BFS/DFS/UCS for Sokoban

Trí tuệ nhân tạo - CS106.M21

Nguyễn Đức Anh Phúc
20520276

03/31/2022

Mục Lục

01.01 Bàn luận về hướng tối ưu của thuật toán UCS

- Hướng mở rộng này được đúc kết từ kinh nghiệm lập trình của bản thân và được chia sẻ trong bài báo cáo này.
- Thuật toán UCS khác với thuật toán Dijkstra ở chỗ rằng từ nút gốc, thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh gốc tới mọi đỉnh. Còn thuật toán UCS sẽ tìm đường đi ngắn nhất từ đỉnh gốc cho đến khgi nào tìm được đến goalState thì dừng thuật toán và sẽ không chạy thêm nữa
- Giả sử cây tìm kiếm $T(V, W)$ có hướng có tập vô hạn hay hữu hạn các trạng thái. Với trạng thái gốc là trạng thái S và trạng thái goalState là E. V là tập các đỉnh trạng thái của cây còn W là tập các *weight* của các cạnh. Ta gọi $\delta(u, v)$ là độ dài đường đi ngắn nhất từ u đến v.
- Chúng ta có một nhận xét rằng thuật toán UCS hay tối thiểu Dijkstra sẽ rơi vào trường hợp xấu nhất là đường đi ngắn nhất cần tìm sẽ là đường đi dài nhất trong cây. Lúc đó việc duyệt toàn bộ cây tìm kiếm là điều không thể tránh khỏi.
- Chúng ta biết trước các trạng thái S và E ban đầu. Như thông thường thuật toán UCS sẽ tìm đường đi mở rộng từ S với nhãn đường đi $d[i]$ là độ dài đường đi từ đang xét từ S đến i. Thuật toán sẽ "cố gắng" tối thiểu hóa $d[i] \rightarrow \delta(i)$.
- Nếu chúng ta cũng thực thi thuật toán UCS đồng thời từ E thì sao? Khi đó chúng ta cần lật ngược các cạnh lại tuy nhiên vẫn giữ nguyên trọng số. Đối với một đồ thị cây đặc (nhiều cạnh) thì việc thuật toán UCS rơi vào trường hợp xấu nhất là độ dài đường đi cần tìm từ $S -> E$ là dài nhất trong T thì việc thực thi thuật toán từ S với việc thực thi thuật toán đồng thời từ S và E và đến một lúc nào đó thì không gian tìm kiếm sẽ cùng gặp nhau tại một trạng thái. Lúc đó việc kết hợp 2 đường đi với nhau vô cùng đơn giản. Việc này giúp giảm thiểu chi phí tính toán rất đáng kể thay vì thuật toán UCS thường.
- Hướng mở rộng này có thể được áp dụng cho graph $G(V, E)$ và đơn giản nên Pseudocode mình có thể tự thiết kế.

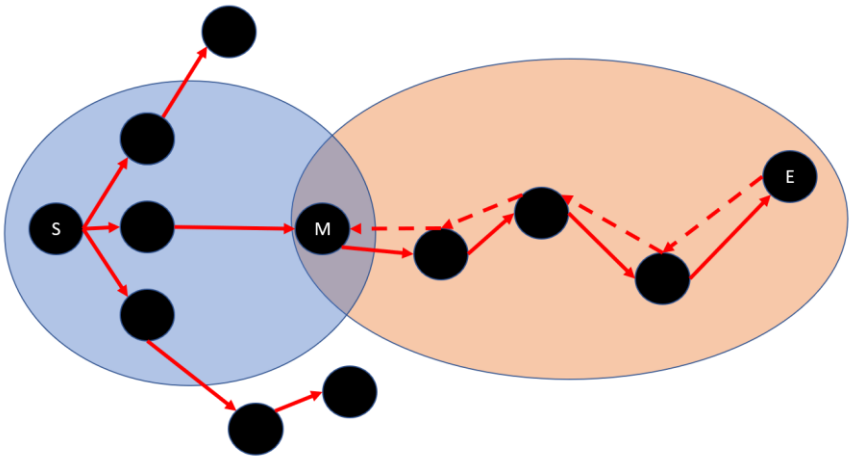

Fig 1. Với tập màu xanh là tập δ_1(từ S) và tập màu cam δ_2(từ E). Khi 2 tập cùng gặp nhau tại một đỉnh M thì $\delta(S, E) = \delta(S, M) + \delta(M, E)$

01.02 Bài toán Sokoban

01.02.01 Mô hình hóa Sokoban

01.02.01.1 Resource

– Các level nằm trong thư mục *assets/sokobanLevels* được biểu diễn bằng các file .txt với tổng cộng là 18 levels

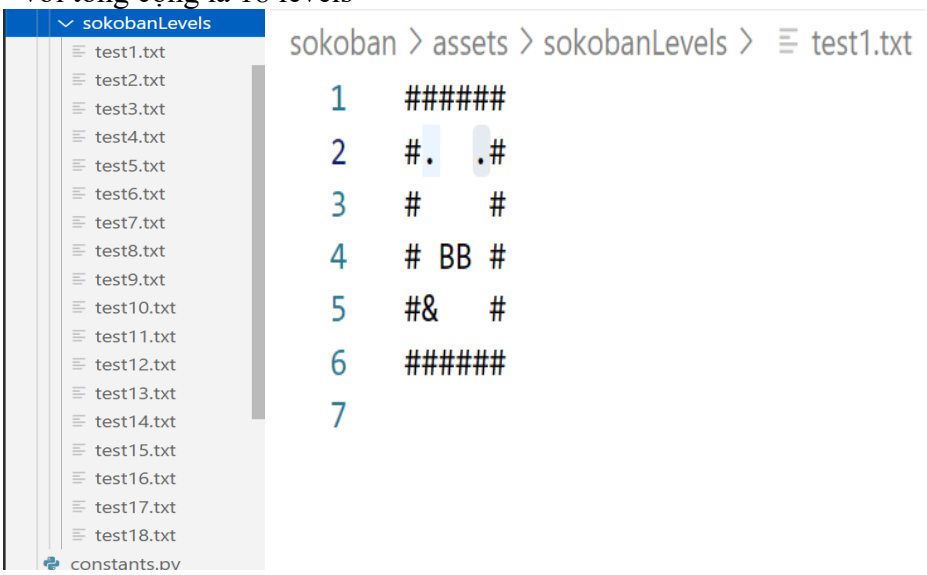

Fig 2. Các levels của game Sokoban & level 1 của game

–

+ Đối tượng Wall được biểu diễn bằng ký tự "#"
+ Đối tượng Player được biểu diễn bằng ký tự "&"
+ Đối tượng Target được biểu diễn bằng ký tự "."
+ Đối tượng Valid Box được biểu diễn bằng ký tự "X"
+ Free Space chỗ trống được biểu diễn bằng ký tự " "

01.02.01.2 File Level.py

– Dùng để convert file *.txt* sang *layout* tương ứng trong hàm *load* vào *position_player* (***tuple***) và container *structure* (***list***) với giá trị theo file constant.py:

```
# Blocks
AIR             = 0
WALL            = 1
PLAYER          = 2
BOX             = 3
TARGET          = 4
TARGET_FILLED   = 5
```

01.02.01.3 Hàm transferToGameState2

– Input: **layout, player_pos**

+ Layout: Layout được nhận từ *structure* từ level.py
+ Player_pos: Được nhận từ *position_player* từ level.py

– Được dùng để convert từ Layout đầu vào (***list***) sang data structure được dùng trong bài toán tương ứng (***numpy***)

01.02.01.4 Hàm get_move

– Input: **layout, player_pos, method**
+ Layout: có được từ *level.structure* được tạo nên từ class *Level* trong level.py (đọc các file .txt tương ứng)
+ Player_pos: có được từ *position_player* được tạo nên từ class *Level* trong level.py
+ Method: Có ba phương án là *dfs,bfs,ucs* tương ứng với Depth First Search, Breadth First Search và Uniform Cost Search.

01.02.01.5 Hàm auto_move

– Dùng để lựa chọn các options *dfs, bfs, ucs*

01.02.01.6 Sokoban.py

– Thực thi để load resources và level từ game thông qua class *Game* trong game.py hàm *load_textures* và *load_level*
– Phần UX thì sẽ liên quan đến các hàm render và update screen nên bài báo cáo này sẽ không bàn kĩ, chỉ nói kĩ đến phần logic và algorithmic source

01.02.02 Trạng thái khởi đầu kết thúc

– Các trạng thái khởi đầu được load từ class *Level* bao gồm contatiner ***list*** chứa dữ liệu được trích xuất từ các file '***assets/.txt***'
– Sau đó, các trạng thái được nạp vào *transferToGameState2* để chuyển thành ***numpy array*** và ***tuple***
+ *gameState* là dãy *numpy* layout và tuple *player_pos*
– 2 biến global vì chúng cố định theo thời gian thể hiện
+ *posWalls* là kết quả của hàm *PosofWalls* dãy các tuple giá trị mà tại đó là Wall ứng với $constant = 1$ trong constant.py
+ *posGoals* là kết quả của hàm *PosofGoals* dãy các tuple giá trị mà tại đó là vị trí các goal ứng với $constant = 3$ hay $constant = 5$ trong constant.py
– 2 biến local vì chúng thay đổi theo thời gian thể hiện
+ *beginBox* là kết quả của hàm *PosofBoxes* là các giá trị ban đầu tuple vị trí của các box
+ *beginPlayer* là kết quả của hàm *PosofPlayers* là giá trị ban đầu tuple vị trí của player
– *isEndState* là hàm được thiết kế để dùng để kiểm tra xem vị trí các box đã khớp với vị trí các target hay chưa bằng cách sort các vector rồi so sánh chúng với nhau
– Mỗi *gameState* là bao gồm ***numpy array*** được trích xuất từ *layout* và ***tuple*** vị trí của *player*. Từ đó người ta mô hình hóa đơn giản trạng thái bao gồm
+ Tuple vị trí của player

+ Các tuple vị trí các box
- Với các dữ liệu toàn cục cố định là Wall và Goals.

01.02.03 Không gian trạng thái

- Không gian trạng thái được quản lí bởi hàm *legalActions*

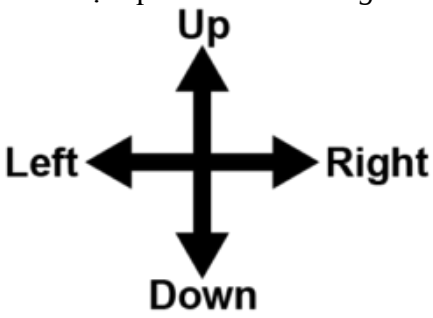

Fig 3. Có 4 hướng đi là Up Down Left Right

- List các hành động $allActions[]$ chứa các cách đi $[x, y, c_1, c_2]$
  + x, y là offset so với vị trí hiện tại
  + c_1, c_2 là l, r, u, d hay L, R, U, D
- c_1, c_2 viết thường tương ứng là việc đi bình thường, viết hoa tương ứng với việc đẩy 1 box.
- Lặp qua mỗi hành động trong $allActions$ với *xPlayer* và *yPlayer* là vị trí hiện tại. Vị trí mới sẽ là $(xPlayer + x,\ yPlayer + y)$
- Kiểm tra tính hợp lệ xong rồi append các hành động hợp lệ dựa vào hàm kiểm tra tính hợp lệ của nước đi
- Kết quả đầu ra của hàm sẽ là các tuple hợp lệ $[x, y, c_1, c_2]$
  + $[x, y, c_1, c_2] \in \{[-1,0,' u',' U'], [1,0,' d',' D'], [0,-1,' l',' L'], [0,1,' r',' R']\}$

01.02.04 Các hành động hợp lệ

- Các hành động hợp lệ được xác định bởi hàm *isLegalActions*
- $(xPlayer + x,\ yPlayer + y)$ với
  + *xPlayer* và *yPlayer* là vị trí hiện tại của *Player*
  + x, y là các offset của action đang xét
- Chúng ta sẽ kiểm tra xem các hành động có bị va vào tường || đi vào trong tường (vi phạm qui định của bài toán) với việc xét xem
  + $(x + xPlayer, y + yPlayer)\ not\ in\ posBox\ +\ posWalls$
- Một nước đi sẽ là hợp lệ nếu như hàm trên thỏa

01.02.05 Hàm tiến triển (Successor function)

- Một hàm với đầu vào là trạng thái thực hiện và một hành động sau đó tính toán chi phí thực hiện hành động đó cũng như đề xuất ra trạng thái kế tiếp.

- Trong DFS và BFS, khi đưa vào một trạng thái, cả 2 phương pháp tìm kiếm này đều duyệt không gian tìm kiếm một cách ngẫu nhiên với dựa vào trạng thái state đó đã thăm hay chưa và sự hợp lệ của hành động
  + State đã thăm
  + Sự hợp lệ của hành động (Hành động hợp lệ thỏa hàm hợp lệ)
- Khác với BFS và DFS, UCS kế thừa từ thuật toán Dijkstra, sử dụng Priority Queue (Min Heap) để đưa vào lựa chọn state thích hợp từ đó từ không gian {Up, Down, Left, Right} đề xuất ra state kế tiếp

```
def cost(actions):
    return len([x for x in actions if x.islower()])
```

- Hàm cost là quãng đường (số actions thực hiện) trong actions (Đi không đẩy thùng)
- Một action có $cost = 1$

01.03 Số bước thực hiện các bản đồ

- 1
  + DFS: 79
  + BFS: 12
  + UCS: 12
- 2
  + DFS: 24
  + BFS: 9
  + UCS: 9
  +
- 3
  + DFS: 403
  + BFS: 15
  + UCS: 15
  +
- 4
  + DFS: 27
  + BFS: 7
  + UCS: 7
- 5
  + DFS: X
  + BFS: 20
  + UCS: 20
- 6
  + DFS: 55
  + BFS: 19
  + UCS: 19
- 7
  + DFS: 707
  + BFS: 21
  + UCS: 21
- 8
  + DFS: 323
  + BFS: 97
  + UCS: 97
- 9
  + DFS: 74
  + BFS: 8
  + UCS: 8
- 10
  + DFS: 37
  + BFS: 33
  + UCS: 33
- 11
  + DFS: 36
  + BFS: 34
  + UCS: 34
- 12
  + DFS: 109
  + BFS: 23
  + UCS: 23
- 13
  + DFS: 185
  + BFS: 31
  + UCS: 31
- 14
  + DFS: 865
  + BFS: 23
  + UCS: 23
- 15
  + DFS: 291
  + BFS: 105
  + UCS: 105
- 16
  + DFS: X
  + BFS: 34
  + UCS: 34
- 17
  + DFS: X
  + BFS X
  + UCS: X
- 18
  + DFS: X
  + BFS: X
  + UCS: X

01.04 Nhận xét

- Lời giải của thuật toán DFS là lời giải chưa tối ưu bởi vì DFS sẽ tìm kiếm kết quả theo chiều sâu và kết quả đó chưa chắc đã là kết quả tối ưu
- Lời giải của thuật toán BFS là lời giải tối ưu cho bài toán này vì BFS sẽ tìm kiếm kết quả theo chiều rộng và kết quả đó chắc chắn là kết quả tối ưu bởi vì ở đây, cost của mỗi action luôn luôn là 1.
- Lời giải của thuật toán UCS là lời giải tối ưu cho bài toán này vì UCS sẽ tìm kiếm kết quả theo thuật toán Dijkstra, và kết quả đó chắc chắn là kết quả tối ưu bởi vì cây tìm kiếm có đều có trọng số $= 1$ $(>= 0)$
- Việc đánh giá thuật toán nào tốt hơn mang tính rất khách quan vì điều đó phụ thuộc rất là nhiều thứ. Nhưng nếu để nói về optimal results thì BFS sẽ chắc chắn tối ưu hơn UCS bởi vì lượng node được push vào priority queue = lượng node được push vào queue vì cost của mỗi action đều $= 1$. Priority queue được xây dựng bằng MinHeap nên độ phức tạp của thuật toán UCS là $O(MlogN)$ còn BFS sẽ là $O(M + N)$
  + M là số cạnh của cây tìm kiếm
  + N là số đỉnh của cây tìm kiếm
- Thường một cây sẽ có số lượng nút là N và số lượng cạnh là $N - 1$
- Map 17,18, 19 không có kết quả hay nói cách khác là không tồn tại trạng thái goal State. Cho dù tìm kiếm bằng phương pháp nào thì cũng không ra được kết quả.
- Riêng map 5 và map 16 thì DFS thực hiện tính toán quá lâu, điều này cũng dễ hiểu bởi vì ở map 5 và 16, có rất nhiều ô trống, điều này dẫn đến việc bài toán cây tìm kiếm trạng thái sẽ sinh ra rất nhiều các trạng thái, duyệt cây DFS sẽ tìm kiếm lần lượt các nhánh. DFS có thể được tối ưu bằng việc ở đầu mỗi nhanh, chúng ta cần có một hàm $f(i)$ để xác định rằng nếu từ subtree i duyệt xuống sẽ không tồn tại kết quả, hay nói cách khác, nút trạng thái i sẽ không là cha của nút trạng thái goalState.

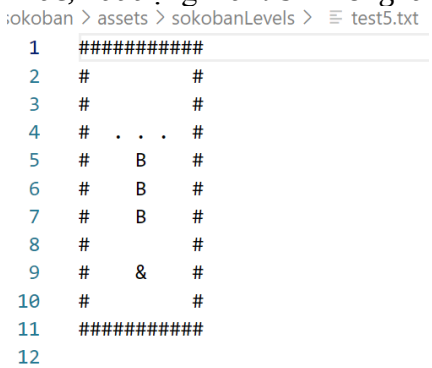

Fig 4. Bản đồ số 5

- Đây cũng là hướng tối ưu cho thuật toán DFS.

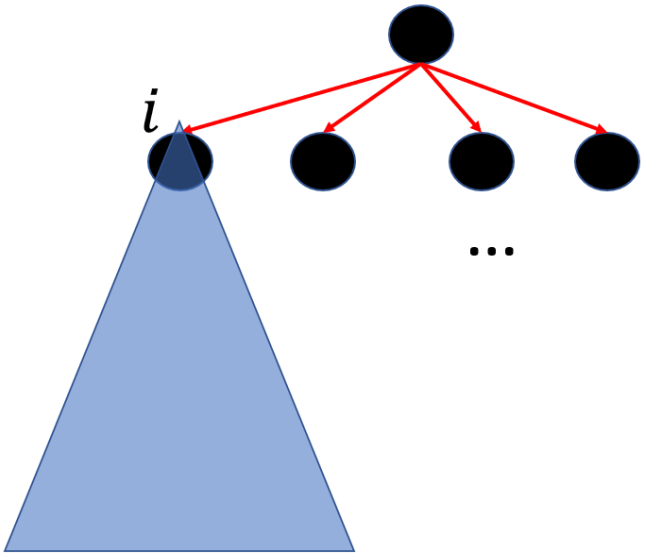

Fig 5. Cây tìm kiếm với mỗi nút *i* sẽ là gốc của subtree gốc *i*, hay nói các khác là *i* là Lowest Common Ancestor(LCA) của các nút thuộc subtree gốc *i*