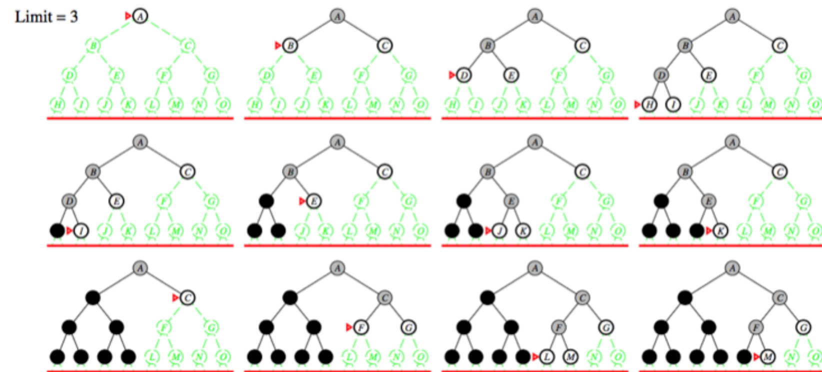
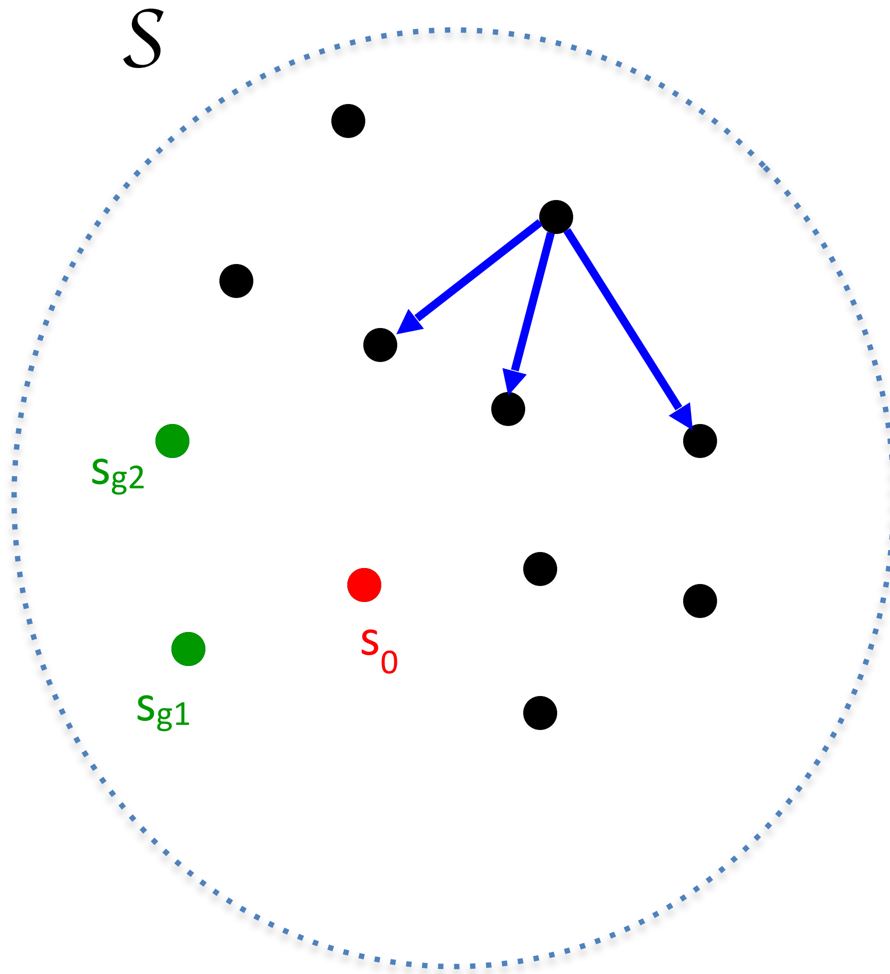
CS143: Artificial Intelligence



Iterative Deepening Search (IDS)
BFS and DFS implementation issues

Drake UNIVERSITY

Recap: 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

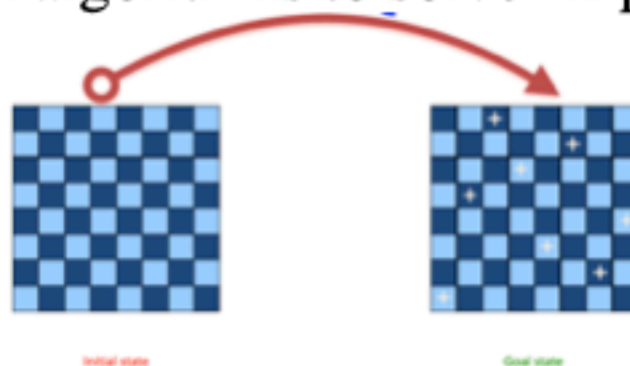
- Set of goal states:

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

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

Recap: Abstraction for Old Puzzle

Search algorithms to solve AI problems



State space:
Initial state:
Goal state:
Successor function:
Cost:

Recap: 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)

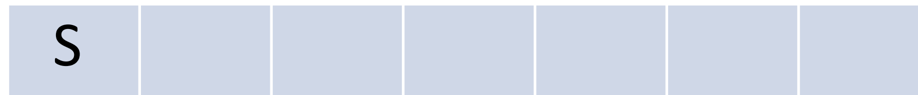
Recap: Search Strategy

- 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

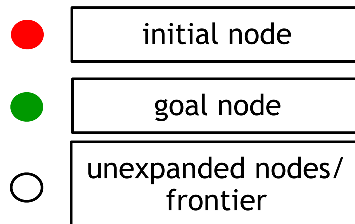
Illustrating Breadth-First Search (BFS) Algorithm

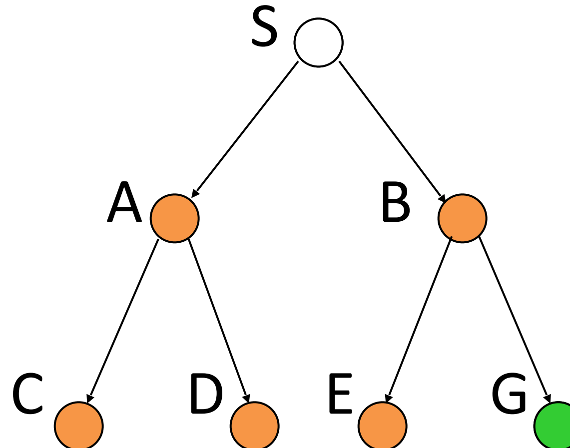
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



append: S

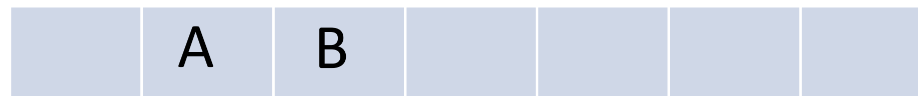




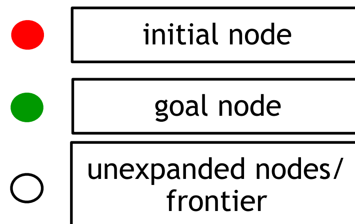
Illustrating Breadth-First Search (BFS) Algorithm

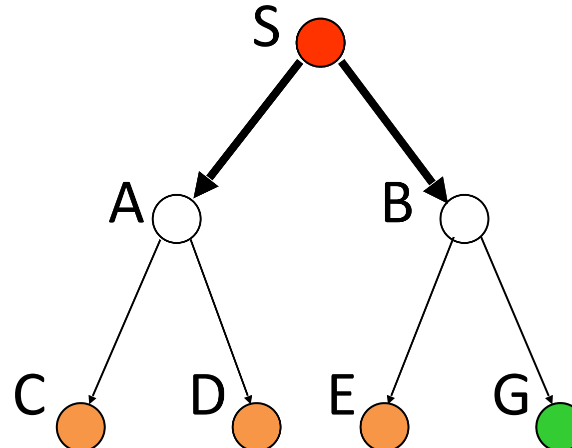
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: S
append: A, B

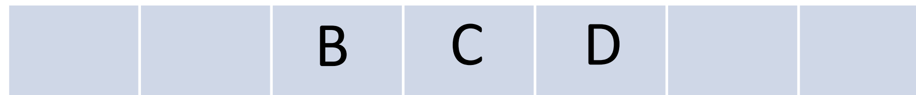




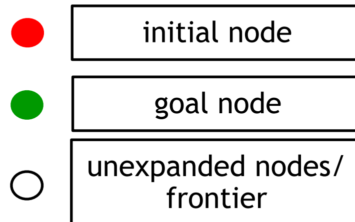
Illustrating Breadth-First Search (BFS) Algorithm

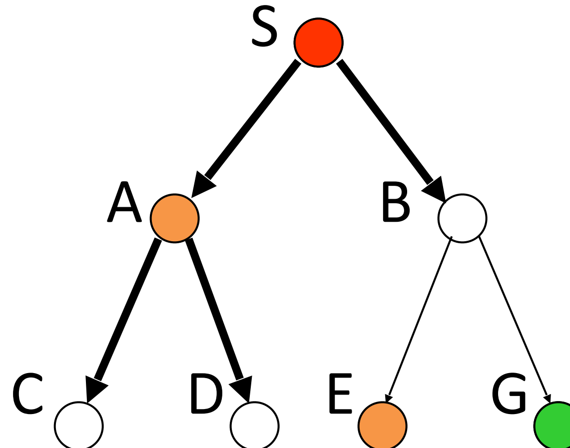
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: A
append: C, D

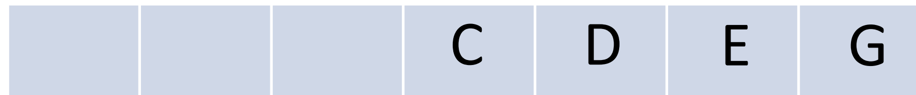




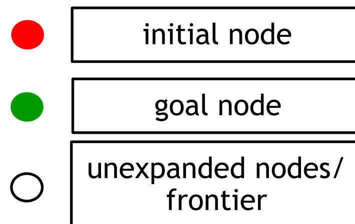
Illustrating Breadth-First Search (BFS) Algorithm

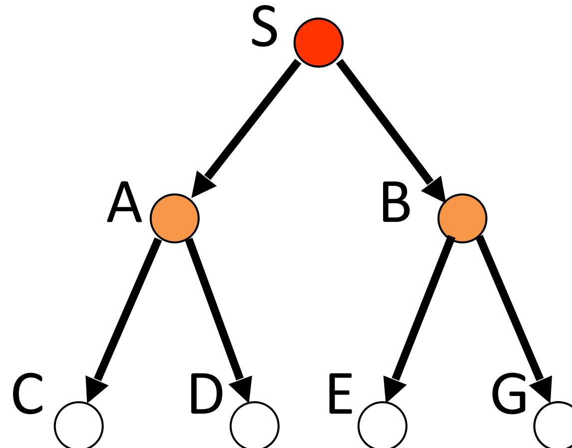
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: B
append: E, G

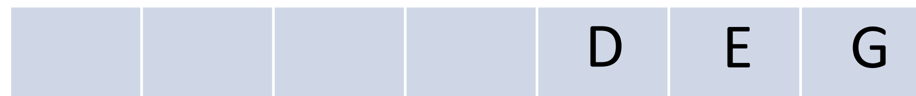




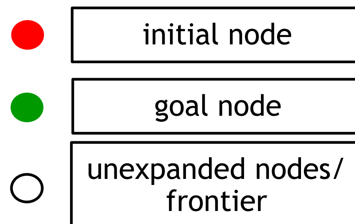
Illustrating Breadth-First Search (BFS) Algorithm

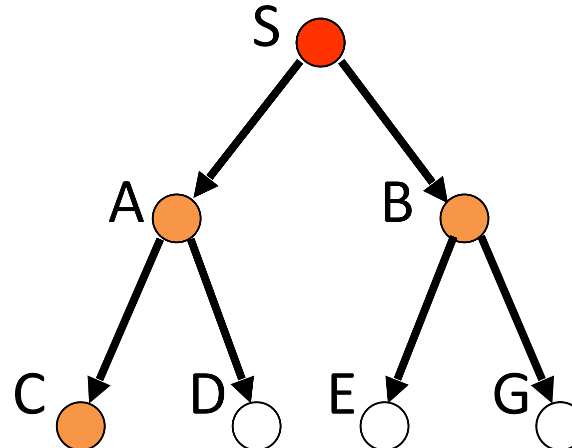
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: C

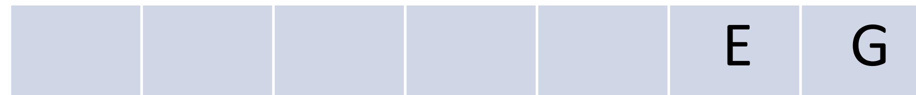




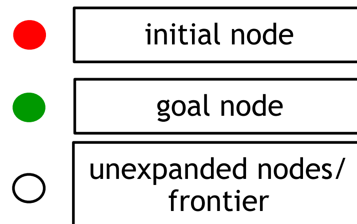
Illustrating Breadth-First Search (BFS) Algorithm

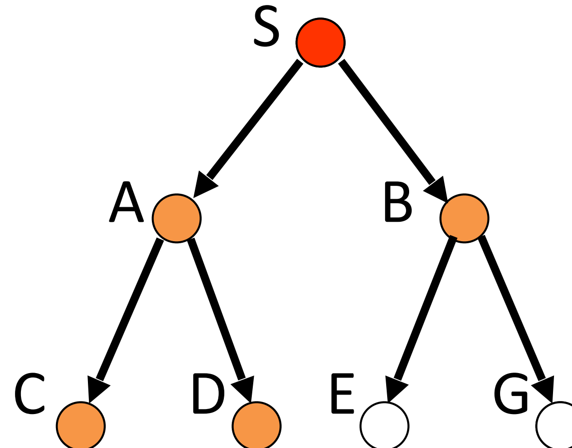
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: D

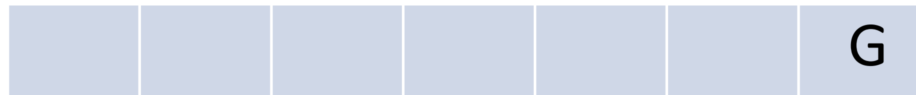




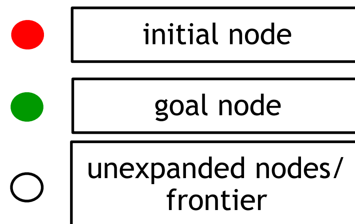
Illustrating Breadth-First Search (BFS) Algorithm

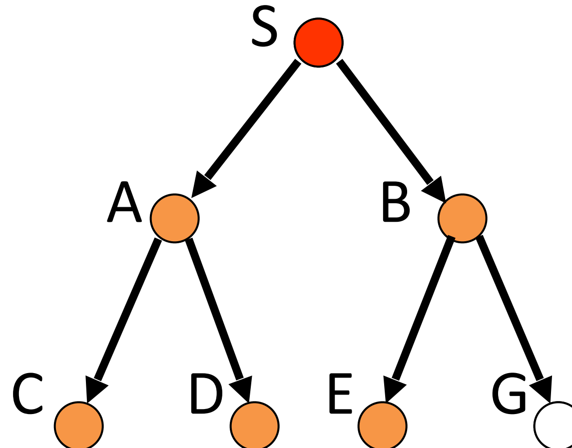
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: E

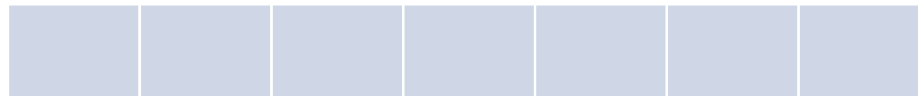




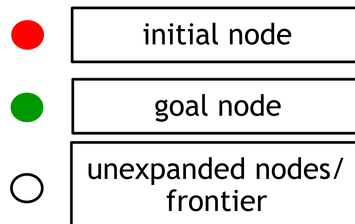
Illustrating Breadth-First Search (BFS) Algorithm

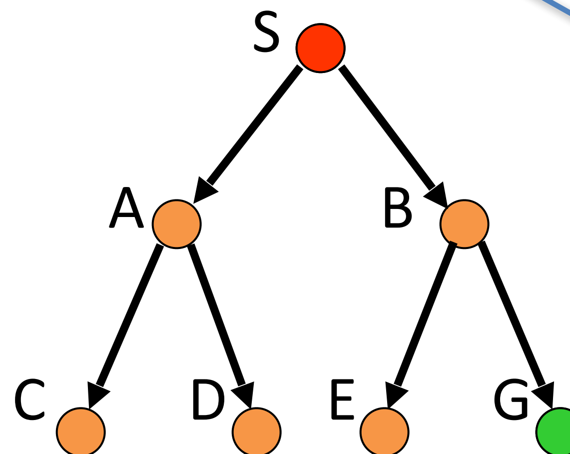
- Nodes are inserted & removed on opposite sides of frontier

Queue (frontier):



remove: G





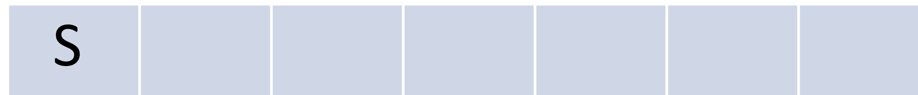
Reached goal state G
BFS stops

FRONTIER may or may not be empty

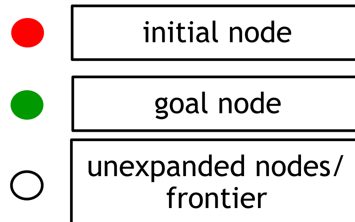
Recap: Illustrating Depth-First Search (DFS)

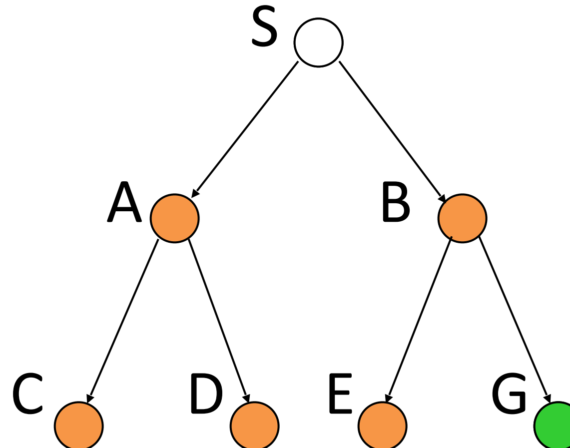
- Nodes are inserted & removed on same sides of frontier

Stack (frontier):



append: S



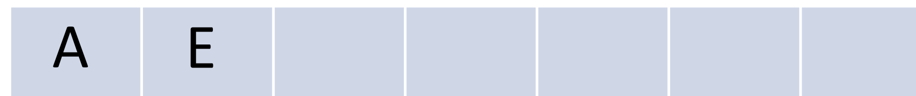


Thin black edges meant those have not been visited during search

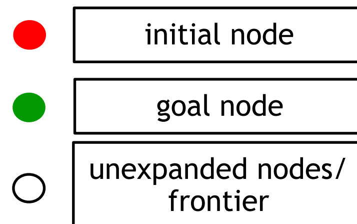
Recap: Illustrating Depth-First Search (DFS)

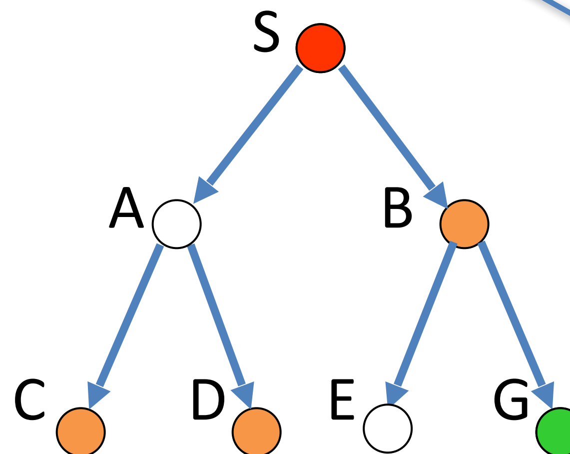
- Nodes are inserted & removed on same sides of frontier

Stack (frontier):



remove: G





Reached goal state G
DFS stops

FRONTIER may or may not be empty. In this case, it's not empty (A and E)

Blue edges meant those have been visited during search

Recap: 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

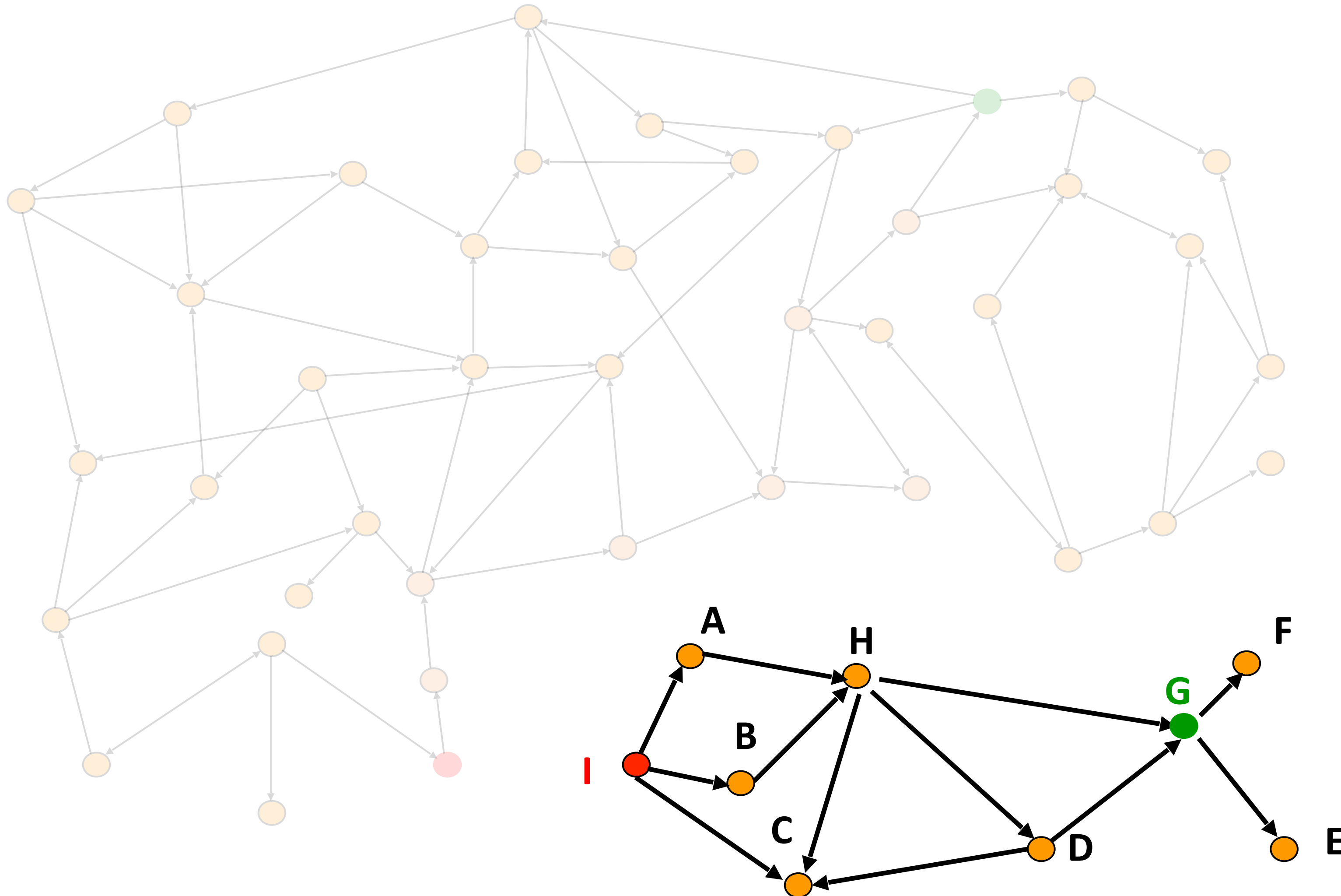
append(*x*)
Add *x* to the right side of the deque.

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.

Exercise: Breadth-First Search on the following Graph



Recap: Useful Data Structure in Python: deque

Group exercise#1:

Implement the BFS traversal on this `frontier` datastructure.

```
frontier = deque([])
my_list = ['a', 'b', 'c', 'd', 'e']
for i in range(len(my_list)):
  item = my_list[i]
  frontier.append(item)
  print(my_list[0:i+1])

# your code ...
print('BFS traversal ...')
while len(frontier) > 0:
  print("item:", frontier.popleft())
```

```
['a']
['a', 'b']
['a', 'b', 'c']
['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd', 'e']
BFS traversal ...
item: a
item: b
item: c
item: d
item: e
```

Group exercise#2:

Implement the DFS traversal on this `frontier` datastructure.

```
frontier = deque([])
my_list = ['a', 'b', 'c', 'd', 'e']
for i in range(len(my_list)):
  item = my_list[i]
  frontier.append(item)
  print(my_list[0:i+1])

# your code ...
print('DFS traversal ...')
while len(frontier) > 0:
  print("item:", frontier.pop())
```

```
['a']
['a', 'b']
['a', 'b', 'c']
['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd', 'e']
DFS traversal ...
item: e
item: d
item: c
item: b
item: a
```

Reza's solution

https://colab.research.google.com/drive/1qNltbJYWKwDfJKHM0kSGR7Z-GgKOwY4y?usp=sharing

Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) Algorithm

- Uses a limited version of DFS called depth-limited search
  - nodes at depth l has no successors

- do a depth-limited search with l = 0, if it fails,
- do a depth-limited search with l = 1, if it fails,
- do a depth-limited search with l = 2, if it fails,

  …

  …

  …

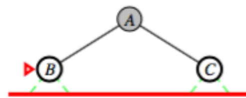
- until you find a goal

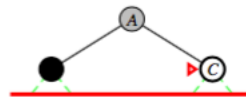
Iterative Deepening Search (IDS) Algorithm

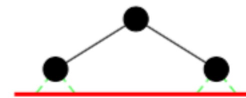
Limit = 0

Iterative Deepening Search (IDS) Algorithm

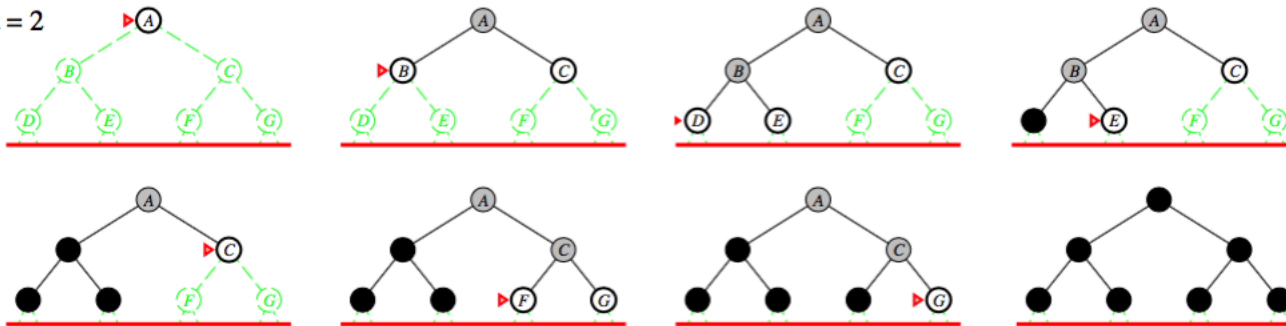
Limit = 1



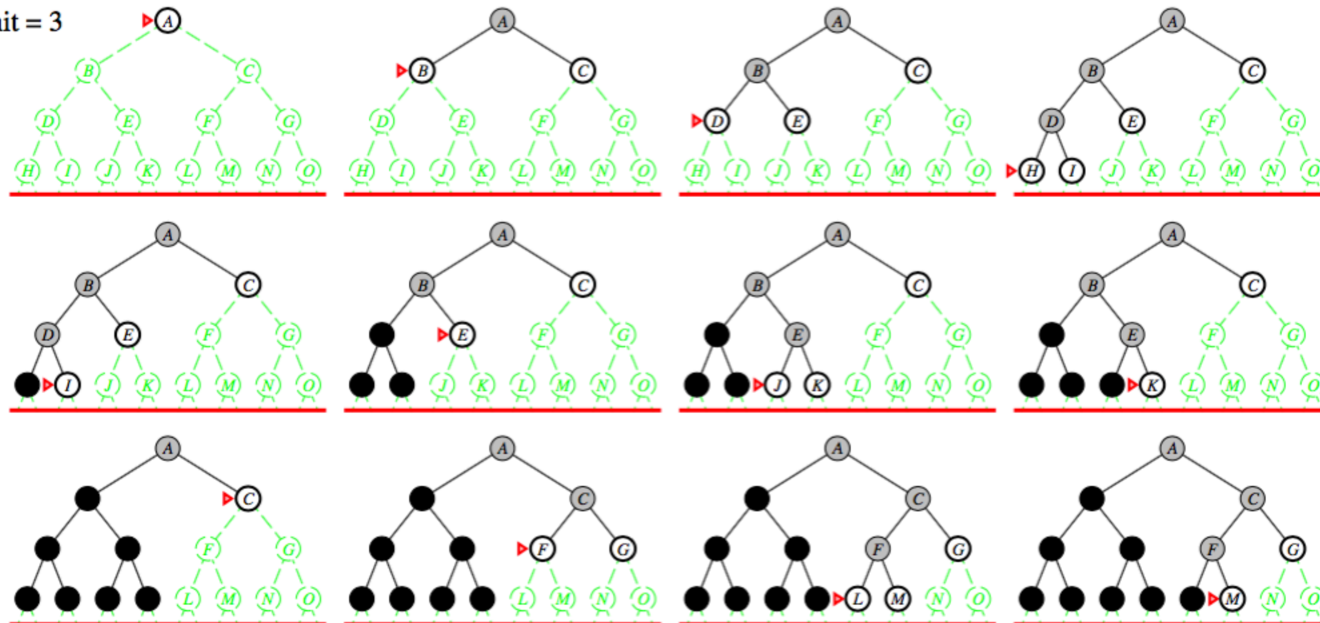




Iterative Deepening Search (IDS) Algorithm



Iterative Deepening Search (IDS) Algorithm



Today: Practical Implementation

- Practical implementation of AI problem solving as search
  - Introduce the python code for search
  - Explore the code in-class