Lecture 3.

Uninformed Search

3.1. Uninformed Search Strategies

Uninformed or blind search performs **without** any additional information beyond the problem definition. In this course, we study 5 basic uninformed search strategies:

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search

3.1.1. Measuring Performance of Search Strategies

We use 4 criteria to evaluate each search strategy:

- **Completeness** – Does it find a solution when there is one?
- **Optimality** – Does it find the optimal solution? (lowest path cost)
- **Time complexity** – How long does it take to get a solution?
- **Space complexity** – How much memory is needed to get a solution?

3.2. Breadth-First Search (BFS)

In BFS, the root node (containing the initial state) is expanded first. Then, all of its successors are expanded next, and so on. Here, the *frontier* is initialized as a **First-In First-Out list**.

3.2.1. Breadth-First Tree Search Algorithm

```
""" uninformed.py """
from node import Node
from collections import deque
from heapq import heappush, heappop, heapify

# Find the index of "state" in "frontier"
def state_index(frontier, state):
  for i, n in enumerate(frontier):
    if n.state == state:
      return i
  return -1

def breadth_first_tree_search(problem):
  # Create the initial node containing the initial state
  initial_node = Node(problem.initial_state(), None, 0, 0)
  # Initialize a deque storing nodes to be visited
  frontier = deque()
  frontier.append(initial_node)
  counter = 0

  while True:
    # Is frontier empty?
    if not frontier:
      return (None, counter)
    else:
      counter += 1
      # Remove a node from the frontier
      n = frontier.popleft()
      # Return the node if it contains a goal state
      if problem.is_goal(n.state):
        return (n, counter)
      else: # Generate successors and add to the frontier
        for s, c in problem.successors(n.state):
          new_node = Node(s, n, n.depth+1, n.cost+c)
          frontier.append(new_node)
```

A *node* is a collection of data for conducting the search. Each node composes of

1. a state of the problem,
2. a parent node for tracing back to the initial node,
3. a cost accumulated from the initial node, and
4. a depth of the node.

```
""" node.py """
from collections import deque
from functools import total_ordering

@total_ordering
class Node:
  def __init__(self, state, parent, depth, cost):
    self.state = state    # State
    self.parent = parent  # Node generating this node
    self.depth = depth    # Depth from the initial node
    self.cost = cost      # Accumulated cost

  def __str__(self):
    return "(" + str(self.state) + "," + str(self.depth) + \
        "," + str(self.cost) + ")"

  def __repr__(self):
    return str(self)

  def __eq__(self, s):
    return isinstance(s, self.__class__) and \
        self.cost == s.cost

  def __lt__(self, s):
    return isinstance(s, self.__class__) and \
        self.cost < s.cost

```

```
28 def print_solution(n):
29   if n is None:
30     print("No solution")
31     return
32   r = deque()
33   while n is not None:
34     r.appendleft(n.state)
35     n = n.parent
36   for s in r:
37     print(s)
```

Example 3.1 Use **breadth-first tree search** to find a solution of the water measuring problem.

```
1 """ water_bfts.py """
2 import uninformed
3 import water
4 from node import print_solution
5
6 n, c  = uninformed.breadth_first_tree_search(water)
7 print_solution(n)
8 print("No. of visited nodes =", c)
```

```
[0, 0]
[0, 5]
[3, 2]
[0, 2]
[2, 0]
[2, 5]
[3, 4]
No. of visited nodes = 537
```

3.2.2. Search Tree

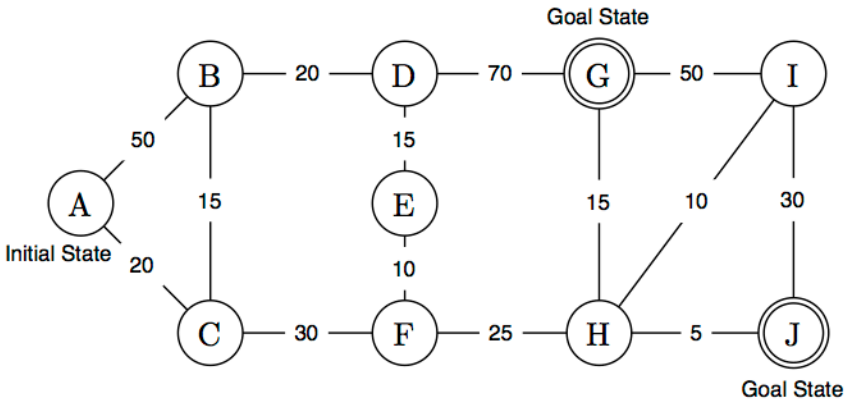
We can also draw a *search tree* to show how the search is conducted.

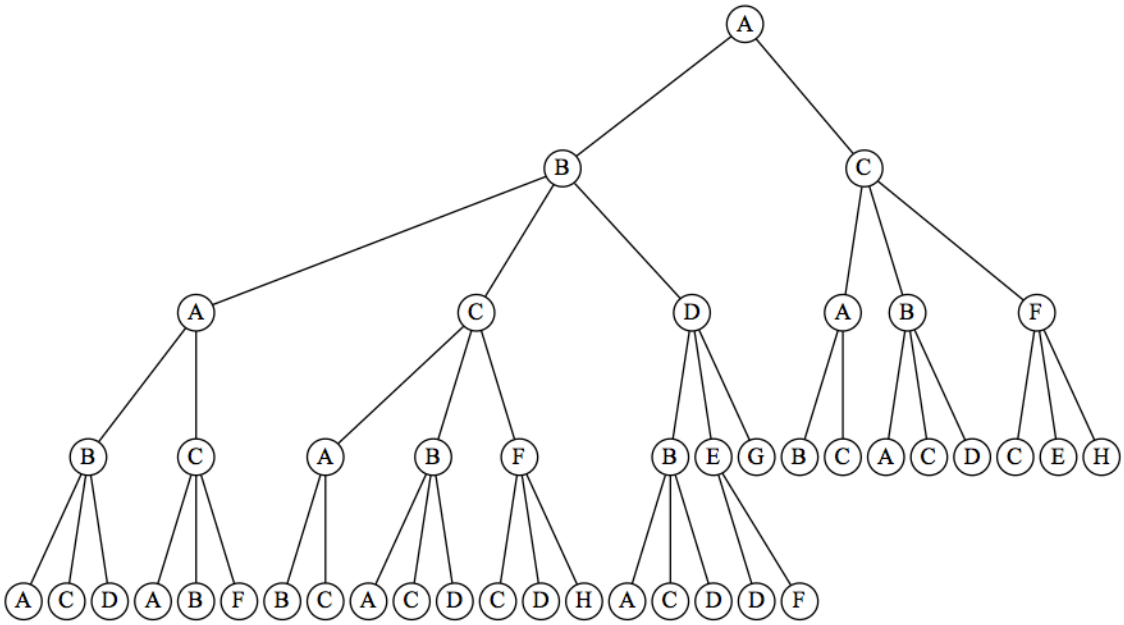
Each node in the search tree represents a node in the search. We label each node with the state corresponding to the node.

A square around a node shows that the node is visited, or removed from the *frontier* for exploration.

We also mark two numbers to a node, i.e. the node's visiting order at the top-right corner, and the path cost at the top-left corner.

Exercise 3.1 Draw a *search tree* when we conduct the *breadth-first tree search* on the following state space.





3.2.3. Breadth-First Graph Search Algorithm

```
37 def breadth_first_graph_search(problem):
38   initial_node = Node(problem.initial_state(), None, 0, 0)
39   frontier = deque()
40   frontier.append(initial_node)
41   # Initialize "explored" as a set to keep the states checked
42   explored = set()
43   counter = 0
44
45   while True:
46     if not frontier:
47       return (None, counter)
48     else:
49       counter += 1
50       n = frontier.popleft()
51       # Add the state to "explored"
52       explored.add(n.state)
53       if problem.is_goal(n.state):
54         return (n, counter)
55       else:
56         for s, c in problem.successors(n.state):
57           # Add "s" to the frontier list only when it is
58           # not in the explored set nor in the frontier list
59           if s not in explored and state_index(frontier, s) < 0:
60             new_node = Node(s, n, n.depth+1, n.cost+c)
61             frontier.append(new_node)
```

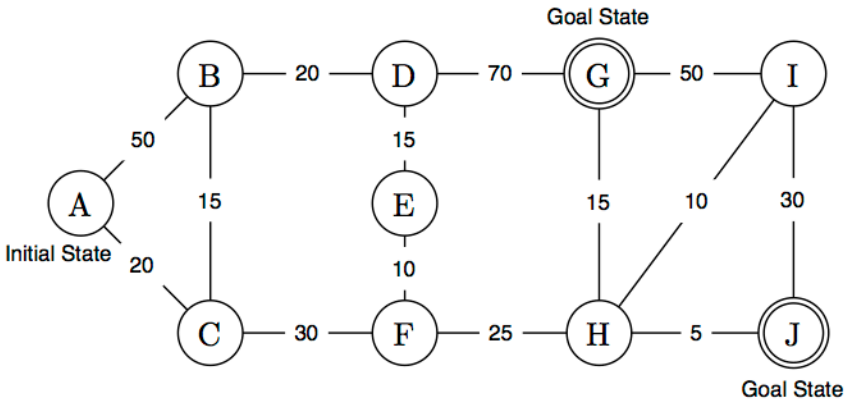
Example 3.2 Use **breadth-first graph search** to find a solution of the water measuring problem.

```
""" water_bfgs.py """
import uninformed
import water
from node import print_solution

n, c  = uninformed.breadth_first_graph_search(water)
print_solution(n)
print("No. of visited nodes =", c)
```

```
[0, 0]
[0, 5]
[3, 2]
[0, 2]
[2, 0]
[2, 5]
[3, 4]
No. of visited nodes = 14
```

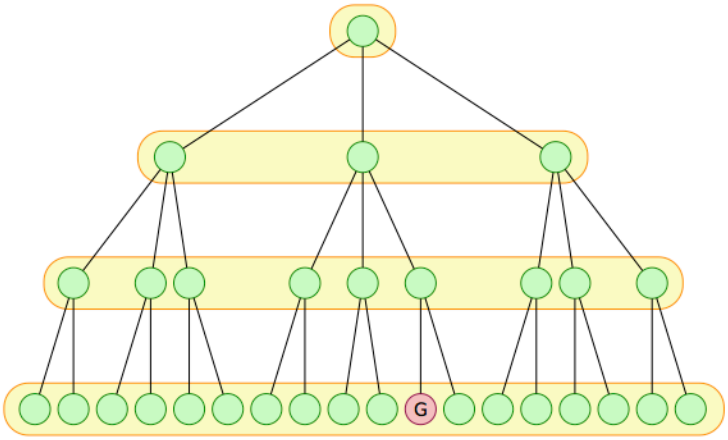
Exercise 3.2 Draw a search tree when we conduct **breadth-first graph search** to find a solution of the following state space.



3.2.4. Evaluating Breadth-first Search

Completeness

A search is *complete* when it is guaranteed to get a solution when there is one.



BFS checks the nodes level-by-level. It therefore eventually reach a goal node at the lowest depth.

Thus, BFS is complete. It always reaches *the shallowest goal*.

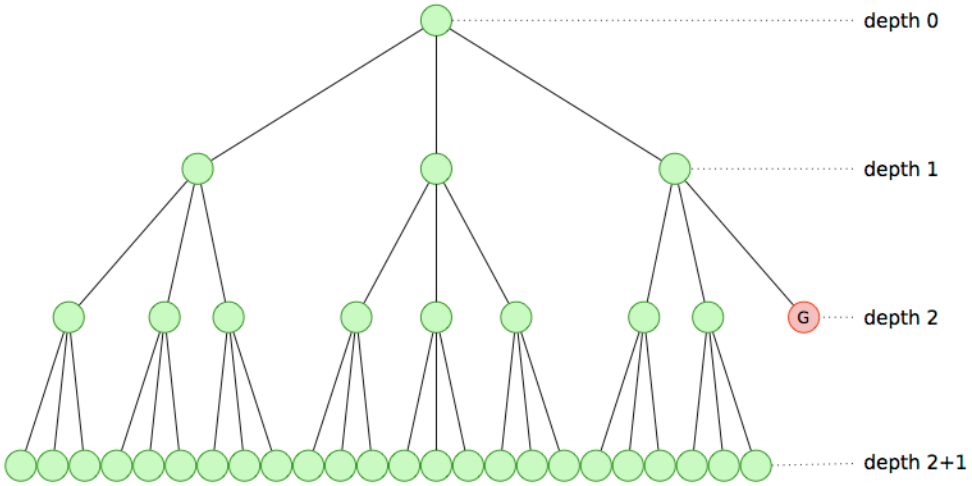
Optimality

Does BFS always return the solution with the minimum cost?
No, step costs are not considered while the search is conducted.

However, BFS is optimal when all step costs are equal.

Time complexity

Time complexity can be measured from *the number of nodes generated until a solution is found.*



Let b be the maximum number of successors generated from one state. b is called the *branching factor*. Let d be the depth of the shallowest goal. We evaluate the time complexity based on *worst-case analysis*. We therefore assume that the goal node is the rightmost node at the depth d.

The number of generated nodes, N, can be computed from

$$\begin{aligned} N &= 1 + b + b^2 + \cdots + b^d + (b^{d+1} - b) \\ &= O(b^{d+1}) \end{aligned}$$

Space complexity

Space complexity is measured from *the maximum number of nodes stored in the memory* until the search ends.

Since we do not have any clue in which branch the goal node is, BFS needs to keep all the generated nodes in the memory until the search ends.

Thus, the space complexity of BFS is $O(b^{d+1})$

Time and memory requirements for breadth-first search assuming branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node. (Fig 3.13)

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabytes
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

3.3. Uniform-cost Search

UCS always expands the node with the lowest path cost. Therefore, the first visited goal node is the one with the lowest path cost.

```
63 def uniform_cost_graph_search(problem):
64   initial_node = Node(problem.initial_state(), None, 0, 0)
65   frontier = []
66   frontier.append(initial_node)
67   explored = set()
68   counter = 0
69
70   while True:
71     if not frontier:
72       return (None, counter)
73     else:
74       counter += 1
75       n = heappop(frontier)
76       explored.add(n.state)
77       if problem.is_goal(n.state):
78         return (n, counter)
79       else:
80         for s, c in problem.successors(n.state):
81           if s not in explored:
82             i = state_index(frontier, s)
83             new_node = Node(s, n, n.depth+1, n.cost+c)
84             if i >= 0:
85               if frontier[i].cost > new_node.cost:
86                 frontier[i] = new_node
87                 heapify(frontier)
88             else:
89               heappush(frontier, new_node)
```

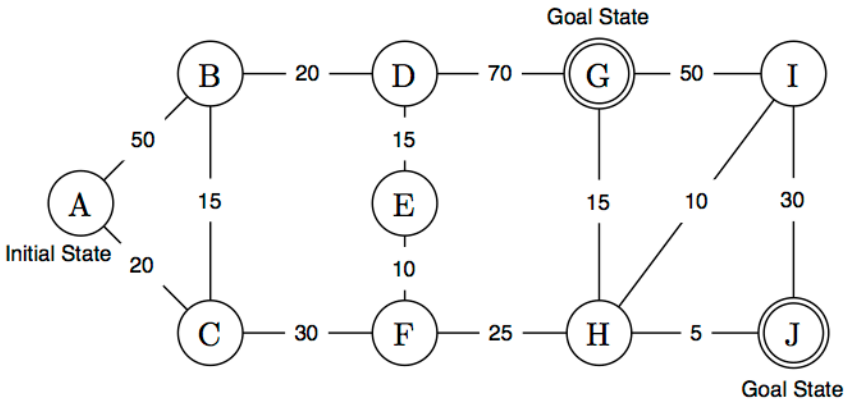
Example 3.3 Use **uniform-cost graph search** to find a solution of the water measuring problem.

```
""" water_bfgs.py """
import uninformed
import water
from node import print_solution

n, c  = uninformed.uniform_cost_graph_search(water)
print_solution(n)
print("No. of visited nodes =", c)
```

```
[0, 0]
[0, 5]
[3, 2]
[0, 2]
[2, 0]
[2, 5]
[3, 4]
No. of visited nodes = 14
```

Exercise 3.3 Draw a search tree when we conduct **uniform-cost graph search** to find a solution of the following state space.



3.3.1. Evaluating Uniform-cost Search

Completeness

UCS is based on the same concept as BFS, but the *cost* is used instead of the *depth*.

Thus, UCS is complete.

Optimality

When all the step costs are positive, the first goal node found by UCS is the one with the minimum cost. Therefore, UCS guarantees the optimal solution.

Time complexity

UCS expands nodes similarly to BFS, but the level in UCS is the *increment of cost* instead of depth.

Time complexity is $O(b^{\lfloor c^*/\epsilon \rfloor + 1})$ where c^* is the cost of the optimal solution and ϵ is the minimum increment.

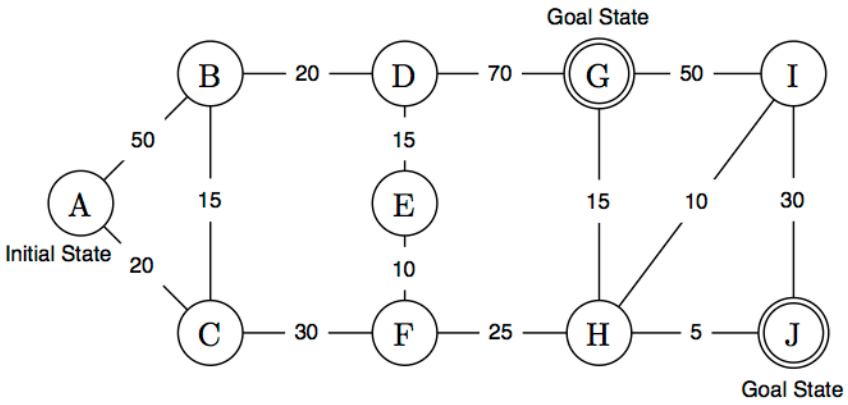
Space complexity

Space complexity is the same as the time complexity since we need to keep all the generated nodes.

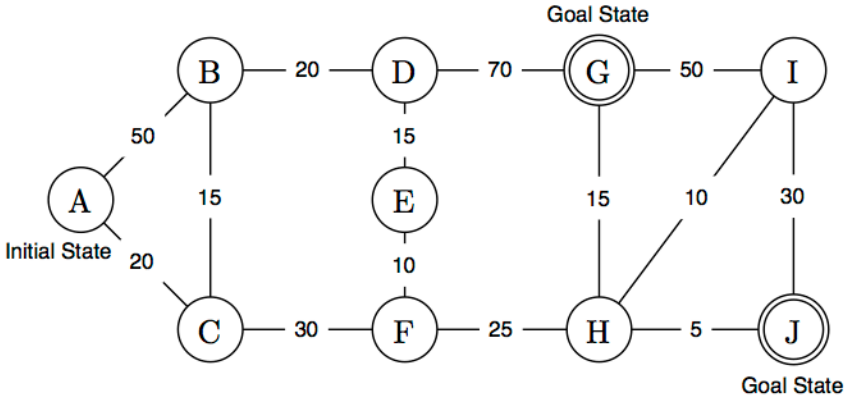
3.4. Depth-first Search

DFS always expands the latest generated node in the *frontier* list. Therefore, a **Last-In First-Out** queue or a stack is used to manage the *frontier*.

Exercise 3.4 Draw a search tree when we use depth-first tree search to find a solution.



Exercise 3.5 Draw a search tree when we use depth-first graph search to find a solution.



3.4.1. Depth-first Graph Search Algorithm

```
110 def depth_first_graph_search(problem):
111   initial_node = Node(problem.initial_state(), None, 0, 0)
112   frontier = deque()
113   frontier.append(initial_node)
114   explored = set()
115   counter = 0
116
117   while True:
118     if not frontier:
119       return (None, counter)
120     else:
121       counter += 1
122       n = frontier.pop()
123       explored.add(n.state)
124       if problem.is_goal(n.state):
125         return (n, counter)
126       else:
127         for s, c in problem.successors(n.state):
128           if s not in explored and state_index(frontier, s) < 0:
129             new_node = Node(s, n, n.depth+1, n.cost+c)
130             frontier.append(new_node)
```

3.4.2. Evaluating Depth-first Search

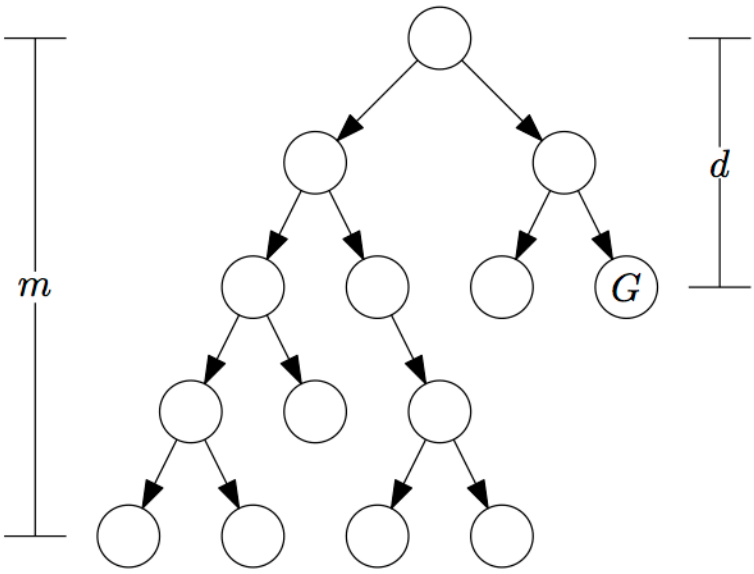
Completeness

1. Depth-first tree search may get stuck in a cycle. Thus, it is not complete.
2. Depth-first graph search is complete when the state space has a finite number of states.

Optimality

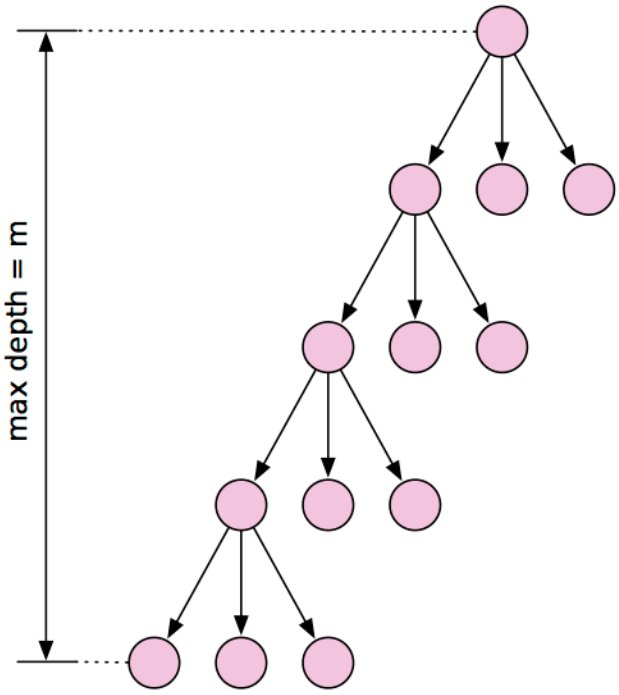
The costs are not taken into account in depth-first search.

Time complexity



Let m be *the maximum depth* from the initial node. The number of generated nodes for DFS is $O(b^m)$. Note that $m \gg d$, and m can be ∞ when the search gets stuck in a cycle.

Space complexity



DFS checks the nodes in *path-by-path*. A part of the path can be removed from the memory after they have been checked.

Therefore, the space complexity is $O(b \cdot m)$.

3.5. Depth-limited Search

DLS limits the expanding depth to a limit l. Only the nodes with $depth \leq l$ are appended to the *frontier*.

The limit needs to be appropriately set so that $l > d$, where d is the depth of the shallowest goal.

```
132 def depth_limited_graph_search(problem, limit):
133   initial_node = Node(problem.initial_state(), None, 0, 0)
134   frontier = deque()
135   frontier.append(initial_node)
136   explored = set()
137   counter = 0
138
139   while True:
140     if not frontier:
141       return (None, counter)
142     else:
143       counter += 1
144       n = frontier.pop()
145       explored.add(n.state)
146       if problem.is_goal(n.state):
147         return (n, counter)
148       elif n.depth < limit:
149         for s, c in problem.successors(n.state):
150           if s not in explored and state_index(frontier, s) < 0:
151             new_node = Node(s, n, n.depth+1, n.cost+c)
152             frontier.append(new_node)
```

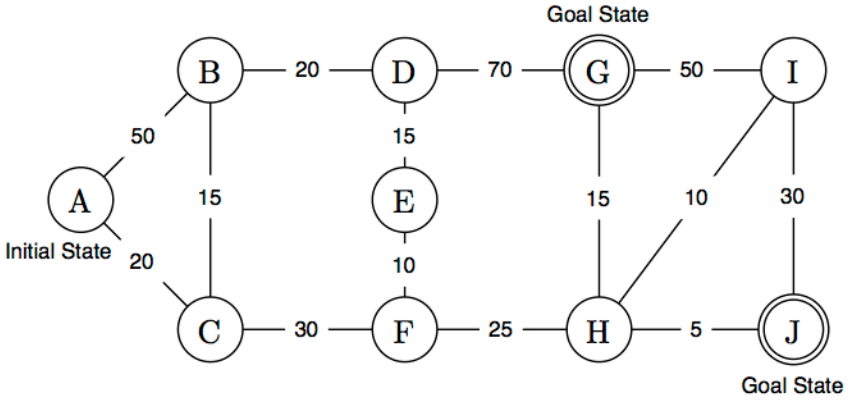
Example 3.4 Use **depth-limited graph search** to find a solution of the water measuring problem.

```
""" water_bfgs.py """
import uninformed
import water
from node import print_solution

n, c  = uninformed.depth_limited_graph_search(water, 5)
print_solution(n)
print("No. of visited nodes =", c)
print("-"*20)
n, c  = uninformed.depth_limited_graph_search(water, 10)
print_solution(n)
print("No. of visited nodes =", c)
```

```
No solution
No. of visited nodes = 12
--------------------
[0, 0]
[0, 5]
[3, 2]
[0, 2]
[2, 0]
[2, 5]
[3, 4]
No. of visited nodes = 7
```

Exercise 3.6 Draw a search tree when we use depth-limited tree search to find a solution when $l = 2$ and $l = 3$



3.5.1. Evaluating Depth-limit Search

Completeness

The performance of DLS depends on the limit l. DLS returns 'NO SOLUTION' when $l < d$ where d is the depth of the shallowest goal.

Therefore, the completeness is not guaranteed.

Optimality

The optimality is not guaranteed since the costs are not taken in account.

Time complexity

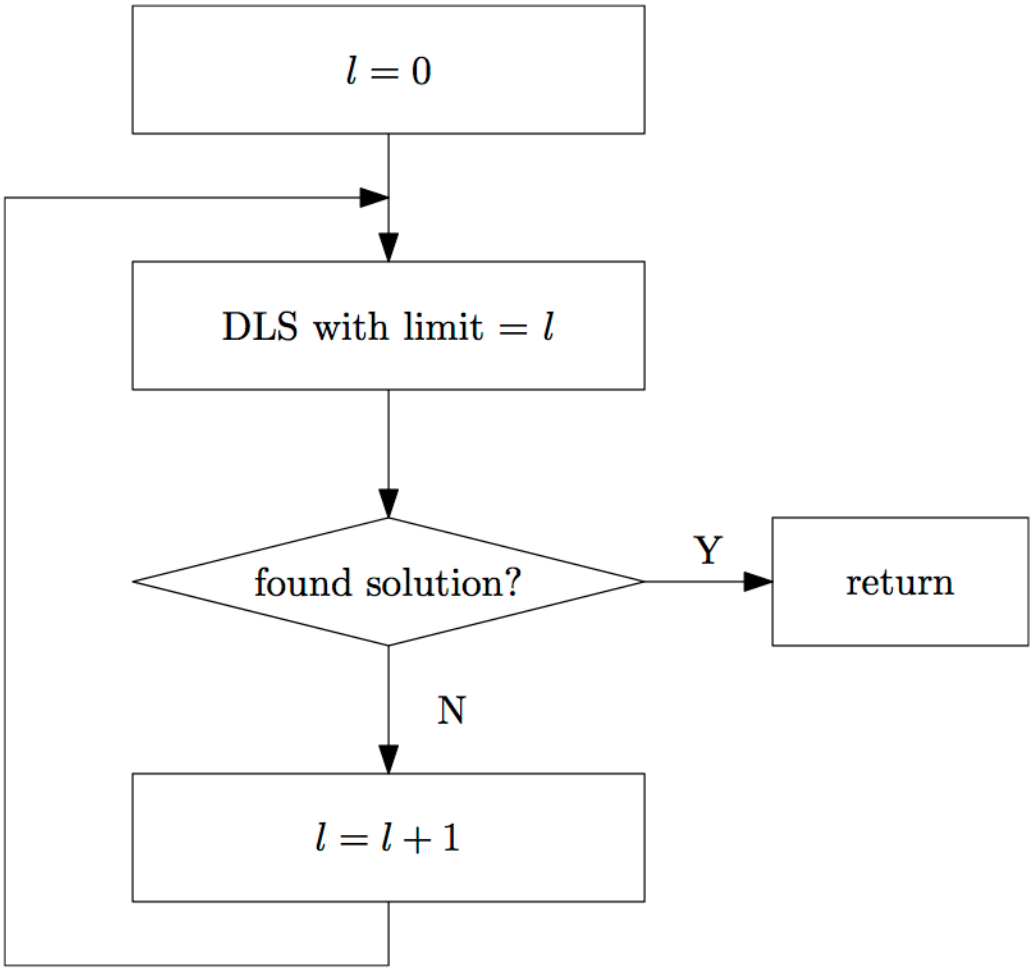
Time complexity is $O(b^l)$.

Space complexity

Space complexity is $O(b \cdot l)$.

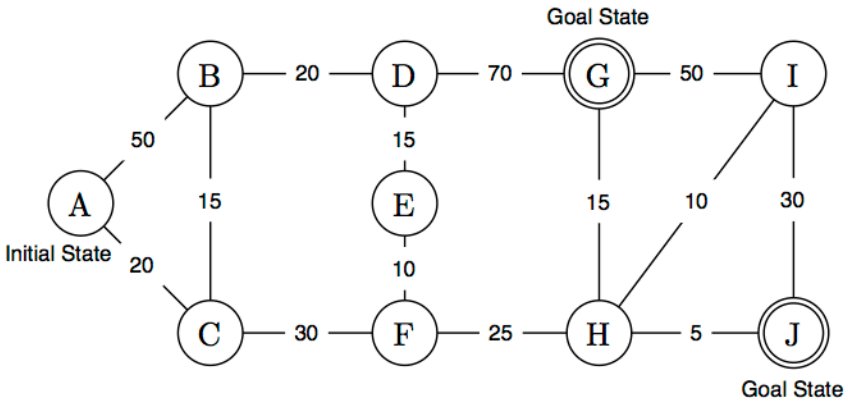
3.6. Iterative Deepening Depth-first Search

To solve the problem of determining an appropriate value of the limit and guarantee the completeness of the search, IDS iteratively conducts DLS with the gradually increased limit.




```
154 def iterative_deepening_graph_search(problem):
155   n = None
156   l = 0
157   counter = 0
158   while n is None:
159      n, c = depth_limited_graph_search(problem, l)
160      counter += c
161      l += 1
162   return n, counter
```

Exercise 3.7 Draw a search tree when we use the **iterative deepening depth-first tree search** algorithm to search for a solution



3.6.1. Evaluating Iterative Deepening Depth-first Search

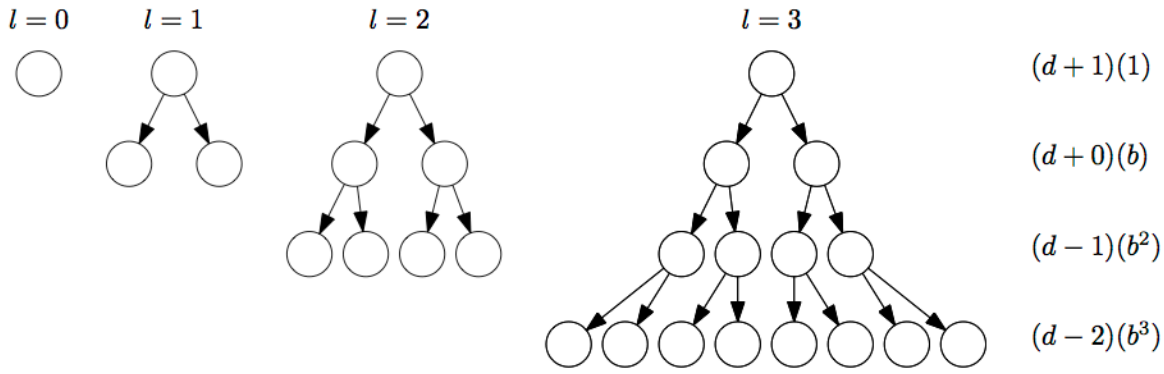
Completeness

IDS conducts DLS with the increasing limit. Therefore, it searches for a goal node level-by-level. IDS is similar to BFS such that it always ends when the shallowest goal is reached.

Optimality

Optimality is not guaranteed.

Time complexity



The number of generated nodes is

$$((d+1)(1)) + ((d)(b)) + \big((d-1)(b^2)\big) + \cdots + \big((1)(b^d)\big) = O(b^d)$$

Space complexity

IDS repeatedly conducts DLS with the increasing limit, and it ends when the limit is the shallowest goal, d. Therefore, the space complexity is $O(b \cdot d)$.

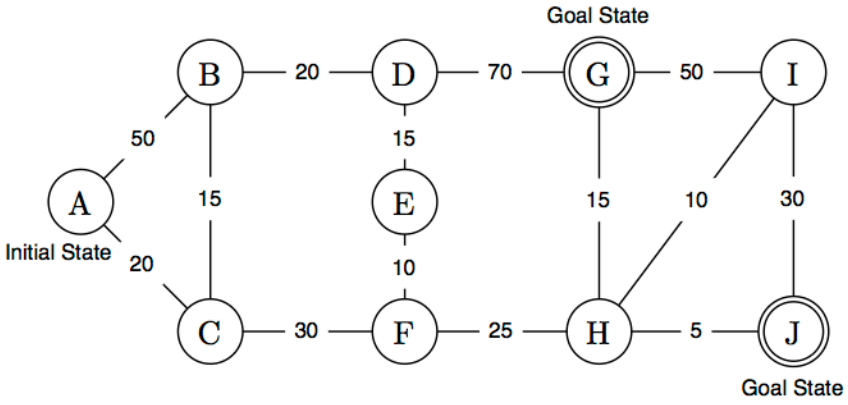
Exercise 3.8 Consider a state space where the start state is number 1 and each state k has two successors: numbers $2k$ and $2k + 1$ (Ex 3.15)

Suppose the goal state is 11. List the order in which nodes will be visited for *breadth-first search*, *depth-limited search with limit* 3, and *iterative deepening search*.

3.7. Bidirectional Search

Conducting two simultaneous searches in the opposite directions: one starts from the initial state, another one starts from a goal state. A solution is found when two frontiers intersect.

Exercise 3.9 Draw a search tree when we use the **bidirectional tree search** algorithm to search for a solution



Exercise 3.10 From the previous exercise, how well would bidirectional search work on the problem? What is the branching factor in each direction of the bidirectional search?

References

Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd edition). Pearson/Prentice Hall.
Michalewicz, F. and Fogel, D. B. (1998). How to Solve It: Modern Heuristics. Springer.