Lab 2: Problem Solving by Searching

Uninformed Search: An uninformed search algorithm generates the search tree without using any domain specific knowledge. Uninformed search algorithms are also called blind search algorithms. The search algorithm produces the search tree without using any domain knowledge, which is a brute force in nature. They don't have any background information on how to approach the goal or whatsoever. But these are the basics of search algorithms in AI.

- **Depth First Search (DFS):** It is a search algorithm where the search tree will be traversed from the root node. It will be traversing, searching for a key at the leaf of a particular branch. If the key is not found the searching retraces its steps back to the point from where the other branch was left unexplored and the same procedure is repeated for that other branch.

- **Breadth-First Search(BFS):** It is another search algorithm in AI which traverses breadth-wise to search the goal in a tree. It begins searching from the root node and expands the successor node. It further expands along breadth-wise and traverses those nodes rather than searching depth-wise.

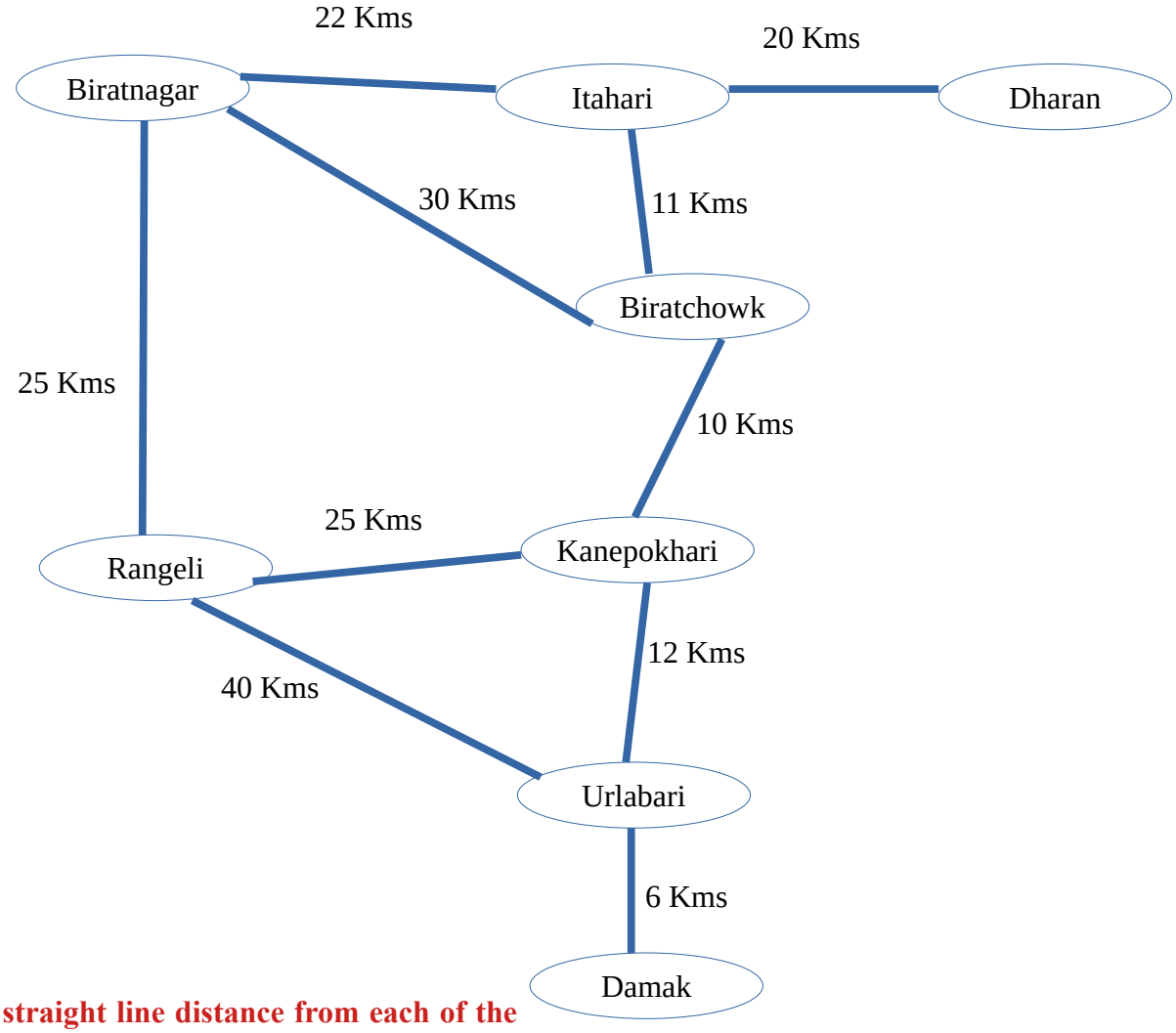
Informed Search: Informed Search algorithms have information on the goal state which helps in more efficient searching. This information is obtained by a function that estimates how close a state is to the goal state.

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). A* search algorithm finds the shortest path through the search space using the heuristic function,

$$f(n) = g(n) + h(n).$$

This search algorithm expands less search tree and provides optimal result faster.

We will use different algorithms to find the path from Biratnagar to Damak in the following map.



The straight line distance from each of the city to Damak is shown below:

Heuristic function h(n)	
Biratnagar	46 Kms
Itahari	39 Kms
Dharan	41 Kms
Rangeli	28 Kms
Biratchowk	29 Kms
Kanepokhari	17 Kms
Urlabari	6 Kms
Damak	0 Kms

Source Code for DFS

```
import pprint

G = {
'biratnagar' : {'itahari' : 22, 'biratchowk' : 30, 'rangeli': 25},
'itahari' : {'biratnagar' : 22, 'dharan' : 20, 'biratchowk' : 11},
'dharan' : {'itahari' : 20},
'biratchowk' : {'biratnagar' : 30, 'itahari' : 11, 'kanepokhari' :10},
'rangeli' : {'biratnagar' : 25, 'kanepokhari' : 25, 'urlabari' : 40},
'kanepokhari' : {'rangeli' : 25, 'biratchowk' : 10, 'urlabari' : 12},
'urlabari' : {'rangeli' : 40, 'kanepokhari' : 12, 'damak' : 6},
'damak' : {'urlabari' : 6}
}

def DFS(G, start, goal):
        stack = list()
        prev = dict()
        visited = set()
        # Push the starting state into the stack
        stack.append(start)
        # Initialize the prev state of starting state to " "
        prev[start] = " "
        # Repeat until stack is not empty
        while(stack):
                poppedState = stack.pop()
                visited.add(poppedState)
                if poppedState == goal:
                        return True, prev
                for chimeki in G[poppedState]:
                        if chimeki not in stack and chimeki not in visited:
                                stack.append(chimeki)
                                prev[chimeki] = poppedState
                                return False, prev



def reconstruct_path(G, previous, goal):
        path = goal
        while previous[goal] != " ":
                path = previous[goal] + " -> "+ path
                goal = previous[goal]
        return path

start = 'dharan'
goal = 'damak'
goalFound, previous = DFS(G, start, goal)
if(goalFound):
        print(reconstruct_path(G, previous, goal))
else:
        print("NO SOLUTION!!")
```

Source Code for A* Search

```
from pprint import pprint
from queue import PriorityQueue

G = {
'biratnagar' : {'itahari' : 22, 'biratchowk' : 30, 'rangeli': 25},
'itahari' : {'biratnagar' : 22, 'dharan' : 20, 'biratchowk' : 11},
'dharan' : {'itahari' : 20},
'biratchowk' : {'biratnagar' : 30, 'itahari' : 11, 'kanepokhari' :10},
'rangeli' : {'biratnagar' : 25, 'kanepokhari' : 25, 'urlabari' : 40},
'kanepokhari' : {'rangeli' : 25, 'biratchowk' : 10, 'urlabari' : 12},
'urlabari' : {'rangeli' : 40, 'kanepokhari' : 12, 'damak' : 6},
'damak' : {'urlabari' : 6}
}

h = {
'biratnagar' : 46,
'itahari' : 39,
'dharan' : 41,
'rangeli' : 28,
'biratchowk' : 29,
'kanepokhari' : 17,
'urlabari' : 6,
'damak' : 0
}

def aStar(G, h, start, goal):
        PQ = PriorityQueue()
        prev = dict()
        visited = set()
        # Enqueue the starting state into the queue
        # The entries in PQ are in the format (f-score, (state,g-score))
        PQ.put((0+h[start], (start, 0)))
        # Initialize the previous state of starting state to " "
        prev[start] = " "
        # Repeat until the PQ is not empty
        while(PQ.empty() == False):
                # Get the state with least f-score from the PQ
                outStateFScore , (outState, outStateGScore) = PQ.get()
                visited.add(outState)
                if outState == goal:
                        return True, prev, outStateFScore
                for chimeki in G[outState]:
                        if chimeki not in visited:
                                chimekiGScore = outStateGScore + G[outState][chimeki]
                                PQ.put((chimekiGScore+h[chimeki], (chimeki, chimekiGScore)))
                                prev[chimeki] = outState
        return False, prev, -1

def reconstruct_path(G, previous, goal):
        path = goal
        while previous[goal] != " ":
                path = previous[goal] + " -> "+ path
                goal = previous[goal]
        return path
```

```
start = 'biratnagar'
goal = 'damak'
goalFound, previous, goalFScore = aStar(G, h, start, goal)
if(goalFound):
        print(reconstruct_path(G, previous, goal))
        print(goalFScore)
else:
        print("NO SOLUTION!!")
```