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

(Always returns the sequence to solve tinyHouse layout)

other 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.

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

runMission(**args)

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)

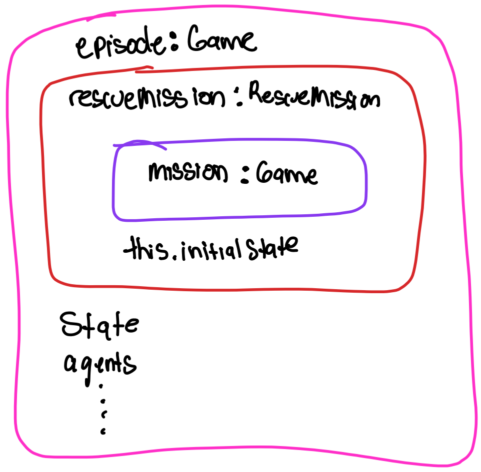
mission contains the following data

Game
agentCrashed : bool
agents : Agent[]
display : RescueGraphics
rules : RescueMission
startingIndex : int
gameOver : bool
muteAgents : bool
catchExpressions : 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()

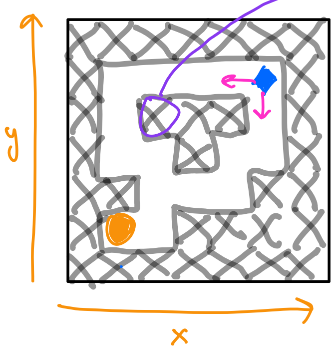
① The game's display is initialized with the layout and all attributes about terrain, survivors, etc and their locations.

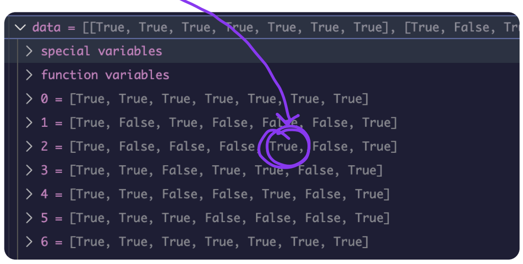
② The agents are initialized with initial state via registerInitialState().

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





```
data = [[True, True, True, True, True, True, True], [True, False, Tr
  > special variables
  > function variables
  > 0 = [True, True, True, True, True, True, True]
  > 1 = [True, False, True, False, False, False, True]
  > 2 = [True, False, False, False, True, False, True]
  > 3 = [True, True, False, True, True, False, True]
  > 4 = [True, True, False, False, True, False, True]
  > 5 = [True, True, True, False, False, False, True]
  > 6 = [True, True, True, True, True, True, True]
```

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

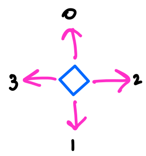
Traverse single branch as deep as possible, then backtrack and try another path until all nodes have been visited.

Setup

① Get successor states from startState
② init 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
⑧ If not visited, get successor states
⑦ If visited, start loop again
⑩ push neighbor states onto stack
⑪ continue until goal found or stack empty

neighbor index



$\underset{\text{direction}}{\overset{\text{node coord.}}{[(x_2, y_2)}, \text{'East'}}, \overset{\text{cost}}{1}] = \text{State}$