AI And AI Programming – Course Work

Name – Kesavarapu Kowshik.

UID – 6678793

Choice of Question – 2 (Prolog).

Table of Contents

1.Executive Summary

This assignment is to learn and understand the logic programming using Prolog. Here the main goal is to build the network of stations from London underground and try to implement search to find a route from one station to other station. Secondary goal is to implement AI search algorithm to improve the search by using background knowledge.

For this assignment I chose to implement Depth first search which is the default search in Prolog and then using the background knowledge of estimated time I will be implementing Best First search.

Depth First Search (DFS) is uninformed search where the goal Is finding the solution by just going through the nodes and hoping to find the solution. This kind of search is not very efficient and often tend to give large solution.

Best First Search (BFS) uses the background knowledge available and tries to find the best solution. This type of search is very suitable for these kinds of applications.

Here I am using 23 different stations and building the network using prebuilt list handling predicate nextto and then created the background knowledge of estimated_time between all the stations. This time is randomly chosen from 1 Minute to 5 Minutes.

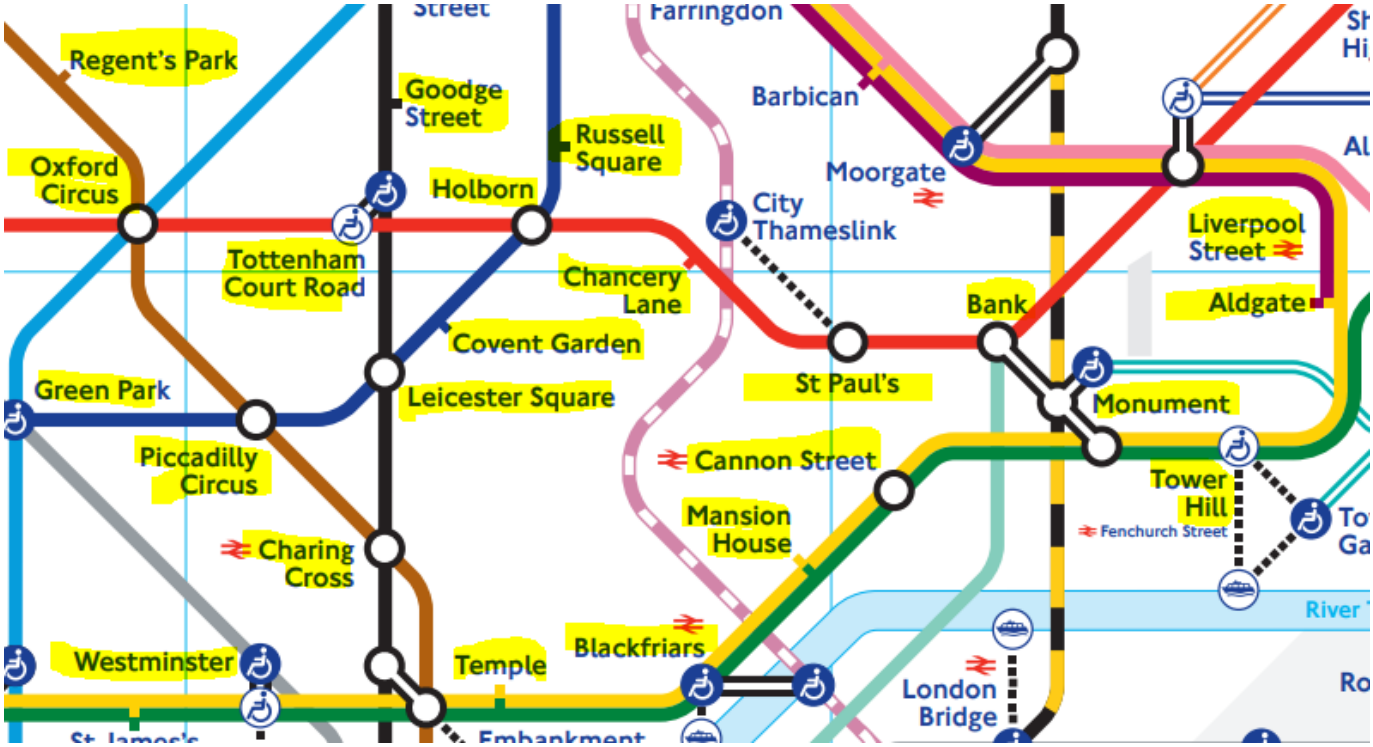
The main goal for me at the end of this assignment is to understand how logic-based programming works and how different search algorithms work. I will be trying to clearly show that BFS works better than DFS at the end of this assignment.

2. Introduction

For this assignment I have chooses 23 stations from the London underground they are Regents_Park, Oxford_Circus, Green_Park, Piccadilly_Circus, Charing_Cross, Westminister, Goodge_Street, Tottenham_Court_Road, Leicester_Square, Temple, Covent_Garden, Holborn, Russel_Square, Cahncery_Lane, Blackfriars, St_Pauls, Cannon_Street, Mansion_House, Monument, Aldgate, Liverpool_Street, Bank, Tower_Hill.

They run across Central line, Northern line, Bakerloo line, Circle line, District line and Metropolitan Line of London Tube.

Below picture shows the chosen stations.



Source - https://content.tfl.gov.uk/standard-tube-map.pdf

3.Prolog representation

I have used prolog's inbuilt list function nextto to represent the stations.

It has list of two lists, and I am using to represent all the stations that connect to a station. For example, oxford_circus connects to three different stations (regents_park, tottenham_court_road and piccadilly_circus) this can be represented as

```
nextto(oxford_circus, [regents_park, tottenham_court_road, piccadilly_circus]).
```

This gives us total of **23 facts** for 23 stations.

Now to implement DFS I have written a rule as follows

```
commute(StartStation,DestinationStation , Route):-
search([], StartStation, DestinationStation, Route).
%If the start point is the destination
search(Route, DestinationStation, DestinationStation, Solution):-
append(Route, [DestinationStation], Solution).
%If the start point is not destination
search(Route, StartStation, DestinationStation, Solution):-
nextto(StartStation, Adjs),
member(N, Adjs),
\+ (member(N, Route)),
append(Route, [StartStation], TemporaryPath),
search(TemporaryPath, N, DestinationStation, Solution).
```

Here commute is the rule which calls Search with four arguments Start Station , Destination Station , route and [] (this is the way for the search to return the solution). if the start and end are the same the search terminates at once this is just to minimize the cost.

If the Start Station and Destination Station are different the search, try to find all the adjacent connected stations and then checks weather the solution is reached or not if not reached the search will run recursively until the solution is reached.

The result route for path between regents_park and temple is

```
?- commute(regents_park, temple, Route).
Route = [regents_park, oxford_circus, tottenham_court_road, leicester_square, piccadilly_circus, charing_c
ross, temple]
```

The result route for path between regents_park and oxford_circus is

```
?- commute(regents_park, oxford_circus, Route).
Route = [regents_park, oxford_circus]
```

Using the trace function, we can see how the program tries to find the solution for commute(regents_park, oxford_circus, Route).

```
[trace]  ?-
|    commute(regents_park, oxford_circus, Route).
   Call: (10) commute(regents_park, oxford_circus, _54304) ? creep
   Call: (11) search([], regents_park, oxford_circus, _54304) ? creep
   Call: (12) nextto(regents_park, _56280) ? creep
   Exit: (12) nextto(regents_park, [oxford_circus]) ? creep
   Call: (12) lists:member(_57796, [oxford_circus]) ? creep
   Exit: (12) lists:member(oxford_circus, [oxford_circus]) ? creep
   Call: (12) lists:member(oxford_circus, []) ? creep
   Fail: (12) lists:member(oxford_circus, []) ? creep
   Redo: (11) search([], regents_park, oxford_circus, _54304) ? creep
   Call: (12) lists:append([], [regents_park], _61630) ? creep
   Exit: (12) lists:append([], [regents_park], [regents_park]) ? creep
   Call: (12) search([regents_park], oxford_circus, oxford_circus, _54304) ? creep
   Call: (13) lists:append([regents_park], [oxford_circus], _54304) ? creep
   Exit: (13) lists:append([regents_park], [oxford_circus], [regents_park, oxford_circus]) ? creep
   Exit: (12) search([regents_park], oxford_circus, oxford_circus, [regents_park, oxford_circus]) ? creep
   Exit: (11) search([], regents_park, oxford_circus, [regents_park, oxford_circus]) ? creep
   Exit: (10) commute(regents_park, oxford_circus, [regents_park, oxford_circus]) ? creep
Route = [regents_park, oxford_circus] .
```

Here we can see the calling of all the different rules. We can see the fail when it was no long was able to find the destination. Then the rule retraced the last step and went through other direction i.e, another route. The search will terminate when we reach the destination station . This is a very ineffective and time-consuming process as we are literally trying to see all the paths and trying to figure out weather the destination can be reached or not.

For example, the route between regents_park and holborn gives the following output

```
?- commute(regents_park, holborn, Route).
Route = [regents_park, oxford_circus, tottenham_court_road, leicester_square, covent_garden, holborn]
```

This is not the optimal path as holborn and Tottenham_court_road are connected directly but the solution reroutes and adds two unnecessary stops and takes us around the way to reach the destination.

Now in order to try and find the optimal path I am building background knowledge of estimated time between two stations. this is represented using fact estimated_time. It contains the names of two stations and the time it takes in between the said stations. This resulted in further 63 new facts. The example of few said facts.

```
estimated_time(regents_park,oxford_circus,2).
estimated_time(oxford_circus,regents_park,3).
estimated_time(oxford_circus,tottenham_court_road,2).
estimated_time(oxford_circus,piccadilly_circus,3).
estimated_time(oxford_circus,regents_park,4).
estimated_time(green_park,westminister,2).
estimated_time(green_park,oxford_circus,4).
estimated_time(green_park,piccadilly_circus,1).
estimated_time(piccadilly_circus,oxford_circus,4).
estimated_time(piccadilly_circus,leicester_square,2).
estimated_time(piccadilly_circus,charing_cross,4).
```

The estimated time is randomly given these range from 1 minute to 5 minutes.

Now with the help of this background knowledge we use BFS to predict the best route and time it takes to reach from station A to Station B.

```
commute_short(Start, End, Visited, Result) :-
    commute_short(Start, End, [Start], 0, Visited, Result).
commute_short(Start, End, Waypoints, TimeAcc, Visited, TotalTime) :-
    estimated_time(Start, End, Time),
    reverse([End|Waypoints], Visited),
    TotalTime is TimeAcc + Time.
commute_short(Start, End, Waypoints, TimeAcc, Visited, TotalTime) :-
    estimated_time(Start, Waypoint, Time),
    \+ member(Waypoint, Waypoints),
    NewTimeAcc is TimeAcc + Time,
    commute_short(Waypoint, End, [Waypoint|Waypoints], NewTimeAcc, Visited, TotalTime).
```

Here I have used commute_short rule to find the solution, it takes four arguments starting station , destination station , Route, and Time. Route is to store the route and time is to store the time taken When it's called for the first time it initializes the time taken to 0 and then calls the third rule with it Now it tries to find the best possible solution by recursively going through all the members and solutions.at the end the time is again calculated by going through reverse the loop and adding the time.

This gives the best solution. Now for the same start and end stations we used to justify that the DFS will not give the best results, BFS gives the following result,

```
?- commute(regents_park, holborn, Route).
Route = [regents_park, oxford_circus, tottenham_court_road, leicester_square, covent_garden, holborn] .

?- commute_short(regents_park, holborn, Route,Time).
Route = [regents_park, oxford_circus, tottenham_court_road, holborn],
Time = 7
```

Here we can see commute _short gave the best possible outcome for the route.

Here with the help of trace let us see how this works for the same example we used above

```
[trace]  ?- commute_short(regents_park, oxford_circus, Route,Time).
   Call: (10) commute_short(regents_park, oxford_circus, _19130, _19132) ? creep
   Call: (11) commute_short(regents_park, oxford_circus, [regents_park], 0, _19130, _19132) ? creep
   Call: (12) estimated_time(regents_park, oxford_circus, _21150) ? creep
   Exit: (12) estimated_time(regents_park, oxford_circus, 2) ? creep
   Call: (12) lists:reverse([oxford_circus, regents_park], _19130) ? creep
   Exit: (12) lists:reverse([oxford_circus, regents_park], [regents_park, oxford_circus]) ? creep
   Call: (12) _19132 is 0+2 ? creep
   Exit: (12) 2 is 0+2 ? creep
   Exit: (11) commute_short(regents_park, oxford_circus, [regents_park], 0, [regents_park, oxford_circus],
 2) ? creep
   Exit: (10) commute_short(regents_park, oxford_circus, [regents_park, oxford_circus], 2) ? creep
Route = [regents_park, oxford_circus],
Time = 2
```

Here we can observe how the calling of different rules works.

Below are few examples of commute and commute short rules and their outputs

```
?- commute(regents_park, green_park, Route).
Route = [regents_park, oxford_circus, tottenham_court_road, leicester_square, piccadilly_circus, charing_cross
 temple, westminister, green_park] .

?- commute_short(regents_park, green_park, Route,Time).
Route = [regents_park, oxford_circus, green_park],
Time = 6
```

There is also a possibility that DFS may also give the same result as BFS which can be seen in the below example.

```
?- commute_short(regents_park, bank, Route,Time).
Route = [regents_park, oxford_circus, tottenham_court_road, leicester_square, covent_garden, holborn, cahncery_
lane, st_pauls, bank],
Time = 20 .

?- commute(regents_park, bank, Route).
Route = [regents_park, oxford_circus, tottenham_court_road, leicester_square, covent_garden, holborn, cahncery_
lane, st_pauls, bank]
```

4.Conclusion

We can clearly see that BFS works better than DFS, As BFS is an informed search algorithm it performed better using the background knowledge. But Un informed search algorithms do have their place in the real world when there is no background knowledge available. So, in conclusion DFS gives the solution provided there is one, but it may not be the most optimal or shortest one and sometimes can route you around to get to the next station itself. BFS gives the best solution with respect to background knowledge of the time it takes to travel from one station to other.

5.References: -

1: - Tube map - https://content.tfl.gov.uk/standard-tube-map.pdf

2:- Prolog Search Techniques –

- https://en.wikibooks.org/wiki/Prolog/Search_techniques
- https://www.csee.umbc.edu/courses/771/current/presentations/prolog%20search.pdf

3: - Sample of London tube - https://swish.swi-prolog.org/p/ltc_underground.swinb

4:- Prolog List manipulation - https://www.swi-prolog.org/pldoc/man?section=lists

6.Appendix

```
nextto(regents_park, [oxford_circus]).

nextto(oxford_circus, [regents_park, tottenham_court_road, piccadilly_circus,green_park]).

nextto(green_park, [westminister, oxford_circus, piccadilly_circus]).

nextto(piccadilly_circus, [oxford_circus, leicester_square, charing_cross, green_park]).

nextto(charing_cross, [piccadilly_circus, temple, leicester_square]).

nextto(westminister, [green_park, temple]).

nextto(goodge_street, [tottenham_court_road]).

nextto(tottenham_court_road, [oxford_circus, goodge_street, leicester_square, holborn]).

nextto(leicester_square, [tottenham_court_road, covent_garden, piccadilly_circus]).

nextto(temple, [charing_cross, blackfriars, westminister]).

nextto(covent_garden, [holborn, leicester_square]).

nextto(holborn, [russel_square, tottenham_court_road, covent_garden, cahncery_lane]).

nextto(russel_square, [holborn]).

nextto(cahncery_lane, [holborn, st_pauls]).

nextto(blackfriars, [mansion_house, temple]).

nextto(st_pauls,[bank, cahncery_lane]).

nextto(cannon_street, [mansion_house, monument]).

nextto(mansion_house, [blackfriars, cannon_street]).

nextto(monument, [bank, tower_hill]).
```

```
nextto(aldgate, [tower_hill, liverpool_street]).
nextto(liverpool_street, [aldgate, bank]).
nextto(bank, [st_pauls, monument, liverpool_street]).
nextto(tower_hill, [monument, aldgate]).
commute(StartStation,DestinationStation , Route):-
search([], StartStation, DestinationStation, Route).
search(Route, DestinationStation, DestinationStation, Solution):-
append(Route, [DestinationStation], Solution).
search(Route, StartStation, DestinationStation, Solution):-
nextto(StartStation, Adjs),
member(N, Adjs),
\+ (member(N, Route)),
append(Route, [StartStation], TemporaryPath),
search(TemporaryPath, N, DestinationStation, Solution).
estimated_time(regents_park,oxford_circus,2).
estimated_time(oxford_circus,regents_park,3).
estimated_time(oxford_circus,tottenham_court_road,2).
estimated_time(oxford_circus,piccadilly_circus,3).
estimated_time(oxford_circus,regents_park,4).
estimated_time(oxford_circus,green_park,4).
estimated_time(green_park,westminister,2).
estimated_time(green_park,oxford_circus,4).
estimated_time(green_park,piccadilly_circus,1).
estimated_time(piccadilly_circus,oxford_circus,4).
estimated_time(piccadilly_circus,leicester_square,2).
estimated_time(piccadilly_circus,charing_cross,4).
estimated_time(piccadilly_circus,green_park,2).
estimated_time(charing_cross,piccadilly_circus,3).
estimated_time(charing_cross,temple,4).
estimated_time(charing_cross,leicester_square,3).
estimated_time(westminister,green_park,4).
```

```
estimated_time(westminister,temple,2).
estimated_time(goodge_street,tottenham_court_road,3).
estimated_time(tottenham_court_road,oxford_circus,4).
estimated_time(tottenham_court_road,goodge_street,3).
estimated_time(tottenham_court_road,leicester_square,4).
estimated_time(tottenham_court_road,holborn,3).
estimated_time(leicester_square,tottenham_court_road,2).
estimated_time(leicester_square,covent_garden,3).
estimated_time(leicester_square,westminister,4).
estimated_time(leicester_square,piccadilly_circus,1).
estimated_time(temple,charing_cross,3).
estimated_time(temple,blackfriars,3).
estimated_time(temple,westminister,3).
estimated_time(covent_garden,holborn,3).
estimated_time(covent_garden,leicester_square,3).
estimated_time(holborn,russel_square,3).
estimated_time(holborn,tottenham_court_road,2).
estimated_time(holborn,covent_garden,1).
estimated_time(holborn,cahncery_lane,2).
estimated_time(russel_square,holborn,4).
estimated_time(cahncery_lane,holborn,1).
estimated_time(cahncery_lane,st_pauls,3).
estimated_time(blackfriars,mansion_house,1).
estimated_time(blackfriars,temple,3).
estimated_time(st_pauls,bank,1).
estimated_time(st_pauls,cahncery_lane,4).
estimated_time(cannon_street,mansion_house,4).
estimated_time(cannon_street,monument,2).
estimated_time(mansion_house,blackfriars,1).
estimated_time(mansion_house,cannon_street,4).
estimated_time(monument,bank,2).
```

```
estimated_time(monument,tower_hill,3).
estimated_time(aldgate,tower_hill,1).
estimated_time(aldgate,liverpool_street,4).
estimated_time(liverpool_street,aldgate,3).
estimated_time(liverpool_street,bank,2).
estimated_time(bank,st_pauls,4).
estimated_time(bank,monument,2).
estimated_time(bank,liverpool_street,3).
estimated_time(tower_hill,monument,4).
estimated_time(tower_hill,aldgate,3).
commute_short(Start, End, Visited, Result) :-
   commute_short(Start, End, [Start], 0, Visited, Result).
commute_short(Start, End, Waypoints, TimeAcc, Visited, TotalTime) :-
   estimated_time(Start, End, Time),
   reverse([End|Waypoints], Visited),
   TotalTime is TimeAcc + Time.
commute_short(Start, End, Waypoints, TimeAcc, Visited, TotalTime) :-
   estimated_time(Start, Waypoint, Time),
   \+ member(Waypoint, Waypoints),
   NewTimeAcc is TimeAcc + Time,
   commute_short(Waypoint, End, [Waypoint|Waypoints], NewTimeAcc, Visited, TotalTime).
```