ALBUKHARY INTERNATIONAL UNIVERSITY

Artificial Intelligence

LAB FIVE & SIX

Problem Solving Using Search

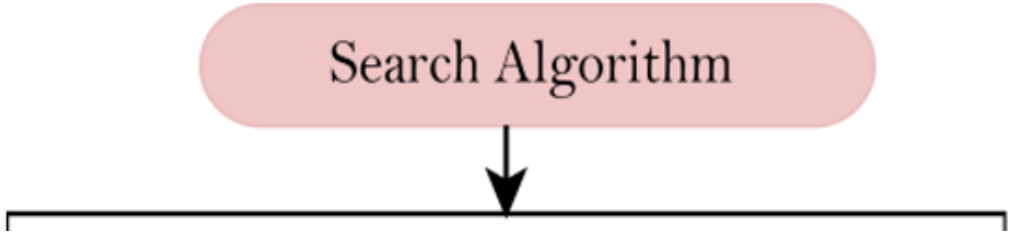
In general, an agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence. This process of looking for such a sequence is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out.

Types of search algorithms

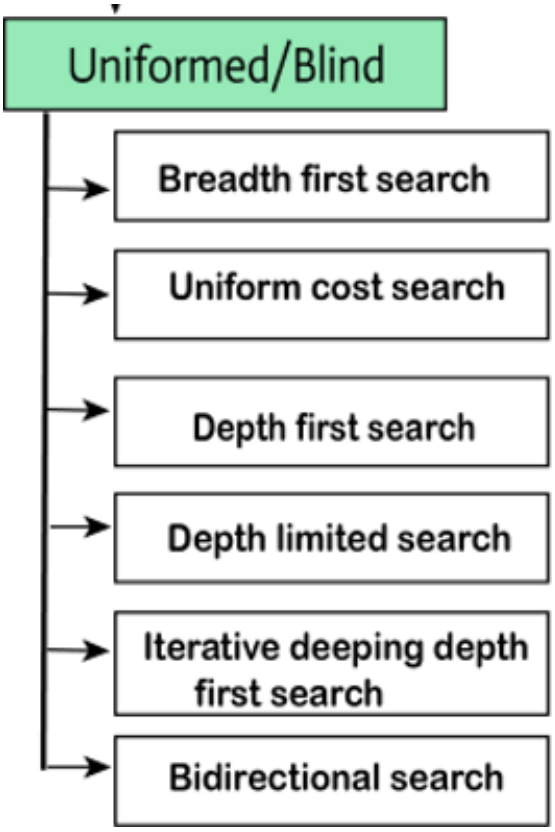
Based on the search problems we can classify the search algorithms into:

1. Uninformed (Blind search) search algorithms
2. Informed search (Heuristic search) algorithms.

As shown in the Figure below:



1. Uninformed Search



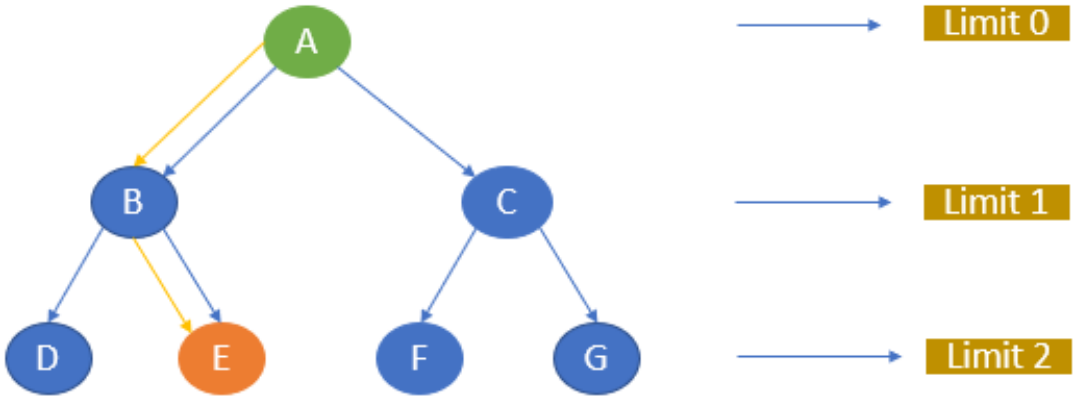
1.4 Iterative deepening depth-first search

Iterative deepening depth-first search is a combination of depth-first search and breadth-first search. IDDFS find the best depth limit by gradually adding the limit until the defined goal state is reached.

Let me try to explain this with the same example tree.

Consider, A as the start node and E as the goal node. Let the maximum depth be 2.

The algorithm starts with A and goes to the next level and searches for E. If not found, it goes to the next level and finds E.



In [ ]:

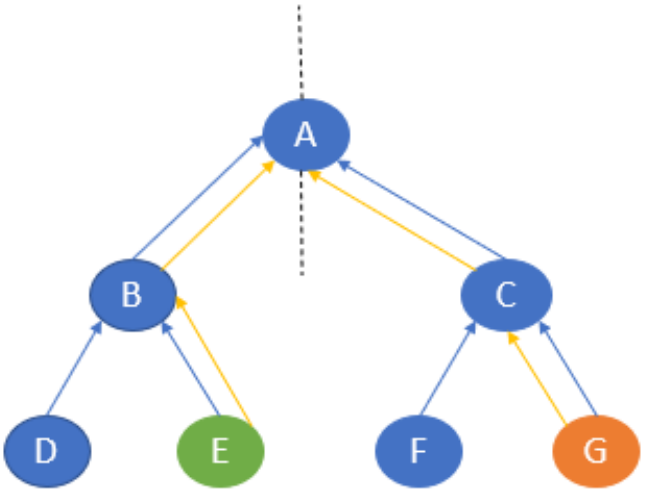
```
graph = {
 'A' : ['B','C'],
 'B' : ['D', 'E'],
 'C' : ['F', 'G'],
 'D' : [],
 'E' : [],
 'F' : [],
 'G' : []
}
path = list()
def DFS(currentNode,destination,graph,maxDepth,curList):
    curList.append(currentNode)
    if currentNode==destination:
        return True
    if maxDepth<=0:
        path.append(curList)
        return False
    for node in graph[currentNode]:
        if DFS(node,destination,graph,maxDepth-1,curList):
            return True
        else:
            curList.pop()
    return False
def iterativeDDFS(currentNode,destination,graph,maxDepth):
    for i in range(maxDepth):
        curList = list()
        if DFS(currentNode,destination,graph,i,curList):
            return True
    return False
if not iterativeDDFS('A','E',graph,3):
    print("Path is not available")
else:
    print("Path exists")
    print(path.pop())
```

1.5 Bidirectional search

The bidirectional search algorithm is completely different from all other search strategies. It executes two simultaneous searches called forward-search and backwards-search and reaches the goal state. Here, the graph is divided into two smaller sub-graphs. In one graph, the search is started from the initial start state and in the other graph, the search is started from the goal state. When these two nodes intersect each other, the search will be terminated.

Bidirectional search requires both start and goal start to be well defined and the branching factor to be the same in the two directions.

Here, the start state is E and the goal state is G. In one sub-graph, the search starts from E and in the other, the search starts from G. E will go to B and then A. G will go to C and then A. Here, both the traversal meets at A and hence the traversal ends.



In [ ]:

```
from collections import deque
class Node:
    def __init__(self, val, neighbors=[]):
        self.val = val
        self.neighbors = neighbors
        self.visited_right = False
        self.visited_left = False
        self.parent_right = None
        self.parent_left = None
def bidirectional_search(s, t):
    def extract_path(node):
        node_copy = node
        path = []
        while node:
            path.append(node.val)
            node = node.parent_right
        path.reverse()
        del path[-1]
        while node_copy:
            path.append(node_copy.val)
            node_copy = node_copy.parent_left
        return path
    q = deque([])
    q.append(s)
    q.append(t)
    s.visited_right = True
    t.visited_left = True
    while len(q) > 0:
        n = q.pop()
        if n.visited_left and n.visited_right:
            return extract_path(n)
        for node in n.neighbors:
            if n.visited_left == True and not node.visited_left:
                node.parent_left = n
                node.visited_left = True
                q.append(node)
            if n.visited_right == True and not node.visited_right:
                node.parent_right = n
                node.visited_right = True
                q.append(node)
    return False
n0 = Node('A')
n1 = Node('B')
n2 = Node('C')
n3 = Node('D')
n4 = Node('E')
n5 = Node('F')
n6 = Node('G')
n0.neighbors = []
n1.neighbors = [n0]
n2.neighbors = [n0]
n3.neighbors = [n1]
n4.neighbors = [n1]
n5.neighbors = [n2]
n6.neighbors = [n2]
```

```
print(bidirectional_search(n4, n6))
```

1.6 Uniform cost search

Uniform cost search is considered the best search algorithm for a weighted graph or graph with costs. It searches the graph by giving maximum priority to the lowest cumulative cost. Uniform cost search can be implemented using a priority queue.

Here, S is the start node and G is the goal node.

From S, G can be reached in the following ways.

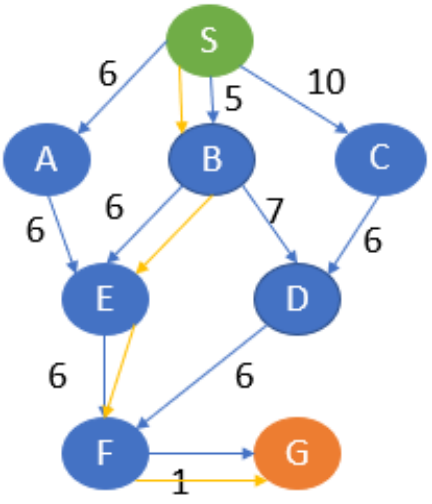
S, A, E, F, G -> 19

S, B, E, F, G -> 18

S, B, D, F, G -> 19

S, C, D, F, G -> 23

Here, the path with the least cost is S, B, E, F, G.



In [ ]:

```
graph=[['S','A',6],
       ['S','B',5],
       ['S','C',10],
       ['A','E',6],
       ['B','E',6],
       ['B','D',7],
       ['C','D',6],
       ['E','F',6],
       ['D','F',6],
       ['F','G',1]]
temp = []
temp1 = []
for i in graph:
  temp.append(i[0])
  temp1.append(i[1])
nodes = set(temp).union(set(temp1))
def UCS(graph, costs, open, closed, cur_node):
  if cur_node in open:
    open.remove(cur_node)
  closed.add(cur_node)
  for i in graph:
    if(i[0] == cur_node and costs[i[0]]+i[2] < costs[i[1]]):
      open.add(i[1])
      costs[i[1]] = costs[i[0]]+i[2]
      path[i[1]] = path[i[0]] + ' -> ' + i[1]
  costs[cur_node] = 999999
  small = min(costs, key=costs.get)
  if small not in closed:
    UCS(graph, costs, open,closed, small)
costs = dict()
temp_cost = dict()
path = dict()
for i in nodes:
  costs[i] = 999999
  path[i] = ' '
open = set()
closed = set()
start_node = input("Enter the Start State: ")
open.add(start_node)
path[start_node] = start_node
costs[start_node] = 0
UCS(graph, costs, open, closed, start_node)
goal_node = input("Enter the Goal State: ")
print("Path with least cost is: ",path[goal_node])
```