Martian Rover Navigation System

A project report submitted to

Delhi Technological University

For Discrete Mathematics (Co-205)
Innovative Project

Bachelor of Technology - Computer Engineering

Submitted by

B.Tech 3rd semester

Anshuman Raj Chauhan (2K19/CO/067)

Anshuman Roy (2K19/CO/068)

Under the esteemed guidance of, Ms Tanya Malhotra

DELHI TECHNOLOGICAL UNIVERSITY

Vision:

"To be a world class University through education, innovation and research for the service of humanity"

Mission:

1. To establish centres of excellence in emerging areas of science, engineering, technology, management and allied areas.
2. To foster an ecosystem for incubation, product development, transfer of technology and entrepreneurship.
3. To create an environment of collaboration, experimentation, imagination and creativity.
4. To develop human potential with analytical abilities, ethics and integrity.
5. To provide environment friendly, reasonable and sustainable solutions for local & global needs.

DEPARTMENT OF COMPUTER SCIENCE

AND ENGINEERING

Vision:

Department of Computer science and engineering to be a leading world class technology department playing its role as a key node in national and global knowledge network, thus empowering the computer science industry with the wings of knowledge and the power of innovation.

Mission:

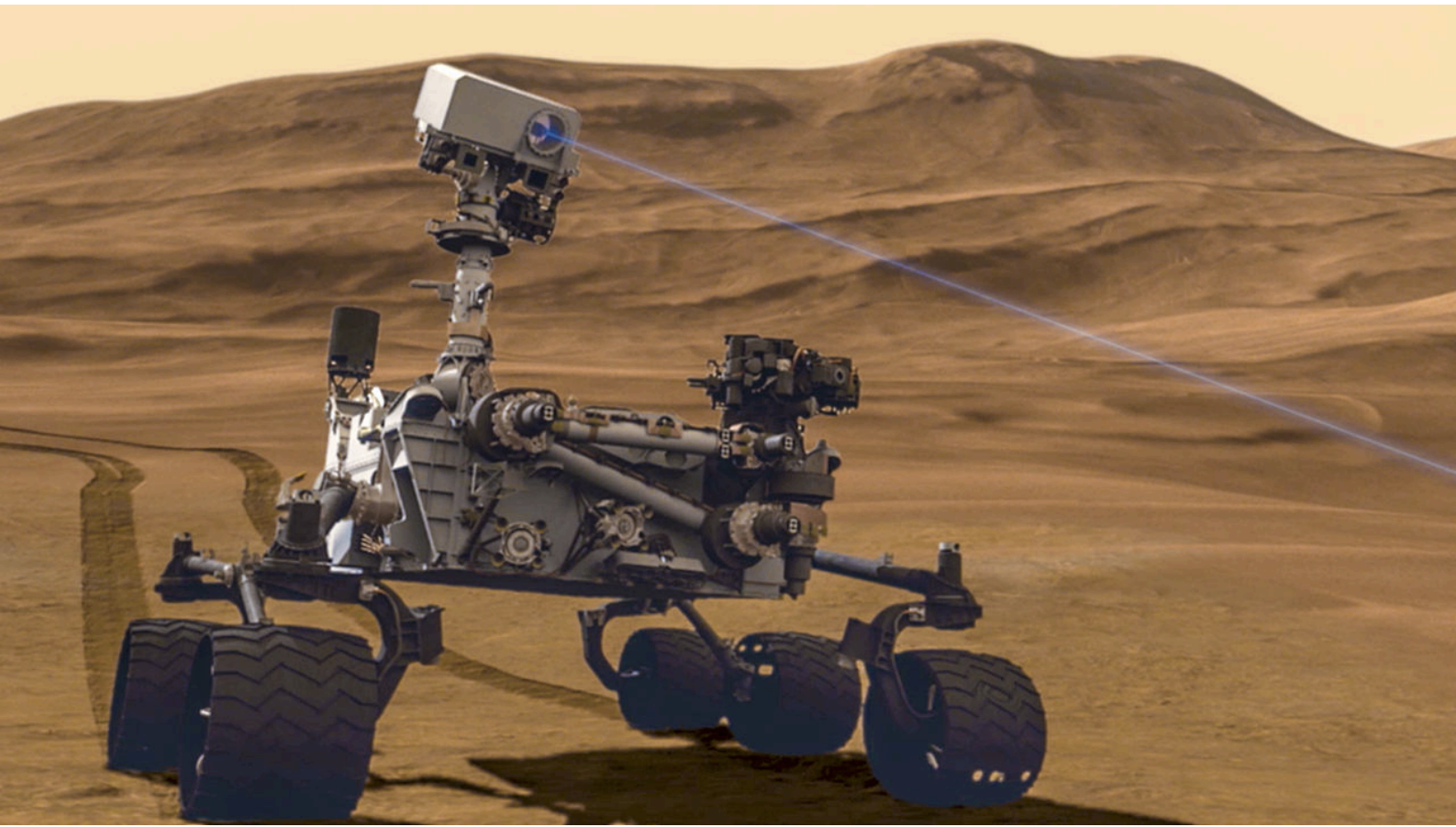
The mission of the department is as follows:

1. To nurture talent of students for research, innovation and excellence in the field of computer engineering starting from undergraduate level.
2. To develop highly analytical and qualified computer engineers by imparting training on cutting edge technology.
3. To produce socially sensitive computer engineers with professional ethics.
4. To focus on R&D environment in close partnership with industry and foreign universities.
5. To produce well-rounded, up to date, scientifically tempered, design oriented engineers and scientists capable of lifelong learning.

Contents

Aim

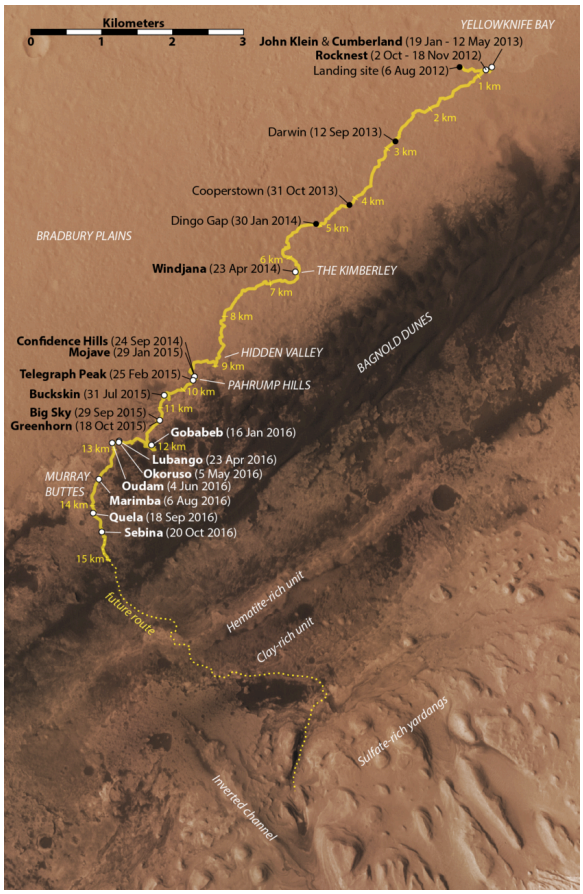
To build a navigation system for Martian rovers in order to maximise research on the red planet.
(ie- finding paths with maximum research with certain time constraints).



Abstract

Humans have always been very curiosity driven. This drive makes us want to explore more, learn more and understand more the world we have around us.

Most recently space exploration has been at the forefront of this human endeavour, and Mars at the centre of it.
Be it NASA, ISRO, SpaceX or any other space agency across the globe, everyone invariably has plans to land their rovers on Mars and understand the red planet in ways we don't yet do.



Curiosity's lifetime journey

Maximising research through these rovers is what this project aims at.
The system developed finds paths with maximum research in constrained time.

Introduction

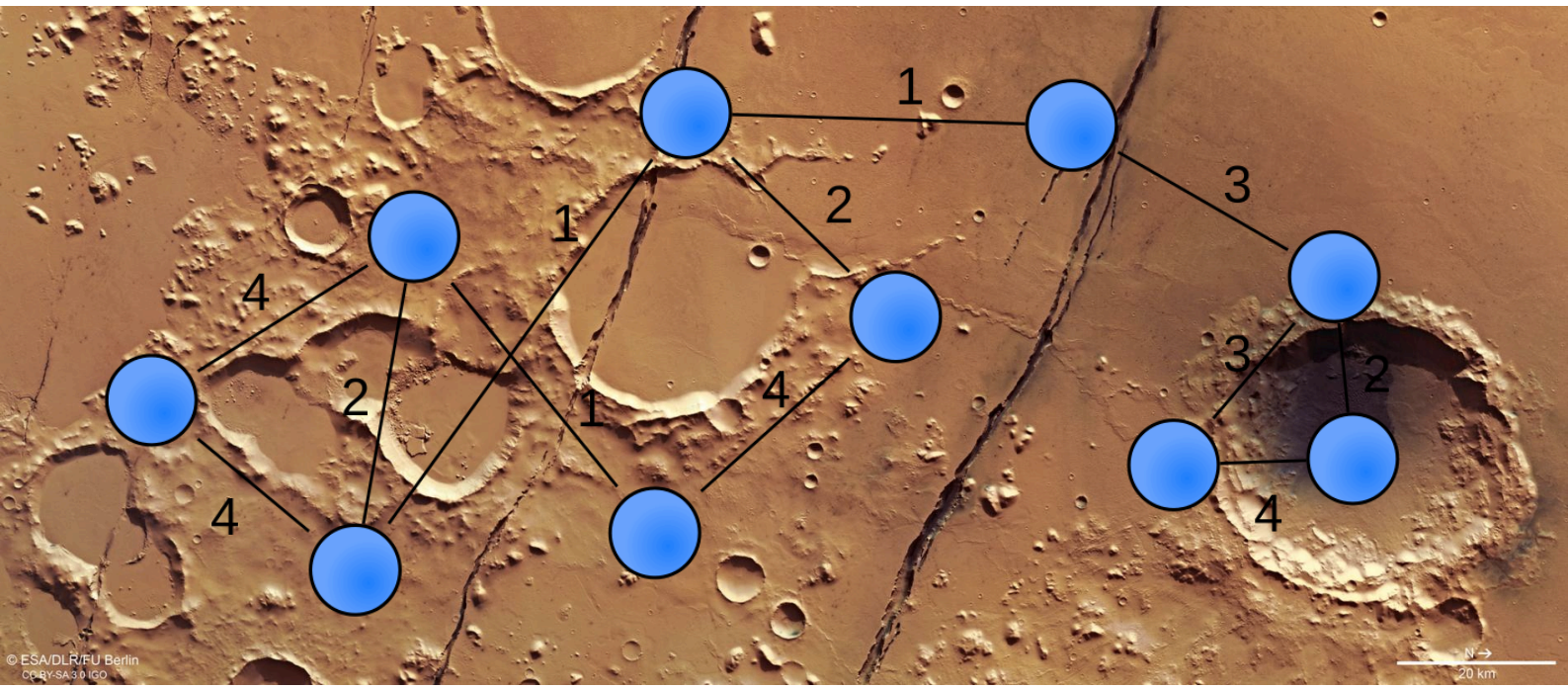
At present, there is one active rover on mars , ‘Curiosity’. Its daily manoeuver is explicitly designed by the NASA’s jet propulsion laboratory.

But it would be more efficient if rather than deciding the daily manoeuver’s each day, we decide the locations on mars that have maximum research potentials and let the rover decide optimum paths in which there is maximum research in certain given time constraints.

Finding these optimum path will not only ensure maximum research in a given amount of time but will also ensure less wheel damage, thereby increasing the rover life.

Mars Simulation-
We use a weighted graph for simulation of the martian surface, in which vertices represent points with research potential and the edges between those vertices represent the paths the rover can take to travel b/w those vertices (edge weights describe time to travel through the edges).

Visual representation of simulation



Technology Used-

1) Recursion and Backtracking
2) Floyd Warshall's algorithm(Optimization)

System Input-

1) Connected Weighted Graph(dynamic ..ie- any number of vertices can be added)
2) Initial vertex
3) Final vertex
4) Time Constraint

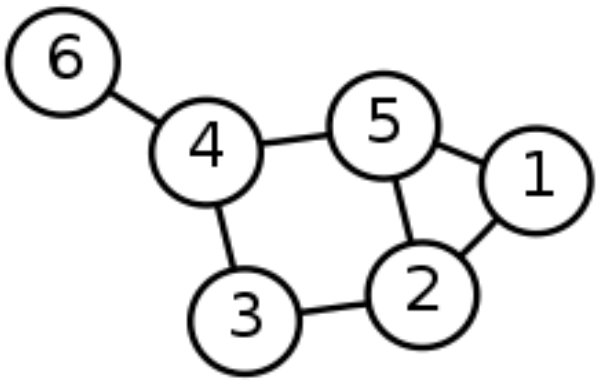
System Output-

Path from initial to final vertex with maximum research in given time constraint / "No path possible".

Preliminaries

Graphs

In Graph Theory, a graph represents a structure that describes the relation between pairs of objects in a set. Each object in the graph is referred to as a vertex and each pair of related objects is referred to as an edge. Graphs are usually represented diagrammatically as circles enclosing objects of the set (vertices). The circles are connected to each other using lines as described by the relation between pairs of objects in the set (edges).



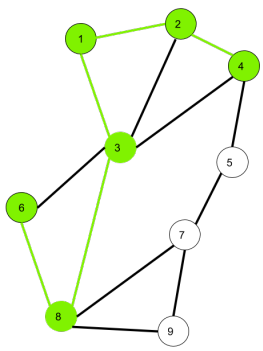
A graph with 6 vertices and 7 edges

Weighted Graph

A weighted graph is a graph where edges are assigned values (Weights). They may be used to describe the various parameters associated with each edge.

Path

A path in a graph is a walk with none of the the vertices and edges getting repeated.



Recursion and Backtracking

Backtracking is a general algorithm to find all possible solutions to computational problems related to constraint satisfaction. It does so by generating candidates to the solutions incrementally, and discards a candidate ("backtracks") as soon as it determines a particular solution cannot possibly satisfy the problem.

Backtracking can be applied effectively for problems which allow the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate many candidates with a single test.

PseudoCode has been discussed in 'Methodology and Main Work'.

Floyd Warshall's Algorithm

The Floyd – Warshall Algorithm is used to find the shortest path between all pairs of vertices in a weighted graph. It does so by comparing each path between each pair of vertices at a time. It has a time complexity of Θ (n3), which isn't very good but once executed, query for any pair of nodes has a time complexity of O(1) (very efficient).

Pseudocode

```
//pseudoCode(0.1)

let dist be a |V| × |V| array of minimum distances initialised to ∞
(infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

$$D_0 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

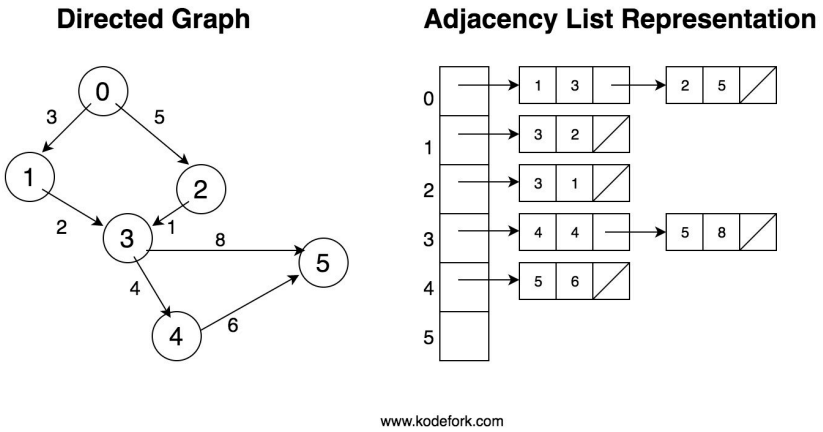
$$D_3 = \begin{pmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{pmatrix}$$

Floyd Warshall algorithm's implementation on adjacency matrix of a weighted graph

In our system, this efficiency has been used in checking for cases where time constraint is less than the minimum time to traverse b/w the input pair of vertices.

Adjacency List

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list contains a number of ordered pairs storing the neighbouring vertex and the weight between them. This is the graph representation used by the recursion and backtracking algorithm in this project.



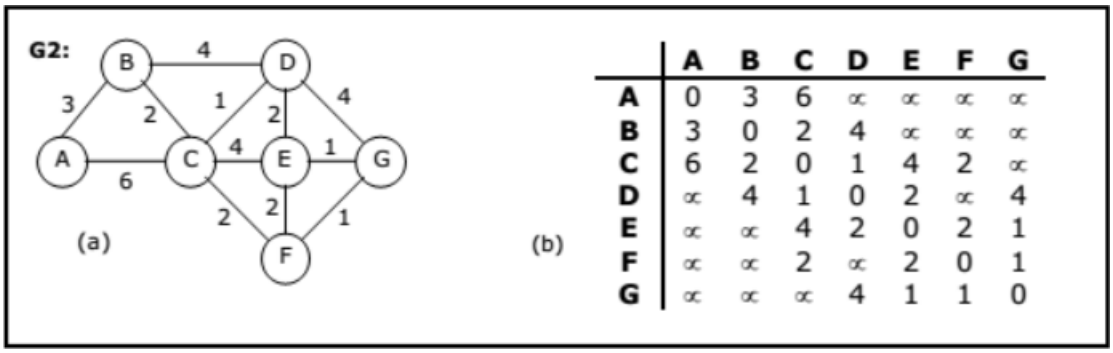
Array

An array is a collection of items stored at contiguous memory locations. In this project research potentials of vertices have been stored in an array.

Adjacency Matrix

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate the weight across the corresponding vertices in the graph.(ie = mat[i][j] stores the wight between vertices i and j.

This is the graph representation being used by the Floyd Warshall algorithm in this project.
Floyd Warshall algorithm will be executed on the adjacency matrix to convert it into a shortest path matrix ('wt').



	A	B	C	D	E	F	G
A	0	3	6	∝	∝	∝	∝
B	3	0	2	4	∝	∝	∝
C	6	2	0	1	4	2	∝
D	∝	4	1	0	2	∝	4
E	∝	∝	4	2	0	2	1
F	∝	∝	2	∝	2	0	1
G	∝	∝	∝	4	1	1	0

Bit-masking

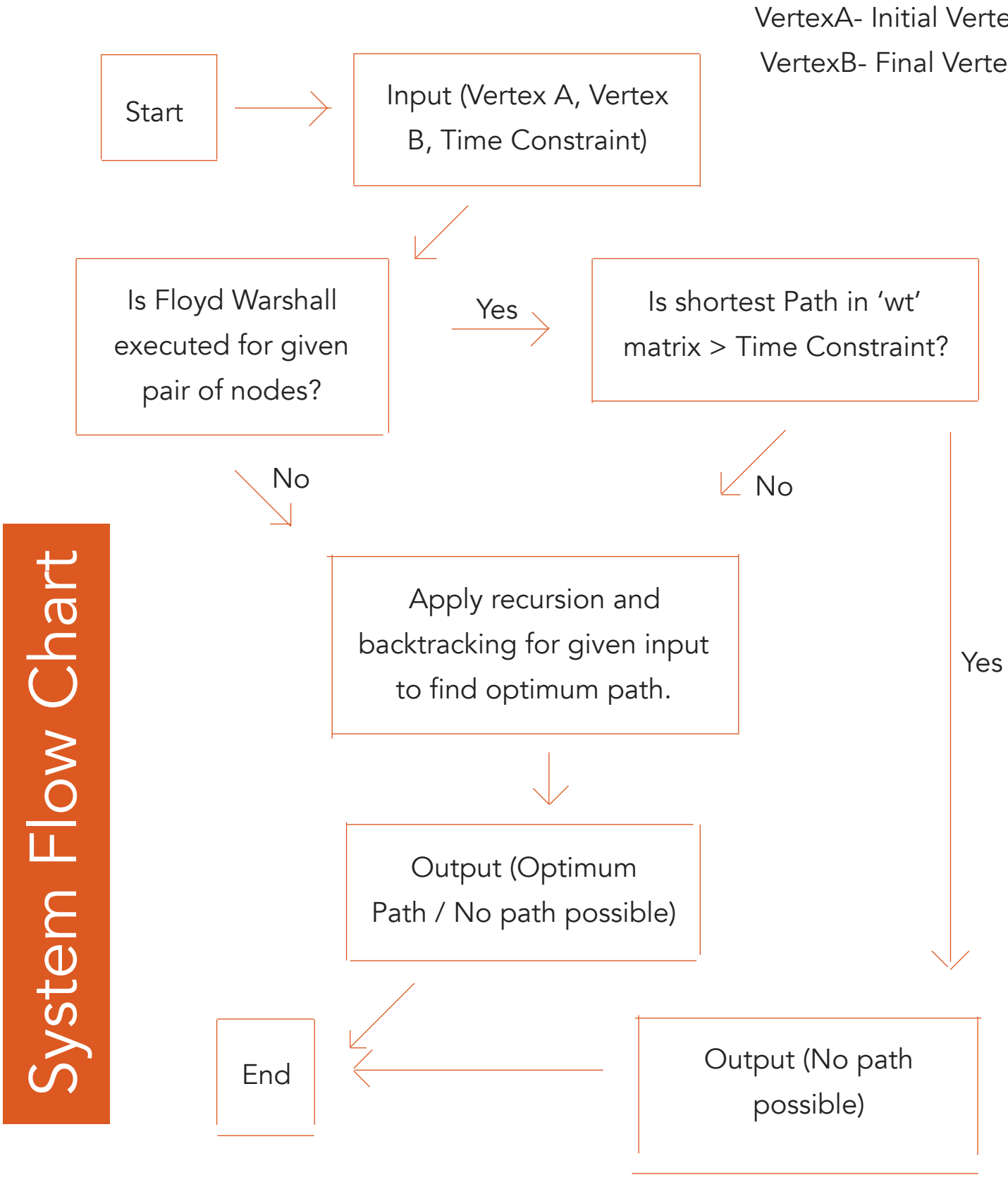
Bit-masking is a way of representing a set and allows O(1) removal and addition of elements into set.
For any integer n, it can be represented as a sequence of bits (0 and 1). Hence, for a given array, we can say the i'th element is in the set if set(i) is 1. Else, the element does not belong to the set.
Hence, integers can be used to represent sets from a given array, and since setting and unsetting bits takes O(1) time, elements can be added or removed in O(1) time.

This has been used to store vertices of path so as to avoid any vertices getting repeated in the optimal path.

Methodology and Main Work

VertexA- Initial Vertex

VertexB- Final Vertex



The above system flow chart tells the overall algorithm.

Our problem was such that we were bound to put weights on vertices(research potentials), and graph of this type hasn't been explored in much detail.
Thus there were not many existing algorithms we could refer to for our problem.
After good research on graphs, we decided to use recursion and backtracking for our purpose.

Recursion and backtracking (usage)-

Assume we want to travel from Vertex 'A' to vertex 'B' in the graph.
To find the optimum path
'A' will ask its adjacent vertices for the path with maximum research from them to 'B' and one that does not exceeds the time constraints.
The path with maximum research among these paths is chosen and returned as the output.
This happens recursively at all the successive vertices, just that in case of intermediate vertices, the output is returned to the parent vertex.
A bit-mask has been used to ensure that none of the vertices get repeated in the process and a state variable 'spareTime' is passed in every successive call to backtrack as soon as spareTime becomes less than 0 (ie- time constraint gets violated).

The following pseudo-code explains in detail how the optimal path has been found-

Pseudo Code

```
//pseudoCode(1.0)

Let optPaths be an array that stores optimal paths to reach final
vertex 'B' for each vertex. ie- optPaths[x] stores <researchDone,
spareTime, ListOfVertices in optimal path form 'x' to 'B'>

Initialise optPath for each vertex to have spareTime = -2, researchDone
= -2 and listOfVertices = [].

Initialise a bit-mask 'mask' as 0.(indicating no vertices visited yet
in current path)

Call traverse function and pass initial vertex 'A', time constraint,
final vertex 'B', 'mask', 'optPaths' as function parameters.
Ie-
Set 'totalResearch' = traverse('A', timeConstraint, 'B', mask,
optPaths)

Let the traverse function be as follows-
Let it return researchDone in traversal from currentVertex to
finalVertex and have function parameters as shown-

researchDone traverse(currentVertex 'curr', spareTime, finalVertex,
mask, *optPaths){

        If spareTime < 0: (ie- time constraint exceeded)
                return -2; (ie - no researchDone)

        If currentVertex == 'B':
                Set optPaths[curr].researchDone <- researchPotential[b]
                Set optPaths[curr].spareTime <- spareTime
                Set optPaths[curr].listOfVertices <- []
                Append 'B' to optPaths[curr].listOfVertices
                return optPaths[curr].researchDone

        Set 'maxResearchDone' <- -1
        Set 'maxSpareTime' <- -1
        Set 'dummy' <- -1

        For every adjacentVertex 'nbr' of 'curr':
                Set 'edge' <- edge weight b/w 'curr' and 'nbr'


                If mask & (1 << curr) == 0: (ie-'nbr' not visited)
                        Call traverse function recursively with arguments
as follows-
                        Set dummy <- traverse(nbr, spareTime - edge,
finalVertex 'B',  mask | (1 << pos), optPaths)
                        If dummy > maxResearchDone or (dummy ==
maxResearchDone and maxSpareTime < optPaths[nbr].spareTime):
                                Set maxResearchDone <- dummy
                                Set maxSpareTime <- optPaths[nbr].sparTime
```

```
                                Set optPaths[curr] <- optPaths[nbr]
                                Append optPaths[curr].listOfVertices with
'curr'
                                Set optPaths[curr].researchDone =
optPaths[curr].researchDone + researchPotential[curr]

        If maxResearchDone == -1:    (ie- no path possible to 'B' from
current vertex in available spareTime)
                return -2
        Else
                return optPaths[curr].researchDone

 }

Since optPaths was passed by address, we can get access to optimum path
from 'A' to 'B' by 'optPaths[A]'.

Print the following:
'Research Done' = optPaths[A].researchDone
'Time taken for traversal' = timeConstraint - optPaths[A].spareTime
'Optimum Path' = optPaths[A].listOfVertices (list should be printed in
reverse order if 'append' was done at end of list.)
```

Floyd Warshall Algorithm(usage)-

A matrix 'wt' has been used for storing the shortest paths found by Floyd Warshall's algorithm. (Shortest path here refer to paths with minimum time of traversal). This 'wt' matrix can be used to check if a path is possible b/w input vertices in given time constraint.
And in cases when it is not, an output telling 'no path possible' can be returned in O(1) time.

Since vertex entry in the graph has been kept dynamic, new vertices are expected to be added on a regular basis.
But these new vertices cannot be added in the 'wt' matrix at the time of vertex addition because time complexity of Floyd Warshall Algorithm is O(n^3); and updating 'wt' matrix every time a vertex is added will result in excess computation time and energy.

Thus the 'wt' matrix should be updated on a monthly/weekly basis so as to get the benefit of Floyd Warshall optimization and avoid its drawback.

The pseudo-code for updating 'wt' is as follows-

```
//PseudoCode(1.1)

Let 'adjList' be the adjacency list of graph.
It stores adjacent vertices of vertex 'A' as follows-
adjList[A] = list<('adjVertex', 'weight' b/w A and adjVertex)>

Delete old 'wt' matrix

Set 'numVertices' <- size of adjList.

Initialise a 2D matrix 'wt' of dimension 'numVertices' * 'numVertices'
Set all values in 'wt' as infinite

for i in range[0, numVertices - 1]:
        for 'nbr' in adjList[i]:
                Set 'j' <- nbr.adjVertex
                Set 'weight' <- nbr.weight
                Set wt[i][j] <- weight

Apply Floyd Warshall Algorithm on 'wt' (pseudoCode 0.1)
```

The pseudo code for the overall system will be as follows-

```
//PseudoCode

Input for operation to perform-
1.addEdge 2.addResearchPotential 3.findPath 4.UpdateMap 0.EXIT
Input code(1/2/3/4/0)

If code == 1 or code == 2:
        update adjacencyList

If code == 3:
        Input initialVertex, finalVertex, timeConstraint.
        If initialVertex, finalVertex < size of 'wt':
              If timeConstraint < wt[initialVertex][finalVertex]:
                      return 'no path possible'
        Else
              Apply recursion and backtracking algorithm(pseudoCode 1.0)

If code == 4:
        Apply update 'wt' algorithm(pseudoCode 1.1)
```

Number of vertices possible in graph:

Any number of vertices can be added as long as there is memory available in the computer. Maximum time constraint allowed is 500 units so as to have good program efficiency.

Time complexity

Time complexity in cases where no path is possible is O(1) because of the Floyd Warshall Algorithm optimisation.

And in cases when path is possible, time complexity largely depends on time constraint. Since we have restricted maximum time constraint to 500 units, the algorithm will work very efficiently saving excess computation time.

Also, in cases when new vertices are added and 'wt' matrix is not updated with these new vertices, the system works fine. (Though it is recommended to update 'wt' matrix on a weekly/ monthly basis)

Source Code

Because of the large size of the source code, it has been uploaded on GitHub.

Link-

https://github.com/its7ARC/martianRoverNavigationSystem/blob/main/navigationSystem.cpp

System Testing

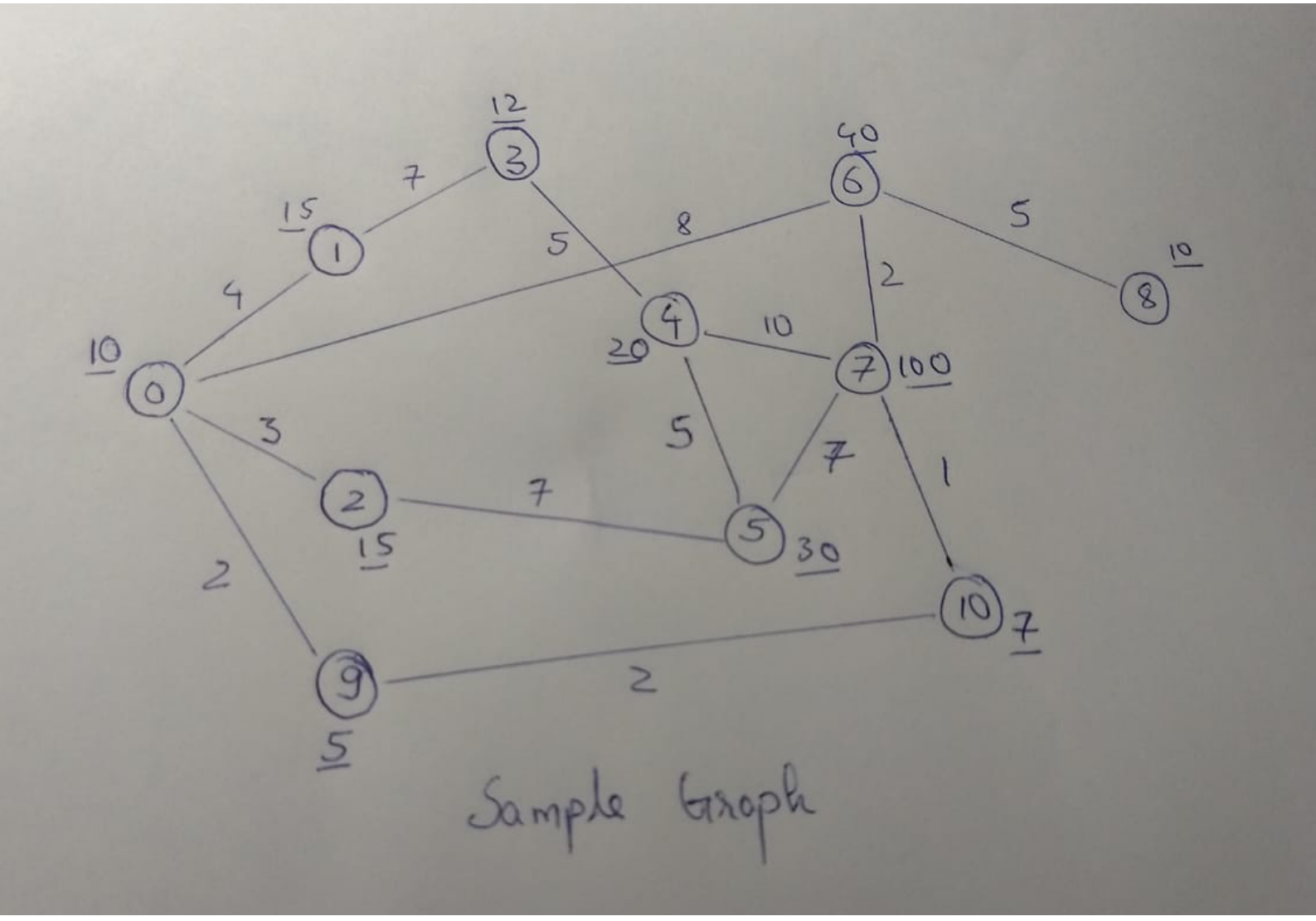
Following is a sample graph for testing.

Circles-> Vertices

Numbers inside circle -> Vertex index

Numbers underlined -> Research Potential of corresponding vertex

Numbers with edges -> Edge weights



Testing on Various Test Cases

Queries -

Input format : operationCode initialVertex finalVertex timeConstraint .. Eg.(3 2 6 5) => findPath, initialVertex = 2, final Vertex = 6, timeConstraint = 5 units.

```
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 5
Enter key of initial, final node and timeConstraint
------------------------------------------------------------
No path possible from Node_2 to Node_6 within 5hrs

------------------------------------------------------------
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 15
Enter key of initial, final node and timeConstraint
------------------------------------------------------------
Optimal Path from Node2 to Node6 within 15hrs:
Research Done = 177units
Time taken = 10hrs
Path: 2,0,9,10,7,6
------------------------------------------------------------
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 25
Enter key of initial, final node and timeConstraint
------------------------------------------------------------
Optimal Path from Node2 to Node6 within 25hrs:
Research Done = 205units
Time taken = 24hrs
Path: 2,5,4,7,6
------------------------------------------------------------
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 50
Enter key of initial, final node and timeConstraint
------------------------------------------------------------
Optimal Path from Node2 to Node6 within 50hrs:
Research Done = 254units
Time taken = 35hrs
Path: 2,5,4,3,1,0,9,10,7,6
------------------------------------------------------------
```

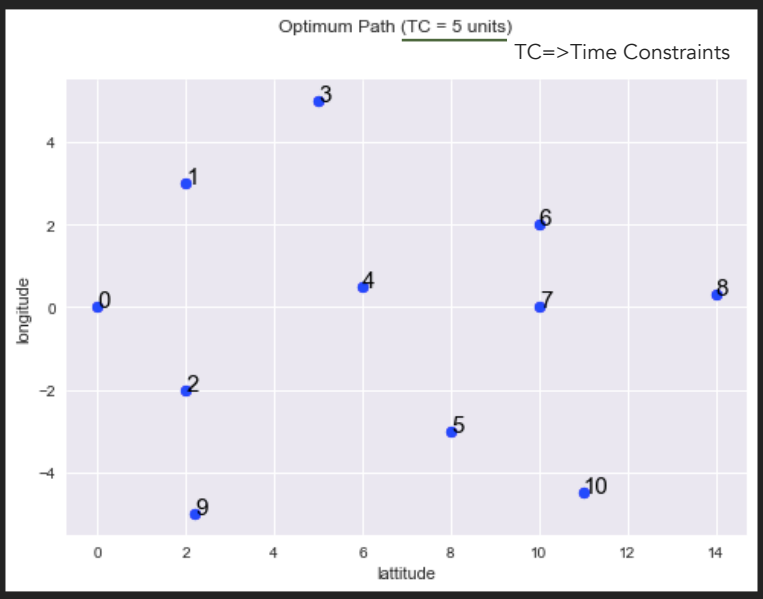
After MapUpdate (ie- ‘wt’ update)

```
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 5
Enter key of initial, final node and timeConstraint
-----------------------------------------------------------
No path possible from Node_2 to Node_6 within 5hrs
Min time to travel from Node2to Node6 = 10hrs( according to last mapUpdate)
-----------------------------------------------------------
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 15
Enter key of initial, final node and timeConstraint
-----------------------------------------------------------
Optimal Path from Node2 to Node6 within 15hrs:
Research Done = 177units
Time taken = 10hrs
Path: 2,0,9,10,7,6
-----------------------------------------------------------
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 25
Enter key of initial, final node and timeConstraint
-----------------------------------------------------------
Optimal Path from Node2 to Node6 within 25hrs:
Research Done = 205units
Time taken = 24hrs
Path: 2,5,4,7,6
-----------------------------------------------------------
Enter operation to perform
1.AddEdge   2.AddResearchPotential  3.FindPath  4.UpdateMap 0.EXIT
3 2 6 50
Enter key of initial, final node and timeConstraint
-----------------------------------------------------------
Optimal Path from Node2 to Node6 within 50hrs:
Research Done = 254units
Time taken = 35hrs
Path: 2,5,4,3,1,0,9,10,7,6
-----------------------------------------------------------
```

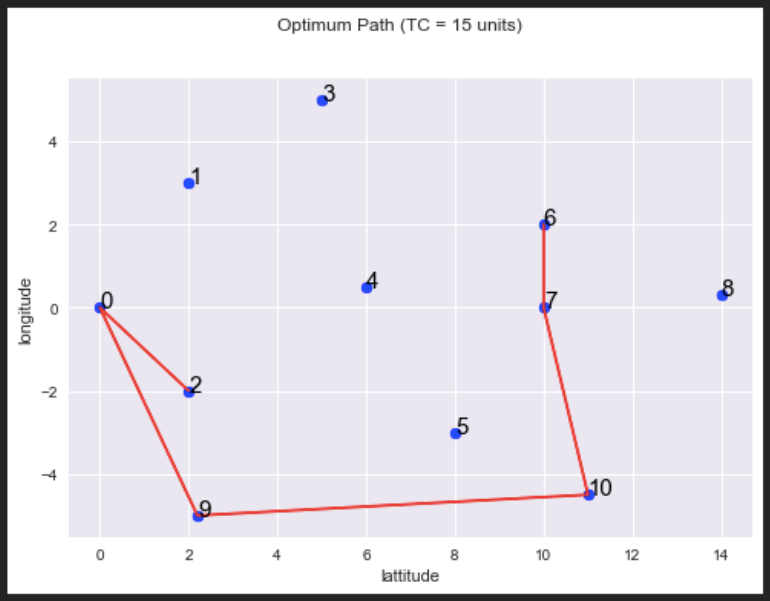
Visual outputs for above queries

Queries before Floyd Warshall execution-

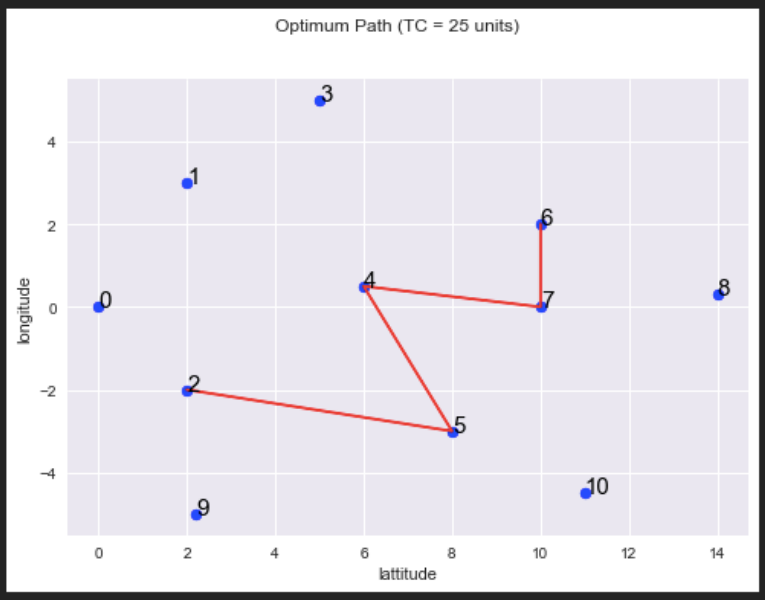
```
Initial Node = 2 | Final Node = 6
Research Done = 0 units | Time taken = 0 units
```



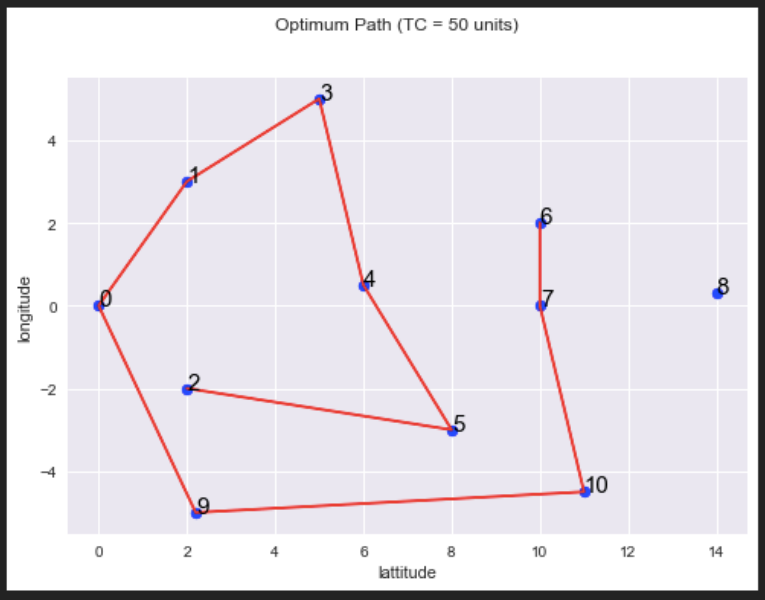

```
Initial Node = 2 | Final Node = 6
Research Done = 177 units | Time taken = 10 units
```




```
Initial Node = 2 | Final Node = 6
Research Done = 205 units | Time taken = 24 units
```

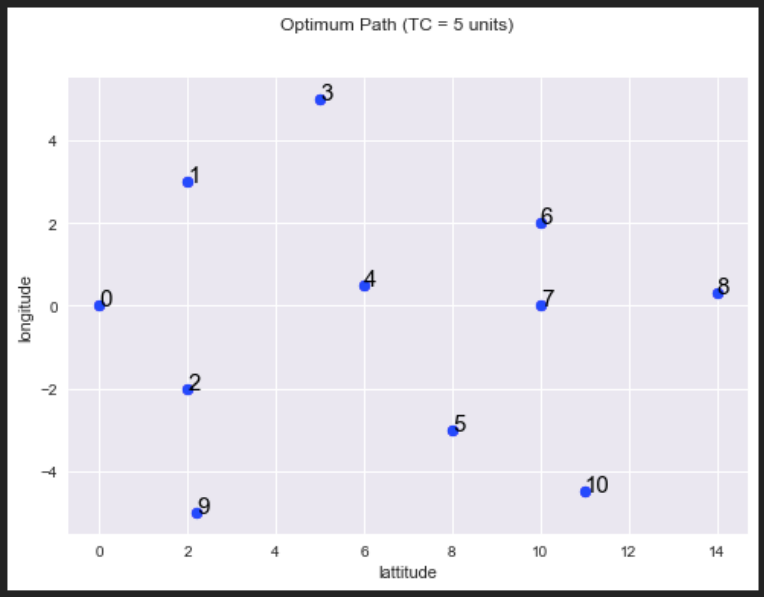



```
Initial Node = 2 | Final Node = 6
Research Done = 254 units | Time taken = 35 units
```

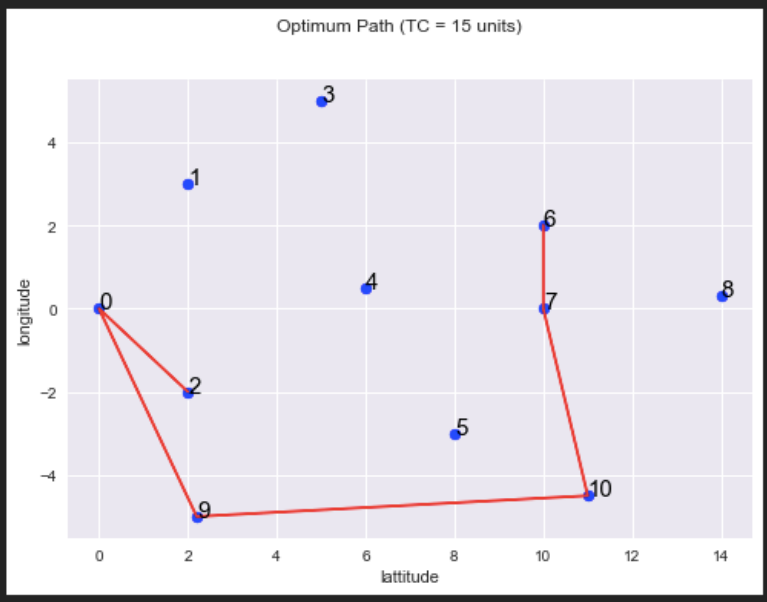


Queries after Floyd Warshall execution ('wt' updated)-

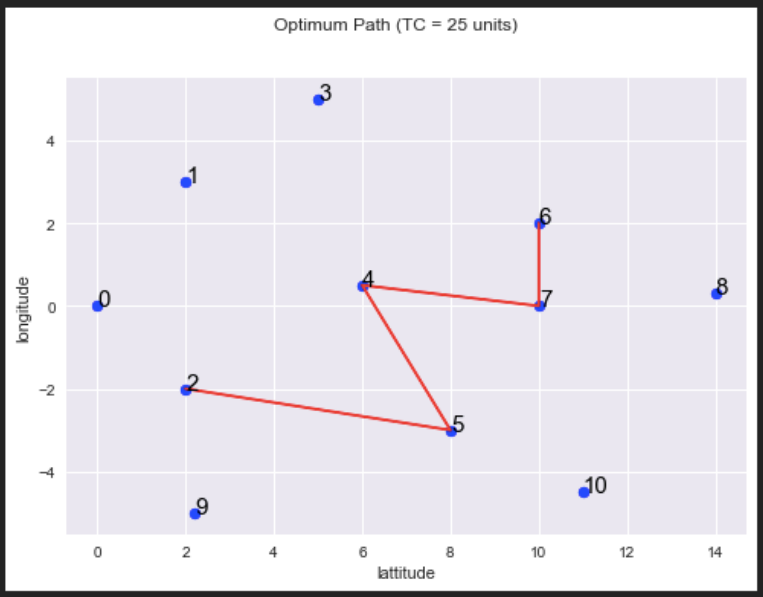
```
Initial Node = 2 | Final Node = 6
Research Done = 0 units | Time taken = 0 units
```



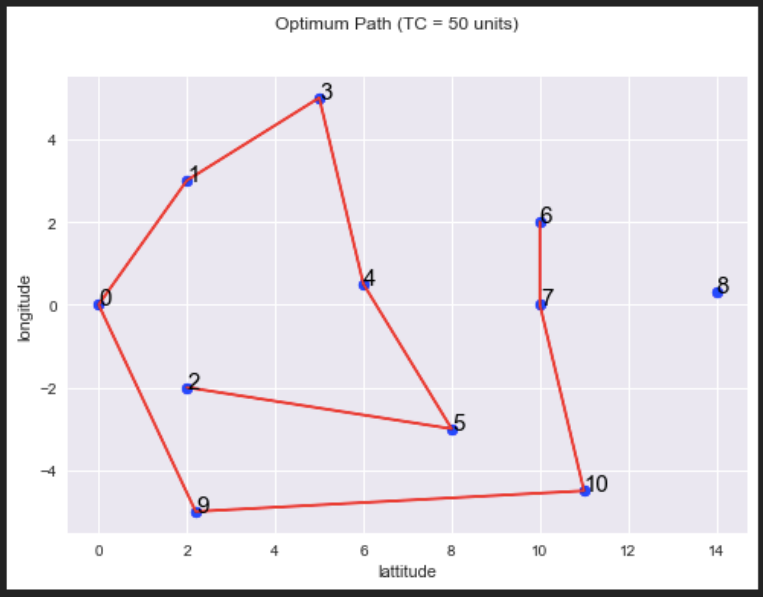

```
Initial Node = 2 | Final Node = 6
Research Done = 177 units | Time taken = 10 units
```




```
Initial Node = 2 | Final Node = 6
Research Done = 205 units | Time taken = 24 units
```




```
Initial Node = 2 | Final Node = 6
Research Done = 254 units | Time taken = 35 units
```



(These outputs verify with intuition that the rover will try to travel through more vertices when it has more time available.)

Result and Conclusion

The system yields the best results in all the test cases. Also it is very robust and can work with or without the Floyd Warshall's optimisation.

Time Complexity is also extremely low for a problem of this kind. Overall, it can be said that a novel approach has been developed for dealing with graphs with weight on vertices.

We conclude that this project is ready for deployment and will contribute in a very positive way in research on the red planet.

Resources Used

1. Standard Template Library
2. Numpy Library
3. Matplotlib
4. Jupyter-Notebook
5. Xcode IDE

Bibliography

1. Wikipedia - Floyd-Warshall Algorithm - [9/10/20] - https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
2. Geeksforgeeks - Floyd-Warshall Algorithm - [9/10/20] - https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/
3. Wikipedia - Graphs - [10/10/20] - https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
4. Ayush Chaturvedi (Geeksforgeeks) - Backtracking - [11/10/20] - https://www.geeksforgeeks.org/backtracking-introduction/
5. Wikipedia - Backtracking - [11/10/20] - https://en.wikipedia.org/wiki/Backtracking
6. Wikipedia - Adjacency matrix - [15/10/20].- https://en.wikipedia.org/wiki/Adjacency_matrix
7. Wikipedia - Adjacency List - [15/10/20] - https://en.wikipedia.org/wiki/Adjacency_list
8. Wikipedia - Bitmasking.- [18/10/20] - https://en.wikipedia.org/wiki/Mask_(computing)