BỘ GIÁO DỤC VÀ ĐÀO TẠO

ĐẠI HỌC KINH TẾ TP HỒ CHÍ MINH

TRƯỜNG CÔNG NGHỆ VÀ THIẾT KẾ

ĐỒ ÁN MÔN HỌC

ĐỀ TÀI:

GIẢI BÀI TOÁN CÁI TÚI (KNAPSACK PROBLEM) BẰNG GIẢI THUẬT HILL CLIMBING

Học phần: Trí Tuệ Nhân Tạo

Nhóm Sinh Viên:

1. NGUYỄN THỊ NGỌC NHI
2. VĂN NGỌC NHƯ QUỲNH
3. NGUYỄN THỊ THU TRANG
4. NGUYỄN NHẬT THẢO VY

Chuyên Ngành: KHOA HỌC DỮ LIỆU

Khóa: K47

Giảng Viên: TS. Đặng Ngọc Hoàng Thành

TP. Hồ Chí Minh, Ngày 30 tháng 05 năm 2023

MỤC LỤC

CHƯƠNG 1. TỔNG QUAN

1.1. Giới Thiệu Về Bài Toán Cái Túi (Knapsack Problem)

1.1.1. Giới thiệu chung về bài toán cái túi (Knapsack Problem)

- Bài toán cái túi (Knapsack Problem) là một bài toán tối ưu hóa trong lĩnh vực khoa học máy tính và tối ưu hóa kinh tế. Bài toán đặt ra câu hỏi: Nếu có một cái túi có thể chứa tối đa một trọng lượng nhất định và một danh sách các vật phẩm có khối lượng và giá trị khác nhau, thì làm thế nào đưa vào túi những vật phẩm để giá trị của chúng lớn nhất có thể mà không vượt quá trọng lượng tối đa của túi?
- Ví dụ: Có tiếng còi báo cháy vang lên tại căn hộ của An và anh ấy quyết định mang theo một vài món đồ có giá trị sau đó rời khỏi tòa chung cư. Anh ấy chỉ có thể sử dụng một chiếc balo mà trọng lượng lớn nhất có thể mang theo là 10kg. Vậy An phải cân nhắc nên mang theo những món đồ nào có giá trị nhất và trọng lượng các món đồ ấy không vượt quá 10kg.
- Knapsack Problem là một bài toán NP-khó, nghĩa là không có thuật toán đa thức thời gian chính xác nào có thể giải quyết nó cho tất cả các trường hợp. Tuy nhiên, có nhiều thuật toán tìm kiếm, quy hoạch động và tham lam có thể được sử dụng để giải quyết bài toán với độ chính xác khác nhau và độ phức tạp thời gian khác nhau.
- Knapsack Problem có nhiều ứng dụng thực tế trong các lĩnh vực như quản lý tài sản, lập kế hoạch sản xuất, vận chuyển, đóng gói và quy hoạch đầu tư.

1.1.2. Các dạng bài toán phổ biến của bài toán cái túi (Knapsack Problem)

- **0/1 Knapsack Problem:** Đây là dạng bài toán cơ bản nhất, trong đó mỗi vật phẩm chỉ có thể được chọn hoặc không được chọn để đưa vào túi, không thể chọn một phần của vật phẩm đó.
- **Bounded Knapsack Problem:** Đây là dạng bài toán tương tự như dạng 0/1, tuy nhiên mỗi vật phẩm có một số lượng giới hạn và có thể được chọn nhiều lần.
- **Unbounded Knapsack Problem:** Đây là dạng bài toán mà trong đó mỗi vật phẩm có thể được chọn nhiều lần và không có giới hạn số lần được chọn.
- **Multiple Knapsack Problem:** Đây là dạng bài toán mà trong đó có nhiều túi và các vật phẩm có thể được đặt vào bất kỳ túi nào. Mục tiêu là tối đa hóa giá trị của các vật phẩm trong tất cả các túi mà không vượt quá khối lượng tối đa của mỗi túi.
- **Bin Packing Knapsack Problem:** Các vật phẩm có kích thước và mục tiêu là sắp xếp chúng vào một số lượng hữu hạn các thùng sao cho tất cả các thùng đều được sử dụng và số lượng thùng được sử dụng là ít nhất có thể.

Mỗi dạng bài toán Knapsack Problem đều có các yêu cầu và ràng buộc khác nhau. Vì vậy, các phương pháp giải quyết cũng khác nhau và phụ thuộc vào dạng bài toán cụ thể.

1.1.3. Ứng dụng của bài toán cái túi (Knapsack Problem):

Bài toán cái túi (Knapsack Problem) được áp dụng rộng rãi trong nhiều lĩnh vực khác nhau như:

- **Tài chính:** Knapsack Problem được sử dụng trong lĩnh vực quản lý tài sản và đầu tư. Ví dụ như là việc chọn các khoản đầu tư để đạt được lợi nhuận cao nhất trong một số hạn chế như mức rủi ro, số tiền đầu tư tối đa, thời gian đầu tư,...
- **Đóng gói và vận chuyển:** Knapsack Problem được sử dụng trong quá trình đóng gói hàng hóa, chọn các sản phẩm đóng gói đảm bảo sự an toàn và hiệu quả trong quá trình vận chuyển. Ví dụ như chọn các sản phẩm để đóng gói trong một chiếc thùng để tiết kiệm không gian và giảm thiểu chi phí vận chuyển.
- **Quản lý tài nguyên:** Knapsack Problem được sử dụng trong quản lý tài nguyên, ví dụ như lập kế hoạch sản xuất để sử dụng nguyên liệu và máy móc một cách hiệu quả nhất.
- **Mã hóa và bảo mật:** Knapsack Problem được sử dụng trong mã hóa và bảo mật.
- **Phân tích dữ liệu:** Knapsack Problem cũng có thể được sử dụng trong việc phân tích dữ liệu và tối ưu hóa quá trình thu thập dữ liệu.

1.2. Phát Biểu Bài Toán

- Cho một túi có trọng lượng tối đa là W gồm n vật phẩm khác nhau, với vật phẩm i có trọng lượng w[i] và giá trị v[i]. Nhiệm vụ đặt ra là chọn ra các vật phẩm để vào túi sao cho tổng trọng lượng của chúng không vượt quá trọng lượng tối đa của túi, và tổng giá trị của các vật phẩm lấy được là lớn nhất.
    - W: trọng lượng tối đa của túi
    - n: số lượng vật phẩm
    - w[i]: mảng chưa trọng lượng của từng vật phẩm i ($1 \leq i \leq n$)
    - v[i]: mảng chứa giá trị của từng vật phẩm i ($1 \leq i \leq n$)
- Ràng buộc:
    - Mỗi vật phẩm chỉ có thể được chọn hoặc không được chọn (phiên bản 0/1 Knapsack).
    - Trọng lượng và giá trị của mỗi vật phẩm là số nguyên dương.
    - Trọng lượng tối đa của túi là một số nguyên dương.
- Bước thực hiện:
    - Xây dựng bảng hoặc mảng hai chiều có kích thước $(n+1) \times (W+1)$.
    - Khởi tạo giá trị cho các ô đầu tiên của bảng là 0.
    - Duyệt qua từng vật phẩm và trọng lượng:
        - Nếu trọng lượng vật phẩm i nhỏ hơn hoặc bằng trọng lượng đang xét, ta so sánh giá trị tại ô (i-1, j) và (i-1, j-w[i]) + v[i]. Chọn giá trị lớn nhất và gán cho ô (i, j).
        - Nếu trọng lượng vật phẩm i lớn hơn trọng lượng đang xét, ta gán giá trị của ô (i-1, j) cho ô (i, j).
    - Giá trị tối ưu của bài toán Knapsack sẽ nằm ở ô cuối cùng của bảng.
    - Truy vết ngược từ ô cuối cùng để xác định các vật phẩm được chọn.

- Độ phức tạp: Thuật toán Hill Climbing có độ phức tạp thời gian là O(∞), độ phức tạp không gian là o(b).

1.3. Một Số Hướng Tiếp Cận Giải Quyết Bài Toán

1.3.1. Giải thuật Greedy

Giải thuật tham lam (Greedy algorithm) là phương pháp đơn giản để giải quyết bài toán Knapsack theo kiểu metaheuristic, trong đó chúng ta tìm kiếm lựa chọn tối ưu địa phương ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục. Thuật toán này có độ phức tạp là O(NlogN) và chỉ yêu cầu một mảng một chiều để ghi lại chuỗi giải pháp. Tuy nhiên, giải pháp cuối cùng có thể chưa được tối ưu toàn cục.

1.3.2. Dynamic Programming

Giải thuật quy hoạch động (Dynamic Programming) là một kĩ thuật giúp đơn giản hóa bài toán Knapsack bằng cách chia bài toán lớn thành các bài toán con và sử dụng lời giải của các bài toán con để tìm lời giải cho bài toán ban đầu. Thuật toán này có độ phức tạp là O và yêu cầu một mảng hai chiều. Vì vậy, đây là một trong những giải thuật chính xác để giải quyết bài toán Knapsack.

CHƯƠNG 2. GIẢI THUẬT HILL CLIMBING

2.1. Giới Thiệu Về Giải Thuật Hill Climbing

2.1.1. Khái niệm

Hill Climbing Algorithm là một thuật toán tìm kiếm cục bộ liên tục di chuyển theo hướng tăng dần giá trị để tìm ra đỉnh hoặc giải pháp tốt nhất cho vấn đề. Nó kết thúc khi nó đạt đến giá trị cao nhất mà không có giải pháp nào có giá trị cao hơn.

2.1.2. Đặc điểm

- Biến thể Tạo và Thử nghiệm: Hill Climbing là biến thể của phương pháp Tạo và Thử nghiệm. Phương pháp Tạo và Kiểm tra tạo ra phản hồi giúp quyết định hướng di chuyển tiếp theo trong không gian tìm kiếm.
- Cách tiếp cận tham lam: Thuật toán tìm kiếm Hill Climbing di chuyển theo hướng tối ưu hóa chi phí.
- Không quay lui: Nó không quay lại không gian tìm kiếm, vì nó không nhớ các trạng thái trước đó.

2.1.3. Các loại thuật toán Hill Climbing

- Simple Hill Climbing: Là cách đơn giản nhất để thực hiện thuật toán Hill Climbing. Nó kiểm tra từng nút lân cận của trạng thái hiện tại và chọn trạng thái đầu tiên giúp tối ưu hóa chi phí hiện tại để đặt làm trạng thái hiện tại. Nó chỉ kiểm tra nếu đó là một trạng thái kế thừa.
- Steepest-Ascent hill climbing: Là một biến thể của Simple Hill Climbing. Thuật toán này kiểm tra tất cả các nút lân cận của trạng thái hiện tại và chọn một nút lân cận gần nhất với trạng thái mục tiêu. Thuật toán này tiêu tốn nhiều thời gian hơn vì nó tìm kiếm nhiều hàng xóm.
- Stochastic hill Climbing: Là một biến thể của Simple Hill Climbing. Thuật toán này không kiểm tra tất cả những nút lân cận của nó trước khi duyệt qua nút khác. Thay vào đó, thuật toán này chọn ngẫu nhiên một nút lân cận và đưa ra quyết định chọn nút đó làm trạng thái hiện tại hay duyệt qua các nút khác.

2.1.3. Mô tả thuật toán

```
Algorithm: Hill Climbing
Current ← initial state
End ← false
while not End do
    Successors ← generate_successor(Current)
    Successors ← sort_and_prune_bad_solutions(Successors, Current)
    if not empty?(Successors) then
        Current ← best_successor(Successors)
    else
        End ← true
    end
end
```

Bước 1: Đánh giá trạng thái ban đầu, nếu là trạng thái mục tiêu thì trả về thành công và dừng, nếu không thì chuyển trạng thái hiện tại thành trạng thái ban đầu.
Bước 2: Vòng lặp cho đến khi tìm ra giải pháp hoặc trạng thái hiện tại không thay đổi.

- Hãy để SUCC là một trạng thái sao cho bất kỳ người kế thừa nào của trạng thái hiện tại sẽ tốt hơn nó.
- Đối với mỗi toán tử áp dụng cho trạng thái hiện tại:
- Áp dụng toán tử mới và tạo trạng thái mới.
- Đánh giá trạng thái mới.
- Nếu đó là trạng thái mục tiêu, hãy trả lại và thoát ra, nếu không, hãy so sánh nó với SUCC.
- Nếu nó tốt hơn SUCC, thì hãy đặt trạng thái mới là SUCC.
- Nếu SUCC tốt hơn trạng thái hiện tại, thì giữ nguyên trạng thái hiện tại là SUCC.

Bước 3: Thoát.

2.2. Ứng Dụng Giải Thuật Hill Climbing Cho Bài Toán Cái Túi (Knapsack Problem)

Giải thuật Hill Climbing sẽ chọn ngẫu nhiên 1 trạng thái cho giải pháp ban đầu (solution) của bài toán, bao gồm các biến:

- solution: thể hiện trạng thái của từng item là đã được cho vào Knapsack hay chưa, lần lượt tương ứng với giá trị 1 hoặc 0.
- solution_value: tổng giá trị của các item trong Knapsack.
- solution_weight: tổng trọng lượng các item trong Knapsack.

Trong trường hợp solution_weight > capability (biến tải trọng tối đa của Knapsack) thì khởi tạo 1 solution khác.

Kế tiếp thuật toán sẽ tạo ra các giải pháp lân cận (neighbor) của giải pháp hiện tại (solution) bằng phương pháp lật bit. Tính toán giá trị và khối lượng của từng neighbor được tạo (neighbor_value và neighbor_weight).

Sau đó so sánh, tìm ra neighbor tốt nhất (best_neighbor, khởi tạo ban đầu giá trị None), bằng cách gán best_value = solution_value, nếu neighbor_weight < capability đồng thời neighbor_value > best_value thì cập nhật lân cận thành giải pháp mới. Gán các giá trị của neighbor cho best_neighbor.

Lặp lại so sánh và cập nhật solution cho đến lúc không tìm được best_neighbor nào tốt hơn (best_neighbor == None).

Solution cuối cùng được ghi nhận có thể là giải pháp tối ưu cục bộ (local maximum) đối với bài toán.

Các bước cài đặt:

1. Khởi tạo ngẫu nhiên 1 mảng 2 chiều (items = [ ]) có [n_items] phần tử (item) với chiều thứ nhất là giá trị (value), chiều thứ hai là trọng lượng (weight).
2. Tạo một giải pháp ban đầu (solution) ngẫu nhiên.
3. Tính giá trị (value) và trọng lượng (weight) của giải pháp (solution).
4. Tạo và lưu vào mảng các giải pháp lân cận (neighbors).
5. Chạy vòng lặp:
    5.1. Khởi tạo và gán best_value = solution_value và best_neighbor = None
    5.2. Tính giá trị neighbor_value và neighbor_weight của các neighbor
    5.3. So sánh neighbor_value của các neighbor thỏa điều kiện về tải trọng (neighbor_weight <= capability) của knapsack với best_value, nếu neighbor_value > best_value thì gán best_neighbor = neighbor, best_value = neighbor_value, solution_weight = neighbor_weight
    5.4. Nếu best_neighbor != None thì cập nhật giải pháp solution = best_neighbor, solution_value = best_value. Ngược lại thoát khỏi vòng lặp nếu không có lân cận nào tốt hơn giải pháp hiện tại.
6. Trả về kết quả solution, solution_value, solution_weight.
7. Chạy thuật toán với thông số đầu vào là items và capability.

CHƯƠNG 3. CÁC KẾT QUẢ THỰC NGHIỆM

3.1. Các Tình Huống

Sample 1: Dữ liệu đầu vào gồm 5 item, khối lượng của mỗi item không vượt quá 50. Trọng lượng tối đa của knapsack là 100. Giá trị của các item được in trong kết quả thực thi.
Kết quả 3 lần chạy thử:

```
Input: 5 items, capability: 100
Items: [(4, 38), (8, 17), (4, 39), (8, 8), (4, 6)]
********************************************************************************
Execution 0
Output
Value: 24
Solution: [1, 1, 0, 1, 1]
Weight: 69
Intial solution: [1, 0, 0, 1, 1]
********************************************************************************
Execution 1
Output
Value: 20
Solution: [1, 1, 1, 0, 1]
Weight: 100
Intial solution: [1, 1, 1, 0, 1]
********************************************************************************
Execution 2
Output
Value: 12
Solution: [0, 1, 0, 0, 1]
Weight: 23
Intial solution: [0, 0, 0, 0, 1]
********************************************************************************
```

Sample 2: Dữ liệu đầu vào gồm 5 item, khối lượng của mỗi item không vượt quá 100. Trọng lượng tối đa của knapsack là 50. Trong mẫu này có 2 item có weight vượt khỏi capability của knapsack.

Kết quả 3 lần chạy thử:

```
Input: 5 items, capability: 50
Items: [(7, 63), (3, 7), (7, 45), (7, 90), (4, 16)]
********************************************************************************
Execution 0
Output
Value: 7
Solution: [0, 0, 1, 0, 0]
Weight: 45
Intial solution: [0, 0, 0, 0, 0]
********************************************************************************
Execution 1
Output
Value: 7
Solution: [0, 1, 0, 0, 1]
Weight: 23
Intial solution: [0, 0, 0, 0, 1]
********************************************************************************
Execution 2
Output
Value: 7
Solution: [0, 1, 0, 0, 1]
Weight: 23
Intial solution: [0, 1, 0, 0, 1]
********************************************************************************
```

3.2. Phân Tích và Đánh Giá

Sample 1: [(4, 38), (8, 17), (4, 39), (8, 8), (4, 6)], capability =100

- Sample này có nhiều hơn 1 giải pháp tối ưu nhất về value. Lần chạy đầu tiên đã đạt được 1 trong những kết quả tối ưu đó. Với giải pháp ban đầu được khởi tạo là: [1, 0, 0, 1, 1] vậy **solution_value = 16, solution_weight = 52.** Các lân cận thuật toán tạo được là:

Neighbor	Value	Weight	Ghi chú
0, 0, 0, 1, 1	12	14	
1, 1, 0, 1, 1	**24**	69	Max value
1, 0, 1, 1, 1	20	91	
1, 0, 0, 0, 1	8	44	
1, 0, 0, 1, 0	12	46	

  Có thể thấy neighbor thứ hai có giá trị tốt nhất, nên neighbor này được chọn để cập nhật giải pháp. Thuật toán bỏ qua neighbor đầu tiên vì nó không tốt hơn giải pháp

ban đầu, sau đó cập nhật neighbor thứ 2 vì nó tốt hơn và dừng việc cập nhật giải pháp vì các neighbor sau đó không tốt hơn giải pháp hiện tại.

- Lần chạy thứ 2 có giải pháp ban đầu được khởi tạo là [1, 1, 1, 0, 1] vậy **solution_value = 20, solution_weight = 100.** Các lân cận:

Neighbor	Value	Weight	Ghi chú
0, 1, 1, 0, 1	16	62	
1, 0, 1, 0, 1	12	83	
1, 1, 0, 0, 1	16	61	
1, 1, 1, 1, 1	**28**	108	> capability
1, 1, 1, 0, 0	16	94	

Ở lần chạy này, không có neighbor nào được cập nhật thành solution. Giải pháp ban đầu được giữ làm solution cuối cùng. Vì value của các neighbor đều nhỏ hơn value ban đầu, trừ neighbor thứ 4 có value lớn hơn. Tuy nhiên weight của neighbor này vi phạm vì vượt khỏi capability của knapsack nên cũng không chọn được. Và solution cuối cùng của lần chạy này chỉ đạt tối ưu cục bộ (local maximum) chứ không phải tối ưu toàn cục (global maximum).

- Lần chạy cuối có giải pháp ban đầu được khởi tạo là [0, 0, 0, 0, 1] vậy **solution_value = 4, solution_weight = 6.** Các lân cận:

Neighbor	Value	Weight	Ghi chú
1, 0, 0, 0, 1	8	44	
0, 1, 0, 0, 1	**12**	33	Max value
0, 0, 1, 0, 1	8	45	
0, 0, 0, 1, 1	**12**	14	Max value
0, 0, 0, 0, 0	0	0	

Trong lần chạy này, có 2 neighbor có cùng giá trị value cao nhất là 12, lúc này thuật toán lấy neighbor đầu tiên. Việc chọn neighbor nào trong trường hợp này tùy thuộc vào điều kiện lúc xây dựng thuật toán. Trong thuật toán này điều kiện là neighbor_value > best_value nên sẽ chọn neighbor trước. Có thể mở rộng thuật toán

hơn nữa để tối ưu giải pháp hơn bằng ràng buộc về weight, có thể xây dựng để thuật toán chọn neighbor có weight nhỏ hơn.

Sample 2: **[(7, 63), (3, 7), (7, 45), (7, 90), (4, 16)]**, capability =50

- Lần chạy đầu: giải pháp ban đầu được khởi tạo là [0, 0, 0, 0, 0] vậy **solution_value = 0, solution_weight = 0.** Các lân cận:

Neighbor	Value	Weight	Ghi chú
1, 0, 0, 0, 0	**7**	63	> capability
0, 1, 0, 0, 0	3	7	
0, 0, 1, 0, 0	**7**	45	Max value
0, 0, 0, 1, 0	**7**	90	> capability
0, 0, 0, 0, 1	4	16	

Lần chạy này có đến 3 neighbor đạt value lớn nhất. Tuy nhiên 2 trong số chúng vi phạm điều kiện vì weight > capability, nên chỉ neighbor thứ 3 được chọn để cập nhật solution. Đây cũng chính là 1 trong 2 solution tối ưu nhất của bài toán (global maximum).

- Lần chạy thứ 2: giải pháp ban đầu được khởi tạo là [0, 0, 0, 0, 1] vậy **solution_value = 4, solution_weight = 16.** Các lân cận:

Neighbor	Value	Weight	Ghi chú
1, 0, 0, 0, 1	**11**	79	> capability
0, 1, 0, 0, 1	7	23	Max value
0, 0, 1, 0, 1	**11**	61	> capability
0, 0, 0, 1, 1	**11**	106	> capability
0, 0, 0, 0,0	0	0	

Lần chạy này có 3 neighbor đạt value lớn nhất là 11 và cũng đồng thời lớn hơn value của solution ban đầu. Tuy nhiên tất cả chúng đều vi phạm điều kiện về weight nên không được chọn. Thuật toán chọn giải pháp tốt nhất trong 2 neighbor còn lại. Đây cũng là solution tối ưu nhất còn lại đối với sample này.

- Lần chạy 3: giải pháp ban đầu được khởi tạo là [0, 1, 0, 0, 1] vậy **solution_value = 7, solution_weight = 23.** Các lân cận:

Neighbor	Value	Weight	Ghi chú
1, 1, 0, 0, 1	**14**	86	> capability
0, 0, 0, 0, 1	4	16	
0, 1, 1, 0, 1	**14**	68	> capability
0, 1, 0, 1, 1	**14**	113	> capability
0, 1, 0, 0, 0	3	7	

Lần chạy này không neighbor nào được cập nhật thành solution. Có 3 neighbor đạt value cao nhất là 14 và cả 3 đều vi phạm điều kiện về weight. 2 neighbor còn lại đều có value bé hơn solution ban đầu, vì vậy solution được giữ nguyên. Trong trường hợp này, giải pháp ban đầu được khởi tạo ngẫu nhiên cũng chính là giải pháp tối ưu nhất của bài toán, vì thế không có bất kỳ neighbor nào được cập nhật.

Dùng giải thuật Hill climbing giải bài toán knapsack có ưu điểm là đơn giản, hiệu quả và phù hợp với bài toán đơn giản, có bộ dữ liệu nhỏ, Tuy nhiên, nó có thể sẽ không đưa ra được phương án tối ưu cho bài toán phức tạp hơn, với kích thước lớn. Vì kết quả từ thuật toán này có thể chỉ là phương án tối ưu cục bộ (local maxima) thay vì toàn cục (global maximum), điều này phụ thuộc vào trạng thái khởi tạo ban đầu.

CHƯƠNG 4. KẾT LUẬN

4.1. Các Kết Quả Đạt Được

Khi giải bài toán Knapsack bằng thuật toán Hill Climbing, kết quả đạt được có thể khác nhau tùy thuộc vào điểm khởi tạo ban đầu và cách lựa chọn các bước tiếp theo. Tuy nhiên, Hill Climbing là một thuật toán tìm kiếm cục bộ, tức là nó chỉ tìm kiếm các giải pháp tốt hơn trong lân cận của giải pháp hiện tại mà không đảm bảo rằng giải pháp tìm được là tối ưu toàn cục. Trong bài toán Knapsack, việc tìm kiếm một lời giải tốt hơn trong lân cận có thể dẫn đến việc bỏ qua các lời giải tốt hơn ở những vị trí xa hơn trong không gian lời giải. Điều này có thể dẫn đến việc bỏ qua lời giải tối ưu và thuật toán Hill Climbing có thể dừng lại ở một điểm cực tiểu cục bộ. Thuật toán Hill Climbing không được coi là hoàn chỉnh khi giải bài toán Knapsack. Để giải bài toán Knapsack một cách hoàn chỉnh và đảm bảo tìm được lời giải tối ưu toàn cục, phương pháp tìm kiếm như quy hoạch động hoặc tìm kiếm nhánh và cận thường được sử dụng.

4.2. Những Hạn Chế và Hướng Phát Triển

Sử dụng thuật toán Hill Climbing để giải bài toán Knapsack có những hạn chế sau:

- Thuật toán Hill Climbing là thuật toán tìm kiếm cục bộ, không đảm bảo tìm được lời giải tối ưu toàn cục. Nó có thể dừng lại ở một điểm cực tiểu cục bộ mà không khám phá toàn bộ không gian lời giải, bỏ qua các lời giải tốt hơn nằm ở xa hơn.
- Bài toán Knapsack có cấu trúc ràng buộc và không gian lời giải rời rạc. Trong khi Hill Climbing thích hợp cho các bài toán liên tục, nó không phù hợp để đảm bảo ràng buộc trọng lượng và lựa chọn các vật phẩm trong bài toán Knapsack.
- Thuật toán Hill Climbing không có khả năng quay lui hoặc thử các lựa chọn ngược lại. Trong bài toán Knapsack, việc lựa chọn hoặc không chọn một vật phẩm có thể ảnh hưởng lớn đến kết quả. Hill Climbing không thể xem xét các bước giảm giá trị hoặc trọng lượng để tìm kiếm lời giải tốt hơn.
- Điểm khởi tạo ban đầu của Hill Climbing có thể ảnh hưởng đáng kể đến kết quả cuối cùng. Nếu điểm khởi tạo ban đầu không tốt hoặc không gần lời giải tối ưu, thuật toán có thể dừng lại ở một lời giải không tối ưu.
- Thuật toán Hill Climbing không có khả năng tiếp tục tìm kiếm khi không có cải tiến. Nếu thuật toán đã đạt được một điểm cực tiểu cục bộ và không có bước tiếp theo nào để cải tiến giá trị, nó có thể dừng lại và không khám phá các vùng không gian lời giải khác có thể chưa lời giải tối ưu.

Các hướng phát triển :

- Thuật toán Hill Climbing cải tiến: cải tiến thuật toán Hill Climbing để tránh rơi vào điểm cực tiểu cục bộ bằng cách áp dụng các kỹ thuật như Hill Climbing ngẫu nhiên (Random Restart Hill Climbing) hoặc Hill Climbing kết hợp với Simulated Annealing. Những phương pháp này giúp tìm kiếm trên không gian lời giải rộng hơn và có khả năng thoát khỏi điểm cực tiểu cục bộ.
- Kết hợp các phương pháp: một cách tiếp cận khác là kết hợp các phương pháp khác nhau để tìm kiếm lời giải tốt nhất. Ví dụ, có thể sử dụng Hill Climbing để

khám phá không gian và sau đó áp dụng phương pháp quy hoạch động để tính toán giá trị tối ưu.

- Áp dụng phương pháp quy hoặc động: phương pháp quy hoạch động là một phương pháp rất hiệu quả cho bài toán Knapsack. Có thể xây dựng một bảng hoặc mảng hai chiều để lưu trữ các giá trị tối ưu và sử dụng phương pháp lập trình động để tính toán giá trị tối ưu và truy vết lời giải.
- Tìm hiểu các thuật toán tối ưu khác: Ngoài Hill Climbing ra thì còn có nhiều thuật toán tối ưu khác có thể được áp dụng cho bài toán Knapsack. Tìm hiểu và thử nghiệm các thuật toán để có thể đưa ra kết quả tốt hơn cho bài toán.

TÀI LIỆU THAM KHẢO

1. Basics of Python

2. Introduction to Hill Climbing – Artificial Intelligence (geeksforgeeks.org)

3. Thuật toán quy hoạch động (viblo.asia)

4. Different Approaches to Solve the 0/1 Knapsack Problem

5. Hill Climbing Algorithm trong trí tuệ nhân tạo (Dục Đoàn Trình)

PHỤ LỤC

Mã nguồn

https://github.com/nnttvy/AI_PRJ-Hill_climbing-Knapsack