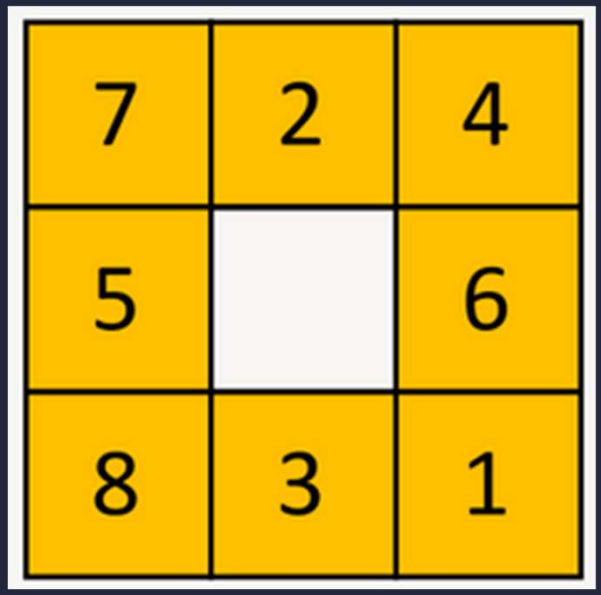
COMP3702 Tutorial 2

Matt Choy | matthew.choy@uq.edu.au

Eight Puzzle

- The Eight Puzzle is a problem containing 8 cells stored in a grid.
- This can be represented as a string of characters, with the blank space denoted by '_'.
- For example, the 8-puzzle to the right can be represented by the string "7245_6831"



Agent Design Problem Components

The following components are required to solve an agent design problem

1. Action Space (A) The set of all possible actions the agent can perform. This is sometimes called the *action set* in the discrete case. An action is denoted $a \in A$.
2. Percept Space (P) The set of all possible things an agent can perceive.
3. State Space (S) The set of all possible configurations of the world the agent is operating in. This is sometimes called the *set of states* in discrete state systems. A state is denoted as $s \in S$
4. World Dynamic / Transfer Function ($T: S \times A \rightarrow S'$) A function that specifies how the world changes when the agent performs actions in it. This maps a 2-tuple of states and actions to a single resulting state (this function specifies how we move from one state to the next.
5. Percept Function ($\mathbb{Z} : S \rightarrow P$) A function that maps a world state to a perception. Note that for fully observable problems, the percept function is the identity function $I(x)$
6. Utility Function ($U: S \rightarrow \mathbb{R}$) A function that maps a state (or sequence of states) to a real number. This indicates how desirable it is for an agent to be in that state (or set of states). $U(s) = cost\ or\ reward$

Search Algorithms

- A generic search algorithm structure is given by:

```
1.  Container = [SearchNode(init_state)]
2.  While Container is not Empty:
3.    Current_Node ← Choose Node from Container (and remove it)
4.    If Current_Node.state is the goal state:
5.      Return Current_Node.actions
6.    Successors ← Current_Node.get_successors()
7.    For s in Successors:
8.      If s not visited (or s visited at higher cost than current cost)
9.        Add SearchNode(s) to the Container
10.       Add s to the visited set.
```

Search Algorithms

- A generic search algorithm structure is given by:
  - What is the difference between depth-first and breadth-first search?

```
1.  Container = [SearchNode(init_state)]
2.  While Container is not Empty:
3.    Current_Node ← Choose Node from Container (and remove it)
4.    If Current_Node.state is the goal state:
5.      Return Current_Node.actions
6.    Successors ← Current_Node.get_successors()
7.    For s in Successors:
8.      If s not visited (or s visited at higher cost than current cost)
9.        Add SearchNode(s) to the Container
10.       Add s to the visited set.
```

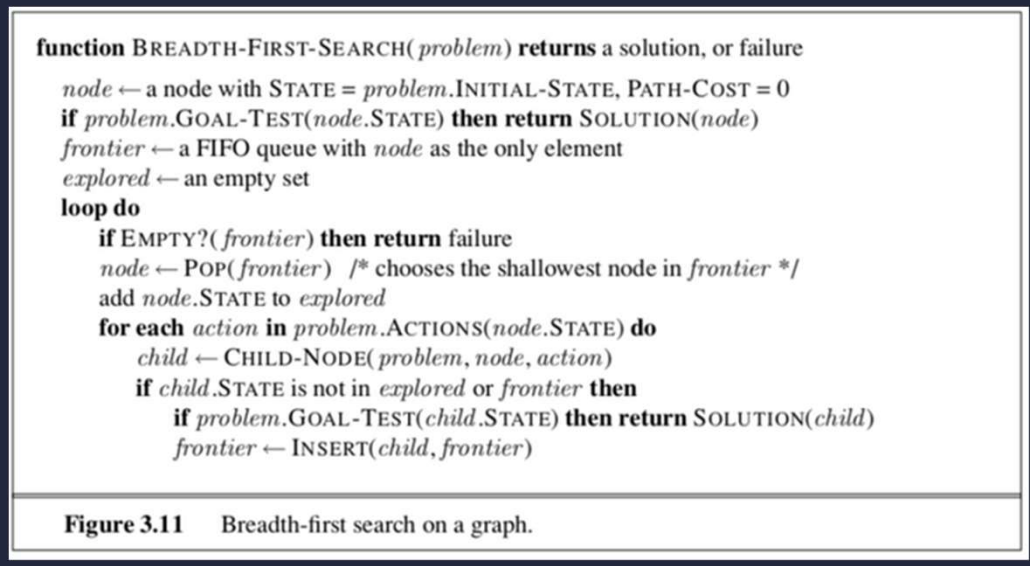
Search Algorithms

- A generic search algorithm structure is given by:
- For Breadth-First Search, choose the first or oldest node in the container (frontier) – First In, First Out stack.
- For Depth-First Search, choose the last or newest node in the container (frontier) – First In, Last Out stack

```
1.  Container = [SearchNode(init_state)]
2.  While Container is not Empty:
3.    Current_Node ← Choose Node from Container (and remove it)
4.    If Current_Node.state is the goal state:
5.      Return Current_Node.actions
6.    Successors ← Current_Node.get_successors()
7.    For s in Successors:
8.      If s not visited (or s visited at higher cost than current cost)
9.        Add SearchNode(s) to the Container
10.       Add s to the visited set.
```

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()



```
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.

Depth-First Search

- DFS uses the same general procedure as BFS, except that successors of a Last-In, First-Out (LIFO) stack
- A LIFO queue means that the most recently generated nodes are expanded first
- This must mean that the deepest unexpanded node is visited because it is one deeper than its parent – which in turn was the deepest unexpanded node when it was selected.
- You can alternatively implement this using Recursion.

Exercise 2.1

Exercise 2.1. Please implement the Breadth First Search (BFS) and Depth First Search (DFS) algorithms to solve the 8-puzzle problem.

Agent Design Problem Components

The following components are required to solve an agent design problem

1. Action Space (A) The set of all possible actions the agent can perform. This is sometimes called the *action set* in the discrete case. An action is denoted $a \in A$.
2. Percept Space (P) The set of all possible things an agent can perceive.
3. State Space (S) The set of all possible configurations of the world the agent is operating in. This is sometimes called the *set of states* in discrete state systems. A state is denoted as $s \in S$
4. World Dynamic / Transfer Function ($T: S \times A \rightarrow S'$) A function that specifies how the world changes when the agent performs actions in it. This maps a 2-tuple of states and actions to a single resulting state (this function specifies how we move from one state to the next.
5. Percept Function ($\mathbb{Z} : S \rightarrow P$) A function that maps a world state to a perception. Note that for fully observable problems, the percept function is the identity function $I(x)$
6. Utility Function ($U: S \rightarrow \mathbb{R}$) A function that maps a state (or sequence of states) to a real number. This indicates how desirable it is for an agent to be in that state (or set of states). $U(s) = cost\ or\ reward$

Exercise 2.1

Exercise 2.1. Please implement the Breadth First Search (BFS) and Depth First Search (DFS) algorithms to solve the 8-puzzle problem.

- REMEMBER: The difference between BFS and DFS is the order in which nodes are expanded.
- Implement a data structure to represent the state (and nodes of the graph)
- Implement BFS and DFS methods

```
1.  Container = [SearchNode(init_state)]
2.  While Container is not Empty:
3.    Current_Node ← Choose Node from Container (and remove it)
4.    If Current_Node.state is the goal state:
5.      Return Current_Node.actions
6.    Successors ← Current_Node.get_successors()
7.    For s in Successors:
8.      If s not visited (or s visited at higher cost than current cost)
9.        Add SearchNode(s) to the Container
10.       Add s to the visited set.
```

Exercise 2.1 Solutions

Exercise 2.1. Please implement the Breadth First Search (BFS) and Depth First Search (DFS) algorithms to solve the 8-puzzle problem.

Exercise 2.1

- First, define the agent design components:
  - Action Space(A)
  - Percept Space(P)
  - State Space(S)
  - Transition Function (T : S x A -> S’)
  - Utility Function (U: S -> R)

Exercise 2.1

First, define the agent design components:

- State Space (S)

Exercise 2.1

First, define the agent design components:

- State Space (S) When designing a Python solution for encoding our state space, we need to consider components of the state that we need to store to ensure that two states are in fact unique:
  - Grid Layout What are the numbers on the grid, and what positions are they in?
    - We will treat this as a list, although there are many ways we can model this

Exercise 2.1

First, define the agent design components:

- State Space (S) When designing a Python solution for encoding our state space, we need to consider components of the state that we need to store to ensure that two states are in fact unique:
  - Grid Layout What are the numbers on the grid, and what positions are they in?
    - We will treat this as a list, although there are many ways we can model this
- Percept Space (P)

Exercise 2.1

First, define the agent design components:

- State Space (S) When designing a Python solution for encoding our state space, we need to consider components of the state that we need to store to ensure that two states are in fact unique:
  - Grid Layout What are the numbers on the grid, and what positions are they in?
    - We will treat this as a list, although there are many ways we can model this
- Percept Space (P) Since the problem is fully observable, the percept space is the same as the state space and therefore, we don't need to do anything special when considering it.

Exercise 2.1

- Action Space (A)

Exercise 2.1

- Action Space (A) In the 8-Puzzle, there are four actions that we can take:
  - { Left, Right, Up, Down }
  - These four actions make up the action space, although these actions are not always possible in every single state – we will model this behaviour in the transition function

Exercise 2.1

- Action Space (A) In the 8-Puzzle, there are four actions that we can take:
  - { Left, Right, Up, Down }
  - These four actions make up the action space, although these actions are not always possible in every single state – we will model this behaviour in the transition function
- Transition Function (T: S x A -> S')

Exercise 2.1

- Action Space (A) In the 8-Puzzle, there are four actions that we can take:
  - { Left, Right, Up, Down }
  - These four actions make up the action space, although these actions are not always possible in every single state – we will model this behaviour in the transition function
- Transition Function (T: S x A -> S') The transition function is the core of this problem – given a state an and action, we need a way to determine the next state.

Exercise 2.1

- Reward Function (U(S) -> R)

Exercise 2.1

- Reward Function (U(S) -> R) We want to reward the system or agent for finding the goal state, and do nothing otherwise:

$$U(s) = \begin{cases} 1 & \text{goal state} \\ 0 & \text{else} \end{cases}$$

Exercise 2.1 Code Solution

Exercise 2.2

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.

Exercise 2.2

BFS takes a relatively small number of steps as compared to DFS

```
Matt@DESKTOP-A0O5OOM MINGW64 ~/Documents/UQ/COMP3702-Tutoring/teaching/t2 (main)
$ python t2-q2.py
BFS: time = 0.02931591510772705 seconds, #actions = 12, #steps = 1243, actions = [1, 2, 0, 2, 1, 1, 3, 0, 0, 2, 1, 3]
DFS: time = 13.347254037857056 seconds, #actions = 27962, #steps = 29334
(base)
```

Exercise 2.2

- 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

BFS takes a relatively small number of steps as compared to DFS

BFS prioritises an optional number of solutions as it searches in an increasing radius from the start state.

DFS is not guaranteed to be optimal - searches to greater depths first.

Space complexity is not measured

- For BFS O(b^d) which is the # of nodes expanded
- For DFS O(bm) which is proportional to the maximum height of the tree.

Exercise 2.3

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 2.3a

BFS performs a complete search of the entire tree, given a finite branching factor.

- BFS will search all possible combinations of paths within the search space and eventually run out.
- Once we have searched the entire search space without finding a solution, we can conclude that no solution exists.
  - Therefore, BFS will output the right answer in this case.

DFS can also perform a complete search if the branching factor (b) and the maximum depth of the tree (m) are finite and states are not revisited.

- DFS will not take the optimal path, but it will eventually output the right result.

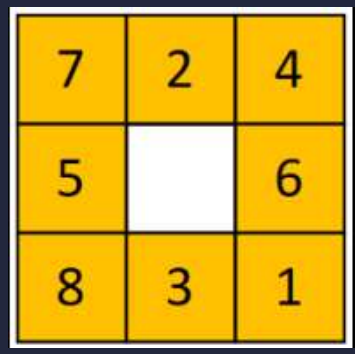
Exercise 2.3b

Each state of the 8-puzzle can be assigned either odd or even parity.

- In the 8-puzzle, parity is irrelevant (i.e., doesn't change) for all valid moves.
  - Therefore, for a solution to exist between the two states, the parity must be the same
- Parity math is both necessary and sufficient for a solution to exist.

Exercise 2.3b

Consider the 8-Puzzle state to the right and its inversions



- N(s, 7) = 6 (out of place w.r.t 2, 4, 5, 6, 3, 1)
- N(s, 2) = 1 (out of place w.r.t 1)
- N(s, 4) = 2 (out of place w.r.t 3, 1)
- N(s, 5) = 2 [out of place w.r.t. 3, 1]
- N(s, 6) = 2 [out of place w.r.t. 3, 1]
- N(s, 8) = 2 [out of place w.r.t. 3, 1]
- N(s, 3) = 1 [out of place w.r.t. 1]
- N(s, 1) = 0 [no tiles out of place]

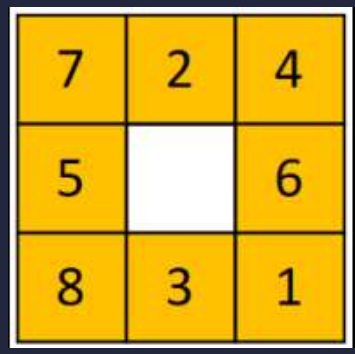
Therefore, the number of inversions N(s) = 16 -> Even parity.

Exercise 2.3b

```
def num_inversions(self):
  total = 0
  for i in range(len(self.squares)):
    if self.squares[i] == '_':
      continue
    si = int(self.squares[i])
    for j in range(i, len(self.squares)):
      if self.squares[j] == '_':
        continue
      sj = int(self.squares[j])
      if si > sj:
        total += 1
  return total

def get_parity(self):
  return self.num_inversions() % 2

# In the main solution code, we write
if p1.get_parity() != p2.get_parity():
        print('No solution')
        return
```



Exercise 2.4

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 2.4

We know that:

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

UCS will find a solution which is optimum in terms of cost, as all actions have non-negative cost

- When the weight of all edges is equal (e.g. all have `cost == 1`), then UCS will expand nodes in the same order as BFS (producing the same solution)