ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH

TRÍ TUỆ NHÂN TẠO

Bài tập lớn 1

Xếp lịch thi đấu thể thao

GVHD: Vương Bá Thịnh
SV: Bùi Anh Nhật - 1612377

TP. HỒ CHÍ MINH, THÁNG 3/2019

Mục lục

Bài báo cáo này trình bày về việc áp dụng Trí tuệ nhân tạo (AI) cùng với các kỹ năng lập trình Python vào việc xếp lịch thi đấu thể thao một cách công bằng nhất.

1 Giới thiệu bài toán

Bài toán xếp lịch thi đấu thể thao có mục đích cuối cùng là tạo nên một lịch thi đấu công bằng nhất có thể phù hợp với các điều kiện sau:

1. Giải đấu có **n** vận động viên (vđv), mỗi vận động viên cần đấu với **k** vận động viên khác.
   - Nghĩa là tổng cộng cần có $(n * k)/2$ trận thi đấu. Do khi vận động viên A đấu với vận động viên B, thì ngược lại trong danh sách thi đấu của B cũng phải có A. Công việc đặt ra là phải giải quyết sao cho đảm bảo được số trận thi đấu cũng như số lượng đối thủ của mỗi người chơi. Vấn đề này sẽ được xử lý bằng **k-regular graph** được nêu ở phần sau.
2. Mỗi vận động viên có một điểm số trong bảng xếp hạng. Cần tối ưu hóa điểm số trung bình các đối thủ của 2 vđv bất kỳ không quá chênh lệch.
   - Với vấn đề này, cần tạo cấu trúc dữ liệu để lưu trữ thông tin cho từng vđv.
   - Cần một phương pháp tìm kiếm trạng thái tối ưu của bài toán. Có thể sử dụng thuật toán vét cạn (brute-force) để giải quyết, tuy nhiên vì thuật toán này cần tìm kiếm trong tất cả mọi khả năng của bài toán, nên với n vđv tham gia thi đấu, có tất cả **n!** trường hợp cần xét (tất cả hoán vị của n phần tử) trong điều kiện không có thêm kinh nghiệm nào khác. Chính vì vậy, để giảm thiểu chi phí cho bài toán, giải thuật **Hill-Climbing** (leo đồi) đã được áp dụng.
   - Cần xác định hàm lượng giá thích hợp để có thể tìm được trạng thái tốt hơn.
3. Sử dụng ngôn ngữ python 3 để lập trình
   - Áp dụng các kỹ năng lập trình và những đặc tính riêng của python để hỗ trợ giải quyết bài toán
4. Định dạng file input và output.
   - Cần đọc file để lưu dữ liệu thích hợp và ghi kết quả ra output. Điều này có thể dễ dàng thực hiện với python.

2 Phân tích bài toán

2.1 K-regular graph và điều kiện bài toán có nghiệm

Như đã trình bày ở trên, để giải quyết điều kiện "Giải đấu có **n** vận động viên (vđv), mỗi vận động viên cần đấu với **k** vận động viên khác.", ta sẽ sử dụng regular graph.

Regular graph là một dạng đồ thị mà trong đó mỗi đỉnh sẽ có số lượng đỉnh nối liền với nó (tạm gọi là hàng xóm) bằng nhau. Đây chính là điều chúng ta cần cho bài toán cần giải quyết.

K-regular graph sẽ có các luật sinh sau:

- Nếu $\mathbf{k = 2m}$ là số chẵn, một đỉnh sẽ có m hàng xóm gần nhất trải đều về mỗi phía.
- Nếu $\mathbf{k = 2m + 1}$ là số lẻ, và **n** là số chẵn, ngoài mỗi phía có m hàng xóm, thì cần nối với đỉnh đối diện của nó.

Dựa vào các điều kiện trên, dễ dàng nhận thấy **điều kiện để bài toán có nghiệm là** $n * k$ **phải là số chẵn.** Hiển nhiên, phải có $n > 1$ và $0 < k < n$

2.2 Đọc dữ liệu từ file input

Thông tin của mỗi vđv sẽ được lưu dưới dạng là một **tuple (score, number)** với score là điểm của vđv, number là số thứ tự của vđv đó. Các tuple này sẽ được lưu trong một list tên **value** để dễ dàng phục vụ cho các công việc sau này.

2.3 Không gian bài toán

Với các điều kiện đã nêu trong phần 1, ta có thể lên ý tưởng hiện thực bài toán dưới dạng **List** trong python.

Đầu tiên, list tổng quát (tạm gọi **player list**) sẽ chứa n list con, với n là số lượng vđv tham gia thi đấu, trong mỗi list đó sẽ chứa thông tin về đối thủ của một vđv tương ứng, nên tạm gọi là **opponent list**.

Ví dụ với n = 3 và k = 2, ta sẽ có dạng cơ bản của từng loại list:

- Player list: [opponent list 1, opponent list 2, opponent list 3]
- Opponent list: [(score1, number1), (score2, number2)]
- Tổng quát: [[(score2, number2), (score3, number3)], [(score3, number3), (score1, number1)], [(score1, number1), (score2, number2)]]
- Trong đó, list con đầu tiên [(score2, number2), (score3, number3)] là danh sách thông tin các đối thủ của vđv có số thứ tự number1.

Vậy, không gian bài toán có thể định nghĩa là **một player list** chứa **n opponent list**, mỗi **opponent list** chứa **k** tuple, thông tin về k đối thủ của player i tương ứng. Các điều kiện ràng buộc sẽ được nêu ở phần sau, khi nói về các bước chuyển trạng thái hợp lệ.

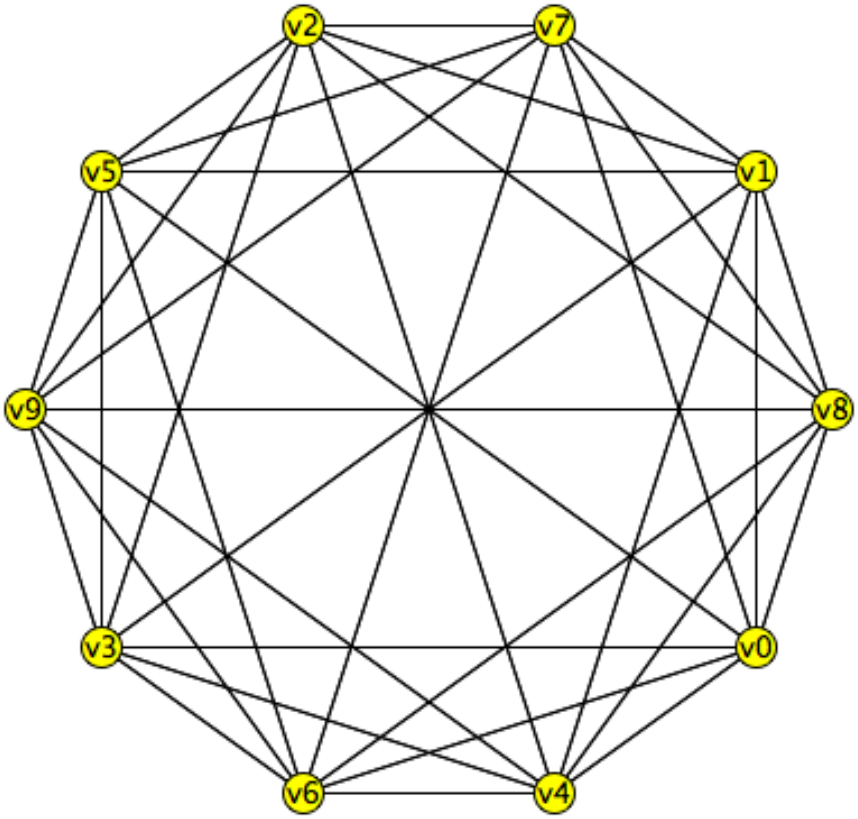
2.4 Trạng thái khởi đầu

Từ dữ liệu ban đầu có được từ phần 2.2, ta sử dụng list **value** để hiện thực hàm initialize(value, k) để sinh trạng thái khởi đầu cho bài toán.

Dựa vào cơ sở lý thuyết của **k-regular graph** và **Không gian bài toán** đã nêu, ta có thể xây dựng một **player list** gồm các **opponent list** của các vđv **i** là: (tạm coi i là một tuple chứa thông tin vđv thứ i để dễ biểu diễn)

- Opponent list i: [i+k, i+(k-1), ..., i+1, i-1, ..., i - k] với k chẵn
- Opponent list i: [i+k, i+(k-1), ..., i+1, i-1, ..., i - k, i+(n/2)] với k lẻ, i+(n/2) chính là đỉnh đối diện đỉnh i như trong k-regular graph.

Ở đây, n vđv được xem như một đồ thị được xếp như một vòng tròn, nên i = n+1 sẽ trở lại i = 1 và ngược lại.



Hình 1: Biểu diễn regular graph với n = 10, k = 7

Để hiểu rõ hơn về trạng thái khởi đầu, ta xét ví dụ với file input sau:

```
4 2
1
2
3
4
```

Dễ dàng nhận thấy n = 4, k = 2, list **value** sẽ được lưu dưới dạng:

```
[(1, 1), (2, 2), (3, 3), (4, 4)]
```

Sau khi thực thi hàm initialize(value, k), ta được trạng thái ban đầu như sau:

```
[[(2, 2), (4, 4)], [(3, 3), (1, 1)], [(4, 4), (2, 2)], [(1, 1), (3, 3)]]
```

Dễ dàng nhận thấy với opponent list của vđv 1 gồm vđv 2 và vđv 4, thỏa i+1 và i-1, tương tự các các vđv khác.

2.5 Trạng thái mục tiêu

Trạng thái mục tiêu lý tưởng nhất là khi điểm số trung bình các đối thủ của 2 vđv bất kỳ (trbinh-i và trbinh-j) **bằng nhau**. Tuy nhiên, trong thực tế, đối với hầu hết các trường hợp thì điều này là **không khả thi**, cho dù nếu khả thi hoặc muốn tiệm cận giá trị này nhất thì sẽ **tốn chi phí tính toán cực kỳ lớn** so với kết quả đem lại được.

Chính vì vậy, trong phạm vi bài báo cáo này, trạng thái mục tiêu chính là một trạng thái phù hợp không gian bài toán và các điều kiện ràng buộc, và thỏa điều kiện điểm số trung bình các đối thủ của 2 vđv bất kỳ **gần nhau nhất**.

Do tính chất hàm lượng giá của bài toán, nghĩa là sẽ swap các giá trị trong không gian bài toán để được một trạng thái tốt hơn, nên trạng thái bài toán sẽ được cải thiện tốt nhất có thể cho đến khi thỏa **điều kiện dừng**:

```
listSumScore[maxIndex] - listSumScore[minIndex] < (avgPlayerScore/2 + 1)
```

Trong đó:

- listSumScore[maxIndex]: Giá trị lớn nhất của tổng điểm các đối thủ của một vđv
- listSumScore[minIndex]: Giá trị nhỏ nhất của tổng điểm các đối thủ của một vđv
- avgPlayerScore: Điểm trung bình của tất cả các vđv

Lý do có điều kiện này là do chênh lệch giữa giá trị lớn nhất và giá trị nhỏ nhất của tổng điểm các đối thủ của vđv chính là do chênh lệch điểm số thứ hạng của các vđv thành phần. Vì vậy,, khi **listSumScore[maxIndex] - listSumScore[minIndex]** $<$ **(avgPlayerScore/2 + 1)** thì có nghĩa là chênh lệch đã đủ nhỏ, có thể coi đó là trạng thái mục tiêu.

Còn tìm được giá trị **avgPlayerScore/2** là do giữa 2 giá trị tổng điểm, cần phải swap 2 điểm số của 2 vđv. Mà điểm số trung bình thường thấy nhất của 1 vđv chính là avgPlayerScore, và khi lấy playerScore1 - playerScore2 thì ta sẽ còn được chênh lệch nhỏ hơn nữa.

Cũng từ cách sinh trạng thái mới trong phần 2.6, **nếu không có cặp giá trị max, min mới được chọn cũng có nghĩa là giải thuật kết thúc**, trạng thái trước đó chính là trạng thái tối ưu tìm được.

Vì các lý do trên cũng như từ thực nghiệm, ta đã xác định được giá trị tối ưu trong bài báo cáo.

2.6 Cách sinh trạng thái mới

Đầu tiên, giải thuật sinh trạng thái mới dựa vào hàm **maxMin(listSumScore, usedList)**, với listSumScore là danh sách tổng điểm các đối thủ của từng vđv, usedList là cặp max, min đã được chọn.

Cụ thể, hàm trên sẽ trả về cặp giá trị maxIndex, minIndex, tương ứng với số thứ tự của vđv có tổng điểm các đối thủ lớn nhất và nhỏ nhất, sao cho cặp này chưa bao giờ được chọn (để giải quyết vấn đề lặp vô hạn nếu do tiếp tục chọn lại cặp max, min đã xuất hiện trước đó).

Về cơ bản, giải thuật dựa vào hàm lượng giá sẽ chọn ra 1 opponent trong opponent list của vđv có tổng điểm các đối thủ lớn nhất (số thứ tự maxIndex) và 1 opponent trong opponent list của vđv có tổng điểm các đối thủ nhỏ nhất (số thứ tự minIndex). Hai **tuple** này sẽ được **hoán đổi** cho nhau, khiến cho tổng điểm các đối thủ của 2 vđv này tiến về trạng thái tối ưu hơn. Trong source code, điều này thể hiện bởi hoán đổi 2 giá trị:

```
listOpponent[minIndex][indexOpponentMin], listOpponent[maxIndex][indexOpponentMax]
```

Ngoài 2 vđv này ra, do tính chất của **k-regular graph** nên khi đổi chỗ 2 opponent cũng sẽ kéo theo việc thay đổi của 2 opponent của 2 vđv liên quan. Trong source code chính là:

```
listOpponent[affectedPlayerMax][indexBeChangedMax],
listOpponent[affectedPlayerMin][indexBeChangedMin]
```

Với cách sinh trạng thái mới này, mặc dù không thể phủ tất cả **n!** trường hợp như brute-force, nhưng cách sinh trạng thái này có thể bao phủ phần lớn các trường hợp trạng thái tốt hơn, nghĩa là đảm bảo đảm bảo trạng thái sinh ra sau dao động trong khoảng giá trị ngày càng tốt hơn trước, tiệm cận đến trạng thái tối ưu.

Tuy nhiên, chưa phủ được nghiệm tối ưu tuyệt đối, đặc biệt khi càng tiệm cận nghiệm tối ưu thì càng cần nhiều chi phí tính toán. Chính vì vậy, giải thuật chỉ tìm kiếm trạng thái gần tối ưu nhất có thể mà vẫn đảm bảo được chi phí tính toán hợp lý như đã nêu trong phần 2.5 về điều kiện dừng.

2.7 Các luật chuyển trạng thái hợp lệ

Một move hợp lệ là một trạng thái mới được sinh ra, nghĩa là chọn được cặp opponent để hoán đổi thỏa các điều kiện sau:

1. Số thứ tự của đối thủ được chọn để hoán đổi phải **khác** số thứ tự của vđv có đối thủ được hoán đổi. Ví dụ:

   ```
   1:  3 2 5
   2:  4 3 1
   3:  5 4 2
   4:  1 5 3
   5:  2 1 4
   ```

   Vđv 1 có đối thủ là 3, 2 và 5, vđv 2 có đối thủ là 4, 3 và 1. Như vậy, đối thủ 2 của vđv 1 không thể hoán vị cho đối thủ 1 của vđv 2 và ngược lại.

2. Số thứ tự của đối thủ được chọn để hoán đổi phải **khác** số thứ tự của các đối thủ đã có sẵn trong danh sách các đối thủ của vđv có đối thủ được hoán đổi. Ví dụ, vđv 1 không thể hoán đổi đối thủ 3 để lấy đối thủ 4 của vđv 2 thì vđv 2 đã có đối thủ 3.

3. Điểm của đối thủ được chọn trong danh sách min (opponentMin[0]) phải **nhỏ hơn** điểm của đối thủ chọn trong danh sách max (opponentMax[0]). Đảm bảo trạng thái mới tốt hơn.

4. Hiệu opponentMax[0] - opponentMin[0] phải nhỏ hơn chênh lệch giữa tổng điểm max và tổng điểm min.

5. Cả 4 đối thủ bị hoán đổi (2 được chọn từ max, min, 2 đối thủ bị ảnh hưởng) phải tạo nên một trạng thái tốt hơn.

2.8 Hàm lượng giá

2.8.1 Định nghĩa

Ta có:

- **scoreDiff = listSumScore[maxIndex] - listSumScore[minIndex]** là chênh lệch giữa tổng điểm max và tổng điểm min.
- **idealScore = (listSumScore[maxIndex] + listSumScore[minIndex])/2** là điểm tổng lý tưởng mong muốn đạt đến
- **newMaxScore = listSumScore[maxIndex] - opponentMax[0] + opponentMin[0]** là điểm tổng max mới (chưa chắc vẫn là max) nếu đổi 2 đối thủ
- **newMinScore = listSumScore[minIndex] - opponentMin[0] + opponentMax[0]** là điểm tổng min mới nếu đổi 2 đối thủ
- **point** là điểm chênh lệch giữa tổng điểm mới và cũ của cả 2 maxIndex và minIndex. Đây cũng là thành phần chính để đánh giá trạng thái tốt hay xấu.
- **bestDiff**, indexOpponentMax, indexOpponentMin: điểm chênh lệch tốt nhất cùng với 2 số thứ tự của vđv được chọn.
- Tương tự, 2 đối thủ bị ảnh hưởng cũng có các giá trị như trên. Gồm idealScoreAffected, newMaxScoreAffected, newMinScoreAffected, pointAffected
- **totalPoint = point + pointAffected**: tổng điểm của tất cả đối tượng bị ảnh hưởng. Quyết định có nên chọn trạng thái mới không.

Nếu **point** $>$ **0** nghĩa là so với trạng thái cũ, trạng thái mới tốt hơn (gần với giá trị lý tưởng, tuy nhiên chưa xét đến 2 đối tượng bị ảnh hưởng) hơn một số là point. Ngược lại, nếu point âm nghĩa là trạng thái mới xấu hơn, cần xét 2 đối thủ bị ảnh hưởng để xác định có nên đổi trạng thái mới không.

Nếu **totalPoint** $\leq$ **0** thì nghĩa là tổng điểm của các đối tượng bị ảnh hưởng cộng lại tạo nên một trạng thái tệ hơn.

Vòng lặp sẽ được sử dụng để tìm totalPoint tốt nhất, lưu vào **bestDiff**, cuối cùng ta sẽ tìm được đối thủ thích hợp nhất để đổi trạng thái tốt nhất có thể đối với cặp max-min hiện tại.

2.8.2 Lý do chọn hàm lượng giá

Với hàm lượng giá hiện tại, ta có thể dễ dàng nhận thấy trạng thái mới có tốt hơn trạng thái cũ không, tốt hơn cụ thể bao nhiêu (số điểm gần trạng thái tối ưu hơn trạng thái cũ), qua vòng lặp sẽ tìm được trạng thái tốt nhất cho cặp max-min hiện tại. Hơn nữa, nhờ hàm **maxMin** đã nêu ở phần 2.6, ta có thể tiếp tục chọn các cặp max-min tiếp theo đến khi đặt gần trạng thái tốt nhất mà không sợ bị lặp vô hạn do chọn phải các cặp đối thủ hoán đổi vị trí cho nhau tạo thành một vòng tròn.

Ngoài ra, hàm lượng giá còn giải quyết được vấn đề có 2 vđv khác có danh sách đối thủ bị ảnh hưởng khi chọn 2 đối thủ để hoán đổi từ max-min. Từ đó, xác định được 2 vđv bị ảnh hưởng là ảnh hưởng xấu hay tốt, kết hợp cả 4 thay đổi để chọn trạng thái tốt nhất.

3 Phân tích kết quả

3.1 Hiệu năng bài toán

- Với **n = 10,000** và **k = 10**, giải thuật có thể cho kết quả trong vòng 5 phút.
- Với **n = 20,000** và **k = 10**, giải thuật có thể cho kết quả trong vòng 30 phút.
- Với **n = 1,000** và **k = 100**, giải thuật có thể cho kết quả trong vòng 1 phút 30 giây.
- Với trường hợp nhỏ **n = 1,000** và **k = 10**, giải thuật có thể cho kết quả trong 1.6 giây.

Nếu muốn tăng tốc độ giải bài toán thì cần phải giảm yêu cầu tối ưu của giải thuật xuống. Cụ thể, nếu điều kiện dừng của giải thuật đổi thành **listSumScore[maxIndex] - listSumScore[minIndex]** $<$ **avgPlayerScore** thì tốc độ với **n = 10,000** và **k = 10** là 1.5 phút.

3.2 Vấn đề gặp khó khăn

3.2.1 Trường hợp lặp vô hạn

Trong quá trình thiết kế giải thuật, một số trạng thái được sinh ra làm cho giải thuật bị lặp vô hạn. Điều này xảy ra do khi tìm 2 giá trị đối thủ để hoán đổi cũng khiến cho 2 giá trị khác bị hoán đổi theo. Nếu gặp trường hợp mà 2 giá trị khác này khiến cho tổng giá trị các đối thủ của vđv của chúng cũng trở thành max-min mới thì sẽ dẫn đến lặp.

Để giải quyết việc này, một danh sách các max-min đã chọn được lưu lại. Tuy nhiên, cách này khiến cho giải thuật ngày càng chậm khi các cặp max-min được xét nhiều hơn. Độ phức tạp bị tăng lên n lần.

Vấn đề đặt ra là cần phát hiện ra giải thuật đã rơi vào vòng lặp để giải quyết nhanh chóng và không tốn nhiều chi phí tính toán.

3.2.2 Vấn đề nghiệm tối ưu

Theo lý thuyết, bằng việc bỏ đi điều kiện dừng **listSumScore[maxIndex]-listSumScore[minIndex]** $<$ **((avgPlayerScore/2) + 1)** thì giải thuật sẽ phủ qua tất cả các trường hợp max-min không trùng lặp hoặc đến khi max = min. Tuy nhiên, điều này chỉ khả thi đối với n nhỏ, tầm 100. Với n lớn hơn thì số trạng thái có thể được sinh ra quá lớn, nên cần giới hạn yêu cầu của bài toán, **thế nào là khoảng chênh lệch chấp nhận được**.

Giải thuật trong bài báo cáo này cần hi sinh độ tối ưu về điểm số chênh lệch để có thể giảm thiểu chi phí tính toán. Tùy vào yêu cầu của người dùng mà có thể chọn giữa nghiệm tối ưu hay có thể chạy được với n lớn hơn.

4 Kết luận

Nhìn chung, giải thuật đã giải quyết được yêu cầu cốt lõi của bài toán: xếp lịch thi đấu cho các vđv với điểm số trung bình các đối thủ của 2 vđv bất kỳ không quá chênh lệch.

Tuy nhiên, giải thuật còn nhiều hạn chế, chưa giải quyết triệt để được vấn đề tối ưu hóa cả về mặt kết quả và về mặt thời gian chạy.

Tài liệu

[1] Slide môn học

[2] How to construct a k-regular graph? "**link: https://math.stackexchange.com/questions/142112/how-to-construct-a-k-regular-graph**", lần truy cập cuối: 31/03/2019.