BTTH02: Hệ thống tìm kiếm

Môn: Trí tuệ nhân tạo

==========

Trần Trung Kiên
Email: ttkien@fit.hcmus.edu.vn

Ngày 18 tháng 3 năm 2017

1 Hướng dẫn tổng thể

Trong bài này, bạn sẽ cài đặt các thuật toán tìm kiếm để giúp Pacman đi trong mê cung của mình. Đầu tiên, bạn download khung chương trình của bài tập này ở đây. Sau khi giải nén ra thư mục `search`, bạn mở cmd, và thay đổi thư mục hiện tại trên cmd thành thư mục `search` bằng cách: gõ câu lệnh `cd <Đường dẫn đến thư mục search>` (nếu ổ đĩa hiện tại khác ổ đĩa bạn muốn đến thì bạn gõ `cd /d <Đường dẫn đến thư mục search>`). Kế đến, bạn có thể test thử chương trình với một số câu lệnh sau [mình có đính kèm file `commands.txt` (nên mở bằng Notepad++ hoặc Sublime); trong file này, đã ghi sẵn các câu lệnh để bạn có thể copy-paste cho nhanh]:

- `python pacman.py`: Câu lệnh này sẽ hiện lên game Pacman để bạn có thể chơi thử.
- `python pacman.py -layout testMaze -pacman GoWestAgent`: Câu lệnh này sẽ tự động chơi game Pacman với mê cung `testMaze` và với chiến lược là luôn đi về phía Tây. Với mê cung này, chiến lược luôn đi về phía Tây sẽ giúp Pacman đi đến đích; tuy nhiên, ta hãy thử với một mê cung khác:
  `python pacman.py -layout tinyMaze -pacman GoWestAgent`
  Với mê cung này, Pacman sẽ bị mắc kẹt ... (bạn có thể ấn `Ctrl+c` ở cửa sổ cmd để thoát khỏi game).

- Ngoài ra, bạn có thể gõ `python pacman.py -h` để xem qua về các option mà mình có thể truyền vào câu lệnh. Các option có thể được truyền vào dưới dạng "long" hoặc "short" (ví dụ: `--layout` hoặc `-l`).

Trong chương trình mà bạn vừa download về, sẽ có một số hàm chưa được viết hoàn chỉnh, và nhiệm vụ của bạn là viết hoàn chỉnh các hàm này. Đề bài sẽ có tất cả 8 câu ứng với 8 yêu cầu lập trình (đề bài sẽ được trình bày cụ thể ở mục 2). Với mỗi câu:

- Đầu tiên, bạn sẽ viết hoàn chỉnh hàm được yêu cầu (trong các file *.py, những chỗ mà bạn cần phải viết code sẽ được ghi chú bằng dòng `"*** YOUR CODE HERE ***"`). Để viết được hàm, có thể bạn sẽ phải đọc các phần code liên quan khác; chiến lược đọc ở đây là: đọc một cách ít nhất có thể, chỉ cần đọc đủ để có thể viết được hàm.

- Kế đến, bạn sẽ chạy các câu lệnh để có thể kiểm tra một cách trực quan phần cài đặt của mình trên game Pacman.

- Cuối cùng, bạn sẽ chạy file `autograder.py` để kiểm tra phần cài đặt của mình với các bộ test khác nhau.

Ghi chú thêm: nếu bạn sử dụng Notepad++ để viết code, bạn có thể chọn `View`, rồi `Function List`; chức năng này sẽ giúp bạn xem một cách tổng thể danh sách các hàm có trong file, cũng như là để di chuyển nhanh chóng đến các hàm.

Nên nhớ mục tiêu chính ở đây là *học, học một cách chân thật*. Bạn có thể thảo luận ý tưởng với bạn khác cũng như là tham khảo các tài liệu, nhưng *code và bài làm phải là của bạn, dựa trên sự hiểu của bạn. Nếu vi phạm thì sẽ bị 0 điểm cho toàn bộ môn học.*

2 Đề bài

2.1 DFS

Trong phần này, bạn sẽ viết hoàn chỉnh hàm `depthFirstSearch` trong file `search.py`.

- Hàm này có tham số đầu vào là bài toán tìm kiếm `problem`. Bạn có thể gọi `problem.getStartState()` để lấy trạng thái bắt đầu, `problem.isGoalState(state)` để kiểm tra trạng thái `state` có phải là trạng thái đích không, và

Đầu vào: trạng thái bắt đầu, hàm successor, hàm kiểm tra trạng thái đích

Đầu ra: kế hoạch tìm được (chuỗi các hành động để đi từ trạng thái bắt đầu đến trạng thái đích)

Quá trình thực hiện:

- Khởi tạo:
  - `fringe`: gồm một kế hoạch ứng với trạng thái bắt đầu
  - `closed set`: rỗng
- Trong khi `fringe` chưa rỗng:
  - Lấy một kế hoạch ra khỏi `fringe` theo một chiến lược nào đó
  - Nếu kế hoạch này đi tới đích (dùng hàm kiểm tra trạng thái đích): RETURN kế hoạch!
  - Nếu trạng thái cuối của kế hoạch này chưa có trong `closed set`:
    - Đưa trạng thái vào `closed set`
    - Mở rộng ra (dựa vào hàm successor) và đưa các kế hoạch mới vào `fringe`
- Nếu ra được khỏi vòng lặp nghĩa là: không tìm thấy lời giải

Hình 1: Khung sườn của thuật toán tìm kiếm (dùng closed set)

`problem.getSuccessors(state)` để mở trạng thái `state` (phương thức này sẽ trả về một danh sách, trong đó mỗi phần tử là một bộ `(successor, action, stepCost)` với `successor` là trạng thái mà có thể đi đến được từ trạng thái `state` hiện tại bằng cách thực hiện hành động `action`, và chi phí của việc chuyển trạng thái này là `stepCost`).

- Trong hàm này, bạn hãy sử dụng `Stack` đã được cung cấp sẵn cho bạn trong file `util.py`.
- Bạn có thể tham khảo khung sườn của thuật toán tìm kiếm nói chung ở hình 1.
- Hàm này sẽ trả về danh sách các hành động để có thể đi từ trạng thái bắt đầu đến trạng thái đích.

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan phần cài đặt của bạn trên game Pacman bằng cách gõ các câu lệnh sau:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=dfs
```

Câu lệnh thứ nhất ứng với mê cung `tinyMaze`: đầu tiên, `SearchAgent` sẽ sử dụng thuật toán DFS mà bạn vừa cài đặt để lên kế hoạch trong đầu; sau khi đã tìm ra được kế hoạch (chuỗi các hành động để đi từ trạng thái bắt đầu đến trạng thái đích), `SearchAgent` sẽ thực thi kế hoạch này. Khi chạy, bạn sẽ thấy các ô ở mê cung có màu đỏ với mức độ đậm khác nhau; mức độ đậm này cho biết thứ tự mở của các ô khi chạy thuật toán tìm kiếm: các ô có màu đỏ càng đậm thì được mở càng sớm. Các câu lệnh còn lại cũng tương tự nhưng mà làm với mê cung `mediumMaze` và `bigMaze`.

Cuối cùng, bạn gõ `python autograder.py -q q1` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau.

Khi đã cài đặt xong thuật toán DFS thì bạn có thể cài đặt các thuật toán BFS, UCS, A* ở các câu kế tiếp một cách tương tự và dễ dàng.

2.2 BFS

Trong phần này, bạn sẽ viết hoàn chỉnh hàm `breadthFirstSearch` trong file `search.py`.

- Hàm này có tham số đầu vào là `problem`. Bạn có thể gọi `problem.getStartState()` để lấy trạng thái bắt đầu, `problem.isGoalState(state)` để kiểm tra trạng thái `state` có phải là trạng thái đích không, và `problem.getSuccessors(state)` để mở trạng thái `state` (phương thức này sẽ trả về một danh sách, trong đó mỗi phần tử là một bộ `(successor, action, stepCost)` với `successor` là trạng thái mà có thể đi đến được từ trạng thái `state` hiện tại bằng cách thực hiện hành động `action`, và chi phí của việc chuyển trạng thái này là `stepCost`).
- Trong hàm này, bạn hãy sử dụng `Queue` đã được cung cấp sẵn cho bạn trong file `util.py`.
- Bạn có thể tham khảo khung sườn của thuật toán tìm kiếm nói chung ở hình 1.
- Hàm này sẽ trả về danh sách các hành động để có thể đi từ trạng thái bắt đầu đến trạng thái đích.

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan phần cài đặt của bạn trên game Pacman bằng cách gõ các câu lệnh sau:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs
```

Bạn cũng có thể kiểm tra hàm BFS của bạn với bài toán 8-puzzle bằng cách gõ `python eightpuzzle.py`

Cuối cùng, bạn gõ `python autograder.py -q q2` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau.

2.3 UCS

Trong phần này, bạn sẽ viết hoàn chỉnh hàm `uniformCostSearch` trong file `search.py`.

- Hàm này có tham số đầu vào là `problem`. Bạn có thể gọi `problem.getStartState()` để lấy trạng thái bắt đầu, `problem.isGoalState(state)` để kiểm tra trạng thái `state` có phải là trạng thái đích không, và `problem.getSuccessors(state)` để mở trạng thái `state` (phương thức này sẽ trả về một danh sách, trong đó mỗi phần tử là một bộ `(successor, action, stepCost)` với `successor` là trạng thái mà có thể đi đến được từ trạng thái `state` hiện tại bằng cách thực hiện hành động `action`, và chi phí của việc chuyển trạng thái này là `stepCost`).

- Trong hàm này, bạn hãy sử dụng `PriorityQueue` đã được cung cấp sẵn cho bạn trong file `util.py`.

- Bạn có thể tham khảo khung sườn của thuật toán tìm kiếm nói chung ở hình 1.

- Hàm này sẽ trả về danh sách các hành động để có thể đi từ trạng thái bắt đầu đến trạng thái đích.

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan phần cài đặt của bạn trên game Pacman bằng cách gõ câu lệnh sau:
`python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`
Với câu lệnh này, ta thấy rằng thuật toán UCS sẽ hoạt động giống như thuật toán BFS vì các cạnh trên đồ thị đều có chi phí là 1. Để thấy được khả năng của UCS với đồ thị có các cạnh có các chi phí khác nhau, bạn hãy thử hai câu lệnh sau:
`python pacman.py -l mediumDottedMaze -p StayEastSearchAgent`
`python pacman.py -l mediumScaryMaze -p StayWestSearchAgent`
Trong đó, với mê cung `mediumDottedMaze`, vùng có nhiều thức ăn sẽ có chi phí thấp; và với mê cung `mediumScaryMaze`, vùng có ma sẽ có chi phí cao.

Cuối cùng, bạn gõ `python autograder.py -q q3` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau.

2.4 A*

Trong phần này, bạn sẽ viết hoàn chỉnh hàm `aStarSearch` trong file `search.py`.

- Ngoài tham số đầu vào là `problem` giống như ở các hàm DFS, BFS, và UCS ở trên, hàm này có một tham số đầu vào nữa là hàm `heuristic`. Hàm `heuristic` này nhận đầu vào là trạng thái `state` và bài toán `problem`, và trả về chi phí ước lượng từ `state` đến đích.
- Trong hàm này, bạn hãy sử dụng `PriorityQueue` đã được cung cấp sẵn cho bạn trong file `util.py`.
- Bạn có thể tham khảo khung sườn của thuật toán tìm kiếm nói chung ở hình 1.
- Hàm này sẽ trả về danh sách các hành động để có thể đi từ trạng thái bắt đầu đến trạng thái đích.

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan phần cài đặt của bạn trên game Pacman bằng cách gõ câu lệnh sau:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

Với câu lệnh này, `SearchAgent` sẽ sử dụng thuật toán A* với heuristic là khoảng cách Manhattan (đã được cài đặt sẵn cho bạn trong file `searchAgents.py`). Nếu so sánh với thuật toán UCS thì bạn sẽ thấy số node mà thuật toán A* phải mở sẽ ít hơn.

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=ucs
```

Cuối cùng, bạn gõ `python autograder.py -q q4` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau.

2.5 Bài toán bốn góc mê cung

Để thấy rõ sức mạnh của A*, ta cần một bài toán tìm kiếm khó hơn. Trong phần này, bạn sẽ thiết kế một bài toán như vậy, và ở phần kế tiếp, bạn sẽ thiết kế heuristic cho bài toán này.

Cụ thể là bạn sẽ viết hoàn chỉnh `class CornersProblem` trong file `searchAgent.py` để thể hiện bài toán tìm kiếm sau: tìm đường đi ngắn nhất cho Pacman trong mê cung sao cho Pacman đi qua bốn góc của mê cung. Lưu ý chọn cách biểu

diễn trạng thái sao cho chỉ chứa đủ (chứ không dư) các thông tin để giải bài toán này. (Bạn có thể tham khảo cách viết của `class FoodSearchProblem` trong file `searchAgents.py` cho bài toán ăn hết tất cả các điểm thức ăn trong mê cung.)

Sau khi cài đặt xong, bạn có thể kiểm tra một cách trực quan trên game Pacman bằng cách gõ các câu lệnh sau:

```
python pacman.py -l tinyCorners -p SearchAgent -a "fn=bfs,prob=CornersProblem"
python pacman.py -l mediumCorners -p SearchAgent -a "fn=bfs,prob=CornersProblem"
```

Lưu ý là, để chạy được hai câu lệnh này, trước đó bạn cần phải hoàn thành xong hàm `breadthFirstSearch` ở mục 2.2. Với mê cung `mediumCorners`, bạn sẽ thấy BFS sẽ mở gần 2000 node; ở phần kế tiếp, bạn sẽ thiết kế heuristic để giúp A* giảm số node phải mở này lại.

Cuối cùng, bạn gõ `python autograder.py -q q5` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau.

2.6 Heuristic cho bài toán bốn góc mê cung

Trong phần này, bạn sẽ thiết kế heuristic cho bài toán bốn góc mê cung ở câu trước. Cụ thể là bạn sẽ viết hoàn chỉnh hàm `cornersHeuristic` trong file `searchAgent.py`. Hàm này nhận hai tham số đầu vào là trạng thái `state` và bài toán `problem`, và trả về chi phí ước lượng từ `state` đến đích. Lưu ý là, để A* tìm được đường đi tối ưu trong Graph Search, heuristic của bạn phải thỏa tính nhất quán (consistency). Thông thường, cách thiết kế một heuristic như sau. Đầu tiên, bạn làm dễ bài toán để có được một heuristic thỏa tính chấp nhận được (admissibility); chẳng hạn, Pacman giống như superman, có thể nhảy đến một ô bất kỳ chỉ trong một bước (để ý là, nếu bạn càng làm dễ bài toán so với bài toán gốc ban đầu thì heuristic sẽ càng dễ tính, nhưng tác dụng định hướng sẽ càng giảm đi; ta nên thiết kế một heuristic trung dung, vừa không quá chính xác để có thể tính được nhanh và vừa không quá không chính xác để có thể giúp định hướng tìm kiếm). Và heuristic chấp nhận được này thường sẽ nhất quán (lưu ý là, tính nhất quán sẽ kéo theo tính chấp nhận được, nhưng chiều ngược lại thì không chắc; ở đây, khi bạn thiết kế heuristic bằng cách làm dễ bài toán, chiều ngược lại thường sẽ xảy ra).

Ngoài ra, để ý là heuristic của bạn cần trả về giá trị không âm, và trả về giá trị 0 với trạng thái đích.

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan trên game Pacman bằng cách gõ câu lệnh:

`python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5`
với `AStarCornersAgent` là "shortcut" của:
`-p SearchAgent -a "fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic"`
Lưu ý là, để chạy được câu lệnh này, trước đó bạn cần phải hoàn thành xong hàm `aStarSearch` ở mục 2.4.

Cuối cùng, bạn gõ `python autograder.py -q q6` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau. Tùy vào chất lượng của heuristic của bạn mà autograder sẽ cho bạn các điểm số khác nhau. Đầu tiên, để có điểm thì heuristic của bạn phải thỏa tính nhất quán (và không tầm thường - non-trivial, nghĩa là không trả về 0 với mọi trạng thái). Kế đến, tùy vào số node mà A* phải mở (phụ thuộc vào chất lượng của heuristic) mà bạn sẽ nhận được các điểm số khác nhau:

- Nếu số node mở > 2000 thì bạn sẽ được 0/3 điểm.
- Nếu $1600 <$ số node mở ≤ 2000 thì bạn sẽ được 1/3 điểm.
- Nếu $1200 <$ số node mở ≤ 1600 thì bạn sẽ được 2/3 điểm.
- Nếu số node mở ≤ 1200 thì bạn sẽ được 3/3 điểm.

2.7 Heuristic cho bài toán ăn hết thức ăn

Bây giờ, ta sẽ thiết kế heuristic cho một bài toán khó hơn: tìm đường đi ngắn nhất cho Pacman trong mê cung sao cho Pacman ăn hết tất cả các điểm thức ăn (ở đây, giả sử không có ma). Bài toán này đã được cài đặt sẵn cho bạn ở `class FoodSearchProblem` trong file `searchAgents.py`. Nhiệm vụ của bạn là viết hoàn chỉnh hàm `foodHeuristic` trong file `searchAgents.py`.

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan trên game Pacman bằng cách gõ câu lệnh:
`python pacman.py -l trickySearch -p AStarFoodSearchAgent`
với `AStarFoodSearchAgent` là "shortcut" của:
`-p SearchAgent -a "fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic"`
Lưu ý là, để chạy được câu lệnh này, trước đó bạn cần phải hoàn thành xong hàm `aStarSearch` ở mục 2.4. Với thuật toán UCS (hay A* với heuristic luôn bằng 0), số node phải mở sẽ vào khoảng hơn 16000.

Cuối cùng, bạn gõ `python autograder.py -q q7` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau. Tùy vào chất lượng của heuristic của bạn mà autograder sẽ cho bạn các điểm số khác nhau. Đầu tiên, để có

điểm thì heuristic của bạn phải thỏa tính nhất quán (và không tầm thường - non-trivial, nghĩa là không trả về 0 với mọi trạng thái). Kế đến, tùy vào số node mà A* phải mở (phụ thuộc vào chất lượng của heuristic) mà bạn sẽ nhận được các điểm số khác nhau:

- Nếu số node mở > 15000 thì bạn sẽ được 1/4 điểm.
- Nếu $12000 <$ số node mở ≤ 15000 thì bạn sẽ được 2/4 điểm.
- Nếu $9000 <$ số node mở ≤ 12000 thì bạn sẽ được 3/4 điểm.
- Nếu $7000 <$ số node mở ≤ 9000 thì bạn sẽ được 4/4 điểm.
- Nếu số node mở ≤ 7000 thì bạn sẽ được 5/4 điểm.

Ở đây, mặc dù autograder đánh giá chất lượng heuristic chỉ thông qua số node phải mở (và giới hạn thời gian chạy trong một khoảng thời gian tối đa cho phép), nhưng bạn hãy để ý thêm đến thời gian chạy để xem thử heuristic của bạn có thật sự hiệu quả hay chưa?

2.8 Tìm kiếm nhanh

Như bạn có thể cảm thấy, với bài toán ăn thức ăn ở trên, khi gặp một mê cung lớn, thuật toán A* với một heuristic tốt vẫn có thể sẽ phải tốn nhiều chi phí để tìm được đường đi tối ưu. Do đó, ta có thể sẽ không muốn tìm một đường đi tối ưu, mà chỉ cần tìm một đường đi hợp lý và tìm nhanh. Trong phần này, bạn sẽ viết một agent để giải quyết nhanh bài toán ăn thức ăn theo chiến lược: ăn điểm thức ăn gần nhất (bạn hình dung như sau: từ ô hiện tại, agent sẽ thực hiện tìm kiếm điểm thức ăn gần nhất, rồi thực đi các hành động để đi đến và ăn điểm thức ăn này; tại ô mới này, agent lại thực hiện tìm kiếm điểm thức ăn gần nhất và thực thi các hành động ...; cứ thế cho đến khi agent ăn hết các điểm thức ăn). `ClosestDotSearchAgent` đã được cài đặt sẵn cho bạn trong file `searchAgents.py` nhưng hàm chính `findPathToClosestDot` chưa được viết hoàn chỉnh; nhiệm vụ của bạn là viết hoàn chỉnh hàm này.

Cách nhanh nhất để viết hoàn chỉnh hàm `findPathToClosestDot` là viết hoàn chỉnh hàm `isGoalState` trong `class AnyFoodSearchProblem`, rồi sau đó trong hàm `findPathToClosestDot`, gọi một hàm tìm kiếm thích hợp với bài toán truyền vào thuộc `class AnyFoodSearchProblem`. Sẽ có tổng cộng 2 dòng code :-).

Sau khi đã cài đặt xong, bạn có thể kiểm tra một cách trực quan trên game Pacman bằng cách gõ câu lệnh:
`python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5`

Cuối cùng, bạn gõ `python autograder.py -q q8` để kiểm tra phần cài đặt của bạn với các bộ test khác nhau.

3 Nộp bài

Trong thư mục `<MSSV>`, bạn đặt hai file `search.py` và `searchAgents.py` rồi nén thư mục này lại và nộp ở link trên moodle.

Enjoy searching :-)