State space search agents

Example: 8 puzzle

Start state

3	1	2
4	7	5
6		8

Goal state

	1	2
3	4	5
6	7	8

Possible actions move the blank space: up, down, left, or right

Example: 8 puzzle

Start state

3	1	2
4	7	5
6		8

Goal state

	1	2
3	4	5
6	7	8

Possible actions move the blank space: up, down, left, or right

An intelligent agent: finds a sequence of moves to go from start state to goal state

Fifteen Puzzle Class

```
class FifteenPuzzle:
    """Implements a generalized fifteen puzzle (n^2 - 1 puzzle).
    The board represents the numbered sliding tiles of sliding tile puzzle.
    The zero on the board indicates a blank space; adjacent tiles can slide
    into the blank space. The goal of the game is to arrange the tiles numbered
    1 ... n^2-1 in inreasing order from the top-left.
    """
    def __init__(self, size=None, board=None, empty_cell=None, goal=None):
        """Initializes the board. If empty_cell or goal are provided they will
        be stored. Otherwise, they will be computed.
        """
```

Useful methods:

```
def goalReached(self):
    """Compares the current board to the goal."""

def getPossibleMoves(self):
    """Returns a subset of [U,D,L,R] indicating the feasible directions
    that the BLANK SPACE could move."""

def nextState(self, move):
    """Create a new game with the board updated according to the move."""
```

`puzzle.board`

The `board` member variable stores a list of lists representing the puzzle state

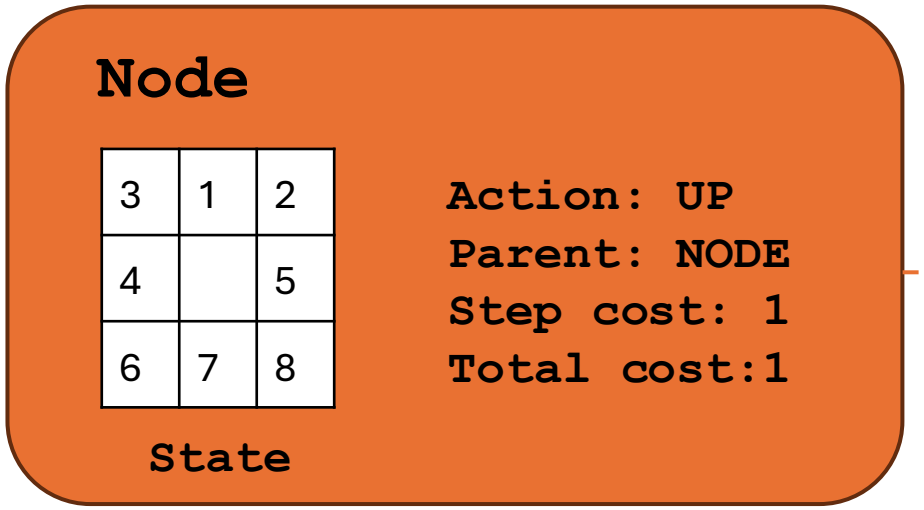
```
>>> print(puzzle.board)
[[ 3  1 15  4]
 [ 9  5 12 14]
 [10  2  8  6]
 [ 7 11 13  0]]
```

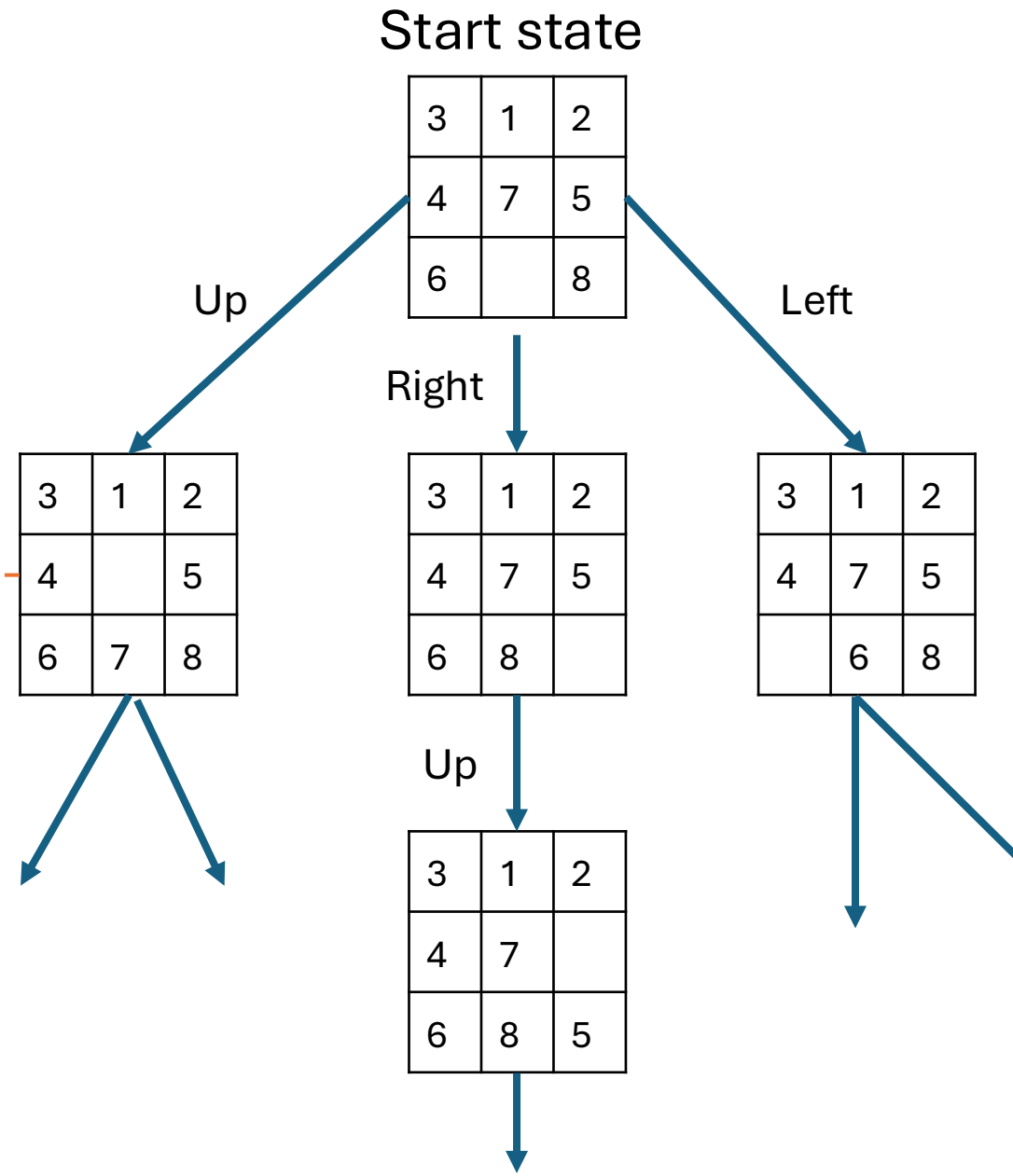
Implementing state space search

- Build a search tree on top of the state space
- Tree will consist of nodes representing each state with links representing actions
- Root will be a node containing the start state
- Frontier will be a data structure containing the successor nodes we need to explore
- Search strategies vary on how nodes are selected from the frontier

Nodes in graph may contain

- State description
- Action taken to get to this state
- Pointer to its parent node
- Step cost
- Total cost so far





Note: Node class is provided in the lab starter code!

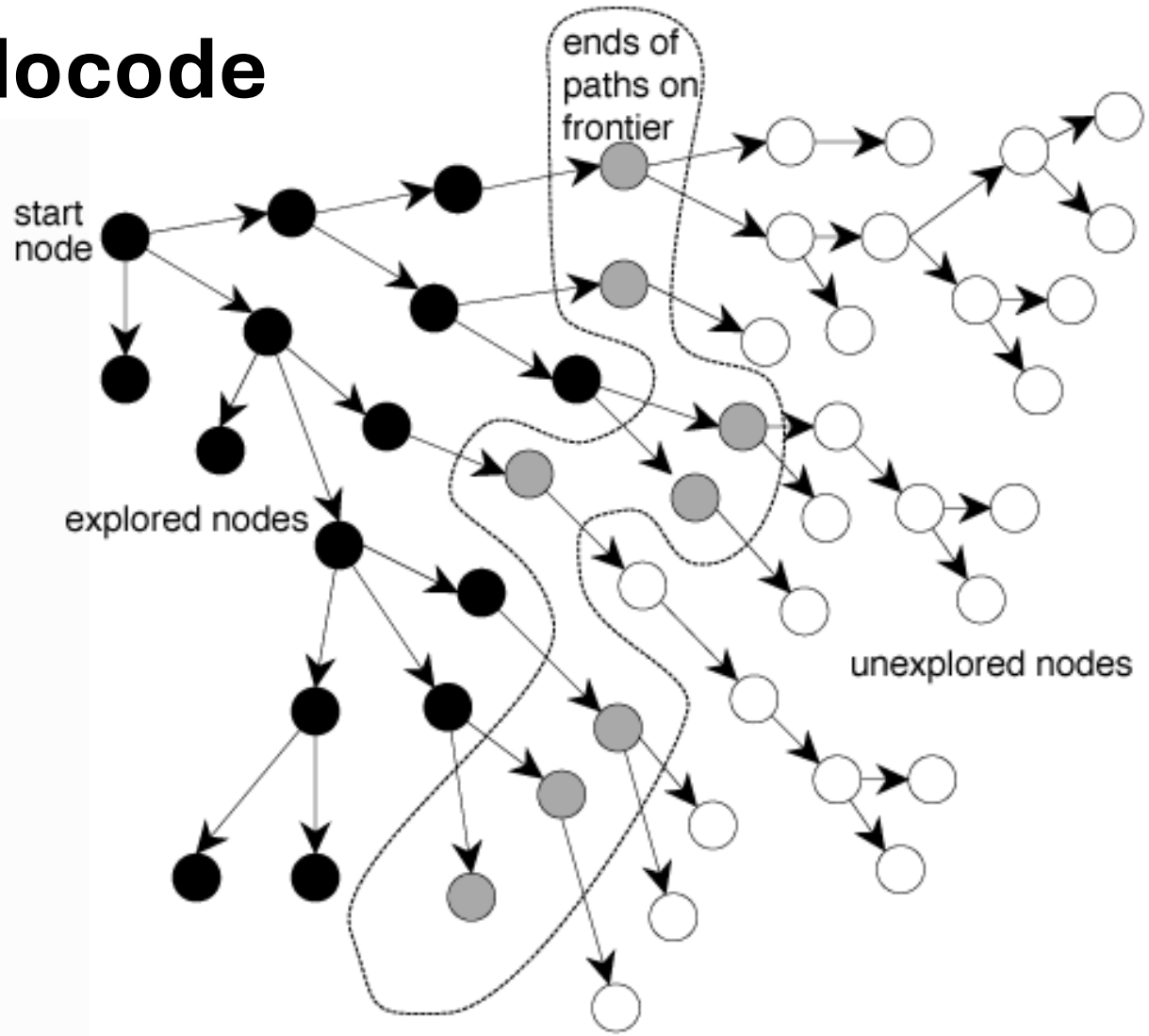
Provided Node Class

```
6   class Node:
7       """Bookkeeping class for state space search."""
8       def __init__(self, state, parent, action, depth):
9           """
10          state     type depends on problem, for TrafficJam it's a
11                    TrafficJam object, for FifteenPuzzle it's a
12                    FifteenPuzzle object
13          parent    pointer to the parent node in the graph
14          action    type depends on problem, represents action taken to get
15                    to this state
16          depth     integer that represents the depth of the node in graph
17          """
18          self.state = state
19          self.parent = parent
20          self.action = action
21          self.depth = depth
22
23      def __str__(self):
24          result = "\nState:\n" + str(self.state)
25          result += "\nDepth: " + str(self.depth)
26          return result
```

Graph search pseudocode

GRAPH-SEARCH(problem) returns a solution or failure

```
Initialize the frontier with a node containing the initial state
Initialize visited to contain the initial state
Use a data structure with efficient lookup for visited

loop
  if the frontier is empty
    return failure

  Remove a node from the frontier

  if the node contains a goal state
    return the solution represented by the node

  Apply operators to the node's state to obtain successor states

  for each successor state
    if the successor state is not in visited
      Add the successor state to visited
      Create a child node containing the successor state
      Set its parent to the chosen node
      Add the child node to the frontier
```



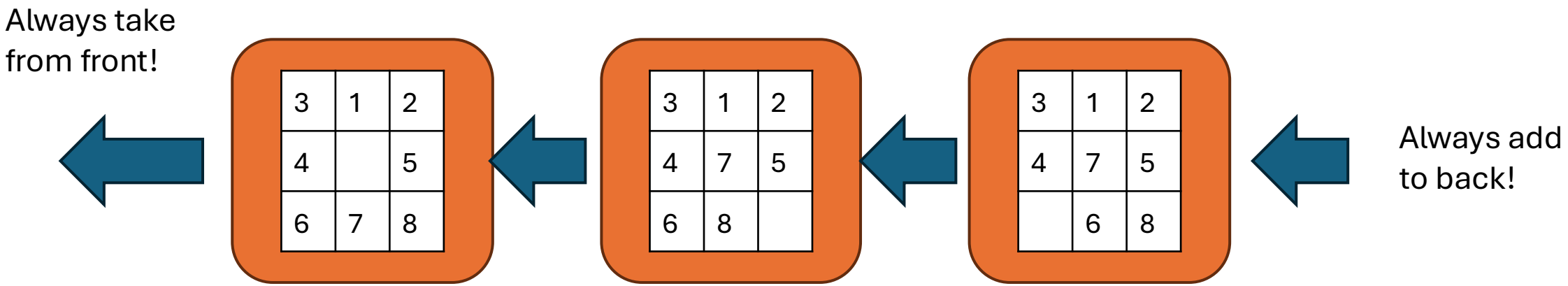
Review: What abstract data type makes sense for *visited?* What data structure might we use to implement it?

Uninformed vs Informed methods

- Uninformed methods are only given the problem definition
  - start state
  - a way to check if at a goal state
  - a way to generate successors
  - Exs: DFS, BFS, Lowest cost, Depth-limited, IDS
- Informed methods are given additional information about the goal
  - estimated cost to reach goal
  - Exs: Greedy (aka Best First), A*
- All of these methods use a similar algorithm, what changes is the frontier data structure

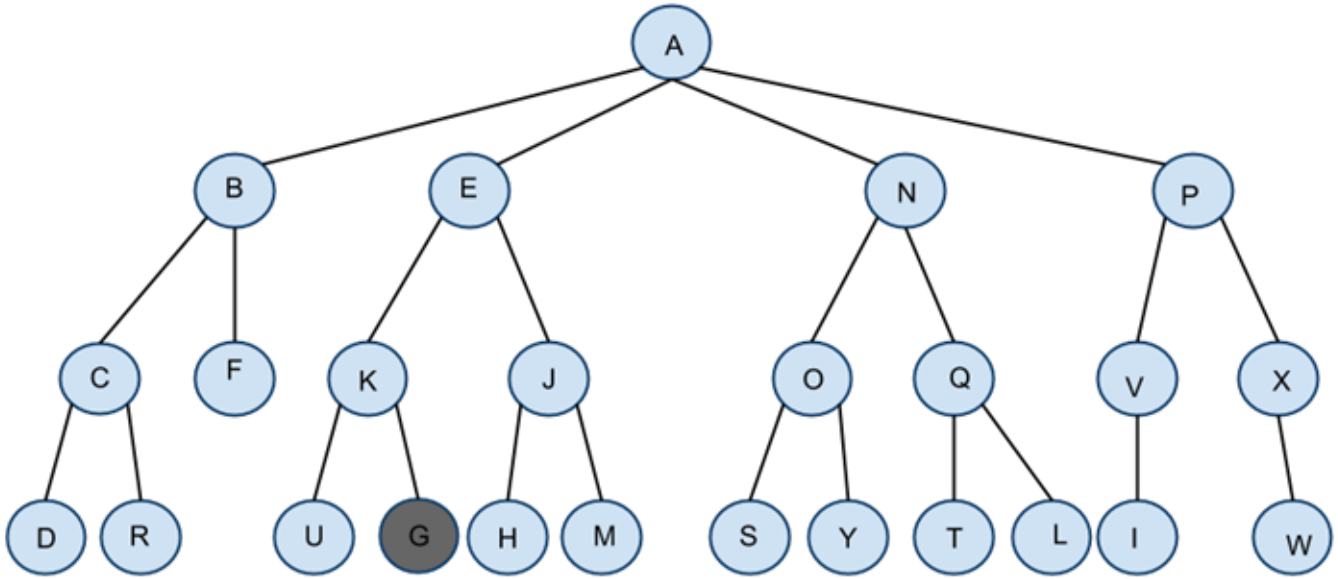
Breadth-first Search

- Uses a queue to represent the frontier
- All nodes at a given level of the tree are expanded before any nodes at the next level
- Always expands the shallowest node in the frontier



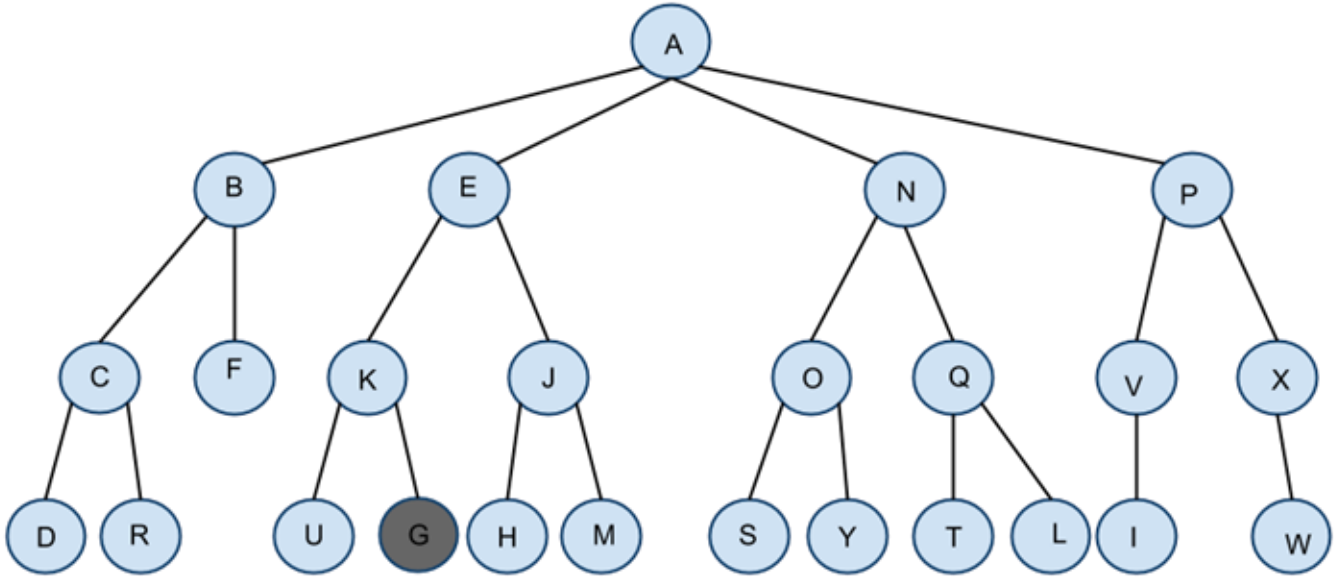
Review: What data structure(s) could we use to implement a queue?

BFS order of exploration?



A is the start state
G is the goal state
Assume neighbors added to frontier in R to L order

BFS order of exploration



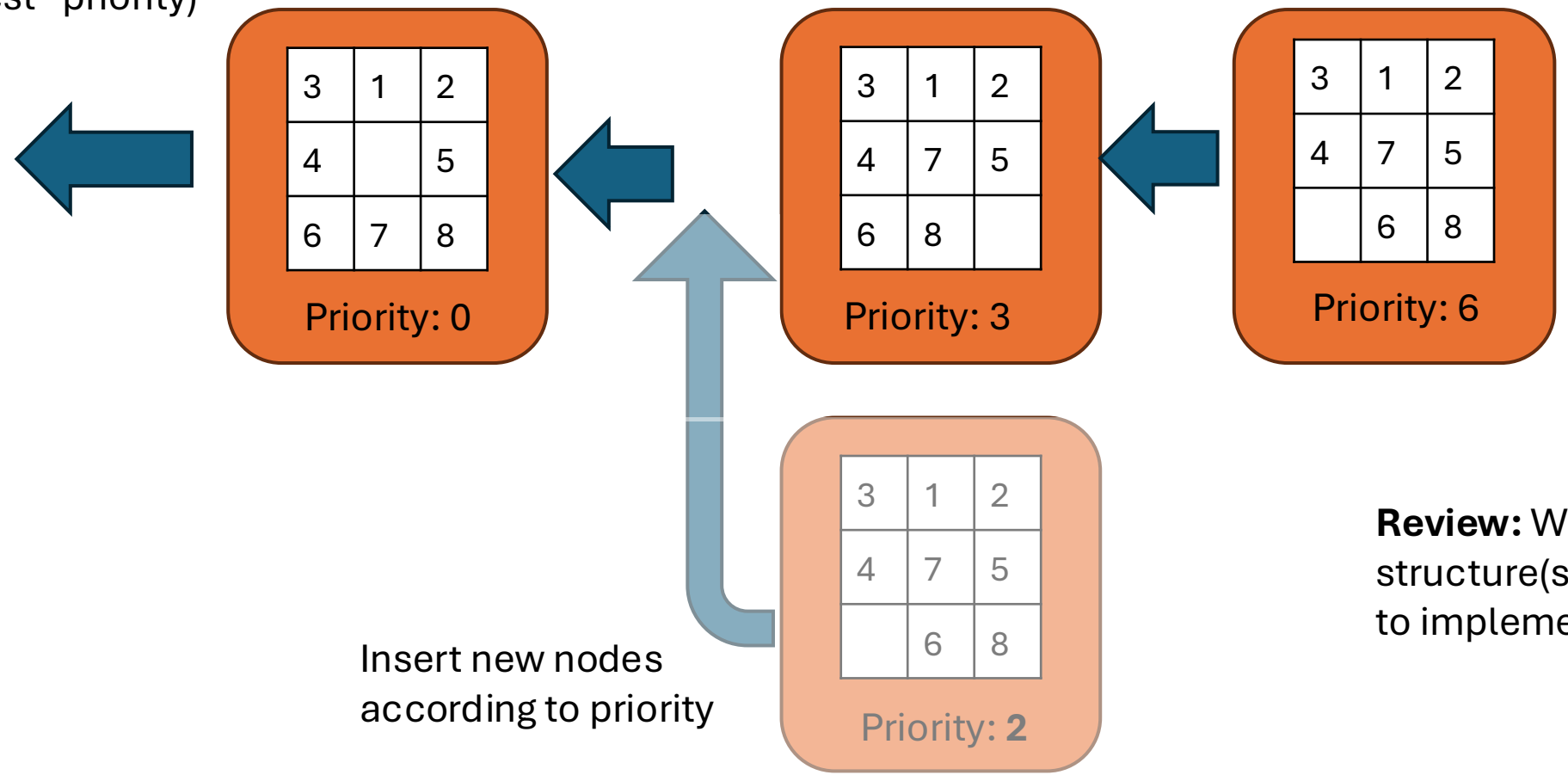
A, P, N, E, B, X, V, Q, O, J, K, F, C, W, I, L, T, Y, S, M, H, G

Uniform Cost Search (better name would be Lowest Cost)

- Similar to BFS, but rather than expanding the shallowest node, expands the node with the lowest path cost
- Uses a priority queue to store the frontier
- More complicated to implement
  - Can’t just ignore a repeated state
  - When a repeated state is found, check to see if the new path cost is less, and if so replace the old node with this new node

Reminder: Priority queue

Always take from front!
(“Highest” priority)



Insert new nodes according to priority

Review: What data structure(s) could we use to implement a queue?

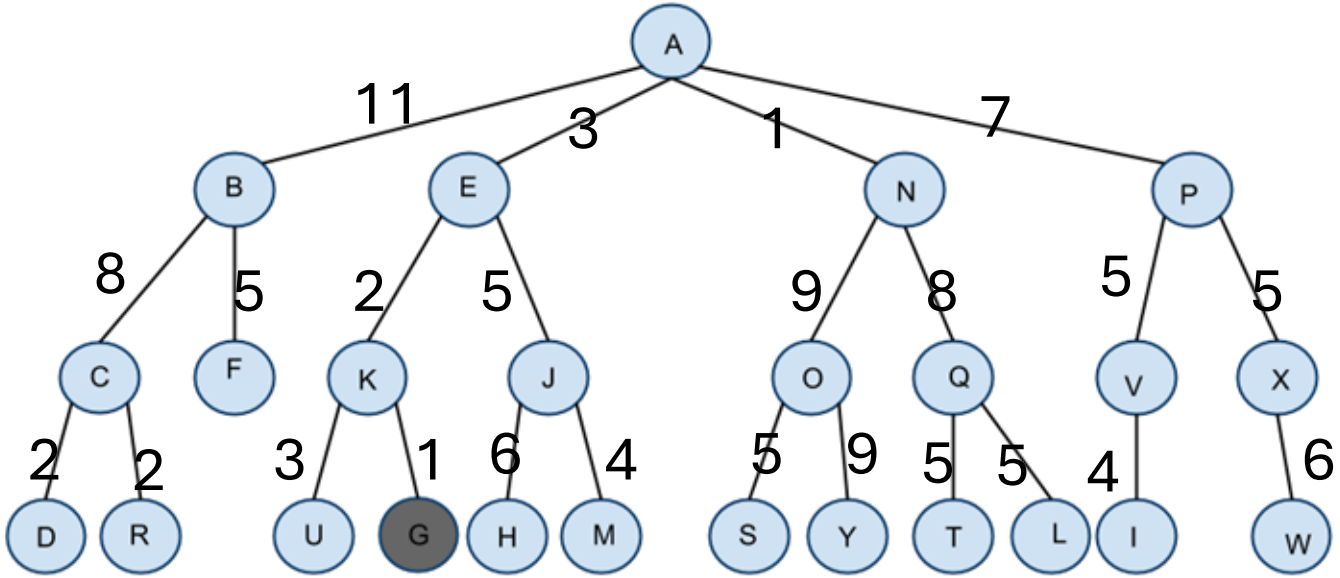
Provided Priority Queue Class

```
7   class PriorityQueue():
8       def __init__(self):
9           """
10          Represent the min priority queue as a heap, which is a complete
11          binary tree represented as a list.
12
13          A user of the PQ should only call the methods: isEmpty,
14          getSize, insert, and remove. All other methods are for internal
15          use only.
16          """
17          self.heap = []
18      def isEmpty(self):
19          """Returns True of the PQ is empty, otherwise False"""
20          return len(self.heap) == 0
21      def getSize(self):
22          """Returns the integer size of the PQ"""
23          return len(self.heap)
24      def insert(self, priority, item):
25          """Inserts the item with given priority"""
26          self.heap.append((priority, item))
27          self._bubbleUp(self.getSize()-1)
28      def remove(self):
29          """Removes the item in the PQ with the minimum priority"""
30          if self.getSize() == 0:
31              raise RuntimeError("can't remove from an empty PQ")
32          removed_item = self.heap[0][1]
33          last_index = self.getSize()-1
34          self.heap[0] = self.heap[last_index]
35          self.heap.pop(last_index)
36          self._bubbleDown(0)
37          return removed_item
```

Inserting an item requires a priority!

Remove also returns the removed item!

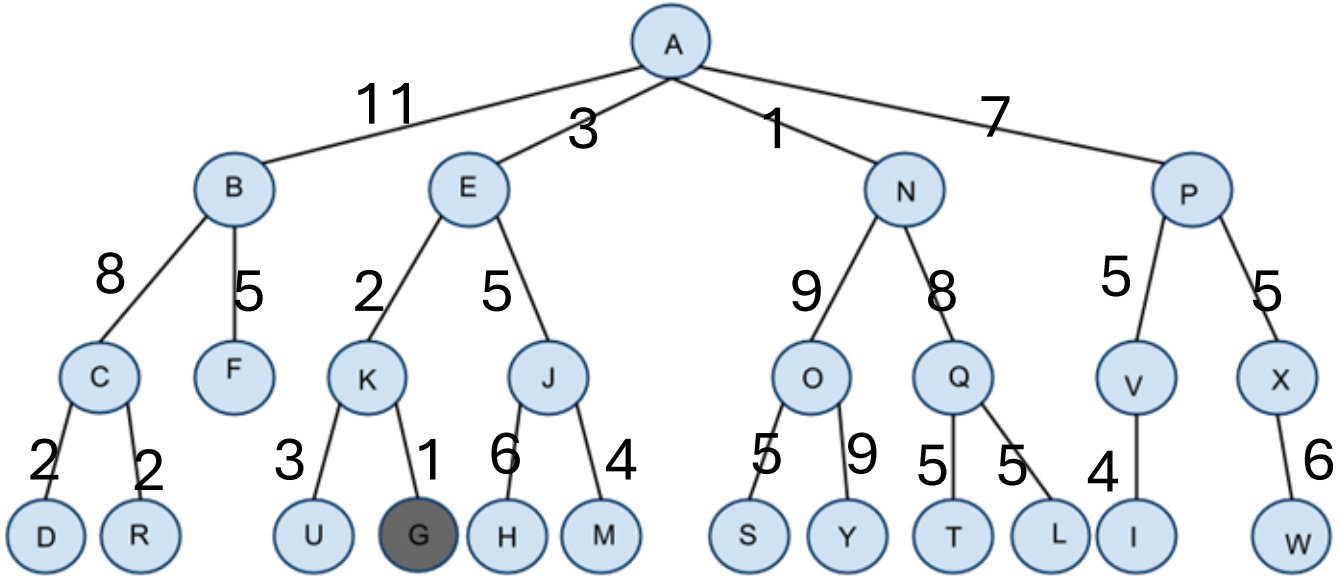
Uniform Cost order of exploration?



A is the start state

G is the goal state

Uniform Cost order of exploration



A, N, E, K, G

Lab time!

Informed Search Methods

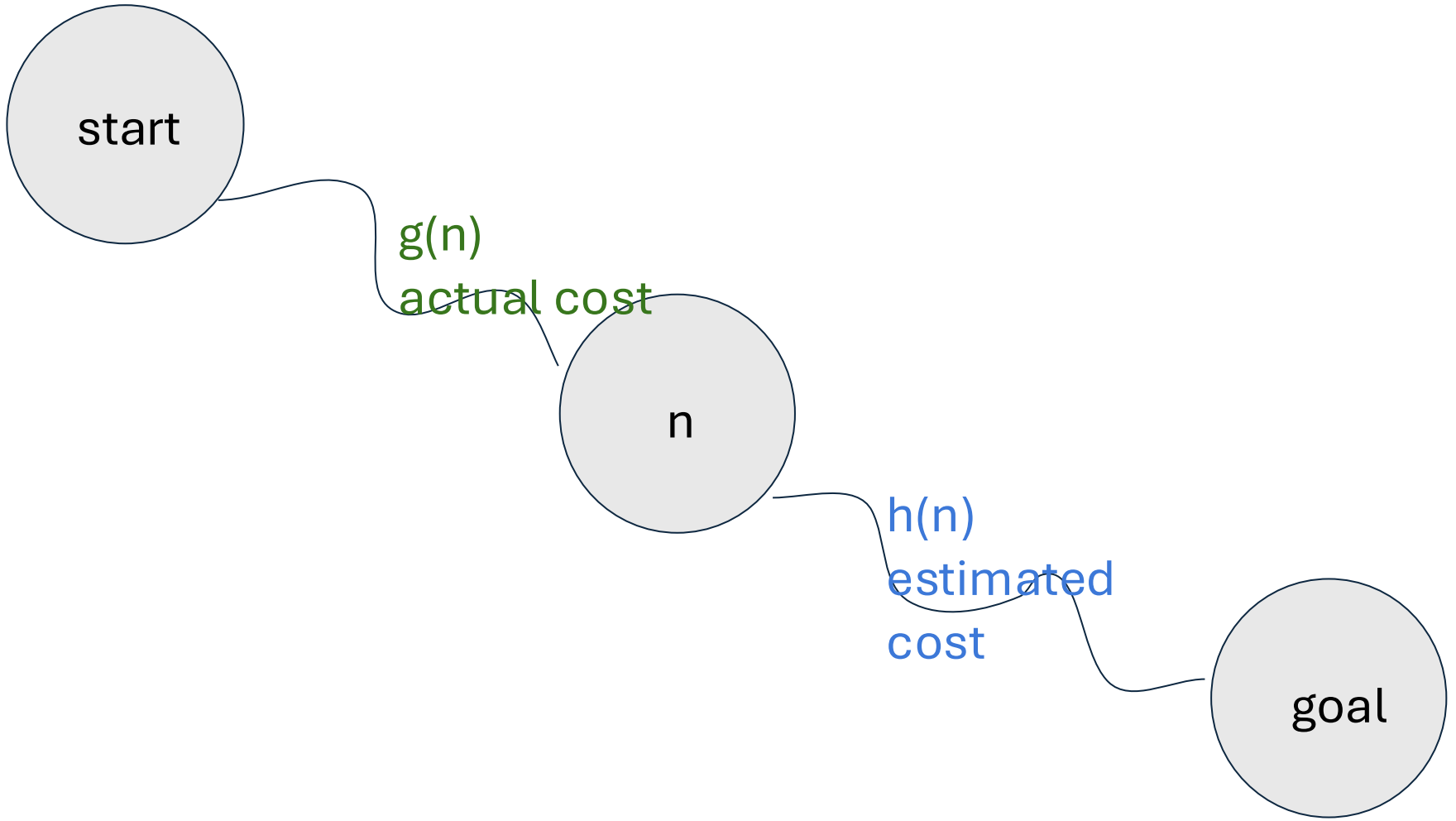
- Uninformed methods (also called blind search) can find solutions by generating successor states and checking for the goal
- Informed methods use problem-specific information to estimate how close a given state is to the goal
- Informed methods can dramatically improve search speed

Terminology

- Let n be the current node
- g(n)
  - actual cost of path from start node to current node n
- h(n)
  - estimated cost of path from current node n to goal
  - “h” stands for heuristic
- f(n) = g(n) + h(n)
  - estimated cost of cheapest solution through node n

Estimated cost of optimal path through n

f(n) = g(n) + h(n)



Informed Search Methods

- Greedy (also called Best-First)
  - Only looks ahead at estimated distance to goal
  - Ignores g(n)
  - Uses a PQ sorted on h(n)
- A*
  - Looks at both path cost and est. distance to goal
  - Uses a PQ sorted on g(n) + h(n)

A* pseudocode

```
AStar(problem) returns a solution or failure

    Create a node containing the initial state with path cost zero
    Add this node to the priority queue frontier with priority zero
    Initialize visited to be empty

    while the frontier is not empty
        Remove the node with the lowest priority from the frontier

        if the current node's state is a goal state
            return the solution represented by the current node

        if the current node's state is not in visited
            Add the current node's state to visited

            for each possible action from the current node's state
                Determine the next state

                if the next state is not in visited
                    Create a child node containing the next state
                    Set its parent to the current node
                    Set its path cost to the current path cost plus the action cost
                    Set its priority to its path cost plus the heuristic estimate
                    Add the child node to the frontier with this priority

    return failure
```

Heursitics in `InformedSearchAgent`

The provided search agent stores a heuristic as a member variable

```
class InformedSearchAgent(object):
    def __init__(self, puzzle, heuristicFunction):
        """Stores the puzzle. Initializes an integer to count the number of
        nodes expanded by the search. Stores the heuristic function to be
        called during A* search."""
        self.initial_puzzle = puzzle
        self.expanded = 0
        self.heuristic = heuristicFunction
```

You can call it as you would a class method (In Python, functions *are* objects!)

```
estimate = self.heuristic(nextState)
```

Reminder: 8 puzzle

Start state

3	1	2
4	7	5
6		8

Goal state

	1	2
3	4	5
6	7	8

Possible actions move the blank space: up, down, left, or right

Inventing Heuristics

- `heuristic(state)`: returns an estimate as a number
- Consider a relaxed version of the problem
- 8 puzzle
  - The number of misplaced tiles (ignores the moves it would require)
  - The sum of the Manhattan distances of each tile from its goal location (ignores the other tiles blocking the moves)

Examples of heuristics applied

Current state

3	7	1
4	8	2
6		5

Goal state

	1	2
3	4	5
6	7	8

Misplaced tiles heuristic: Only tile 6 is in the right location, **returns 7**

Manhattan dist heuristic: Sum of distance each tile must move, tiles 7 and 8 are two steps away, tile 6 is 0 steps away, all others are 1 step away, **returns 9**

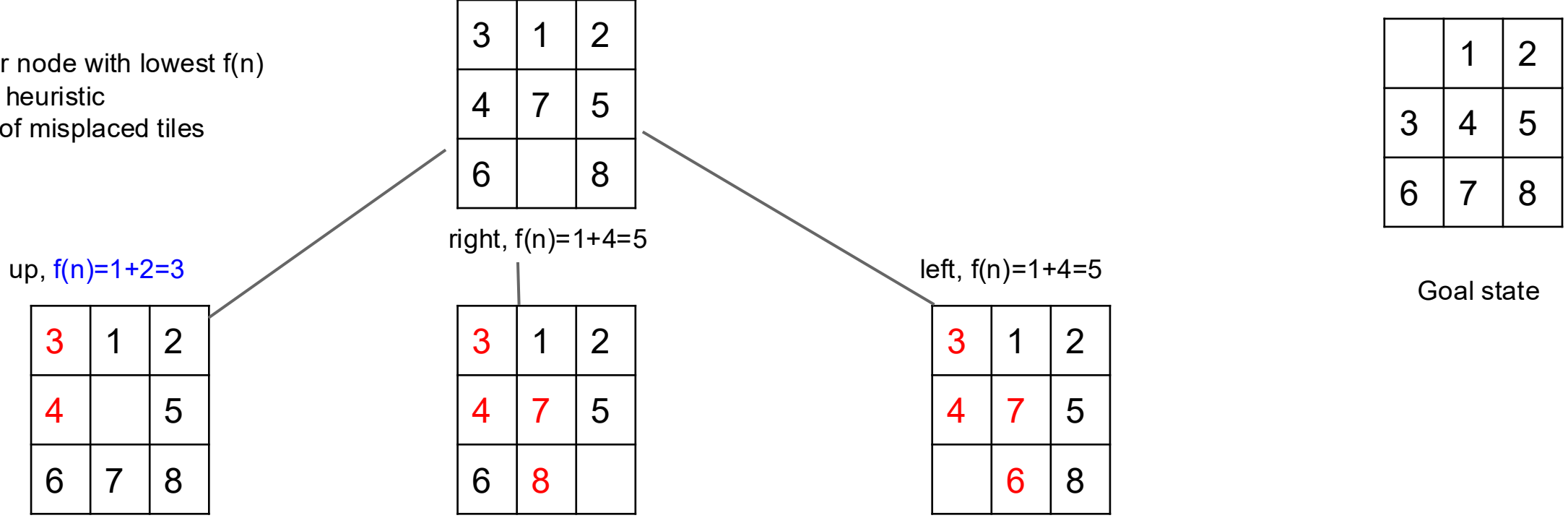
NOTE: We ignore the blank in these computations

Trace A*

A* expands frontier node with lowest f(n)

f(n) = costSoFar + heuristic

heuristic: Number of misplaced tiles

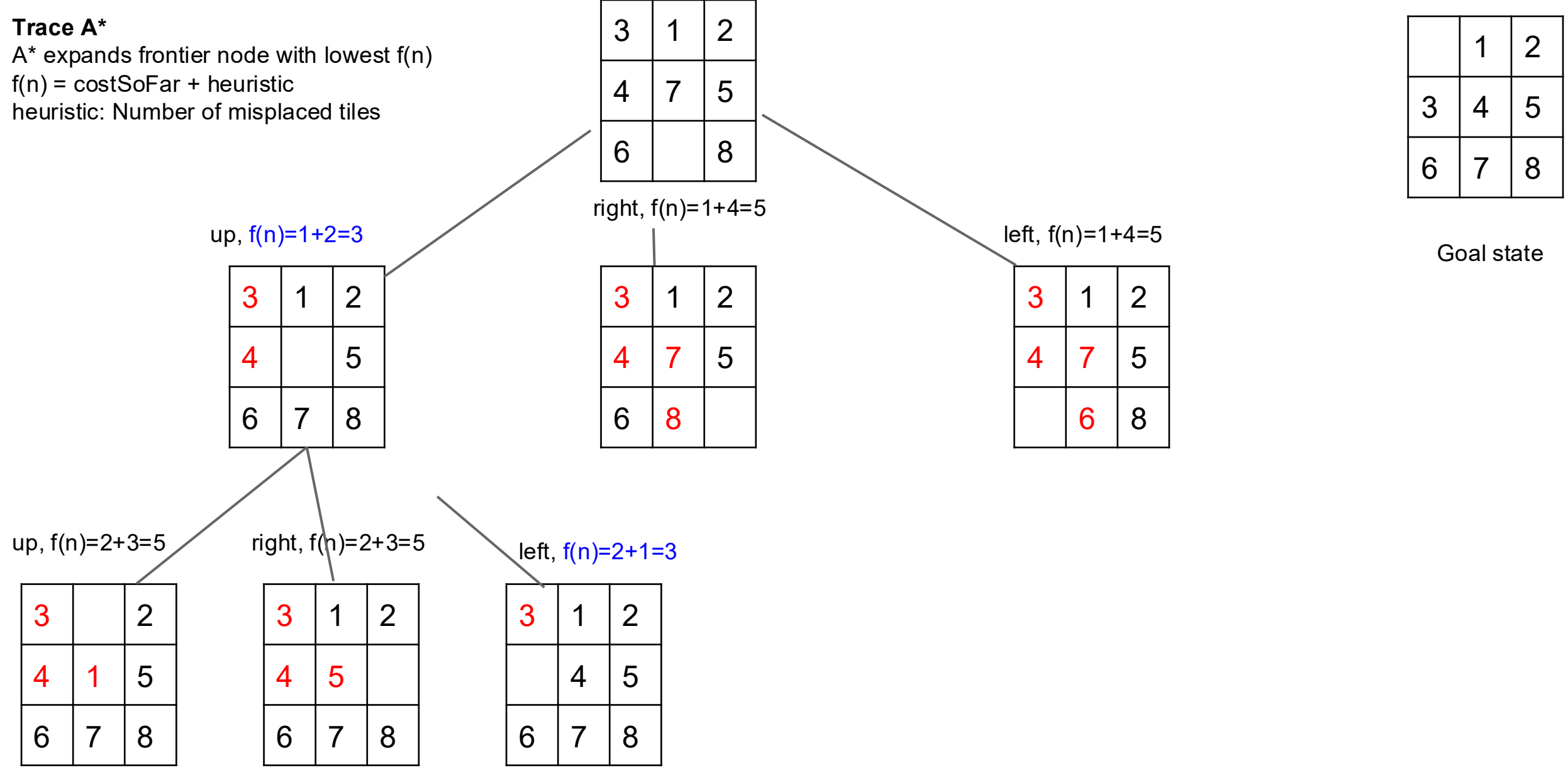


Trace A*

A* expands frontier node with lowest f(n)

f(n) = costSoFar + heuristic

heuristic: Number of misplaced tiles



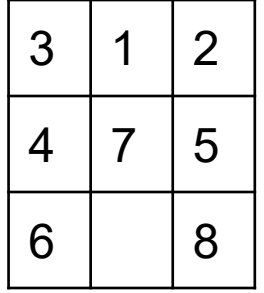
Goal state

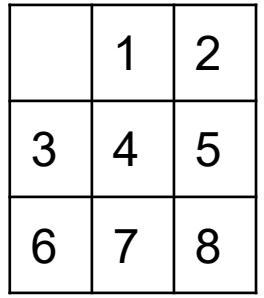
Trace A*

A* expands frontier node with lowest f(n)

f(n) = costSoFar + heuristic

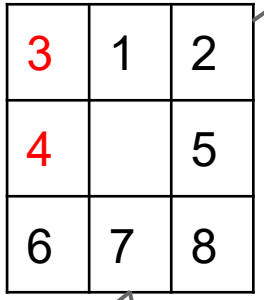
heuristic: Number of misplaced tiles



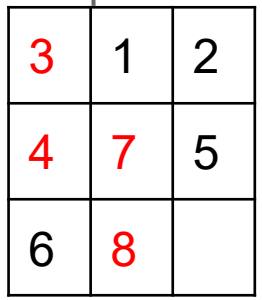


Goal state

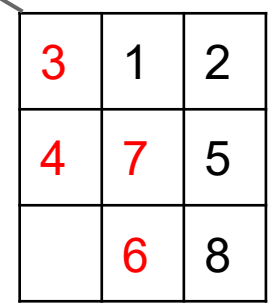
up, f(n)=1+2=3



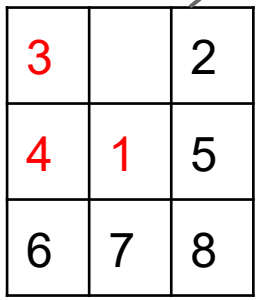
right, f(n)=1+4=5



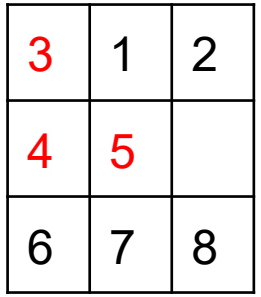
left, f(n)=1+4=5



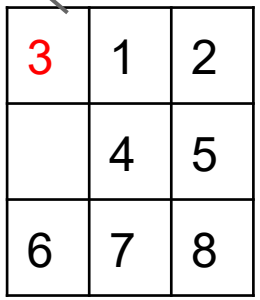
up, f(n)=2+3=5



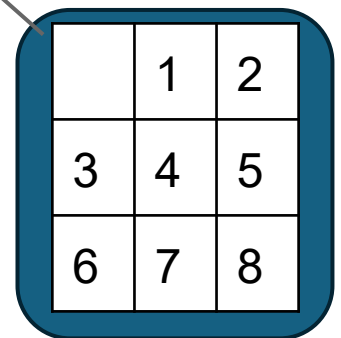
right, f(n)=2+3=5



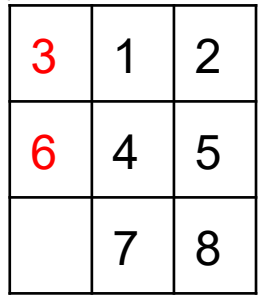
left, f(n)=2+1=3



A. up, f(n)=3+0=3



down, f(n)=3+2=5



Rules for heuristic (more on this tomorrow!)

- Heuristics must be admissible
  - Never overestimates the cost to reach the goal
  - Why are the misplaced tiles and Manhattan distances admissible?
- Heuristics must be consistent
  - Let n be a node, n' be a successor of n, a be the action that leads from n to n'
  - h(n) <= cost(n, a, n') + h(n')
- Consistency is a stronger condition
- If h(n) is consistent, then the values of f(n) along any path are nondecreasing

Can't decrease estimated distance by more than cost!

Consistency: h(n) <= cost(n,a,n') + h(n')