Artificial Intelligence Mini Project

A report for

Topic: **Efficient Route Planning for Visiting 32 Cities in Assam using Heuristic Search**

Submitted by

Jukta Goyari (CSE-17/19)
Khushi Shukla (CSE-19/19)

B. Tech. 8th Semester
Department of Computer Science and Engineering
Dibrugarh University Institute of Engineering and Technology
Dibrugarh University
Dibrugarh-786004, Assam

May, 2023

Contents

1 Abstract

This project aims to find the minimum-cost path between two cities(eg.: Guwahati and Dibrugarh) in Assam, considering road travel. A graph-based approach is used, representing cities as nodes and roads as edges. Algorithms such as A* search, IDA* search and Greedy Best First search are employed to efficiently determine the optimal path by evaluating cumulative costs and considering travel modes. The project successfully identifies the path with the minimum cost, offering an effective solution for navigating between the cities while considering various transportation options in Assam.

2 Introduction

Assam, a state in northeastern India, boasts diverse landscapes and cultural heritage. Traveling between different cities within Assam often involves choosing between different routes. In this project, our objective is to find the path with the minimum cost between two key cities, Guwahati and Dibrugarh, considering different route options.

To tackle this problem, we employ a graph-based approach where cities are represented as nodes, and the roads and airways connecting them are represented as edges in the graph[1]. Each edge is assigned a weight or cost reflecting the distance or expense of travel between the cities. By constructing this graph, we create a comprehensive representation of the transportation network in Assam.

To efficiently find the path with the minimum cost, we utilize heuristic search algorithms such as IDA*search, A* search and Greedy Best First search. These algorithms take into account the cumulative costs associated with traveling from the starting city, Guwahati, to the destination city, Dibrugarh. By iteratively exploring the graph, evaluating costs, and considering the travel modes (road) available for each city pair, the algorithms guide us towards the most economical route.

By implementing these algorithms and incorporating relevant information about city connectivity and associated costs, our project aims to provide an optimal and cost-effective solution for navigating between Guwahati and Dibrugarh. This solution will assist travelers and transportation planners in making informed decisions, considering both road and air travel options in Assam.

3 Problem Statement

In this project, we need to find a path with minimum cost between two cities of Assam (e.g. Guwahati and Dibrugarh). The vehicle can only travel on road between two adjacent cities.
In the given map below :The 32 cities of Assam are displayed.
The starting city is Guwahati.
The ending city is Dibrugarh

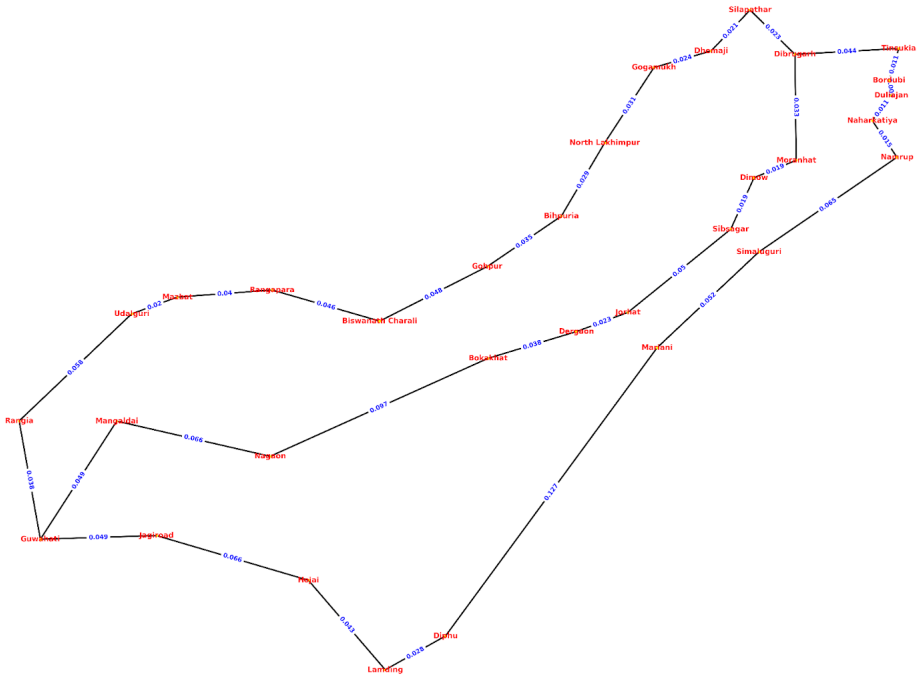

Figure 1: Map of 32 cities of Assam

4 Objectives

1. The problem consists of 32 cities in Assam.
2. The objective is to find the shortest possible route to visit all 32 cities and return to the starting point, while minimizing the cost of travel.
3. The project will implement heuristic search algorithms in Python such as greedy best-first search, A* search, and IDA* search.
4. The program will take the coordinates of the cities as input and by employing a heuristic search algorithm that identifies the closest neighboring points, we can construct a route.
5. The output of the project will be the optimal route between the cities with the goal of reducing travel time and cost.

5 Methodology

Here's a point-wise methodology for solving the Travelling Salesman Problem using heuristic approaches like Greedy best-first search, A* search, and IDA* search in Python:

1. Input is prepared for the problem with a list or dictionary of cities and their locations.
2. A cost function is defined to calculate the cost of the tour.
3. A heuristic function is implemented to estimate the remaining cost of the tour.
4. A search algorithm such as Greedy best-first search, A* search, or IDA* search is chosen.
5. The search space is defined as a tree representing all possible routes.
6. A* algorithm is implemented using a priority queue to explore the search space.
7. IDA* algorithm is implemented using iterative deepening and depth-first search.
8. The optimal route found by the algorithm is outputted.

Python is an ideal choice for solving route planning problems due to its libraries and tools for data analysis, visualization, and optimization. NumPy and Pandas are popular for data manipulation, while Matplotlib and Seaborn aid in data visualization. Optimization libraries like SciPy and PuLP can be used to tackle optimization problems. Python's flexibility and extensive ecosystem make it a powerful platform for implementing algorithms, analyzing data, visualizing results, and optimizing route planning tasks.

6 Project Outcome

1. The program will output the shortest possible route that minimizes both the total distance traveled and the amount of money needed to travel. By considering both distance and cost factors, it aims to find an optimal solution that minimizes the overall travel expenses while still ensuring the shortest route possible. The output of the program will provide the route that achieves this dual objective of minimizing distance and cost.

2. The output of the program will be a sequence of cities that represents the route, as well as the total distance travelled and the amount of money needed to travel along the route.

7 Algorithms For Problem Solving

Informed Search: Informed heuristic search, also known as informed search or heuristic search, is a problem-solving technique used in artificial intelligence and computer science to efficiently navigate search spaces. It utilizes heuristic information to guide the search process towards promising regions of the search space, with the goal of finding a solution more quickly.

In informed heuristic search, a heuristic function is used to estimate the "goodness" or "cost" of a particular state or node in the search space. The heuristic function provides an informed estimate of how close a given state is to the goal state. This estimation allows the search algorithm to prioritize exploring states that are more likely to lead to the goal.

One important aspect of informed heuristic search is the quality of the heuristic function. A good heuristic should provide accurate estimates of the remaining cost to reach the goal state while being computationally efficient. However, it is not always possible to find perfect heuristics, and sometimes approximate or admissible heuristics are used.

Informed heuristic search algorithms can be applied to various problem domains, including pathfinding, puzzle solving, and optimization. By leveraging heuristic information, these algorithms can significantly improve search efficiency and find solutions more quickly compared to uninformed search methods, such as breadth-first search or depth-first search.

Overall, informed heuristic search is a powerful technique for navigating large and complex search spaces, using heuristic information to guide the search process towards promising regions and improving the efficiency of finding solutions.

Heuristics choice: We decided to choose the straight-line distance (SLD) to be our heuristics for the Informed Search algorithms[2]. This heuristic is admissible as there exists no path that is shorter than the straight one. Moreover, the general triangle inequality is satisfied when each side is measured by the straight-line distance, thus our SLD heuristic is also consistent.

1. **Greedy Best-first search (GBFS):** It is an algorithm that selects and expands nodes based on their estimated closeness to the goal state, guided by a heuristic function. It has good time complexity when used with a dependable heuristic, as it prioritizes exploring paths that appear to be the closest to the goal.

Algorithm 1 GREEDY-BEST-FIRST-SEARCH

```
1: function GREEDY-BEST-FIRST-SEARCH(start(node), goal(node))
2:     start(node) ← a node with start(node).name = name, start(node).parent = None,
   path_cost = 0
3:     if start(node) is stop(node) then
4:         return empty path to stop(node)
5:     open ← a priority queue ordered by Distance(h(n)), with start(node) as the only
   element
6:     closed ← an empty priority queue ordered by Distance(h(n))
7:     while loop do
8:         if EMPTY?(open) then
9:             return failure
10:        parent(node) ← POP(open)
11:        if parent(node) is stop(node) then
12:            return SOLUTION(node) is parent(node)
13:        add parent(node) to closed
14:        for (child, cost) in successors(parent(node)) do
15:            g ← g[parent][child] + cost
16:            h ← h[child]
17:            child(node) ← Node(name, h, g)
18:            if child is not in closed or child is not in open then
19:                child(node).parent ← parent
20:                add child(node) to open, the priority node will be placed to the front
```

2. **A* Search:**Even though GBFS is complete (in our finite search space), it is not optimal. Therefore, we wanted to see more improvements by trying implementing A* Search. A* is one of the most popular choices for path finding problems. If combined with the given SLD heuristics, A* would assure the optimality and completeness. Furthermore, the branching factor and the sample space of this problem is not considerably large, thus we believe A* would be the most promising algorithm to solve the problem.

Algorithm 2 A* Search Algorithm

```
1:  function A-Star-Search(start(node), goal(node))
2:      start(node) ← a node with start(node).name=name,
3:      start(node).parent=None
4:      evaluationCode ← heuristic(start, goal)
5:      frontier ← a priority queue ordered by evaluationCost, with
6:      start(node) as the only element
7:      visited ← an empty set
8:      flyCost ← flyCost[start][goal]
9:      evaluationCost ← distance[node][child] +
10:     heuristic[node][child] + tollPrice[node][child]
11:     while loop do
12:         if EMPTY?(frontier) then
13:             return failure
14:         node ← POP(frontier)
15:         if node is goal then
16:             if evaluationCode ≥ flyCost then
17:                 return flyCost
18:             else
19:                 return SOLUTION(node)
20:         add node to visited
21:         for each child in successor(node) do
22:             evaluationCost ← distance[node][child] + toll[node][child] +
23:             Heuristic[node][child]
24:             if child is not in visited or child is not in frontier then
25:                 frontier ← INSERT(child, frontier)
26:             else
27:                 if child is in frontier with higher evaluationCost then
28:                     replace that frontier node with child
```

3. **IDA*:**IDA* is a search algorithm that shares similarities with A* search in terms of completeness (able to find a solution if one exists) and optimality (able to find the best solution). However, it offers a significant advantage in terms of space complexity compared to A*.

   Unlike A*, which requires storing the entire search tree in memory, IDA* takes a more memory-efficient approach. It uses a depth-first search strategy, limiting the memory usage to the current path being explored. This makes IDA* particularly advantageous when dealing with large search spaces or environments with limited memory resources.

   However, the tradeoff for the reduced space complexity is that IDA* can potentially have poorer time complexity. This is because during its search process, IDA* may end up regenerating the same nodes multiple times, leading to redundant computations and revisiting of nodes. As a result, IDA* might take longer to find a solution compared to A* search.

Algorithm 3 IDA Search

```
1:  function IDA(start(node), goal(node), heuristic(function), threshold)
2:      frontier ← stack with start node as the only element
3:      while length(frontier) > 0 do
4:          node, cost ← POP(frontier)
5:          if node is goal then
6:              return node
7:          for each node in successor(node) do
8:              pathCost ← cost + distance(node, nextNode) + heuristic(nextNode)
9:              if pathCost > threshold then
10:                 frontier.PUSH(nextNode, pathCost − heuristic(nextNode))
11:         threshold ← min value in frontier
```

8 Algorithms implementation

Firstly, we chose to randomise the input and output of the problem. Therefore, we need to fetch the straight-line distance between every city pair which seems to be an endless process of search-and-fill. We also need to run around 50 instances to test if there are any mistakes in our heuristics data, which is another time-consuming task. However, we are quite satisfied with our heuristics as it is consistent and admissible, thus our A* and IDA* will always be optimal and complete. We tried implementing IDA* algorithm using recursion at first. However, it became really troublesome when we wanted to print the final path, thus we decided to switch to stack implementation to overcome this problem[3].

8.1 A* Search Algorithm

8.1.1 Code

```
import time
import numpy as np
import pandas as pd
from random import randint
import copy
class Graph:
    def __init__(self, graph_dict):
        self.graph_dict = graph_dict
    def neighbors(self, v):
        return self.graph_dict[v]
    def h(self, n):
        return H(n)
    def a_star_algorithm(self, start, stop):
        count = 0
        if start not in cities_lst or stop not in cities_lst:
            raise Exception('City name incorrect!')
        active_lst = set([start])
        finished_lst = set([])
        cost = {}
        cost[start] = 0
        par = {}
        par[start] = start
        while len(active_lst) > 0:
            n = None
            for v in active_lst:
                if n == None or cost[v] + self.h(v) < cost[n] + self.h(n):
                    n = v
            if n == None:
                print('Path does not exist!')
                return None
            if n == stop:
                reconst_path = []
                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]
                reconst_path.append(start)
                reconst_path.reverse()
                s = 0
                for i in range(len(reconst_path) - 1):
                    for j in map[reconst_path[i]]:
                        if j[0] == reconst_path[i + 1]:
                            s += j[1]
                print('Total node expanded:', count)
                print('Total cost: ', s)
                return reconst_path
            for m, weight in self.neighbors(n):
                count += 1
                if m not in active_lst and m not in finished_lst:
                    active_lst.add(m)
                    par[m] = n
                    cost[m] = cost[n] + weight
                else:
                    if cost[m] > cost[n] + weight:
                        cost[m] = cost[n] + weight
                        par[m] = n
                        if m in finished_lst:
                            finished_lst.remove(m)
                            active_lst.add(m)
            active_lst.remove(n)
            finished_lst.add(n)
        print('Path does not exist!')
        return None
# travelling cost between adjacent cities
df = pd.read_csv('/content/route-info1.csv')
data = df.to_numpy()
highway_data = data
fee = highway_data[:, 2:3] * 10 + highway_data[:, 3:4]
highway_fee = np.hstack((highway_data[:, :2], fee))
final_fee = highway_fee
print(final_fee[0:5])
```

```
map = dict()
for i in final_fee:
    map[i[0]] = map.get(i[0], [])
    map[i[0]].append((i[1], i[2]))
    map[i[1]] = map.get(i[1], [])
    map[i[1]].append((i[0], i[2]))
cities_lst = map.keys()
print(list(cities_lst))
# getting heuristics data
df2 = pd.read_csv('/content/city-label1.csv', header=None)
da = df2.to_numpy()
da = da.reshape((2, -1))
label = dict()
da = list(da)
for i in range(len(da[0])):
    label[da[0][i]] = i
df1 = pd.read_csv('/content/heuristics.csv', header=None)
data1 = df1.to_numpy()
# find straight-line distance from start city to other cities
start = 'Guwahati'
destination = 'Dibrugarh
def find_heuristic(start, destination):
    if destination not in cities_lst:
        raise Exception("This city is not included in the map")
    data3 = []
    index = label[destination]
    for i in label:
        data3.append([i, data1[index, label[i]]])
    return data3
data4 = find_heuristic(start, destination)
print('Heuristics')
h = dict()
for i in data4:
    h[i[0]] = h.get(i[0], 0) + float(i[1] * 10)
def H(n):
    return h[n]
print(h)
# Output
begin = time.time()
solver = Graph(map)
path = solver.a_star_algorithm(start, destination)
print('Path found:')
print(*path, sep=' --> ')
end = time.time()
print('Time: ', end - begin)
```

8.1.2 Output

```
Total node expanded: 45
Total cost:  4074.2999999999997
Path_found:
Guwahati --> Rangia --> Udalguri --> Mazbat --> Rangapara --> Biswanath Charali --> Gohpur --> Bihpuria --> North Lakhimpur --> Gogamukh --> Dhemaji --> Silapathar --> Dibrugarh
time:  0.004695892333984375
```

Figure 2: Output of A* algorithm

8.2 IDA* Search

8.2.1 Code

```
import numpy as np
import pandas as pd
import time
from random import randint
# import time
import copy

class Graph:
    def __init__(self, graph_dict):
        self.graph_dict = graph_dict

```

```
    def neighbors(self, v):
        return self.graph_dict[v]

    def h(self, n):
        return H(n)

    def iterative_deepening_a_star(self, start, goal):
        if start not in cities_lst or goal not in cities_lst:
            raise Exception('City name incorrect!')

        min_fringe = 0
        threshold = self.h(start)
        self.u = 0
        count_node = []

        def IDA(start, goal, threshold, min_fringe):
            global path_cost
            global par
            global real_path
            global par1
            global s
            global u
            global count
            global visited
            global visited1
            global min_goal
            li = 0
            count = 0
            s = 0
            par = dict()
            visited = []
            visited1 = []
            stack = [[start, 0]]
            min_goal = float('inf')

            while len(stack) != 0:
                if len(stack) <= li:
                    visited.pop()

                node, cost1 = stack.pop()
                visited.append((node))
                li = len(stack)

                if node == goal:
                    s += 1
                    # save the shortest route to destination
                    if cost1 < min_goal:
                        min_goal = cost1
                        par1 = par.copy()

                # check whether its neighbor's value of f surpasses threshold
                for m, weight in self.neighbors(node):
                    p = (m, weight)
                    if p not in visited:
                        count += 1
                        path_cost = cost1 + weight + self.h(m)
                        # print('FF: ', m, path_cost)
                        # if f(neighbor) > threshold => put it in the fringe, the new threshold will be the smallest value in the fringe
                        if path_cost > threshold:
                            if path_cost < min_fringe:
                                min_fringe = path_cost
                            else:
                                par[(m, path_cost - self.h(m))] = (node, cost1)
                                stack.append([m, path_cost - self.h(m)])

            self.u = min_fringe

        while True:
            min_fringe = float('inf')
            IDA(start, goal, threshold, min_fringe)
            count_node.append(count)
            threshold = self.u
            if s != 0:
                break

```

```
        print('Number of nodes expanded in each iteration: ', count_node)
        print('Total nodes expanded: ', sum(count_node))
        curr = (goal, min_goal)

        # print path
        lst = []

        while (curr != None):
            lst.append(curr)
            if curr == (start, 0):
                break
            curr = par1[curr]

        lst.reverse()
        lst1 = [i[0] for i in lst]
        s1 = 0

        for i in range(len(lst1) - 1):
            for j in map[lst1[i]]:
                if j[0] == lst1[i+1]:
                    s1 += j[1]

        print('Total cost: ', s1, 'Rupees')
        print('Path: ')
        return lst1


# travelling cost between adjacent cities
df = pd.read_csv('/content/route-info1.csv')
data = df.to_numpy()
highway_data = data
fee = highway_data[:, 2:3] * 10 + highway_data[:, 3:4]
highway_fee = np.hstack((highway_data[:, :2], fee))
final_fee = highway_fee
print(final_fee[0:5])
map = dict()
for i in final_fee:
    map[i[0]] = map.get(i[0], [])
    map[i[0]].append((i[1], i[2]))
    map[i[1]] = map.get(i[1], [])
    map[i[1]].append((i[0], i[2]))
cities_lst = map.keys()
# getting heuristics data
df2 = pd.read_csv('/content/city-label1.csv', header=None)
da = df2.to_numpy()
da = da.reshape((2, -1))
label = dict()
da = list(da)
for i in range(len(da[0])):
    label[da[0][i]] = i
df1 = pd.read_csv('/content/heuristics.csv', header=None)
data1 = df1.to_numpy()
# find straight-line distance from start city to other cities
start = 'Guwahati'
destination = 'Dibrugarh'
def find_heuristic(start, destination):
    if destination not in cities_lst:
        raise Exception("This city is not included in the map")
    data3 = []
    index = label[destination]
    for i in label:
        data3.append([i, data1[index, label[i]]])
    return data3
data4 = find_heuristic(start, destination)
print('Heuristics')
h = dict()
for i in data4:
    h[i[0]] = h.get(i[0], 0) + float(i[1] * 10)
def H(n):
    return h[n]
# output
begin = time.time()
solver = Graph(map)
path1 = solver.iterative_deepening_a_star(start, destination)
print(*path1, sep=' --> ')
end = time.time()
```

```
print('Time: ', end - begin)
```

8.2.2 Output

```
Number of nodes expanded in each iteration:  [501, 533, 535, 541, 557, 561, 563, 567, 575, 579, 639, 647, 679, 689, 691, 699, 727, 729, 731, 735, 739, 745, 761, 765, 771, 779, 7
Total nodes expanded:  747365
Total cost:  4074.2999999999997 Rupees
Path:
Guwahati --> Rangia --> Udalguri --> Mazbat --> Rangapara --> Biswanath Charali --> Gohpur --> Bihpuria --> North Lakhimpur --> Gogamukh --> Dhemaji --> Silapathar --> Dibrugarh
time:  7.246728420257568
```

Figure 3: Output of IDA* algorithm

8.3 Greedy BFS

8.3.1 Code

```
import queue
import time
import numpy as np
import pandas as pd
from random import randint
import copy
class Node:
    def __init__(self, name, h=0, g=0, par=None):
        self.name = name
        self.h = h
        self.g = g
        self.par = par
    def __lt__(self, other):
        if other == None:
            return False
        return self.h < other.h
    def __eq__(self, other):
        if other == None:
            return False
        return self.name == other.name
class Graph:
    def __init__(self, graph_dict):
        self.graph_dict = graph_dict
        self.Path = []
    def neighbors(self, v):
        return self.graph_dict[v]
    def h(self, n):
        return heuristic[n]
    def getPath(self, node):
        if node.par != None:
            self.getPath(node.par)
        self.Path.append(node.name)
    def BestFirst_Search(self, start, stop):
        Open = queue.PriorityQueue()
        Closed = queue.PriorityQueue()
        Open.put(Node(start))
        while True:
            if Open.empty() == True:
                print('Cannot be found!')
                return []
            node = Open.get()  # current node
            Closed.put(node)
            if node.name == stop:
                print('\nSuccessful! The solution of the Greedy Best-First-Search algorithm is:')
                print('Total cost: {}'.format(node.g))
                self.getPath(node)
                return self.Path
            for (child, cost) in self.graph_dict[node.name]:
                g = node.g + cost
                h = heuristic[child]
                tmp = Node(name=child, h=h, g=g)

                if (tmp not in Open.queue) and (tmp not in Closed.queue):
                    tmp.par = node
```

```
                    Open.put(tmp)
def Input():
    global start
    global destination
    start = 'Guwahati'
    destination = 'Dibrugarh'
def ToGraph():
    global cities_lst
    global map
    global heuristic
    global label
    global data1
    # travelling cost between adjacent cities
    df = pd.read_csv('/content/route-info1.csv')
    highway_data = df.to_numpy()
    fee = highway_data[:, 2:3] * 10 + highway_data[:, 3:4]
    highway_fee = np.hstack((highway_data[:, :2], fee))
    final_fee = highway_fee
    map = dict()
    for i in final_fee:
        map[i[0]] = map.get(i[0], [])
        map[i[0]].append((i[1], i[2]))
        map[i[1]] = map.get(i[1], [])
        map[i[1]].append((i[0], i[2]))
    cities_lst = map.keys()
    # getting heuristics data
    df2 = pd.read_csv('/content/city-label1.csv', header=None)
    da = df2.to_numpy()
    da = da.reshape((2, -1))
    label = dict()
    da = list(da)
    for i in range(len(da[0])):
        label[da[0][i]] = i
    df1 = pd.read_csv('/content/heuristics.csv', header=None)
    data1 = df1.to_numpy()
def Solver():  # for a single test with input
    solver = Graph(map)
    path = solver.BestFirst_Search(start, destination)
    print(*path, sep=' --> ')
def Heuristic(start, destination):
    global heuristic
    def find_heuristic(start, destination):
        if destination not in cities_lst:
            raise Exception("This city is not included in the map")
        data3 = []
        index = label[destination]
        for i in label:
            data3.append([i, data1[index, label[i]]])
        return data3
    data4 = find_heuristic(start, destination)
    heuristic = dict()
    for i in data4:
        heuristic[i[0]] = heuristic.get(i[0], 0) + float(i[1] * 10)
# # For test purpose, uncomment the two lines below
ToGraph()
Input()
start_time = time.time()
Heuristic(start, destination)
Solver()
end_time = time.time()
print('Total time:', end_time-start_time)
```

8.3.2 Output

```
Successfull! The solution of Greedy Best-First-Search algorithm is:
Total cost: 6596.6
Guwahati --> Jagiroad --> Hojai --> Lamding --> Diphu --> Mariani --> Simaluguri --> Namrup --> Naharkatiya --> Duliajan --> Bordubi --> Tinsukia --> Dibrugarh
Total time: 0.0012249946594238281
```

Figure 4: Output of GBFS algorithm

9 Results

Here are the results obtained from three different methods of calculating the minimum cost of travel between two cities:

Algorithm used	Cost	Time taken	Time Complexity	Space Complexity	Optimality
Greedy BFS	6596.6	0.00084	$O(b^m)$	$O(b \cdot d)$	Not guaranteed can get trapped in local optima if the heuristic is not admissible or not consistent.
A* Search	4074.299	0.004312	$O(b^d)$	$O(b^d)$	Guarantees an optimal solution if the heuristic function is admissible and consistent.
IDA* Search	4074.299	9.4541	$O(b^d)$	$O(b \cdot d)$	Guarantees an optimal solution if the heuristic function is admissible and consistent.

Table 1: Comparison Table

10 Conclusion

In conclusion, for our problem, A* search is the most suitable choice. Both A* search and IDA* are complete and optimal algorithms. However, on average, A* search tends to have a slightly better running time.

IDA* is a viable option, but it suffers from regenerating the same nodes multiple times, leading to poor time complexity, especially when dealing with large state spaces. Despite this drawback, IDA* is still considered a fairly good algorithm.

On the other hand, GBFS (Greedy Best-first search) exhibits stable and efficient space complexity. Moreover, when guided by the SLD heuristic, it can have even better time complexity. However, it's important to note that GBFS cannot guarantee optimality in the context of our Route Planning problem.

To summarize, A* search is the preferred choice due to its completeness, optimality, and slightly better average running time compared to IDA*. While GBFS has good space and time complexity, it lacks the guarantee of optimality in this particular problem domain.

References

[1] G. Brewka, "Artificial intelligence—a modern approach by stuart russell and peter norvig, prentice hall. series in artificial intelligence, englewood cliffs, nj.," *The Knowledge Engineering Review*, vol. 11, no. 1, pp. 78–79, 1996.

[2] S. Edelkamp and S. Schrodl, *Heuristic search: theory and applications*. Elsevier, 2011.

[3] C. Wilt, J. Thayer, and W. Ruml, "A comparison of greedy search algorithms," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 1, 2010, pp. 129–136.