Kinh nghiệm thiết kế thuật toán: Greedy

Hoàng Ngọc Quân

November 2023

1 Introduction

1.1 Lời nói đầu

*

- Các phương pháp thiết kế thuật toán như: ***Quay lui (Backtracking), Nhánh và cận (Branch and Bound), Tham lam (Greedy Method), Chia để trị (Divide and Conquer) và Quy hoạch động (Dynamic Programming)*** là lớp các chiến lược có tính **tổng quát**, mỗi chiến lược thể hiện cho một **khía cạnh góc nhìn riêng** khi tiếp cận và tìm thuật toán phù hợp cho bài toán.

- Trên thực tế các bài toán là muôn hình vạn trạng, ta không thể xây dựng một phương pháp vạn năng để giải quyết mọi bài toán (*"One size fit all"*). Các phương pháp thiết kế thuật toán bên trên chỉ là **các chiến lược**, có tính định hướng giúp chúng ta nhanh chóng tìm được lời giải phù hợp cho bài toán. Song việc áp dụng các chiến lược để tìm ra thuật toán cho một bài toán cụ thể còn đòi hỏi nhiều sự **sáng tạo**.

- Cần chú ý rằng đây là các phương pháp để thiết kế thuật toán không phải là một thuật toán cụ thể. Vì vậy khi viết Greedy Algorithm**s** ta cần hiểu rằng đây là một lớp các thuật toán được thiết kế bằng chiến lược tham lam chứ không phải đang nói đến một thuật toán cho một bài toán cụ thể như *QuickSort Algorithm, BinarySearch Algorithm,....*

- Do đó chúng ta sẽ không tìm hiểu về các chiến lược nói trên theo cách mà chúng ta vẫn thường "học"với các thuật toán cụ thể khác (*step-to-step*), mà ta sẽ tìm hiểu thông qua việc nắm bắt được **ý tưởng chung** của chiến lược, **tính chất** của các bài toán, **mô hình** triển khai thuật toán, cũng như những **ưu nhược điểm** của chiến lược thông qua việc khảo sát một số bài toán mà tôi sẽ gọi là các **case-study**.

- Cách tìm hiểu như trên mang tính **kinh nghiệm** (experience) nhiều hơn là **lý thuyết học thuật** (academic theory) và có phần phụ thuộc vào case-study đại diện mà chúng ta sẽ chọn để tìm hiểu, tức là nó không thể hoàn toàn thể hiện hết một cách trọn vẹn và đầy đủ về các chiến lược, nhưng hy vọng sẽ cho chúng ta một cái nhìn chung, nền tảng, **xây dựng trực giác** để chúng ta tiếp tục tự tìm hiểu về các bài toán và vấn đề khác.

1.2 Case-study

- Naruto
- One Piece
- Nobita
- Fairy Tail

1.3 Đặt vấn đề

*

- Trong nhiều *bài toán tối ưu*, việc tìm *lời giải* cho bài toán được quy về việc tìm *nghiệm* - tìm *vector cấu hình* hữu hạn $(x_1, x_2, x_3, ..., x_K)$ thỏa mãn *điều kiện ràng buộc* G và làm cho *hàm mục tiêu* F đạt giá trị lớn nhất (nhỏ nhất) có thể.

- (*) Trong bài báo cáo này, các từ ngữ: *nghiệm, cấu hình nghiệm, lời giải,...* tuy không mang nghĩa hoàn toàn giống nhau, nhưng trong ngữ cảnh có thể hiểu chúng đang cùng chỉ đến một thuật ngữ.

- Xét case-study Naruto: việc *biểu diễn lời giải* cho bài toán có thể quy về việc biểu diễn vector (c, k) tương ứng với lượng chakra và lượng phân thân mà Naruto cần sử dụng. Trong đó điều kiện ràng buộc G là: $0 <= c, k <= N, |c - k| <= X$ và hàm mục tiêu F là: $ac + bk$ đạt giá trị lớn nhất có thể.

- Vậy có thể thấy với các cách tiếp cận theo chiến lược thông thường như: *Brute Force hay Backtracking* thì với mỗi bài toán điều cần thiết nhất là chúng ta chỉ cần xác định đúng *điều kiện ràng buộc* G và *hàm mục tiêu* F rồi sau đó tìm cách **biểu diễn lời giải** cho phù hợp là coi như bài toán đã có thể giải được (bỏ qua chi tiết cài đặt).

- Tuy nhiên các cách tiếp cận trên gặp một số hạn chế sau đây:

  - (1) là ***không gian nghiệm*** S - tức tập các lời giải tiềm năng cần được biểu diễn có thể quá lớn.

  - (2) là chi phí cho việc thực hiện các thao *tác kéo theo* sau đó như: kiểm tra điều kiện G và tính toán hàm mục tiêu F có thể không đơn giản.

- Đối với vấn đề (2) ta có thể đã được tìm hiểu cách giải quyết thông qua bộ môn *DSA* bằng cách dựa vào các **cấu trúc dữ liệu** và các **kỹ thuật** nhất định trên nguyên lý **tái tổ chức** lại dữ liệu hoặc dựa vào các **điểm chung (lặp lại) trong các thành phần tính toán** để có thể giảm thiểu được chi phí.

- Nhưng còn vấn đề (1) thì sao? Rõ ràng nếu ngay từ ban đầu mà chi phí cho việc **biểu diễn lời giải** đã không phù hợp rồi thì cho dù ta có sử dụng các cấu trúc dữ liệu hay các kỹ thuật tốt đến bao nhiêu thì chi phí cho cả bài toán cũng sẽ không phù hợp được. Đây là lúc mà chúng ta cần thay đổi hướng tiếp cận bài toán bằng các chiến lược "tiên tiến"hơn nhằm làm giảm đáng kể chi phí (1).

- Có nhiều phương pháp để làm được điều đó và **Greedy** là một phương pháp thiết kế thuật toán có hướng tiếp cận bài toán làm giảm đáng kể chi phí (1).

- Ví dụ như case-study Naruto, lúc đầu ta biểu diễn lời giản của bài toán bằng một không gian nghiệm là một miền $2D$, nhưng sau đó ta nhận ra rằng ta chỉ cần biểu diễn c (hoặc k) là đủ do việc hàm F đạt giá trị lớn nhất thì k luôn phụ thuộc vào c -> từ $2D$ ta giảm về $1D$. Cuối cùng bằng kiến thức về **Quy hoạch tuyến tính** ta biết rằng hàm F chỉ luôn đạt giá trị lớn nhất (nhỏ nhất) trên các đỉnh của miền G -> từ $1D$ ta giảm về $0D$.

2 Greedy

Greedy là một phương pháp giải các **bài toán tối ưu**. Các thuật toán tham lam dựa vào sự đánh giá **tối ưu cục bộ** (local optimum) để đưa ra quyết định tức thì **tại mỗi bước - giai đoạn** lựa chọn, với hy vọng cuối cùng sẽ tìm ra được phương án **tối ưu toàn cục** (global optimum).

- Ví dụ: Tập hợp của các cá nhân mạnh nhất –> Team mạnh nhất. Cũng giống như thuật toán tham lam, ví dụ trên cho chúng ta thấy cách mà thuật toán tham lam vận hành tuy nhiên nó cũng chỉ ra một điều rằng trong thực tế công thức trên không phải lúc nào cũng đúng.

2.1 Tính chất

- Các bài toán có thể giải được bằng phương pháp tham sẽ có 2 tính chất (đặc trưng) sau đây:

(1) - Optimal Substructure

(2) - Greedy choice property

2.2 Nguyên lý

*

- Khác với các thuật toán thông thường khác (thậm chí là cả ***Dynamic Programming***) là ta phải đi xét hết tất cả các trường hợp có thể của *nghiệm* và tìm *nghiệm tối ưu nhất*, còn *tham lam* tức là ta đã **biết dạng của nghiệm tối ưu** như thế nào rồi và ta chỉ đang đi xây dựng nên nghiệm tối ưu đó bằng các công cụ của máy tính mà thôi.

- Giả sử rằng ta cần tìm nghiệm tối ưu $X^*(x_1, x_2, ..., x_K)$ cho bài toán G thỏa mãn một số *điều kiện ràng buộc* và *hàm mục tiêu* nhất định như đã trình bày. Thì phương pháp tham lam sẽ vận hành như sau:

- Dựa vào tính chất (2) ta chỉ ra rằng trong số các nghiệm có thể xem là nghiệm tối ưu thì nghiệm mà có $x_1 = i_1^*$ luôn được chọn.

- Sau khi $x_1 = i_1^*$ được chọn, việc tìm nghiệm tối ưu cho bài toán G ban đầu trở thành việc tìm nghiệm tối ưu $X^{**}(x_2, x3, ..., x_K)$ cho bài toán G' với *điều kiện ràng buộc* và *hàm mục tiêu* **hoàn toàn tương đồng** với bài toán G ban đầu. Đây được gọi là tính chất (1) - **Optimal Substructure**.

- Lặp lại tính chất (2) cho bài toán G' ta sẽ đưa bài toán G' về thành G''. Cứ như vậy sau hữu hạn bước ta sẽ hoàn tất được bài toán.

*

- Như vậy để có thể sử dụng phương pháp tham lam để giải quyết một bài toán cụ thể ta chỉ cần giải quyết 2 vấn đề sau:

- Một là chỉ ra rằng bài toán trên có cấu trúc con tối ưu, tức là ta có thể giải bài toán trên bằng cách lặp qua nhiều giai đoạn mà mỗi giai đoạn ta đang đi giải các bài toán hoàn toàn tương đồng nhau.

- Hai là cần tìm một **hàm chọn** để thực hiện tính chất (2). Nếu hàm chọn của chúng ta là chính xác hoàn toàn thì nghiệm mà chúng ta thu được sẽ thật sự là một nghiệm tối ưu. Ngược lại ta chỉ có thể thu được một **nghiệm gần tối ưu** hay nói cách khác thuật toán tham lam mà chúng ta thiết kế chỉ là một thuật toán **gần đúng** mà thôi.

- Trong bài báo cáo này sẽ chủ yếu trình bày và đề cập về các case-study mà tại đó ta hoàn toàn có thể xây dựng một **hàm chọn chính xác hoàn toàn**. Việc tìm hiểu các hàm chọn hoàn toàn chính xác cho các case-study nói trên hoàn toàn có thể giúp bạn có những ý tưởng, kinh nghiệm nền tảng và trực giác cho việc xây dựng các hàm chọn gần chính xác cho các bài toán khác trong một số lĩnh vực hoặc tình huống đặc thù. (Tìm hiểu thêm về các ***thuật giải Heuristic*** và ứng dụng của chúng)

2.3 Nhận xét

*

- Nhắc lại một lần nữa đó là tham lam là ta đã biết dạng của nghiệm tối ưu sẽ như thế nào rồi và ta chỉ đang đi xây dựng nên nghiệm tối ưu đó dựa vào công cụ máy tính mà thôi. Ta không liệt kê tất cả các nghiệm và rồi tìm nghiệm tối ưu trong số các nghiệm đó.

–> Thành ra sẽ chỉ có **một và duy nhất một nghiệm** được xét đến như là nghiệm tối ưu. Do đó **chi phí cho không gian lời giải** S luôn là $O(1)$.

–> Vì vậy mà **độ phức tạp của bài toán** sẽ chỉ phụ thuộc vào chi phí của các *thao tác kéo theo* sau đó: thường bằng số giai đoạn lặp nhân với chi phí cho hàm chọn. Thành ra nếu một bài toán có thể giải được bằng nhiều phương pháp, phương pháp tham lam thường cho tốc độ tốt nhất.

- Tại mỗi giai đoạn - bước lặp: hàm chọn sẽ chỉ **chọn một phần tử duy nhất**. Nếu chọn nhiều hơn một, phương pháp tham lam có thể bị biến tướng trở thành **phương pháp chia để trị**. Từ đây gợi ý cho chúng ta **một cách tiếp cận của phương pháp tham lam từ phương pháp chia để trị**. (Vì chương trình thiết kế thì tham lam sẽ học trước nên sau khi học xong chia để trị các bạn tự tìm hiểu về ý tưởng này)

- Về góc nhìn: Nếu một bài toán mà các bạn **không nhận ra** rằng chúng có 2 tính chất đã trình bày bên trên thì điều đó không khẳng định được rằng bài toán ấy không thể giải bằng phương pháp tham lam - do có thể vấn đề đã **bị ẩn** đi khiến các bạn chưa nhìn thấy được. Ngược lại nếu một bài toán có đầy đủ 2 tính chất trên thì cũng không nhất thiết phải luôn thiết kế thuật toán giải bằng phương pháp tham lam - do **một bài toán có thể có nhiều chiến lược phù hợp khác nhau** và mỗi chiến lược chỉ thể hiện một góc nhìn riêng về cách tiếp cận bài toán đó.

+ Danh ngôn: "Vẻ đẹp của một cô gái nằm trong ánh mắt của kẻ si tình". Điều đó có nghĩa rằng nếu bạn nghĩ rằng một cô gái là đẹp thì không đồng nghĩa rằng cả thế giới sẽ cũng thấy cô ấy đẹp. Ngược lại là nếu bạn không nhận thấy cô ấy đẹp có thể là do bạn chưa tìm hiểu đủ sâu về cô ấy để có thể nhìn ra được những vẻ đẹp của cô ấy.

+ Tuy nhiên, bất cứ khi nào bạn nghi ngờ rằng một bài toán có thể giải bằng phương pháp tham lam, hãy thử kéo bài toán theo hướng của 2 tính chất đã trình bày bên trên (đưa cái bị ẩn ra ánh sáng để có thể nhìn thấy được vẻ đẹp).

2.4 Mô hình

*

- Có lẽ đây là phần ít quan trọng nhất. Bởi lẽ trong thực tế, đôi khi xuất phát từ nhu cầu cải tiến tốc độ thuật toán sẽ có những sự **linh hoạt, điều chỉnh** nhất định về cách biểu diễn thuật toán và cài đặt code so với ý tưởng gốc ban đầu chứ không phải lúc nào cũng sẽ lắp theo một form nhất định.

- Tuy nhiên, từ cấu trúc mô hình bên trên ta cần chú ý đến một số điều sau đây:

- Một là các **thành phần có thể có** trong một thuật toán tham lam mà chúng ta cần **xác định, định nghĩa** khi thiết kế và xây dựng:

+ *Tập ứng cử viên* để tạo ra *lời giải* - nghiệm tối ưu.

+ *Hàm chọn*: để theo đó lựa chọn ứng viên tốt nhất để bổ sung vào lời giải

+ *Hàm khả thi (feasibility)*: dùng để quyết định nếu một ứng viên có thể được dùng để xây dựng lời giải

+ *Hàm mục tiêu*

+ *Hàm đánh giá*: chỉ ra khi nào ta tìm ra một lời giải hoàn chỉnh - thuật toán dừng các giai đoạn lặp.

- Hai là sau khi các bạn đã thiết kế và xây dựng cho một số lượng các bài toán tham lam và bây giờ các bạn cần xây dựng và thiết kế cho một bài toán mới. Hãy cố gắng nhìn ra **các điểm chung** của các thiết kế và xây dựng cũ dựa vào mô hình bên trên để tạo thành các **mẫu có sẵn** và rồi **thiết kế và**

xây dựng cho bài toán mới dựa vào các mẫu có sẵn đó.

–> Điều này giúp tăng đáng kể khả năng *Pattern Recognition* trong *Computational Thinking* của các bạn và giúp các bạn thiết kế và triển khai một cách nhanh chóng với ít sự sai sót (*bug*) có thể gặp phải hơn.

2.5 Framework

*

- Như đã trình bày bên trên, **chi phí của phương pháp tham lam** là phụ thuộc nhiều vào chi phí của các thao tác kéo theo sau đó hơn là chi phí của không gian lời giải như mọi phương pháp khác.

- Do đó việc biết đủ nhiều các **công cụ, kỹ thuật** và cần sự hỗ trợ của các công cụ, kỹ thuật đủ mạnh để **tối ưu tốc độ** cho thuật toán là một điều cần thiết.

- Một số công cụ, kỹ thuật thường gặp với python:
+ **Hàm sort()**
+ *Priority Queue*: thư viện **pqueue**
+ *Cây nhị phân, tìm kiếm nhị phân, tìm kiếm trên cây*: thư viện **bintree, bisect**
+ *Tính toán song song*: thư viện **numpy**
+ ...

3 Case-Study

*

- Với mỗi case-study các bạn chỉ cần tự mình trả lời các câu hỏi sau đây là được. Các câu hỏi trên được hỏi và trả lời sau khi các bạn đã giải các case-study trên thì càng tốt, sẽ giúp bạn tự nhìn nhận lại quá trình giải của mình và rút ra nhiều kinh nghiệm.

- Câu 1: Chỉ dùng **3-5 câu** hãy tóm gọn (abstraction) lại bài toán. (*Abstraction*)

- Câu 2: Đưa ra một vài **nhận xét** về **đặc trưng** của bài toán mà bạn nhìn ra được. (*Observation and Detection*)

- Câu 3: Điều gì khiến bạn nghĩ rằng phương pháp tham lam là một cách tiếp cận phù hợp với bài toán. (*Recognition*)

- Câu 4: Nếu phải thiết kế thuật toán và cài đặt code cho bài toán trên bạn nghĩ sẽ có các **thành phần** nào cần được thiết kế và chú trọng? Đối chiếu lại với code của chính mình đã làm xem? (*Components*)

- Câu 5: Xâu chuỗi các thành phần đó lại thành một **quy trình** (step-to-step) để tạo thành một thuật toán hoàn chỉnh? (*Algorithm*)

- Câu 6: Tự đánh giá **độ phức tạp** cho thuật toán mình vừa thiết kế? (*Complexity*)

- Câu 7: Hãy tưởng tượng rằng bạn là **leader** của một team, hãy nêu ra **cơ sở logic** và **lý giải** cách chọn tối ưu cục bộ của bạn là một cách chọn hoàn toàn chính xác để thuyết phục team làm theo thiết kế của bạn? (*Explain*)

- Câu 8: Suy nghĩ cách giải cho các **bài toán mở rộng: Nobita, Fairy Tail** đã được đề xuất để biết cách đưa một bài toán A' về bài toán A khi vấn đề bị dấu đi? (biến lạ thành quen)

- Câu 9: Sau khi đã giải một số lượng các bài toán tham lam, các bạn có nhận xét gì về **ưu nhược điểm** của phương pháp này? Điểm nổi bật và những điều cần lưu ý? (*pros and cons*)

–> Các câu hỏi trên có tính định hướng giúp các bạn tìm hiểu và giải quyết nhiều bài toán.

4 Application

*

- **Đồ thị - Cặp ghép**:

+ Phần lớn các thuật toán trên đồ thị đều là các thuật toán tham lam, có thể kể đến như: Dijkstra, Kruskal, Prim, Euler,..

- **Định thời - Cấp phát bộ nhớ**:

+ Nếu đã và đang học bộ môn Hệ điều hành thì các bạn có thể thấy phần lớn các giải thuật định thời đều có hàm chọn và được thiết kế theo giải thuật tham lam nhằm tối ưu về phía người dùng hoặc hệ thống.

+ Việc tối ưu cấp phát bộ nhớ trên máy tính có thể quay về việc giải một bài toán *cặp ghép*.

- **DNS - định tuyến router**:

+ Học đến tầng network trong môn Mạng các bạn sẽ thấy sự xuất hiện của các giải thuật tham lam trên đồ thị nhằm tối ưu thời gian phản hồi và lưu lượng trên đường truyền mạng.

- **Machine Learning**:

+ **Gradient descent**: Tự nhớ lại cách người ta đã nghĩ ra thuật toán Gradient Descent mà thầy Hoàng đã trình bày để thấy được rằng đó là một tư tưởng của thuật toán tham lam. Thay vì đi tìm cực trị cho hàm mục tiêu đã cho, người ta đi tìm cực trị cho một hàm bậc 2 tại mỗi bước lặp của thuật toán.

+ **Nén ảnh Huffman**: Đằng nào qua bộ môn xử lý ảnh các bạn cũng sẽ học thuật toán này thôi. Nó là một thuật toán tham lam khá nổi tiếng rồi.

+ **Clustering**: các thuật toán phân cụm ứng dụng tham lam rất nhiều, tìm hiểu các paper về ứng dụng tham lam trong phân cụm.

5 End

- Trên đây là toàn bộ tài liệu về tham lam mà chúng tôi biên soạn được
- Tài liệu có chứa nhiều thuật ngữ - khái niệm nên khá khó để đọc cũng như còn có nhiều thiếu sót, lủng củng nên rất mong nhận được sự thông cảm và góp ý từ các bạn.
- Xin chân thành cảm ơn.