COMP 6721 Applied Artificial Intelligence (Fall 2023)

Lab Exercise #2: State Space Search

Solutions

Question 1 Once upon a time a farmer went to the market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river and rented a boat. But in crossing the river by boat, the farmer could carry only himself and a single one of his purchases – the fox, the goose, or the bag of the beans.

If left alone, the fox would eat the goose, and the goose would eat the beans. The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact.

Represent this problem as a search problem. Choose a representation for the problem's states and:

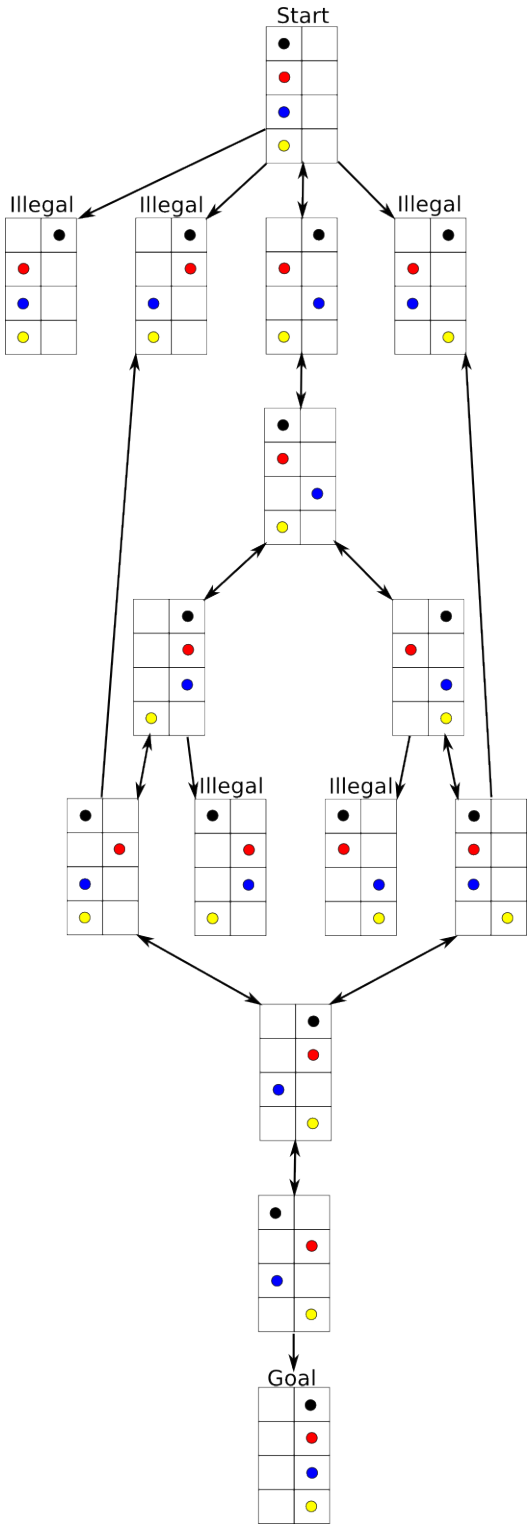
(a) Write down the initial state
Let position(farmer, fox, goose, beans) represent the position of the farmer, the fox, the goose and the beans with respect to the river. The possible positions are o *for "original bank" and* f *for "far bank".*
Initially, the state is: position(o,o,o,o)

(b) Write down the goal state
position(f,f,f,f)

(c) Write down all illegal states
positions(f,o,o,o), positions(f,o,o,f), positions(o,f,f,o), positions(o,f,f,f), positions(f,f,o,o), positions(o,o,f,f).

(d) Write down the possible actions
moveFar(farmer, empty), moveFar(farmer, fox), moveFar(farmer, goose), moveFar(farmer, beans), moveBack(farmer, empty), moveBack(farmer, fox), moveBack(farmer, goose), moveBack(farmer, beans)

(e) Draw the state space for this problem

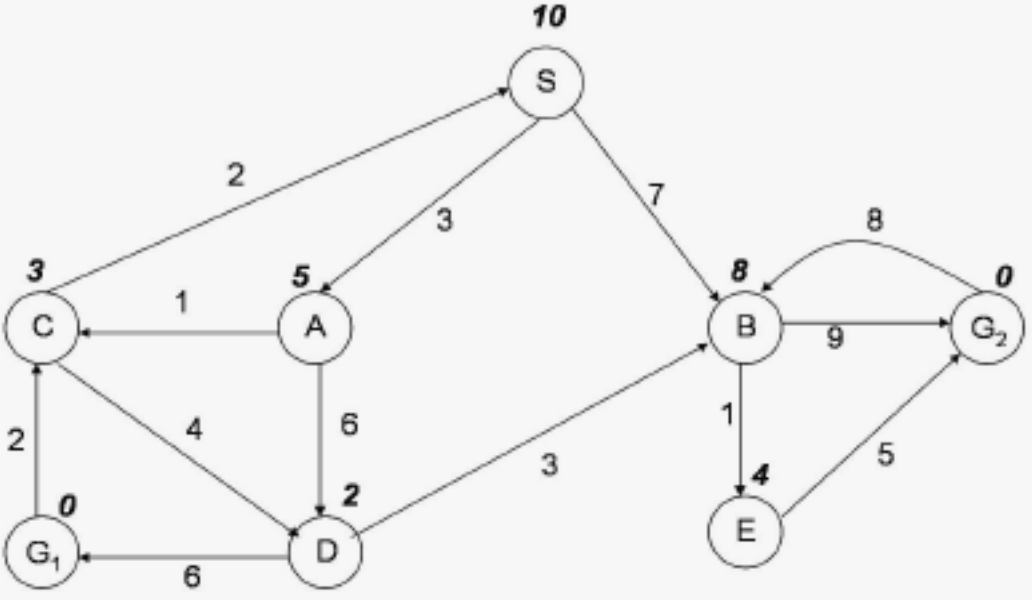


(f) Find a series of moves to solve this problem
moveFar(farmer, goose), moveBack(farmer, empty), moveFar(farmer, fox), moveBack(farmer, goose), moveFar(farmer, beans), moveBack(farmer, empty), moveFar(farmer, goose).

OR

moveFar(farmer, goose), moveBack(farmer, empty), moveFar(farmer, beans), moveBack(farmer, goose), moveFar(farmer, fox), moveBack(farmer, empty), moveFar(farmer, goose).

Question 2 Assume that S is the initial state and G_1 and G_2 are the goal states. The possible actions between states are indicated by arrows. The number labelling each arc is the actual cost of the action. For example, the cost of going from S to A is 3. The number in bold italic near each state is the value of the heuristic function h at that state. For example, the value of h at state C is 3. When all else is equal, expand states in alphabetical order.



For the following search strategies, show the states visited, along with the open and closed lists at each step (where it applies).

(a) Breadth-first search

Step	Visited State	Closed List	Open List
1			S
2	S	S	A, B
3	A	S, A	B, C, D
4	B	S, A, B	C, D, E, G2
5	C	S, A, B, C	D, E, G2
6	D	S, A, B, C, D	E, G2, G1
7	E	S, A, B, C, D, E	G2, G1
8	G2	S, A, B, C, D, E, G2	G1

(b) Depth-first search

Step	Visited State	Closed List	Open List
1			S
2	S	S	A, B
3	A	S, A	C, D, B
4	C	S, A, C	D, B
5	D	S, A, C, D	G1, B
6	G1	S, A, C, D, G1	B

(c) Iterative deepening depth-first search

Note: Depth of each state in the open list is in parenthesis.

Depth 1:

Step	Visited State	Closed List	Open List
1			S(1)
2	S	S	

Depth 2:

Step	Visited State	Closed List	Open List
1			S(1)
2	S	S	A(2), B(2)
3	A	S, A	B(2)
4	B	S, A, B	

Depth 3:

Step	Visited State	Closed List	Open List
1			S(1)
2	S	S	A(2), B(2)
3	A	S, A	C(3), D(3), B(2)
4	C	S, A, C	D(3), B(2)
5	D	S, A, C, D	B(2)
6	B	S, A, C, D, B	E(3), G2(3)
7	E	S, A, C, D, B, E	G2(3)
8	G2	S, A, C, D, B, E, G2	

(d) Uniform cost search

Step	Visited State	Closed List	Open List
1			S(0)
2	S	S	A(3), B(7)
3	A	S, A	C(4), B(7), D(9)
4	C	S, A, C	B(7), D(8); lower cost to D replaces previous cost
5	B	S, A, C, B	D(8), E(8), G2(16)
6	D	S, A, C, B, D	E(8), G1(14), G2(16)
7	E	S, A, C, B, D, E	G2(13), G1(14); G2(13) replaces G2(16)
8	G2	S, A, C, B, D, E, G2	

(e) Hill climbing

Note: Hill climbing does not use open and closed lists, instead the first neighboring state with a better heuristic than the current state is visited.

Step	Visited State	Evaluations
1	S	A(5) $<$ S(10)
2	A	C(3) $<$ A(5)
3	C	D(2) $<$ C(3)
4	D	B(8) $\not<$ D(2); G1(0) $<$ D(2)
5	G1	

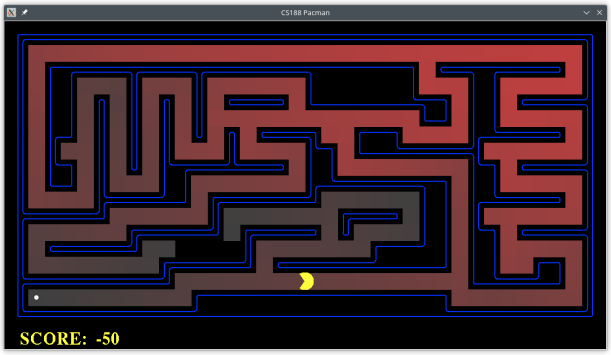
(f) Best-first search

Step	Visited State	Closed List	Open List
1			S(10)
2	S	S	A(5), B(8)
3	A	S, A	D(2), C(3), B(8)
4	D	S, A, D	G1(0), C(3), B(8)
5	G1	S, A, D, G1	C(3), B(8)

(g) Algorithm A

Step	Visited State	Closed List	Open List
1			S(10)
2	S	S	A(3+5=8), B(7+8=15)
3	A	S, A	C(3+1+3=7), D(3+6+2=11), B(15)
4	C	S, A, C	D(3+1+4+2=10), B(15)
5	D	S, A, C, D	G1(3+1+4+6+0=14), B(15)
6	G1	S, A, C, D, G1	

Question 3 *Help Pacman to find his food!*[1] In this question, your goal is to implement the depth-first search and breadth-first search algorithms shown in the lecture. To make it more interesting, we'll use an existing framework where you write generic search algorithms and run them in a Pacman simulator environment.



For the source code download and a list of files to edit, please refer to the supplemental lab page on Moodle.

(a) Finding a Fixed Food Dot using Depth First Search.
In the file `search.py`, find the function `depthFirstSearch`:

```
def depthFirstSearch(problem):
"""
Search the deepest nodes in the search tree first.

Your search algorithm needs to return a list of actions that reaches
the goal. Make sure to implement a graph search algorithm.
...
"*** YOUR CODE HERE ***"
```

One of the first things you'll probably want to do is to check if Pacman found the goal state:

```
if(problem.isGoalState(problem.getStartState())):
   return []
```

Remember that DFS uses a *stack* data structure. Make sure you use the data structure that come with the provided `util.py`; you can just use:

```
open = util.Stack()
```

and use the stack functions as you'd expect (`push, pop` etc.).

Read the code carefully to understand the data your function must return: Pacman expects a list of *actions* to perform (move left, right, up, down):

```
return actions
```

[1]Based on UC Berkeley's Pacman AI project, http://ai.berkeley.edu

You'll find possible actions when you create successor states from each state; use the commented out example debug code:

```
print("Start's successors:", problem.getSuccessors(problem.getStartState()))
```

to see these; thus, a legal result would be a list of actions like `['West', 'South', 'South', 'East', 'West']`. Make sure you return the *solution path* here (path from start to goal), not the *search path* (including all the states you visited while searching). Make sure you read the note on the lab page regarding where to put the *loop check* in your implementation.

There are many ways to implement the lecture DFS algorithm in Python, we show some ideas here. First, we have to think about representing the information we need to keep during the search (see the slide "Data Structures" from the lecture): At the very least, we need to keep track of the parent node of a given state and the action that brought us there. We could create a separate class for this, but a simpler solution is to manage this through a dictionary:

```
# Dictionary to keep track of actions leading to each state
actions = {}
actions[problem.getStartState()] = []
```

In this dictionary, the key is the state and the value is a list of actions leading to that state.

We have to initialize the open queue (stack data structure) with the starting node as well:

```
# Initialize the open stack with successors of the start state
open = util.Stack()
for successor in problem.getSuccessors(problem.getStartState()):
   state, action, _ = successor
   open.push((state, action, problem.getStartState()))
```

Note that we can ignore the third element in the tuple, since we're at the start state.

We have to initialize the closed set as well:

```
closed = set()
closed.add(problem.getStartState())
```

The main loop will search until `open` is empty:

```
while not open.isEmpty():
```

Then, we retrieve the first element from the stack:

```
   state, action, previous = open.pop()
```

If the state was already explored, skip it (loop check):

```
if state in closed:
    continue
```

Now we have to update the list of actions that brought us to the current state:

```
# Update the list of actions for the current state by appending
# the current action to the actions of the previous state
if state not in actions:
    actions[state] = actions[previous] + [action]
```

In the lecture slides, we just return `success` in case we found the goal, but here we need to return the `actions` that brought us from the start state to the goal state, which is then visualized by moving Pacman through the simulator:

```
if problem.isGoalState(state):
    return actions[state]
```

Then, we mark the current state as explored:

```
closed.add(state)
```

and push the successors of the current state to the open stack:

```
for successor in problem.getSuccessors(state):
    s_state, s_action, _ = successor
    open.push((s_state, s_action, state))
```

When you're done, you can test your implementation on the three mazes shown on the Moodle page. You should also run the unit tests to check your code for correctness using:

```
python autograder.py -q q1
```

If everything is correct, you'll receive a score of 3/3 (this score is just fyi, there is no submission for this lab exercise).

(b) Breadth First Search.
Like above, but now we search with BFS. Remember that the only difference to DFS is the data structure used to manage the open list (fringe). Test your code using `python autograder.py -q q2`

Well, the lazy solution is to copy the DFS code and change the data structure to a `Queue`. However, this approach duplicates a lot of code, making it harder to maintain, debug, or extend. A smarter solution would be to add a function that can be provisioned with the data structure needed:

```
def generalSearch(problem, data_structure)
```

and then call it using the required data structure:

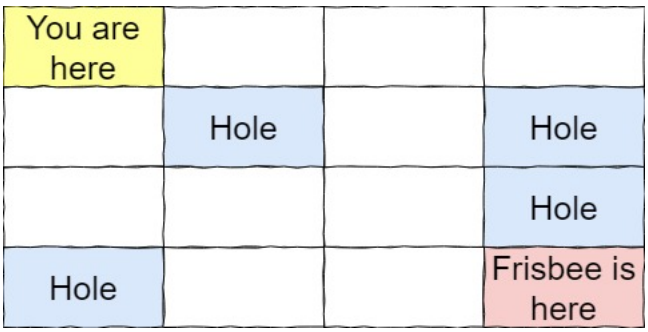
```python
def depthFirstSearch(problem):
    return generalSearch(problem, util.Stack())

def breadthFirstSearch(problem):
    return generalSearch(problem, util.Queue())
```

By doing this, we reduce code redundancy, make our codebase easier to maintain, and ensure that any changes made to the core logic of the search are automatically reflected in both BFS and DFS.

(c) Solving the 8-Puzzle.
But what about the 8-puzzle we discussed in the lecture? If you implemented your search algorithms correctly, they don't include anything specific about Pacman, so you can run them with the provided 8-puzzle code `python eightpuzzle.py` to find the solution for a random puzzle:

```
(comp6721) :~/COMP6721/Pacman/search> python eightpuzzle.py
A random puzzle:
-------------
| 5 | 4 | 1 |
-------------
| 3 | 7 | 2 |
-------------
| 6 |   | 8 |
-------------
BFS found a path of 11 moves: ['up', 'up', 'left', 'down', 'right',
  'up', 'right', 'down', 'left', 'left', 'up']
After 1 move: up
-------------
| 5 | 4 | 1 |
-------------
| 3 |   | 2 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
```

Question 4 *Bonus take-home exercise from OpenAI:*[2] Winter is here. You and your friends were tossing around a Frisbee at the park when you made a wild throw that left the Frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international Frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc as soon as you can. The surface is described using a rectangular grid like the figure below:



(a) What are the initial and goal state?
Let a 4×4 matrix represent the above grid. The position of each cell in the grid can then be addressed by the indices of the elements of the matrix. Hence we have:

- *Initial state:* $(1, 1)$
- *Goal state:* $(4, 4)$

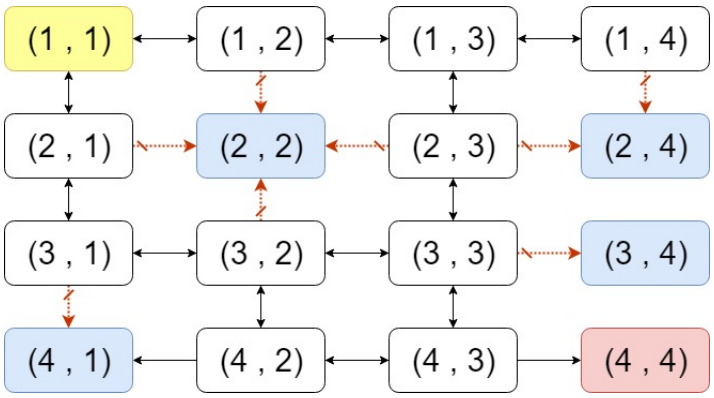
(b) What are the illegal states?
The illegal states are the holes in the grid with the following positions:

Illegal State 1: $(2, 2)$
Illegal State 2: $(2, 4)$
Illegal State 3: $(3, 4)$
Illegal State 4: $(4, 1)$

(c) What are the possible actions and what should be their costs?
The possible actions in a grid world are moving left, right, up, and down. Since we would like to reach the goal state as soon as possible (i.e., minimizing the number of actions), then we can assign a constant uniform cost for each action, for example a cost of 1*. The algorithm aims to minimize the cost while reaching the goal state.*

(d) Draw the state space for this problem.
The state space is shown in the figure below, where the initial state is colored yellow, the goal state is colored red, and illegal states are colored blue.

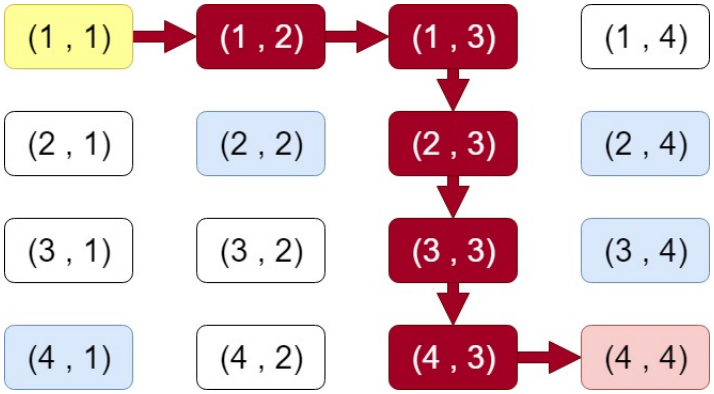
[2] https://www.gymlibrary.dev/environments/toy_text/frozen_lake/



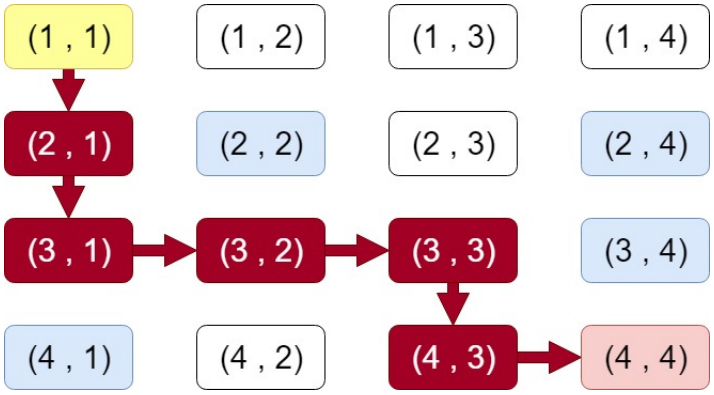
(e) Find a series of moves to solve this problem.

Any of the following series of moves constitute a possible solution for this problem:

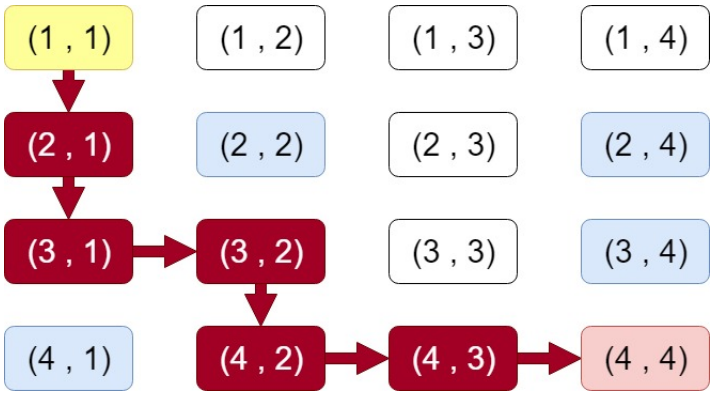
Path 1: $(1,1) \to (1,2) \to (1,3) \to (2,3) \to (3,3) \to (4,3) \to (4,4)$



Path 2: $(1,1) \to (2,1) \to (3,1) \to (3,2) \to (3,3) \to (4,3) \to (4,4)$



Path 3: $(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2) \rightarrow (4,3) \rightarrow (4,4)$



(f) Perform the following search strategies, show the states visited, along with the open and closed lists at each step.

I Breadth-first search

The detailed execution of breadth-first search is listed below. A pictorial representation of the progress of this algorithm can be seen in Figure 1.

Step 1
- *Visited:*
- *open:* $\{(1,1)\}$
- *closed:* $\emptyset$

Step 2
- *Visited:* $(1,1)$
- *open:* $\{(1,2),(2,1)\}$
- *closed:* $\{(1,1)\}$

Step 3
- *Visited:* $(1,2)$
- *open:* $\{(2,1),(1,3)\}$
- *closed:* $\{(1,2),(1,1)\}$

Step 4
- *Visited:* $(2,1)$
- *open:* $\{(1,3),(3,1)\}$
- *closed:* $\{(2,1),(1,2),(1,1)\}$

Step 5
- *Visited:* $(1,3)$
- *open:* $\{(3,1),(1,4),(2,3)\}$
- *closed:* $\{(1,3),(2,1),(1,2),(1,1)\}$

Step 6
- *Visited:* $(3,1)$
- *open:* $\{(1,4),(2,3),(3,2)\}$
- *closed:* $\{(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 7
- *Visited:* $(1,4)$
- *open:* $\{(2,3),(3,2)\}$
- *closed:* $\{(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 8
- *Visited:* $(2,3)$

- *open:* $\{(3,2),(3,3)\}$
- *closed:* $\{(2,3),(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 9
- *Visited:* $(3,2)$
- *open:* $\{(3,3),(4,2)\}$
- *closed:* $\{(3,2),(2,3),(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 10
- *Visited:* $(3,3)$
- *open:* $\{(4,2),(4,3)\}$
- *closed:* $\{(3,3),(3,2),(2,3),(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 11
- *Visited:* $(4,2)$
- *open:* $\{(4,3)\}$
- *closed:* $\{(4,2),(3,3),(3,2),(2,3),(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 12
- *Visited:* $(4,3)$
- *open:* $\{(4,4)\}$
- *closed:* $\{(4,3),(4,2),(3,3),(3,2),(2,3),(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$

Step 13
- *Visited:* $(4,4)$
- *open:* $\{\}$
- *closed:*

$$\{(4,4),(4,3),(4,2),(3,3),(3,2),(2,3),(1,4),(3,1),(1,3),(2,1),(1,2),(1,1)\}$$

- *The goal state is observed at this point and the search has terminated.*

II Uniform cost search

Since we have assumed that all step cost are equal, then the uniform-cost search and breadth-first search will have the same result.

III Depth-first search

The detailed execution of depth-first search is listed below. A pictorial representation of the progress of this algorithm can be seen in Figure 2.

Step 1
- *Visited:*
- *open:* $\{(1,1)\}$
- *closed:* $\emptyset$

Step 2
- *Visited:* $(1,1)$
- *open:* $\{(1,2),(2,1)\}$
- *closed:* $\{(1,1)\}$

Step 3
- *Visited:* $(1,2)$
- *open:* $\{(1,3),(2,1)\}$
- *closed:* $\{(1,2),(1,1)\}$

Step 4
- *Visited:* $(1,3)$

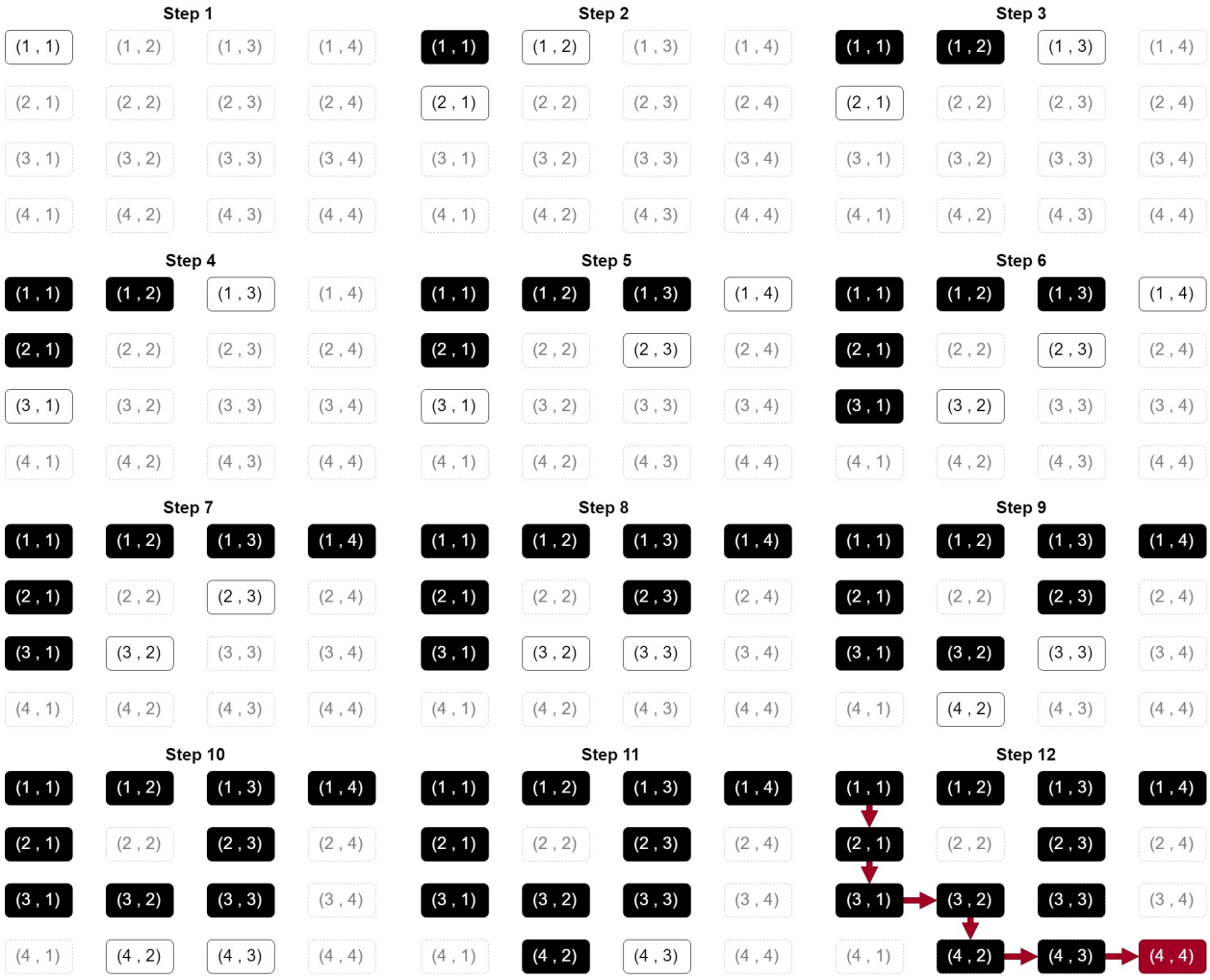


Figure 1: Progress of breadth-first search algorithm is depicted above. The cells with solid white background represent the open set. The cells with solid black background represent the closed set and the cell with solid red background is the goal state.

- *open:* $\{(1,4),(2,3),(2,1)\}$
- *closed:* $\{(1,3),(1,2),(1,1)\}$

Step 5
- *Visited:* $(1,4)$
- *open:* $\{(2,3),(2,1)\}$
- *closed:* $\{(1,4),(1,3),(1,2),(1,1)\}$

Step 6
- *Visited:* $(2,3)$
- *open:* $\{(3,3),(2,1)\}$
- *closed:* $\{(2,3),(1,4),(1,3),(1,2),(1,1)\}$

Step 7
- *Visited:* $(3,3)$
- *open:* $\{(3,2),(4,3),(2,1)\}$
- *closed:* $\{(3,3),(2,3),(1,4),(1,3),(1,2),(1,1)\}$

Step 8
- *Visited:* $(3,2)$
- *open:* $\{(3,1),(4,2),(4,3),(2,1)\}$
- *closed:* $\{(3,2),(3,3),(2,3),(1,4),(1,3),(1,2),(1,1)\}$

Step 9
- *Visited:* $(3,1)$
- *open:* $\{(4,2),(4,3),(2,1)\}$
- *closed:* $\{(3,1),(3,2),(3,3),(2,3),(1,4),(1,3),(1,2),(1,1)\}$

Step 10
- *Visited:* $(4,2)$
- *open:* $\{(4,3),(2,1)\}$
- *closed:* $\{(4,2),(3,1),(3,2),(3,3),(2,3),(1,4),(1,3),(1,2),(1,1)\}$

Step 11
- *Visited:* $(4,3)$
- *open:* $\{(4,4),(2,1)\}$
- *closed:* $\{(4,3),(4,2),(3,1),(3,2),(3,3),(2,3),(1,4),(1,3),(1,2),(1,1)\}$

Step 12
- *Visited:* $(4,4)$
- *open:* $\{(2,1)\}$
- *closed:* $\{(4,4),(4,3),(4,2),(3,1),(3,2),(3,3),(2,3),(1,4),(1,3),(1,2),(1,1)\}$
- *The goal state is observed at this point and the search has terminated.*

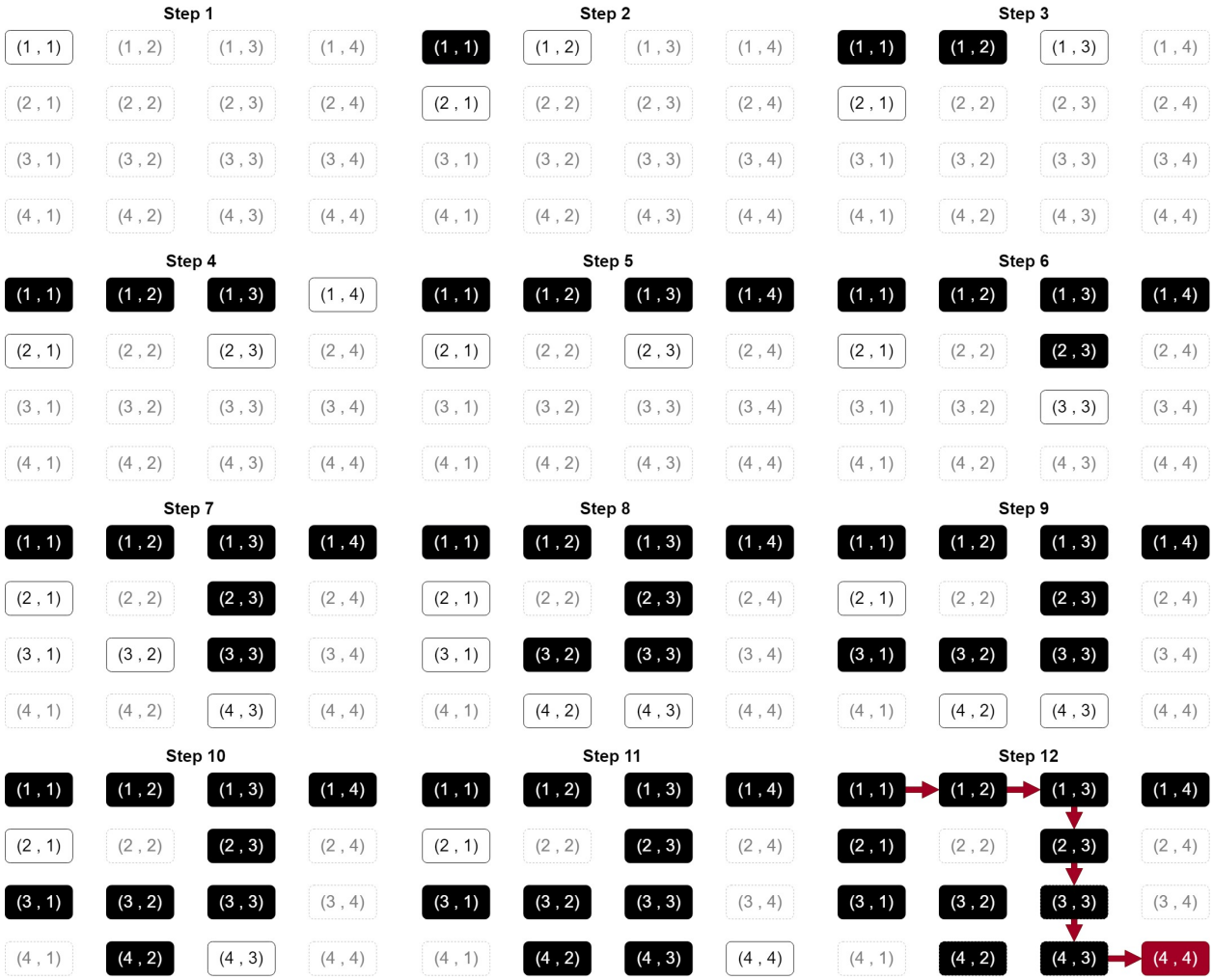


Figure 2: Progress of depth-first search algorithm is depicted above. The cells with solid white background represent the open set. The cells with solid black background represent the closed set and the cell with solid red background is the goal state.