AI PROJECT

BI DIRECTIONAL SEARCH IMPLEMENTATION

Team Members:

Nidhinandana Salian 140905594

Shubhang Periwal 140905552

Rajat Kumar Singh 140905538

ACKNOWLEDGEMENT

We would like to express the deepest appreciation to our Artificial Intelligence teacher, Mrs Jimcymol James, she continually and convincingly conveyed a spirit of adventure in regard to research and practical implementation of learnt knowledge, and an excitement in regard to teaching. Without her guidance and persistent help this project would not have been possible.

We would also like to thank our Head of the Department, Dr. Ashalatha Nayak, for providing us with this opportunity to apply what he have learnt and enjoy a practical experience of how artificial intelligence algorithms are run in legitimate applications to provide efficient, optimal solutions real world problems.

In addition, we would like to express our immense gratitude to our Director Dr. G. K. Prabhu, whose enthusiasm for projects and research work has proven to be a lasting inspiration.

ABSTRACT

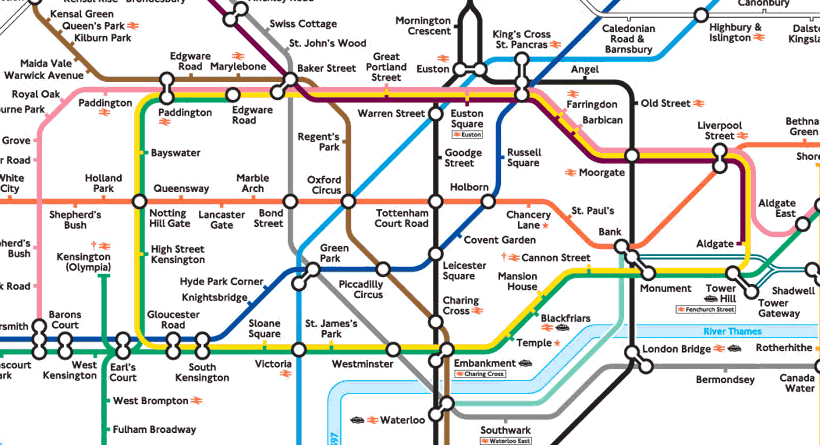
Bidirectional search involves running two simultaneous searches, one from the initial state and the other from the goal state, both of which terminate when they coincide at some node in between, producing the shortest path.

We are implementing Bi-directional search to find the shortest path between any two stations on the London Underground Network, where we are assuming equal distance between any two underground stations. Input would be 2 stations, i.e. source and destination. We would use bfs (breadth first search) strategy for both source and destination simultaneously and check for any simultaneously occurring common node. If the searches intersect at a node, we print out that path. The path would be the most optimal one considering the number of stations we are going to travel.

INTRODUCTION

The London Underground (also known simply as the Underground, or by its nickname the Tube) is a public rapid transit system serving Greater London and some parts of the adjacent counties of Buckinghamshire, Essex and Hertfordshire in the United Kingdom.

The network has expanded to 11 lines, and in 2015–16 carried 1.34 billion passengers,[3] making it the world's 11th busiest metro system. The 11 lines collectively handle approximately 4.8 million passengers a day.



Our project intends to identify the most optimal path between any two stations inputted by the user, assuming the path with the least number of stations(nodes) to be the most optimal. To achieve this objective, we have applied bidirectional search from both the source station and destination station simultaneously to find the best route along various network lines. We have created our project in C++ programming language.

PROBLEM STATEMENT

Given two stations, i.e., the source and the destination, find the path with least cost, assuming the cost of travelling between any two stations is equal, by implementing bidirectional breadth first search algorithm. Whenever we reach the destination (or a path is determined), we print the entire path that needs to be travelled by the user to go from one station to another. Most optimal path between two stations such that the user has to travel the least number of stations (nodes).

LITERATURE SURVEY

Most bi‐directional search algorithms contain two sets of OPEN and CLOSED nodes: OPENF and CLOSEDF for search in the Forward direction, and OPENB and CLOSEDB for search in thenBackward direction. The algorithm starts with the Forward direction, putting s in CLOSEDF, its successors in OPENF and computing their heuristic values and evaluation functions. After the first Forward iteration, it does the first Backward iteration, using t, OPENB and CLOSEDB , and proceeding in a reverse‐A\* like manner, generating parent nodes instead of child nodes from the graph. In the iterations that follow, the search alternates between Forward and Backward directions; however the alternation is not strict. A common strategy employed by most bidirectional search algorithms is to search in the direction which has the smaller size of OPEN;

thus if the current search direction is Forward and |OPENF| < |OPENB| at the end of this iteration, then the next search direction will still be Forward – it will not reverse. Similar rule holds for the Backward pass. Ensuring that we always search from the direction of smaller OPEN seems quite intuitive and may help reduce the width of the search graph.  There are mainly two types of bidirectional searches: where the evaluation function measures the distance from the selected node to a goal node (known as front‐to‐end

evaluation), and where the distance is measured from the selected node to any node in the

opposite OPEN (known as front‐to‐front evaluation.)

IMPLEMENTATION

**Design:**

PEAS description of the Task Environment considered:

1. Performance measure:

* Most optimal path between two stations such that the user has to travel the least number of stations (nodes)
* Memory Space required to traverse the nodes.

1. Environment:

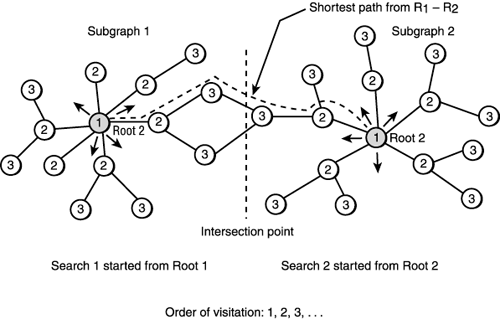
* A map of the London Underground   List of stations in each line
* Intersections of london underground(at intersection and end points)

1. Actuators:

* The agent can move to any one of its immediate neighbors that hasn't already been visited
* If it comes to a circuit end then it stops at that node

1. Sensors:

* Check whether the station found in bfs of source is in the visited list of destination or vice versa
* Whenever we reach the destination (or a path is determined), we print the entire path that needs to be traavelled by the user to go from one station to another.



**Code Snippets:**

**Our input is in a form of a file which contains a list of all stations separated by lines. First the aim was to create nodes of each station as shown in the structure below.**

typedef struct station

{

char name[50];

int no\_of\_neighbors;

struct station \*neighbors;

struct station \*s\_parent;

struct station \*d\_parent;

int count;

\*}STATION;

**Number of neighbors represent the number of neighbors that a node has. Here the neighbors pointer points to all the neighbors that a given station has. Parent pointer points to where it came from for a given path i.e. if we have to go from Oxford Circus to Charing Cross with Piccadilly Circus in between. The Piccadilly Circus's parent would contain Oxford Street as its parent in order to retrace it to display its path in the end.**

**We are implementing BFS from both the directions i.e. the source as well as the destination and then displaying the intersection point as well as the entire path that the person needs to travel. The path would contain minimum number of stations (Bidirectional is optimal in case of uniform costs).**

void bfs(STATION \*node)

{

int flag=0;

for(int i=0;i<node->no\_of\_neighbors;i++)

{

for(int j=0;j<v\_d;j++)

{

if(!strcmp(node->neighbors[i].name,visited\_d[j]->name))

{

flag=1;

}

}

if(!flag)

{

visited\_d[v\_d] = &(node->neighbors[i]);

v\_d++;

flag=0;

//cout<< node->neighbors[i].name<<" ";

node->neighbors[i].d\_parent = visited\_d[i];

}

flag=0;

}

}

**Now the next part is to implement BFS on both the nodes one by one and then adding them to visited array if already not present in it. We are maintaining 2 visited arrays one for source side and the other for destination side. If any of the elements in the arrays match then we print that node as well as the path in which it lies.**

while(!done)

{

int temp = v\_s;

for(i=top1;i<temp;i++)

{

bfs1(visited\_s[i]);

top1++;

}

cout<<endl<<"source over "<<endl;

temp = v\_d;

for(i=top2;i<temp;i++)

{

bfs2(visited\_d[i]);

top2++;

}

cout<<endl<<"destination over "<<endl;

for(i=0;i<v\_s;i++)

{

for(j=0;j<v\_d;j++)

{

if(!strcmp(visited\_s[i]->name,visited\_d[j]->name))

{

done = 1;

cout<<endl;

cout<<"the intersection meet is ";

cout<<endl;

cout<<visited\_d[j]->name;

}

}

}

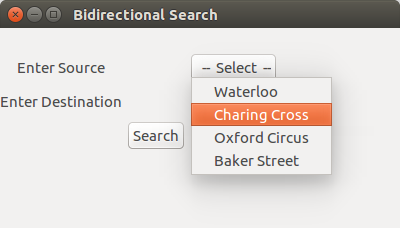
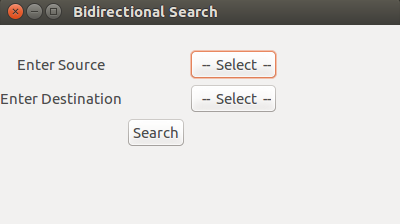
for(i=0;i<v\_s;i++)

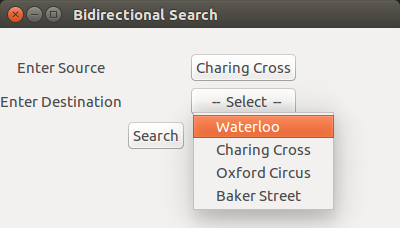
if(!strcmp(destination,visited\_s[i]->name))

cout<<"Found";

}

**Screenshots(UI):**



****

CONCLUSION

Bidirectional search seems attractive for its *O*(bd/2)performance, but things are not that easy, especially the implementation part.

Since one of the crucial requirements for bidirectional search to be applicable to a problem is that its states should be reversible, the London Underground network, or any transportation network is a viable option to consider its implementation upon. We can compute the successors at every station and easily detect an intersection / lack of one therof, and terminate the program.

ADVANTAGES

The merit of bidirectional search is its speed. Sum of the time taken by two searches (forward and backward) is much less than the O(bd) complexity.

It requires less memory.

DISADVANTAGES

Implementation of bidirectional search algorithm is difficult because additional logic must be included to decide which search tree to extend at each step.

One should have known the goal state in advance.

The algorithm must be too efficient to find the intersection of the two search trees.

It is not always possible to search backward through possible states.

REFERENCES

Sturtevant, N.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In IJCAI- 09, 609–614.

Pohl, I. 1969. Bi-directional and heuristic search in path problems.Technical Report 104, Stanford Linear Accelerator Center