

# CS 188 Assignment 2: Part 1 Sample Solutions

CS188 Staff

## 1 Search algorithms

### 1.1 Structure

In the textbook and its online code repository, each search algorithm can be succinctly formulated as an instance of the general tree-search and graph-search algorithms. For example, the only difference between depth-first and breadth-first search is the data structure used to store unexplored nodes: a LIFO stack versus a FIFO queue, respectively. The choice can be passed in as an argument to the general search algorithms. The uniform-cost and  $A^*$  search algorithms are instances of best-first search, which is just tree-search or graph-search using an evaluation function to select nodes for expansion: use the path-cost  $g(n)$  for UCS, or the heuristic estimate  $g(n) + h(n)$  for  $A^*$ . Given Python's support for functional programming, it would also be fairly easy to pass in such an evaluation function as an argument to the general search algorithms.

Despite the elegance of exposition which might be gained by abstractly linking these various algorithms, we will present them more independently. All will be graph-search algorithms, using a *closed list* to avoid visiting repeated states. Although Questions 4 and 5 did not specify that uniform-cost and  $A^*$  search should be graph-search algorithms, you would likely have found that a tree-search implementation would be very inefficient for exploring a large state space with reversible actions.

In addition to the `Stack`, `Queue`, and `PriorityQueue` data structures provided in `util.py`, it is helpful to define some helpers (in `search.py`, for example): a `Node` data structure for bookkeeping, and a `tracepath` function that follows node parent pointers to derive a path from the root of the search tree.

```
class Node:
    def __init__(self, state, parent, pathcost):
        self.state = state
        self.parent = parent
        self.pathcost = pathcost

def tracepath(node):
    if node.parent:
        return tracepath(node.parent) + [node.state]
    else:
        return [node.state]
```

Alternatively, you could define a node as a tuple, such as `(state, parent, pathcost)`. Also, you could store parent pointers in a separate dictionary, rather than inside the node data structure.

### 1.2 Depth-first search

```
class DepthFirstSearchAgent(SearchAgent):
    def solve(self, searchProblem):
        fringe = util.Stack()
        fringe.push(Node(searchProblem.getStartState(), None, 0.0))
        closed = set() # faster than list for membership test

        while not fringe.isEmpty():
```

```

node = fringe.pop()
if searchProblem.isGoalState(node.state):
    return (tracepath(node), node.pathcost)
elif node.state not in closed:
    closed.add(node.state)
    for (nextstate, nextcost) in searchProblem.getSuccessors(node.state):
        fringe.push(Node(nextstate, node, node.pathcost + nextcost))
return (None, 0.0)

```

```

-----
|                                     |
|          #####                     |
|          #####xxxxE               |
|                #####              |
|          Sxxxxxxx#####           |
|                x#####            |
|                x#####            |
|                x#####            |
|                x#####            |
|                x#####            |
|                x#####            |
|                xxxxxxxx           |
|                                     |
-----

```

x - cell used in solution  
o - cell expanded during search

```

-----
Solution cost: 61.0
Number of nodes expanded: 61
Number of unique nodes expanded: 61

```

Note that the DFS agent quickly finds a solution, which in this case is conveniently constructed from every cell expanded during the search. However, this is not the lowest-cost solution because the search is just returning with the first solution it encounters, without exploring potentially shorter paths to the goal.

### 1.3 Breadth-first search

```

class BreadthFirstSearchAgent(SearchAgent):
    def solve(self, searchProblem):
        fringe = util.Queue()
        fringe.enqueue(Node(searchProblem.getStartState(), None, 0.0))
        closed = set() # faster than list for membership test

        while not fringe.isEmpty():
            node = fringe.dequeue()
            if searchProblem.isGoalState(node.state):
                return (tracepath(node), node.pathcost)
            elif node.state not in closed:
                closed.add(node.state)
                for (nextstate, nextcost) in searchProblem.getSuccessors(node.state):
                    fringe.enqueue(Node(nextstate, node, node.pathcost + nextcost))
        return (None, 0.0)

```

```

-----
|oooooooooooooooooooooooooooooooooo|
|oooooooooooo#####ooo |

```

```

|ooooooooo#####xxx|E|
|oooooooooooooooooooo###xooo|
|ooooooooooooooooSoooooo###xooo|
|ooooooooooooooooxoooooooo###xooo|
|ooooooooooooooooxoooooooo###xooo|
|ooooooooooooooooxoooooooo###xooo|
|ooooooooooooooooxoooooooo###xooo|
|ooooooooooooooooxoooooooo###xooo|
|ooooooooooooooooxoooooooo###xooo|
|ooooooooooooooooxxxxxxxxxxxxooo|
-----
x - cell used in solution
o - cell expanded during search
-----
Solution cost: 29.0
Number of nodes expanded: 277
Number of unique nodes expanded: 277

```

```

-----
|###ooooooo|
|###ooooooo~~~|
|###ooooooo~~|
|###ooooooo~|
|###ooooooo|
|#SxxxxxxxxxxxxxE|
|###ooooooo|
|###ooooooo~|
|###ooooooo~~|
|###ooooooo~~~|
|###ooooooo|
-----
x - cell used in solution
o - cell expanded during search
-----
Solution cost: 70.0
Number of nodes expanded: 143
Number of unique nodes expanded: 143

```

The BFS agent is able to find a solution path that has the shortest length. If all step costs are equal, this will also be a lowest-cost solution, as in `maze1`. For `maze2`, with higher step costs associated with traversing water cells, the shortest path is not the cheapest. Note that this agent explores a large number of cells prior to reaching the goal; this is because a BFS search will examine all paths that are shorter than the optimal solution.

## 1.4 Uniform-cost search

```

class UniformCostSearchAgent(SearchAgent):
    def solve(self, searchProblem):
        fringe = util.PriorityQueue()
        fringe.setPriority(Node(searchProblem.getStartState(), None, 0.0), 0.0)
        closed = set() # faster than list for membership test

        while not fringe.isEmpty():
            node = fringe.dequeue()
            if searchProblem.isGoalState(node.state):

```

```

    return (tracepath(node), node.pathcost)
elif node.state not in closed:
    closed.add(node.state)
    for (nextstate, nextcost) in searchProblem.getSuccessors(node.state):
        g = node.pathcost + nextcost
        fringe.setPriority(Node(nextstate, node, g), g)
return (None, 0.0)

```

```

-----
|#####xxxxxxxxxxxxxx|
|#####xooooooooooooo|x|
|#####xooooooooo~~~~x|
|#####xooooo~~~~~~x|
|#####xooooo~~~~~~x|
|#Sxxxooooo~~~~~~E|
|#####ooooo~~~~~~o|
|#####ooooo~~~~~~o|
|#####ooooooooo~~~o|
|#####ooooooooooooo|
|#####ooooooooooooo|
-----

```

x - cell used in solution  
o - cell expanded during search

```

-----
Solution cost: 28.0
Number of nodes expanded: 119
Number of unique nodes expanded: 119

```

```

-----
|~~~~oooooooo~~~~|
|~~~oooooooooooo~~~~|
|~ooooooooooooo~~~~|
|~ooooooooooooo~~~~|
|#####oooooooo~~~~|
|oooooooooooo#####|
|oooooSxxxoooo#~~~~|
|#####o#x#####oo~~~~|
|ooooo#o#xxxxoooo~~~~|
|o#o#o#####x###~~~~|
|o#ooooooooxxx#~~~~|
|oooo###x###~~~~|
|ooooooooxxxxxxxxxE|
-----

```

x - cell used in solution  
o - cell expanded during search

```

-----
Solution cost: 24.0
Number of nodes expanded: 147
Number of unique nodes expanded: 147

```

The UCS agent is able to find the lowest-cost solution for both mazes. Its optimality is due to its exhaustive search over all paths that are cheaper than the lowest-cost solution. For `maze3`, note that many cells in the upper part of the maze are expanded to search on paths that fruitlessly extend farther away from the goal.



## 2 Eight-puzzle search problem formulation

In the class `EightPuzzleSearchProblem`, fill in the following methods:

```
def getStartState(self):
    return self.puzzle

def isGoalState(self, state):
    return state.isGoal()

def getSuccessors(self, state):
    return [(state.result(move), 1.0) for move in state.legalMoves()]
```

The BFS agent will always find an optimal solution to the eight-puzzle because all step costs (tile moves) have a cost of one, so the shortest solution has the lowest cost. For puzzles with short solutions, BFS is fairly good; however, note that BFS search can be very slow if the solution is long (as in `puzzle1`).

## 3 Eight-puzzle heuristics

For the eight-puzzle heuristics, many students treated the blank space as if it were a tile. Note that this mistake makes the heuristic inadmissible, as it could sometimes overestimate the true optimal solution cost.

### 3.1 Misplaced tiles

```
def misplacedTiles(state, eightPuzzleSearchProblem):
    current = 0
    total = 0.0
    for row in range( 3 ):
        for col in range( 3 ):
            if current and current != state.cells[row][col]:
                total += 1.0
                current += 1
    return total
```

### 3.2 Manhattan Distance

```
def manhattanDistance(state, eightPuzzleSearchProblem):
    goalCoordinates = [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)]
    total = 0.0
    for row in range( 3 ):
        for col in range( 3 ):
            actual = state.cells[row][col]
            goalrow, goalcol = goalCoordinates[actual]
            dist = abs(goalrow-row) + abs(goalcol-col)
            if actual:
                total += dist
    return total
```

Notice that  $A^*$  search performs better (faster, expanding fewer nodes) when applied with the Manhattan distance heuristic. If one admissible heuristic *dominates* another (Manhattan distance is always greater than or equal to the number of misplaced tiles), then it will always be more efficient (AIMA, pg. 106).

### 3.3 Gaschnig's heuristic

```
def gaschnig(state, eightPuzzleSearchProblem):
    total = 0.0
    currentCoordinates = range(9)
    for row in range( 3 ):
        for col in range( 3 ):
            currentCoordinates[state.cells[row][col]] = (row,col)
    goalCoordinates = [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)]
    while currentCoordinates != goalCoordinates:
        blankLocation = currentCoordinates[0]
        if blankLocation != (0,0):
            tiletomove = goalCoordinates.index(blankLocation)
            currentCoordinates[0] = currentCoordinates[tiletomove]
            currentCoordinates[tiletomove] = blankLocation
        else:
            tiletomove = 1
            while currentCoordinates[tiletomove] == goalCoordinates[tiletomove]:
                tiletomove += 1
            currentCoordinates[0] = currentCoordinates[tiletomove]
            currentCoordinates[tiletomove] = blankLocation
        total += 1.0
    return total
```

Gaschnig's heuristic can be computed by the following procedure: if the location of the blank space should be occupied by a tile, move that tile to the blank space; otherwise, move any misplaced tile to the blank space. Note that the misplaced-tiles problem is a relaxation of Gaschnig's problem relaxation – thus Gaschnig's heuristic dominates the misplaced tiles heuristic.

Comparing performance on `puzzle1`, the misplaced tiles heuristic expands 15352 nodes, Gaschnig's heuristic expands 13227, and the Manhattan distance heuristic expands 894. By taking the highest estimate from a set of admissible heuristics, we can easily write a heuristic function that is always at least as good as the Manhattan and Gaschnig's heuristic (expanding 892 nodes for `puzzle1`):

```
def max_heuristic(state, eightPuzzleSearchProblem):
    return max([manhattanDistance(state,None), gaschnig(state, None)])
```