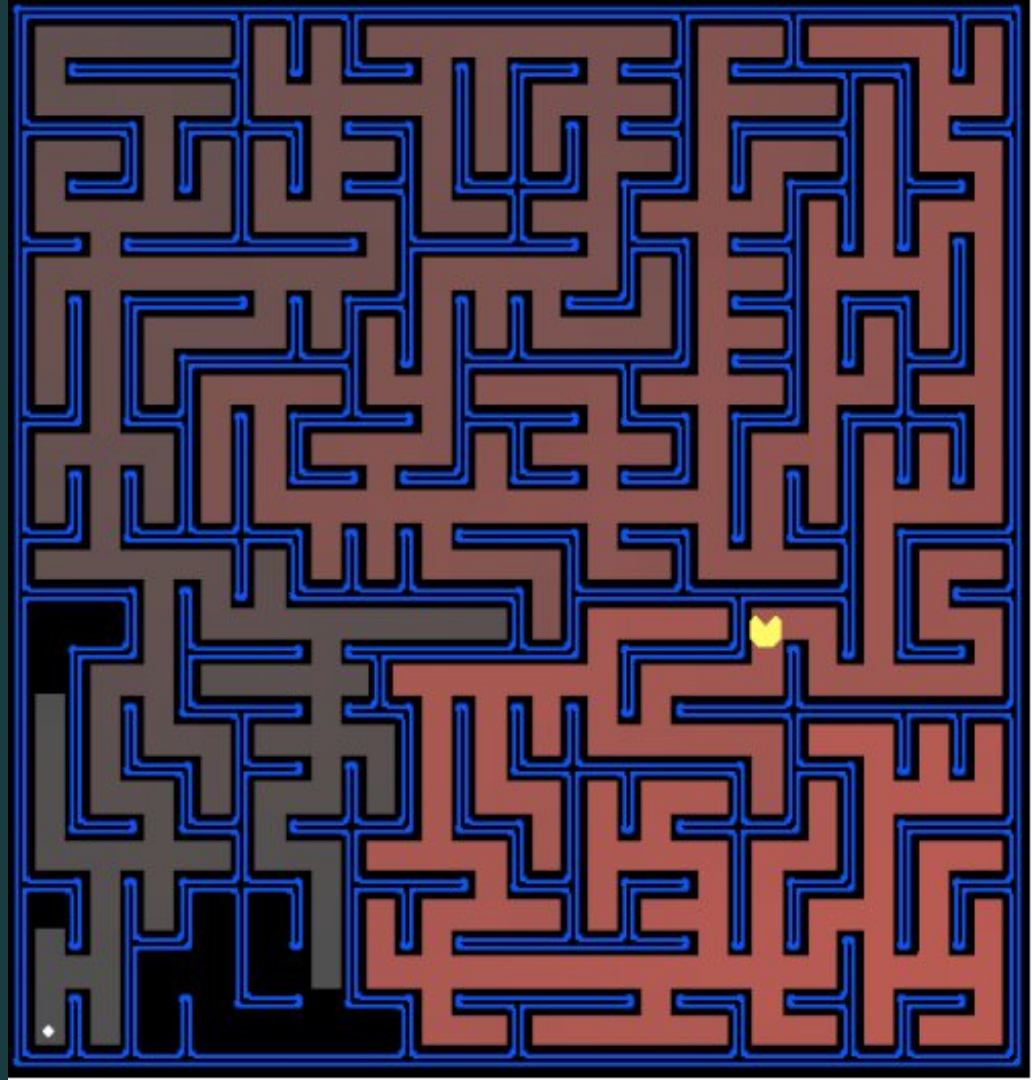
Trường Đại học Bách Khoa - Đại học Đà Nẵng
Khoa Công nghệ Thông tin

Search in Pacman

Môn học: Trí tuệ nhân tạo



Nội dung

Giới thiệu bài toán

Bài toán lập trình nhân vật Pacman để tìm đường đi trong mê cung với 2 nhiệm vụ cụ thể:

- Đến một vị trí chỉ định
- Thu thập thức ăn một cách hiệu quả nhất

Mục tiêu: Xây dựng một thuật toán tìm kiếm tổng quát và áp dụng vào các tình huống cho nhân vật Pacman.

Mã nguồn gồm nhiều tệp Python

- Một số tệp cần đọc/ hiểu để hoàn thành bài tập
- Một số tệp có thể bỏ qua

Tất cả mã nguồn và tài liệu đi kèm được cung cấp trong tệp này zip archive.

Các tệp cần chỉnh sửa

- `search.py`: File cài đặt **tất cả thuật toán tìm kiếm.**
- `searchAgents.py`: Nơi cài đặt **tất cả tác nhân dựa trên tìm kiếm**.

Các tệp nên tham khảo

- `pacman.py` : Tệp chính để chạy game Pacman. Chứa mô tả Pacman GameState được sử dụng trong bài.
- `game.py`: Chứa logic hoạt động của thế giới Pacman, định nghĩa các lớp hỗ trợ như Agent State, Agent, Direction, Grid.
- `util.py` : Chứa các cấu trúc dữ liệu hỗ trợ cho việc cài đặt thuật toán tìm kiếm.

Các tệp hỗ trợ có thể bỏ qua

- `graphicsDisplay.py, graphicsUtils.py:` Xử lý đồ họa cho Pacman.
- `textDisplay.py:` Hiển thị ASCII cho Pacman.
- `ghostAgents.py:` Điều khiển tác nhân ma.
- `keyboardAgents.py:` Giao diện điều khiển Pacman bằng bàn phím.
- `layout.py:` Đọc và lưu trữ thông tin layout mê cung.

Chạy thử Pacman

- Tải file search.zip -> giải nén -> vào thư mục **search**
- Chạy game với câu lệnh: ***python pacman.py***
- Trong file `searchAgent.py`, Agent đơn giản nhất được gọi là **GoWestAgent** bởi vì Agent này luôn đi về hướng Tây. Thỉnh thoảng, Agent này vẫn có thể thắng
  - Thử với layout dễ, chạy lệnh: `python pacman.py --layout testMaze --pacman GoWestAgent`
  - Thử với layout khó hơn: `python pacman.py --layout tinyMaze --pacman GoWestAgent`
- Nếu muốn thoát game, nhấn tổ hợp phím Ctrl + C
- Nếu muốn xem tùy chọn của game, gõ lệnh: `python pacman.py -h`
- File **commands.txt** có chứa lệnh mẫu; trên UNIX/Mac có thể chạy toàn bộ bằng lệnh: `bash commands.txt`

Thuật toán tìm kiếm cơ bản
DFS và BFS

- **Mục tiêu của DFS và BFS trong Pacman:**
  - **Bối cảnh:** Trong game, nhân vật sẽ phải tìm đường đi từ vị trí ban đầu tới đích (goal)
  - **Tác dụng của 2 thuật toán:**
    - Xác định đường đi từ vị trí Start -> Goal trong mê cung
    - **DFS:**
      - Khám phá sâu trước, ưu tiên đi tới cuối một nhánh rồi quay lại.
      - Dùng để khảo sát toàn bộ mê cung hoặc khi không quan tâm tối ưu độ dài đường đi.
    - **BFS:**
      - Khám phá rộng trước, đi qua các vị trí gần Start trước.
      - Đảm bảo tìm được đường đi ngắn nhất
  - 2 thuật toán cơ bản này là vô cùng quan trọng trong việc lập trình đường đi cho nhân vật trong game vì ảnh hưởng đến thời gian tìm và độ dài đường đi.

- **SearchAgent và yêu cầu dành cho bài tập**
  - File searchAgent.py có chứa SearchAgent:
    - Lập kế hoạch đường đi qua thế giới Pacman
    - Thực thi kế hoạch theo từng bước
  - Thuật toán tìm kiếm (DFS, BFS, UCS, A*) **chưa được cài đặt → nhiệm vụ của sinh viên**.
  - Lệnh kiểm tra SearchAgent:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

- **Yêu cầu chung khi cài đặt thuật toán:**
  - Trả về **danh sách hành động hợp lệ** dẫn Agent từ **start** → **goal**.
    - Mỗi hành động tương ứng với một hướng di chuyển hợp lệ trong mê cung (North, South, East, West).
  - Cần duy trì một tập hợp các trạng thái đã được mở rộng (visited set) để tránh mở rộng lại các trạng thái này.
    - **Việc tránh lặp lại giúp:**
      - Tiết kiệm thời gian (giảm số lần duyệt trạng thái trùng lặp).
      - Ngăn vòng lặp vô hạn trong mê cung có đường vòng.
  - Quản lý fringe (biên tìm kiếm)
    - Fringe là tập các nút (node) đang chờ được mở rộng.
    - Điểm khác biệt chính giữa các thuật toán là cách quản lý và chọn node tiếp theo từ fringe:
      - **DFS**: Sử dụng **Stack** → mở rộng theo **chiều sâu**, ưu tiên node mới thêm **gần nhất**.
      - **BFS**: Sử dụng **Queue** → mở rộng theo **chiều rộng**, ưu tiên node được thêm **sớm nhất**.
      - **UCS**: Sử dụng **PriorityQueue** với độ ưu tiên **theo chi phí đường đi**.
      - **A***: Sử dụng **PriorityQueue** với độ ưu tiên = **chi phí g + hàm heuristic h**.

- **Cấu trúc dữ liệu hỗ trợ (định nghĩa sẵn trong util.py):**
  - Stack → cho DFS.
  - Queue → cho BFS.
  - PriorityQueue → cho UCS và A*.

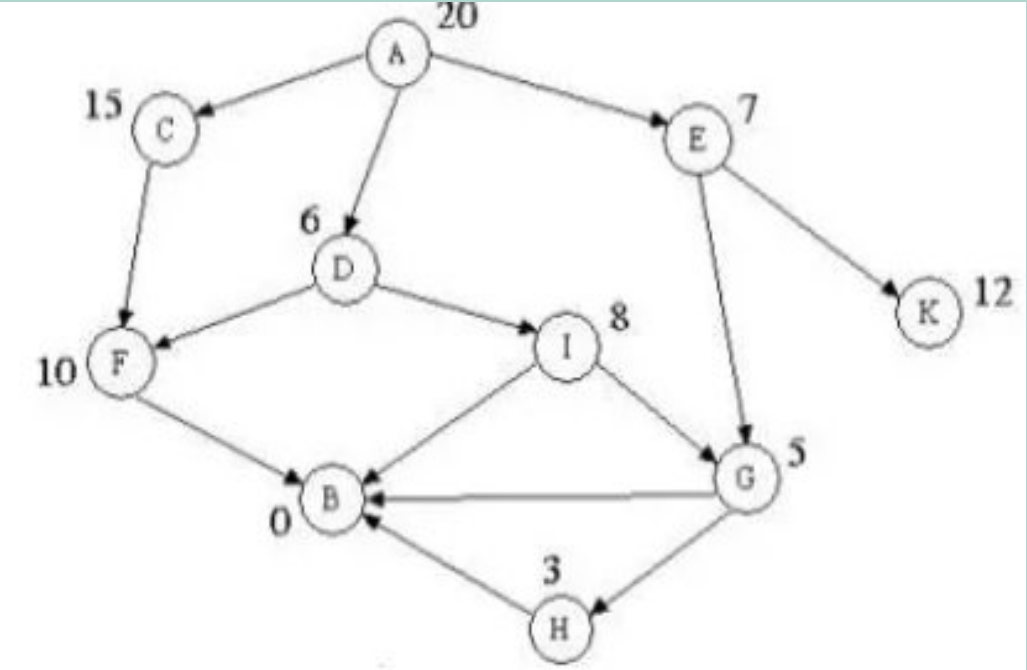
- **Câu 1 – DFS (10 điểm):**
  - Cài đặt trong **depthFirstSearch – search.py.**
  - Dùng **graph search** để tránh lặp trạng thái.
  - Lệnh chạy thử:
    - python pacman.py -l tinyMaze -p SearchAgent
    - python pacman.py -l mediumMaze -p SearchAgent
    - python pacman.py -l bigMaze -z .5 -p SearchAgent

- **Gợi ý**
  - Dùng Stack.
  - Với **mediumMaze**, độ dài lời giải chuẩn: **130 bước** (push theo thứ tự **getSuccessors**).
  - DFS **không** đảm bảo đường đi ngắn nhất.

- **Câu 2 – BFS (10 điểm):**
  - Cài đặt trong **breadthFirstSearch – search.py.**
  - BFS dùng để tìm đường đi ngắn nhất.
  - Lệnh chạy thử:
    - python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
    - python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

- **Gợi ý**
  - Dùng Queue.
  - Mặc định Agent Pacman sẽ di chuyển chậm để dễ quan sát với **--frameTime > 0.** Nếu Agent Pacman chạy nhanh nhất có thể theo tốc độ xử lý của máy tính thì ta có điều chỉnh là **--frameTime 0**

Tìm kiếm theo chi phí đồng nhất



Uniform Cost Search UCS

- **Mục tiêu của UCS trong Pacman:**
  - Tìm đường từ start → goal sao cho tổng chi phí nhỏ nhất (chi phí trên các ô có thể khác nhau).
  - **Nguyên tắc:**
    - Dùng PriorityQueue; priority = cost_so_far (g(n)).
    - Luôn mở rộng node có chi phí tích lũy nhỏ nhất.
    - Lưu và cập nhật chi phí tốt nhất đã thấy cho mỗi trạng thái.
  - Lệnh chạy (kiểm tra):
    - python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
    - python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
    - python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
      - Hai agent ***StayEastSearchAgent*** và ***StayWestSearchAgent*** dùng hàm chi phí khác nhau
      - Chi phí đường đi sẽ rất thấp cho ***StayEastSearchAgent*** và rất cao cho ***StayWestSearchAgent*** do hàm chi phí có dạng **lũy thừa.**

- **Một số lưu ý khi cài đặt**
  - Fringe chứa (state, path, cost); priority = cost.
  - Dùng util.PriorityQueue.
  - Dùng bảng (dict) lưu best_cost[state] để:
    - Tránh mở rộng lại trạng thái với chi phí cao hơn đã thấy.
    - Cập nhật fringe nếu tìm được đường rẻ hơn tới cùng state.
  - UCS đảm bảo tối ưu nếu mọi chi phí ≥ 0.

Thuật toán tìm kiếm A*

- **Mục tiêu của A-Star trong Pacman:**
  - Tìm đường từ start → goal với chi phí thấp nhất, sử dụng cả thông tin chi phí đường đi và heuristic để dẫn hướng tìm kiếm.
  - Nhanh hơn UCS khi heuristic tốt, vì mở rộng ít node hơn.
- **Ý tưởng chính:**
  - **Công thức:** A* = UCS + Heuristic
  - **f(n) = g(n) + h(n)**
    - **g(n)**: chi phí từ start → n
    - **h(n)**: ước lượng chi phí từ n → goal (heuristic)
  - Chọn node trong fringe có giá trị **f(n) nhỏ nhất** để mở rộng.
- **Một số lưu ý khi cài đặt:**
  - Trả về danh sách hành động hợp lệ dẫn Pacman từ start → goal.
  - Không mở rộng lại trạng thái đã duyệt.
  - Dùng PriorityQueue trong util.py (ưu tiên theo f(n)).
  - Heuristic mặc định: nullHeuristic (h = 0) → A* trở thành UCS.
  - Có thể thay heuristic khác để tìm đường nhanh hơn.
- **Câu lệnh chạy thử:** `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`
  - **bigMaze:** bản đồ lớn để thấy khác biệt tốc độ so với UCS.
  - **-z .5**: giảm thời gian frame để Pacman chạy nhanh.
  - **heuristic=manhattanHeuristic**: heuristic Manhattan (tính khoảng cách ô vuông).

Bài toán 4 góc và thiết kế Heuristic

Câu 5 – CornersProblem:

- **Mục tiêu:**
  - Tìm đường ngắn nhất qua mê cung, đi qua cả 4 góc (dù có hoặc không có food ở đó)

- **Yêu cầu cài đặt:**
  - Viết class **CornersProblem** trong **searchAgents.py.**
  - **State representation** phải chứa đủ thông tin để biết đã đến những góc nào.
  - Không dùng nguyên GameState làm state (vì sẽ rất chậm và sai).
  - Chỉ cần:
    - Vị trí ban đầu của Pacman.
    - Tọa độ 4 góc mê cung.
  - BFS có thể giải được:
    - python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
    - python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

- **Nếu hiệu quả: BFS chuẩn sẽ mở được gần 2000 node ở mediumCorners.**

Câu 6 – Heuristic cho CornersProblem:

- **Mục tiêu:**
  Giảm số node phải mở bằng A* + heuristic phù hợp.

- **Yêu cầu heuristic (cornersHeuristic):**
  - **Non-trivial:** khác 0, không tính chi phí thật.
  - **Admissible:** Luôn ≤ chi phí thật.
  - **Consistent:** $h(n) \leq c(n, n') + h(n')$.
  - Không âm, và **h(goal) = 0.**

- **Cách test:**
  - Chạy lệnh

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

- **Đánh giá hiệu quả nếu :**
  - BFS chuẩn sẽ mở được gần 2000 node.
  - Heuristic tốt: <1600 node, giỏi: <800 node.

- **Gợi ý heuristic khả thi:**
  - Tính khoảng cách **Manhattan** từ vị trí hiện tại đến **góc chưa thăm gần nhất**.
  - Cộng thêm khoảng cách giữa các góc chưa thăm theo thứ tự tối ưu (MST hoặc greedy).

Admissibility và Consistency

- **Admissible Heuristic**
  - **Định nghĩa:** Heuristic h(n) là ***admissible*** nếu nó **không bao giờ đánh giá quá thấp** chi phí thực tế từ trạng thái n đến vị trí đích (và luôn ≥ 0).
  - **Ý nghĩa:** Đảm bảo **A*** không bỏ qua đường đi tối ưu vì đánh giá sai quá thấp.
  - **Ví dụ:** Nếu khoảng cách thực tế đến goal là 10, thì h(n) có thể là 0, 5, 9 nhưng **không được** là 11.
  - **Yêu cầu:** Ở trạng thái goal, **h (goal) = 0**

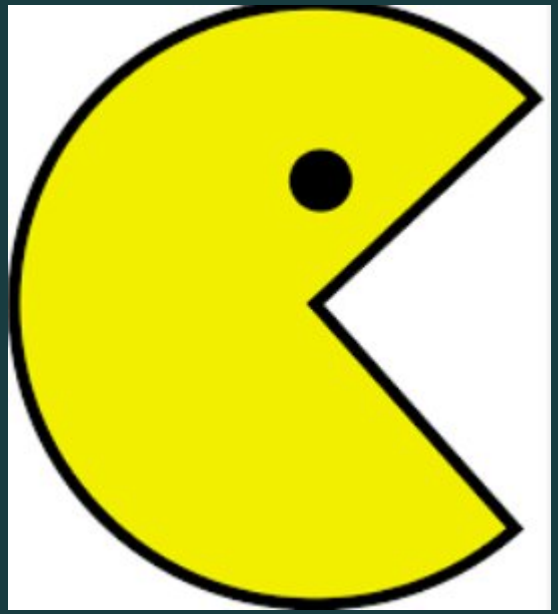
- **Consistency Heuristic**
  - **Định nghĩa:** Heuristic là consistent nếu luôn thỏa mãn bất đẳng thức tam giác:
    - $h(n) \leq c(n,a,n') + h(n')$
    - Tức là khi đi từ n sang n′ với chi phí c, giá trị heuristic giảm nhiều nhất là c.
  - **Ý nghĩa:** Đảm bảo f-value $f(n)=g(n)+h(n)$ $f(n) = g(n) + h(n)$ không giảm khi duyệt các trạng thái, giúp graph search không mở rộng lại node đã duyệt.
  - **Ví dụ:** Nếu di chuyển một bước tốn 3, thì heuristic từ node sau nhiều nhất giảm 3.

Admissibility và Consistency

- **Quan hệ giữa admissibility và consistency**
  - Trong tree search, chỉ cần admissibility là đủ để đảm bảo A* tìm được đường đi tối ưu.
  - Trong graph search, admissibility chưa đủ, phải cần consistency để tránh mở rộng lại các node và đảm bảo tính đúng đắn.
  - Hầu hết heuristic admissible trong thực tế thường cũng consistent, đặc biệt nếu lấy từ problem relaxation (nới lỏng ràng buộc bài toán).

- **Kiểm tra heuristic có consistent không:** Nếu UCS và A* cho **kết quả đường đi khác nhau** → heuristic của bạn **không** consistent.

- **Lưu ý:**
  - Khi code A* trong Pacman, phải viết heuristic vừa admissible vừa consistent để chạy đúng với graph search, tránh trivial heuristic, và tuân thủ điều kiện giá trị không âm, =0 ở goal.



Các bài tập lý thuyết và nâng cao

- **Câu 7: Tạo một ví dụ không gian tìm kiếm để minh họa việc Iterative Deepening Search (IDS) có thể trả về lời giải không tối ưu.**
  - **Giải thích:** IDS ưu tiên độ sâu nhỏ hơn nhưng không đảm bảo chi phí thấp nhất → nếu chi phí đường đi không đồng nhất, có thể lấy phải đường đi đắt hơn.

- **Câu 8: Mô tả một thuật toán hiệu quả để quyết định xem một heuristic nhất định có phù hợp không**
  - **Input:** Một heuristic h(n) và đồ thị trạng thái.
  - **Ý tưởng:**
    - Với mọi cạnh (n, n'), kiểm tra điều kiện: **$h(n) \leq c(n,n') + h(n')$**
    - Nếu vi phạm ở bất kỳ cạnh nào → **heuristic không consistent.**
  - Độ phức tạp: O(E) với E là số cạnh.

- **Câu 9: Tạo cây tìm kiếm nhỏ nhất có thể, minh họa việc h(n) > h*(n) ở một node → A* chọn nhầm đường đi không tối ưu.**
  - **Minh họa**
    - S → A (1), S → G (2), A → G (1)
    - h(A) = 3, h(G) = 0
    - f(S→A→G) = 1+1+3 = 5
    - f(S→G) = 2+0 = 2 → A* chọn S→G dù chi phí thực bằng nhau

- **Câu 10: Nghịch lý chi phí cạnh âm: Không gian trạng thái hữu hạn, nghiệm tối ưu không bao giờ đạt đến đích.**
  - **Ý tưởng:** Có vòng lặp chi phí âm → agent liên tục quay vòng để giảm chi phí tổng → không bao giờ dừng ở goal.
  - Ví dụ:
    - S --1--> A --(-2)--> S
    - S --5--> G
    - Lặp S→A→S mãi để giảm chi phí trước khi đi S→G.
- **Câu 11: Bài toán ký hiệu: Dãy ký hiệu có độ dài ≤ 5, bảng chữ cái có hàng ngàn ký tự.**
  - **Mô hình:**
    - State: Chuỗi ký tự đã chọn.
    - Action: Thêm 1 ký tự mới từ alphabet.
    - Goal: Chuỗi đúng theo yêu cầu.
  - **Chọn BFS hay DFS:**
    - DFS tốn ít bộ nhớ hơn, nhưng BFS đảm bảo shortest path.
    - Có thể cải tiến bằng heuristic pruning hoặc beam search.
- **Câu 12: Chỉnh sửa tìm kiếm lặp sâu dần để tìm đường đi có chi phí thấp nhất.**
  - **Ý tưởng:** Thay vì giới hạn độ sâu thì giới hạn theo ngưỡng chi phí và tăng dần ngưỡng này qua mỗi vòng lặp. Mỗi nút khi mở rộng cần lưu tổng chi phí từ gốc và chỉ tiếp tục nếu chưa vượt quá ngưỡng. Quá trình lặp kết thúc khi tìm được nút đích với chi phí nhỏ nhất.

Thank you

Conclude the presentation by thanking your audience for listening and participating.