LAB 3 Uninformed Search – I

In Lab-03, we introduced methods and representations that are used for solving problems using Artificial Intelligence techniques such as search. In this Lab, we will implement an Uninformed Search Algorithm to search for solving a particular problems.

1.1 Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed.

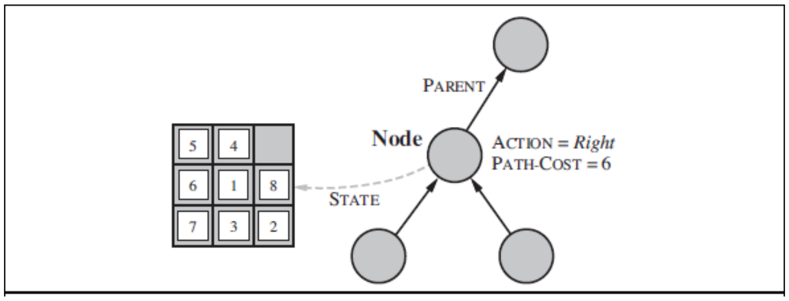


Figure 1-1. Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

For each node n of the tree, we have a structure that contains four components:

- `n.STATE`: the state in the state space to which the node corresponds;
- `n.PARENT`: the node in the search tree that generated this node;
- `n.ACTION`: the action that was applied to the parent to generate the node;
- `n.PATH-COST`: the cost, traditionally denoted by `g(n)`, of the path from the initial state to the node, as indicated by the parent pointers.

1.2 Breadth First Search:

A pseudocode implementation of breadth-first search is given below. BFS is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. In Breadth-first search the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is *generated* rather than when it is selected for expansion.

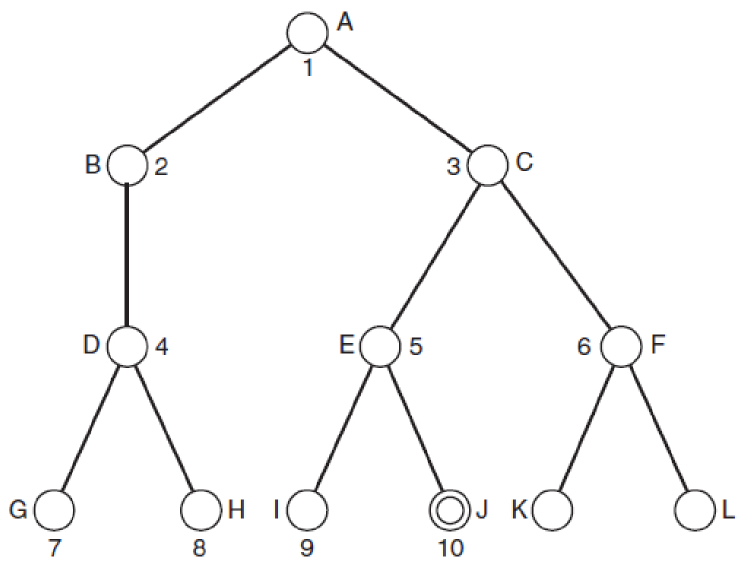


Figure 1-2 Illustration of Breadth First Search

```
Function breadth ()
{
     queue = [];      // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
{
          if is_goal (state)
               then return SUCCESS
          else add_to_back_of_queue (successors (state));
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

Figure 1-3 Breadth First Search Algorithm Pseudocode

1.3 Depth-First Search:

Depth-first search always expands the *deepest* node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors. Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

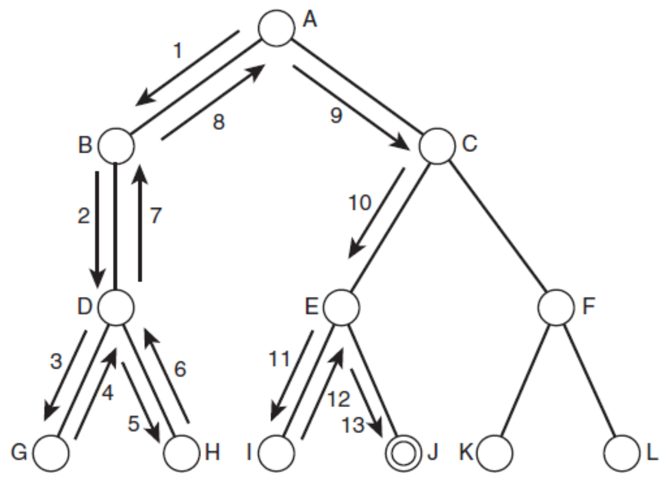
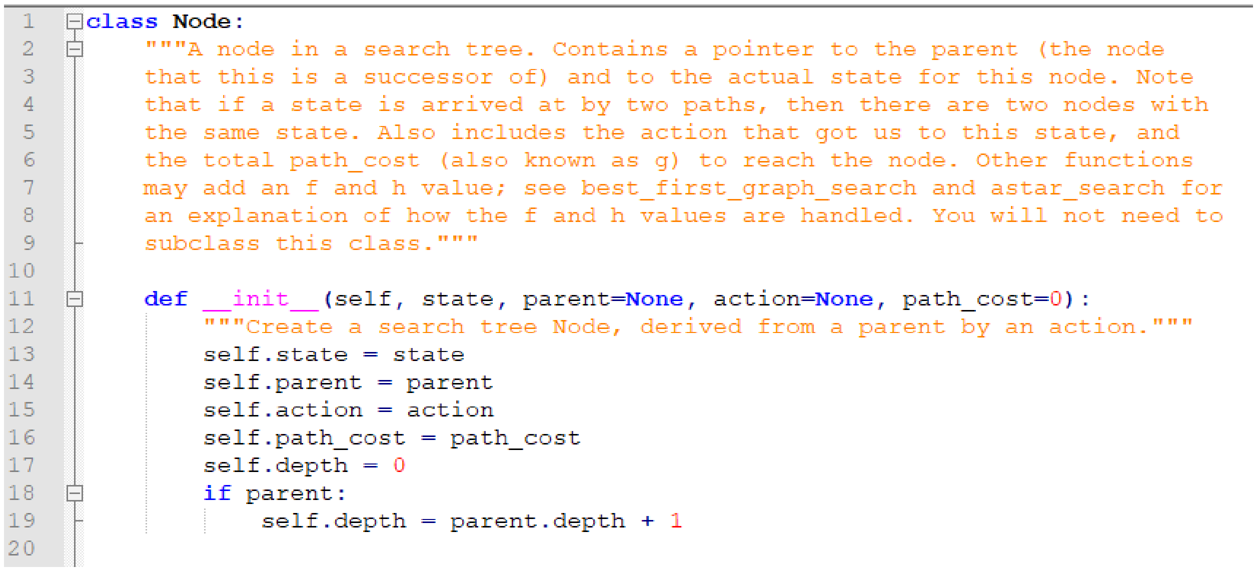


Figure 1-4 Illustration of Depth First Search

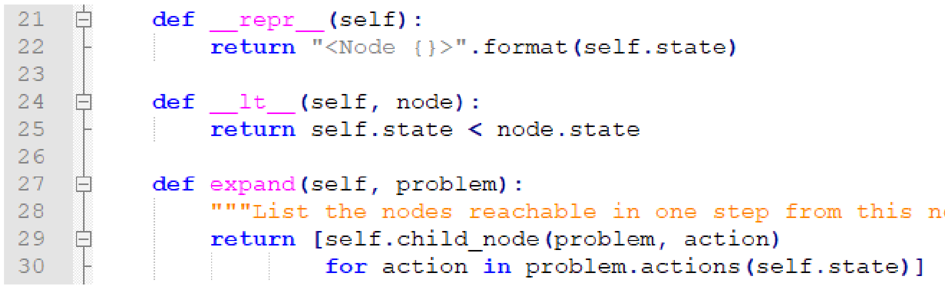
```
Function depth ()
{
     queue = [];     // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
     {
          if is_goal (state)
               then return SUCCESS
          else add_to_front_of_queue (successors (state));
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

Figure 1-5 Depth First Search Algorithm Pseudocode

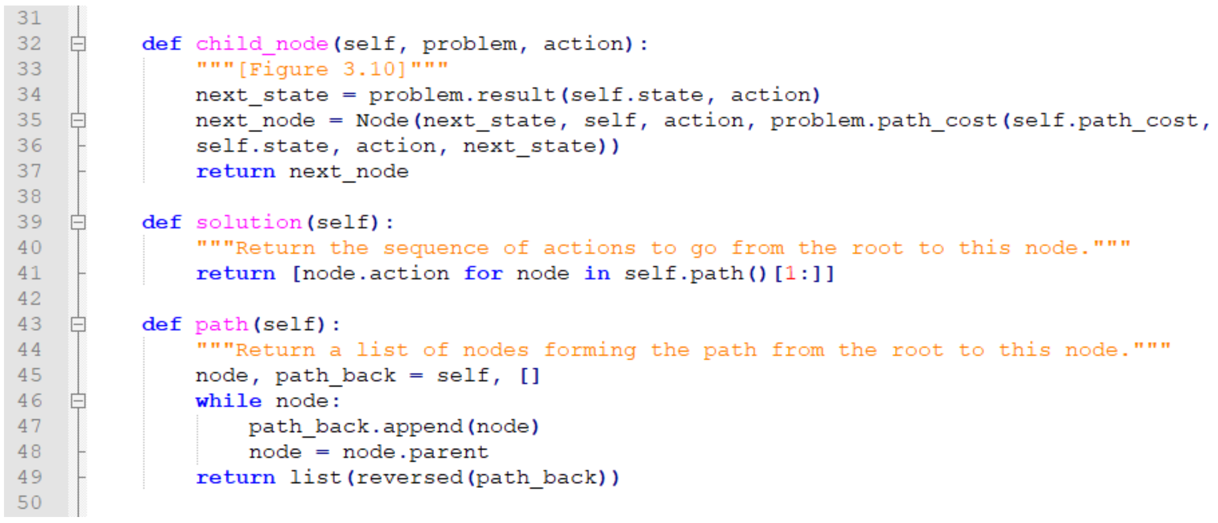
1.4 Code Structure:



```python
class Node:
    """A node in a search tree. Contains a pointer to the parent (the node
    that this is a successor of) and to the actual state for this node. Note
    that if a state is arrived at by two paths, then there are two nodes with
    the same state. Also includes the action that got us to this state, and
    the total path_cost (also known as g) to reach the node. Other functions
    may add an f and h value; see best_first_graph_search and astar_search for
    an explanation of how the f and h values are handled. You will not need to
    subclass this class."""

    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, derived from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

```



```python
    def __repr__(self):
        return "<Node {}>".format(self.state)

    def __lt__(self, node):
        return self.state < node.state

    def expand(self, problem):
        """List the nodes reachable in one step from this node."""
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]
```

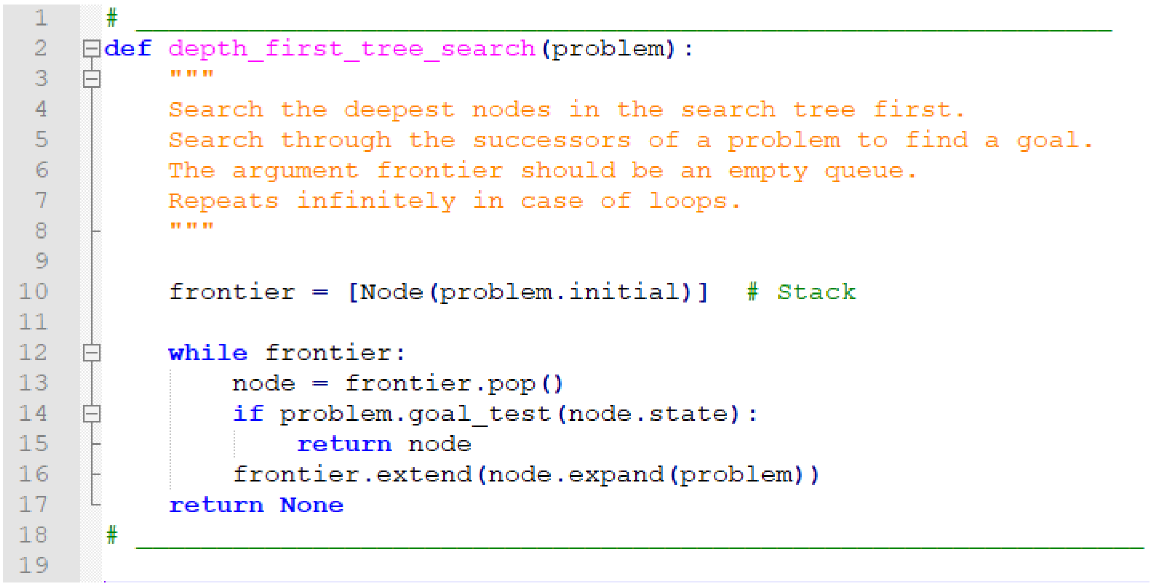


```python

    def child_node(self, problem, action):
        """[Figure 3.10]"""
        next_state = problem.result(self.state, action)
        next_node = Node(next_state, self, action, problem.path_cost(self.path_cost,
        self.state, action, next_state))
        return next_node

    def solution(self):
        """Return the sequence of actions to go from the root to this node."""
        return [node.action for node in self.path()[1:]]

    def path(self):
        """Return a list of nodes forming the path from the root to this node."""
        node, path_back = self, []
        while node:
            path_back.append(node)
            node = node.parent
        return list(reversed(path_back))

    # We want for a queue of nodes in breadth_first_graph_search or
    # astar_search to have no duplicated states, so we treat nodes
    # with the same state as equal. [Problem: this may not be what you
    # want in other contexts.]

    def __eq__(self, other):
        return isinstance(other, Node) and self.state == other.state

    def __hash__(self):
        # We use the hash value of the state
        # stored in the node instead of the node
        # object itself to quickly search a node
        # with the same state in a Hash Table
        return hash(self.state)
# ______________________________________________________________________________
```

1.4.1 Breadth First Search – Sample Code:

```python
# ______________________________________________________________________________
# Uninformed Search algorithms

def breadth_first_tree_search(problem):
    """
    Search the shallowest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = deque([Node(problem.initial)])  # FIFO queue

    while frontier:
        node = frontier.popleft()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
# ______________________________________________________________________________
```

1.4.2 Depth First Search – Sample Code:

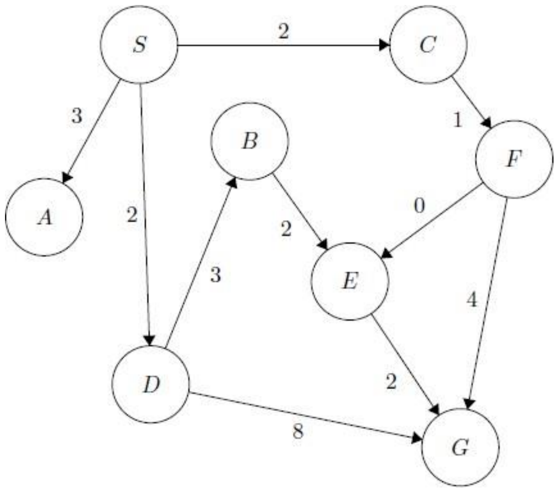


```
# ______________________________________________________________
def depth_first_tree_search(problem):
    """
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = [Node(problem.initial)]  # Stack

    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
# ________________________________________________________________
```

1.5 Lab Tasks

Exercise 4.1.

Using Breadth First Search (BFS) algorithm, write out the order in which nodes are added to the explored set, with **start state S** and **goal state G**. Break ties in alphabetical order. Additionally, what is the path returned by each algorithm? What is the total cost of each path?



Exercise 4.2.

The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. Implement and solve the problem optimally using following search algorithm. Is it a good idea to check for repeated states?

a. BFS
b. DFS

Exercise 4.3.

In Figure 1 you will find a matrix (list of lists) that represents a simple labyrinth. 0 represents a wall and 1 represents a passage. 2 represents a door that requires a key to be opened and 3 represents a supply of keys.

```
labyrinth = \
    [[0,0,0,0,0,0,1,0],
     [0,1,0,1,1,1,1,0],
     [0,1,1,1,0,1,0,0],
     [0,1,0,0,0,0,0,0],
     [0,1,1,0,1,1,3,0],
     [0,0,1,1,1,0,0,0],
     [0,1,2,0,1,1,1,0],
     [0,1,0,0,0,0,0,0]]
```

```
############. ##
##  ##. . . . ##
##. . . ##  ####
##. ############
##. . ##. . .K##
####. . . ######
##. .D##      ##
##. ############
```

a. Write a function that takes a labyrinth matrix like the one shown above as argument and draws it using the characters `##` for walls, two spaces for empty passages, `D` for doors and `K` for keys. This function should also accept a list of `(x,y)` tuples that represent positions visited by a robot in the labyrinth, you should print the character . at these points (note that you should print either a whitespace, D or K in addition to the dot).
b. Write a function that takes a labyrinth matrix like the one above and the x and y coordinates for a place in the labyrinth as arguments, and returns a list of the coordinates (tuples of x and y) of all adjacent places (vertically and horizontally) that are passages (i.e. have value 1). Note that the function must handle all input coordinates in a consistent way, and not access memory outside the size of the labyrinth array under any circumstance.
c. Write a recursive function that finds a way through the labyrinth. The entrance is at (6,0) the exit at (1,15) for the big labyrinth. The function should effectively perform a depth-first search. The base case is obviously that the goal is reached, and should return the path leading there. Therefore, you need to keep track of the path traversed so far. In the recursive case, you need to try all possible ways you can take the next step, except for those cases when you return to a state previously visited.