COMP3702
Artificial Intelligence

Tutorial 2: Uninformed Search

Semester 2, 2022

The University of Queensland
School of Information Technology and Electrical Engineering

Problem description

The 8-puzzle problem contains 9 cells stored in a grid. This can be represented as a string of 9 characters, with an '_' where the empty space is:

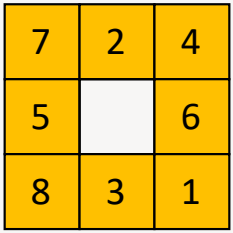

Figure 1: Example 8-puzzle state, which can be represented as the string 7245_6831

Exercise 2.1

Exercise 2.1 — Task

Task:
Please implement the Breadth First Search (BFS) and Depth First Search (DFS) algorithms to solve the 8-puzzle problem.

Exercise 2.1 — Implement BFS and DFS in 8-puzzle

Action Space:

- The action space only consists of 4 actions that are, move the blank space: {`up`, `down`, `left`, `right`}.
- But some actions are invalid, e.g. at the boundaries.
- One way to identify invalid actions programmatically is to use the row and column of the blank space, using 1-indexing (i.e. the first element in the string will be at index 1):
  - How might we determine that the blank space is in the top row, so that '`up`' is invalid? Or the bottom row?
  - How can we identify if '`left`' and '`right`' are invalid moves?

Exercise 2.1 — Implement BFS and DFS in 8-puzzle

State Space:

- There are a total of 9! states in the 8-puzzle problem (= 362,880). It would take a long time to enumerate and store each one of them.
- Instead, the search tree can be created as the search is performed.
- A node (vertex) in a tree can be represented as in Figure 2.

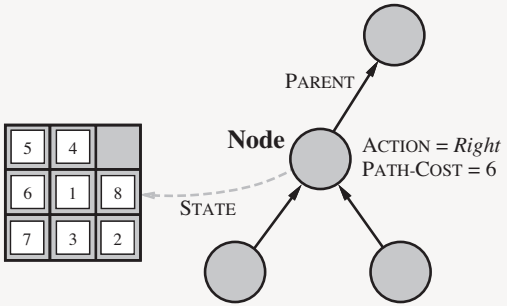

Figure 2: Example of a node which represents its STATE, stores its PARENT, the ACTION that was performed from the parent to arrive at the node, and the PATH-COST to arrive at that node (Source: R&N textbook)

Exercise 2.1 — Breadth-first search

- In BFS, the successors of a node get added to a *First-In First-Out* (FIFO) queue.
- This can be implemented in Python using `container.append(next_node)`, and `container.pop(0)`.
- The successors will be the the neighbours of the current state. For each state there can be a maximum of 4 neighbours.
- E.g. `1348627_5` has 3 neighbours: `(1348_2765, 134862_75, 13486275_)`
- `1348627_5` would be the first state to get constructed and added to the queue:
  1. We check if it is goal node,
  2. it gets marked as expanded, and then
  3. its neighbours get constructed and added to the queue.

Exercise 2.1 — Breadth-first search

The pseudocode for BFS is shown below (Source: R&N textbook)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Alternatively, see P&M's discussion of BFS (Ch 3.5.1)

Exercise 2.1 — Depth-first search

- DFS uses the same procedure as BFS, except that the successors of a node get added to a *Last-In First-Out* (LIFO) stack.
- This can be implemented in Python using `container.append(next_node)`, and `container.pop()`
- A LIFO queue means that the most recently generated nodes are expanded first.
- This must be the deepest unexpanded node because it is one deeper than its parent – which, in turn, was the deepest unexpanded node when it was selected.
- (An alternative implementation is to implement depth-first search with a recursive function that calls itself on each of its children in turn.)
- See also P&M's discussion of DFS in (Ch 3.5.2)

Exercise 2.2

Exercise 2.2 — Task

Exercise 2.2. Please test your implementation of BFS and DFS with the following initial and goal states:

- From 1348627_5 to 1238_4765
- From 281_43765 to 1238_4765
- From 281463_75 to 1238_4765

In each test for each algorithm, please output:

(a) The sequence of actions to move from the initial to the goal state.

(b) The number of steps to solve the problem.

(c) The time your implementation takes to solve the problem.

Based on the results of (b) and (c), please explain the relation between the time and the number of steps that each algorithm takes to solve the problem. Is the relation you found in-line with the complexity results of the algorithms? If they are not in-line, please find out and explain why.

Exercise 2.2 — Run your puzzle solvers

Generate the Sequence of Actions, Number of Steps, and Time taken.

Example values (with goal-check on generation):

- startState = 1348627_5; goalState = 1238_4765
  BFS: steps = 6, visited = 22, time = 0.006s;
  DFS: steps = 836, visited = 854, time = 0.0172s
- startState = 281_43765; goalState = 1238_4765
  BFS: steps = 10, visited = 219; time = 0.03s;
  DFS: steps = 69434, visited = 72441, time = 253.44s
- startState = 281463_75; goalState = 1238_4765
  BFS: steps = 13, visited = 1120; time = 0.266s
  DFS: steps = 82141, visited = 86490; time = 361.0s

Discuss these results: Which takes more time and which uses more space (i.e. requires more nodes to be stored)? Why might this be the case for 8-puzzle?

Exercise 2.3

Exercise 2.3 — Task

Exercise 2.3. Suppose there is no solution to the 8-puzzle problem, i.e., there is no way the 8-puzzle can move from the given initial to goal configurations.

(a) Will BFS be able to output the right answer (i.e., no solution)? How about DFS?

(b) We can actually identify if a given 8-puzzle problem is solvable or not without search, which is via the concept of parity (see the appended material on inversion and parity). Please implement parity computation, so that your program can identify if a problem is solvable without performing search.

Exercise 2.3 — (a) Solvability

Suppose there is no solution to the 8-puzzle problem, i.e., there is no way the 8-puzzle can move from the given initial to goal configurations.

(a) Will BFS be able to output the right answer (i.e., no solution)? How about DFS?

Exercise 2.3 — (b) Parity check

Parity can be calculated by finding the number of inversions of a state mod 2.

- The number of inversions of tile-i ($N(s, i)$) is the number of tiles that appear after tile-i that have a smaller value than tile-i.
- The parity is the total sum of the inversions of each tile, mod 2
  Parity $= \sum_{i=1}^{n} N(s, i) \bmod 2$

Exercise 2.4

Exercise 2.4 — Task

Exercise 2.4. Imagine the tiles have been tampered with and it is no longer equally easy to move the tiles in each direction. Suppose that the cost for 'moving the empty tile' is as follows:

cost = {'up':1, 'down':2, 'left':3, 'right':4}

(a) Will Uniform Cost search find the least cost path for this problem?

(b) Compare Uniform Cost Search and BFS when the weight of all edges in the state graph are equal.

Exercise 1.4 — Uniform cost search

- Uniform Cost Search (UCS) expands the node with least path cost. In P&M it is called Lowest-Cost-First Search (see Ch 3.5.4), which is a better name, but much less widely used.
- UCS is like BFS, in that the successors of a node get added to a queue.
- However, UCS uses a *priority queue* ordered by the arc cost function.
- This can be implemented in python using a priority queue object `pq`, where `pq.push(key, value)` inserts (key, value) into the queue, and `pq.pop()` returns the key with the lowest value and removes it from the queue.

(a) Will UCS find the least cost path for this problem? Yes, but why?

(b) When costs are all equal, BFS and UCS should be identical, but this can vary depending on implementation details.