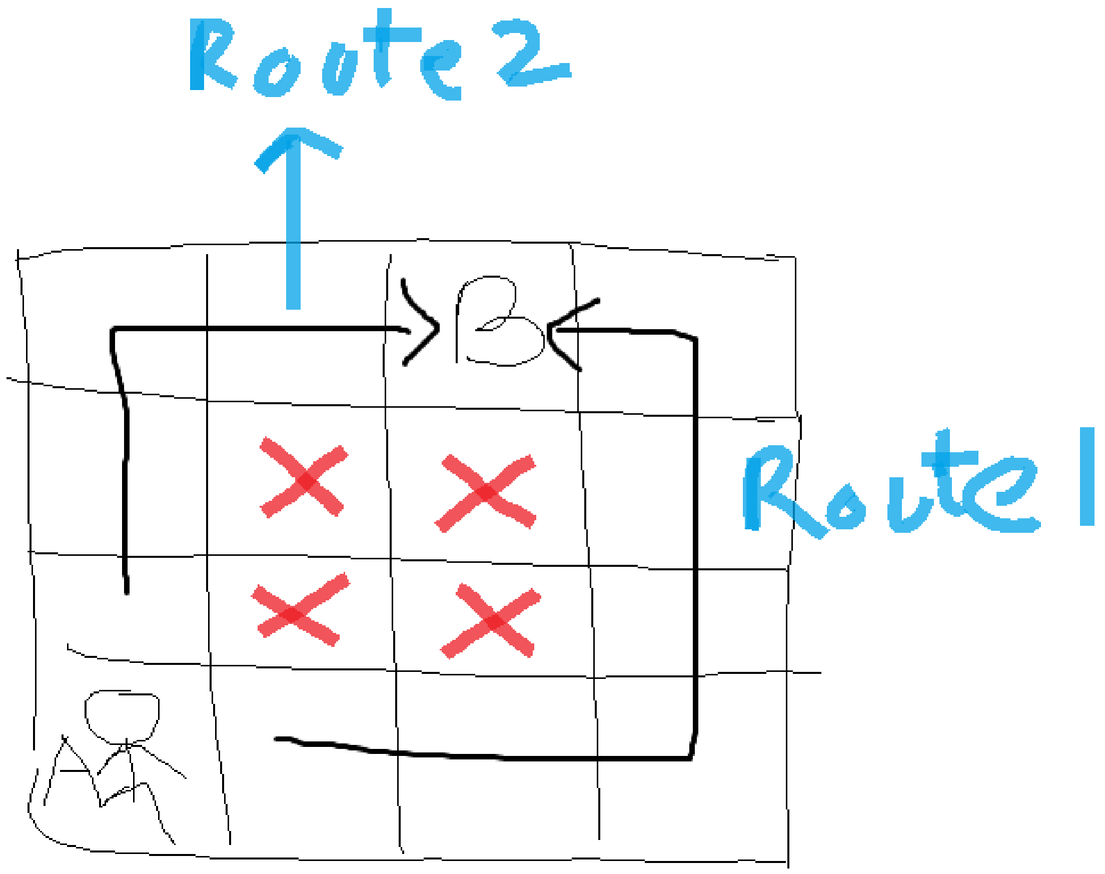
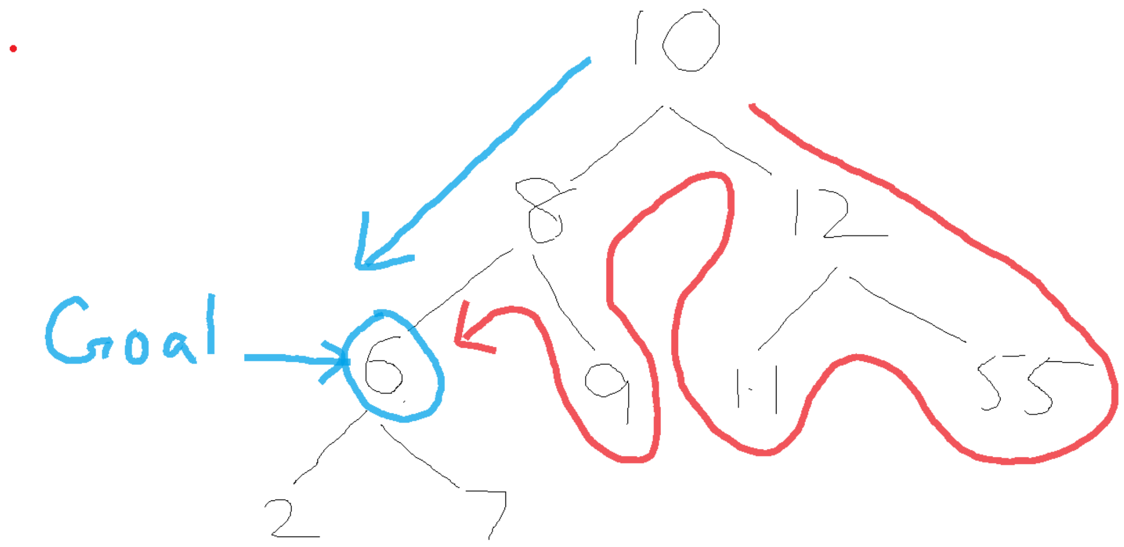
AI Lab 04
Section A

In this lab, we will be exploring several search techniques which are used in finding the goal. Goal can be anything. In the world of robotics, goal can be the destination of a robot like if it starts from A it can reach its goal B in may be 7 steps (Route 1). There can several paths to your goal. Approach can vary. One can say that any solution is great like reaching the destination by any means. Another can say that we should select the optimal path i.e. in the problem discussed above, robot can reach its goal B by taking another path which takes 5 steps only (Route 2).



In the field of searching objects, like searching for a record in a data structure. Goal is the record but what matters is the path which is being used to find the goal. Finding shortest path saves time and resources. For example, in an image shared below, to find 6: we can either take path from 10->8->6 or from 10->12->15->11->8->9->6. What matters is picking the right data structure and direction.



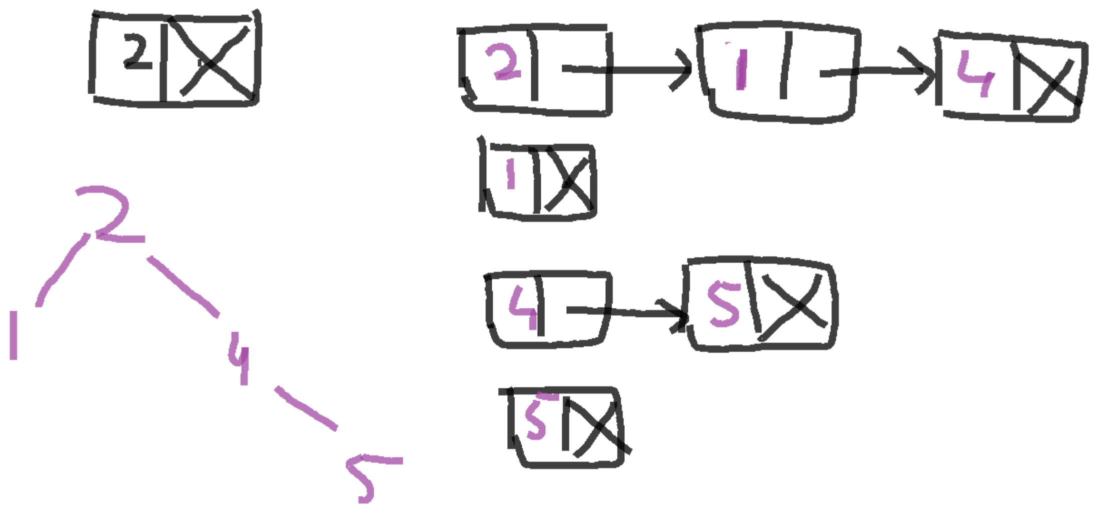
Problem:

Understanding Graphs and Adjacency List representation:

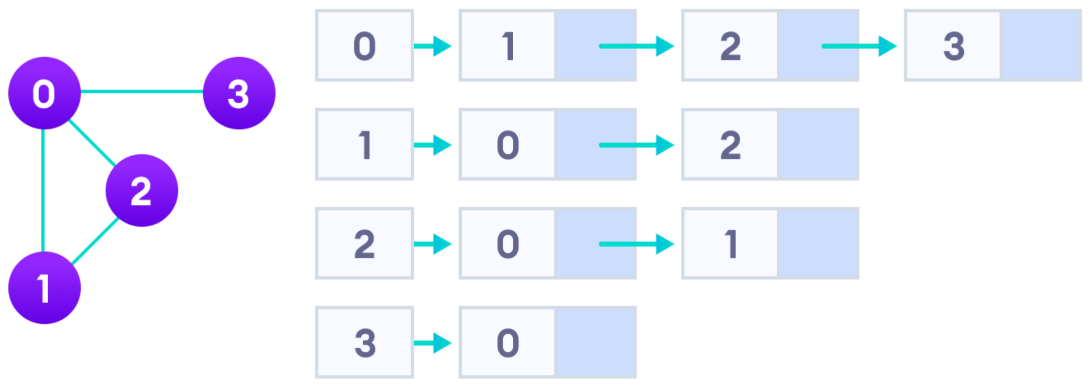
We know graphs just like in figure 2 have nodes, leaves and intermediatory nodes. In figure 2, 10 is the root; 2, 7, 11 and 55 are leaves while rest are intermediatory nodes. Question is how to save this graph in data structures? Shall we use arrays, lists or variables etc.? We are sharing a link which discusses graph implementation in different languages. Our recommendation is to see them all and understand how they are doing this.

https://www.programiz.com/dsa/graph-adjacency-list

You need to implement this in Python. We will use adjacency list for storing which are lists within list. A node, itself is a data structure. It contains data and pointer/reference to the other node. This reference points to its children as shown in figure 3.



Also, see figure 4 and its actual graphical representation to understand adjacency lists.



Task # 1:

- Create adjacency list in python and store your graph in it. You need to create your class of **Graph** and **node**. Graph class will contain functions like **add_edge**, **print-graph** and **delete_edge** etc.
- Implement **graph.get_connected_nodes(**node): Given a node name, return a list of all node names that are connected to the specified node directly by an edge.
- Implement **graph.get_edge**(node1, node2): Given two node names, return the edge that connects those nodes, or None if there is no such edge.

- Implement **graph.are_connected(**node1, node2): Return True iff there is an edge running directly between node1 and node2; False otherwise.
- Implement **graph.is_valid_path**(path): Given 'path' as an ordered list of node names, return True iff there is an edge between every two adjacent nodes in the list, False otherwise.

In addition, you're expected to know how to access elements in lists and dictionaries at this point. For some portions of this lab, you may want to use lists like either stacks or queues, as documented at

<http://docs.python.org/tut/node7.html>.

However, you should NOT import other modules (such as deque), because it will confuse the tester. You also may need to sort Python lists. Python has built-in sorting functionality, documented at

http://wiki.python.org/moin/HowTo/Sorting

The Agenda:

Different search techniques explore nodes in different orders, and we will keep track of the nodes remaining to explore in a list we will call the **agenda** (in class we called this the **queue**). Some techniques will add paths to the top of the agenda, treating it like a stack, while others will add to the back of the agenda, treating it like a queue. Some agendas are organized by heuristic value, others are ordered by path distance, and others by depth in the search tree. Your job will be to show your knowledge of search techniques by implementing different types of search and making slight modifications to how the agenda is accessed and updated.

Extending a path in the agenda:

In this problem set, a path consists of a list of node names. When it comes time to extend a new path, a path is selected from the agenda. The last node in the path is identified as the node to be extended. The nodes that connect to the extended node, the adjacent nodes, are the possible extensions to the path. Of the possible extensions, the following nodes are NOT added:

- nodes that already appear in the path.

- nodes that have already been extended (if an extended-nodes set is being used.)

As an example, if node A is connected to nodes S, B, and C, then the path ['S', 'A'] is extended to the new paths ['S', 'A', 'B'] and ['S', 'A', 'C'] (but not ['S', 'A', 'S']). The paths you create should be new objects. If you try to extend a path by modifying (or "mutating") the existing path, for example by using list .append(), you will probably find yourself in a world of hurt.

<u>The Extended-Nodes Set:</u>

An extended-set, sometimes called an "extended list" or "visited set" or "closed list", consists of nodes that have been extended, and lets the algorithm avoid extending the same node multiple times, sometimes significantly speeding up search. You will be implementing types of search that use extended-sets. Note that an extended-nodes set is a set, so if, e.g., you're using a list to represent it, then be careful that a maximum of one of each node name should appear in it. Python offers other options for representing sets, which may help you avoid this issue. The main point is to check that nodes are not in the set before you extend them, and to put nodes into the extended-set when you do choose to extend them.

<u>Returning a Search Result</u>

A search result is a path which should consist of a list of node names, ordered from the start node, following existing edges, to the goal node. If no path is found, the search should return an empty list, [].

<u>Exiting the search</u>

Non-optimal searches such as DFS, BFS, Hill-Climbing and Beam may exit either:

- when it finds a path-to-goal in the agenda

- when a path-to-goal is first removed from the agenda.

Optimal searches such as branch and bound and A* must always exit:

- When a path-to-goal is first removed from the agenda.

(This is because the agenda is ordered by path length (or heuristic path length), so a path-to-goal is not necessarily the best when it is added to the agenda, but when it is removed, it is guaranteed to have the shortest path length (or heuristic path length).)

For the sake of consistency, you should implement all your searches to exit:

• When a path-to-goal is first removed from the agenda.

<u>Task # 2:</u>

Your task is to implement following functions:

def breadth_first_search(graph, start, goal):

def depth_first_search(graph, start, goal)

The inputs to the functions are:

- graph: The graph
- start: The name of the node that you want to start traversing from
- goal: The name of the node that you want to reach

When a path to the goal node has been found, return the result as explained in the section **Returning a Search Result** (above).