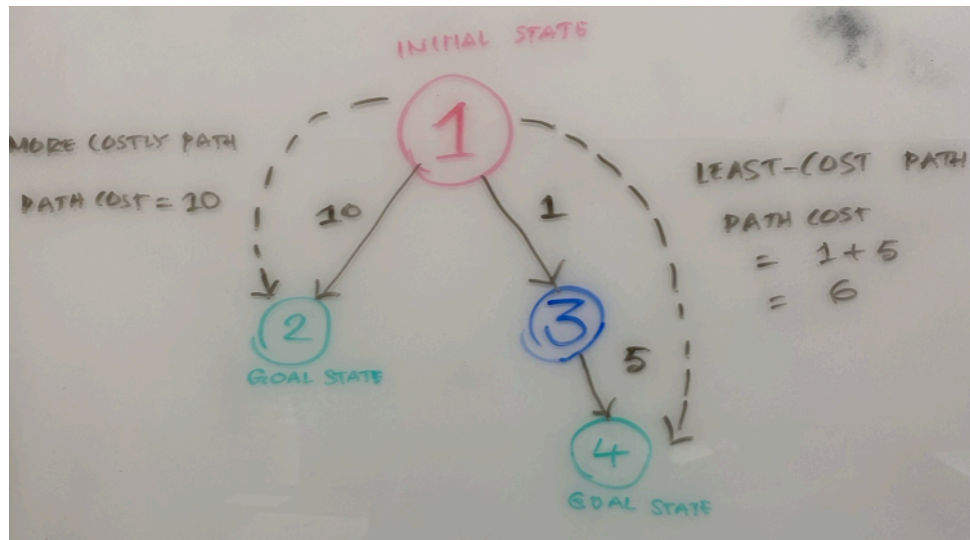
CS143: Artificial Intelligence



Analysis of Search Algorithms (DFS, IDS)
Uniform Cost Search (UCS)

Review: Analysis of BFS

Properties of Breadth-First Search (BFS)

- **Completeness:**
  - a search algorithm is **complete** if it finds a solution whenever one exists
- **Optimality:**
  - a search algorithm is **optimal** if it returns a minimum-cost path if a solution exists
- **Time complexity:**
  - time required by the algorithm
- **Space complexity:**
  - memory required by the algorithm

Properties of Breadth-First Search (BFS)

- Is BFS Complete?
- Is BFS Optimal?
- What is the running time complexity of BFS?
- What are the memory requirement of BFS?

Properties of Breadth-First Search (BFS)

b: branching factor
d: depth of shallowest goal node

- Is BFS complete?
  - BFS is complete for a finite branching factor b
- Is BFS optimal?
  - BFS is optimal if step cost is 1
  - BFS will find the goal node at the shallowest depth before reaching goal nodes at a deeper level

Properties of Breadth-First Search (BFS)

b: branching factor
d: depth of shallowest goal node

- Time complexity of BFS?

  Number of nodes generated (in the worst-case):

  $1 + b + b^2 + \ldots + b^d + b*(b^d - 1) = O(b^{d+1})$

  Time complexity is $O(b^{d+1})$

- Let's check — how did we arrive at above formula.

Properties of Breadth-First Search (BFS)

b: branching factor
d: depth of shallowest goal node

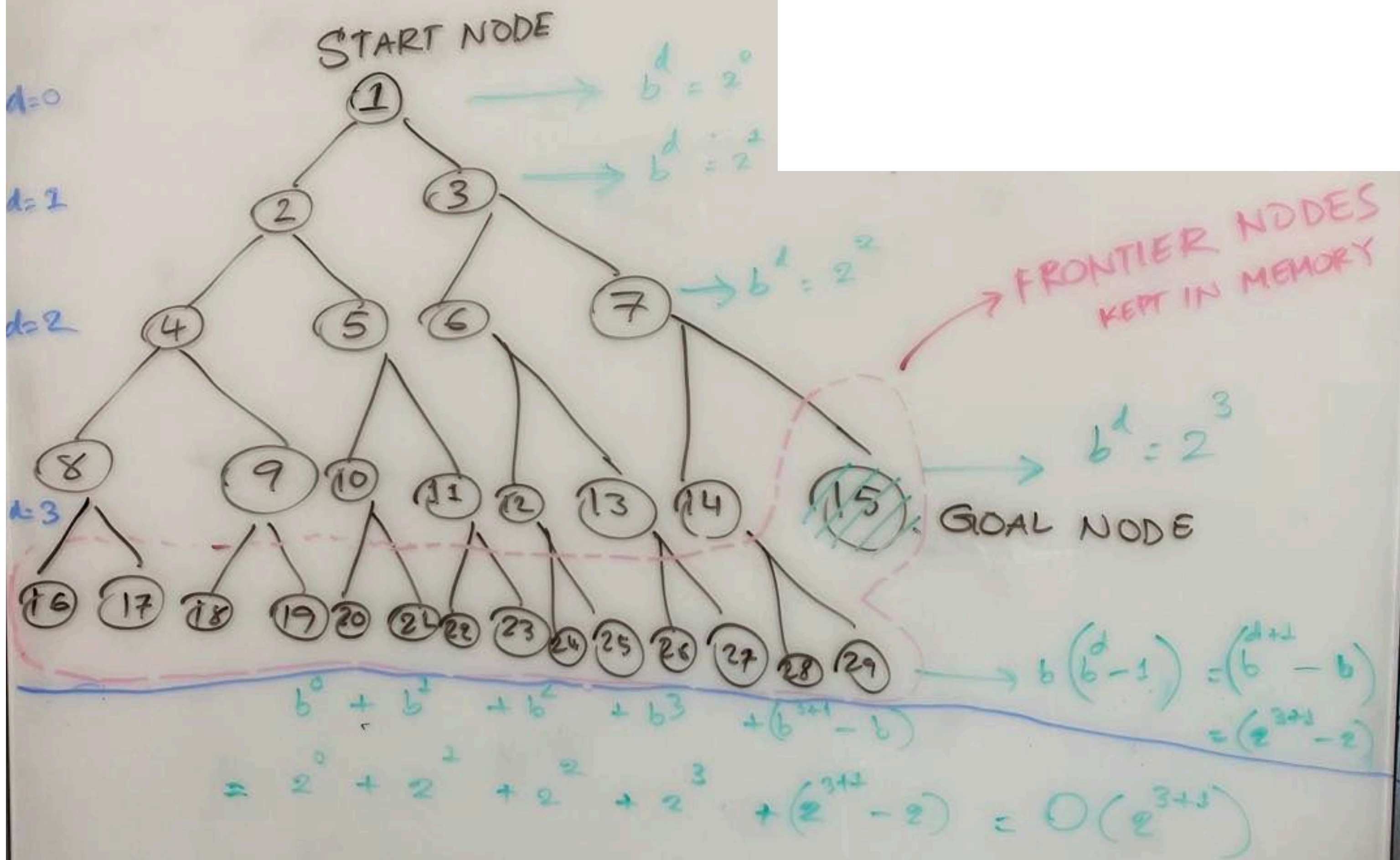
- Memory complexity of BFS?

Number of nodes kept in the frontier

$b*(b^d - 1) = O(b^{d+1})$

Time and space complexity is $O(b^{d+1})$

- Memory is a big concern for BFS.
  - can easily generate nodes at 100MB/second so gigabytes of space required if BFS runs for hours


b = 2 : BRANCHING FACTOR
d = 3 : GOAL NODE AT THE SHALLOWEST DEPTH
TIME COMPLEXITY
START NODE
d=0
d=1
d=2
d=3
FRONTIER NODES
KEPT IN MEMORY
GOAL NODE

Properties of Breadth-First Search (BFS)

- Time is a big concern for BFS
  - If the problem has a solution at depth 16 with branching factor 10, it may take about 350 years for BFS to find it

d	# Nodes	Time	Memory
2	111	0.11 msec	11 Kbytes
4	11,111	11.11 msec	1 Mbyte
6	$\sim 10^6$	1 sec	100 Mb
8	$\sim 10^8$	100 sec	10 Gbytes
10	$\sim 10^{10}$	2.8 hours	1 Tbyte
12	$\sim 10^{12}$	11.6 days	100 Tbytes
14	$\sim 10^{14}$	3.2 years	10,000 Tbytes

Assumptions:

- b = 10
- A computer can process 1,000,000 nodes/sec

New Content: Analysis of DFS

Properties of Depth-First Search

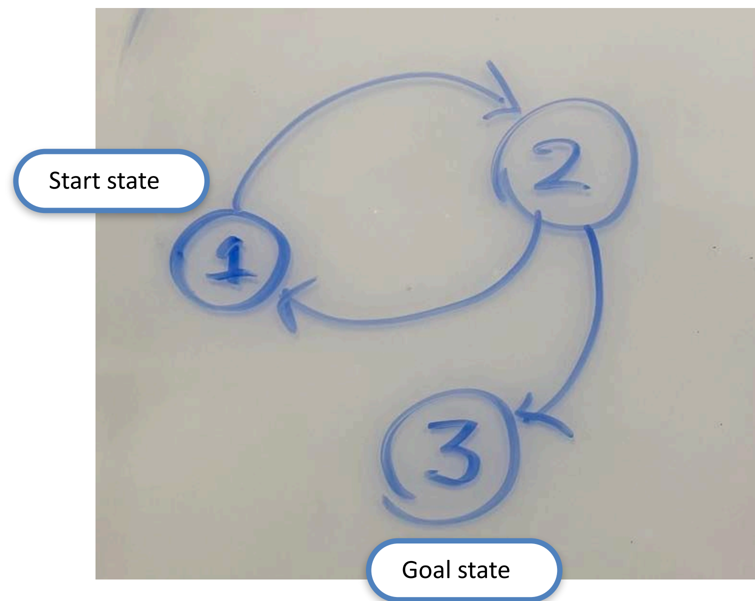
- **Completeness:**
  - a search algorithm is **complete** if it finds a solution whenever one exists
- **Optimality:**
  - a search algorithm is **optimal** if it returns a minimum-cost path if a solution exists
- **Time complexity:**
  - time required by the algorithm
- **Space complexity:**
  - memory required by the algorithm

Properties of Depth-First Search

- Is DFS Complete?
- Is DFS Optimal?
- What is the running time complexity of DFS?
- What are the memory requirement of DFS?

Properties of Depth-First Search

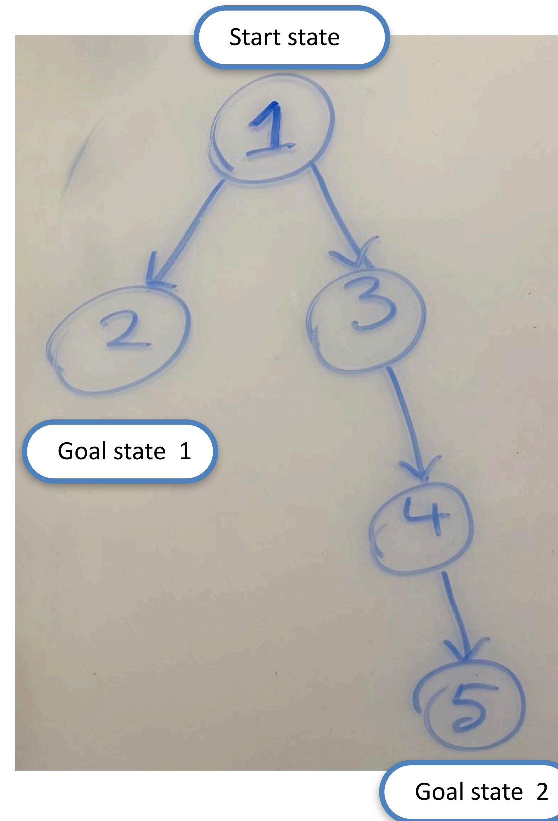
- Is DFS Complete?



- It is not complete even if the state graph is finite
  - The search tree could be infinite
  - Eg, [1]->[2]->[3, 1]->[3, 2]->[3, 3, 1]->[3, 3, 2]-> …

Properties of Depth-First Search

- Is DFS Optimal?



- It is not optimal
  - The search tree could not reach goal state at a lower depth
  - Eg, [1]->[2,3]->[2, 4]->[2, 5]

Properties of Depth-First Search

b: branching factor
l: depth of deepest leaf node

- Depth-first search
  - Not complete
  - Not optimal

- Number of nodes generated:
  $1 + b + b^2 + \ldots + b^l \quad = \; O(b^l)$

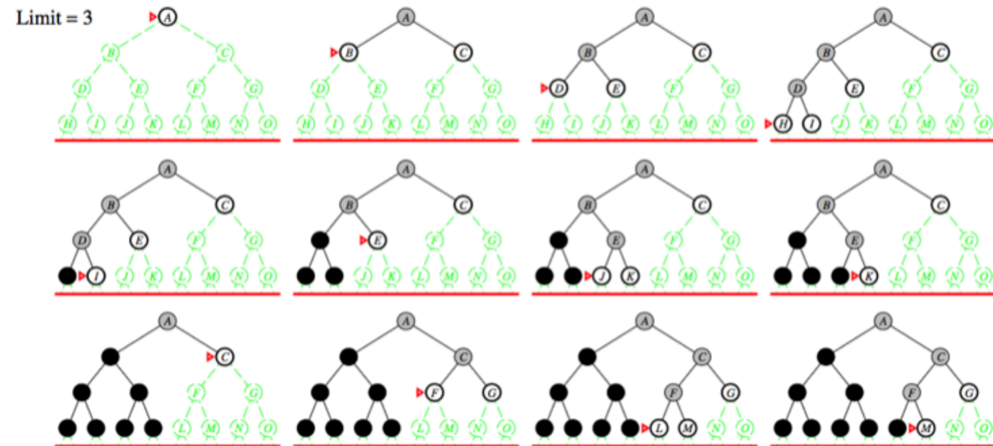
- Time and space complexity is O(b*l)

Analysis of IDS

Properties of Iterative Deepening Search (IDS)

- Is IDS Complete?
- What is the running time complexity of IDS?
- What are the memory requirement of IDS?
- Is IDS Optimal?

Properties of Iterative Deepening Search (IDS)



- IDS is complete
- IDS is optimal
- Number of nodes generated:

$$(d+1)b^0+db^1+(d-1)b^2+\ldots+b^d = O(b^d)$$

- Space complexity is $O(b*d)$

Number of Generated Nodes: BFS vs IDS

b: branching factor
d: depth of shallowest goal node

- b = 2 and d = 5

	BFS	IDS
Nodes at depth 1	1	1 x 6 = 6
Nodes at depth 2	2	2 x 5 = 10
Nodes at depth 3	4	4 x 4 = 16
Nodes at depth 4	8	8 x 3 = 24
Nodes at depth 5	16	16 x 2 = 32
Nodes at depth 5	32	32 x 1 = 32
	63	120

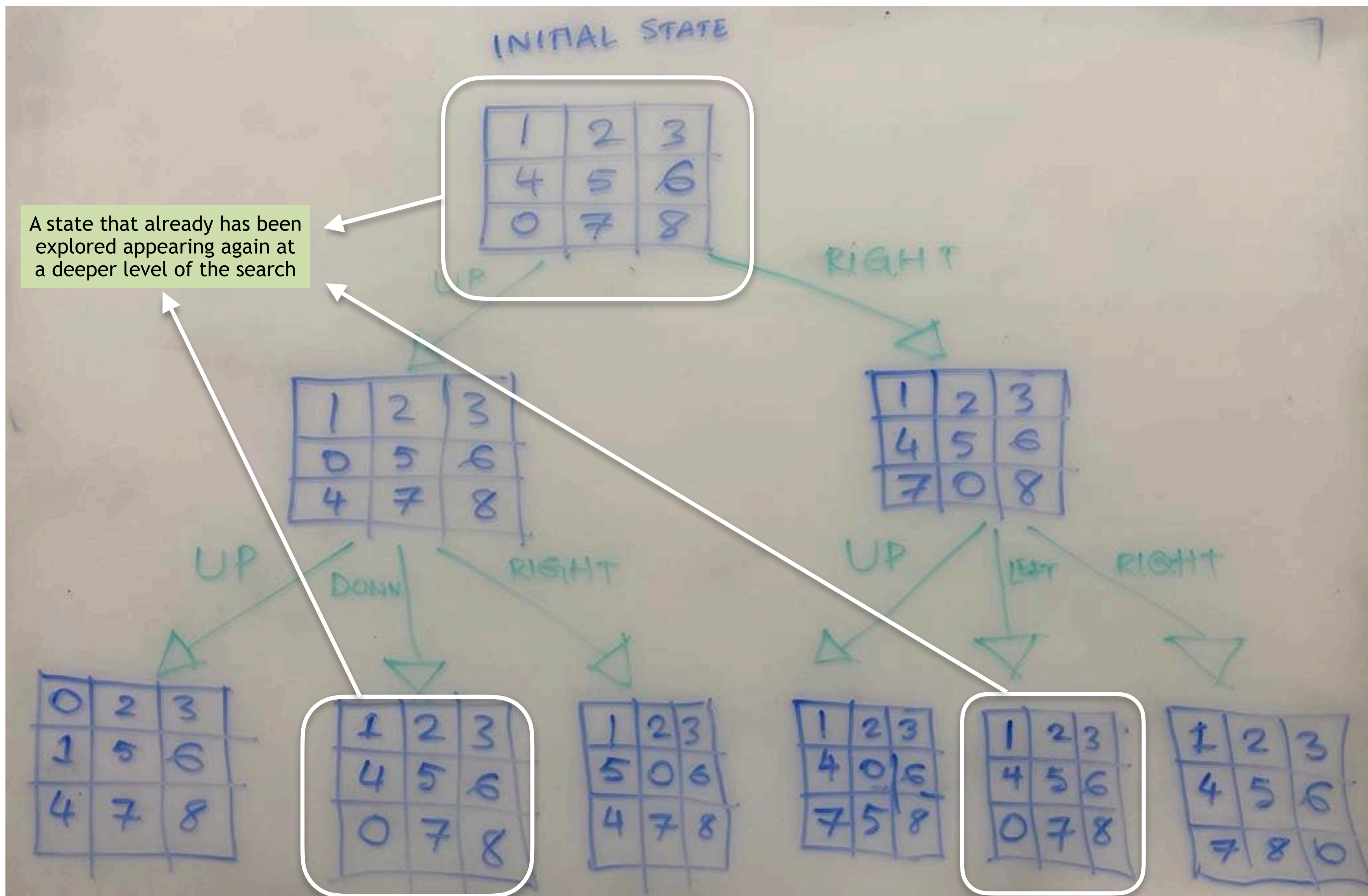
Number of Generated Nodes: BFS vs IDS

b: branching factor
d: depth of shallowest goal node

- b = 10 and d = 5

BFS	IDS
1	6
10	50
100	400
1,000	3,000
10,000	20,000
100,000	100,000
111,111	123,456

Revisit 8-puzzle: Breadth-First Search



How can you stop revisiting states that have been explored?

Avoiding Revisited States

- Requires comparing state descriptions

- Introduce a data-structure **EXPLORED**
  - Recall that we store unexpanded nodes in the **EXPLORED**

- For Breadth-First Search (BFS):
  - Store all states associated with generated nodes in **EXPLORED**
  - If a new node you found is in **EXPLORED**, then discard the node

Python Set

- Python Set: "unordered collections of unique elements"
  - https://docs.python.org/2/library/sets.html

- Useful methods:
  - Given set `s` and an element `x`
    - Test for membership: `if x in s:`
    - Add an element: `s.add(x)`

Python Set: example code

```
# ------------- make a new set with three items ----------------
print("Printing the current elements in the set: ", end="")
my_set = set(['b', 'c', 'd'])          # make a new set with three items
for elem in my_set:                     # iterate over the set elements
    print(elem, end="")                 # note that a set doesn't maintain an order
print("")
```

```
>>> %Run set_datastructure_example.py

  Printing the current elements in the set: dbc
```

Python Set: example code

```
# ------------- make a new set with three items ----------------
print("Printing the current elements in the set: ", end="")
my_set = set(['b', 'c', 'd'])         # make a new set with three items
for elem in my_set:                   # iterate over the set elements
    print(elem, end="")               # note that a set doesn't maintain an order
print("")

# ------------- add a new entry to the set ---------------------
print("After adding 'e' and 'a' current elements in the set: ", end="")
my_set.add('e')                       # add a new entry
my_set.add('a')                       # add a new entry
print(my_set)                         # show the representation of the set
                                      # again note that a set doesn't maintain an
```

```
>>> %Run set_datastructure_example.py

  Printing the current elements in the set: dbc
  After adding 'e' and 'a' current elements in the set: {'b', 'a', 'd', 'c', 'e'}
```

Python Set: example code

```
# ------------- make a new set with three items -----------------
print("Printing the current elements in the set: ", end="")
my_set = set(['b', 'c', 'd'])        # make a new set with three items
for elem in my_set:                   # iterate over the set elements
    print(elem, end="")               # note that a set doesn't maintain an order
print("")

# ------------- add a new entry to the set ----------------------
print("After adding 'e' and 'a' current elements in the set: ", end="")
my_set.add('e')                      # add a new entry
my_set.add('a')                      # add a new entry
print(my_set)                        # show the representation of the set
                                     # again note that a set doesn't maintain an

# ------------- remove an entry from the set --------------------
elem = my_set.pop()                   # return and remove an arbitrary element
print("Popped element (randomly) from the set: ", elem)

print("The set that remains is: ", end="")
print(my_set)
```

```
>>> %Run set_datastructure_example.py

  Printing the current elements in the set: dbc
  After adding 'e' and 'a' current elements in the set: {'b', 'a', 'd', 'c', 'e'}
  Popped element (randomly) from the set:  b
  The set that remains is: {'a', 'd', 'c', 'e'}
```

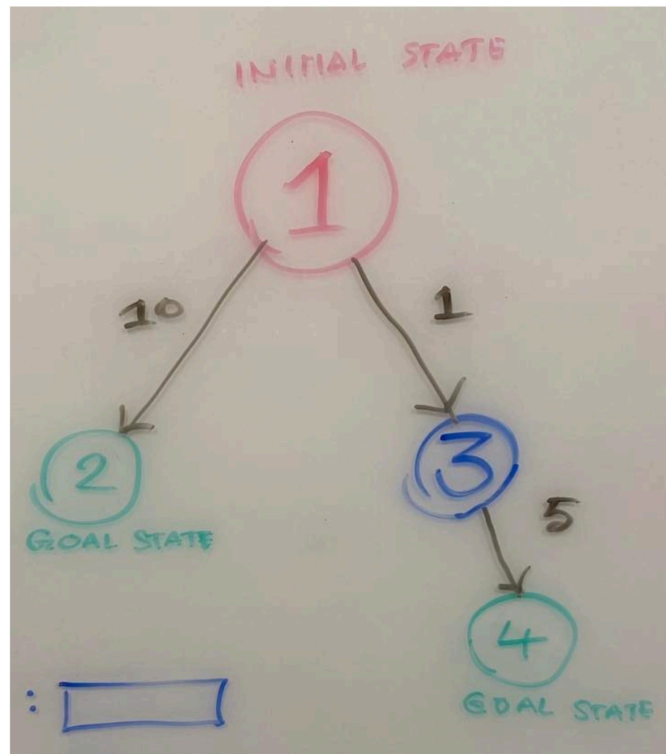
Python Set: example code

```
# ------------- test membership in the set ---------------
print("Testing membership in the set: ", end="")
elem = 'c'
if elem in my_set:
    print (elem, " is in the set")
else:
    print(elem, " is not in the set")
```

```
Testing membership in the set: c  is in the set
```

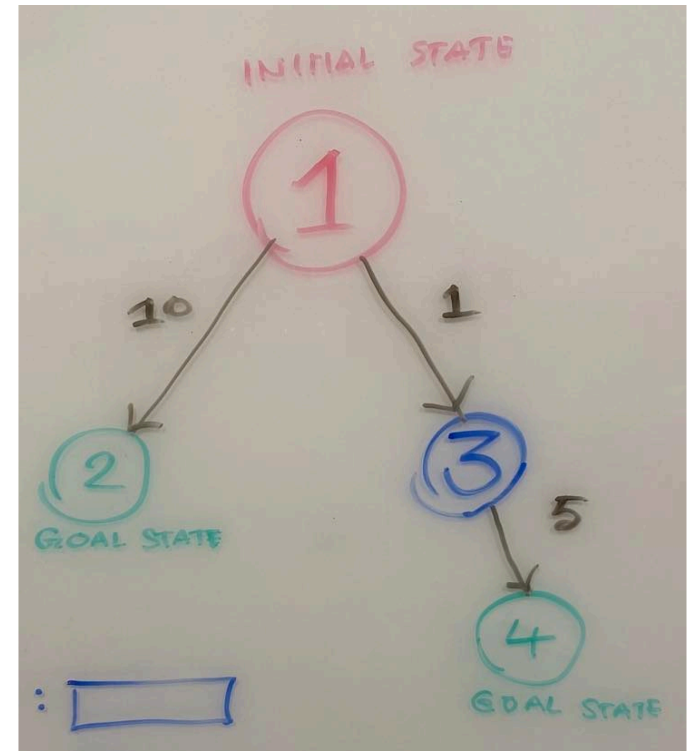
Non-uniform cost function

- Lets add a cost to each edge in the state graph

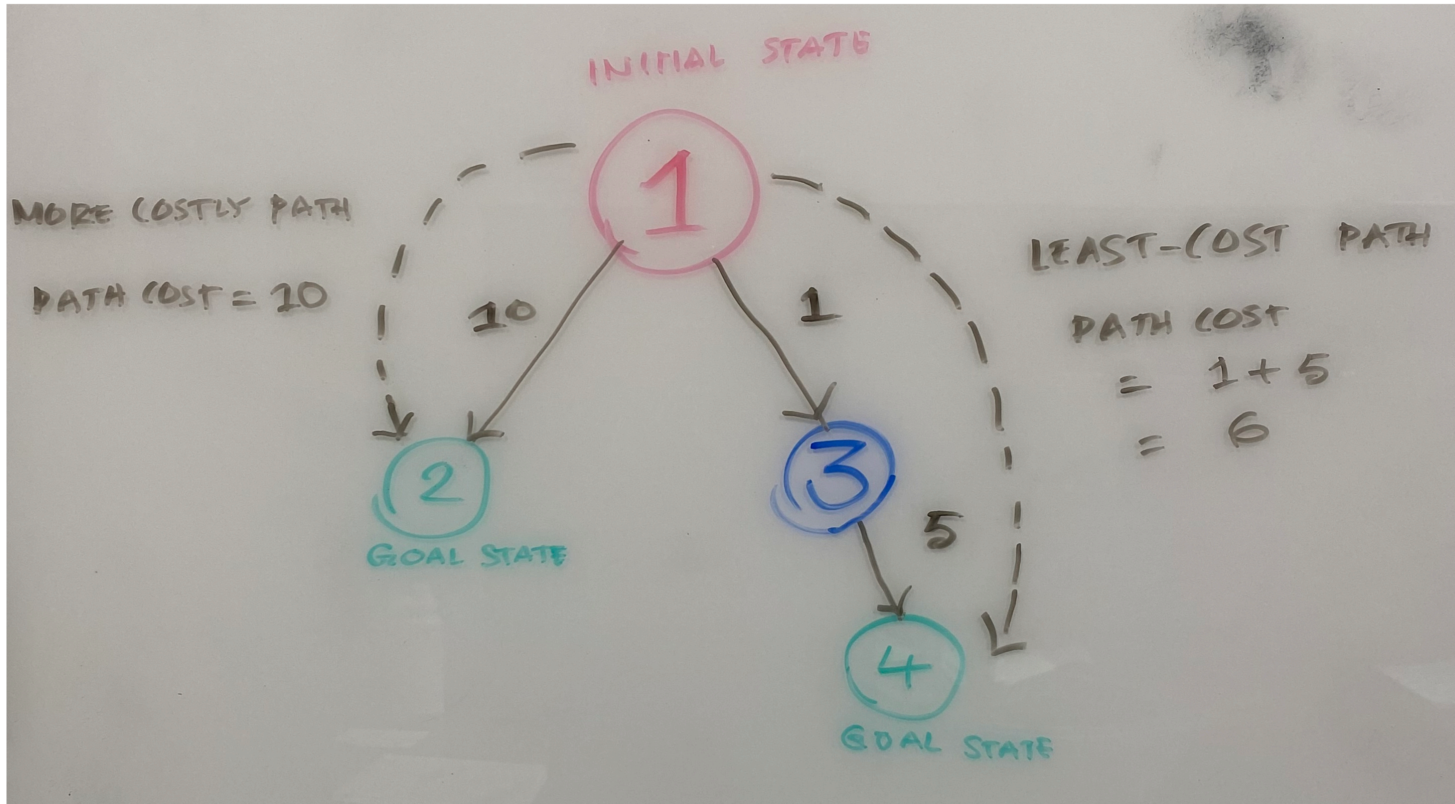


Non-uniform cost function

- What is the optimal cost path in this state graph?
- What will BFS do in this state graph?
- Can BFS reach a goal node with an <u>optimal cost</u>?

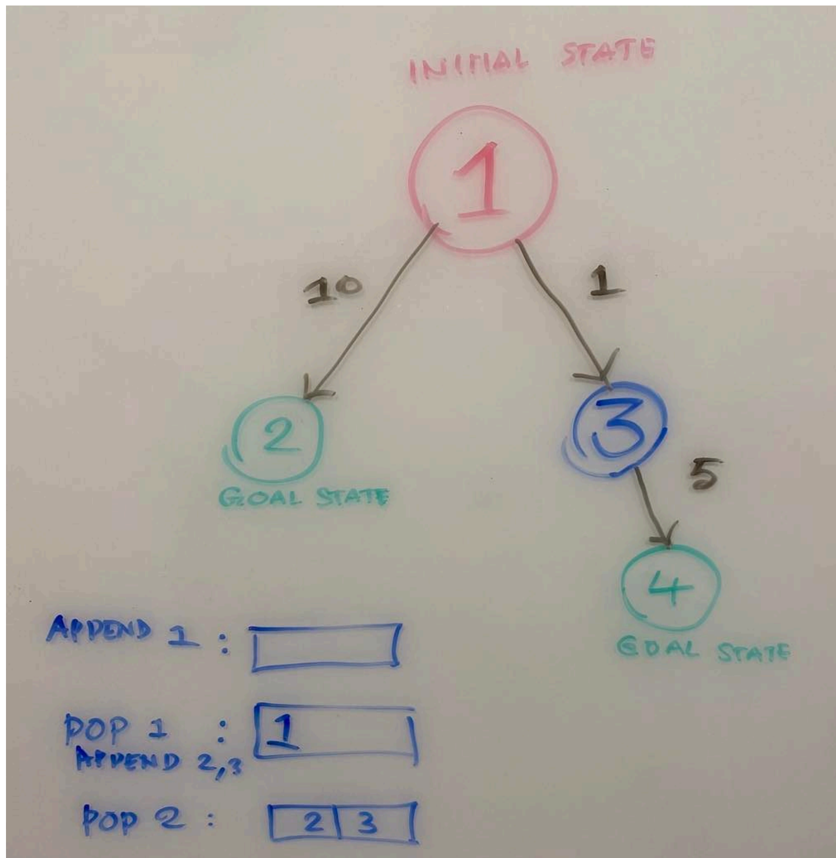


Non-uniform cost function

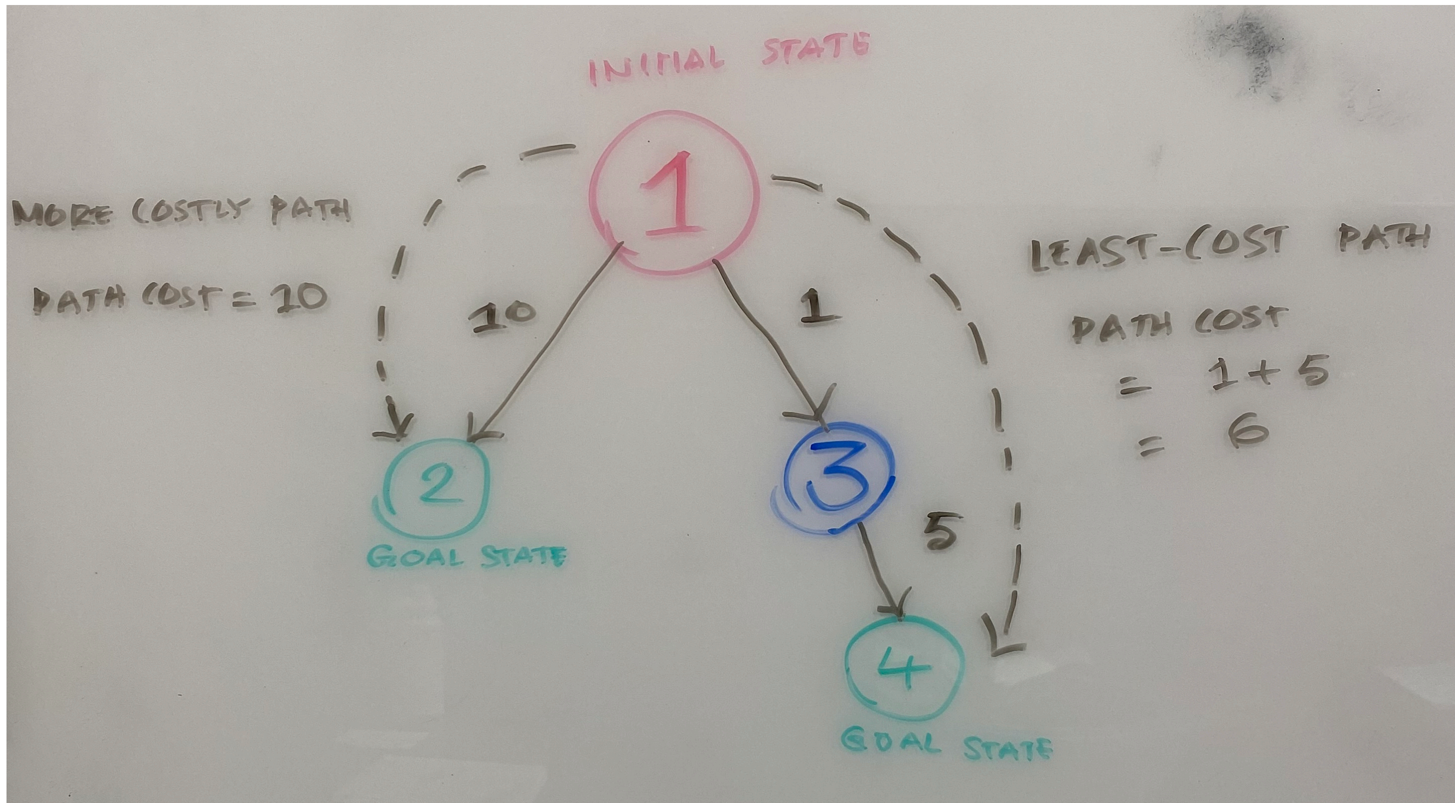


Non-uniform cost function

- BFS is no longer optimal
  - when BFS reaches node 2; stops the search



Non-uniform cost function



- BFS will find the more costly path hence not optimal

Non-uniform cost function

- Since BFS is no longer optimal, so what can we do:
  - instead of expanding the node in FRINGE with the shortest path from the initial state, we want to the one with the **least costly** path

Uniform Cost Search

- If we keep track of a "cost" for expanding a node, then we desire an optimal solution with a minimal total path cost

- The node from the frontier that is selected to be expanded is the node that has the minimum total path cost
    - thus the frontier is implemented as a *priority queue*

Uniform Cost Search

- pseudo-code from Russell & Norvig textbook

use Priority Queue
instead of deque

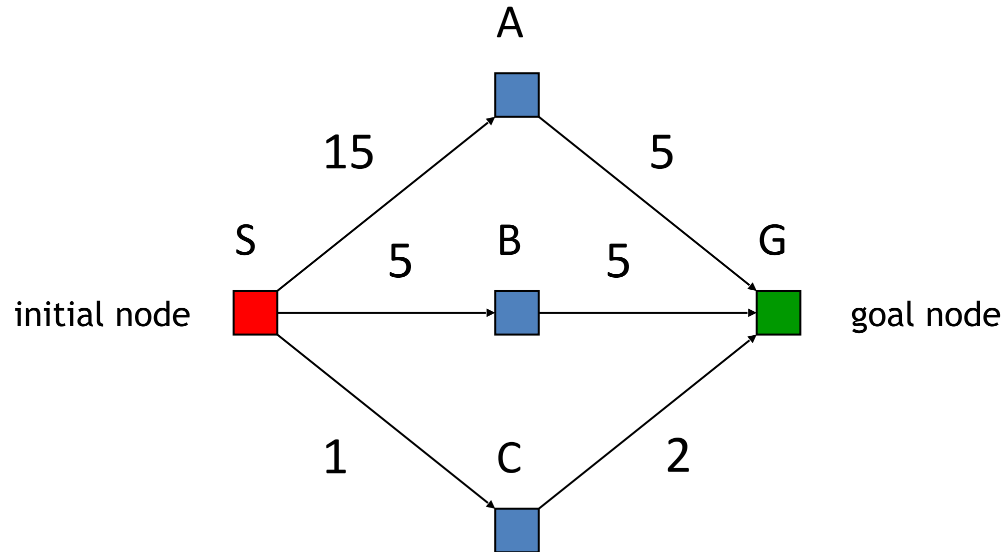
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     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
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Uniform Cost Search

- Uniform Cost Search is essentially graph search algorithm
  - It terminates if goal state is *removed* from frontier (priority queue)
  - Checks if a shorter path to a current node on the frontier is possible

Priority Queue to the rescue

- Can "Uniform Cost Search" (with FRONTIER implemented using Priority Queue) reach a goal node with an optimal cost?



Priority Queue to the rescue

- Search tree with node expansions are as follows:
  - "Uniform Cost Search" (with FRONTIER implemented using Priority Queue) reaches a goal node with an optimal cost

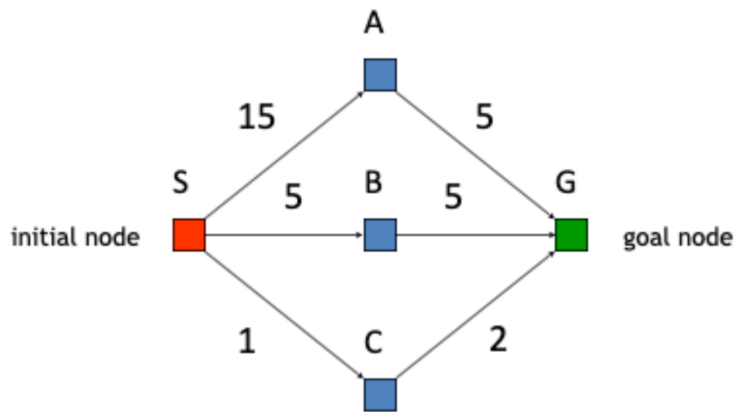


Figure: State Graph

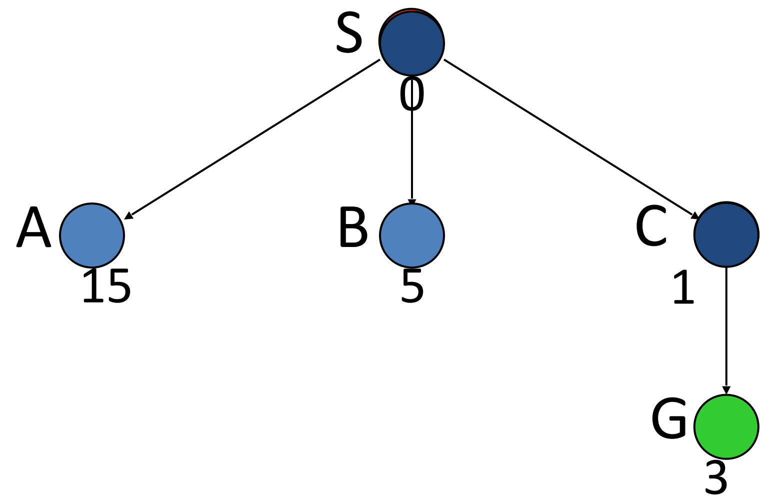


Figure: Order of node expansions

Categorizing Search Strategies

- Search algorithms we have seen so far
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Depth-Limited Search (DLS)
  - Iterative Deepening Search (IDS)
  - Uniform Cost Search (UCS)