Thuyết trình AI

Backtracking

Ý tưởng: xây dựng trên nền tảng DFS. Nếu có những biến có domain lớn hơn 1 giá trị, thì ta chọn một giá trị để thử, đưa vào sau đó gán giá trị cho các ô khác, nếu chúng ta gán được cho tất cả các ô thì đã tìm được solution , nhưng nếu chúng ta gán sau 1 số bước thì gặp một ô có domain là rỗng thì không tìm được solution, chứng tỏ giá trị chúng ta chọn ban đầu đã sai, chúng ta loại bỏ giá trị đó khỏi domain của ô đó.

Mô tả thuật toán:

- Đầu vào là một csp, đầu ra là một solution hoặc failure. Thuật toán này bản mang chất là DFS nên sẽ làm thuật toán complete, chỉ khi nào nhánh đi sâu vô tận thì mới không có solution. Còn nếu trả về failure thì bài toán có điều kiện quá khó nên không thể giải được.
- Cài đặt theo kiểu đệ quy h àm Backtrack: nhận vào assignment, csp (đối với bài toán sodoku thì assignment ban đầu là các ô đã có giá trị), (nếu chưa có gì thì đưa tập rỗng vào assignment)
- Nếu assignment đã hoàn thành, có nghĩa là tất cả các biến đều được gán bằng một giá trị duy nhất, thì assignment đó cũng chính là solution, ta return assignment.
- Nếu assignment chưa hoàn thành thì ta chọn ra một biến chưa được gán giá trị.
- Gán cho biến vừa chọn ra một giá trị từ tập domain của nó. Thực hiện vòng lặp:
    - Nếu giá trị vừa gán cho biến đó consistent với assignment hiện tại thì ta qua
    - Đưa giá trị đó cho biến và thêm nó vào assignment.
    - Kiểm tra xem tập domain của các biến xung quanh có giảm hay không.
    - Nếu inferences không gặp biến nào có tập domain là rỗng thì qua
    - Thêm inferences vào assignment nếu nó có tập domain chỉ có 1 giá trị
    - Gọi đệ quy, lấy biến khác và tiếp tục gán giá trị cho nó.
    - Nếu result khác failure thì return result

    - Nếu inferences gặp trạng thái failure thì ta remove {var = value} và inferences khỏi assignment.
- Return false và qua nhánh khác thực hiện.

Comment thuật toán:

- Select next variable and value
    - Chọn biến theo minimum remaining-values heuristic : chọn ra biến có domain nhỏ nhất (Ưu tiên).
    - Chọn biến theo Degree heuristic : chọn ra biến có số lượng degree lớn nhất (degree : số lượng constraint mà các biến còn liên quan tới).
    - Chọn value theo Least-constraning-value heuristic : chọn ra biến để lại cho chúng ta nhiều lựa chọn nhất cho các biến khác.
- Inference
    - Forward checking : chỉ kiểm tra domain của các biến lân cận nó. Ưu điểm: tốc độ nhanh
    - Maintaining arc consistency : kiểm tra domain của tất cả các biến. Ưu điểm: không bị sót các biến.
    - Hai cách đều có ưu khuyết điểm nên không thể chọn cái nào tốt hơn cái nào.
- Backtracking strategies (chiến lược)
    - Backtrack theo Backjumping : tạo ta tập conflict-set , chứa các assignments có khả năng xung đột với biến hiện tại mình đang xét.
    - Khi gặp failure thì ngay lập tức nhảy về tập conflict-set.
    - Để làm được chúng ta phải cải tiến hàm, có thể dùng con trỏ gọi hàm để thực hiện nó.

Min-conflict

Ý tưởng : ban đầu sẽ khởi tạo giá trị cho tất cả các biến, sau đó chọn ra một biến và thay đổi giá trị biến này sao cho số lượng contrainst bị vi phạm của biến nhỏ nhất có thể.

Đầu vào : csp, max_step (có chức năng giới hạn số lần duyệt)

Đầu ra : solution hoặc failure (failure chỉ mang ý nghĩa chạy hết max-step)

Mô tả thuật toán:

- Gán một giá trị nào đó cho biến hiện tại (current)
- Lặp lại max_step lần:
  - Kiểm tra current có phải là giải pháp mình cần tìm thì mình sẽ dừng thuật toán và trả về solution.
  - Ngược lại, chúng ta chọn một biến nào đó giá trị còn conflict (vi phạm constraint).
  - Lấy một giá trị nhỏ nhất lấy từ hàm CONFLICTS gán cho biến vừa chọn.
- Nếu như đi hết số lần lặp tối đa (max_step) mà vẫn chưa tìm ra solution thì trả về failure.

Tree CSP solver

Đặc điểm: thuật toán này chỉ chạy được trên tree, nếu không phải là tree thì loại bỏ các node (cycle cutset) tạo thành vòng trong graph. Thời gian giải bài toán chỉ tính trên thời gian tuyến tính và không cần dùng backtrack.

Các bước chính:

- Bước 01: chọn một variable (node) làm root. Sau đó sắp xếp các biến trên tree thành có thứ tự (biến đổi graph thì mảng một chiều).
- Chạy thuật toán nào đó trên constraint hai biến (có thể là AC3) duyệt qua tất cả các biến để thu hẹp domain
- Chọn các giá trị trong domain gán cho biến.

Đầu vào : csp

Đầu ra : solution hoặc failure (bài toán không có lời giải)

Mô tả thuật toán:

- Gọi biến n là số lượng variables X.
- Khởi tạo biến assignment (dùng để lưu những biến đã được gán giá trị, ban đầu là rỗng).

- Chọn một biến bất kì là root.
- Sau khi có root thì thực hiện TopologicalSort, sau bước này thì tập X sẽ là tập có thứ tự và được xem như mảng một chiều.
- Thực hiện vòng lặp chạy ngược (từ cuối mảng) để lấy ra cặp biến (nút đang xét và nút cha của nó).
    - Chạy thuật toán bất kỳ (AC3) để thu hẹp domain.
    - Nếu như có một domain nào đó bằng rỗng thì dừng thuật toán và trả về failure.
- Nếu vòng lặp trên được duyệt qua, tất cả domain được thu hẹp nhưng không có có domain nào bị rỗng thì sẽ thực hiện vòng lặp tiếp theo để gán giá trị. Đi theo chiều từ đầu mảng.
    - Lấy lần lượt từng biến ra để gán giá trị consistent
    - Nếu không có giá trị consistent thì dừng thuật toán và trả về failure.
- Sau khi vòng lặp trên kết thúc thì sẽ tìm được solution.

Reinforcement Learning (RL)

1. **Khái niệm:** Reinforcement Learning là một phần của Machine Learning. Ở đây, các agents được tự học về cơ chế khen thưởng và trừ điểm. Các agent sẽ thực hiện các hành động hoặc đi con đường tốt nhất có thể để đạt được số điểm cộng tối đa và điểm trừ tối thiểu thông qua các quan sát trong một tình huống cụ thể.

2. **Các thuật ngữ trong RL**

   Agent – Là người ra quyết định và người học duy nhất

   Environment – một thế giới vật lý nơi agent học hỏi và quyết định các hành động được thực hiện.

   Action – Danh sách các hành động mà agent có thể thực hiện

   State – trạng thái hiện tại của agent trong môi trường.

   Reward – phần thưởng cho agent

   Policy – agent chuẩn bị các chiên lược (quyết định) để lập ra bản đồ các tình huống cho actions.

Value Function – Hàm giá trị cho thấy phần thưởng đạt được tính đến thời điểm các policy được thực hiện.

3. **Ưu - nhược điểm: (Không thuyết trình)**

- Ưu điểm:
  - Có thể giải quyết các vấn đề phức tạp và bậc cao.
  - Là mô hình học hỏi liên tục nên một sai lầm được thực hiện trước đó sẽ khó có thể xảy ra trong tương lai.
  - Phần tốt nhất là ngay cả khi không có dữ liệu đào tạo, nó sẽ học được thông qua kinh nghiệm mà nó có được từ việc xử lý dữ liệu đào tạo.
- Nhược điểm:
  - Chúng ta sẽ lãng phí sức mạnh xử lý và không gian không cần thiết bằng cách sử dụng nó cho các vấn đề đơn giản hơn.
  - Trong thực tế, chi phí bảo trì rất cao.

4. **Ứng dụng:**

- ***Trong kinh doanh, tiếp thị và quảng cáo***: phân tích sở thích của khách hàng và giúp quảng cáo sản phẩm tốt hơn.
- ***Trong trò chơi***: ứng dụng các thuật toán như AlphaGo, Alpha Zero để tính toán chiến thuật chơi các trò chơi như cờ vua, shogi và cờ vây. Ngoài ra RL còn cung cấp môi trường chơi game tốt hơn bằng cách sửa đổi trình giả lập.
- ***Trong hệ thống gợi ý, đề xuất***: chúng ta dễ thấy ứng dụng này trong các nền tảng như Youtube, Netflix. Thuật toán này dựa vào sở thích của khách hàng để hiển thị các chương trình xu hướng mới nhất.

4. **Giới thiệu về thư viên OpenAI Gym:**

OpenAI Gym là một bộ công cụ cung cấp nhiều loại môi trường mô phỏng (trò chơi Atari, tr ò chơi trên bàn, mô phỏng vật lý 2D và 3D, v.v.), có thể đào tạo các agents, so sánh chúng hoặc phát triển các thuật toán Máy học mới (Reinforcement Learning).

5. **Cross-entropy method**

- Ý tưởng:
  - Bước 1: cho chạy N episodes (được hiểu như là các lần done)

- Bước 2: tính toán total reward cho từng episode và quyết định ngưỡng reward. Thường thì chúng ta sẽ dùng percentile cho các rewards (50th và 70th)
- Bước 3: bỏ các episode có total reward nhỏ hơn ngưỡng.
- Bước 4: Huấn luyện các “elite” episodes bằng cách sử dụng các quan sát làm input các actions làm đầu ra mong muốn.
- Bước 5: lặp lại bước 1 cho đến khi có được kết quả như mong đợi.

Các câu hỏi vấn đáp

1. **So sánh BFS và DFS**

- BFS:
  - Completeness: thuật toán BFS luôn cho ra được solution
  - Optimality: thuật toán BFS optimality khi có ràng buộc điều kiện PAHT-COST phải là hàm tăng dần theo chiều sâu
  - Complexity: thuật toán BFS rất tốn kém về thời gian và bộ nhớ
- DFS:
  - DFS giải quyết được vấn đề tồn động lại của BFS và Uniform cost search là Complexity bằng cách đổi loại hàng đợi của tập frontier thay vì dùng FIFO và Priority queue ta sử dụng LIFO.
  - Về code thì DFS và BFS cơ bản giống nhau chỉ thay đổi hàng đợi của tập frontier. Chính vì sử dụng LIFO nên thuật toán sẽ duyệt theo chiều sâu nó sẽ bắt lấy một nhánh và đào sâu vào đến khi đến đường cùng thì nó loại bỏ nhánh đó hoàn toàn khỏi bộ nhớ và sang nhánh khác đây là mấu chốt giúp cho DFS sử dụng ít bộ nhớ và thời gian hơn BFS
  - Tuy nhiên nó sẽ phát sinh ra vấn đề DFS sẽ cho ra solution không Optimality. Vì phải duyệt theo chiều sâu nên DFS có thể không Completeness khi nó chọn vào nhánh có chiều sâu vô cùng và không tìm thấy goal_state.

2. **So sánh A* với Uniform, Best First Search**

- Best First Search: Đặc điểm của Best-first search là f(n) = h(n) (với heuristic function h(n) là ước lượng từ node n đến goal_state). Best-first search còn tồn động vấn đề là không tối ưu (not optimal).

- Ước lượng chi phí của nó không đầy đủ thiếu chính xác f(n) = h(n) là ước lượng chi phí của Best-first search tuy nhiên để đầy đủ hơn f(n) = chi phí từ vị trí ban đầu đến node hiện tại (PATH-COST) + h(n).
- Ước lượng không chính xác h(n) khác biệt rất lớn so với thực tế dẫn đến việc không tối ưu.

- A*: Để giải quyết các vấn đề của Best-first search A* search có một vài cải tiến như sau:
  - f(n) = PATH-COST + h(n) việc bổ sung thêm PATH-COST giúp cho ước lượng của A* search trở nên đáng tin cậy và đầy đủ.
  - A* search ràng buộc thêm điều kiện cho h(n) (consistent heuristic satisfies): h(n) <= cost(n, n') + h(n') bất đẳng thức này đơn giản chỉ là bất đẳng thức tam giác.

  → Những cải tiến giúp cho A* search complete và optimal* (h(n) consistent)
- Best First Search: Về mô tả thuật toán thì tương tự như BFS nhưng thay vì tập frontier sử dụng loại hàng được FIFO thì ta sử dụng Priority queue ta xét theo PATH-COST.
  - Trong khi thực hiện vòng lặp ta lấy node từ tập frontier thay vì lấy như BFS thì ta xét xem node nào có PATH-COST ít nhất thì được ưu tiên lấy ra trước
  - Sau khi expand các node con từ node đang xét tương ứng với một action ngoài điều kiện là node con này không thuộc tập frontier và tập explored mới được đưa vào tập frontier thì trường hợp node con này đã có trong tập frontier ta sẽ so sánh nếu node con có PAHT-COST ít hơn node con đã tồn tại trong tập frontier thì ta thay thế node con cho node đã tồn tại trong tập frontier

  → Tuy thuật toán Uniform - cost search giải quyết được vấn đề Optimality của BFS nhưng nó vẫn chưa giải quyết được vấn đề về Complexity

3. **Những vấn đề của Hilling Search và cách giải quyết**

- Quá nhiều successors. Cách giải quyết: dùng Stochastic hill climbing để sinh ngẫu nhiên các successor cho đến khi tìm được successor có value lớn nhất.
- Local optimum. Cách giải quyết:

- Dùng Random-restart hill-climbing. Nghĩa là trong trường hợp bị rơi vào local optimum, ta sẽ chạy lại thuật toán hill-climbing với giá trị INITIAL_STATE là một state được chọn random (khác với Initial State ở trường hợp lúc đầu).
- Dùng Local beam search (beam: chùm ánh sáng): (cải tiến của random restart hill-climbing).
- Instead of trying k random searches sequentially, do them simultaneously. Nghĩa là đồng thời thực hiện việc tìm kiếm theo k hướng khác nhau (đa nhiệm), thay vì lượt lượt thử từng cách một, nếu chưa thấy thì restart.
- Local beam search: not just simultaneously, choose best of all searches. Không chỉ chọn best neighbor cho mỗi search, mà nó chọn best of all searches (lấy cái tốt nhất của các best successors thành phần → tốn thời gian để tìm max).
- Stochastic beam search: combines Local beam search and Stochastic hill-climbing. Giải quyết vấn đề này sinh của Local beam search là phải tìm best successor. Khi có quá nhiều successor sẽ dẫn đến tốn nhiều chi phí cho việc tính Value cho mỗi successor và tìm max.
  - Không tìm max
  - Chọn successor ngẫu nhiên với xác suất dựa theo VALUE

4. **Nêu những thành phần chính của CSP**

- Trong CSP có 3 components :
  - X (tập hợp tất cả các biến)
  - D (tập hợp các giá trị mà các biến được phép nhận)
  - C (tập hợp các ràng buộc mà các biến phải thỏa mãn)

5. **Các thuật toán của CSP**

- AC – 3 algorithm (arc consistency)
- PC2 (tương tự AC3)
- K cositstence
- Backtracking search algorithm
- Min-conflict algorithm.

- Tree CSP solver