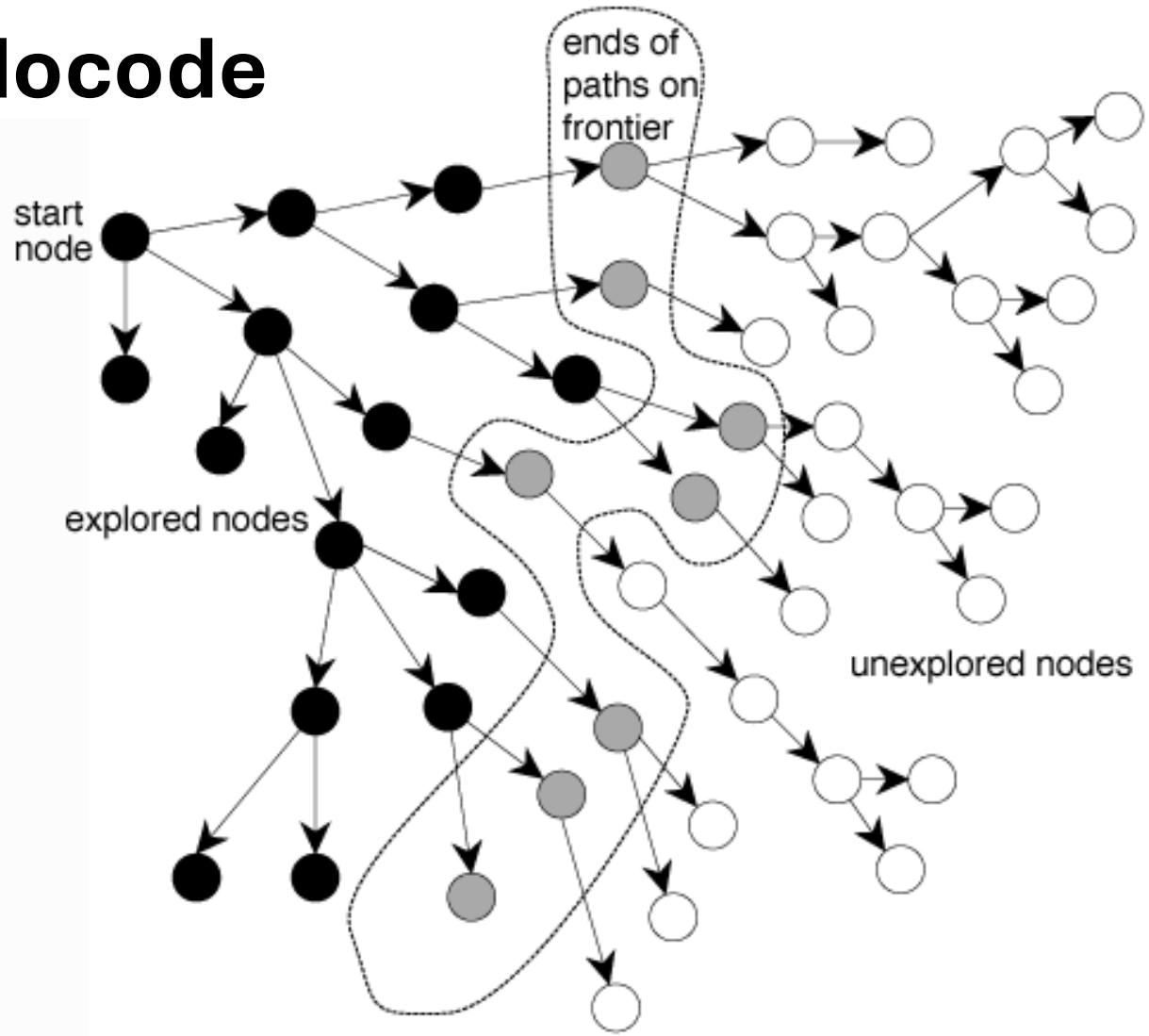
Informed search

First: iClicker registration and Quiz

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?

Measuring search strategy performance

1. Completeness: Is the search guaranteed to find a solution if it exists?
2. Optimality: Is the search guaranteed to find the best solution in terms of cost?
3. Time complexity: How long does it take to find a solution?
4. Space complexity: How much memory is needed to perform the search?

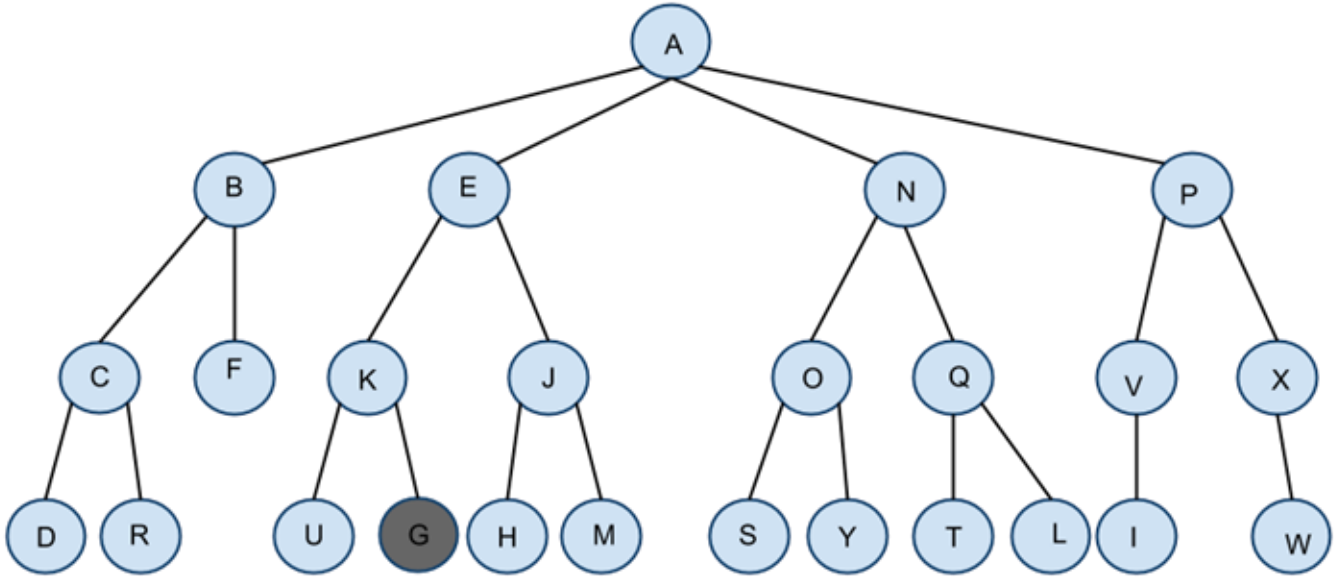
Assume:

- b=branching factor
- d=depth of the shortest path to the goal

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

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

Breadth-First Search Performance

- Complete?
- Optimal?
- Space efficient?
- Time efficient?

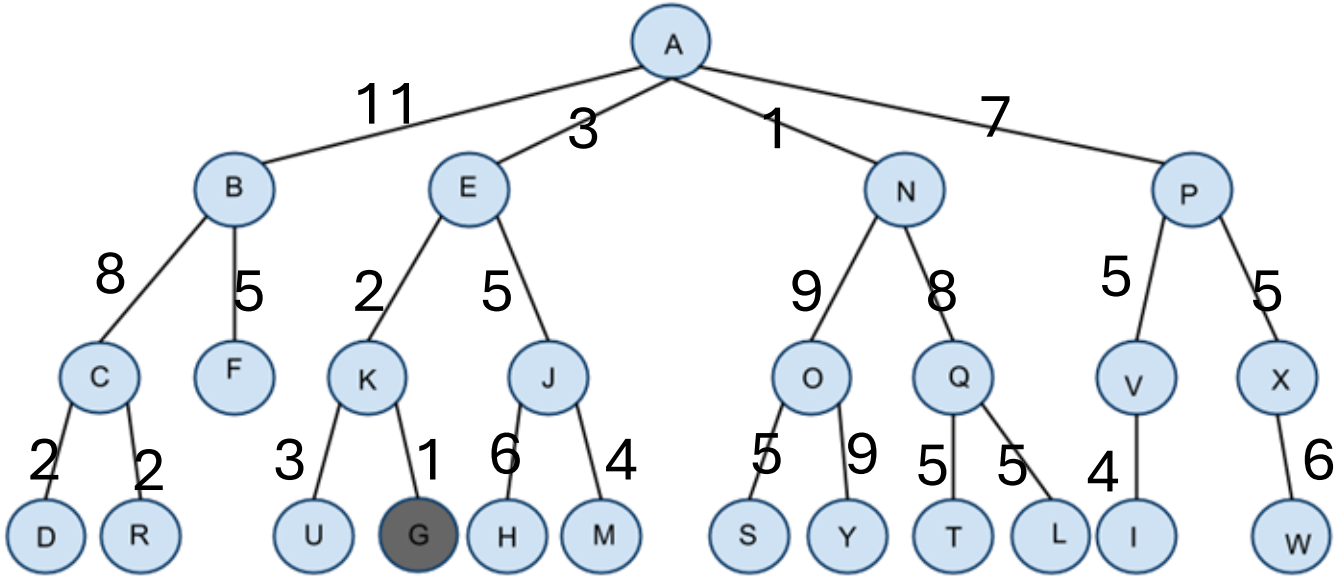
Breadth-First Search Performance

- Complete? Yes
  - Guaranteed to find shallowest goal node
- Optimal? Yes*
  - *If step costs are equal
- Space efficient? No
  - Nodes expanded $b + b^2 + b^3 + \ldots + b^d$
  - Frontier will contain $O(b^d)$ nodes, which can be huge
- Time efficient? No

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

Uniform Cost order of exploration



A, N, E, K, G

Uniform Cost Search Performance

- Complete?
- Optimal?
- Space efficient?
- Time efficient?

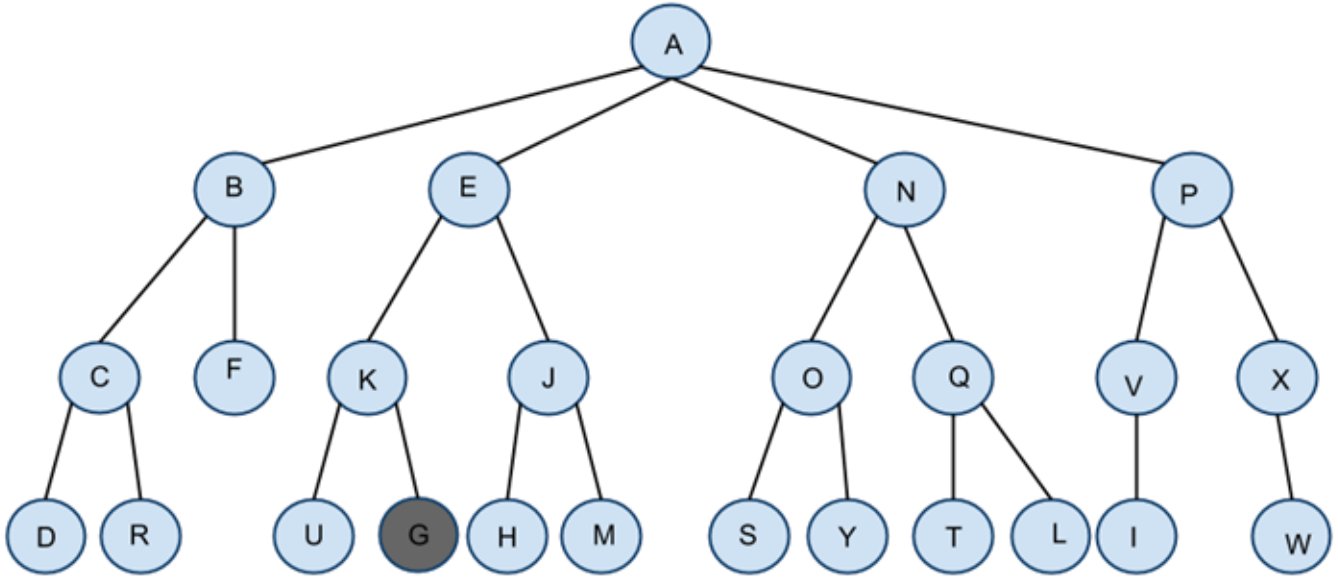
Uniform Cost Search Performance

- Complete? Yes
  - As long as step costs are positive
- Optimal? Yes
  - Expands nodes in order of their path cost (not depth)
  - So when goal is found, guaranteed to be lowest cost
- Space efficient? No, but better than BFS
- Time efficient? No, but better than BFS

Depth-First Search

- Uses a stack to represent the frontier
- Always expands the deepest node in the frontier
- Can get caught in loops if repeated states are not checked

DFS order of exploration?

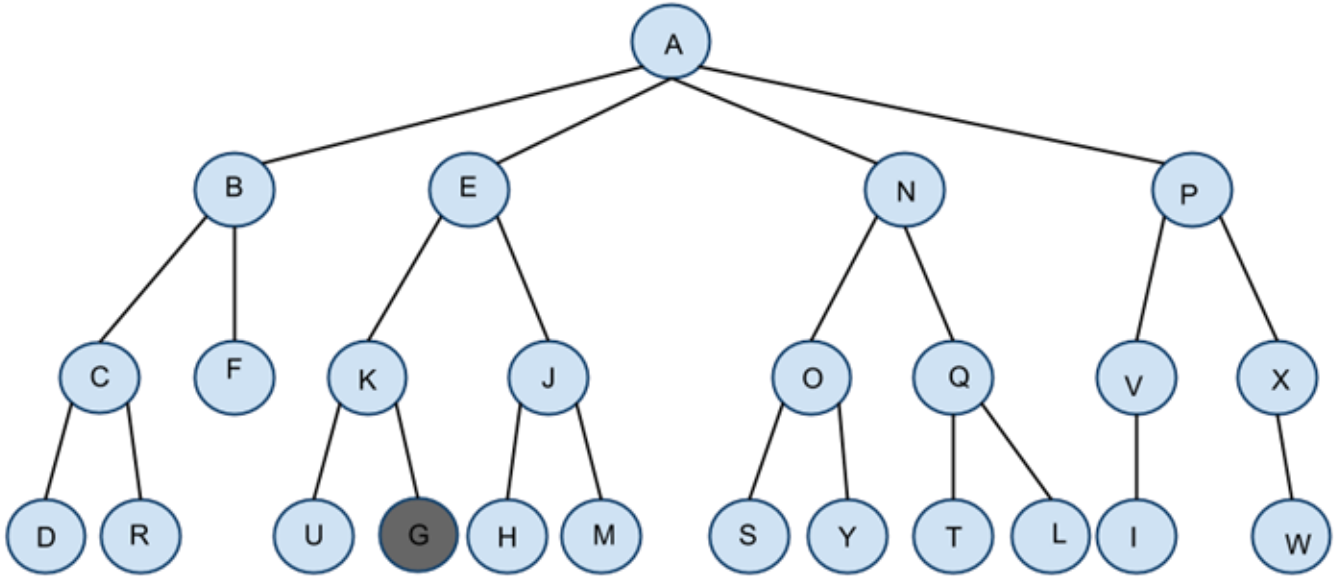


A is the start state

G is the goal state

Assume neighbors added to frontier in R to L order

DFS order of exploration



A, B, C, D, R, F, E, K, U, G

Depth-First Search Performance

- Complete?
- Optimal?
- Space efficient?
- Time efficient?

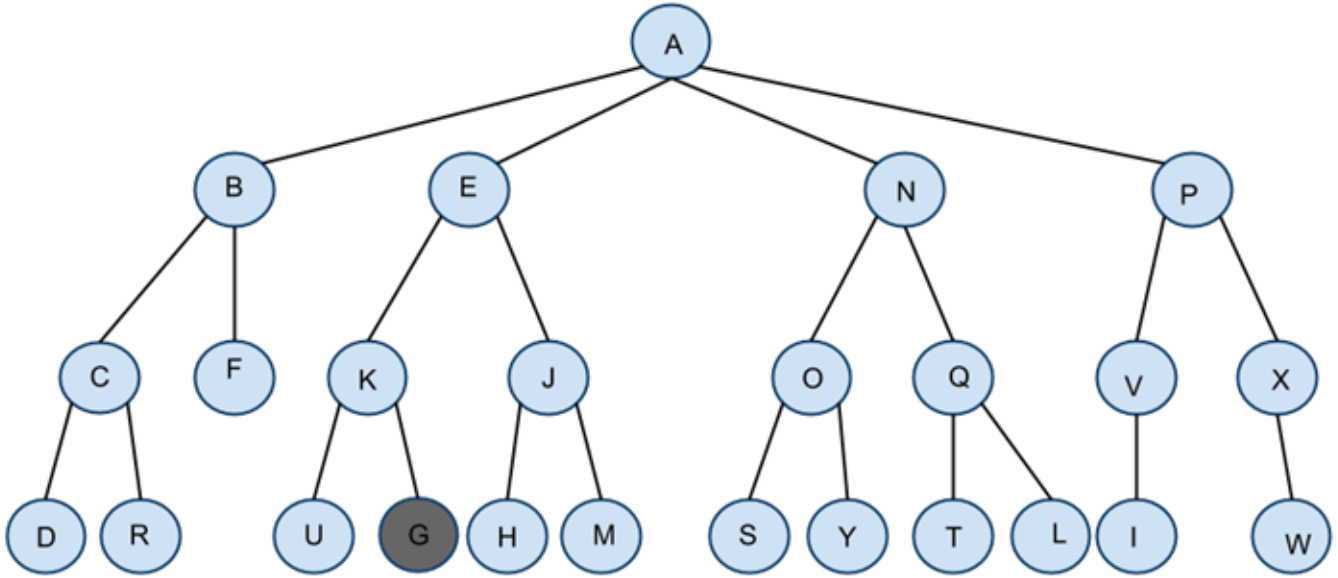
Depth-First Search Performance

- Complete? No
  - Can go forever down an infinite path that doesn't lead to a goal
- Optimal? No
  - May find a suboptimal path, when shorter paths exist
- Space efficient? Yes
  - Only needs to store a single path from the root to a leaf node
- Time efficient? Depends
  - Sometimes finds solutions very quickly

Depth-Limited Search

- Improvement on DFS, can handle infinite state spaces
- Provide a predetermined depth limit, when limit is reached, backtrack
- New problem: How best to determine an appropriate limit?

Depth-limited order of exploration?

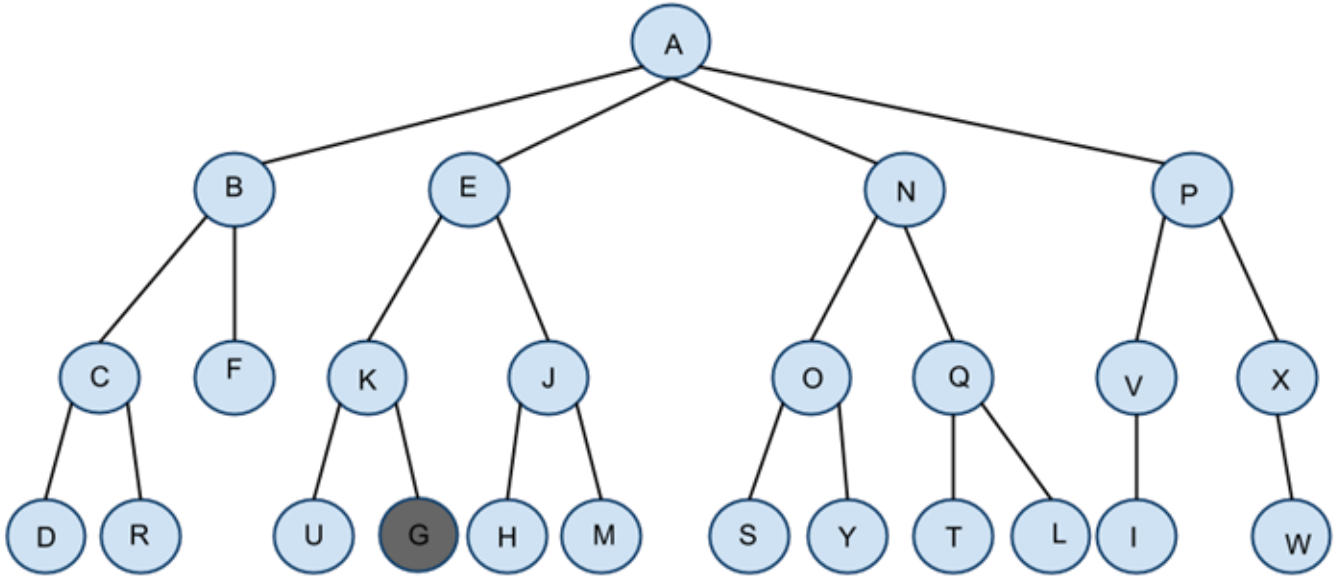


A is the start state

G is the goal state

Assume neighbors added to frontier in R to L order and depth limit is 2

Depth-limited order of exploration



With depth limit of 2:
A, B, C, F, E, K, J, N, O, Q, P, V, X, fails

Depth-Limited Search Performance

- Complete?
- Optimal?
- Space efficient?
- Time efficient?

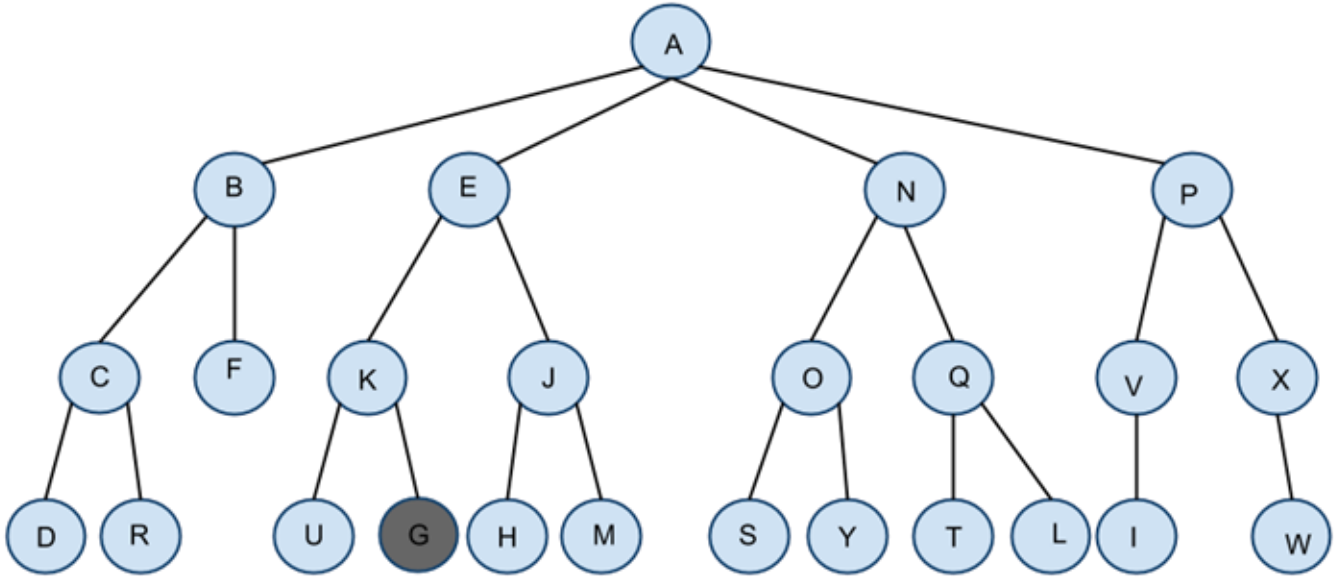
Depth-Limited Search Performance

- Complete? No
  - Only complete if shallowest goal is within chosen limit
- Optimal? No
  - May find a suboptimal path when shorter paths exist
- Space efficient? Yes
- Time efficient? Similar to DFS

Iterative Deepening DFS

- Improvement on Depth-Limited Search, because no predetermined limit is needed
- Perform repeated Depth-Limited searches, where the limit starts at 0 and is incremented each time

Iterative deepening order of exploration?

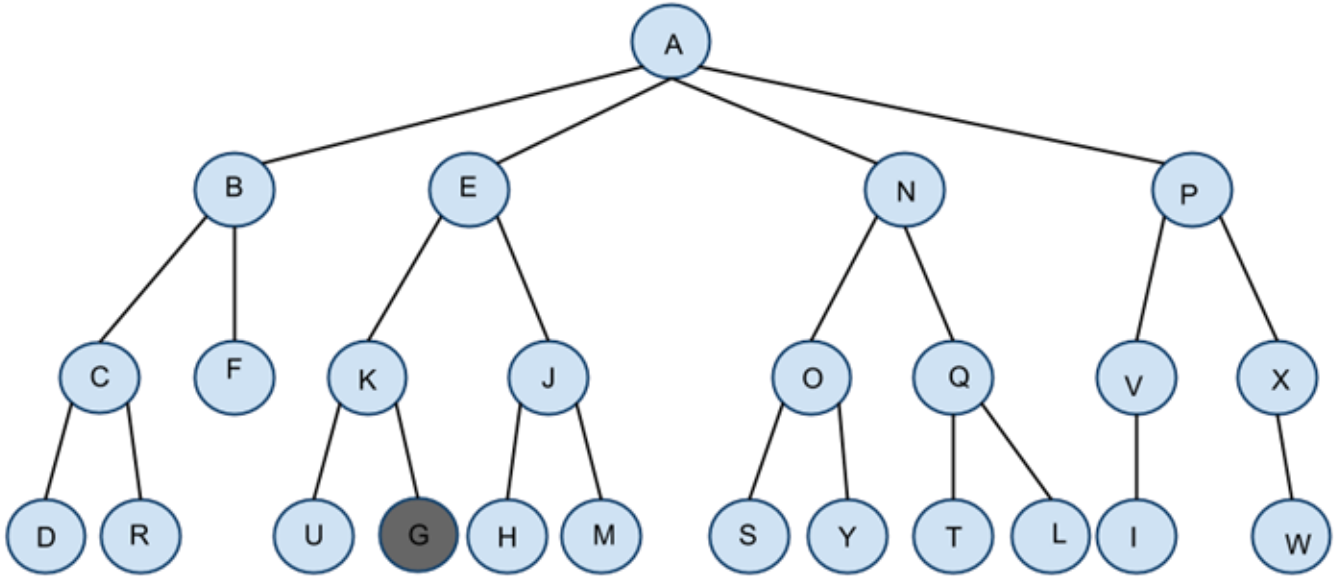


A is the start state

G is the goal state

Assume neighbors added to frontier in R to L order

Iterative deepening order of exploration



limit 0: A
limit 1: A, B, E, N, P
limit 2: A, B, C, F, E, K, J, N, O, Q, P, V, X
limit 3: A, B, C, D, R, F, E, K, U, G

Iterative Deepening Performance

- Complete?
- Optimal?
- Space efficient?
- Time efficient?

Iterative Deepening Performance

- Complete? Yes
  - Each repetition goes one level deeper into the tree
  - Can handle infinite state spaces
- Optimal? Yes*
  - *Will find the shallowest goal node, so optimal when step costs are equal
- Space efficient? Yes, similar to DFS
- Time efficient? Surprisingly, no worse than BFS
  - Most of the nodes in a tree are at the bottom not the top, so repeatedly re-searching the top is not that expensive

Iterative Deepening Performance

- Combines the benefits of both BFS & DFS
- Completeness and optimality of BFS
- Space efficiency of DFS
- This is the preferred method of search when the search space is large and the depth of the solution is unknown

Informed Search Methods

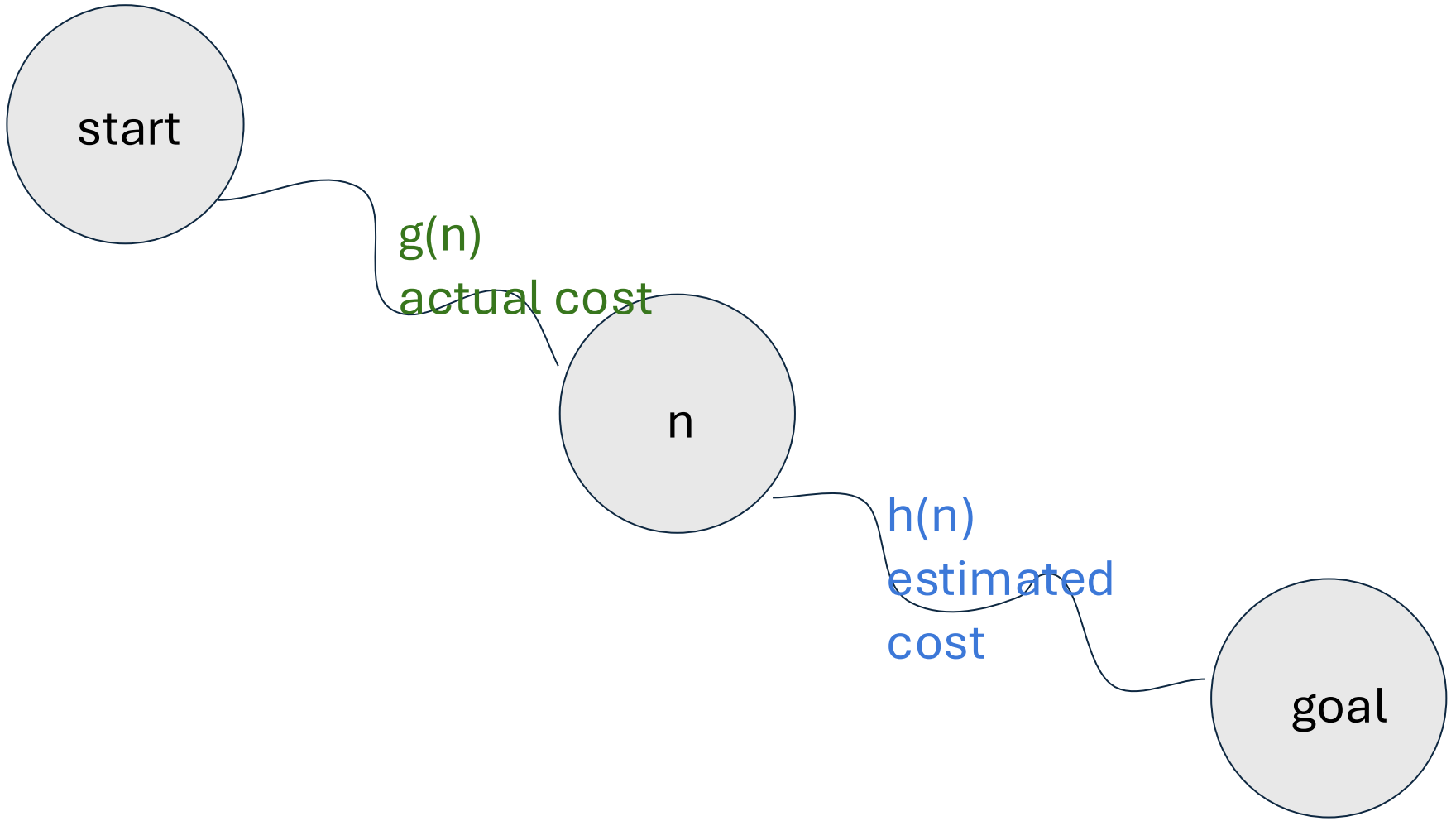
- Uninformed methods (also called blind search) can find solutions by generating successor states and checking for the goal
- Informed methods use problem-specific information to estimate how close a given state is to the goal
- Informed methods can dramatically improve search speed

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**

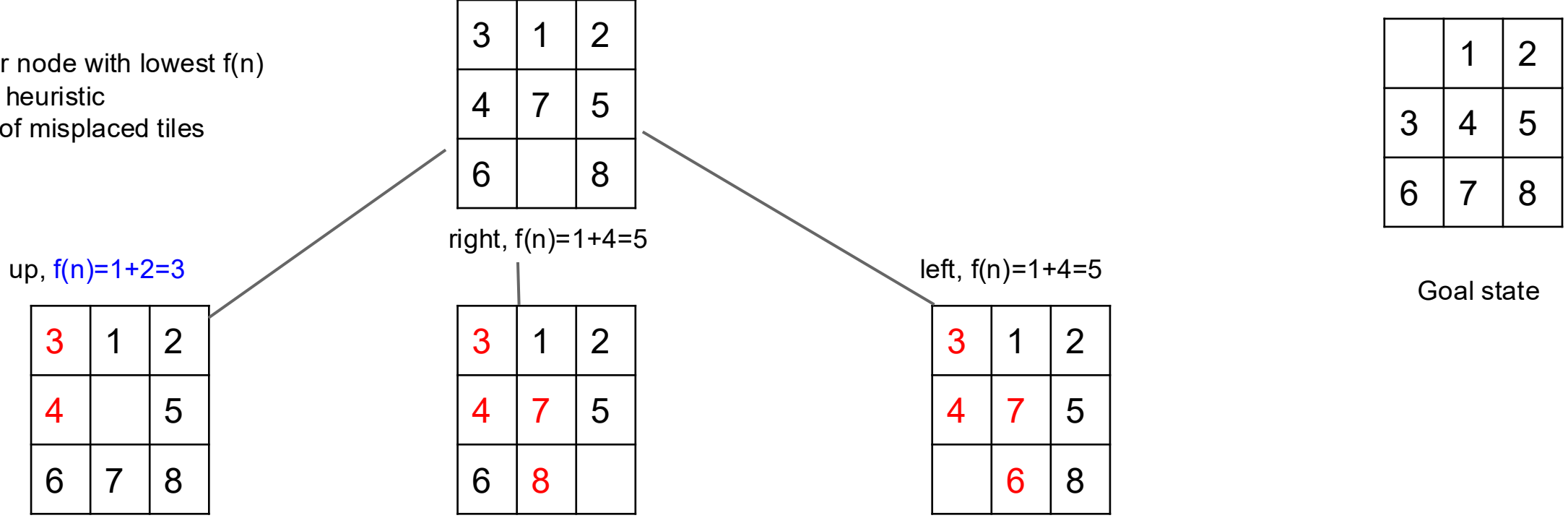
NOTE: We ignore the blank in these computations

Trace A*

A* expands frontier node with lowest f(n)

f(n) = costSoFar + heuristic

heuristic: Number of misplaced tiles

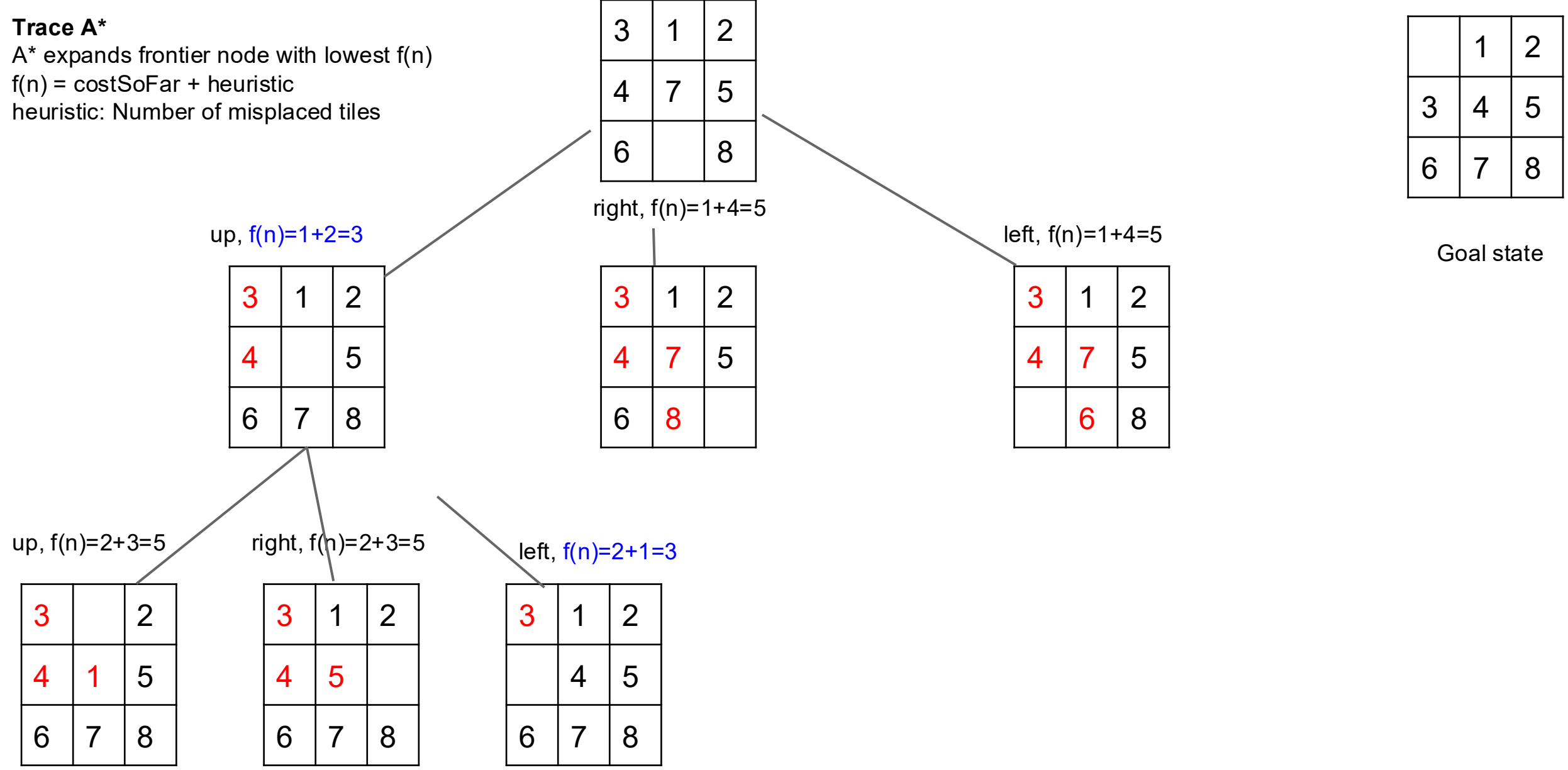


Trace A*

A* expands frontier node with lowest f(n)

f(n) = costSoFar + heuristic

heuristic: Number of misplaced tiles



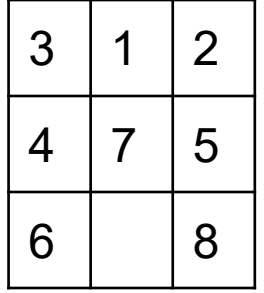
Goal state

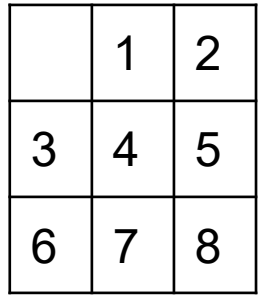
Trace A*

A* expands frontier node with lowest f(n)

f(n) = costSoFar + heuristic

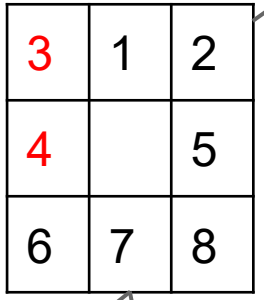
heuristic: Number of misplaced tiles



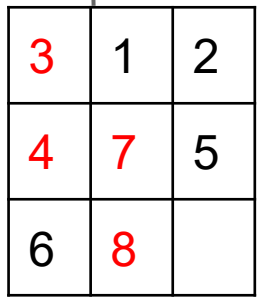


Goal state

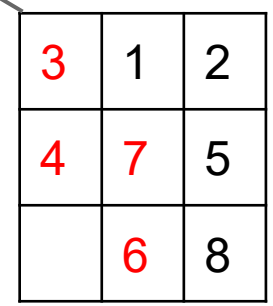
up, f(n)=1+2=3



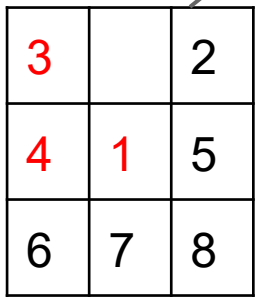
right, f(n)=1+4=5



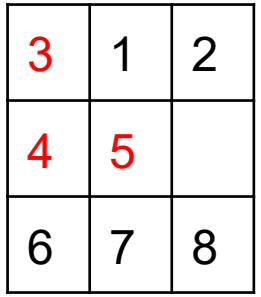
left, f(n)=1+4=5



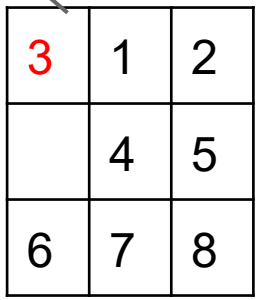
up, f(n)=2+3=5



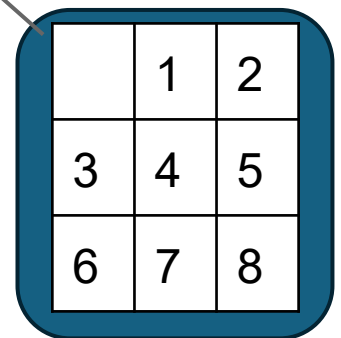
right, f(n)=2+3=5



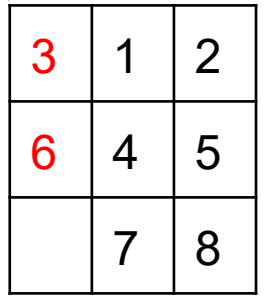
left, f(n)=2+1=3



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



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

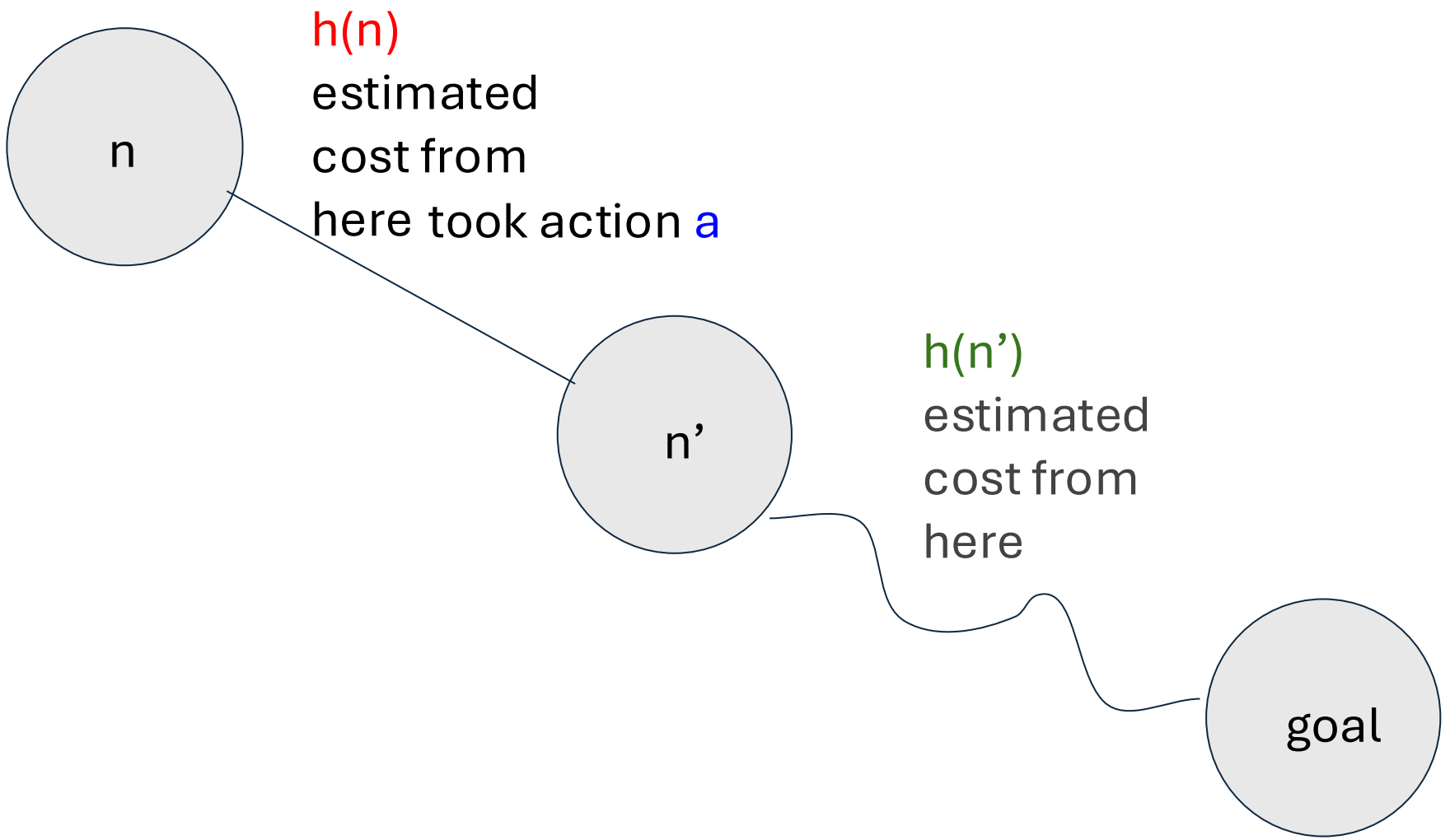


A* is complete and optimal

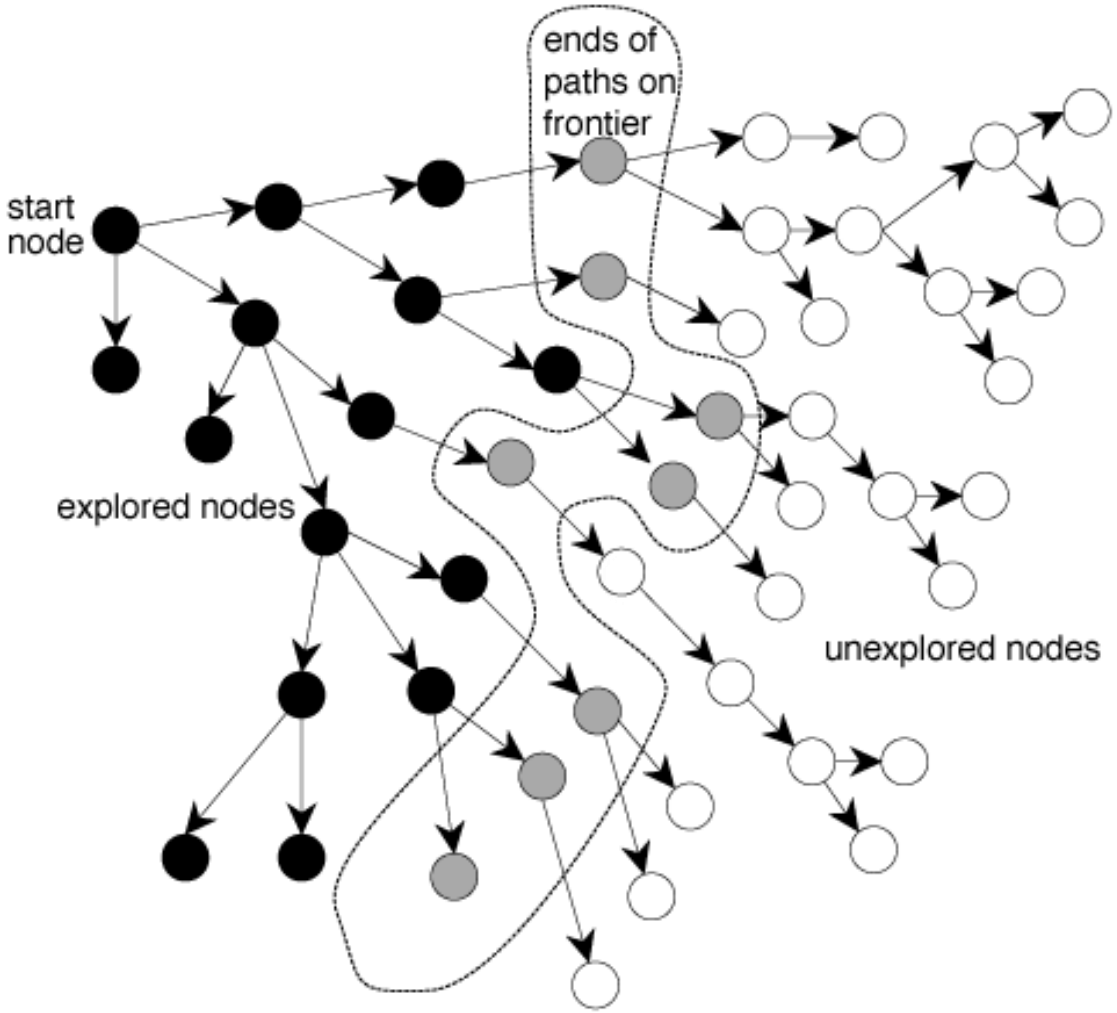
- When heuristic is admissible
  - Never overestimates the cost to reach the goal
  - Why are the misplaced tiles and Manhattan distances admissible?
- When heuristic is consistent
  - Let n be a node, n' be a successor of n, a be the action that leads from n to n'
  - h(n) <= cost(n, a, n') + h(n')
- 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')



Visualizing how A* operates



- Let C* be the cost of the optimal solution path
- A* will explore all nodes with f(n) < C*
- A* will explore some nodes with f(n) == C* until goal is found

A* is optimally efficient

- For any given consistent heuristic, no other optimal algorithm will expand fewer nodes
- Any algorithm that does NOT expand all nodes with f(n) < C* runs the risk of missing the optimal solution
- Only possible difference could be in which nodes are expanded when f(n) == C*

Choosing the best heuristic

- Highest estimate that is still admissible and consistent will lead to the best effective branching factor for A*
- For 8 puzzle, Manhattan distances heuristic is better than tiles misplaced heuristic

Exercise: Trace A* using number of tiles misplaced heuristic

1	4	2
3		5
6	7	8

Initial state

	1	2
3	4	5
6	7	8

Goal state