`Python3 -P SimpleSurvivorProblem -f tinyHouseSearch -l tinyHouse -x 1.0 -z 2.0`

↳ (Always returns the sequence to solve tinyHouse layout.)

Gother layouts fail

↓

`readCommand(argv)`

PROBLEM_CHOICES → ① SimpleSurvivor
→ ② MultiSurvivor

mandatory args

Flags =

-p	problem	(one of the problem choices)
-f	function	(search: BFS, tinyHouse, A*)
-h	heuristic	(nullHeuristic, manhattanHeuristic)
-l	layout	(layout file: burning office, powerplant, etc)
-t	textgraphics	(display steps text output
-q	quietgraphics	(no graphics, only final output)
-z	zoom	(float multiplier for window zoom)
-r	recordActions	(writes actions to a file)
-c	catchExceptions	(turns on exception handling during mission)

- Loads layout, identifies rescuer, and passes args to runMission.

↓

`runMission(**args)`

args = {**} → layout
rescuer
display
record
catchException

This calls newMission on the RescueMission instance (RescueMission)

`newMission(layout, rescuer, display, False, catchExceptions)`

↓

Initializes a new RescueState (initState) that contains:

- Rescuer position
- Survivors remaining
- Terrain layout

A new Game is created (mission)

`Game(agents, display, self, catchExceptions)`

↳ mission contains the following data

Game

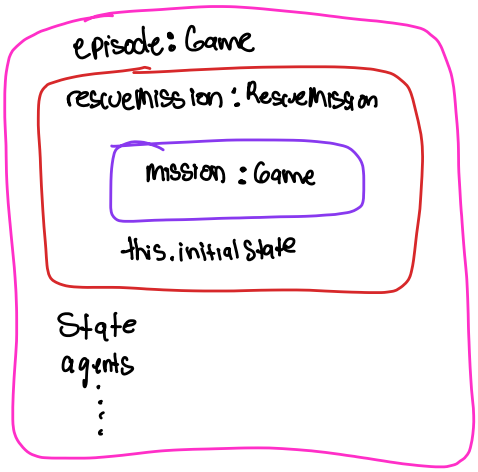
agentCrashed: bool
agents: Agent[]
display: RescueGraphics
rules: RescueMission
startingIndex: int
gameOver: bool
muteAgents: bool
catchExceptions: bool
moveHistory: Array[]
totalAgentTimes: Array[]
totalAgentTimeWarnings: Array[]
agentTimeout: bool

We set mission's state to initState, then set rescueMission's initialState to a deepcopy of initState, and set rescueMission's quiet value.

This object is now calle episode.

We now call run() on episode object.

run()



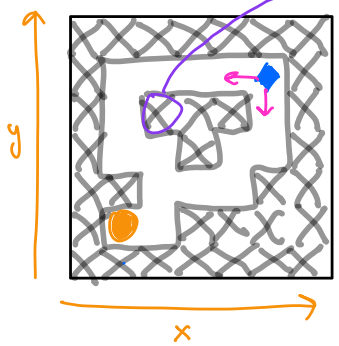
① The game's display is initialized with the layout and all attributes about terrain, survivors, etc and their locations.

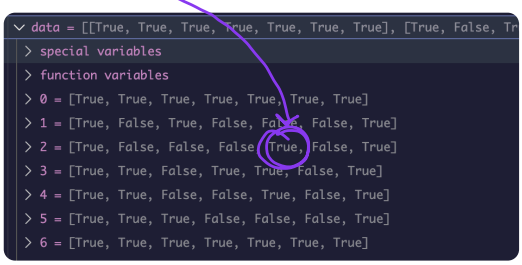
② The agents are initialized with initialState via registerInitialState().

Only A* uses heuristics

③ The agent (SearchAgent) sets the problem (search problem) by calling the init from either SimpleSearch.

This problem contains locations of walls





using the problem, we pass it to the searchFunction() so we can get the sequence of moves to solve the given layout.

[this is where we must add the search algorithms!] (search.py)

This is when we get the cost based on the terrain conditions.

In getCostOfActions, using the differentials dx and dy and the terrain type crossed over, we add up the cost of the path taken.

We then draw each step on the map in sequence.

DFS

Traverse single branch as deep as possible, then backtrack and try another path until all nodes have been visited.

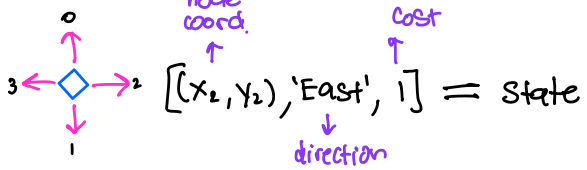
Setup

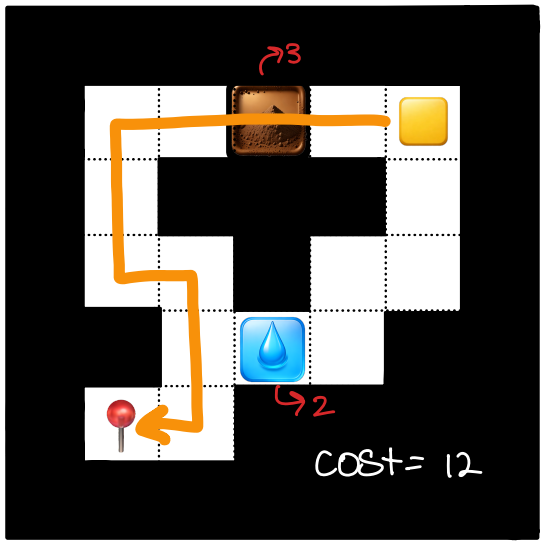
1. Get successor states from startState
2. Initialize stack

Search (while stack is not empty)

1. Peek state from stack → currState
2. Check if currNode is the goal
3. Visit currNode
4. Get currState's neighbors list
5. If neighbors, pop last neighbor
6. If no neighbors, pop next state from stack
7. If not visited, get successor states
8. If visited, start loop again
9. push neighbor states onto stack
10. continue until goal found or stack empty

neighbor index





AI:

- Explain the general idea behind DFS
- Help me debug why the stack is emptying before the goal is found?

BFS

Explore every node level by level by visiting direct neighbors before visiting the next layer.

Setup

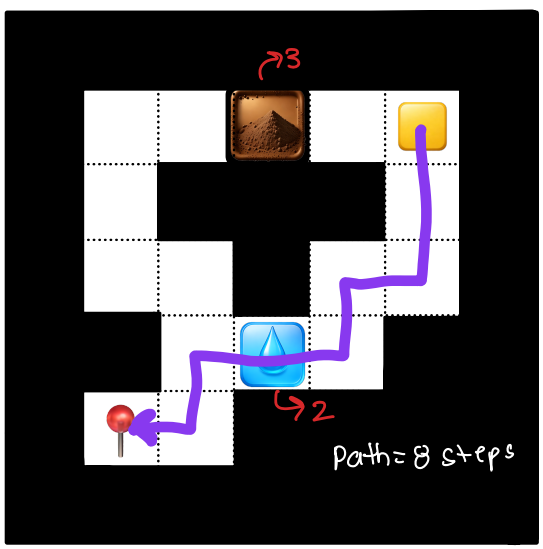
1. Get successor states from startState
2. Initialize a queue, parent dictionary, and set
3. mark start node as visited

Search (while queue is not empty)

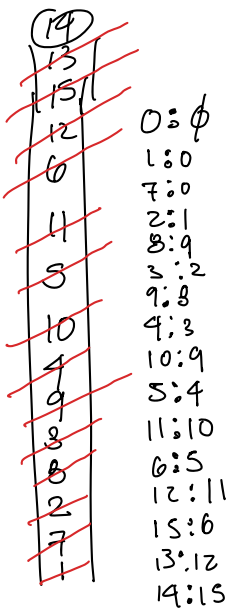
3. pop 1st state in line
4. check if it's node is the goal
5. mark node as visited
6. Get all neighbors
7. If neighbor is not visited,
8. Mark as visited
9. Enque neighbor
10. Add it to dictionary w/ current Node as parent
12. End when queue is empty or goal found

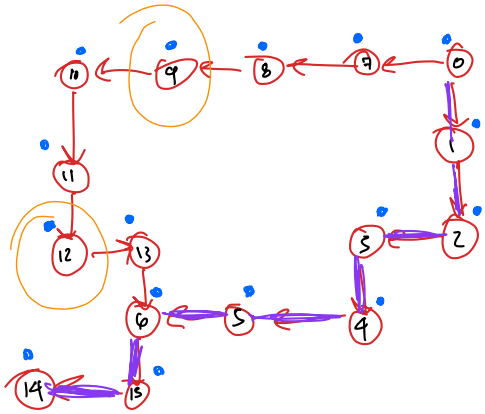
AI

- My code works, it visits nodes on each level but I need help reconstructing the path.



walkthrough





Uniform Cost Search

To find cheapest path, we use dijkstra's algorithm which greedily explores the graph by visiting the newest unvisited node next, updating cost as it finds cheaper routes.

AI: The textbook implementation of Dijkstra's algorithm initializes all nodes to ∞ at first. How do I change it so it don't do this?

Setup

① Init cost dict. to 0 and came_from dict. to ∅
② create empty set to keep track of seen nodes
③ Add start node to priority queue

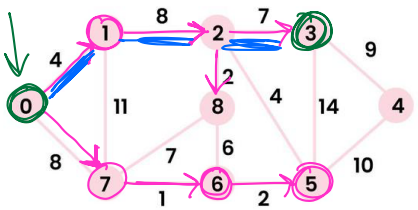
Search (while queue is not empty)

① Pop lowest-cost node
② If not seen, mark as seen. Else, skip loop
③ If current node is goal, build path
④ For each neighbor calculate new cost
⑤ If neighbor not in cost dict. or newcost is cheaper than existing cost to reach that neighbor:
- 5a update the cost of reaching it to newcost
- 5b add neighbor to came_from dict. with current
- 5c push neighbor and new cost to PQ

Example

cost [0] = 0
cost [1] = 4
cost [2] = 12
cost [3] = 25 19
cost [4] = ∞
cost [5] = 11
cost [6] = 8
cost [7] = 8
cost [8] = ~~15~~ 14

prev [0] = ∅
prev [1] = 0
prev [2] = 1
prev [3] = ~~5~~ 2
prev [4] = ∅
prev [5] = 6
prev [6] = 7
prev [7] = 0
prev [8] = ~~7~~ 2



for start

PQ
(8,14)
(8,14)
(9,21)
(3,25)
(5,11)
(6,9)
(8,15)
(2,12)
(7,8)
(1,4)
(0,0)

Seen = {0, 1, 7, 6, 5, 2, 8,

(0,0) curr_cost = 0
[1] n_c = 4 ✓
[7] n_c = 8 ✓

(1,4) curr_cost = 4
[2] n_c = 12 ✓

(7,8) curr_cost = 8
[8] n_c = 15 ✓
[6] n_c = 9 ✓

(6,9) curr_cost = 9
[5] n_c = 11 ✓

(5,11) curr_cost = 11
[3] n_c = 25 ✓
[4] n_c = 21 ✓

(2,12) curr_cost = 12
[8] n_c = 14
[3] n_c = 19

(8,14)

A* Search

A* uses a heuristic to bias expansion of nodes toward the goal and ignores nodes that are cheap but in the wrong direction.

AI: I know A* and Dijkstra differ in that one uses a heuristic. what's the best pythonic way of re-using my Dijkstra function so I can have two versions?

MultiSurvivor

To find the least costly path that saves each survivor we consider the MST of the graph connecting the starting node with all the victim nodes.

With help from Claude, I now know that I can build a virtual graph using manhattan distance to each survivor and running prim to get cost of mst of virtual graph to feed the heuristic in A*.

AI: I know I need the MST to find all survivors but how do I do it without knowing all edge costs in advance?

AI: I would end up running dijkstra many times during each Prim iteration though, that's too slow. Other suggestions?

AI: Precomputing dijkstra at the start from each key node can feed a prim algorithm that uses real costs but can I optimize this further?

AI: Give me pseudocode for Prim.