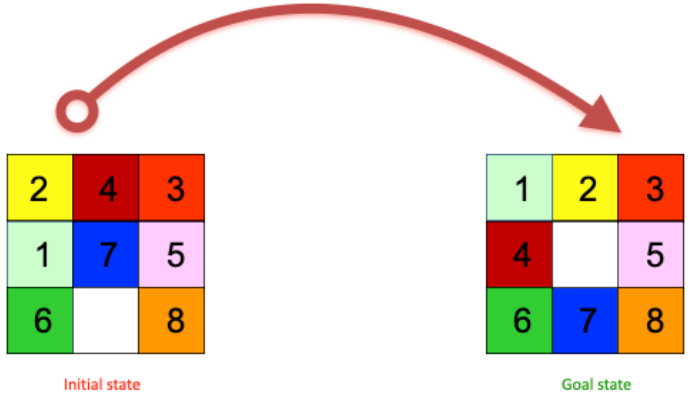
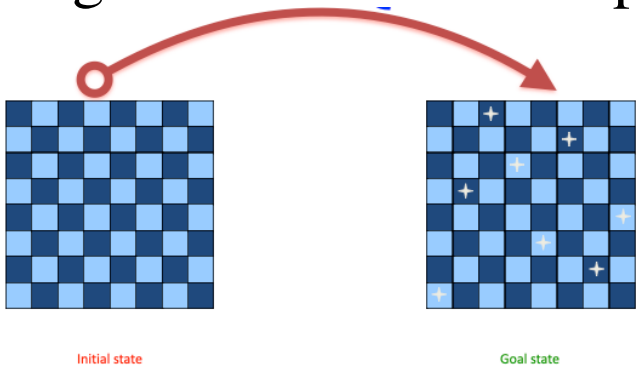
CS143: Artificial Intelligence



Search algorithms to solve AI problems



Drake UNIVERSITY

Recap

- Project Assignment 1
  - Discussed street sweeping reflex agents
  - Improve it by making it more intelligent ie model-based agent

- Posing AI problem solving as search
  - Introduced puzzle as an AI problem to solve
  - Discussed how to use mathematical notations to create abstract version of the problem

Recap: Posing AI problem solving as searches

Recap: Old puzzle

Problem Statement:

- There's a man, a goat, a wolf, and a cabbage on one side of a river.
- There's a boat, and it can only hold the man plus another thing.
- Unfortunately, the man can't leave the wolf and goat unattended or else the wolf will eat the goat.
- Similarly he can't leave the cabbage and goat unattended or else the goat will eat the cabbage.
- How can the man get everything (including himself) across the river safely?

Recap: Exploring Choices

- Many AI problems that seem to require intelligence usually require exploring multiple choices.

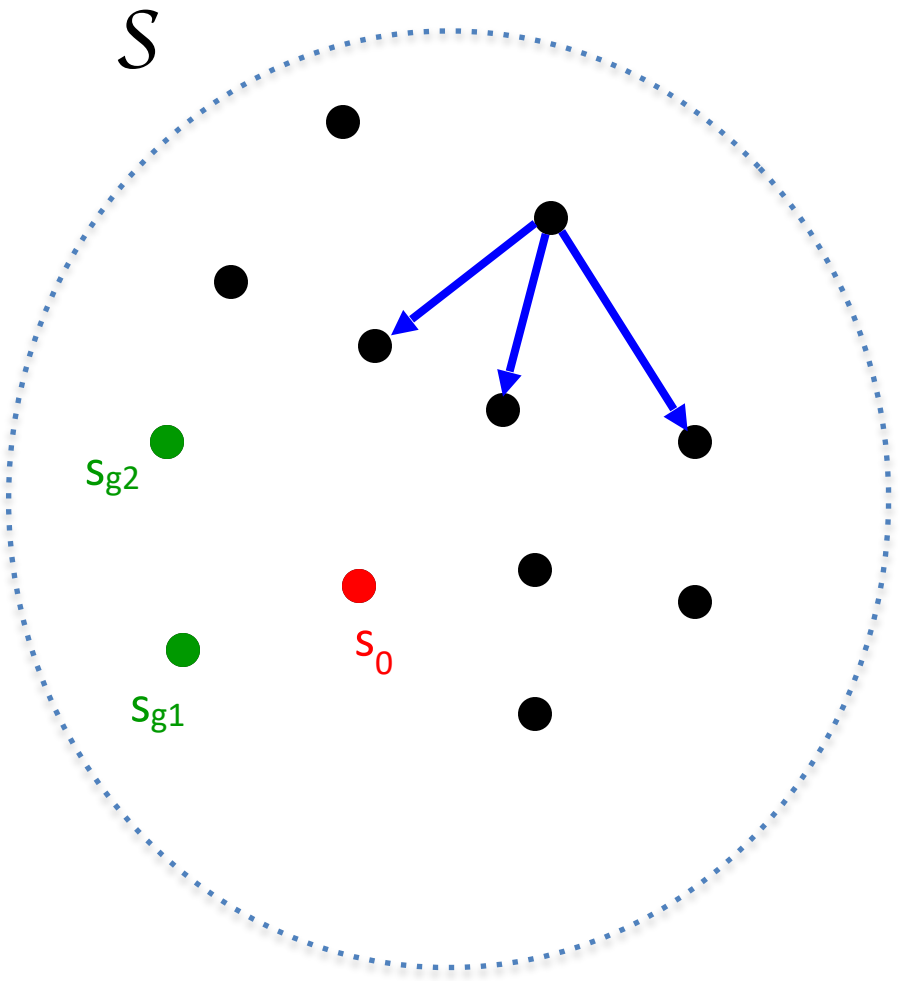
- Search: a systematic way of exploring choices.

Recap: Exploring Choices

- In other words, if we approach a new problem correctly, we don't need a specific new algorithm to solve:
  - Eg, the problem of the cabbage, man, goat, and wolf can be solved with **an existing search algorithm**

- We just need to think about the "cabbage, man, goat, and wolf" problem in the right way, and then we can use existing algorithms to solve it automatically.

Recap: 5 parts of abstraction to define a search problem



- State space $\mathcal{S}$
- Initial state s_0
- Successor function:

  $x \in \mathcal{S} \rightarrow$ SUCC(s): that encodes

possible transitions of the system from state s to another state x

- Set of goal states:

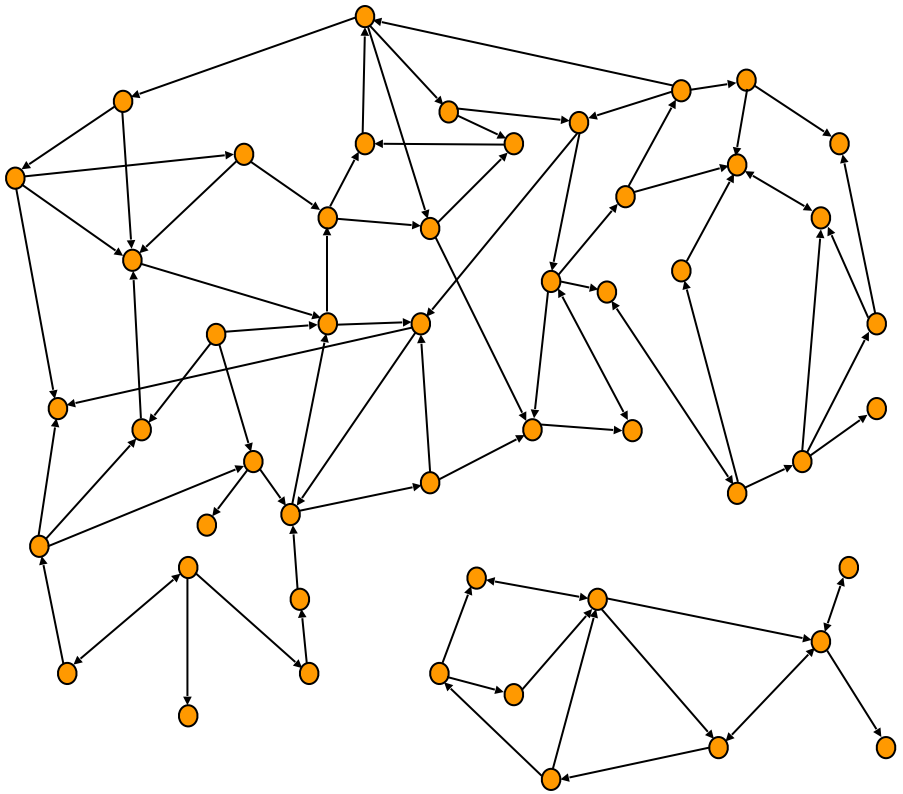
  $x \in \mathcal{S} \rightarrow$ GOAL?(x) = T or F

- Cost: that calculates how "expensive" a given set of moves is

Recap: State Graph

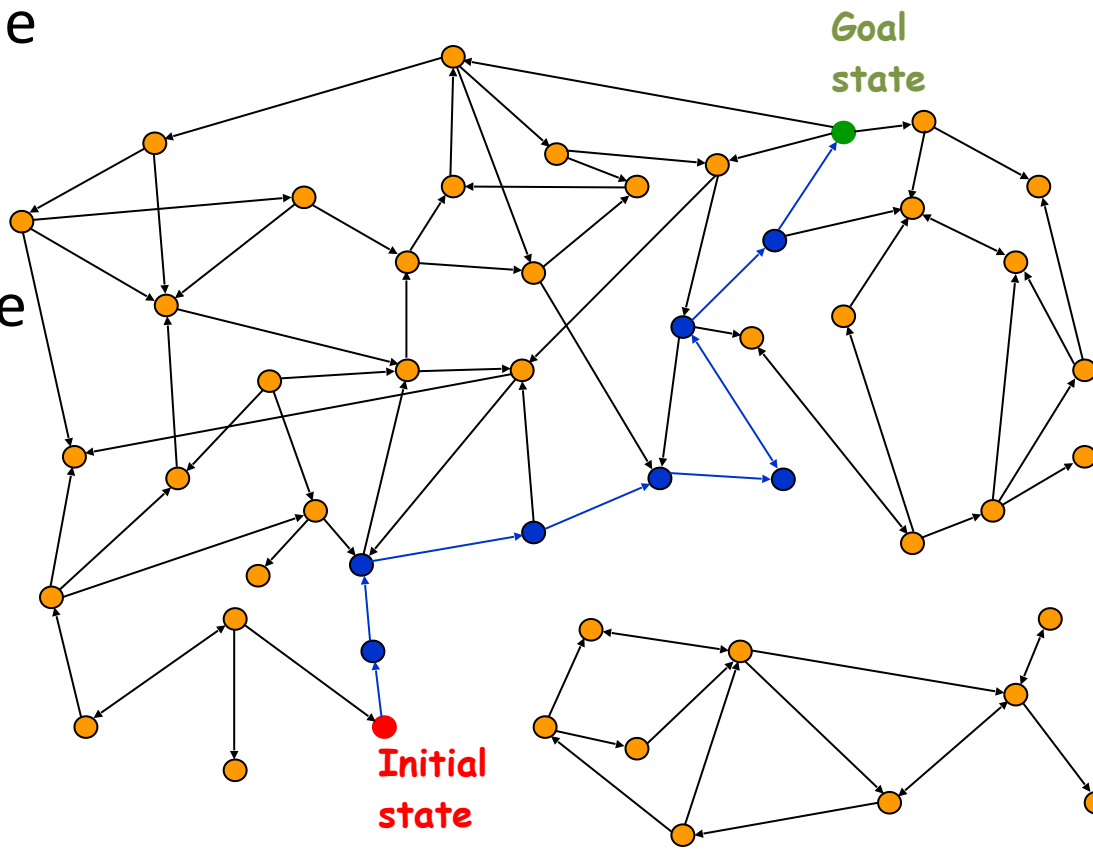
- Each state is represented by a distinct node
- An edge connects a node s to a node x if

  $x \in S \rightarrow succ(s) \in 2^S$
- The state graph may contain more than one connected component



Recap: Solution to the Search Problem

- A solution is a path from the initial to any goal node
- Path cost is sum of the edge costs along the path
- An optimal solution has minimum cost
  - There might be no solution!
  - There might be multiple solutions



Abstraction for Old Puzzle

C
G
M
W

Initial state

Goal state

C
G
M
W

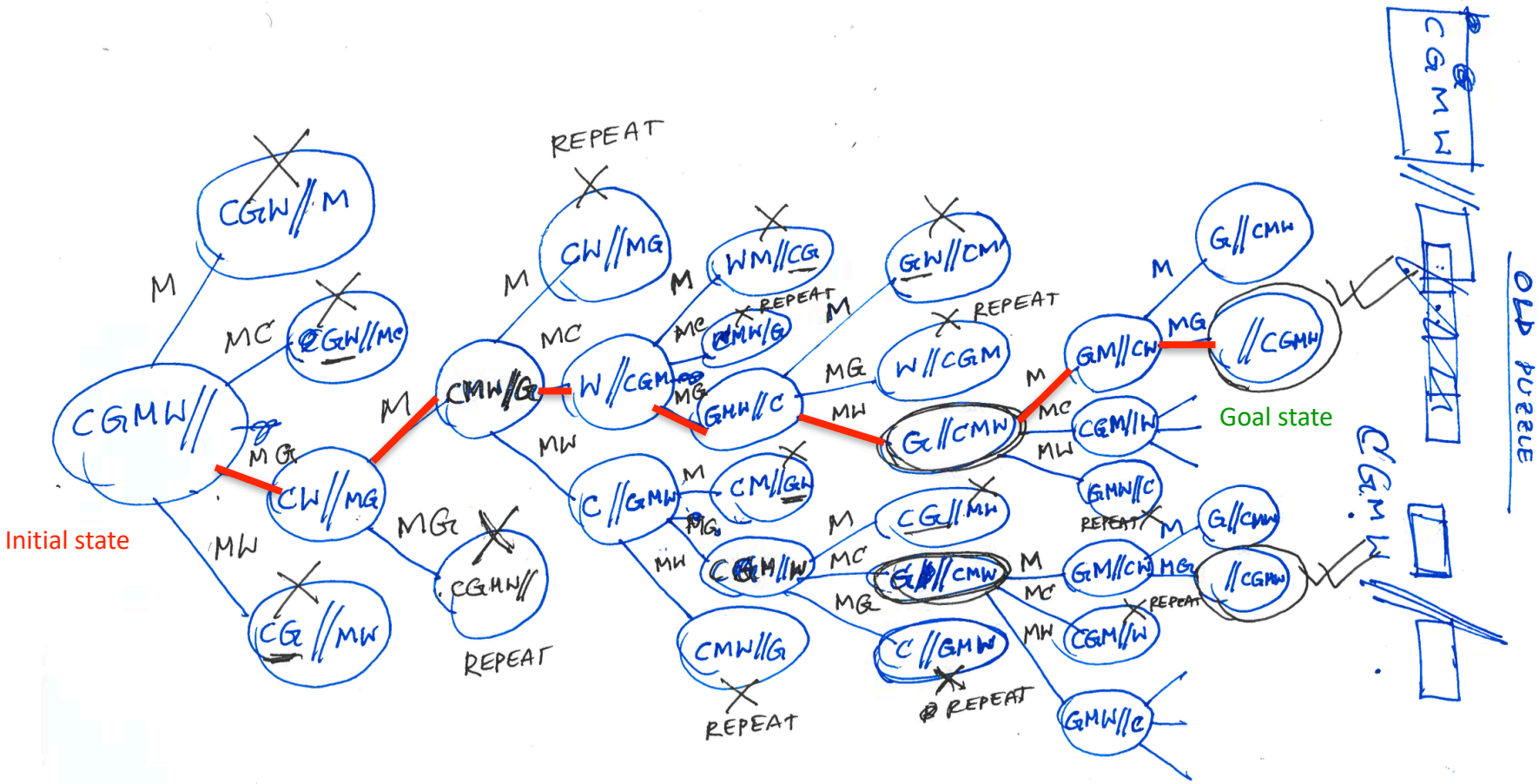
State: Any arrangement of 4 symbolic representation C, G, M, W and a slash symbol / to represent the river

Successor function: given by available actions taken by the man: M, MC, MG, and MW

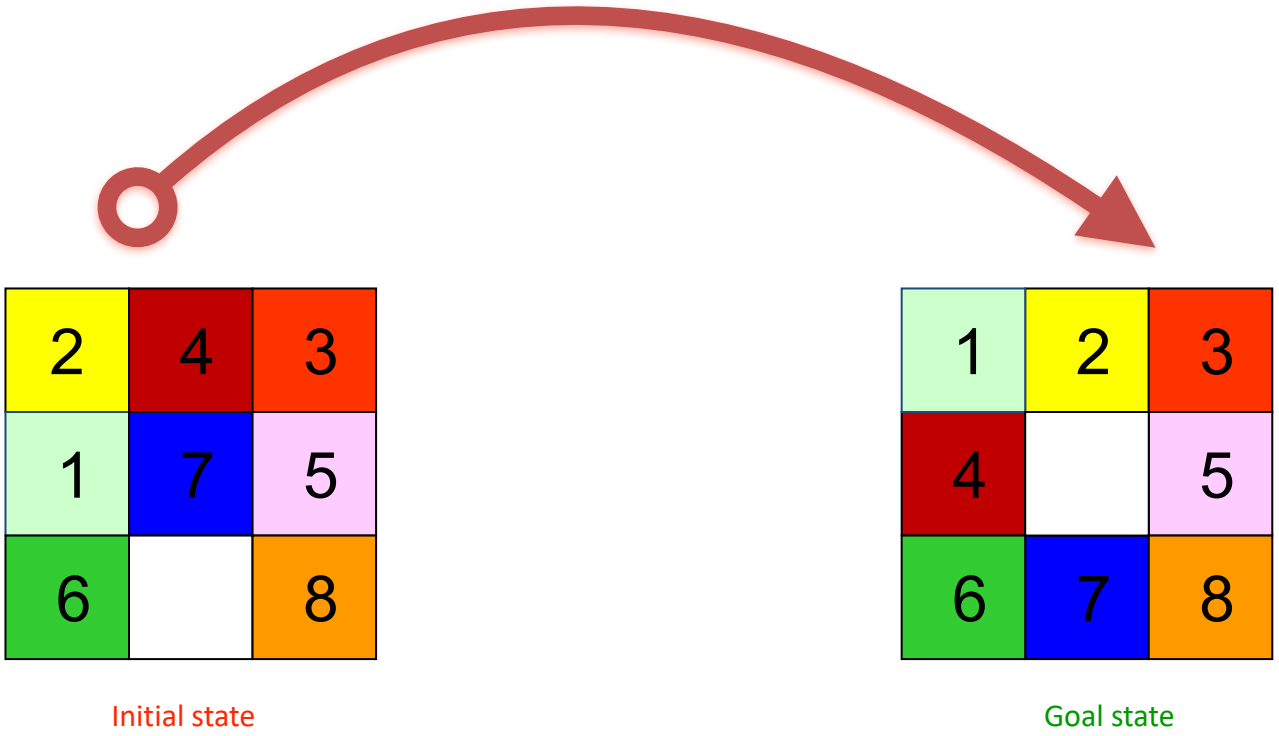
Cost: How many moves were performed

Solution to Old Puzzle

- One path to the solution (from the starting state to goal state)

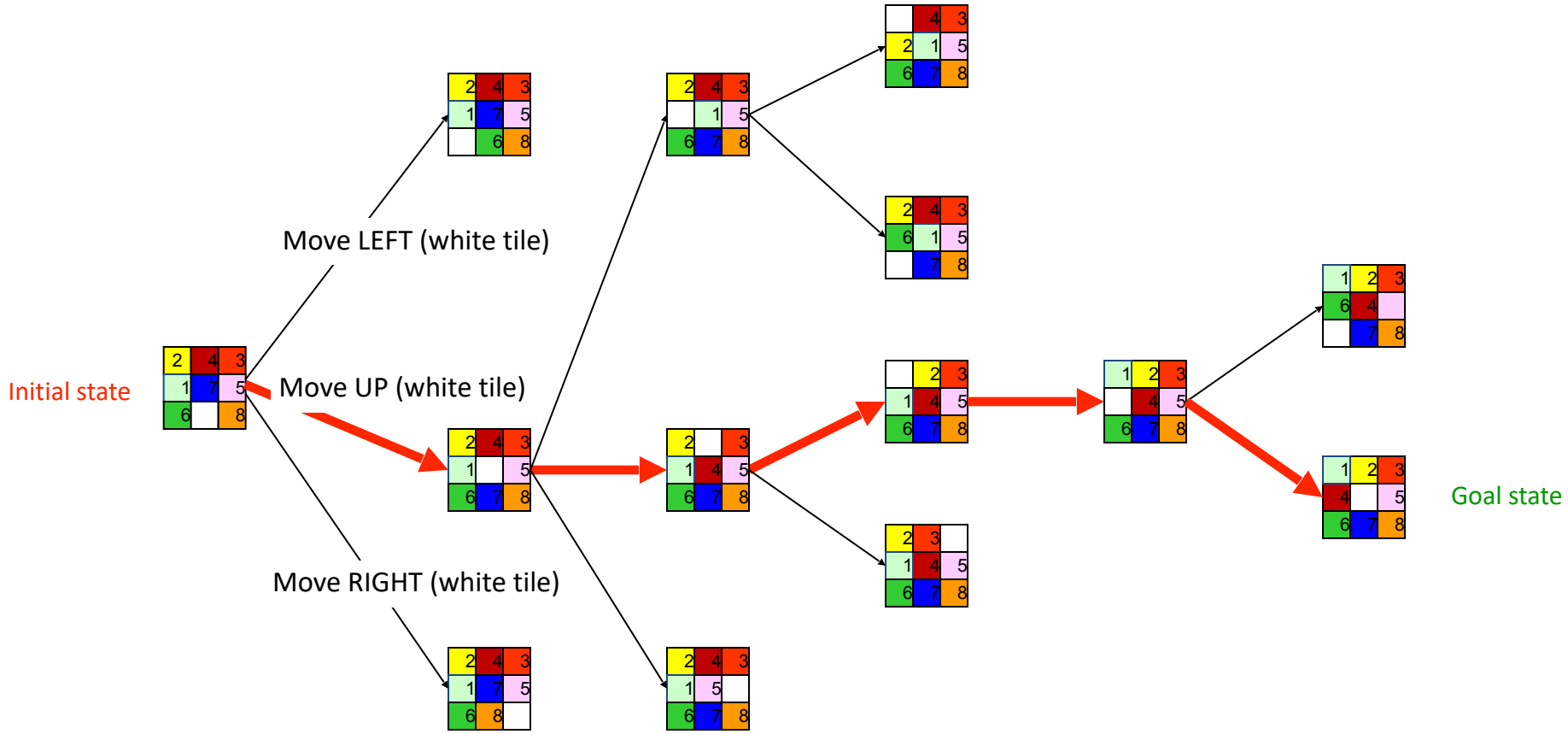


Abstraction for 8-Puzzle

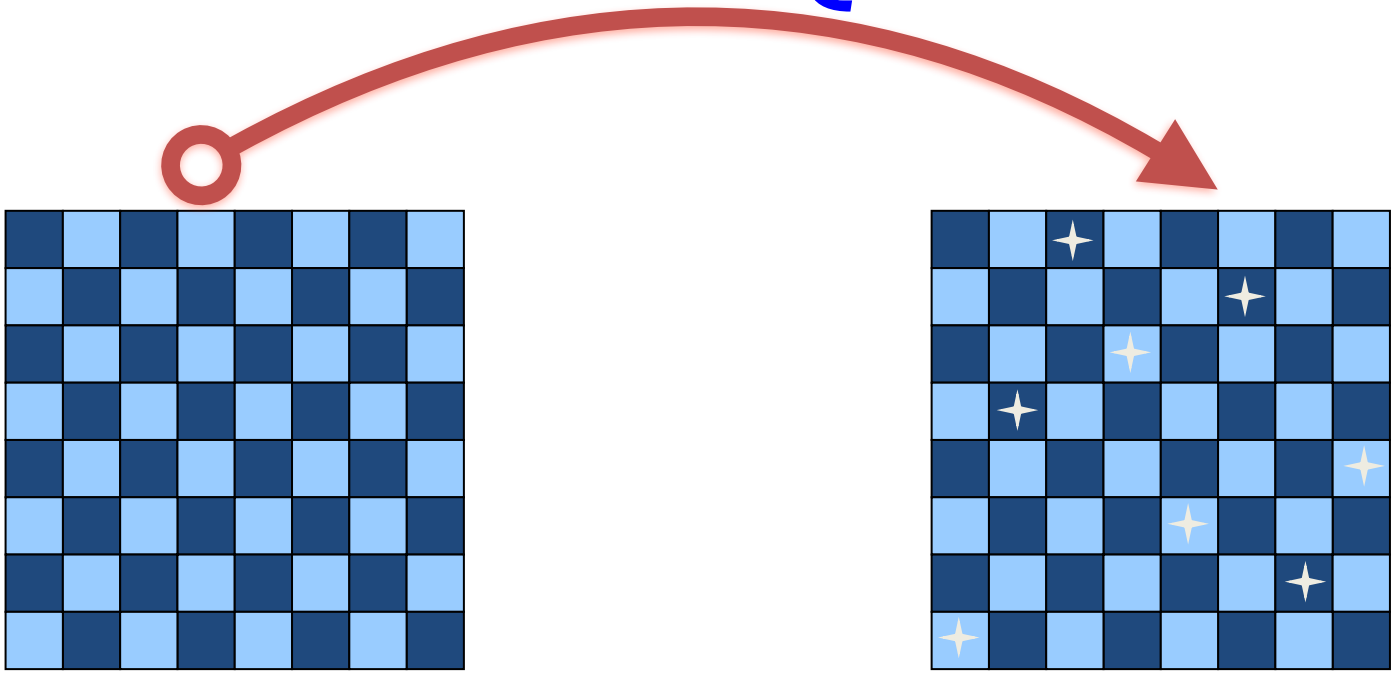


State: Any arrangement of 8 numbered tiles and an empty tile on a 3x3 board
Successor function: given by available actions (sliding the white tile) L, R, U, D.
Cost: How many moves were performed

Solution to 8-Puzzle



Abstraction for 8-Queen Problem



Initial state

Goal state

State: Any configuration of 0-8 non-conflicting queens in columns starting from left

Successor function: place queen in leftmost empty column so that there is no conflict

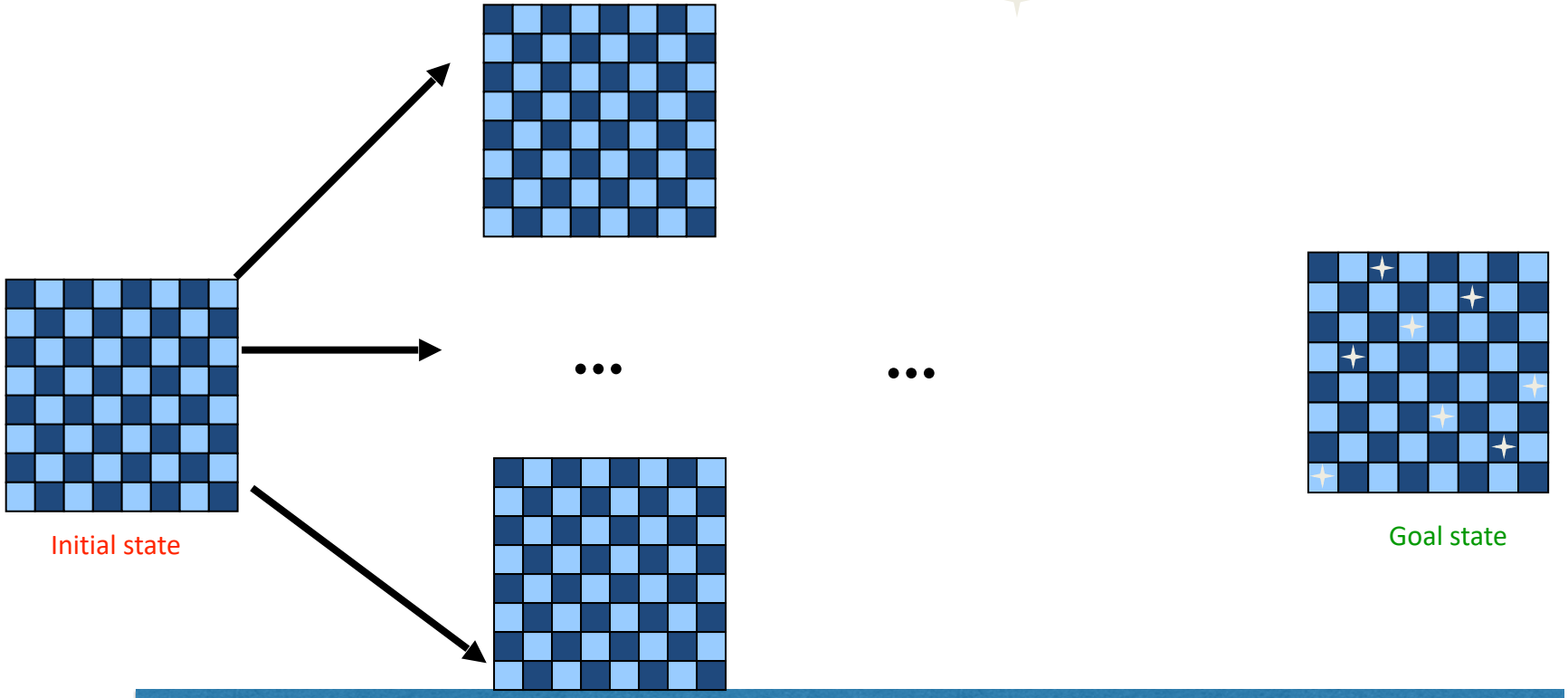
Cost: How many moves were performed

Recall that conflicts happen in a chessboard when two queens are in the:

- same row or
- same column or
- same diagonal

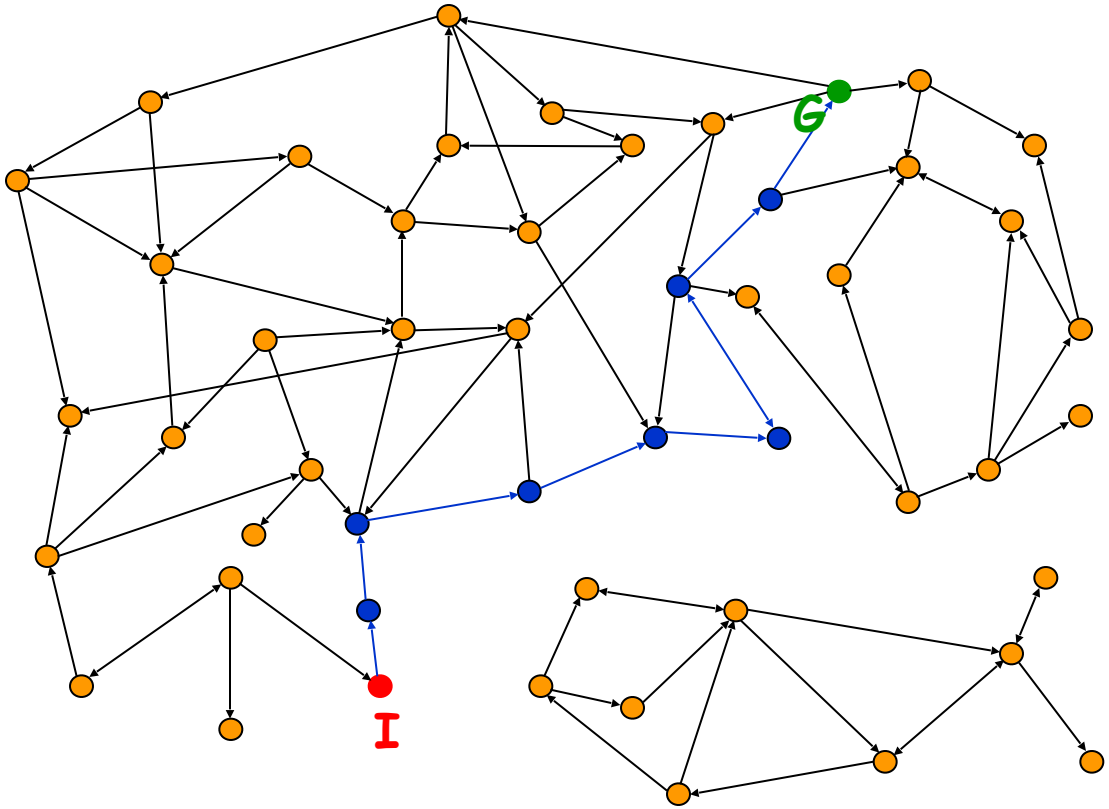
Exercise: Generate the states for the following formulation of 8-Queens problem

- **State:** configuration of 0-8 non-conflicting queens in columns starting from left
- **Initial state:** 0 queens
- **Successor function:** place queen in leftmost empty column so that there is no conflict
- **Goal :** State with 8 queens



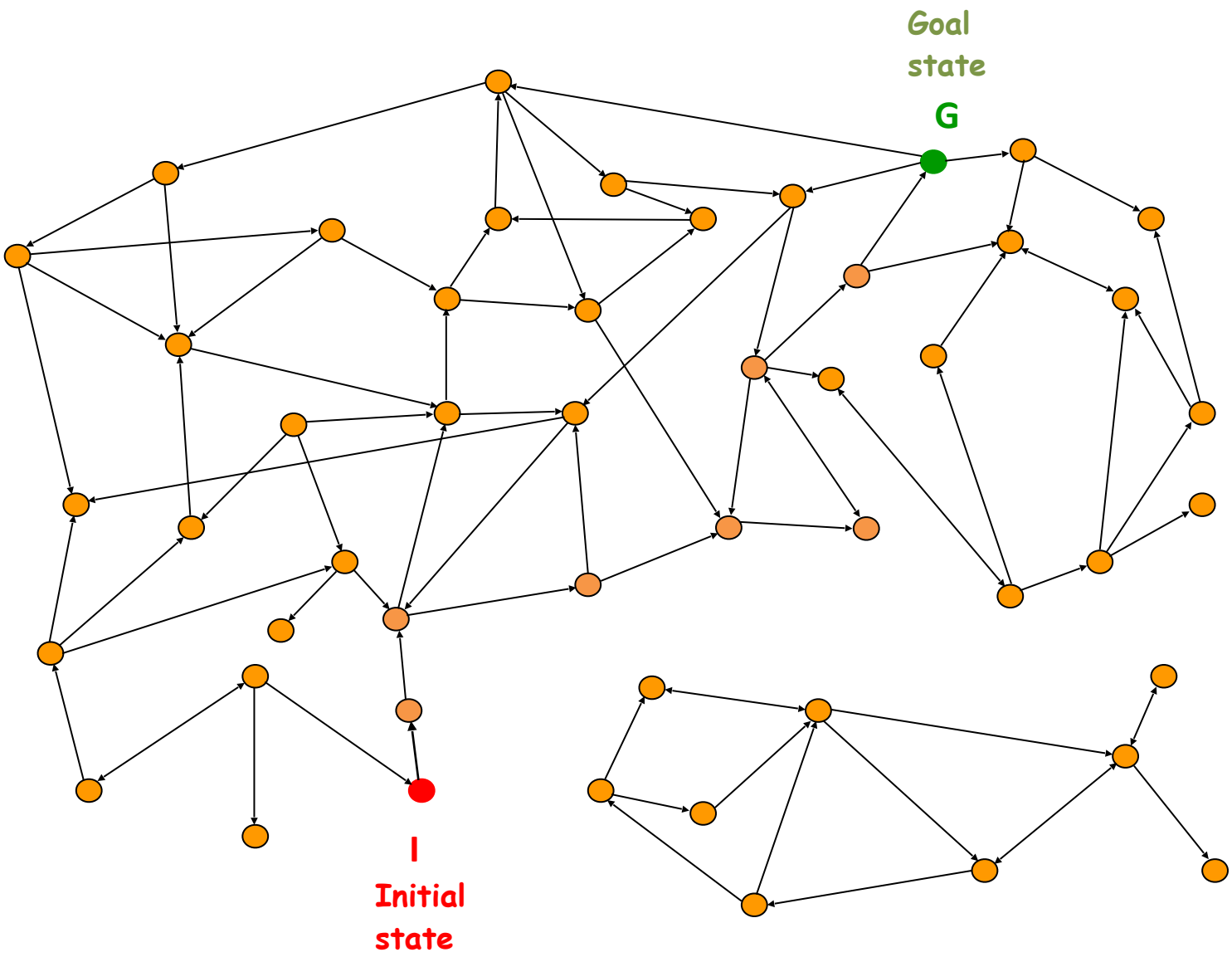
Pathless Problems

- Sometimes the path doesn't matter
- A solution is any goal node
- Edges represent potential state transformations
  - E.g. 8-queens, Map coloring



Search Algorithms

Search Algorithm

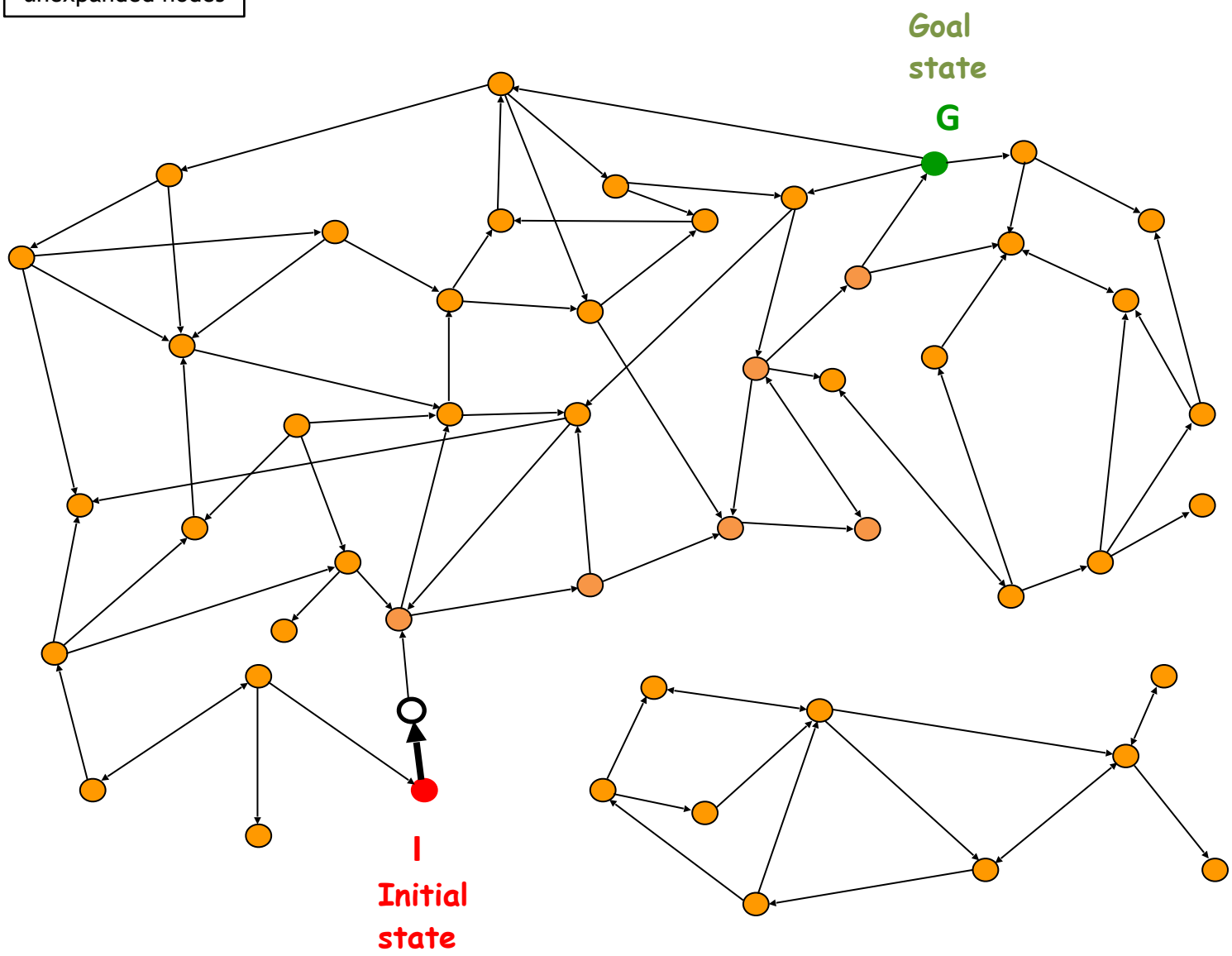


initial node

goal node

unexpanded nodes

Search Algorithm

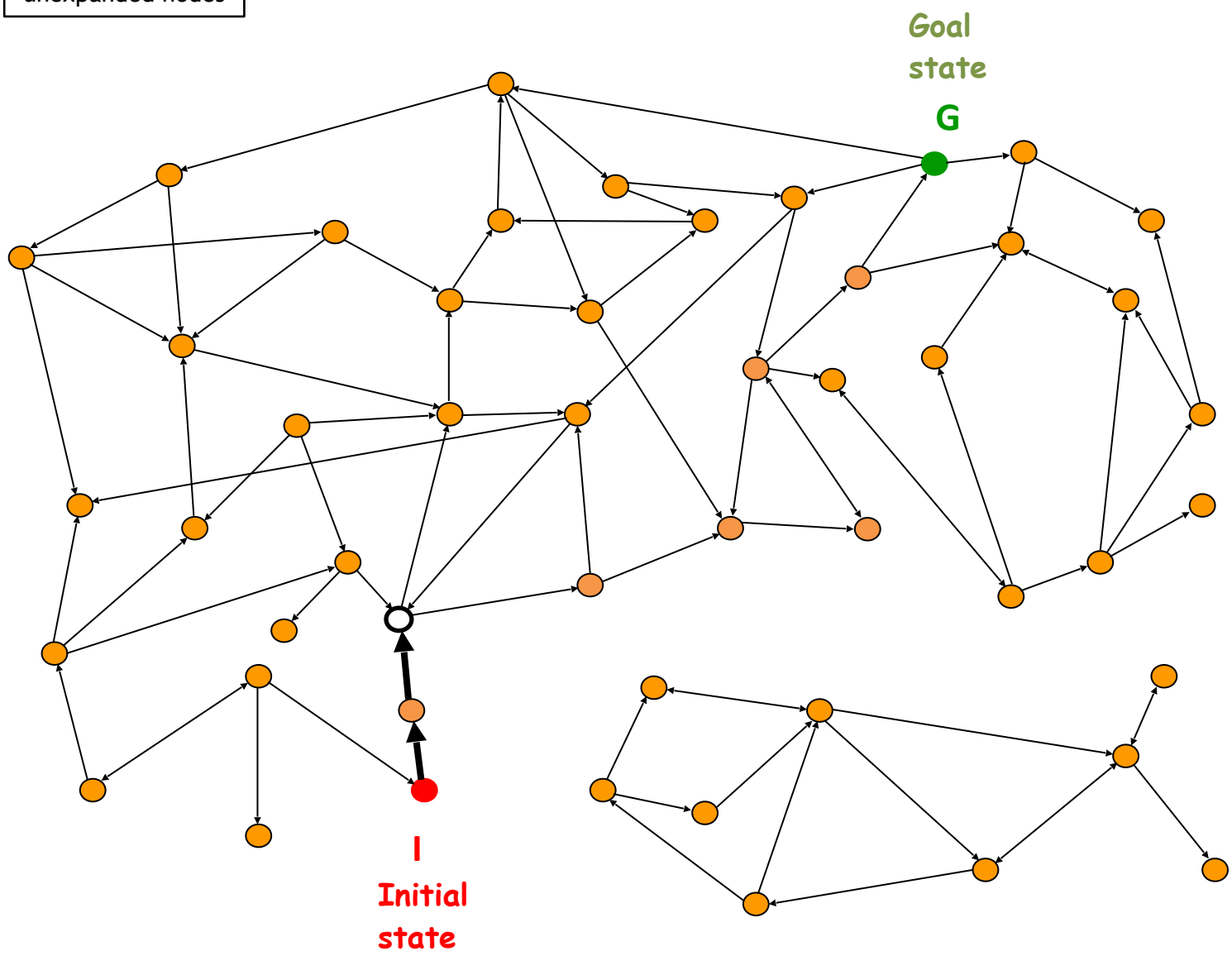


initial node

goal node

unexpanded nodes

Search Algorithm

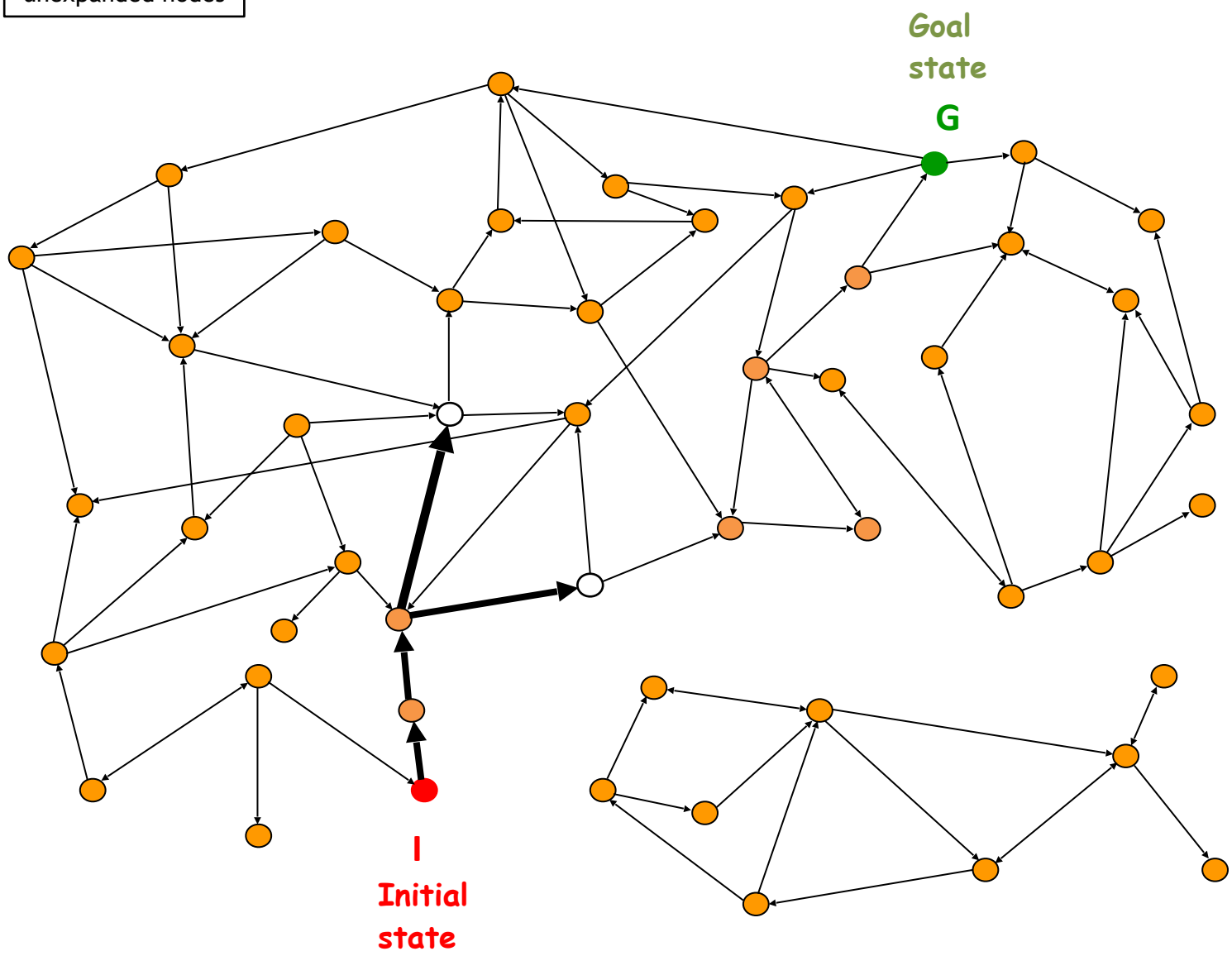


initial node

goal node

unexpanded nodes

Search Algorithm

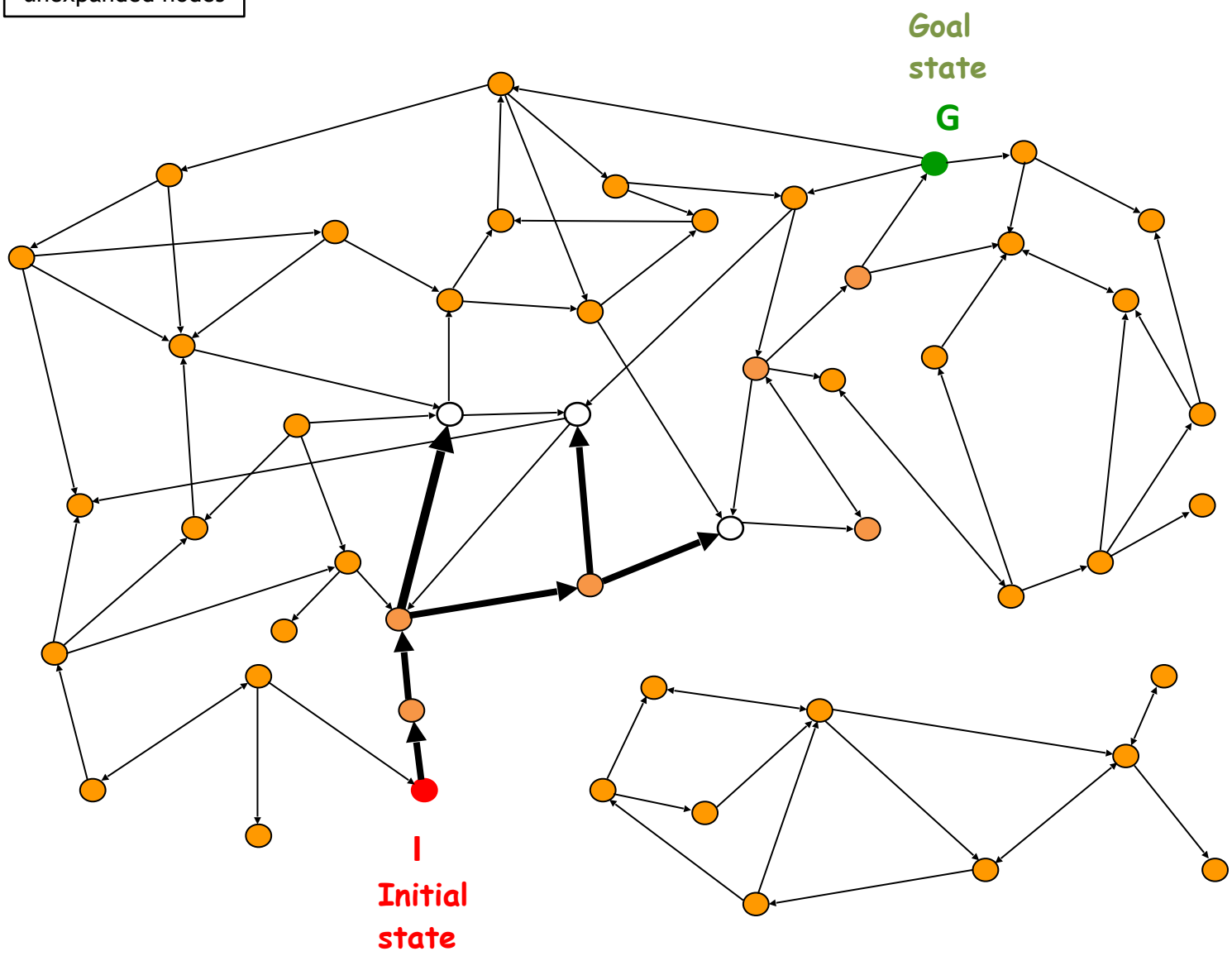


initial node

goal node

unexpanded nodes

Search Algorithm

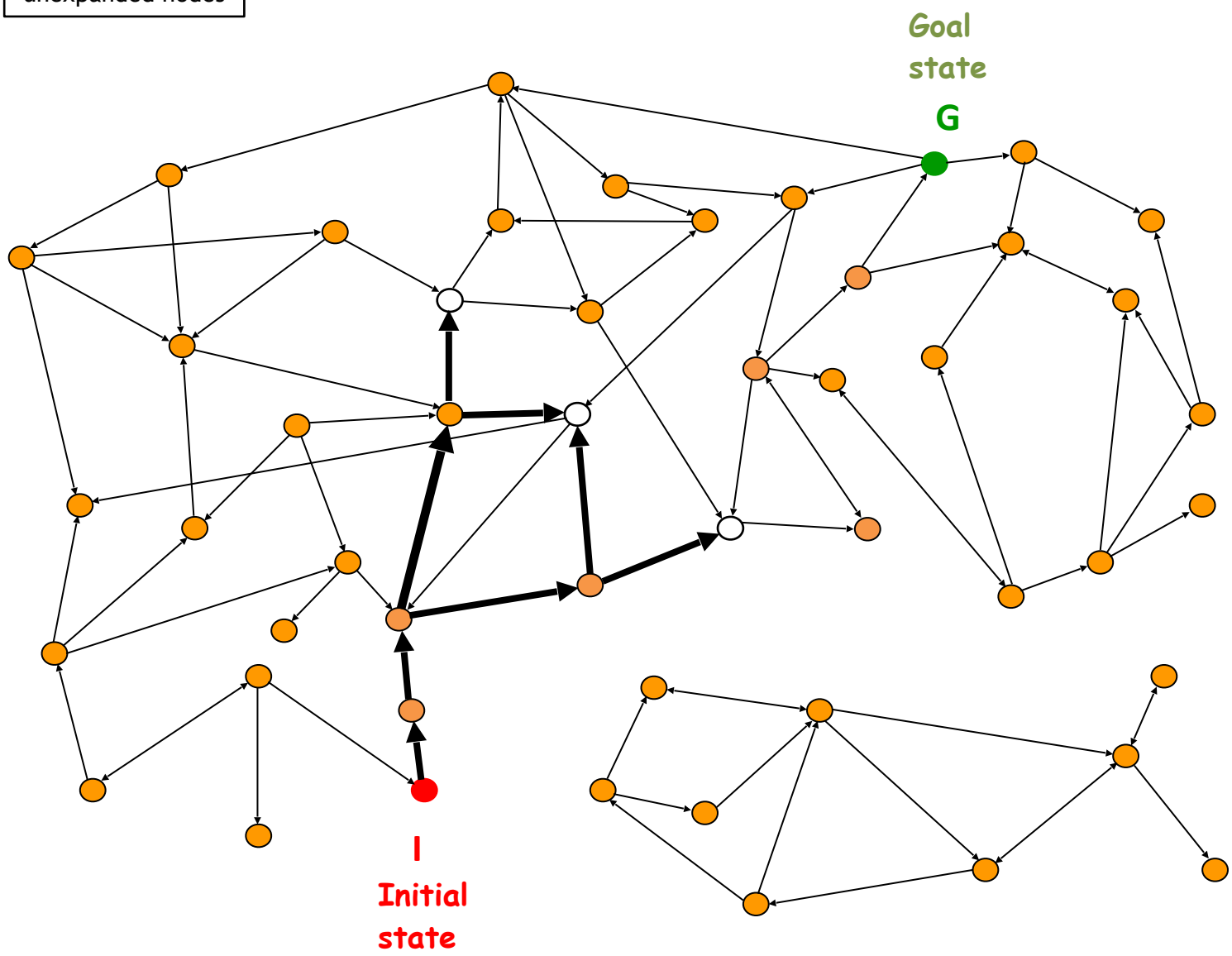


initial node

goal node

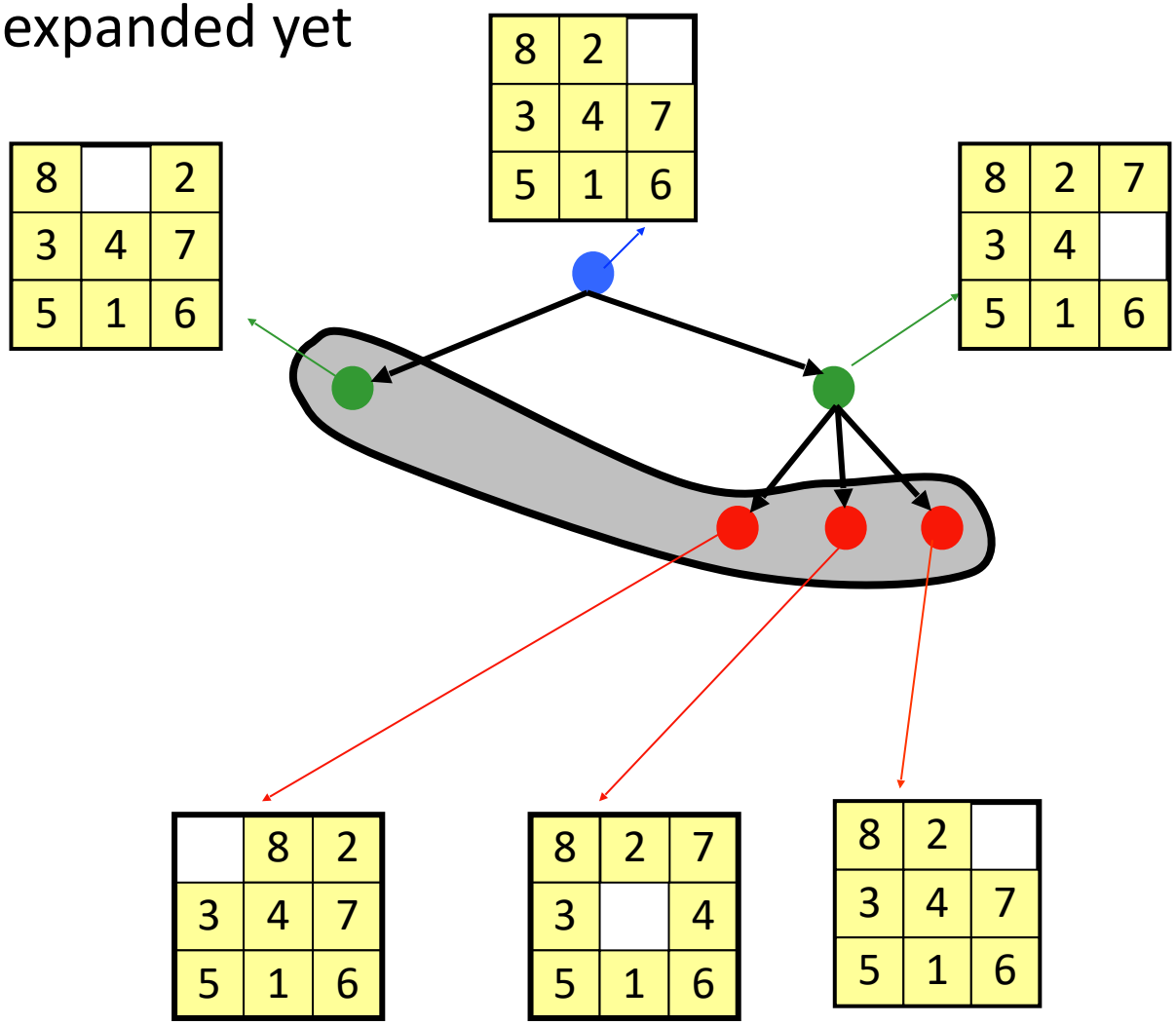
unexpanded nodes

Search Algorithm



Data Structure: frontier

- The frontier (or fringe) is the set of all search nodes that haven't been expanded yet

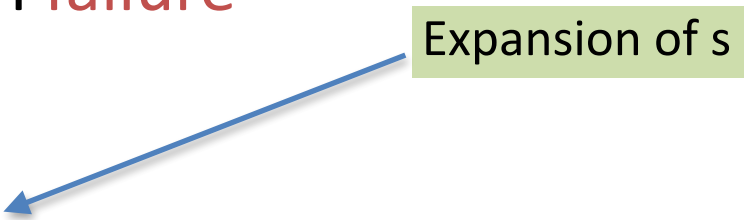


Pseudocode for Search Algorithm from Russel & Norvig Textbook

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Concrete Instantiation: Search Algorithm #1

1. If goal-state == initial-state then return initial-state
2. INSERT(initial-state, frontier)
3. repeat:
4. If empty(frontier) then return failure
5. s ← REMOVE(frontier)
6. for every state s' in SUCC(s):
7. If goal-state == s' then return s' and/or path
8. INSERT(s', frontier)

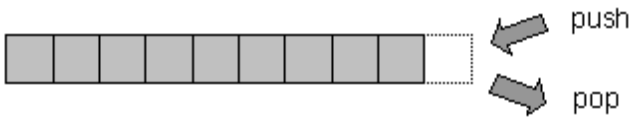


Search Strategy

- frontier is a data structure with two operations:
  - INSERT(node, frontier)
  - REMOVE(frontier)
- Surprisingly, this algorithm supports many different search strategies depending on implementation of frontier
  - yet another example of the power of abstraction

Stacks, Queues, Priority Queue

- Stack



- Queue

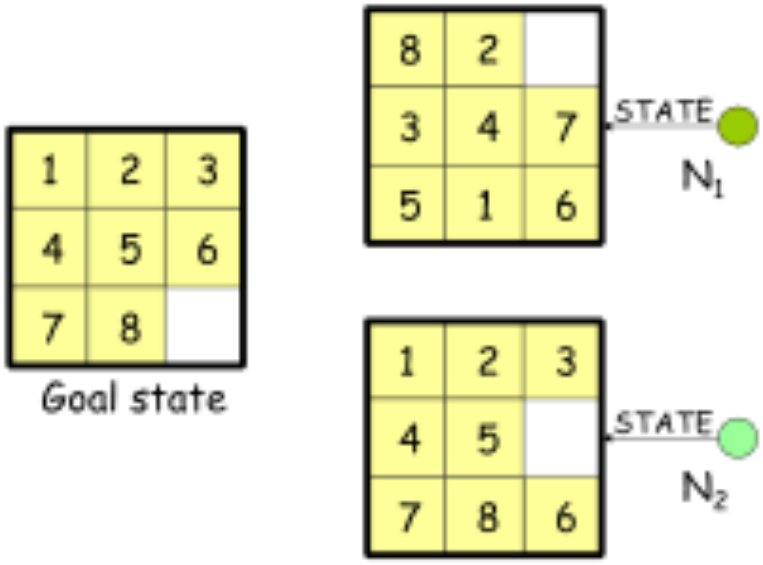
- Priority Queue
  - You put (item, priority) pairs into queue
  - You remove the highest-priority item from the queue

Blind vs. Heuristic Search Strategies

- Blind (or un-informed) strategies do not use properties of states to order frontier. All states are treated the same.
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Iterative Deepening Search (IDS)

- Heuristic (or informed) strategies order frontier so that more "promising" states are considered first
  - Uniform Cost Search (UCS)
  - Greedy Search
  - A* Search

Example



For a blind strategy, N_1 and N_2 are just two nodes (at some position in the search tree)

For a heuristic strategy, N_2 seems more promising than N_1 (fewer misplaced tiles)

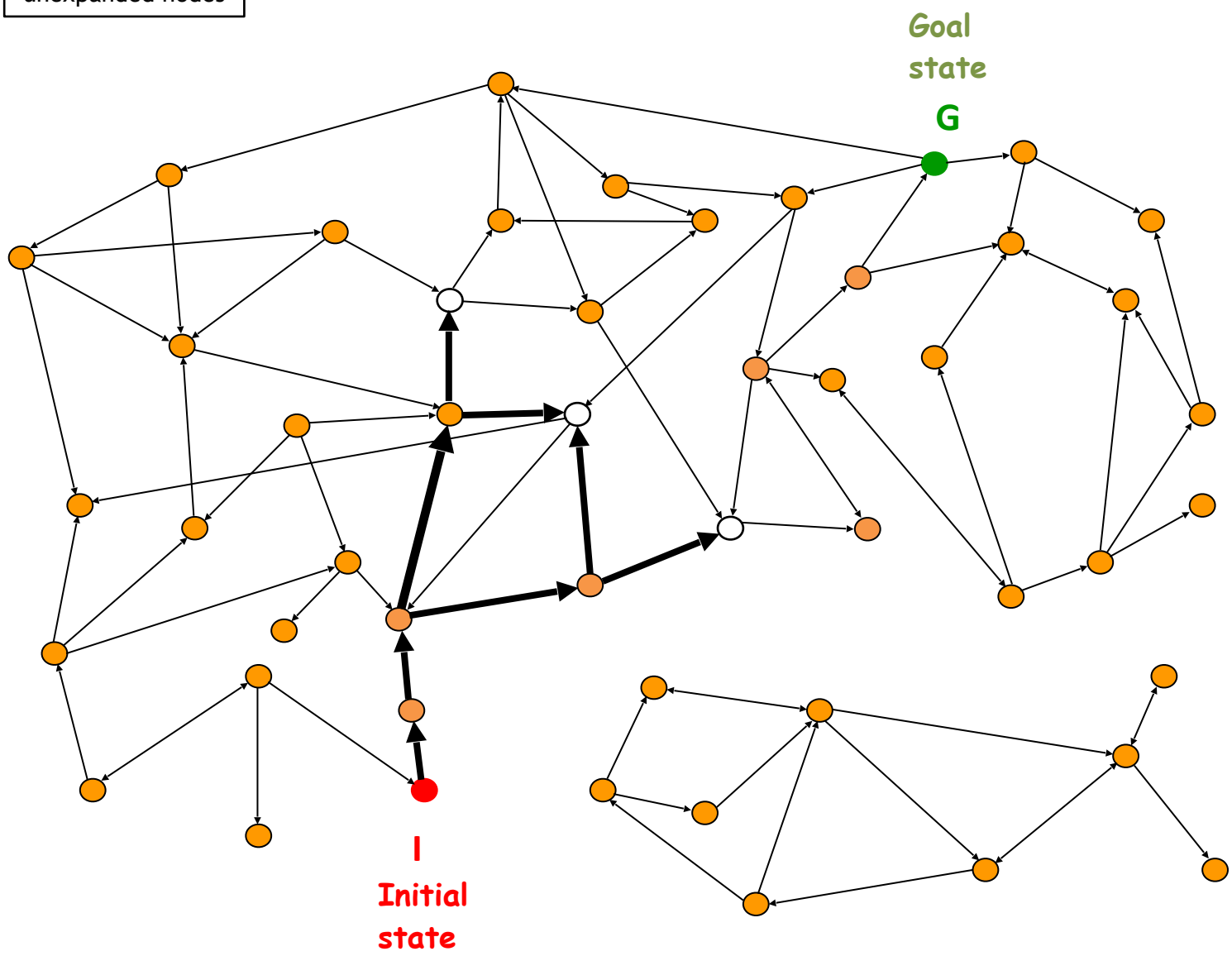
Illustrating Breadth-First Search Algorithm: A Blind Strategy

initial node

goal node

unexpanded nodes

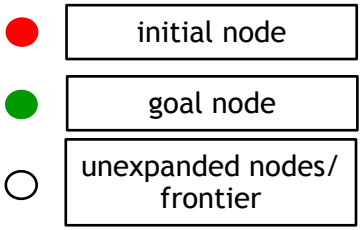
Search Algorithm

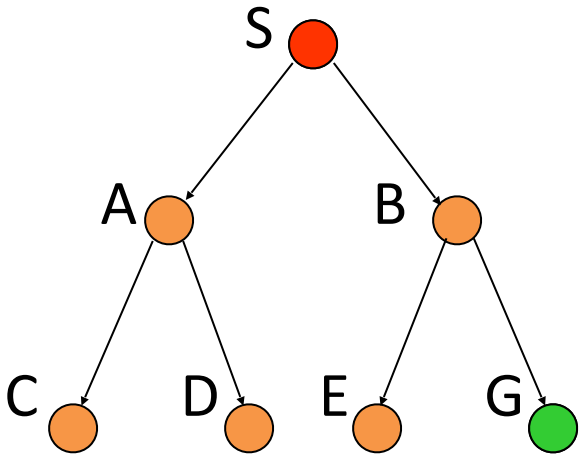


Illustrating Breadth-First Search (BFS) Algorithm

- Nodes are inserted & removed on opposite sides of frontier
  - Should we use Stack or Queue for frontier?

- Let's see the execution of BFS
  - starting at initial node **S** and searching for goal node **G**





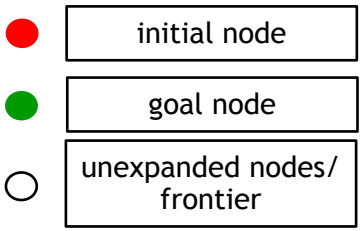
Illustrating Breadth-First Search (BFS) Algorithm

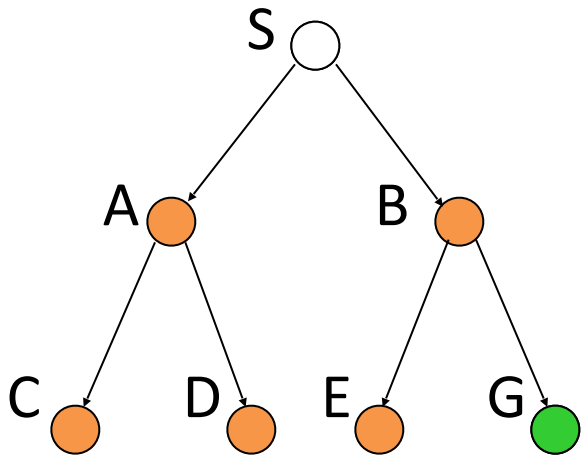
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

S						

append: S





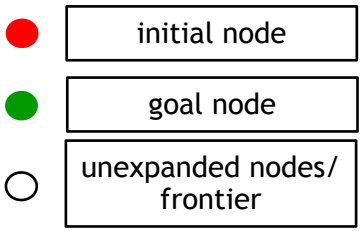
Illustrating Breadth-First Search (BFS) Algorithm

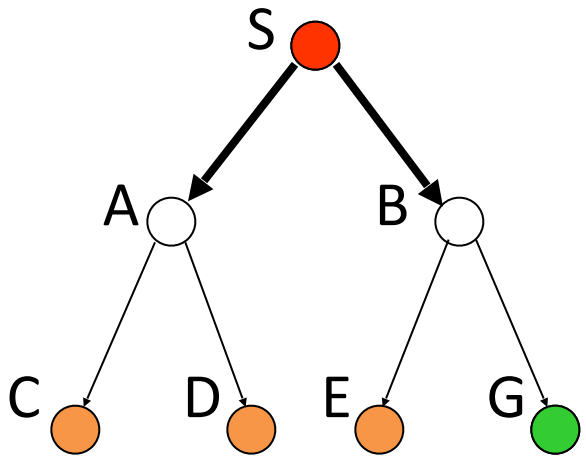
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

	A	B				

remove: S
append: A, B





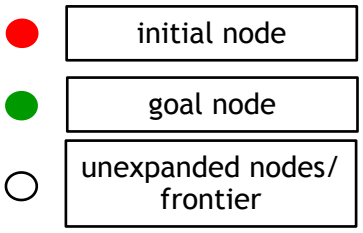
Illustrating Breadth-First Search (BFS) Algorithm

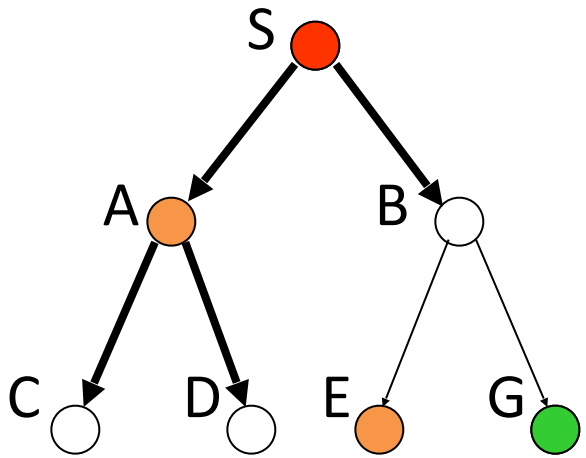
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

		B	C	D		

remove: A
append: C, D





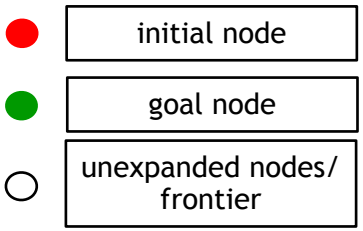
Illustrating Breadth-First Search (BFS) Algorithm

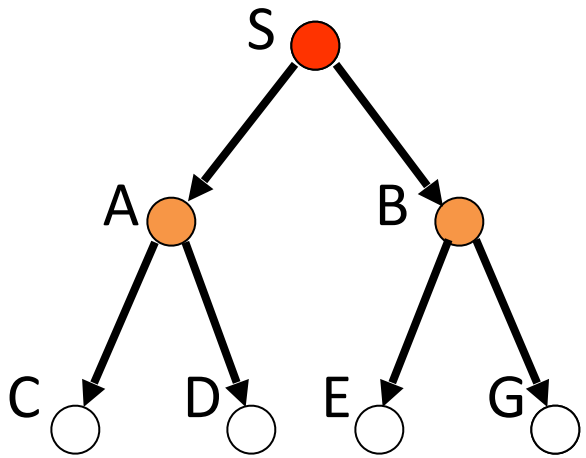
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

			C	D	E	G

remove: B
append: E, G





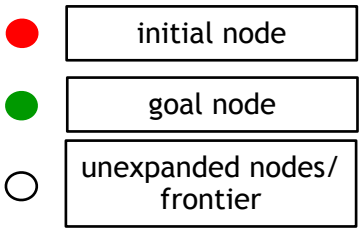
Illustrating Breadth-First Search (BFS) Algorithm

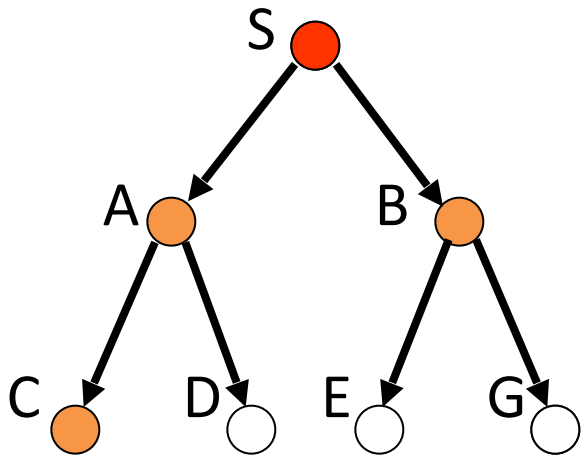
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

				D	E	G

remove: C





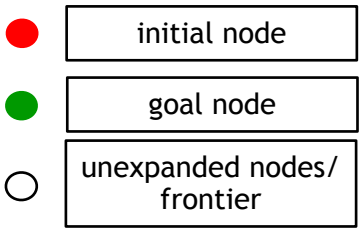
Illustrating Breadth-First Search (BFS) Algorithm

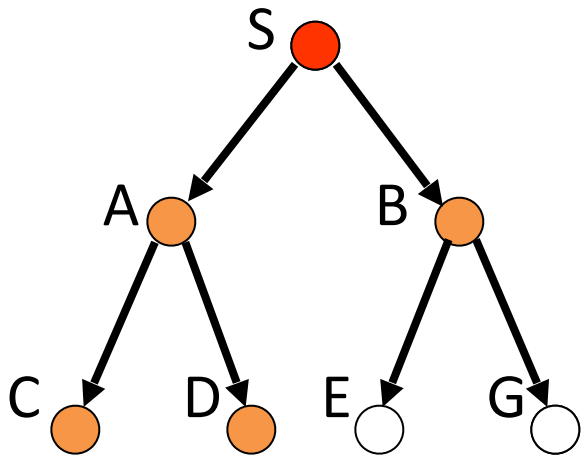
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

					E	G

remove: D



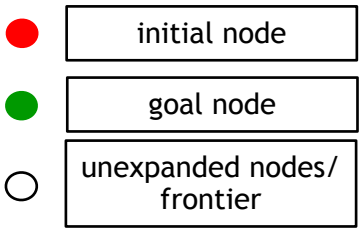


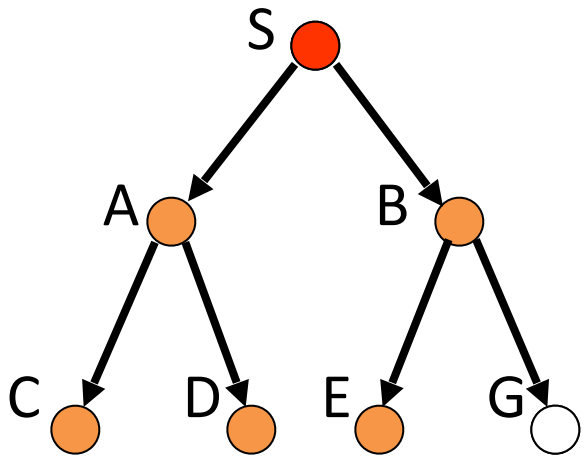
Illustrating Breadth-First Search (BFS) Algorithm

- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):

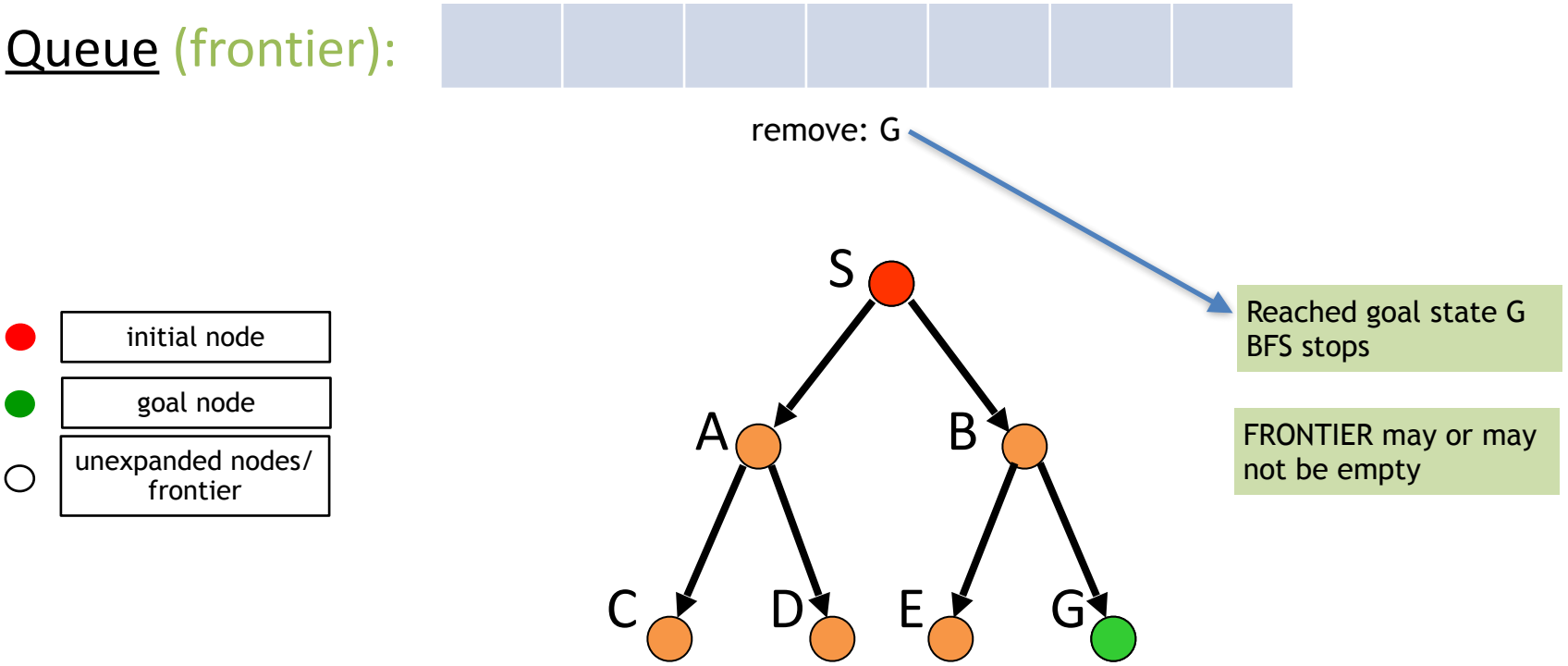
remove: E





Illustrating Breadth-First Search (BFS) Algorithm

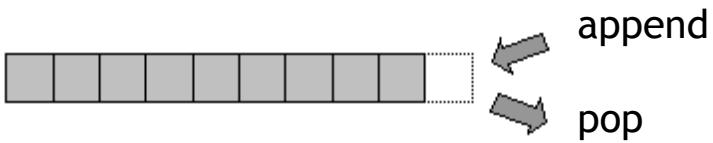
- Nodes are inserted & removed on opposite sides of frontier



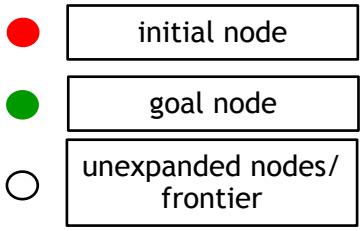
Illustrating Depth-First Search (DFS) Algorithm

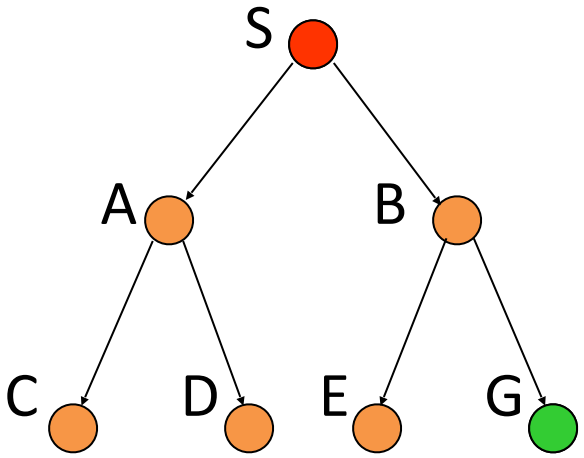
Illustrating Depth-First Search (DFS) Algorithm

- Nodes are inserted & removed on same sides of frontier
  - Should we use Stack or Queue for frontier?



- Let's see the execution of DFS
  - starting at initial node **S** and searching for goal node **G**



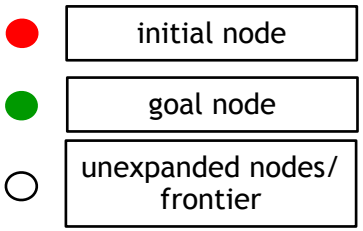


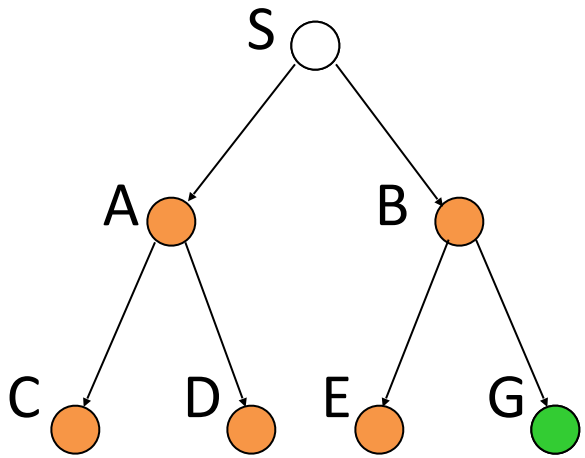
Illustrating Depth-First Search (DFS) Algorithm

- Nodes are inserted & removed on same sides of frontier

Stack (frontier):

append: S





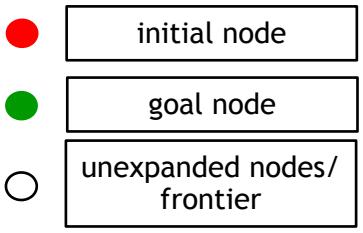
Illustrating Depth-First Search (DFS) Algorithm

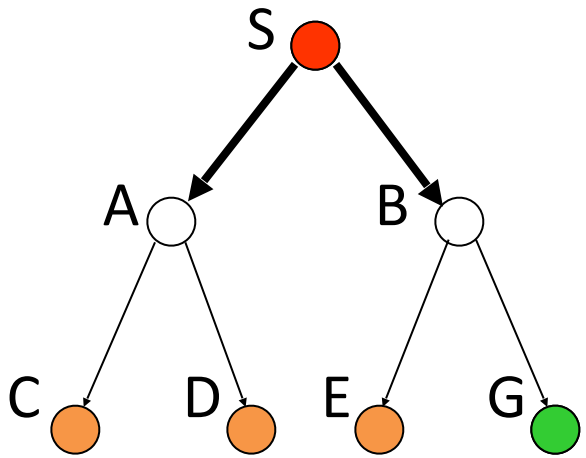
- Nodes are inserted & removed on same sides of frontier

Stack (frontier):

A	B					

remove: S
append: A, B





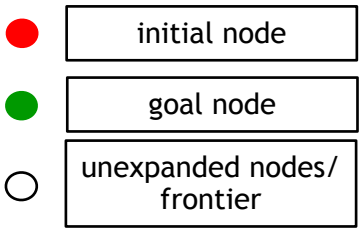
Illustrating Depth-First Search (DFS) Algorithm

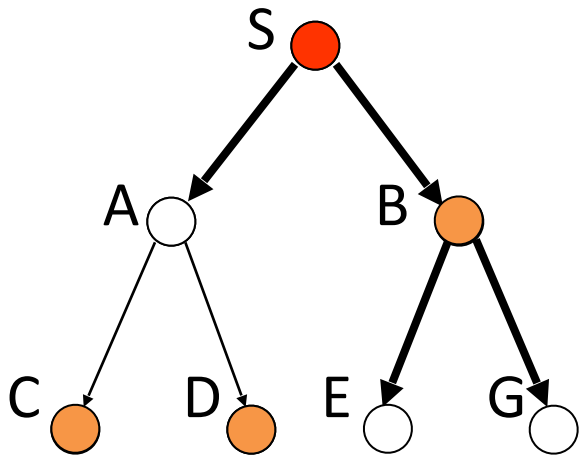
- Nodes are inserted & removed on same sides of frontier

Stack (frontier):

A	E	G				

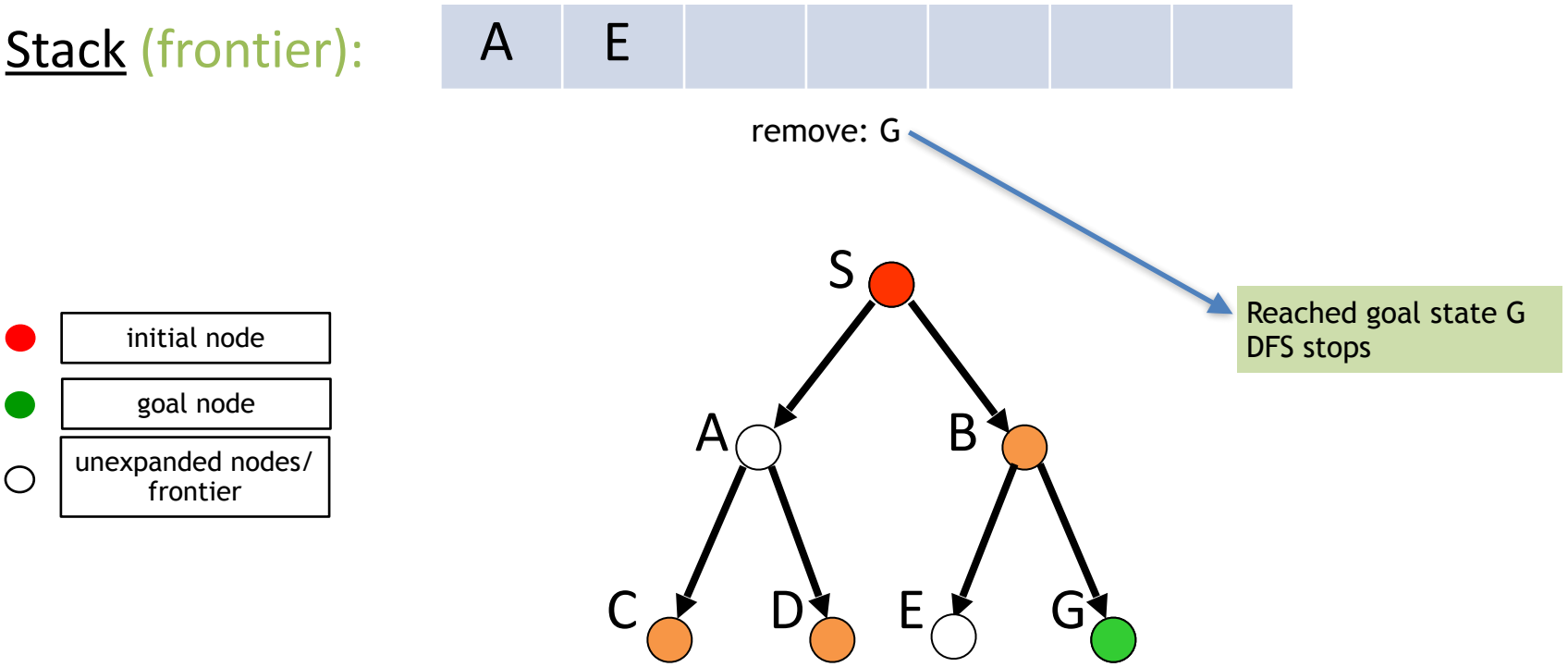
remove: B
append: E, G





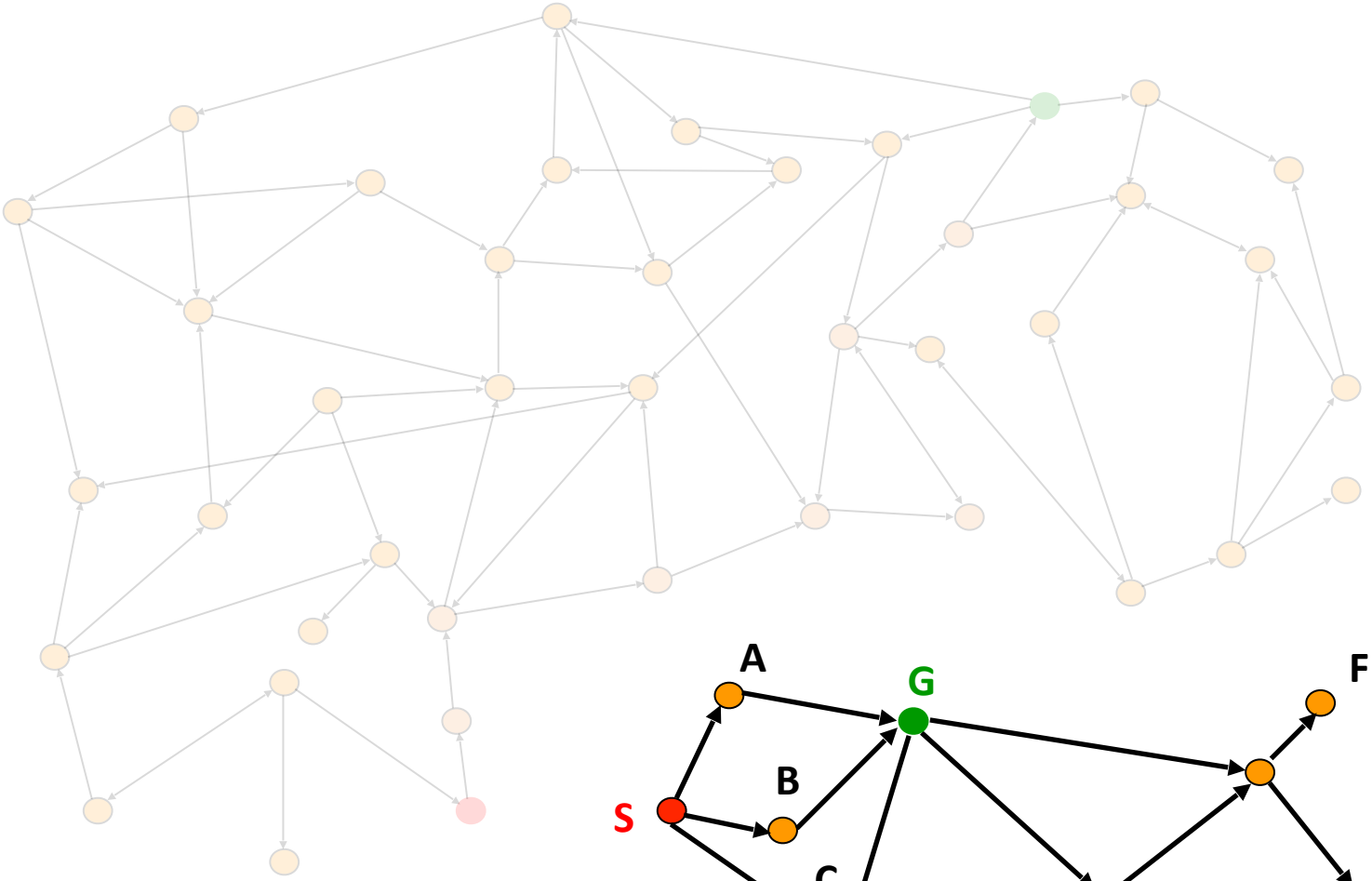
Illustrating Depth-First Search (DFS) Algorithm

- Nodes are inserted & removed on same sides of frontier



Activity: Breadth-First Search Algorithm

Exercise: Breadth-First Search Algorithm on the following Graph



Useful Data Structure (for frontier) in Python: **deque**

Useful Data Structure in Python: deque

- deque is a "double-ended" queue.
  - Reference: https://docs.python.org/2/library/collections.html#collections.deque
- Useful methods in deque

append(*x*)
Add *x* to the right side of the deque.

appendleft(*x*)
Add *x* to the left side of the deque.

pop()
Remove and return an element from the right side of the deque

popleft()
Remove and return an element from the left side of the deque.

Useful Data Structure in Python: deque

- deque data-structure has useful methods for both ‘Stack’ and ‘Queue’

- Initializing deque with some elements

```
from collections import deque

# ------------- make a new deque with three items -----------------

my_deque = deque(['b', 'c', 'd'])
for elem in my_deque:                    # iterate over the deque's elements
    print(elem, end="")
print("")
```

Useful Data Structure in Python: deque

- deque data-structure has useful methods for both ‘Stack’ and ‘Queue’

- Add a new element to deque

```
# ------------- add a new entry to the deque ---------------------

my_deque.append('e')                    # to the right side
my_deque.appendleft('a')                # to the left side
print(my_deque)                         # show the representation of the deque
```

Useful Data Structure in Python: deque

- deque data-structure has useful methods for both ‘Stack’ and ‘Queue’

- Removing an element from deque

```
# ------------- remove an entry from the deque --------------------

elem = my_deque.pop()                          # return and remove the rightmost item
print("popped element ", elem)

elem = my_deque.popleft()                      # return and remove the leftmost item
print("popped lefmost element ", elem)

print("The deque that remains is:")
print(my_deque)
```

Useful Data Structure in Python: deque

- deque data-structure has useful methods for both ‘Stack’ and ‘Queue’

- Test membership of an element in the deque

```
# ------------- test membership in the deque ----------------
elem = 'c'
if elem in my_deque:
    print (elem, " is in the deque")
else:
    print(elem, " is not in the deque")
```