quad j \in J0, \tag{3.31}$$

và

$$\sum_{j \in J1} w_j \leq c < \sum_{j \in J1 \cup JC} w_j, \tag{3.32}$$

Thủ tục này được đề xuất bởi Balas và Zemel vào năm 1980, quyết định dần $J1$ và $J0$ bằng cách sử dụng phép lặp. Tại mỗi vòng lặp, một giá trị tạm thời λ để phân hoạch một tập hợp các mặt hàng "free" hiện tại $N \setminus (J1 \cup J0)$. Khi phân hoạch cuối cùng đã được xác định, phần tử chủ chốt s được xác định bằng cách lắp vào sức chứa còn lại của ba-lô $c - \sum_{j \in J1} w_j$ với các mặt hàng trong JC theo bất kỳ thứ tự nào.

Việc tìm kiếm trung vị của m phần tử tốn thời gian là $O(m)$, thế nên mỗi vòng lặp "while" tốn $O(|JC|)$. Bởi vì ít nhất một nửa số lượng phần tử của JC được lược bỏ tại mỗi vòng lặp, độ phức tạp thời gian tổng thể của thủ tục là

Algorithm 1: Thủ tục CRITICAL_ITEM

Input: $n, c, (p_j), (w_j)$
Output: s

```
1  J1 := ∅
2  J0 := ∅
3  JC := {1, ..., n}
4  c̄ := c
5  partition := "no"
6  while partition := "no" do
7      Xác định median λ của các giá trị trong R = {p_j/w_j : j ∈ JC}
8      G := {j ∈ JC : p_j/w_i > λ}
9      L := {j ∈ JC : p_j/w_i < λ}
10     E := {j ∈ JC : p_j/w_i = λ}
11     c' := Σ_{j∈E} w_j
12     c'' := c' + Σ_{j∈E} w_j
13     if c' > c̄ then
           /* giá trị λ quá nhỏ                                    */
14         J0 := J0 ∪ L ∪ E
15         JC := G
16     else
           /* giá trị λ quá lớn                                    */
17         J1 := J1 ∪ G ∪ E
18         JC := L
19         c̄ := c̄ − c''
20 J1 := J1 ∪ G
21 J0 := J0 ∪ L
22 JC := E(= {e_1, e_2, ..., e_q})
23 c̄ := c̄ − c'
24 σ := min{j : Σ_{i=1}^{j} w_{e_i} > c̄}
25 s := e_σ
```

$O(n)$.

Nghiệm của $C(KP)$ có thể được quyết định ngay sau đó bởi:

$$\bar{x}_j = 1 \quad \text{với} \quad j \in J1 \cup \{e_1, e_2, \ldots, e_{\sigma-1}\}; \tag{3.33}$$

$$\bar{x}_j = 0 \quad \text{với} \quad j \in J0 \cup \{e_{\sigma+1}, \ldots, e_q\}; \tag{3.34}$$

$$\bar{x}_s = \left(c - \sum_{j \in N \setminus \{s\}} w_j \bar{x}_j \right) / w_s. \tag{3.35}$$

3.3.3. Nới lỏng Larange

Một cách thay thế để nới lỏng KP là sử dụng tiếp cận Larangian. Cho trước một nhân tử không âm λ, (Lagrangian relaxation) của KP ($L(KP, \lambda)$) là:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j + \lambda \left(c - \sum_{j=1}^{n} w_j x_j \right) \tag{3.36}$$

$$\text{subject to} \quad x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n. \tag{3.37}$$

Hàm mục tiêu có thể được viết lại như sau:

$$z(L(KP, \lambda)) = \sum_{j=1}^{n} \widetilde{p}_j x_j + \lambda c, \tag{3.38}$$

trong đó $\widetilde{p}_j = p_j - \lambda w_j$ với $j = 1, \ldots, n$ và nghiệm tối ưu của $L(KP, \lambda)$ có thể dễ dàng có được trong thời gian $O(n)$:

$$\widetilde{x}_j = \begin{cases} 1 & \text{nếu} \quad \widetilde{p}_j > 0, \\ 0 & \text{nếu} \quad \widetilde{p}_j < 0, \end{cases} \tag{3.39}$$

Khi $\widetilde{p}_j = 0$, giá trị của $\widetilde{x}_j$ là không xác định. Do đó, bởi định nghĩa $J(\lambda) = \{j :$

$p_j/w_j > \lambda\}$, giá trị nghiệm của $L(KP, \lambda)$ là:

$$z(L(KP, \lambda)) = \sum_{j \in J(\lambda)}^{n} \widetilde{p}_j x_j + \lambda c. \tag{3.40}$$

Với bất kỳ $\lambda \geq 0$, tồn tại một chặn trên $z(KP)$ mà có thể không thể tốt hơn chặn U_1 của Dantzig. Thật vậy, Biểu thức (3.39) cũng cho ra nghiệm của nới của $L(KP, \lambda)$, thế nên:

$$z(L(KP, \lambda)) = z(C(L(KP, \lambda))) \geq z(C(KP)). \tag{3.41}$$

Giá trị của λ sinh ra giá trị nhỏ nhất của $z(L(KP, \lambda))$ là $\lambda^* = p_s/w_s$. Thật vậy, với giá trị này, ta có $\widetilde{p}_j \geq 0$ với $j = 1, \ldots, s-1$ và $\widetilde{p}_j \leq 0$ với $j = s, \ldots, n$, thế nên $J(\lambda^*) \subseteq \{1, \ldots, s-1\}$. Do đó mà, $\widetilde{x}_j = \bar{x}_j$ với $j \in N \setminus \{s\}$ (trong đó $\bar{x}_j$ được định nghĩa bởi Định lý 1, và tử các Biểu thức (3.38) - (3.39), $z(L(KP, \lambda^*)) = \sum_{j=1}^{s-1}(p_j - \lambda^* w_j) + \lambda^* c = z(C(KP))$. Và cũng để ý rằng, với $\lambda = \lambda^*$, $\widetilde{p}_j$ trở thành:

$$p_j^* = p_j - w_j \frac{p_s}{w_s}; \tag{3.42}$$

Ta thấy $|p_j^*|$ là mức giảm mà ta thu được trong Biểu thức của $z(L(KP, \lambda^*))$ bởi thiết lập $\widetilde{x}_j = 1 - \widetilde{x}_j$, và do đó một chặn dưới (lower bound) tương ứng với sự giảm này trong giá trị nghiệm liên tục (bởi vì giá trị tối ưu λ thay đổi bởi sự áp đặt của các điều kiện trên). Giá trị $|p_j^*|$ sẽ rất hữu ích trong các thiết lập nhằm cải thiện các chặn của bài toán.

3.4. Cải thiện các chặn bài toán

Ta xem xét chặn trên thống trị bởi Dantzig, nó rất hữu ích trong việc cải thiện hiệu quả trung bình của các thuật toán cho KP. Bởi vì tính chất thống trị này, tỷ lệ hiệu suất trường hợp xấu nhất của các chặn này là nhiều nhất 2. Thực vậy, nó chính xác bằng 2 vì có thể dễ dàng xác minh thông qua một chuỗi các

ví tương tự mà có $p_j = w_j$ với mọi j (thế nên các chặn nhận giá trị tầm thường c).

3.4.1. Chặn từ các ràng buộc bổ sung

Martello và Toth thu được một chặn trên thống trị đầu tiên so với Dantzig bằng cách áp đặt tính toàn vẹn của biến chủ chốt (critical variable) x_s.

Định lý 2 (Martello và Toth, 1977)**.** Đặt:

$$U^0 = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c}\frac{p_{s+1}}{w_{s+1}} \right\rfloor \tag{3.43}$$

$$U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}} \right\rfloor \tag{3.44}$$

trong đó s và $\bar{c}$ là các giá trị được định nghĩa bởi Biểu thức (3.22) và (3.26). Thì

(a) một chặn trên $z(KP)$ là

$$U_2 = \max(U^0, U^1); \tag{3.45}$$

(b) với bất kỳ thể hiện của KP, ta có $U_2 \leq U_1$.

Chứng minh. Chứng minh ý (i). Bởi vì x_s không thể nhận giá trị phân số, nghiệm tối ưu của bài toán KP có thể thu được từ nghiệm liên tục $\bar{x}$ của $C(KP)$ mà không cần thêm phân tử s (tức là, áp đặt $\bar{x}_s = 0$) hay thêm nó (tức là, áp đặt $\bar{x}_s = 1$) và do đó ta loại bỏ ít nhất một trong $s-1$ item đầu tiên. Trong một trường hợp tổng quát hơn, nghiệm tối ưu không thể vượt quá U^0 mà tương ứng với trường hợp lấp đầy phần dư $\bar{c}$ với các item có giá trị phù hợp nhất của p_j/w_j (tức là p_{s+1}/w_{s+1}. Hơn nữa, nó không thể vượt quá U^1 trong khi đó ta giả định rằng item cần được loại bỏ có giá trị

nhỏ nhất là w_j (tức là $w_s - \bar{c}$ và giá trị xấu nhất có thể của p_j/w_j (tức là p_{s-1}/w_{s-1}).

Chứng minh ý (ii) Đánh giá $U^0 \leq U^1$ có thể suy ra trực tiếp từ Biểu thức (3.28), (3.43) và (3.15). Để chứng minh $U^1 \leq U_1$ cũng thỏa mãn, ta để ý rằng $p_s/w_s \leq p_{s-1}/w_{s-1}$ (từ Biểu thức (3.15)) và $\bar{c} < w_s$ (từ biểu thức (3.22), (3.26)). Thế nên

$$(\bar{c} - w_s)\left(\frac{p_s}{w_s} - \frac{p_{s-1}}{w_{s-1}}\right)$$

và bằng cách biến đổi đại số, ta có:

$$\bar{c}\frac{p_s}{w_s} \geq p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}},$$

Suy ra điều phải chứng minh.

Chứng minh hoàn tất. □

Độ phức tạp thời gian cho việc tính toán U_2 là $O(n)$ khi một phần tử chủ chốt đã biết trước.

Ví dụ 1. Xem xét một thể hiện của bài toán KP được định nghĩa như sau:

$$\begin{aligned} n &= 8 \\ (p_j) &= (15, 100, 90, 60, 40, 15, 10, 1) \\ (w_j) &= (2, 20, 20, 30, 40, 30, 60, 10) \\ c &= 102. \end{aligned}$$

Nghiệm tối ưu là $x = (1, 1, 1, 1, 0, 1, 0, 0)$ có giá trị $z = 280$. Từ Biểu thức

(3.22), ta có $s = 5$. Do vậy:

$$U_1 = 265 + \left\lfloor 30 \times \frac{40}{40} \right\rfloor = 295$$
$$U^0 = 265 + \left\lfloor 30 \times \frac{15}{30} \right\rfloor = 280$$
$$U^1 = 265 + \left\lfloor 40 - 10 \times \frac{60}{30} \right\rfloor = 285$$
$$U_2 = 285$$

Chặn Martello và Toth có thể được khám phá sâu hơn nữa để tính toán toán các chặn trên ngặt hơn U_2. Ta có thể đạt được bằng cách thay thế các giá trị U^0 và U^1 với các giá trị chặt hơn, đặt là $\overline{U}^0$ và $\overline{U}^1$ mà dùng (inlusion) và (exclusion) item s cẩn thận hơn. Hudson (1977) đã đề xuất cách tính toán $\overline{U}^1$ với giá trị nghiệm của nới lỏng liên tục của KP với các ràng buộc bổ sung $x_s = 1$. Fayard và Plateau (1982) cùng với Villela và Bornstein (1983) độc lập đề xuất cách tính toán $\overline{U}^0$ với giá trị nghiệm của $C(KP)$ với các ràng buộc bổ sung $x_s = 0$.

Bởi định nghĩa các phần tử chủ chốt lần lượt là $\sigma^1(j)$ và $\sigma^0(j)$ khi ta đặt $x_j = 1(j \geq s)$ và $x_j = 0(j \leq s)$ thì

$$\sigma^1(j) = \min \left\{ k : \sum_{i=1}^{k} w_i > c - w_j \right\}, \tag{3.46}$$

$$\sigma^0(j) = \min \left\{ k : \sum_{i=1, i \neq j}^{k} w_i > c \right\}, \tag{3.47}$$

$$\tag{3.48}$$

ta thu được

$$\overline{U}^0 = \sum_{j=1, j\neq s}^{\sigma^0(s)-1} p_j + \left\lfloor \left(c - \sum_{j=1, j\neq s}^{\sigma^0(s)-1} w_j \frac{p_{\sigma^0(s)}}{w_{\sigma^0(s)}} \right) \right\rfloor, \tag{3.49}$$

$$\overline{U}^1 = p_s + \sum_{j=1}^{\sigma^1(s)-1} p_j + \left\lfloor \left(c - w_s - \sum_{j=1}^{\sigma^1(s)-1} w_j \frac{p_{\sigma^1(s)}}{w_{\sigma^1(s)}} \right) \right\rfloor, \tag{3.50}$$

$$\tag{3.51}$$

và ta có một chặn trên mới

$$U_3 = \max\left(\overline{U}^0, \overline{U}^1\right). \tag{3.52}$$

Hiển nhiên, ta có:

(a) $\overline{U}^0 \leq U^0$ và $\overline{U}^1 \leq U^1$, thế nên $U_3 \leq U_2$;

(b) Độ phức tạp thời gian cho tính toán giá trị chặn U_3 bằng với khi tính toán chặn U_1 và U_2, $O(n)$.

Ví dụ 2. Từ các Biểu thức (3.46)-(3.50), ta có:

$$\sigma^0(5) = 7, \overline{U}^0 = 280 + \left\lfloor 0 \times \frac{10}{60} \right\rfloor = 280;$$
$$\sigma^1(5) = 4, \overline{U}^1 = 40 + 205 + \left\lfloor 20 \times \frac{60}{30} \right\rfloor = 285;$$
$$U_3 = 285.$$

3.4.2. Chặn từ các nới lỏng Larange

Một chặn tính toán khác trong thời gian $O(n)$ có thể được mô tả thông qua nới lỏng Larangian của bài toán này. Nhắc lại:

$$z(C(KP)) = z(L(KP, \lambda^*))$$

và $|p_j^*$ là một chặn dưới giảm của $z(C(KP))$ tương ứng với sự thay đổi của biến thứ j từ $\bar{x}_j$ thành $1 - \bar{x}_j$.

Muller-Merbach (1977) đã nhận tháy rằng để mà thu được một nghiệm nguyên từ bài toán liên tục, hoặc (a) tỷ lệ biến $\bar{x}_s$ phải được giảm về 0 (mà không có bất kỳ sự thay đổi nào của các biến khác) hoặc (b) có ít nhất một trong các biến khác, ký hiệu là $\bar{x}_j$ thay đổi giá trị của nó (từ 1 thành 0 hoặc từ 0 thành 1). Trong trường hợp (a), giá trị của $z(C(KP))$ giảm bằng $\bar{c}p_s/w_s$, trong trường hợp (b) thì giảm bằng ít nhất $|p_j^*|$. Do đó, ta có chặn Muller-Merbach như sau:

$$U_4 = \max\left(\sum_{j=1}^{s-1} \max\{\lfloor z(C(KP)) - |p^* *_j |\rfloor : j \in N \setminus \{s\}\}\right). \tag{3.53}$$

Ta có thể trực tiếp suy ra $U_4 \leq U_1$. Thay vào đó, không có sự thống trị nào tồn tại giữa U_4 và các chặn khác. Như ở Ví dụ 1, ta có $U_3 = U_2 < U_4$, nhưng không khó để tìm thấy ví dụ mà $U_4 < U_3 \leq U_2$.

Ý tưởng đằng sau các chặn U_2, U_3 và U_4 được nghiên cứu sâu hơn bởi Dudzinski và Walukiewicz. Họ đã thu được một chặn thống trị tất cả các chặn trên. Ta xem xét bất kỳ nghiệm khả thi $\hat{x}$ nào của bài toán KP mà ta có thể có được từ bài toán liên tục như sau: và định nghĩa $\hat{N} = \{j \in N \setminus \{s\} : \hat{x}_j = 0\}$ ($\hat{x}$ gần với

Algorithm 2: Thủ tục tìm nghiệm khả thi Dudzinski-Walukiewicz

```
for với mỗi k ∈ N \ {s} do
    x̂_k := x̄_k;
x̂_s := 0;
for với mỗi k mà x̂_k = 0 do
    if w_k ≤ c − Σ_{j=1}^{n} w_j x̂_j then
        x̂_k := 1
```

nghiệm nhận được bởi thủ tục tham lam nhất). Để ý rằng một nghiệm nguyên tối ưu có thể nhận được trong trường hợp (a) bởi thiết lập $x_s = 1$ hoặc trong trường hợp (b) bởi thiết lập $x_s = 0$ và $x_j = 1$ với ít nhất một $j \in \hat{N}$, ta có chặn

Dudzinski-Walukiewicz như sau:

$$\begin{aligned} U_5 = \max(\, & \min(\overline{U}^1, \max\{\lfloor z(C(KP)) - |p^* *_j| \rfloor : j = 1, \ldots, s-1\}), \\ & \min(\overline{U}^0, \max\{\lfloor z(C(KP)) + |p^* *_j| \rfloor : j \in \hat{N}\}), \\ & \sum_{j=1}^{n} p_j \hat{x}_j) \end{aligned} \tag{3.54}$$

trong đó $\overline{U}^0$ và $\overline{U}^1$ được cho bởi các Biểu thức (3.49) và (3.50). Độ phức tạp thời gian là $O(n)$.

Ví dụ 3. Tiếp tục xem xét ví dụ 1. Từ biểu thức (3.42), ta có:

$$(p_j^*) = (13, 80, 70, 30, 0, -15, -50, -9)$$

Do đó:

$$U_4 = \max(265, \max\{282, 215, 225, 265, 280, 245, 286\}) = 286$$

$$(\hat{x}_j) = (1, 1, 1, 1, 0, 1, 0, 0)$$

$$U_5 = \max(\min(285, \max\{282, 215, 225, 265\}), \min(280, \max\{245, 286\}), 280) = 282.$$

3.4.3. Chặn từ liệt kê riêng phần

Chặn U_3 có thể được xem như kết quả của việc ứng dụng chặn Dantzig tại hai nút lá (terminal nodes) của một câu quyết định tương ứng với bài toán KP và hai (descendent nodes) $N0$ và $N1$ tương ứng với phép bao và loại trừ phần từ s. Rõ ràng cực đại giữa các chặn trên tương ứng với tất cả các nút lá của một cây quyết định biểu diễn một chặn trên hợp lệ cho bài toán gốc ứng với nút gốc. Vì thế, nếu $\overline{U}^0$ và $\overline{U}^1$ là các chặn Dantzig tương ứng với các nút $N0$ và $N1$, thì U_3 biểu diễn một chặn trên hợp lệ cho bài toán KP.

Một chặn cải thiện $U6$ có thể thu được nhờ việc xem xét các cây quyết định có nhiều hơn hai nút lá được đề xuất bởi Martello và Toth (1988).

Để trình bày nội dung của chặn này, ta giả sử s đã được xác định, đặt r, t lần lượt là bất kỳ item nào mà thỏa mãn $1 < r \leq s$ và $s \leq t < n$. Ta thu được một nghiệm khả thi cho bài toán KP bằng cách thiết lặp $x_j = 1$ với $j < r$, $x_j = 0$ với $j > t$ và tìm kiếm nghiệm tối ưu cho bài toán KP con $KP(r, t)$ bằng cách định nghĩa các item $r, r+1, \ldots, t$ với lực lượng giảm $c(r) = c - \sum_{j=1}^{r-1} w_j$.

Giả sử rằng bài toán $KP(r, t)$ được giải thông qua một cây quyết định nhị phân cơ sở mà phát sinh các cặp của các nút quyết định bởi thiết lập lần lượt $x_j = 1$ và $x_j = 0$ với $j = r, r+1, \ldots, t,$; mỗi nút k (đã thu được, đặt cố định x_j) phát sinh ra một cặp con cháu (cố định bởi x_{j+1}) nếu và chỉ nếu $j < t$ và nghiệm tương ứng k tồn tại. Với mỗi nút k của cây kết quả, gọi $f(k)$ là item từ nút này được phát sinh (bởi thiết lặp $x_{f(k)} = 1 \vee 0$) và đặt $x_j^k (j = r, \ldots, f(k))$ là dãy các giá trị được gán đến các biến $x_r, \ldots, x_{f(k)}$ dọc theo đường đi trong cây từ nút gốc đến k. Tập hợp các nút lá của cây có thể được phân hoạch thành:

$$L_1 = \left\{ l : \sum_{j=r}^{f(l)} w_j x_j^l > c(r) \right\} \quad \text{lá không khả thi (infeasible leaves)},$$

$$L_2 = \left\{ l : f(l) = t \quad \wedge \quad \sum_{j=r}^{f(l)} w_j x_j^l \leq c(r) \right\} \quad \text{lá khả thi (feasible leaves)}.$$

Với mỗi $l \in L1 \cup L2$, gọi u_l là bất kỳ chặn trên nào của bài toán được định nghĩa bởi biểu thức (3.4), (3.5) và

$$\begin{cases} x_j = x_j^l \quad \text{nếu} \quad j \in \{r, \ldots, f(l)\}, \\ x_j = 0 \vee 1 \quad \text{nếu} \quad j \in N \setminus \{r, \ldots, f(l)\}, \end{cases} \tag{3.55}$$

Bởi vì tất cả các nút không phải lá là có thể tìm thấy bởi cây, một chặn trên hợp lệ cho bài toán KP được cho bởi Biểu thức sau:

$$U_6 = \max\{u_l : l \in L1 \cup L_2\}. \tag{3.56}$$

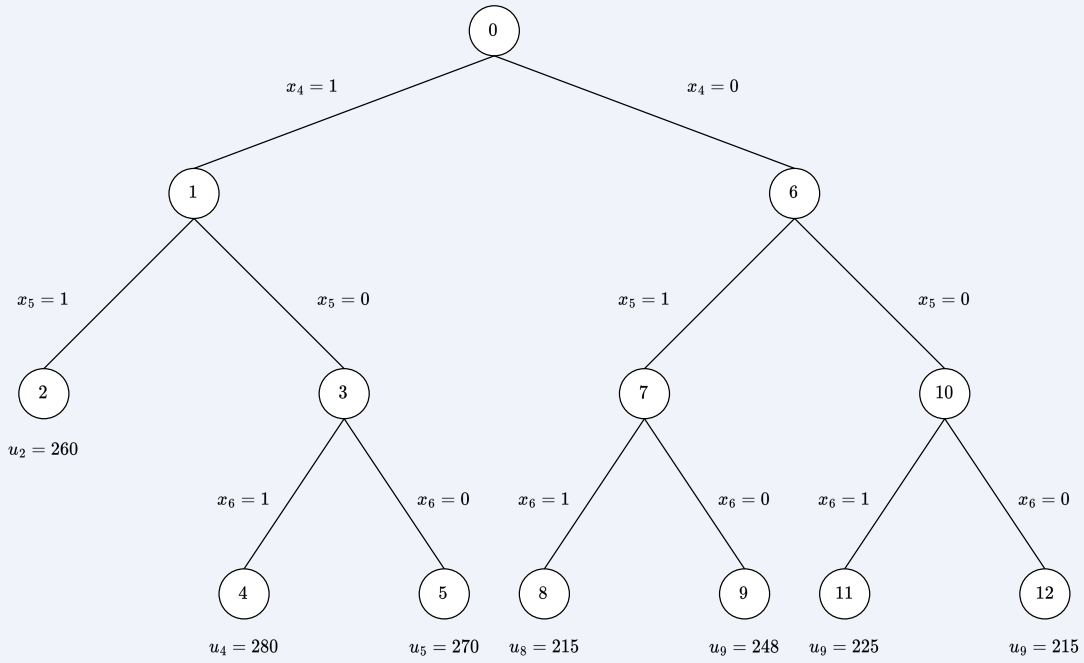
Một cách để tính nhanh u_l có thể được mô tả như sau. Gọi $p^l = \sum_{j=1}^{r-1} p_j + \sum_{j=r}^{f(l)} p_j x_j^l$ và $d^l = |c(r) - \sum_{j=r}^{f(l)} w_j x_j^l|$; thì

$$u_l = \begin{cases} \left\lfloor p_l - d^l \frac{p_{r-1}}{w_{r-1}} \right\rfloor & \text{nếu} \quad l \in L_1; \\ \left\lfloor p_l + d^l \frac{p_{t+1}}{w_{t+1}} \right\rfloor & \text{nếu} \quad l \in L_2; \end{cases} \tag{3.57}$$

rõ ràng là một chặn trên cho giá nghiệm liên tục của bài toán (3.4), (3.5) và (3.55).

Tính toán của U_6 cần $O(n)$ để quyết định phần tử chủ chốt và định nghĩa bài toán $KP(r,t)$, cộng với $O(2^{t-r})$ để duyệt . Nếu $t-r$ bị chặn bởi một hằng, thì tổng độ phức tạp thời gian là $O(n)$.

Ví dụ 4. Giả sử rằng $r = 4$ và $t = 6$. Sức chứa được giảm thiểu là $c(r) = 60$. Cây nhị phân được minh họa ở Hình 3.1. Tập các lá $L_1 = \{2, 8\}$, $L_2 = \{4, 5, 9, 11, 12\}$. Suy ra, $U_6 = 280$ là giá trị nghiệm tối ưu.



Hình 3.1: *Cây nhị phân của chặn U_6 cho Ví dụ 1*

Các chặn trên tại các nút lá có thể được đánh giá bằng cách sử dụng bất kỳ

chặn đã được mô tả phía trên thay vì sử dụng (3.57). Nếu $U_k(k = 1, \ldots, 5$ được sử dụng, thì rõ ràng $U_6 \leq U_k$; Nếu chặn ở biểu thức (3.57) được sử dụng thì không có thống trị tồn tại giữa U_6 và chặn Dudzinski-Walukiewicz, thế nên chặn trên tốt nhất cho bài toán KP là

$$U = \min(U_5, U_6).$$

Chặn U_6 có thể ngặt với một lượng nhỏ công sức tính toán bởi đánh giá $w_m = \min\{w_j : j > t\}$. Không quá khó để thấy rằng khi $l \in L_2$ và $d^l < w_m$, u_l có thể được tính toán bởi công thức như sau:

$$u_l = \max\left(p^l, \left\lfloor p^l + w_m \frac{p_{t+1}}{w_{t+1}} - (w_m - d^l)\frac{p_{r-1}}{w_{r-1}} \right\rfloor\right). \tag{3.58}$$

Cuối cùng, ta để ý rằng tính toán của U_6 có thể được đẩy nhanh bằng cách sử dụng một thuật toán nhánh cận bất kỳ để giải quyết bài toán $KP(r, t)$. Tại bất kỳ vòng lặp nào của thuật toán, gọi $\bar{z}(r, t)$ là giá trị của nghiệm tốt nhất. Với các nút không phải lá bất kỳ k của cây quyết định, gọi $\bar{u}_k$ là một chặn trên của nghiệm tối ưu của bài toán con được định nghĩa bởi các item $r, \ldots, n$ với $c(r)$, tức là bài toán con thu được nhờ sử dụng thiết lặp $x_j = 1$ với $j = 1, \ldots, r-1$. $\bar{u}_k$ có thể được tính như là một chặn trên của giá trị nghiệm liên tục của bài toán, tức:

$$\bar{u}_k = \sum_{j=r}^{f(k)} p_j x_j^k + \sum_{j=f(k)+1}^{s(k)-1} p_j \tag{3.59}$$

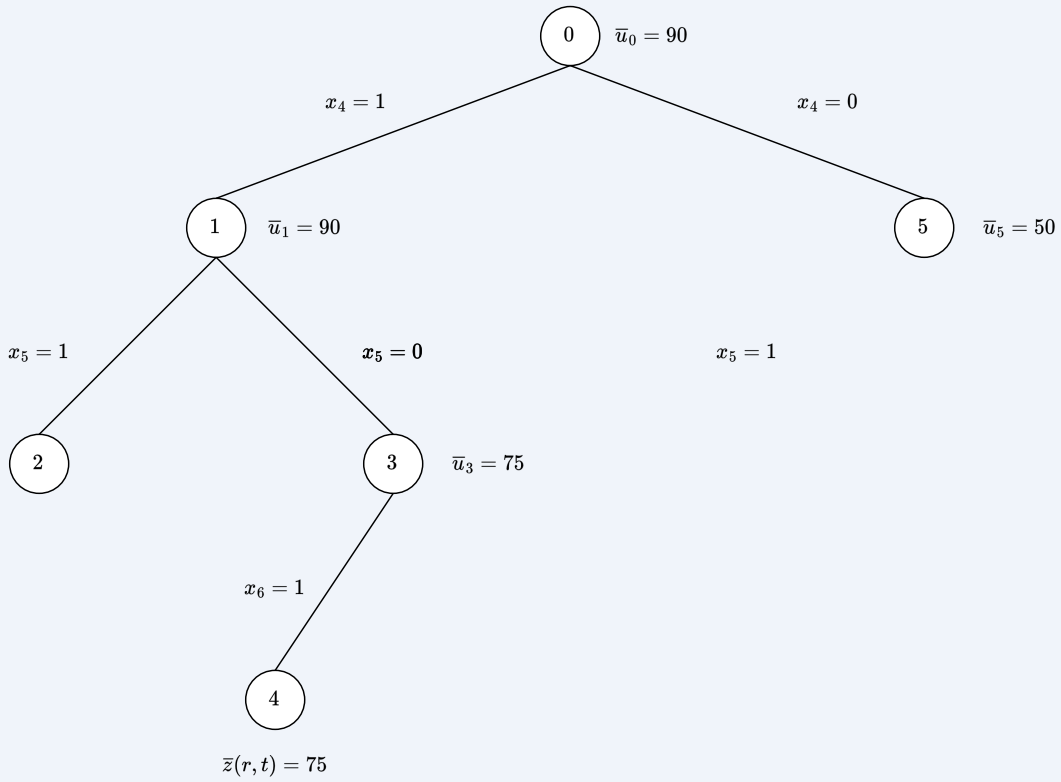
$$= \left\lfloor \left(c(r) - \left(\sum_{j=r}^{f(k)} w_j x_j^k + \sum_{j=f(k)+1}^{s(k)-1} w_j \right) \right) \frac{p_{s(k)}}{w_{s(k)}} \right\rfloor \tag{3.60}$$

trong đó $s(k) := \min(t+1, \min\{i : \sum_{j=r}^{f(k)} w_j x_j^k + \sum_{j=f(k)+1}^{i} w_j > c(r)\})$. Nếu ta có đánh giá $\bar{u}_k \leq \bar{z}(r, t)$, các đỉnh con cháu từ k không được phát sinh. Thật vậy,

với bất kỳ nút lá l nào là con cháu từ k, nó sẽ có kết quả là

$$u_l \leq \sum_{j=1}^{r-1} p_j + \bar{u}_k \leq \sum_{j=1}^{r-1} p_j + z(KP(r,t)) \leq U_6$$

Ví dụ 5. Thực hiện tính toán thông qua Biểu thức 3.59, ta thu được cây quyết định lược giản như Hình sau:



Hình 3.2: *Cây nhị nhánh cận của chặn U_6 cho Ví dụ 1*

CHƯƠNG 4

THUẬT TOÁN CHO BÀI TOÁN 0-1 KNAPSACK

4.1. Thuật toán tham lam

Một trong những cách trực tiếp để xác định một nghiệm xấp xỉ của bài toán KP là tìm kiếm nghiệm $\bar{x}$ của nới lỏng liên tục của bài toán mà chỉ có một biến phân số, $\bar{x}_s$ (Xem Định lý 1). Ta thiết lập biến này bằng 0 để lấy một nghiệm khả thi của bài toán KP có giá trị là:

$$z' = \sum_{j=1}^{s-1} p_j.$$

Ta có thể kỳ vọng z' này tương đối gần với giá trị nghiệm tối ưu z. Thật vậy, $z' \leq z \leq U_1 \leq z' + p_s$, tức là độ lỗi tuyệt đối bị chặn bởi một số p_s. Nhưng, tỷ lệ hiệu suất trường hợp xấu nhất hiển nhiên tệ. Điều này có thể được chứng minh bởi một chuỗi các bài toán với $n = 2, p_1 = w_1 = 1, p_2 = w_2 = k$ và $c = k$ mà trong đó $z' = 1$ và $z = k$, vì thế tỷ lệ z'/z gần với 0 khi k đủ lớn.

Để ý rằng, trường hợp trên xuất hiện khi mà p_s đủ lớn, ta có thể thu được một heuristic cải thiện hơn cũng bằng cách xem xét nghiệm khả thi được cho bởi phần tử chủ chốt duy nhất và lấy ra nghiệm tốt nhất trong hai giá trị nghiệm, tức là:

$$z^h = \max(z', p_s). \tag{4.1}$$

Tỷ lệ trường hợp xấu nhất của heuristic mới này là $\frac{1}{2}$. Thật vậy, $z \leq z' + p_s$, thế nên từ Biểu thức (4.1), ta được $z \leq 2z^h$. Để thấy rằng tỷ lệ $\frac{1}{2}$ là chặt, ta xem xét chuỗi bài toán với $n = 3, p_1 = w_1 = 1, p_2 = w_2 = p_3 = w_3 = k$ và $c = 2k$, ta có: vì $z^h = k + 1$ và $z = 2k$, nên z^h/z nhận giá trị bất kỳ gần với $\frac{1}{2}$ khi k đủ lớn.

Tính toán của z^h có độ phức tạp thời gian là $O(n)$ khi mà phần tử chủ chốt đã biết. Nếu các item được sắp xếp như trong Biểu thức (3.15), một thuật toán hiệu quả hơn có thể được sử dụng cho chúng giảm dần tương ứng với các chỉ số và thêm vào một phần tử mới cho ba-lô nếu nó đầy. Để ý rằng $1, \ldots, s-1$ luôn luôn được thêm vào, vì thế mà giá trị nghiệm ít nhất là z'. Một phương pháp heuristic phổ biến cho bài toán KP thường được gọi là *thuật toán tham lam (Greedy algorithm)*. Một lần nữa, hiệu suất trường hợp xấu nhất có thể tệ, nhưng được cải thiện đến gần $\frac{1}{2}$ nếu ta xem xét nghiệm được cho bởi phần tử của lợi ích cực đại như trong cài đặt thủ tục phía sau đây. Giả sử rằng các item có thứ tự ứng với Biểu thức (3.15).

Algorithm 3: Thủ tục GREEDY

Input: $n, c, (p_j), (w_j)$
Output: $z^g, (x_j)$

```
1  c̄ := c;
2  z^g := 0
3  j* := 1;
4  for j := 1 to n do
5      if w_j > c̄ then
6          x_j := 0
7      else
8          x_j := 1;
9          c̄ := c̄ − w_j;
10         z^g := z^g + p_j;
11     if p_j > p_{j*} then
12         j* := j
13 if p_{j*} > z^g then
14     z^g := p_{j*};
15     for j := 1 to n do
16         x_j := 0;
17     x_{j*} := 1
```

Tỷ lệ hiệu suất xấu nhất là $\frac{1}{2}$ vì:

- $p_j^* \geq p_s$ nên $z^g \geq z^h$;
- Chuỗi các bài toán được thêm vào cho z^h chứng minh tính ngặt

Độ phức tạp thời gian là $O(n)$, cộng với $O(n \log n)$ với thủ tục sắp xếp khởi tạo.

Ví dụ 6. Giả sử chúng ta có 4 vật phẩm với trọng lượng và giá trị như sau:

Vật phẩm	Trọng lượng(w_j)	Giá trị(p_j)
1	2	100
2	4	120
3	3	40
4	5	50

Dung lượng của ba lô $c = 5$.

$$\text{Khởi tạo:} \quad \bar{c} = 5, z^g = 0, j^* = 1$$

Thực hiện thuật toán GREEDY

1 . Vật phẩm 1:

- $w_1 = 2$, $w_1 \leq \bar{c}$, chọn $x_1 = 1$, cập nhật $\bar{c} = 5-2 = 3$, $z^g = 0+100 = 100$.
- So sánh $p_1 = 100$ với $p_{j^*} = 100$, không thay đổi j^*.

2 . Vật phẩm 2:

- $w_2 = 4$, $w_2 \geq \bar{c}$, không chọn x_2.
- So sánh $p_2 = 100$ với $p_{j^*} = 120$, cập nhật $j^* = 2$.

3. Vật phẩm 3:

- $w_3 = 3$, $w_3 = \bar{c}$, chọn $x_3 = 1$, cập nhật $\bar{c} = 3 - 3 = 0$, $z^g = 100 + 40 = 140$.

- So sánh $p_2 = 100$ với $p_{j^*} = 40$, không cập nhật.

4. Vật phẩm 4:

- $w_4 = 5$, $w_4 > \bar{c}$, không chọn $x_4 = 0$.

Kiểm tra giá trị lớn nhất

- So sánh $p_{j^*} = 120$ với $z^g = 140$, không cần thay đổi.

Kết quả cuối cùng

$$z^g = 140, \quad (x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0)$$

Thuật toán tham lam đã chọn vật phẩm 1 và 3 với tổng giá trị là 140.

Khi bài toán knapsack nhị phân được phát biểu trong dạng cực tiểu thì nó có thể được giải quyết bằng cách sử dụng tham lam đến dạng cực đại tương ứng của nó, và ta cũng có thể thu được một nghiệm khả thi, nhưng hiệu suất trường hợp xấu nhất không được bảo toàn. Thật vậy, xem xét chuỗi bài toán tối tiểu với $n = 3, p_1 = w_1 = k, p_2 = w_2 = 1, p_3 = w_3 = k$ và $q = 1$, có giá trị nghiệm tối ưu bằng 1. Khi áp dụng thủ tục tham lam cho dạng cực đại (với $c = 2k$), ta có thể $z^g = k + 1$ và vì thế ta thu được một nghiệm heuristic tệ của giá trị k cho bài toán tối tiểu.

Ví dụ 7. Cho một ba lô có sức chứa $W = 50$ và ba vật phẩm có giá trị và trọng lượng như sau:

Vật phẩm	Giá trị(p)	Trọng lượng(w)	$\frac{p}{w}$ (Tỉ lệ giá trị/trọng lượng)
1	60	10	6.0
2	100	20	5.0
3	120	30	4.0

Mục tiêu là chọn các vật phẩm sao cho tổng giá trị là lớn nhất mà tổng trọng lượng không vượt quá $W = 50$.

Áp dụng thuật toán Greedy:

Thuật toán *greedy* chọn các vật phẩm theo thứ tự tỉ lệ giá trị/trọng lượng $\frac{p}{w}$ lớn nhất trước. Lần lượt chọn như sau:

- Vật phẩm 1 có $\frac{p}{w} = 6.0$, chọn vật phẩm 1 với trọng lượng 10, giá trị 60.
- Vật phẩm 2 có $\frac{p}{w} = 5.0$, chọn vật phẩm 2 với trọng lượng 20, giá trị 100.
- Vật phẩm 3 có $\frac{p}{w} = 4.0$, nếu chọn vật phẩm 3 (trọng lượng 30), tổng trọng lượng sẽ là $10 + 20 + 30 = 60$, vượt quá giới hạn 50, nên không thể chọn.

Tổng trọng lượng của các vật phẩm được chọn là $10 + 20 = 30$, và tổng giá trị là $60 + 100 = 160$.

Nghiệm tối ưu:

Nếu chọn vật phẩm 2 và vật phẩm 3, tổng trọng lượng sẽ là $20 + 30 = 50$, vừa bằng giới hạn của ba lô, và tổng giá trị là $100 + 120 = 220$.

Kết luận:

Thuật toán *greedy* đưa ra nghiệm với tổng giá trị là 160, trong khi nghiệm tối ưu thực tế có giá trị 220. Điều này chứng tỏ thuật toán *greedy* không luôn tìm được nghiệm tối ưu trong bài toán ba lô.

Từ kết quả của ví dụ trên, ta thấy rằng, dù áp dụng chiến lược nào, thuật toán Greedy cho bài toán Knapsack 0/1 vẫn có thể dẫn đến nghiệm không tối ưu, mặc dù bài toán CKP (Continuous Knapsack Problem) luôn đạt được nghiệm chính xác. Do đó, để tìm nghiệm tối ưu cho bài toán Knapsack 0/1, cần phải sử dụng các phương pháp khác. Tuy nhiên, khi xử lý bài toán với dữ liệu lớn và chỉ cần một nghiệm đủ tốt, việc áp dụng thuật toán Greedy là lựa chọn hợp lý nhờ độ phức tạp thấp hơn.

4.2. Thuật toán quy hoạch động

Một tiếp cận cơ bản khác để giải quyết bài toán này là quy hoạch động. Đây là kỹ thuật được phát triển đầu tiên mà cho phép giải chính xác bài toán này và mặc dù mức độ quan trọng của nó đã giảm đáng kể so với kỹ thuật nhánh cận, nhưng nó vẫn rất thú vị để tìm hiểu.

Cho trước một cặp số nguyên $m(1 \leq m \leq n)$ và $\hat{c}(0 \leq \hat{c} \leq c)$, xem xét một thể hiện con của bài toán KP bao gồm các đồ vật $1, \ldots, m$ và dung lượng ba-lô $\hat{c}$. Đặt $f_m(\hat{c})$ là giá trị nghiệm tối ưu của nó, tức là:

$$f_m(\hat{c}) = \max \left\{ \sum_{j=1}^{m} p_j x_j : \sum_{j=1}^{m} x_j x_j \leq \hat{c}, x_j = 0 \quad \text{or} \quad 1 \quad \text{với} \quad j = 1 \ldots, m \right\} \quad (4.2)$$

Hiển nhiên, ta có:

$$f_1(\hat{c}) = \begin{cases} 0, & \text{với} \quad \hat{c} = 0, \ldots, w_1 - 1; \\ p_1, & \text{với} \quad \hat{c} = w_1, \ldots, c. \end{cases} \quad (4.3)$$

Quy hoạch động bao gồm quá trình n giai đoạn (với m tăng dần từ 1 đến n) và tại mỗi giải đoạn $m > 1$, ta tính toán các giá trị $f_m(\hat{c})$ (với $\hat{c}$ tăng dần từ 0 đến c) bằng cách sử dụng kỹ thuật đệ quy cổ điển (như các công trình của Bellman - 1954, Dantzig - 1957):

$$f_m(\hat{c}) = \begin{cases} f_{m-1}(\hat{c}) & \text{với} \quad \hat{c} = 0, \ldots, w_1 - 1; \\ \max(f_{m-1}(\hat{c}), f_{m-1}(\hat{c} - w_m) + p_m) & \text{với} \quad \hat{c} = w_1, \ldots, c. \end{cases}$$

Ta gọi các nghiệm khả thi tương ứng với các giá trị $f_m(\hat{c})$ là các trạng thái (states). Nghiệm tối ưu của bài toán là trạng thái tương ứng với $f_n(c)$.

Vào năm 1980, Toth đã suy ra được từ Biểu thức đệ quy Bellman một thủ tục hiểu quả để mà tính toán trạng thái của một giai đoạn. Trước tiên, ta cần

định nghĩa được các giá trị sau đây trước khi thực hiện một giai đoạn m:

$$v = \min\left(\sum_{j=1}^{m-1} w_j, c\right); \tag{4.4}$$

$$b = 2^{m-1}; \tag{4.5}$$

$$P_{\hat{c}} = f_{m-1}(\hat{c}), \quad \text{với} \quad \hat{c} = 0, \ldots, v; \tag{4.6}$$

$$X_{\hat{c}} = \{x_{m-1}, x_{m-2}, \ldots, x_1\} \quad \text{với} \quad \hat{c} = 0, \ldots, v; \tag{4.7}$$

trong đó x_j định nghĩa giá trị của biến thứ j trong nghiệm tối ưu riêng phần tương ứng với $f_{m-1}(\hat{c})$, tức là

$$\hat{c} = \sum_{j=1}^{m-1} w_j x_j \quad \text{và} \quad f_{m-1}(\hat{c}) = \sum_{j=1}^{m-1} p_j x_j. \tag{4.8}$$

Từ góc độ tính toán, ta có thể tối ưu tính toán bằng cách mã hóa mỗi tập $X_{\hat{c}}$ thành một chuỗi bit,

Algorithm 4: Thủ tục REC1

Input: $v, b, (P_{\hat{c}}), (X_{\hat{c}}), (w_m), p_m$

Output: $v, b, (P_{\hat{c}}), (X_{\hat{c}})$

```
1  if v < c then
2      u := v;
3      v := min(v + w_m, c);
4      for ĉ := u + 1 to v do
5          P_ĉ := P_u;
6          X_ĉ := X_u;
7  for ĉ := v to w_m step −1 do
8      if P_ĉ < P_{ĉ−w_m} + p_m then
9          P_ĉ := P_{ĉ−w_m} + p_m;
10         X_ĉ := X_{ĉ−w_m} + b;
11 b := 2b;
```

Algorithm 5: Thủ tục DP1

Input: $n, c, (p_j), (w_j)$

Output: $z, (x_j)$

for $\hat{c} := 0$ *to* $w_1 - 1$ **do**
    $P_{\hat{c}} := 0$;
    $X_{\hat{c}} := 0$;
$v := w_1$;
$b := 2$;
$P_v := p_1$;
$X_v := 1$;
for $m := 2$ *to* n **do**
    Gọi thủ tục REC1
$z := P_c$;
Xác định (x_j) bằng cách decode X_C

Thủ tục REC1 có độ phức tạp $O(c)$ nên độ phức tạp của thủ tục DP1 là $O(nc)$. Độ phức tạp không gian là $O(nc)$. Do ta đã mã hóa $X_{\hat{c}}$ thành một chuỗi bit theo ngôn ngữ máy gồn d bit, nên không gian lưu trữ cần thiết là $(1+\lceil n/d \rceil)c$, trong đó $\lceil a \rceil$ là số nguyên nhỏ nhất mà không nhỏ hơn a.

4.2.1. Khử các trạng thái bị thống trị

Số lượng các trạng thái được xem xét tại mỗi giai đoạn trong thuật toán có thể được giảm đi đáng kể bằng cách khử các *trạng thái bị thống trị (dominated states)*. Cụ thể, trong những trạng thái $(P_{\hat{c}}, X_{\hat{c}})$ mà tồn tại một trạng thái (P_y, X_y) với $P_y \geq P_{\hat{c}}$ và $y < \hat{c}$ (bất kỳ nghiệm có thể thu được từ $(P_{\hat{c}}, X_{\hat{c}})$ thì có thể thu được từ (P_y, X_y)). Kỹ thật này được sử dụng bởi Horowitz và Sahni (1974) và Ahrens và Finke (1975). Những trạng thái không bị thống trị (undominated states) của giai đoạn thứ m có thể được tính toán thông qua một thủ tục được

đề xuất bởi Toth (1980). Trước khi bắt đầu thực thi thủ tục cho giai đoạn m, ta định nghĩa các biến giá trị như sau:

$$s = \text{số lượng trạng thái ở giai đoạn } (m-1); \tag{4.9}$$

$$b = 2^{m-1}; \tag{4.10}$$

$$W1_i = \text{tổng trọng số của trạng thái thứ } i (i = 1, \ldots, s); \tag{4.11}$$

$$P1_i = \text{tổng lợi ích của trạng thái thứ } i (i = 1, \ldots, s); \tag{4.12}$$

$$X1_i = \{x_{m-1}, x_{m-2}, \ldots, x_1\}, \quad i = 1, \ldots, s, \tag{4.13}$$

trong đó x_j định nghĩa giá trị của biến thứ j trong nghiệm tối ưu riêng phần của trạng thái thứ i, tức là:

$$W1_i = \sum_{j=1}^{m-1} w_j x_j \quad \text{và} \quad P1_j = \sum_{j=1}^{m-1} p_j x_j.$$

Vector $W1$ (và do đó, $P1$) được giả sử rằng đã được sắp thứ tự tăng dần nghiêm ngặt.

Thủ tục sử dụng chỉ số i để quet1 các trạng thái của giai đoạn hiện tại và chỉ số k để lưu trữ các trạng thái của giai đoạn mới. Mỗi trạng thái hiện tại có thể tạo ra một trạng thái mới của tổng trọng số $y = W1_i + w_m$, nên các trạng thái hiện tại của tổng trọng số $W1_h < y$, và sau đó trạng thái mới được lưu trong giai đoạn mới chỉ khi nào chúng không bị thống trị bởi một trạng thái được lưu trữ. Sau khi thực thi, các giá trị của Biểu thức (4.9) và (4.10) được liên quan đến giai đoạn mới, còn các giá trị mới của Biểu thức (4.11), (4.12), (4.13) lần lượt được cho bởi $(W2_k), (P2_k)$ và $X2_k$. Các tập hợp $X1_i$ và $X2_k$ được mã hóa thành các chuỗi bit. Các vector $(W2_k)$ và $(P2_k)$ được sắp thứ tự tăng dần nghiêm ngặt. Đầu vào của thủ tục được giả định rằng $W1_0 = P1_0 = X1_0 = 0$.

Algorithm 6: Thủ tục REC2

Input: $s, b, (W1_i), (P1_i), (X1_i), w_m, p_m, c$

Output: $s, b, (W2_k), (P2_k), (X2_k)$

```
1  i := 0
2  k := 0
3  h := 1;
4  y := w_m
5  W1_{s+1} := +∞
6  W2_0 := 0
7  P2_0 := 0
8  X2_0 := 0
9  while min(y, W1_h) ≤ c do
10     if W1_h ≤ y then
           /* Định nghĩa trạng thái kế tiếp (p, x)                    */
11         p := P1_h; x := X1_h
12         if W1_h = y then
13             if P1_i + p_m > p then
14                 p := P1_i + p_m
15                 x := X1_i + b
16             i := i + 1
17             y := W1_i + w_m
           /* Lưu giữ trạng thái kế tiếp nếu không bị thống trị       */
18         if p > P2_k then
19             k := k + 1
20             W2_k := W1_h
21             P2_k := p
22             X2_k := x
23         h := h + 1
24     else
           /* Lưu giữ trạng thái mới nếu không bị thống trị           */
25         if P1_i + p_m > P2_k then
26             k := k + 1
27             W2_k := y
28             P2_k := P1_i + p_m
29             X2_k := X1_i + b
30         i := i + 1
31         y := W1_i + w_m
32     s := k; b := 2b
```

Một thuật toán quy hoạch động sử dụng thủ tục REC2 để giải bài toán KP như sau:

Algorithm 7: Thủ tục DP2

Input: $n, c, (p_j), (w_j)$

Output: $z, (x_j)$

$W1_0 := 0$

$P1_0 := 0$

$X1_0 := 0$

$s := 1$

$b := 2$

$W1_1 := w_1$

$P1_1 := p_1$

$X1_1 := 1$

for $m := 2$ *to* n **do**

    Gọi thủ tục REC2

    Đổi tên lần lượt $W2, P2$ và $X2$ thành $W1, P1$ và $X1$

$z := P1_s$

Xác định (x_j) bằng cách decode $X1_s$

Độ phức tạp thời gian của REC2 là $O(s)$ do s bị chặn bởi $\min(2^m - 1, c)$. Do đó, độ phức tạp thời gian của thuật toán DP2 là $O(min(2^{n+1}, nc))$.

Thủ tục DP2 không cần thứ tự cụ thể của các item. Điều này thể hiện tính hiệu quả của nó, tuy nhiên nếu chúng được sắp thứ tự giảm dần theo các tỷ lệ p_j/w_j thì trong trường hợp này, số lượng các trạng thái không bị thống trị bị giảm đi. Do đó, yêu cầu về thứ tự vẫn tiếp tục được giả định.

Ví dụ 8. Xem xét một thể hiện của KP được định nghĩa như sau:

$$n = 6;$$
$$(p_j) = (50, 50, 64, 46, 50, 5);$$
$$(w_j) = (56, 59, 80, 64, 75, 17);$$
$$c = 190.$$

Các bước thực hiện

Bước 1: Khởi tạo Tập trạng thái ban đầu là:

$$\texttt{states} = \{(0, 0)\}$$

Trong đó, mỗi phần tử là một cặp (value, weight), nghĩa là giá trị và trọng lượng đều bằng 0.

Bước 2: Xét từng vật phẩm

Vật phẩm 1: $p_1 = 50, w_1 = 56$

- **Trạng thái ban đầu:**

$$\texttt{states} = \{(0, 0)\}$$

- **Thêm vật phẩm 1:**

$$(0 + 50, 0 + 56) = (50, 56)$$

- **Trạng thái mới:**

$$\texttt{states} = \{(0, 0), (50, 56)\}$$

- **Khử trạng thái thống trị:** Không có trạng thái nào bị thống trị, giữ nguyên.

Vật phẩm 2: $p_2 = 50, w_2 = 59$

- **Trạng thái ban đầu:**

$$\texttt{states} = \{(0,0), (50,56)\}$$

- **Thêm vật phẩm 2:**

$$(0+50, 0+59) = (50,59)$$

$$(50+50, 56+59) = (100,115)$$

- **Trạng thái mới:**

$$\texttt{states} = \{(0,0), (50,56), (50,59), (100,115)\}$$

- **Khử trạng thái thống trị:** Loại bỏ trạng thái $(50,59)$ do bị thống trị bởi trạng thái $(50,56)$.

Vật phẩm 3: $p_3 = 64, w_3 = 80$

- **Trạng thái ban đầu:**

$$\texttt{states} = \{(0,0), (50,56), (100,115)\}$$

- **Thêm vật phẩm 3:**

$$(0+64, 0+80) = (64,80)$$

$$(50+64, 56+80) = (114,136)$$

$$(100+64, 115+80) = (164,195) \quad \text{(loại bỏ vì vượt quá sức chứa)}$$

- **Trạng thái mới:**

$$\texttt{states} = \{(0,0), (50,56), (64,80), (100,115), (114,136)\}$$

- **Khử trạng thái thống trị:** Không có trạng thái nào bị thống trị, giữ nguyên.

Vật phẩm 4: $p_4 = 46, w_4 = 64$

- **Trạng thái ban đầu:**

$$\texttt{states} = \{(0,0), (50,56), (64,80), (100,115), (114,136)\}$$

- **Thêm vật phẩm 4:**

$$(0+46, 0+64) = (46, 64)$$

$$(50+46, 56+64) = (96, 120)$$

$$(64+46, 80+64) = (110, 144)$$

$$(100+46, 115+64) = (146, 179)$$

$$(114+46, 136+64) = (160, 200) \quad \text{loại vì quá sức chứa}$$

- **Trạng thái mới:**

$$\begin{aligned}\texttt{states} = \{&(0,0), (50,56), (64,80),\\ &(100,115), (114,136), (46,64),\\ &(96,120), (110,144), (146,179)\}\end{aligned}$$

- **Khử trạng thái thống trị:** Loại bỏ $(46, 64)$ do bị thống trị bởi $(50, 56)$;

$(110, 144)$ do bị thống trị bởi $(114, 136)$; $(96, 120)$ do bị thống trị bởi $(100, 115)$

Vật phẩm 5: $p_5 = 50, w_5 = 75$

- **Trạng thái ban đầu:**

$$\texttt{states} = \{(0,0), (50,56), (64,80), (100,115), (114,136), (146,179)\}$$

- **Thêm vật phẩm 5:**

$$(0 + 50, 0 + 75) = (50, 75)$$

$$(50 + 50, 56 + 75) = (100, 131)$$

$$(64 + 50, 80 + 75) = (114, 155)$$

$$(100 + 50, 115 + 75) = (150, 190) \quad \text{(vừa đúng sức chứa)}$$

$$(114 + 50, 136 + 75) = (164, 211) \quad \text{loại vì quá sức chứa}$$

$$(146 + 50, 179 + 75) = (196, 254) \quad \text{loại vì quá sức chứa}$$

- **Trạng thái mới:**

$$\texttt{states} = \{(0,0), (50,56), (64,80), (100,115), (114,136), (146,179), (50,75), (100,131), (114,155), (150,190)\}$$

- **Khử trạng thái thống trị:** Loại bỏ $(50, 75)$, $(50, 56)$, và $(100, 131)$ do

bị thống trị bởi $(100, 115)$, và $(114, 155)$ do bị thống trị bởi $(114, 136)$.

Vật phẩm 6: $p_6 = 5, w_6 = 17$

- **Trạng thái ban đầu:**

$$\texttt{states} = \{(0,0), (50,56), (64,80), (100,115), (114,136), (146,179), (150,190)\}$$

- **Thêm vật phẩm 6:**

$$(0+5, 0+17) = (5, 17)$$

$$(50+5, 56+17) = (55, 73)$$

$$(64+5, 80+17) = (69, 97)$$

$$(100+5, 115+17) = (105, 132)$$

$$(114+5, 136+17) = (119, 143)$$

$$(146+5, 179+17) = (151, 196) \quad \text{loại vì vượt sức chứa}$$

$$(150+5, 190+17) = (155, 207) \quad \text{loại vì vượt sức chứa}$$

- **Trạng thái mới:**

$$\begin{aligned}\texttt{states} = \{&(0,0), (50,56), (64,80),\\ &(100,115), (114,136), (146,179),\\ &(150,190), (5,17), (55,73),\\ &(69,97), (105,132), (119,143)\}\end{aligned}$$

- **Khử trạng thái thống trị:** Không có trạng thái nào bị thống trị.

Bước 3: Kết quả Giá trị lớn nhất có thể đạt được là: 150, tương ứng với nghiệm $x = (1, 1, 0, 0, 1, 0)$.

4.2.2. Thuật toán Horowitz-Sahni

Horowitz và Sahni (1974) đã trình bày một thuật toán dựa trên sự phân chia nhỏ thứ phân (subdivision) của bài toán gốc gồm n biến thành hai bài toán con lần lượt là $q = \lceil n/2 \rceil$ và $r = n - q$. Với mỗi bài toán con, một danh sách chứa tất cả các trạng thái không bị thống trị liên quan đến trạng thái cuối cùng được tính toán; sau đó hai danh sách được kết hợp để mà tìm kiếm một nghiệm tối ưu.

Đặc trưng cốt lõi của thuật toán trong trường hợp xấu nhất là chỉ cần hai danh sách $2^q - 1$ cho mỗi trạng thái thay vì một danh sách đơn gồm $2^n - 1$ trạng thái. Do đó mà độ phức tạp thời gian và không gian giảm xuống $O(min(2^{n/2}, nc))$ với sự cải thiện căn bậc hai trong hầu hết các trường hợp. Trong nhiều trường hợp, số lượng các trạng thái bị thống trị nhỏ hơn nhiều so với $2^{n/2}$ bởi vì

(a) Nhiều trạng thái bị thống trị

(b) Và với n đủ lớn, ta có $c \ll 2^{n/2}$.

Ahrens và Finke (1975) đã đề xuất một thuật toán mà kết hợp kỹ thuật của Horowitz và Sahni (1974) với một thủ tục nhánh cận để tiết kiệm chi phí lưu trữ hơn nữa. Thuật toán hoạt động rất tốt với các bài toán khó mà có n giá trị thấp và các w_i và c có giá trị rất cao. Tuy nhiên, nhược điểm về độ phức tạp tính toán luôn luôn tồn tại trong các thủ tục nhánh cận cho dù chi phí lưu trữ không đáng kể đi nữa.

Ví dụ 9. Vẫn xét đầu vào như sau:

$$n = 6;$$
$$(p_j) = (50, 50, 64, 46, 50, 5);$$
$$(w_j) = (56, 59, 80, 64, 75, 17);$$
$$c = 190.$$

Ở đây, ta tính được $q = 3$. Do đó ta có 2 tập con như sau:

$$\{(p_1, w_1), (p_2, w_2), (p_3, w_3)\} = \{(50, 56), (50, 59), (64, 80)\}$$

và

$$\{(p_4, w_4), (p_5, w_5), (p_6, w_6)\} = \{(46, 64), (50, 75), (5, 17)\}$$

Áp dụng thuật toán DP2 cho tập đầu

Các bước trình bày chi tiết, xem lại ví dụ 8, ta thu được các trạng thái cuối cùng sẽ là:

$$\textbf{states} = \{(0, 0), (50, 56), (64, 80), (100, 115), (114, 136)\}$$

Áp dụng thuật toán DP2 cho tập cuối Các bước trình bày chi tiết, xem lại ví dụ 8, ta thu được các trạng thái cuối cùng sẽ là:

$$\textbf{states} = \{(0, 0), (5, 17), (46, 64), (50, 75), (51, 81), (55, 92), (96, 139), (101, 156)\}$$

Kết hợp hai trạng thái Ta thu được tổ hợp như sau:

$$\begin{aligned}\textbf{states} = \{&(0,0),(5,17),(46,64),(50,75),\\&(51,81),(55,92),(96,139),(101,156),\\&(50,56),(55,73),(96,120),(100,131),\\&(101,137),(105,148),(64,80),(69,97),\\&(110,144),(114,155),(115,161),(119,172),\\&(100,115),(105,132),(146,179),(150,190),\\&(114,136),(119,153)\}\end{aligned}$$

Chọn kết quả tốt nhất trong các states là: $(150, 190)$.

4.2.3. Thuật toán Toth

Vào năm 1980, Toth đã trình bày một thuật toán quy hoạch động dựa trên

(a) Sự khử của các trạng thái không cần thiết

(b) Và một phép kết hợp của các thủ tục REC1 và REC2.

Nhiều trạng thái được tính toán bởi các thủ tục REC1 hay REC2 không thật sự cần thiết cho các giai đoạn phía sau đó bởi vì ta chỉ cần quan tâm đến các trạng thái có khả năng sinh ra nghiệm tối ưu trong giai đoạn cuối cùng. Các trạng thái không cần thiết được sinh ra bởi thủ tục REC1 có thể khử đi nhờ một luật như sau:

Nếu một trạng thái được định nghĩa bởi giai đoạn thứ m có một tổng trọng số W thỏa mãn một trong các điều kiện sau:

$$(i) \quad W < c - \sum_{j=m+1}^{n} w_j,$$

$$(ii) \quad c - \min_{m<j\leq n}\{w_j\} < W < c,$$

thì trạng thái sẽ không bao giờ sinh ra được P_c và do đó có thể được khử đi.

Một luật tương tự có thể được đưa ra cho thủ tục REC2. Trong trường này, nó cần giữ lại các trạng thái với trọng số lớn nất thỏa mãn (i), và trạng thái cuối cùng tức là trạng thái s. Luật không thể được mở rộng cho thuật toán Horowitz-Sahni bởi vì để kết hợp hai danh sách, tất cả các trạng thái không bị thống trị liên quan đến hai bài toán con phải được biết.

Ví dụ 10. Các trạng thái được sinhra bởi thủ tục DP2 với REC2 được cải thiện thông qua luật khử như sau:

i	$m=1$		$m=2$		$m=3$		$m=4$		$m=5$		$m=6$	
	W_i	P_i	W_i	P_i	W_i	P_i	W_i	P_i	W_i	P_i	W_i	P_i
0	0	0	0	0	0	0	0	0	0	0	0	0
1	56	50	56	50	56	50	80	64	136	114	190	150
2			115	100	80	64	115	100	190	150		
3					115	100	136	114				
4					136	114	179	146				

Nhìn chung, thuật toán DP2 có tính hiệu quả hơn thuật toán DP1 bởi vì số trạng thái được sinh ra ít hơn. Để ý rằng, với việc tính toán của một trạng thái đơn, độ phức tạp thời gian và không gian cần của thuật toán DP2 cao hơn so với DP1. Thế nên, với các bài toán khó mà trong đó có rất ít các trạng thái bị thống trị và cả hai thuật toán đều sinh ra các danh sách hầu như giống nhau, thì thuật toán DP1 được cân nhắc lựa chọn hơn so với DP2. Một thuật toán quy hoạch động mà giải quyết được cả hai thể loại bài toán dù dễ hay khó có thể tạo ra được nhờ kết hợp các đặc trưng tốt của hai hướng tiếp cận. Bằng cách sử dụng thủ tục REC2 miễn là số lượng các trạng thái được sinh ra là thấp và sau đó cho vào thủ tục REC1.

4.3. Nhận xét

Tóm lại, cả thuật toán knapsack tham lam và knapsack 0/1 đều có sự đánh đổi khác nhau giữa tính tối ưu và hiệu quả. Các giải pháp nhanh có thể đến từ knapsack tham lam nhưng các giải pháp như vậy không tối ưu trong một số trường hợp trong khi knapsack 0/1 đảm bảo điều đó với cái giá phải trả là độ phức tạp tính toán cao. Sự hiểu biết này sẽ hình thành cơ sở để lựa chọn thuật toán phù hợp cho một bài toán knapsack nhất định. Sau đây là sự so sánh giữa hai thuật toán này:

Tiêu chuẩn	**Greedy**	**Quy hoạch động**
Hướng tiếp cận	Chiến lược tham lam, lựa chọn tối ưu cục bộ	Lập trình động, xem xét tất cả các tùy chọn
Cơ sở	Dựa trên tỷ lệ giá trị trên trọng lượng	Xem xét tất cả các kết hợp có thể
Độ phức tạp	O(n log n) đã sắp xếp	O(nW) – Trong đó n là số lượng mục, W là dung lượng
Lời giải tối ưu	Không phải lúc nào cũng tối ưu	Luôn luôn tối ưu
Đối tượng **Sử dụng bộ nhớ**	Phù hợp cho bài toán CKP Yêu cầu ít bộ nhớ hơn	Phù hợp bài toán 0/1 KP Yêu cầu nhiều bộ nhớ hơn do bảng DP
Loại thuật toán	Tham lam	Lập trình động
Sắp xếp	Yêu cầu sắp xếp dựa trên các tiêu chí nhất định	Không cần sắp xếp
Tốc độ	Nhanh hơn do lựa chọn tham lam	Chậm hơn do tìm kiếm đầy đủ
Trường hợp sử dụng	Xấp xỉ nhanh, tập dữ liệu lớn	Dữ liệu nhỏ, đảm bảo tính tối ưu

Bảng 4.1: *So sánh giữa thuật toán tham lam và Quy hoạch động*

CHƯƠNG 5

LẬP TRÌNH VÀ THỰC NGHIỆM

5.1. Cài đặt

Trong phần này, chúng ta sẽ trình bày môi trường phát triển và các công cụ được sử dụng trong quá trình lập trình và thực nghiệm. Python được chọn làm ngôn ngữ lập trình chính nhờ vào sự đơn giản, tính linh hoạt và khả năng hỗ trợ nhiều thư viện mạnh mẽ cho tính toán và thực nghiệm. Sau đây là các yêu cầu để cài đặt như sau:

5.1.1. Windows

1. Truy cập trang web chính thức của Python tại https://www.python.org/downloads/.
2. Tải xuống phiên bản Python 3.x mới nhất.
3. Khi cài đặt, đánh dấu tùy chọn **Add Python to PATH** để tự động thêm Python vào biến môi trường PATH của hệ thống.
4. Hoàn tất cài đặt bằng cách làm theo các hướng dẫn trên màn hình.
5. Sau khi cài đặt, mở (CMD) và kiểm tra bằng cách gõ lệnh:

```
python --version
```

Lệnh này sẽ trả về phiên bản Python đã cài đặt nếu quá trình cài đặt thành công.

5.1.2. MacOS

1. Mở Terminal và cài đặt Homebrew (nếu chưa cài đặt) bằng lệnh:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebr
```

2. Sau khi cài đặt Homebrew, cài đặt Python bằng lệnh:

```
brew install python
```

3. Kiểm tra phiên bản Python đã cài đặt bằng lệnh:

```
python3 --version
```

5.1.3. Linux (Ubuntu)

1. Mở và cập nhật danh sách gói bằng lệnh:

```
sudo apt update
```

2. Cài đặt Python 3 bằng lệnh:

```
sudo apt install python3
```

3. Kiểm tra phiên bản Python đã cài đặt:

```
python3 --version
```

5.1.4. Cài đặt công cụ quản lý gói (pip)

`pip` là công cụ quản lý gói của Python, dùng để cài đặt các thư viện và gói mở rộng. Sau khi cài đặt Python, bạn có thể kiểm tra phiên bản `pip` bằng lệnh:

```
pip --version
```

Nếu chưa cài đặt, bạn có thể cài đặt `pip` theo các bước sau:

Cài đặt ***`pip`*** *trên Windows/MacOS/Linux*

```
python -m ensurepip --upgrade
```

5.1.5. Cài đặt các gói thư viện cần thiết

Các gói thư viện cần thiết sẽ được lưu trữ trong tệp requirements.txt kèm theo với mã nguồn. Để cài đặt tất cả các thư viện từ file `requirements.txt`, bạn sử dụng lệnh:

```
pip install -r requirements.txt
```

5.2. Thử nghiệm và đánh giá

5.2.1. Thử nghiệm với các ví dụ nhỏ

```
402
403
404   knapsack = Knapsack(
405       capacity=190,
406       item_weights=(56, 59, 80, 64, 75, 17),
407       item_values=(50, 50, 64, 46, 50, 5),
408   )
409   # ([1, 1, 0, 0, 1, 0], 150, 190)
410   # ([1, 1, 0, 0, 1, 0], 150, 190)
411   # ([1, 1, 0, 0, 1, 0], 150, 190)
412   # ([1, 1, 0, 0, 1, 0], 150, 190)
413
414   items = knapsack.sort()
415   print(knapsack.dynamic_programming_vanilla(knapsack.items))
416   print(knapsack.dynamic_programming_Toth(knapsack.items))
417   print(knapsack.dynamic_programming_elimination_dominated(knapsack.items))
418   print(knapsack.dynamic_programming_Horowitz_Sahni(knapsack.items))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
┌─[lnhutnam@lnnam-arch] - [~/Workspace/Knapsack] - [2024-10-02 11:13:17]
└─[0] <> python knapsack_v2.py
([1, 1, 0, 0, 1, 0], 150, 190)
([1, 1, 0, 0, 1, 0], 150, 190)
([1, 1, 0, 0, 1, 0], 150, 190)
([1, 1, 0, 0, 1, 0], 150, 190)
┌─[lnhutnam@lnnam-arch] - [~/Workspace/Knapsack] - [2024-10-02 11:14:22]
```

Hình 5.1: *Thí nghiệm Ví dụ 1.*

```
403
404  knapsack = Knapsack(
405      capacity=102,
406      item_values=(15, 100, 90, 60, 40, 15, 10, 1),
407      item_weights=(2, 20, 20, 30, 40, 30, 60, 11),
408  )
409  # ([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
410  # ([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
411  # ([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
412  # ([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
413
414  items = knapsack.sort()
415  print(knapsack.dynamic_programming_vanilla(knapsack.items))
416  print(knapsack.dynamic_programming_Toth(knapsack.items))
417  print(knapsack.dynamic_programming_elimination_dominated(knapsack.items))
418  print(knapsack.dynamic_programming_Horowitz_Sahni(knapsack.items))
419
420
421  # knapsack = Knapsack(
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[lnhutnam@lnnam-arch] - [~/Workspace/Knapsack] - [2024-10-02 11:17:50]
[0] <> python knapsack_v2.py
([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
([1, 1, 1, 1, 0, 1, 0, 0], 280, 102)
[lnhutnam@lnnam-arch] - [~/Workspace/Knapsack] - [2024-10-02 11:17:51]
```

Hình 5.2: *Thí nghiệm Ví dụ 2.*

5.2.2. Đánh giá thời gian và độ phức tạp bộ nhớ thực thi

Do quá trình thực thi các thuật toán "dynamic_programming_Horowitz_Sahni" và "dynamic_programming_elimination_dominated". tốn rất nhiều tài nguyên tính toán, trong khi tài nguyên không đáp ứng đủ. Vì vậy, trong phần đánh giá này, chúng tôi sẽ không đánh giá cho kết quả hai thuật toán trên.

Data đánh giá bao gồm 3 bộ dữ liệu là **00Uncorrelated**, **02StronglyCorrelated** và **12Circle**. Mỗi bộ gồm 4 tập tin chứa thông tin của bài toán Knapsack với số lượng đồ vật lần lượt là 50 và 100; dung lượng túi tăng dần.

Phức tạp thời gian

Thuật toán	Thời gian (ms)
greedy_binary	0.06644
greedy	0.05246
dynamic_programming_vanilla	7956.2574
dynamic_programming_Toth	1407.7398

Bảng 5.1: *Thời gian chạy của các thuật toán*

Phức tạp bộ nhớ

Thuật toán	RAM (MB)
greedy_binary	0.00016
greedy	0.00130
dynamic_programming_vanilla	4.7073
dynamic_programming_Toth	1.0294

Bảng 5.2: *RAM sử dụng của các thuật toán*

Nhận xét: Với họ thuật toán Greedy sẽ cho kết quả thời gian và sử dụng RAM tốt hơn so với hai thuật toán còn lại, tuy nhiên như đã trình bày, hai thuật toán này không cho kết quả tối ưu, chúng tôi đưa vào đây để lấy làm căn cứ so sánh. Trong khi đó, thuật toán quy hoạch động Toth cho kết quả tốt trên cả thời gian và độ phức tạp bộ nhớ (tương ứng là 6x và 3x). Điều này phù hợp với lý thuyết chúng tôi đã trình bày ở chương trước.

5.2.3. Hướng dẫn chạy đánh giá

1. Chạy đánh giá trên tập dữ liệu

```
python3 knapsack.py --data [input-data]
                    --output [output-save] --mode eval
```

Ví dụ:

```
python3 knapsack.py --data kplib --output output --mode eval
```

2. Chạy test kết quả:

```
python3 knapsack.py --mode infer
```

- Màn hình hỏi: **input capacity:**, nhập dung lượng túi, ví dụ: 10

- Màn hình hỏi: **input weights, separated by ‘,‘:**, nhập khối lượng từng vật phân cách bởi dấu “,“, ví dụ: 5,7,3

- Màn hình hỏi: **input values, separated by ‘,‘:**, nhập giá trị từng vật phân cách bởi dấu “,“, ví dụ: 5,15,6.

- Kết quả:

```
input capacity:
10
input weights, separated by `,`:
5,7,3
input values,  separated by `,`:
5,15,6
greedy:
solution: (0, 1, 1)
value: 21
greedy_binary:
solution: (0, 1, 1)
value: 21
dynamic_programming_vanilla:
solution: [0, 1, 1]
value: 21
```

```
dynamic_programming_Toth:
solution: [0, 1, 1]
value: 21
dynamic_programming_Horowitz_Sahni:
solution: [0, 1, 1]
value: 21
dynamic_programming_elimination_dominated:
solution: [0, 1, 1]
value: 21
```

CHƯƠNG 6

KẾT LUẬN

Thuật toán Knapsack là một trong những bài toán tối ưu hóa cổ điển trong khoa học máy tính và toán học. Thông qua quá trình tìm hiểu và thực hiện lại thuật toán, chúng ta có thể rút ra một số kết luận quan trọng sau:

- **Sự phức tạp của bài toán**: Bài toán Knapsack tồn tại dưới nhiều biến thể khác nhau như bài toán Knapsack 0/1, bài toán Knapsack phân số (Fractional Knapsack), và bài toán Knapsack nhiều chiều (Multi-dimensional Knapsack). Mỗi biến thể đều có độ phức tạp riêng, trong đó bài toán Knapsack 0/1 là dạng phổ biến nhất nhưng có độ phức tạp cao, đặc biệt là khi số lượng vật phẩm lớn.

- **Các phương pháp giải quyết**: Thuật toán tham lam (Greedy) là phương pháp đơn giản và hiệu quả nhất cho bài toán Knapsack phân số, nhưng không áp dụng được cho bài toán Knapsack 0/1. Với bài toán Knapsack 0/1, các thuật toán quy hoạch động và quay lui (backtracking) thường được sử dụng để tìm ra lời giải tối ưu. Tuy nhiên, những thuật toán này có độ phức tạp thời gian cao, do đó cần tối ưu hóa hiệu suất khi triển khai trên thực tế.

- **Hiệu quả của các phương pháp**: Trong quá trình thực hiện lại thuật toán, chúng ta nhận thấy rằng thuật toán quy hoạch động có khả năng tìm ra kết quả chính xác với độ phức tạp thời gian $O(nc)$ (trong đó n là số lượng vật phẩm và c là dung lượng của túi), nhưng đòi hỏi bộ nhớ lớn. Thuật toán tham lam tuy nhanh và đơn giản nhưng chỉ phù hợp cho bài toán phân số, và không luôn cho ra kết quả tối ưu trong bài toán Knapsack 0/1.

- **Ứng dụng thực tiễn**: Thuật toán Knapsack có nhiều ứng dụng thực tế trong các lĩnh vực như tối ưu hóa tài nguyên, lập kế hoạch sản xuất, quản lý tài chính, và các hệ thống phân phối hàng hóa. Hiểu rõ về cách hoạt động và hạn chế của từng thuật toán giúp đưa ra những giải pháp phù hợp cho các bài toán thực tiễn.

- **Những cải tiến trong tương lai**: Việc nghiên cứu và áp dụng các thuật toán tiến hóa hoặc heuristic như thuật toán di truyền (genetic algorithm) hoặc thuật toán bầy đàn (swarm optimization) có thể là một hướng đi hứa hẹn để giải quyết bài toán Knapsack với các dữ liệu lớn, nơi mà các phương pháp truyền thống gặp phải hạn chế về thời gian và tài nguyên.

Tóm lại, việc tìm hiểu và thực hiện lại thuật toán Knapsack đã giúp chúng ta hiểu sâu hơn về bản chất của các bài toán tối ưu hóa, từ đó có thể ứng dụng vào nhiều lĩnh vực khác nhau. Tuy nhiên, để đạt được hiệu quả cao, việc lựa chọn phương pháp giải quyết phù hợp với từng tình huống cụ thể là vô cùng quan trọng.

PHỤ LỤC 1: MÃ NGUỒN PYTHON CHO THUẬT TOÁN KNAPSACK

.1. Mã nguồn:

Knapsack algorithmn

.2. Các lớp bổ trợ

```python
class Item:
    def __init__(self, weight, value, id):
        """
        Initialize the Item class.

        Parameters
        ----------
        weight : int
            The weight of the item.
        value : int
            The value of the item.
        id : int
            The id of the item.

        Returns
        -------
        None
        """
        self.weight = weight
        self.value = value
```

```
        self.ratio = value / weight
        self.id = id
```

```
    class Knapsack:
    def __init__(self, capacity, item_weights, item_values):
        """
        Initialize the Knapsack class.

        Parameters
        ----------
        capacity : int
            The maximum weight the knapsack can hold.
        item_weights : list of int
            The weights of each item.
        item_values : list of int
            The value of each item.

        Attributes
        ----------
        capacity : int
            The maximum weight the knapsack can hold.
        item_weights : list of int
            The weights of each item.
        item_values : list of int
            The value of each item.
        num_items : int
            The number of items.
        """
        self.capacity = capacity
        self.items = [
```

```
            Item(w, v, i) for i, (w, v) in
            ↪ enumerate(zip(item_weights, item_values))
        ]
        self.num_items = len(self.items)

    def sort(self):
        return sorted(self.items, key=lambda x: x.value,
        ↪ reverse=True)
```

.3. Cài đặt thuật toán tham lam

```
    def greedy(self, items: list[Item]):
        """
        Find the maximum value that can be stored in the knapsack
        ↪ using the greedy algorithm.

        Returns
        -------
        int
            The maximum value that can be stored in the knapsack.
        """
        result = [0] * self.num_items
        total_value = 0
        total_weight = 0
        items = self.sort(items)
        for i, item in enumerate(items):
            if item.weight + total_weight <= self.capacity:
                result[i] = 1
                total_weight += item.weight
                total_value += item.value
            else:
```

```
20                remain = self.capacity - total_weight
21                if remain == 0:
22                    continue
23                total_value = total_value + item.value * remain /
                  ↪  item.weight
24                result[i] = 1
25                break
26
27        # return solution based on original ids
28        result, _ = zip(*sorted(zip(result, items), key=lambda x:
          ↪  x[1].id))
29        return result, total_value
30
31    def greedy_binary(self, items: list[Item]):
32        """
33        Find the maximum value that can be stored in the knapsack
          ↪  using the greedy algorithm.
34
35        Returns
36        -------
37        int
38            The maximum value that can be stored in the knapsack.
39        """
40        # Sort the items by value/weight ratio in descending order
41        remain_weight = self.capacity
42        result = [0] * self.num_items
43        total_value = 0
44        best_idx = 0
45        items = self.sort(items)
46        for i, item in enumerate(items):
47            value = item.value
```

```
            weight = item.weight
            if weight <= remain_weight:
                result[i] = 1
                remain_weight -= weight
                total_value += value
                best_idx = i
            if items[best_idx].value > value:
                best_idx = i

        if items[best_idx].value > total_value:
            total_value = items[best_idx].value
            result = [1 if i == best_idx else 0 for i in
            ↪ range(self.num_items)]

        # return solution based on original ids
        result, _ = zip(*sorted(zip(result, items), key=lambda x:
        ↪ x[1].id))
        return result, total_value
```

.4. Cài đặt thuật toán quy hoạch động

Thủ tục quy hoạch động thuần

```
    def dynamic_programming_vanilla(self, items: list[Item]):
        """
        Find the maximum value that can be stored in the knapsack
        ↪ using dynamic programming.

        The time complexity is O(n*W) where n is the number of
        ↪ items and W is the capacity of the knapsack.

```

```
        Returns
        -------
        result : list
            A list of 0/1 indicating whether the item should be
            ↪ included or not.
        total_value : int
            The total value of the items.
        total_weight : int
            The total weight of the items.
        """
        dp_table = np.zeros((self.num_items + 1, self.capacity + 1))
        capacity = self.capacity
        for i in range(1, self.num_items + 1):
            for w in range(1, capacity + 1):
                if items[i - 1].weight > w:
                    dp_table[i, w] = dp_table[i - 1, w]
                else:
                    dp_table[i, w] = max(
                        dp_table[i - 1, w],
                        dp_table[i - 1, w - items[i - 1].weight] +
                        ↪ items[i - 1].value,
                    )

        # traceback chosen items
        chosen_items = [0] * self.num_items
        total_weight = 0
        for i in range(self.num_items, 0, -1):
            if dp_table[i, capacity] > dp_table[i - 1, capacity]:
                chosen_items[i - 1] = 1
                total_weight = total_weight + self.items[i -
                ↪ 1].weight
```

```
                capacity = capacity - self.items[i - 1].weight

        return chosen_items, int(max(dp_table[self.num_items]))
```

.4.1. Cài đặt thủ tục khử thống trị

```
    def dynamic_programming_elimination_dominated(self, items:
    ↪ list[Item]):
        def rec2(s, b, W1, P1, X1, w_m, p_m, c):
            i = 0
            k = 0
            h = 1
            y = w_m
            W1[s + 1] = float("inf")  # W1[s+1] = infinity

            # Khởi tạo W2, P2, X2
            W2 = [0] * (len(W1) + 1)
            P2 = [0] * (len(P1) + 1)
            X2 = [0] * (len(X1) + 1)

            while min(y, W1[h]) <= c:
                if W1[h] <= y:
                    # Định nghĩa trạng thái kế tiếp (p, x)
                    p = P1[h]
                    x = X1[h]

                    if W1[h] == y:
                        if P1[i] + p_m > p:
                            p = P1[i] + p_m
                            x = X1[i] + b
                        i = i + 1
```

```
                        y = W1[i] + w_m

                    # Lưu giữ trạng thái kế tiếp nếu không bị thống
                    ↪ trị
                    if p > P2[k]:
                        k = k + 1
                        W2[k] = W1[h]
                        P2[k] = p
                        X2[k] = x
                    h = h + 1
                else:
                    # Lưu giữ trạng thái mới nếu không bị thống
                    ↪ trị
                    if P1[i] + p_m > P2[k]:
                        k = k + 1
                        W2[k] = y
                        P2[k] = P1[i] + p_m
                        X2[k] = X1[i] + b
                    i = i + 1
                    y = W1[i] + w_m

            s = k
            b = 2 * b  # b := 2b

            return s, b, W2, P2, X2

        W1 = [0] * (self.num_items + 1)
        P1 = [0] * (self.num_items + 1)
        X1 = [0] * (self.num_items + 1)

        s = 1
```

```
        b = 2

        # init value for the first state
        W1[1] = items[0].weight
        P1[1] = items[0].value
        X1[1] = 1

        for m in range(2, self.num_items):
            s, b, W1, P1, X1 = rec2(
                s, b, W1, P1, X1, items[m - 1].weight, items[m -
                ↪  1].value, self.capacity
            )

        max_value = P1[s]  # z = P1_s
        x = [0] * self.num_items

        while b > 0 and X1[s] != 0:
            for i in range(self.num_items - 1, -1, -1):
                if X1[s] & (1 << i):
                    x[i] = 1
                    b -= 2

        return x, max_value
```

.4.2. Cài đặt thuật toán Horowitz-Sahni

```
    def dynamic_programming_Horowitz_Sahni(self, items: list[Item]):
        def generatesubset(items):
            n = len(items)
            subsets = []
            for i in range(n + 1):
```

```
            for comb in combinations(items, i):
                weight = sum(item.weight for item in comb)
                profit = sum(item.value for item in comb)
                subset_items = [item for item in comb]
                # subsets.append((weight, profit))
                subsets.append((weight, profit, subset_items))
        return subsets

    def horowitz_sahni_knapsack(items):
        q = math.ceil(self.num_items / 2)
        items1 = items[:q]
        items2 = items[q:]

        subsets1 = generatesubset(items1)
        subsets2 = generatesubset(items2)

        # Sort subsets by weight
        subsets2.sort(key=lambda x: x[0])

        # Filter the subsets2 to keep only the ones with the
        ↪ highest profit for each weight
        filtered_subsets2 = []
        max_profit = -1
        for weight, profit, subset_items in subsets2:
            if profit > max_profit:
                # filtered_subsets2.append((weight, profit))
                filtered_subsets2.append((weight, profit,
                ↪ subset_items))
                max_profit = profit

        # Initialize maximum profit
```

```
            max_profit = 0
            best_combination = []

            # For each subset in subsets1, find the best complement
            ↪ in subsets2
            for weight1, profit1, subset1_items in subsets1:
                remaining_capacity = self.capacity - weight1
                if remaining_capacity < 0:
                    continue

                # Binary search to find the best subset in subsets2
                ↪ that fits within the remaining capacity
                lo, hi = 0, len(filtered_subsets2) - 1
                while lo <= hi:
                    mid = (lo + hi) // 2
                    if filtered_subsets2[mid][0] <=
                    ↪ remaining_capacity:
                        lo = mid + 1
                    else:
                        hi = mid - 1

                # If we found a valid subset in subsets2, update
                ↪ the max profit
                if hi >= 0:
                    # max_profit = max(max_profit, profit1 +
                    ↪ filtered_subsets2[hi][1])
                    total_profit = profit1 +
                    ↪ filtered_subsets2[hi][1]
                    if total_profit > max_profit:
                        max_profit = total_profit
                        best_combination = (
```

```
                            subset1_items + filtered_subsets2[hi][2]
                        )  # Combine items from both subsets

            # return subsets1, subsets2, max_profit
            # return max_profit
            return best_combination, max_profit

        selected_items, max_profit = horowitz_sahni_knapsack(items)

        solution = [1 if item in selected_items else 0 for item in
        ↪  items]
        return solution, max_profit
```

.4.3. Cài đặt thuật toán Toth

```
    def dynamic_programming_Toth(self, items: list[Item]):
        """
        Find the maximum value that can be stored in the knapsack
        ↪  using dynamic programming
        with the Toth algorithm.

        The time complexity is O(n*W) where n is the number of
        ↪  items and W is the capacity of the knapsack.

        This algorithm is a variant of the dynamic programming
        ↪  algorithm for 0/1 knapsack problem.
        It uses a binary search tree to store the maximum value
        ↪  that can be stored in the knapsack
        with the given capacity.

        Returns
```

```
13          -------
14          result : list
15              A list of 0/1 indicating whether the item should be
                ↪ included or not.
16          total_value : int
17              The total value of the items.
18          total_weight : int
19              The total weight of the items.
20          """
21
22          def rec(v, c, P, X, w_m, p_m, b):
23              """
24              Update the dynamic programming table P and X from the
                ↪ given v to v + w_m.
25
26              Parameters
27              ----------
28              v : int
29                  The current value of the maximum value that can be
                    ↪ stored in the knapsack.
30              c : int
31                  The capacity of the knapsack.
32              P : list
33                  The dynamic programming table to store the maximum
                    ↪ value that can be stored in the knapsack.
34              X : list
35                  The dynamic programming table to store the number
                    ↪ of items that are included in the optimal
                    ↪ solution.
36              w_m : int
37                  The weight of the current item.
```

```
38              p_m : int
39                  The value of the current item.
40              b : int
41                  The power of 2 that is used to calculate the next
                    ↪ value of v.
42
43              Returns
44              -------
45              v : int
46                  The updated value of the maximum value that can be
                    ↪ stored in the knapsack.
47              b : int
48                  The updated value of b.
49              P : list
50                  The updated dynamic programming table P.
51              X : list
52                  The updated dynamic programming table X.
53              """
54              if v < c:
55                  u = v
56                  v = min(v + w_m, c)
57                  for hat_c in range(u + 1, v + 1):
58                      P[hat_c] = P[u]
59                      X[hat_c] = X[u]
60
61              for hat_c in range(v, w_m - 1, -1):
62                  if P[hat_c] < P[hat_c - w_m] + p_m:
63                      P[hat_c] = P[hat_c - w_m] + p_m
64                      X[hat_c] = X[hat_c - w_m] + b
65
66              b *= 2
```

```
            return v, b, P, X

        P = [0] * (self.capacity + 1)
        X = [0] * (self.capacity + 1)

        for hat_c in range(0, items[0].weight):
            P[hat_c] = 0
            X[hat_c] = 0

        v = items[0].weight
        b = 2
        P[v] = items[0].value
        X[v] = 1

        for i in range(1, self.num_items):
            v, b, P, X = rec(v, self.capacity, P, X,
             ↪  items[i].weight, items[i].value, b)

        z = P[self.capacity]
        x = [0] * self.num_items
        remain_weight = self.capacity

        while remain_weight > 0 and X[remain_weight] != 0:
            for i in range(self.num_items - 1, -1, -1):
                if X[remain_weight] & (1 << i):
                    x[i] = 1
                    remain_weight = remain_weight - items[i].weight
        if remain_weight > 0:
            return [1] * self.num_items, max(P), self.capacity
        return x, z
```

TÀI LIỆU THAM KHẢO

Tiếng Anh:

[1] Silvano Martello and Paolo Toth (1987), "Algorithms for knapsack problems", *North-Holland Mathematics Studies*, 132, pp. 213–257.

[2] Silvano Martello and Paolo Toth (1990), *Knapsack problems: algorithms and computer implementations*, John Wiley & Sons, Inc., USA, ISBN: 0471924202.

Chỉ mục