**Uninformed Search:**

# Lab 5A

Recall your previous lab on Graphs in Python where you implemented a basic path finding algorithm for Directed, Weighted graphs.

## Depth-first search:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

### Pseudocode:

Input: A graph G and a vertex v of G

Output: All vertices reachable from v labelled as discovered

A recursive implementation of DFS:

1. procedure DFS(G,v):
2. label v as discovered
3. for all edges from v to w in G.adjacentEdges(v) do
4. if vertex w is not labeled as discovered then
5. recursively call DFS(G,w)

A non-recursive implementation of DFS:

1. procedure DFS-iterative(G,v):
2. let S be a stack
3. S.push(v)
4. while S is not empty
5. v = S.pop()
6. if v is not labeled as discovered:
7. label v as discovered
8. for all edges from v to w in G.adjacentEdges(v) do
9. S.push(w)

## Iterative Deepening Depth-first search:

Iterative Deepening Depth-first search (ID-DFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found. IDDFS is equivalent to breadth-first search, but uses much less memory; on each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

### Pseudocode:

The following pseudocode shows IDDFS implemented in terms of a recursive depth-limited DFS (called DLS).

**function** IDDFS(root)

**for** depth **from** 0 **to** ∞

found ← DLS(root, depth)

**if** found ≠ null

**return** found

**function** DLS(node, depth)

**if** depth = 0 **and** node is a goal

**return** node

**if** depth > 0

**foreach** child of node

found ← DLS(child, depth−1)

**if** found ≠ null

**return** found

**return** null

## Lab Journal 5A:

Consider a room service robot that has three rooms (A, B, C) to serve. There is a service room as well.

* The robot can serve one room at a time and has to go back to the service room before

serving the next room.

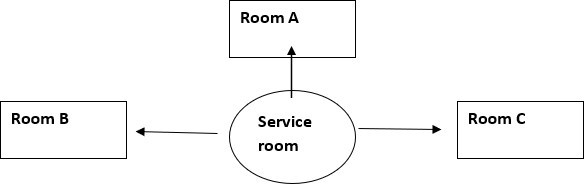
* Each room will get exactly one service.

**Goal:** all rooms get service

**States:** rooms and robot status

**Initial state:** all rooms need service, robot is in service room

**Actions:** Left move + serve, right move+ serve, up move+ serve, back



Make a state space tree. Apply Depth first and Iterative Deepening search to find the goal. Display the path and path cost. Consider ***Left move + serve***, ***right move+ serve***, have cost 5, whereas ***up move+ serve*** has unit cost and ***back operation*** has zero cost.

Your functions of DFS and IDDFS should each return the first path they find from the start node to the goal node, as well as the total cost of the path undertaken. Note that it is possible to return more than one item/variable in python using the syntax:

return variable\_1, variable\_2

rooms = ['A', 'B', 'C']

service\_room = 'S'

def is\_goal(state):

# The goal is achieved when all rooms have been serviced

return all(s == 'S' for s in state.values())

def successors(state):

# Generate all valid actions and their resulting states

moves = []

if any(s != 'S' for s in state.values()):

for room in rooms:

if state[room] != 'S':

# Move to the room and service it

new\_state = dict(state)

new\_state[room] = 'S'

moves.append(('{}+serve'.format(room), new\_state, 5))

# Move back to the service room

new\_state = dict(state)

for room in rooms:

if new\_state[room] != 'S':

new\_state[room] = 'S'

moves.append(('back', new\_state, 0))

return moves

def dfs(start\_state):

stack = [(start\_state, [])]

visited = set()

while stack:

state, path = stack.pop()

if is\_goal(state):

return path, sum(c for \_, \_, c in path)

visited.add(str(state))

for action, next\_state, cost in successors(state):

if str(next\_state) not in visited:

stack.append((next\_state, path + [(action, next\_state, cost)]))

return None, None

def iddfs(start\_state):

depth = 0

while True:

result = dls(start\_state, depth)

if result is not None:

return result

depth += 1

def dls(state, depth, path=[]):

if depth == 0:

if is\_goal(state):

return path, sum(c for \_, \_, c in path)

return None

if is\_goal(state):

return path, sum(c for \_, \_, c in path)

for action, next\_state, cost in successors(state):

result = dls(next\_state, depth - 1, path + [(action, next\_state, cost)])

if result is not None:

return result

return None

# Test the search functions

start\_state = {room: 'dirty' for room in rooms}

start\_state[service\_room] = 'S'

print('DFS:')

path, cost = dfs(start\_state)

if path is not None:

for action, state, \_ in path:

print('Action: {}, State: {}'.format(action, state))

print('Total cost: {}'.format(cost))

else:

print('No solution found.')

print('IDDFS:')

path, cost = iddfs(start\_state)

if path is not None:

for action, state, \_ in path:

print('Action: {}, State: {}'.format(action, state))

print('Total cost: {}'.format(cost))

else:

print('No solution found.')

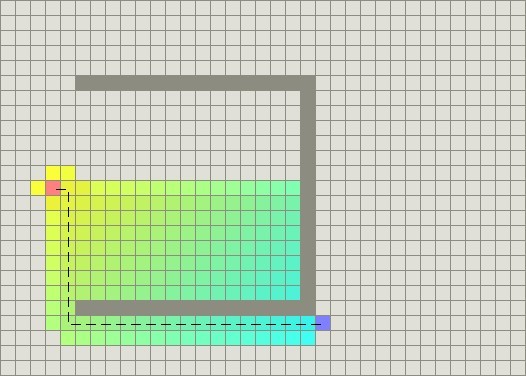
**Searching:**

# Lab 5B

Some computational problems can have many solutions. We say that these solutions are lying in the search space. Out of the many possible solutions, we try to use a searching algorithm that finds the optimal solution present in the search space.

## A\* Search:

A\* Search Algorithm was developed in 1968 to combine heuristic approaches like Greedy Best- First-Search and formal approaches like Dijsktra‘s algorithm. It‘s a little unusual in that heuristic approaches usually give you an approximate way to solve problems without guaranteeing that you get the best answer. However, A\* is built on top of the heuristic, and although the heuristic itself does not give you a guarantee, A\* *can* guarantee a shortest path. A\* is like Dijkstra‘s algorithm in that it can be used to find a shortest path. A\* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search.



The secret to its success is that it combines the pieces of information that Dijkstra‘s algorithm uses (favouring vertices that are close to the starting point) *and* information that Greedy Best- First- Search uses (favouring vertices that are close to the goal). In the standard terminology used when talking about A\*, g(n) represents the *exact cost* of the path from the starting point to any vertex n, and h(n) represents the heuristic *estimated cost* from vertex n to the goal.

In the above diagram, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A\* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

The heuristic can be used to control A\*‘s behaviour.

* At one extreme, if h(n) is 0, then only g(n) plays a role, and A\* turns into Dijkstra‘s algorithm, which is guaranteed to find a shortest path.
* If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A\* is guaranteed to find a shortest path. The lower h(n) is, the more node A\* expands, making it slower.
* If h(n) is exactly equal to the cost of moving from n to the goal, then A\* will only follow the best path and never expand anything else, making it very fast. Although you can‘t make this happen in all cases, you can make it exact in some special cases. It‘s nice to know that given perfect information, A\* will behave perfectly.
* If h(n) is sometimes greater than the cost of moving from n to the goal, then A\* is not guaranteed to find a shortest path, but it can run faster.
* At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A\* turns into Greedy Best-First-Search.

So we have an interesting situation in that we can decide what we want to get out of A\*. At exactly the right point, we‘ll get shortest paths really quickly. If we‘re too low, then we‘ll continue to get shortest paths, but it‘ll slow down. If we‘re too high, then we give up shortest paths, but A\* will run faster.

## Importance of Scale:

A\* computes f(n) = g(n) + h(n). To add two values, those two values need to be at the same scale. If g(n) is measured in hours and h(n) is measured in meters, then A\* is going to consider g or h too much or too little, and you either won‘t get as good paths or you A\* will run slower than it could.

## Algorithm:

The A\* algorithm, stripped of all the code, is fairly simple. There are two sets, OPEN and CLOSED. The OPEN set contains those nodes that are candidates for examining. Initially, the OPEN set contains only one element: the starting position. The CLOSED set contains those nodes that have already been examined. Initially, the CLOSED set is empty. Graphically, the OPEN set is the ―frontier and the CLOSED set is the ―interior‖ of the visited areas. Each node also keeps a pointer to its parent node so that we can determine how it was found.

There is a main loop that repeatedly pulls out the best node n in OPEN (the node with the lowest f value) and examines it. If n is the goal, then we‘re done. Otherwise, node n is removed from OPEN and added to CLOSED. Then, its neighbours n′ are examined. A neighbour that is in CLOSED has already been seen, so we don‘t need to look at it. You do need to check to see if the node‘s g value can be lowered, and if so, you re-open it. A neighbour that is in OPEN is scheduled to be looked at, so we don‘t need to look at it now. Otherwise, we add it to OPEN, with its parent set to n. The path cost to n′, g(n′), will be set to g(n) + movementcost(n, n′).

OPEN = priority queue containing START CLOSED = empty set

while lowest rank in OPEN is not the GOAL: current = remove lowest rank item from OPEN add current to CLOSED

for neighbours of current:

cost = g(current) + movementcost(current, neighbour) if neighbour in OPEN and cost less than g(neighbour):

remove neighbour from OPEN, because new path is better if neighbour in CLOSED and cost less than g(neighbour):

remove neighbour from CLOSED

if neighbour not in OPEN and neighbour not in CLOSED: set g(neighbour) to cost

add neighbour to OPEN

set priority queue rank to g(neighbour) + h(neighbour) set neighbour's parent to current

reconstruct reverse path from goal to start by following parent pointers

## Lab Journal 5B:

1. Group of four tourists are on their way to their camp. On the way to their destination there is a bridge they have to cross. Since it is nighttime and there is no moonlight to show the way, the tourists must light their way across the bridge with a flashlight. Tourists have only one flashlight. Unfortunately, the old and rusty bridge is very narrow and allows only for two people to cross it at the same time. Each tourist crosses the bridge at his own pace. If the two tourists are crossing the bridge together, they are crossing it at the pace of the slower one. The tourists are in hurry, so they‘re trying to cross the bridge as quickly as possible.

Your task is to write a computer program that finds an acceptable solution to the problem using state space search. An acceptable solution to the problem is the sequence of bridge crossings from bank A to bank B, that will bring the tourists across the bridge in approx. minimum amount of total time. You must solve the problem using **A\* search.**

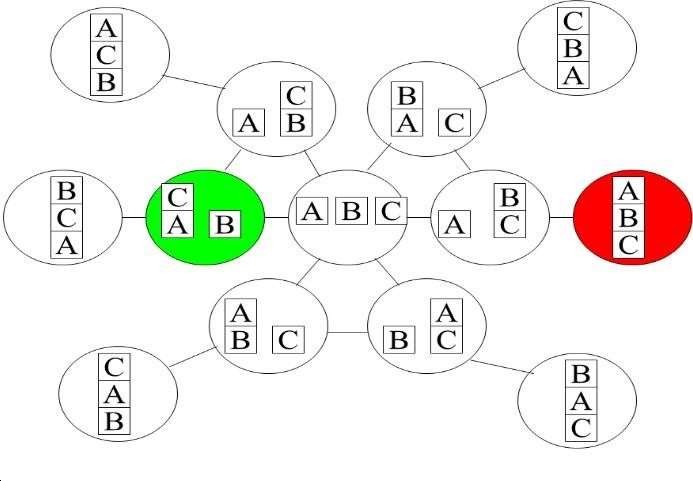
The program should take the time taken by each tourist by the user. The program should display the number of the states visited during the search, total time taken for all tourists to cross the bridge and the sequence of actions that lead to the solution.

### States: location (bank A /bank B) of the tourists Action: A  B, B  A

**Initial state: all tourists are on bank A Goal: all tourists are on bank B**

### g(n) : total actual time taken from initial state to state n h(n): total crossing time of the tourists on Bank A on state n f(n): g(n)+ h(n)

1. You need to arrange three boxes labeled as A, B and C. State space graph of the problem is shown below.



Start state is and Goal state is

|  |  |  |
| --- | --- | --- |
| b | C | A |

|  |  |  |
| --- | --- | --- |
| A | B | C |

Apply A\* search to reach the goal. Calculate total cost for goal.

h(n) = “# of blocks misplaced (if the upper block is not at its goal position” g(n) = “# of steps so far”

**code:**

class Graph:

def \_\_init\_\_(self, nodes=None, edges=None):

"""

Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes.

"""

self.nodes, self.adj = {}, {}

if nodes is not None:

self.add\_nodes\_from(nodes)

if edges is not None:

self.add\_edges\_from(edges)

def length(self):

"""

Returns the number of nodes in the graph.

"""

return len(self.nodes)

def traverse(self):

return 'V: %s\nE: %s' % (list(self.nodes.keys()), self.adj)

def add\_node(self, n):

if n not in self.nodes:

self.nodes[n] = len(self.nodes)

self.adj[n] = []

def add\_edge(self, u, v):

# undirected unweighted graph

if u in self.nodes and v in self.nodes:

self.adj[u].append(v)

self.adj[v].append(u)

def number\_of\_nodes(self):

return len(self.nodes)

def number\_of\_edges(self):

return sum(len(l) for l in self.adj.values()) // 2

class DGraph(Graph):

def add\_edge(self, u, v):

if u in self.nodes and v in self.nodes:

self.adj[u].append(v)

def compare(self, a, b):

h = 0

for i in range(len(a)):

if a[i] != b[i]:

h += 1

return h

def a(self, start, end, path=None, g=0):

if path is None:

path = [start]

else:

path.append(start)

if start == end:

return path, g

if start not in self.adj:

return None

adj\_nodes = self.adj[start]

h = []

min\_h = float('inf')

node\_h = None

for i in range(len(adj\_nodes)):

fn = g + self.compare(start, adj\_nodes[i])

h.append(fn)

if fn < min\_h:

min\_h = fn

node\_h = adj\_nodes[i]

if node\_h is None:

return None

result = self.a(node\_h, end, path, g=g + 1)

if result is not None:

return result

return None

class WGraph(Graph):

def \_\_init\_\_(self, nodes=None, edges=None):

"""

Initialize a graph object.

Args:

nodes: Iterator of nodes. Each node is an object.

edges: Iterator of edges. Each edge is a tuple of 2 nodes and a weight.

"""

super().\_\_init\_\_(nodes, edges)

self.weight = {}

def add\_edge(self, u, v, w):

if u in self.nodes and v in self.nodes:

self.adj[u].append(v)

self.adj[v].append(u)

self.weight[(u, v)] = w

self.weight[(v, u)] = w

def get\_weight(self, u, v):

return self.weight.get((u, v), float('inf'))

class DWGraph(WGraph):

def add\_edge(self, u, v, w):

if u in self.nodes and v in self.nodes:

self.adj[u].append(v)

self.weight[(u, v)] = w

D = DGraph()

nodes = ['ACB', 'BCA', 'CAB', 'BAC', 'ABC', 'CBA', 'aCB', 'bAC', 'cAB', 'bAC', 'aBC', 'cBA', 'abc']

edges = [('ACB', 'aCB'), ('aCB', 'abc'), ('BCA', 'bCA'), ('bCA', 'abc'), ('CAB', 'cAB'), ('cAB', 'abc'),

('abc', 'bAC'), ('bAC', 'BAC'), ('abc', 'cBA'), ('cBA', 'CBA'), ('abc', 'aBC'), ('aBC', 'ABC')]

for node in nodes:

D.add\_node(node)

for edge in edges:

D.add\_edge(edge[0], edge[1])

result = D.a('bCA', 'ABC')

if result is not None:

path, cost = result

print("Path:", ' -> '.join(path))

print("Cost:", cost)

else:

print("No path found.")