UNIVERSITY OF SURREY

# AI and AI Programming

(EEEM005)

2019/2020

Assignment 1

Student ID: 6608512

## Table of Contents

## 1. Executive Summary

This assignment is about programming in Prolog and to implement AI search schemes for the London Underground, other than the default Prolog search search algorithms.

The objective of this assignment is to understand how Prolog program execution works and to give me practice in thinking declaratively and writing program that way.

The goal of this Prolog coding is to provide us the route of the London Underground. That means, it will give us the direction to the destination you want (from where you are).

Input: Starting Point, Ending Point
Output: Route

By definition from Wikipedia, In computer science, a **search algorithm** is any algorithm which solves the search problem, namely, to retrieve information stored within some data structure, or calculated in the search space of a problem domain, either with discrete or continuous values.[2]

Basically, there are 2 types of the AI search algorithms, uninformed search and informed search. Uninformed search is a search algorithm that has no additional information on the goal node and so it meets the goal by trying in every direction as there is no information about the goal provided. Example: depth first search, breadth first search. However, informed search has information about the goal state, and this will lead us to get the search to meet the goal efficiently. Example: best first search.

In this assignment, there were two (2) search algorithms performed. Depth first search and best first search.

As a Prolog default search algorithm, depth-first search (DFS), which is sometimes not convenient: DFS doesn't always find the solution for all problems, and when it does, it might not find the *optimal* solution. [1]

Among all the AI search algorithms, best first search is the ideal solution for the London underground system where it will get us the shortest way with minimum travelling time (under assumption that shorter way will usually require shorter travelling time) from station A to station B. In this best first search, it uses a heuristics function, that is to calculate the distance between the stations and to estimate how close the current state is to our ending point (our destination).

As a conclusion, depth first search will give you a complete search if there is a solution. It will not always give you optimal solution, and therefore its cost might be high, like the time and money spent for the traveller. Example, it may provide you a big round to your destination. However, as compared with best first search, best first search will always give us optimal solution, that is the shortest path to our destination, and its time complexity is lesser than depth first search. Hence it is the best solution because of its efficiency for this travelling assignment.

After this assignment, I have better understanding of those search algorithms and hands-on on this declarative programming language, Prolog.

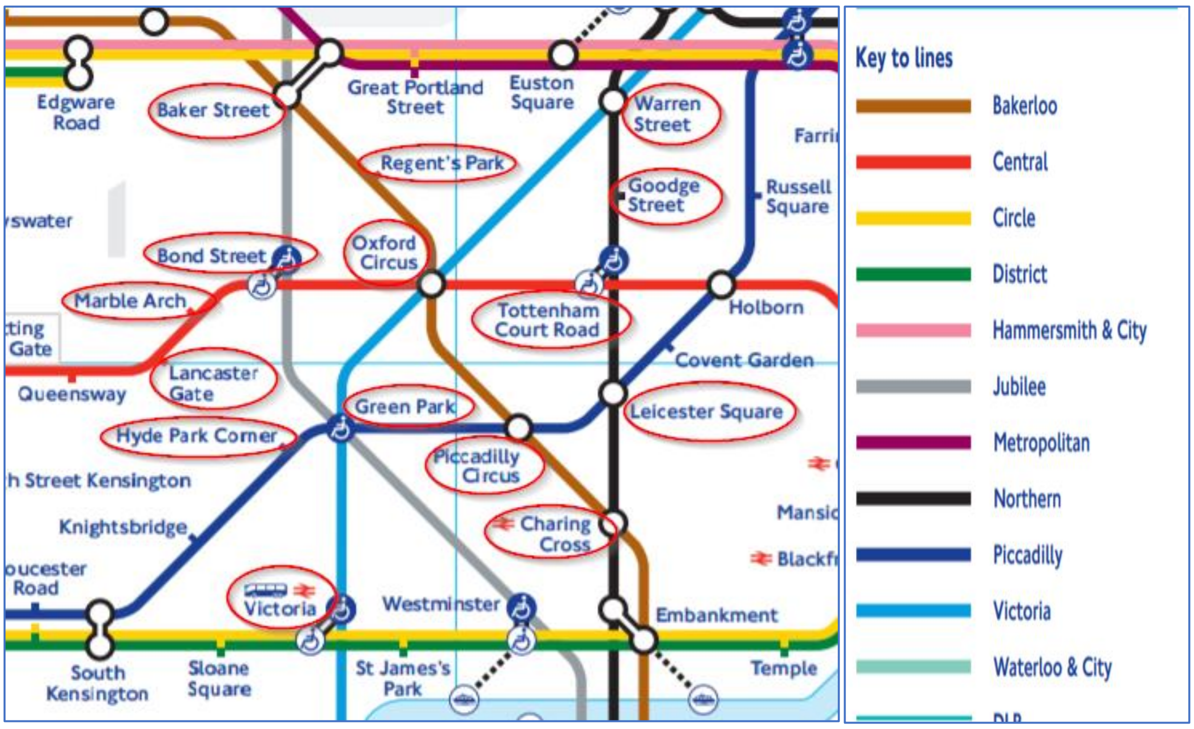
## 2. Introduction

In this hustle and bustle modern city, time is precious for everybody. Hence, most of the people is seeking for the fastest way to reach their destination in London.

This assignment is to design a route planner in London by using the London Tube System.

To get a large enough for interesting queries (e.g. getting station A to B, with minimum time or fewest changes - the times can be estimated or use the TFL journey planner for accurate times), there are 15 stations of the London Underground are constructed for this assignment.

See the stations and its map as below:-



[source: http://content.tfl.gov.uk/large-print-tube-map.pdf]

## 3. Prolog Code/ Documentation

Prolog is a logical and a declarative programming language.

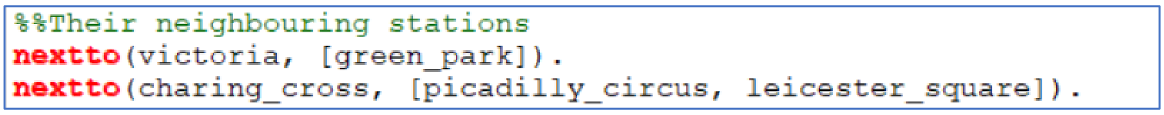
Firstly, a knowledge database of the London underground has to be created. In this assignment, there is consists of 15 stations in London zone 1. These 15 stations are connected by five (5) lines (Bakerloo, Central, Piccadilly, Victoria and Northern lines) of London Underground.

### 3.1 Depth First Search

**Depth-first search** (**DFS**) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. [3]

**Fact: Nextto**

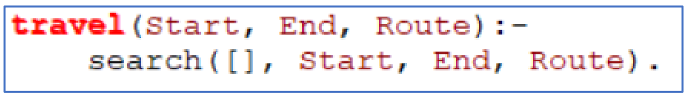
In predicate nextto, we are building the map of stations by connecting them up. The first argument is the station name, and the second argument is the list of its neighbouring stations.



```
%%Their neighbouring stations
nextto(victoria, [green_park]).
nextto(charing_cross, [picadilly_circus, leicester_square]).
```

**Rule: Travel (Depth First Search)**

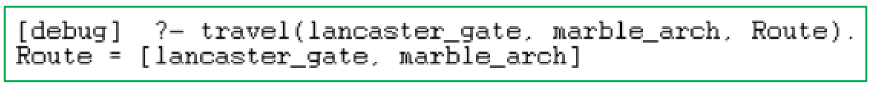
Travel is created for the query and the result Route will be returned.



```
travel(Start, End, Route):-
    search([], Start, End, Route).
```

In order to ask the query, user has to key in his starting point and ending point and followed by variable. That is, variable name has to start with capital letter or underscore (_).

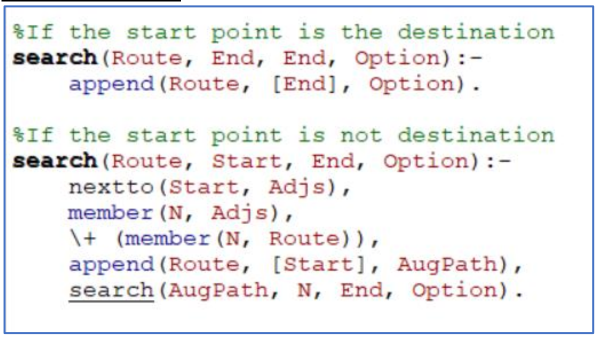
For example:



```
[debug]  ?- travel(lancaster_gate, marble_arch, Route).
Route = [lancaster_gate, marble_arch]
```

Route is returned, that is Route = [lancaster_gate, marble_arch].

**Rule: Search**

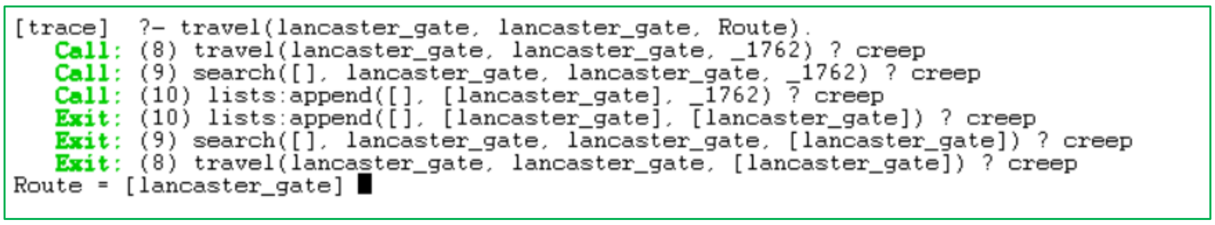


```
%If the start point is the destination
search(Route, End, End, Option):-
    append(Route, [End], Option).

%If the start point is not destination
search(Route, Start, End, Option):-
    nextto(Start, Adjs),
    member(N, Adjs),
    \+ (member(N, Route)),
    append(Route, [Start], AugPath),
    search(AugPath, N, End, Option).
```

The first search is where the starting point is the destination, as such it will end the search since the answer is found, that is the starting point is the destination.

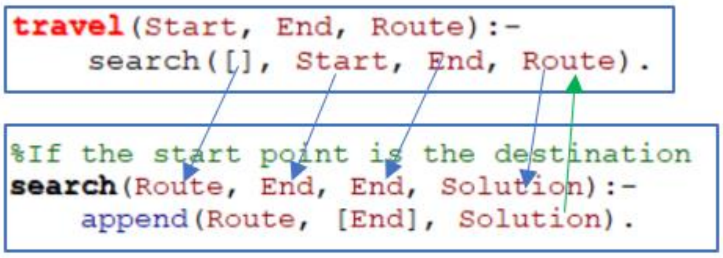
Example:



```
[trace]  ?- travel(lancaster_gate, lancaster_gate, Route).
   Call: (8) travel(lancaster_gate, lancaster_gate, _1762) ? creep
   Call: (9) search([], lancaster_gate, lancaster_gate, _1762) ? creep
   Call: (10) lists:append([], [lancaster_gate], _1762) ? creep
   Exit: (10) lists:append([], [lancaster_gate], [lancaster_gate]) ? creep
   Exit: (9) search([], lancaster_gate, lancaster_gate, [lancaster_gate]) ? creep
   Exit: (8) travel(lancaster_gate, lancaster_gate, [lancaster_gate]) ? creep
Route = [lancaster_gate]
```

? travel(lancaster_gate, lancaster_gate, Route).

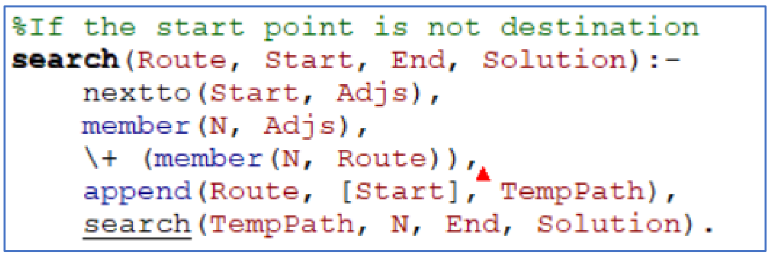
travel function is called with parameters (Start = lancaster_gate, End = lancaster_gate, and Route is a variable, it is a solution we would like to know). Route will be returned if search is fulfilled.
Hence the first search is called with mapping parameters (Route, End = lancaster_gate, End = Lancaster_gate, Solution), it is true where the starting point is same as ending point. See the mapping method shown as below. Next, the Route is an empty list and by calling append, it appends the ending point to the empty list to Solution. Lastly, the Solution is mapped to the Route (see green arrow).



```
travel(Start, End, Route):-
    search([], Start, End, Route).
```

```
%If the start point is the destination
search(Route, End, End, Solution):-
    append(Route, [End], Solution).
```

The second search is where the starting point is not the destination, as such we have to continue searching. Hence, it's a recursive predicate, search function is called until the first search function is fulfilled.

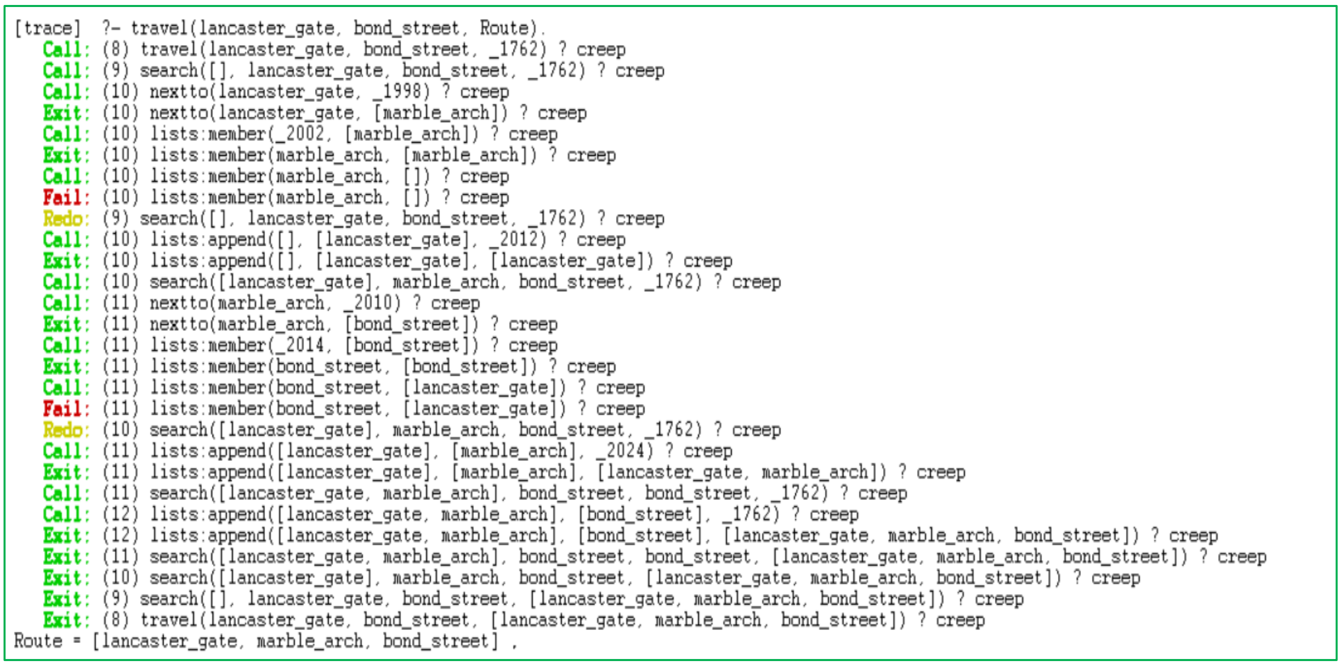


```
%If the start point is not destination
search(Route, Start, End, Solution):-
    nextto(Start, Adjs),
    member(N, Adjs),
    \+ (member(N, Route)),
    append(Route, [Start], TempPath),
    search(TempPath, N, End, Solution).
```

For example (where the starting point is not ending point): ? travel(Lancaster_gate, bond_street, Route). First search function is not fulfilled, hence it will go to second search function. In order to fulfil the search function, we have to fulfil all the following functions.

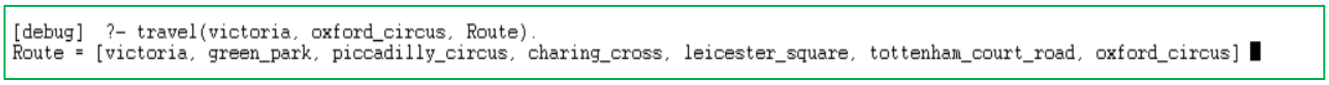
nextto(Lancaster_gate, marble_arch), hence Adjs is Marble Arch.
member(N, marble_arch), hence N is marble_arch.
\+ (member(marble_arch, Route)). Route is an empty list, hence marble arch is not in the empty list. The fact is true.
append([],[lancaster_gate], TempPath) -> append lancaster_gate to TempPath.

search(lancaster_gate, marble_arch, bond_street, lancaster_gate). Continue with a new search of the next station with new parameters this time. That is, Route is TempPath (i.e.: lancaster_gate), Start is N (i.e.: marble_arch), End is remained (i.e: bond_street). Solution is an unknown variable to be returned when a completed route is formed.



```
[trace]  ?- travel(lancaster_gate, bond_street, Route).
   Call: (8) travel(lancaster_gate, bond_street, _1762) ? creep
   Call: (9) search([], lancaster_gate, bond_street, _1762) ? creep
   Call: (10) nextto(lancaster_gate, _1998) ? creep
   Exit: (10) nextto(lancaster_gate, [marble_arch]) ? creep
   Call: (10) lists:member(_2002, [marble_arch]) ? creep
   Exit: (10) lists:member(marble_arch, [marble_arch]) ? creep
   Call: (10) lists:member(marble_arch, []) ? creep
   Fail: (10) lists:member(marble_arch, []) ? creep
   Redo: (9) search([], lancaster_gate, bond_street, _1762) ? creep
   Call: (10) lists:append([], [lancaster_gate], _2012) ? creep
   Exit: (10) lists:append([], [lancaster_gate], [lancaster_gate]) ? creep
   Call: (10) search([lancaster_gate], marble_arch, bond_street, _1762) ? creep
   Call: (11) nextto(marble_arch, _2010) ? creep
   Exit: (11) nextto(marble_arch, [bond_street]) ? creep
   Call: (11) lists:member(_2014, [bond_street]) ? creep
   Exit: (11) lists:member(bond_street, [bond_street]) ? creep
   Call: (11) lists:member(bond_street, [lancaster_gate]) ? creep
   Fail: (11) lists:member(bond_street, [lancaster_gate]) ? creep
   Redo: (10) search([lancaster_gate], marble_arch, bond_street, _1762) ? creep
   Call: (11) lists:append([lancaster_gate], [marble_arch], _2024) ? creep
   Exit: (11) lists:append([lancaster_gate], [marble_arch], [lancaster_gate, marble_arch]) ? creep
   Call: (11) search([lancaster_gate, marble_arch], bond_street, bond_street, _1762) ? creep
   Call: (12) lists:append([lancaster_gate, marble_arch], [bond_street], _1762) ? creep
   Exit: (12) lists:append([lancaster_gate, marble_arch], [bond_street], [lancaster_gate, marble_arch, bond_street]) ? creep
   Exit: (11) search([lancaster_gate, marble_arch], bond_street, bond_street, [lancaster_gate, marble_arch, bond_street]) ? creep
   Exit: (10) search([lancaster_gate], marble_arch, bond_street, [lancaster_gate, marble_arch, bond_street]) ? creep
   Exit: (9) search([], lancaster_gate, bond_street, [lancaster_gate, marble_arch, bond_street]) ? creep
   Exit: (8) travel(lancaster_gate, bond_street, [lancaster_gate, marble_arch, bond_street]) ? creep
Route = [lancaster_gate, marble_arch, bond_street] .
```

However, this depth first search is not an informed search as  it doesn't have any additional information about the search so it won't give us the optimal answer (shortest path). For example, if we would like to travel from Victoria to Oxford Circus.



```
[debug]  ?- travel(victoria, oxford_circus, Route).
Route = [victoria, green_park, piccadilly_circus, charing_cross, leicester_square, tottenham_court_road, oxford_circus] 
```

By going through depth first search algorithm, it will not necessary give us the shortest path, but a long way route may be returned.

Depth First Search: [Victoria, Green Park, Piccadilly Circus, Charing Cross, Leicester Square, Tottenham Court Road, Oxford Circus].

However, the optimal solution should be the shortest one.
Shortest Path: [Victoria, Green Park, Oxford Circus].

## 3.2 Best First Search

Best first search is an informed search. It is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.[4] In this case, we are using distance as a heuristic function.

Here is the estimated distance between the stations meant for heuristic function.

| | victoria | charing _cross | piccadilly _circus | green _park | hyde_park _corner | leicester _square | tottenha m_court_ road | oxford_ circus | bond_ street | lancaster _gate | marble _arch | goodge _street | warren_ street | regents _park | baker_ street |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| victoria | 0 | 4 | 4 | 6 | 5 | 10 | 14 | 13 | 17 | 16 | 18 | 23 | 25 | 18 | 20 |
| charing_cross | 4 | 0 | 2 | 7 | 8 | 5 | 7 | 10 | 13 | 12 | 13 | 10 | 12 | 15 | 20 |
| piccadilly_circus | 4 | 2 | 0 | 2 | 5 | 3 | 4 | 5 | 6 | 6 | 8 | 8 | 12 | 10 | 15 |
| green_park | 6 | 7 | 2 | 0 | 2 | 8 | 6 | 5 | 6 | 8 | 8 | 10 | 12 | 8 | 13 |
| hyde_park_corner | 5 | 8 | 5 | 2 | 0 | 8 | 10 | 6 | 5 | 7 | 6 | 12 | 14 | 8 | 12 |
| leicester_square | 10 | 5 | 3 | 8 | 8 | 0 | 4 | 10 | 12 | 8 | 9 | 6 | 8 | 12 | 20 |
| tottenham_court_road | 14 | 7 | 4 | 6 | 10 | 4 | 0 | 4 | 8 | 12 | 10 | 4 | 6 | 7 | 12 |
| oxford_circus | 13 | 10 | 5 | 5 | 6 | 10 | 4 | 0 | 4 | 7 | 6 | 6 | 8 | 3 | 8 |
| bond_street | 17 | 13 | 6 | 6 | 5 | 12 | 8 | 4 | 0 | 5 | 3 | 8 | 12 | 6 | 5 |
| lancaster_gate | 16 | 12 | 6 | 8 | 7 | 8 | 12 | 7 | 5 | 0 | 3 | 15 | 18 | 8 | 12 |
| marble_arch | 18 | 13 | 8 | 8 | 6 | 9 | 10 | 6 | 3 | 3 | 0 | 15 | 17 | 5 | 8 |
| goodge_street | 23 | 10 | 8 | 10 | 12 | 6 | 4 | 6 | 8 | 15 | 15 | 0 | 5 | 7 | 15 |
| warren_street | 25 | 12 | 12 | 12 | 14 | 8 | 6 | 8 | 12 | 18 | 17 | 5 | 0 | 8 | 15 |
| regents_park | 18 | 15 | 10 | 8 | 8 | 12 | 7 | 3 | 6 | 8 | 5 | 7 | 8 | 0 | 8 |
| baker_street | 20 | 20 | 15 | 13 | 12 | 20 | 12 | 8 | 5 | 12 | 8 | 15 | 15 | 8 | 0 |

## **<u>Rule: travel_short</u>**

```
%Best first search
travel_short(Origin, Destination, Route):- search_bfs([[Origin]], Destination, Route).
```

Predicate travel_short will return a Route from an Origin to a Destination by using best-first search algorithm.

## **<u>Rule: search_bfs</u>**

```
%If existing path arrived to destination
search_bfs(PathQueue, End, Route):-
    PathQueue = [Route|_],
    last(Route, End).

% Recurvise search after a solution is found
search_bfs(PathQueue, End, SolutionPath):-
    PathQueue = [Route|Routes],
    last(Route, End),
    search_bfs(Routes, End, SolutionPath).

% If the current route does not end in destination, goes for its
% adjacents and enqueue new route to continue search.
search_bfs(PathQueue, End, SolutionPath):-
    PathQueue = [Route|Routes],
    last(Route, Last),
    Last \= End,
    append_adjacents(Route, AugPaths),
    append(Routes, AugPaths, NewQueue),
    sort_by_distance(NewQueue, End, SortedQueue),
    search_bfs(SortedQueue, End, SolutionPath).
```

Predicate search_bfs works in a way that it has a priority queue, where the priority is only given by estimated distance between stations. It maintains a “priority queue” (PathQueue) of partial paths from the origin. All paths in PathQueue:
- start at the origin,
- do not contain cycles, and
- are sorted by the distance of the last station, to the destination.

If a path does not end in destination, when picked (front of queue), it is removed and all “extensions” are added to the queue. A route R ending at station X is extended by appending to R the stations that are adjacent to X.
For instance, for route: [victoria, green_park]

Its extensions are
[victoria, green_park, oxford_circus],
[victoria, green_park, picadilly_circus],
[victoria, green_park, hyde_park_corner],
**[victoria, green_park, victoria], <- will be removed.**

as oxford_circus, victoria, picadilly_circus and hyde_park_corner are the stations next to green_park. Before continuing the search, the paths that contain repeated stations (cycles) are removed; thus, path [victoria, green_park, victoria] is removed (not considered).

Here is the summary of the third search_bfs predicate,

```
% If the current route does not end in destination, goes for its
% adjacents and enqueue new route to continue search.
search_bfs(PathQueue, End, SolutionPath):-
    PathQueue = [Route|Routes],
    last(Route, Last),
    Last \= End,
    append_adjacents(Route, AugPaths),
    append(Routes, AugPaths, NewQueue),
    sort_by_distance(NewQueue, End, SortedQueue),
    search_bfs(SortedQueue, End, SolutionPath).
```

1. It takes the first route (Route) in the sorted queue of routes (PathQueue).
2. Checks the last station (Last) of Route.
3. Checks that it’s not the destination (Last \= End), otherwise it will use the “base case” rules of the predicate.
4. Extends current route (Route) with the stations adjacent to Last (obtaining AugPaths), and taking care of not including paths with cycles (see definition of predicate append_adjacents).
5. Remove Route (just keeps Paths, the tail of PathQueue), and append augmented paths (AugPaths), obtaining the new queue (NewQueue).
6. Sort the paths in NewQueue according to distance to destination(see predicate sort_by_distance), obtaining SortedQueue.
7. Recursively call search_bfs with the updated queue.

## Rule: sort_by_distance

Sort_by_distance is used to sort the route by its distance.

```
%Sort the queue by estimated distance to destination

sort_by_distance([], _, []).
sort_by_distance([X], _, [X]).
sort_by_distance([X|Xs], End, [Min|SortedXs]):-
    min_path([X|Xs], End, Min),
    delete([X|Xs], Min, RemovedMin),
    sort_by_distance(RemovedMin, End, SortedXs).
```

## Rule: min_path

min_path is meant to use to check the route where its last node is closer to destination.

```
% It will check if the last node the path is closer to destination.
min_path([X], _, X).
min_path([X1|[X2|Xs]], End, Min):-
    cost(X1, End, C1),
    cost(X2, End, C2),
    C1 =< C2,
    min_path([X1|Xs], End, Min).

min_path([X1|[X2|Xs]], End, Min):-
    cost(X1, End, C1),
    cost(X2, End, C2),
    C1 > C2,
    min_path([X2|Xs], End, Min).
```

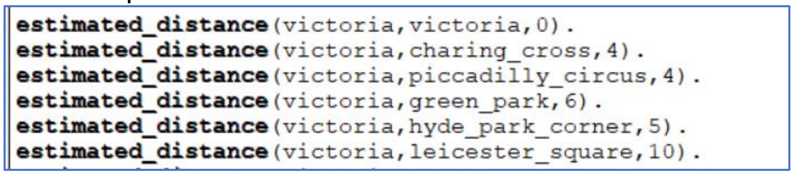
## Rule: cost

Estimated cost from last station of a path to a destination station

```
%Estimated distance table.
cost(P, End, C):- last(P, X), estimated_distance(X, End, C).
```

**Fact: estimated_distance**
A list of knowledge fact that is used as a heuristic function for best first search. An estimated distance between stations are given.

For example:



```
estimated_distance(victoria,victoria,0).
estimated_distance(victoria,charing_cross,4).
estimated_distance(victoria,piccadilly_circus,4).
estimated_distance(victoria,green_park,6).
estimated_distance(victoria,hyde_park_corner,5).
estimated_distance(victoria,leicester_square,10).
```

## 4. Conclusion

As a conclusion, depth first search will give you a complete search if there is a solution. It will not always give you optimal solution, and therefore its cost might be high, like the time and money spent for the traveller. Example, it may provide you a big round to your destination. However, as compared with best first search, best first search will always give us optimal solution, that is the shortest path to our destination, and its time complexity is lesser than depth first search. Hence it is the best solution because of its efficiency for this travelling assignment.

## 5. Reference

1. https://en.wikibooks.org/wiki/Prolog/Search_techniques
2. https://en.wikipedia.org/wiki/Search_algorithm
3. https://en.wikipedia.org/wiki/Depth-first_search
4. https://en.wikipedia.org/wiki/Best-first_search
5. https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/
6. https://www.cet.edu.in/noticefiles/271_AI%20Lect%20Notes.pdf
7. https://swish.swi-prolog.org/p/ltc_underground.swinb

## 6. Appendix

```
%%The station and its neighbouring station(s)
nextto(victoria, [green_park]).
nextto(charing_cross, [piccadilly_circus, leicester_square]).
nextto(piccadilly_circus, [green_park, charing_cross, leicester_square, oxford_circus]).
nextto(leicester_square, [piccadilly_circus, charing_cross, tottenham_court_road,regents_park, bond_street]).
nextto(oxford_circus, [piccadilly_circus, green_park, tottenham_court_road]).
nextto(tottenham_court_road, [oxford_circus,piccadilly_circus, leicester_square, goodge_street]).
nextto(green_park, [victoria, piccadilly_circus, oxford_circus, hyde_park_corner]).
nextto(hyde_park_corner, [green_park]).
nextto(lancaster_gate, [marble_arch]).
nextto(marble_arch, [bond_street]).
nextto(bond_street, [marble_arch, baker_street, oxford_circus]).
nextto(regents_park, [baker_street, oxford_circus]).
nextto(baker_street, [regents_park]).
nextto(goodge_street, [tottenham_court_road, warren_street]).
nextto(warren_street, [goodge_street]).
```

```
%Rules
travel(Start, End, Route):-
   search([], Start, End, Route).


%If the start point is the destination
search(Route, End, End, Solution):-
   append(Route, [End], Solution).

%If the start point is not destination
search(Route, Start, End, Solution):-
   nextto(Start, Adjs),
   member(N, Adjs),
   \+ (member(N, Route)),
   append(Route, [Start], TempPath),
   search(TempPath, N, End, Solution).

%%%
%%%Takes a non-empty path, and extends it with all adjacent of last element (as long as they don't repeat elements, to
avoid cycles)
%%%
append_adjacents(Route, AugPaths):- last(Route, Last), nextto(Last, Adjs), append_all(Route, Adjs, AugPaths).

append_all(_, [], []).
append_all(Route, [X|Xs], L):- member(X,Route), append_all(Route, Xs, L).
append_all(Route, [X|Xs], [AugPath|L]):- \+ (member(X,Route)), append(Route, [X], AugPath), append_all(Route, Xs, L).

%Best first search
travel_short(Start, End, Route):- search_bfs([[Start]], End, Route).

%If existing path arrived to destination
search_bfs(PathQueue, End, Route):-
   PathQueue = [Route|_],
   last(Route, End).

% Recurvise search after a solution is found
search_bfs(PathQueue, End, SolutionPath):-
   PathQueue = [Route|Routes],
   last(Route, End),
   search_bfs(Routes, End, SolutionPath).

% If the current route does not end in destination, goes for its
% adjacents and enqueue new route to continue search.
search_bfs(PathQueue, End, SolutionPath):-
   PathQueue = [Route|Routes],
   last(Route, Last),
   Last \= End,
   append_adjacents(Route, AugPaths),
   append(Routes, AugPaths, NewQueue),
   sort_by_distance(NewQueue, End, SortedQueue),
   search_bfs(SortedQueue, End, SolutionPath).

%Sort the queue by estimated distance to destination

sort_by_distance([], _, []).
sort_by_distance([X], _, [X]).
sort_by_distance([X|Xs], End, [Min|SortedXs]):-
   min_path([X|Xs], End, Min),
   delete([X|Xs], Min, RemovedMin),
   sort_by_distance(RemovedMin, End, SortedXs).

% It will check if the last node the path is closer to destination.
min_path([X], _, X).
min_path([X1|[X2|Xs]], End, Min):-
    cost(X1, End, C1),
    cost(X2, End, C2),
    C1 =< C2,
    min_path([X1|Xs], End, Min).

min_path([X1|[X2|Xs]], End, Min):-
    cost(X1, End, C1),
    cost(X2, End, C2),
    C1 > C2,
    min_path([X2|Xs], End, Min).


%Estimated distance table.
cost(P, End, C):- last(P, X), estimated_distance(X, End, C).


estimated_distance(victoria,victoria,0).
estimated_distance(victoria,charing_cross,4).
estimated_distance(victoria,piccadilly_circus,4).
estimated_distance(victoria,green_park,6).
estimated_distance(victoria,hyde_park_corner,5).
estimated_distance(victoria,leicester_square,10).
estimated_distance(victoria,tottenham_court_road,14).
estimated_distance(victoria,oxford_circus,13).
estimated_distance(victoria,bond_street,17).
estimated_distance(victoria,lancaster_gate,16).
estimated_distance(victoria,marble_arch,18).
estimated_distance(victoria,goodge_street,23).
estimated_distance(victoria,warren_street,25).
estimated_distance(victoria,regents_park,18).
estimated_distance(victoria,baker_street,20).
estimated_distance(charing_cross,victoria,4).
estimated_distance(charing_cross,charing_cross,0).
estimated_distance(charing_cross,piccadilly_circus,2).
estimated_distance(charing_cross,green_park,7).
estimated_distance(charing_cross,hyde_park_corner,8).
estimated_distance(charing_cross,leicester_square,5).
estimated_distance(charing_cross,tottenham_court_road,7).
estimated_distance(charing_cross,oxford_circus,10).
estimated_distance(charing_cross,bond_street,13).
estimated_distance(charing_cross,lancaster_gate,12).
estimated_distance(charing_cross,marble_arch,13).
estimated_distance(charing_cross,goodge_street,10).
estimated_distance(charing_cross,warren_street,12).
estimated_distance(charing_cross,regents_park,15).
estimated_distance(charing_cross,baker_street,20).
estimated_distance(piccadilly_circus,victoria,4).
estimated_distance(piccadilly_circus,charing_cross,2).
estimated_distance(piccadilly_circus,piccadilly_circus,0).
estimated_distance(piccadilly_circus,green_park,2).
estimated_distance(piccadilly_circus,hyde_park_corner,5).
estimated_distance(piccadilly_circus,leicester_square,3).
estimated_distance(piccadilly_circus,tottenham_court_road,4).
estimated_distance(piccadilly_circus,oxford_circus,5).
estimated_distance(piccadilly_circus,bond_street,6).
estimated_distance(piccadilly_circus,lancaster_gate,6).
estimated_distance(piccadilly_circus,marble_arch,8).
estimated_distance(piccadilly_circus,goodge_street,8).
estimated_distance(piccadilly_circus,warren_street,12).
estimated_distance(piccadilly_circus,regents_park,10).
estimated_distance(piccadilly_circus,baker_street,15).
estimated_distance(green_park,victoria,6).
estimated_distance(green_park,charing_cross,7).
estimated_distance(green_park,piccadilly_circus,2).
estimated_distance(green_park,green_park,0).
estimated_distance(green_park,hyde_park_corner,2).
estimated_distance(green_park,leicester_square,8).
estimated_distance(green_park,tottenham_court_road,6).
estimated_distance(green_park,oxford_circus,5).
estimated_distance(green_park,bond_street,6).
estimated_distance(green_park,lancaster_gate,8).
estimated_distance(green_park,marble_arch,8).
estimated_distance(green_park,goodge_street,10).
estimated_distance(green_park,warren_street,12).
estimated_distance(green_park,regents_park,8).
estimated_distance(green_park,baker_street,13).
estimated_distance(hyde_park_corner,victoria,5).
estimated_distance(hyde_park_corner,charing_cross,8).
estimated_distance(hyde_park_corner,piccadilly_circus,5).
estimated_distance(hyde_park_corner,green_park,2).
estimated_distance(hyde_park_corner,hyde_park_corner,0).
estimated_distance(hyde_park_corner,leicester_square,8).
estimated_distance(hyde_park_corner,tottenham_court_road,10).
estimated_distance(hyde_park_corner,oxford_circus,6).
estimated_distance(hyde_park_corner,bond_street,5).
estimated_distance(hyde_park_corner,lancaster_gate,7).
estimated_distance(hyde_park_corner,marble_arch,6).
estimated_distance(hyde_park_corner,goodge_street,12).
estimated_distance(hyde_park_corner,warren_street,14).
estimated_distance(hyde_park_corner,regents_park,8).
estimated_distance(hyde_park_corner,baker_street,12).
estimated_distance(leicester_square,victoria,10).
estimated_distance(leicester_square,charing_cross,5).
estimated_distance(leicester_square,piccadilly_circus,3).
estimated_distance(leicester_square,green_park,8).
estimated_distance(leicester_square,hyde_park_corner,8).
estimated_distance(leicester_square,leicester_square,0).
estimated_distance(leicester_square,tottenham_court_road,4).
estimated_distance(leicester_square,oxford_circus,10).
estimated_distance(leicester_square,bond_street,12).
estimated_distance(leicester_square,lancaster_gate,8).
estimated_distance(leicester_square,marble_arch,9).
estimated_distance(leicester_square,goodge_street,6).
estimated_distance(leicester_square,warren_street,8).
estimated_distance(leicester_square,regents_park,12).
estimated_distance(leicester_square,baker_street,20).
estimated_distance(tottenham_court_road,victoria,14).
estimated_distance(tottenham_court_road,charing_cross,7).
estimated_distance(tottenham_court_road,piccadilly_circus,4).
estimated_distance(tottenham_court_road,green_park,6).
estimated_distance(tottenham_court_road,hyde_park_corner,10).
estimated_distance(tottenham_court_road,leicester_square,4).
estimated_distance(tottenham_court_road,tottenham_court_road,0).
estimated_distance(tottenham_court_road,oxford_circus,4).
estimated_distance(tottenham_court_road,bond_street,8).
estimated_distance(tottenham_court_road,lancaster_gate,12).
estimated_distance(tottenham_court_road,marble_arch,10).
estimated_distance(tottenham_court_road,goodge_street,4).
estimated_distance(tottenham_court_road,warren_street,6).
estimated_distance(tottenham_court_road,regents_park,7).
estimated_distance(tottenham_court_road,baker_street,12).
estimated_distance(oxford_circus,victoria,13).
estimated_distance(oxford_circus,charing_cross,10).
estimated_distance(oxford_circus,piccadilly_circus,5).
estimated_distance(oxford_circus,green_park,5).
estimated_distance(oxford_circus,hyde_park_corner,6).
estimated_distance(oxford_circus,leicester_square,10).
estimated_distance(oxford_circus,tottenham_court_road,4).
estimated_distance(oxford_circus,oxford_circus,0).
estimated_distance(oxford_circus,bond_street,4).
estimated_distance(oxford_circus,lancaster_gate,7).
estimated_distance(oxford_circus,marble_arch,6).
estimated_distance(oxford_circus,goodge_street,6).
estimated_distance(oxford_circus,warren_street,8).
estimated_distance(oxford_circus,regents_park,3).
estimated_distance(oxford_circus,baker_street,8).
estimated_distance(bond_street,victoria,17).
estimated_distance(bond_street,charing_cross,13).
estimated_distance(bond_street,piccadilly_circus,6).
estimated_distance(bond_street,green_park,6).
estimated_distance(bond_street,hyde_park_corner,5).
estimated_distance(bond_street,leicester_square,12).
estimated_distance(bond_street,tottenham_court_road,8).
estimated_distance(bond_street,oxford_circus,4).
estimated_distance(bond_street,bond_street,0).
estimated_distance(bond_street,lancaster_gate,5).
estimated_distance(bond_street,marble_arch,3).
estimated_distance(bond_street,goodge_street,8).
estimated_distance(bond_street,warren_street,12).
estimated_distance(bond_street,regents_park,6).
estimated_distance(bond_street,baker_street,5).
estimated_distance(lancaster_gate,victoria,16).
estimated_distance(lancaster_gate,charing_cross,12).
estimated_distance(lancaster_gate,piccadilly_circus,6).
estimated_distance(lancaster_gate,green_park,8).
estimated_distance(lancaster_gate,hyde_park_corner,7).
estimated_distance(lancaster_gate,leicester_square,8).
estimated_distance(lancaster_gate,tottenham_court_road,12).
estimated_distance(lancaster_gate,oxford_circus,7).
estimated_distance(lancaster_gate,bond_street,5).
estimated_distance(lancaster_gate,lancaster_gate,0).
estimated_distance(lancaster_gate,marble_arch,3).
estimated_distance(lancaster_gate,goodge_street,15).
estimated_distance(lancaster_gate,warren_street,18).
estimated_distance(lancaster_gate,regents_park,8).
estimated_distance(lancaster_gate,baker_street,12).
estimated_distance(marble_arch,victoria,18).
estimated_distance(marble_arch,charing_cross,13).
estimated_distance(marble_arch,piccadilly_circus,8).
estimated_distance(marble_arch,green_park,8).
estimated_distance(marble_arch,hyde_park_corner,6).
estimated_distance(marble_arch,leicester_square,9).
estimated_distance(marble_arch,tottenham_court_road,10).
estimated_distance(marble_arch,oxford_circus,6).
estimated_distance(marble_arch,bond_street,3).
estimated_distance(marble_arch,lancaster_gate,3).
estimated_distance(marble_arch,marble_arch,0).
estimated_distance(marble_arch,goodge_street,15).
estimated_distance(marble_arch,warren_street,17).
estimated_distance(marble_arch,regents_park,5).
estimated_distance(marble_arch,baker_street,8).
estimated_distance(goodge_street,victoria,23).
estimated_distance(goodge_street,charing_cross,10).
estimated_distance(goodge_street,piccadilly_circus,8).
estimated_distance(goodge_street,green_park,10).
estimated_distance(goodge_street,hyde_park_corner,12).
estimated_distance(goodge_street,leicester_square,6).
estimated_distance(goodge_street,tottenham_court_road,4).
estimated_distance(goodge_street,oxford_circus,6).
estimated_distance(goodge_street,bond_street,8).
estimated_distance(goodge_street,lancaster_gate,15).
estimated_distance(goodge_street,marble_arch,15).
estimated_distance(goodge_street,goodge_street,0).
estimated_distance(goodge_street,warren_street,5).
estimated_distance(goodge_street,regents_park,7).
estimated_distance(goodge_street,baker_street,15).
estimated_distance(warren_street,victoria,25).
estimated_distance(warren_street,charing_cross,12).
estimated_distance(warren_street,piccadilly_circus,12).
estimated_distance(warren_street,green_park,12).
estimated_distance(warren_street,hyde_park_corner,14).
estimated_distance(warren_street,leicester_square,8).
estimated_distance(warren_street,tottenham_court_road,6).
estimated_distance(warren_street,oxford_circus,8).
estimated_distance(warren_street,bond_street,12).
estimated_distance(warren_street,lancaster_gate,18).
estimated_distance(warren_street,marble_arch,17).
estimated_distance(warren_street,goodge_street,5).
estimated_distance(warren_street,warren_street,0).
estimated_distance(warren_street,regents_park,8).
estimated_distance(warren_street,baker_street,15).
estimated_distance(regents_park,victoria,18).
estimated_distance(regents_park,charing_cross,15).
estimated_distance(regents_park,piccadilly_circus,10).
estimated_distance(regents_park,green_park,8).
estimated_distance(regents_park,hyde_park_corner,8).
estimated_distance(regents_park,leicester_square,12).
estimated_distance(regents_park,tottenham_court_road,7).
estimated_distance(regents_park,oxford_circus,3).
estimated_distance(regents_park,bond_street,6).
estimated_distance(regents_park,lancaster_gate,8).
estimated_distance(regents_park,marble_arch,5).
estimated_distance(regents_park,goodge_street,7).
estimated_distance(regents_park,warren_street,8).
estimated_distance(regents_park,regents_park,0).
estimated_distance(regents_park,baker_street,8).
estimated_distance(baker_street,victoria,20).
estimated_distance(baker_street,charing_cross,20).
estimated_distance(baker_street,piccadilly_circus,15).
estimated_distance(baker_street,green_park,13).
estimated_distance(baker_street,hyde_park_corner,12).
estimated_distance(baker_street,leicester_square,20).
estimated_distance(baker_street,tottenham_court_road,12).
estimated_distance(baker_street,oxford_circus,8).
estimated_distance(baker_street,bond_street,5).
estimated_distance(baker_street,lancaster_gate,12).
estimated_distance(baker_street,marble_arch,8).
estimated_distance(baker_street,goodge_street,15).
estimated_distance(baker_street,warren_street,15).
estimated_distance(baker_street,regents_park,8).
estimated_distance(baker_street,baker_street,0).
```