

**BLG435E**

*Artificial Intelligence*

2013 Fall



# **Report of Project 1**

Date of Submission : 29.10 2013

Student Name: Volkan İlbeyli

Student Number : 040100118

Instructor : Sanem Sarıel

Q1)

Agent Type	Performance Measure	Environment	Actuators	Sensors
<i>Mars Rover</i>	Meaningful data about Mars' atmosphere and geology related to a possible life form in Mars	Mars, desert, rocky terrain	Wheels, arm	Camera, infrared, various sensors (pressure, temperature, chemicals, etc.)
<i>Personalized Music Recommendation System</i>	Accurate music recommendation based on taste of an individual	Website, database	Data mining and analysis algorithms	A program to track and record individual's played songs
<i>Autonomous Vacuum Cleaner Robot</i>	Cleanliness of the room	Room, carpets, floor	Wheels, suction apparatus	Camera, laser
<i>Activity Recognition and Anomaly Detection Software Agent In a Hospital</i>	Accurate activity sensing, correct anomaly detection	Hospital, halls, rooms	Pattern recognition and data analysis algorithms	Cameras (nightvision, temperature, chemical, etc.)

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
<i>Mars Rover</i>	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
<i>Personalized Music Recommendation System</i>	Fully	Multiagent	Deterministic	Sequential	Static	Discrete
<i>Autonomous Vacuum Cleaner Robot</i>	Partially	Single	Deterministic	Episodic	Static	Discrete
<i>Activity Recognition and Anomaly Detection Software Agent In a Hospital</i>	Fully	Single	Deterministic	Episodic	Dynamic	Continuous

**Mars Rover:** A utility-based agent fits best for the mars rover since it does not have a concrete goal to achieve, but rather it has to collect lots of data and optimize the knowledge that can be extracted from the data and utility-based agents are more suitable for a complex agent which requires a high-quality behaviour.

**Personalized Music Recommendation System:** A goal-based system suits well for this type of agent. The aim of the agent is to recommend similar music to what the individual listens by maintaining a level of similarity based on the comparison criterion. Since the environment is fully observable, there is no need for a model-based agent and in addition to that; since the task is complicated for a reflex-based agent and is simple for a utility-based agent, goal-based agent is the choice of agent architecture.

**Autonomous Vacuum Cleaner Robot:** A simple reflex agent will suffice for this type of agent since its actions are episodic (current action not affecting the future action) and the complexity of actions and environment is low.

**Activity Recognition and Anomaly Detection Software:** A reflex agent will be the best choice of this agent. The behaviour of agent is episodic and the environment complexity is rather lower compared to mars rover. The algorithm will compare the values that are read from the sensors to some normal values and then decide whether there is an activity or anomaly (deterministic), a simple reflex agent architecture will be sufficient for this agent.

## Q2)

If a heuristic  $h(n)$  is consistent, then

$$h(n) \leq c(n, a, n') + h(n')$$

where  $n'$  is a successor of  $n$ ,  $a$  is the action which changes state to  $n'$  from  $n$  and  $c()$  is the cost function from  $n$  to  $n'$ . In plain English, this inequality is a triangle inequality.

Admissible heuristic is a heuristic that never overestimates the true solution. Straight line distance heuristics are admissible because there is no shorter distance between two points than a straight line. Admissible heuristics are by nature optimal.

Since consistency is a must for optimality, a consistent heuristic is admissible at the same time. The other way around doesn't have to be true.

### Q3)

Data Structures used during the implementation of Tree-Search and Graph-Search DFS and BFS:

A Route class:

```
class Route
{
    public Route()
    {
        stops = new List<Stop>();
    }

    public void Print()
    {
        Console.WriteLine("Id: " + id + "\tName: " + name);
        Console.WriteLine("Stops:");
        foreach (Stop stop in stops)
        {
            Console.WriteLine(stop.name + " ");
        }
        Console.WriteLine();
    }

    public int id { get; set; }
    public String name { get; set; }
    public List<Stop> stops { get; set; }
}
```

A Stop Class:

```
class Stop
{
    public void Print()
    {
        Console.WriteLine("Id: " + id + "\tName: " + name);
    }

    public int id { get; set; }
    public double latitude { get; set; }
    public double longitude { get; set; }
    public String name { get; set; }
}
```

And for the solution tree there are Node and Tree classes:

```
class Tree
{
    public Tree()
    {
        nodeList = new List<Node>();
    }

    public Node root { get; set; }
    public List<Node> nodeList { get; set; }
}
```

```

class Node
{
    public Node()
    {
        Parent = null;
        Child = new List<Node>();
        isRoot = false;
        value = null;
    }

    public void AddChild(Stop stop, Tree tree)
    {
        Node node = new Node();
        node.Parent = this;
        node.value = stop;
        Child.Add(node);
        tree.nodeList.Add(node);
    }

    public Stop value { get; set; }
    public bool isRoot { get; set; }
    public Node Parent { get; set; }
    public List<Node> Child { get; set; }
}

```

## Breadth-First Search

The general implementation of the BFS is as follows:

- Initial and Goal states are determined
- The solution tree is initialized. The algorithm does two things: searching a way from source the destination and meanwhile building a solution tree made of expanded nodes. Tree consist of nodes that contains "Stop" data and a parent pointer. Therefore, when the searched item is found, i.e. goal state, the tree simply calls the parent() function to reach the initial state while saving the traversed nodes into the solution list.
- Frontier is initialized.
- During the loop:
  - Frontier count is checked, loop terminated if 0. (No solution case).
  - A currentStop is dequeued from the frontier.
  - The routes passing from the currentStop is found in order to create child nodes of the currentStop.
  - Goal destination is checked, whether the currentStop is goal or not. If yes, the solution tree is traversed and solution is printed as explained above.
  - If currentStop is not the goal state, then every adjacent stop of the currentStop (found via the routes passing from the currentStop, by determining the index of the currentStop in those routes and retrieving the adjacent stops by incrementing and decrementing the index) is added to the frontier queue.

The main difference between tree and graph search are the use of explored list, which determines who goes into the frontier queue. In graph search, if a stop is explored or already in the frontier, it does not go into frontier again. Therefore reducing the running time dramatically when there are loops in the given data set graph.

## Complexity Analysis

Consider a branching factor  $b$ , that is every node has at most  $d$  children. When we illustrate an expanding tree, we get the following:

$L_0 = 1$  Node

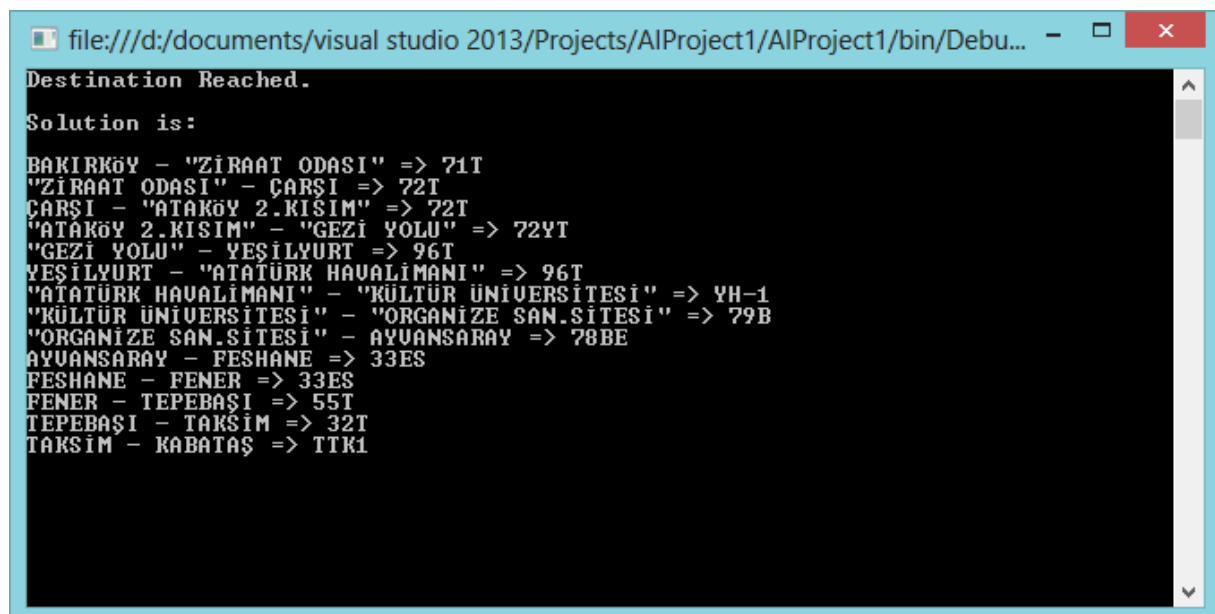
$L_1 = b$  Nodes

$L_2 = b^2$  Nodes

...

If the answer we are looking for is in the depth  $d$ , then  $b^d$  will be the dominant factor in the frontier. Hence BFS has a complexity of  $O(b^d)$  in both time and space.

As for the implementation, Graph-Search of BFS runs quickly when finding a path compared to tree-search of BFS. Actually, the tree-search algorithm causes an exception due to the huge memory consumption caused by repeatedly visited nodes while searching and expanding which are caused by the loops in the data set graph. Below is a screenshot for a solution for going to Kabatas from Bakirkoy using Graph-Search BFS while Tree-Search BFS is still calculating:



```
file:///d:/documents/visual studio 2013/Projects/AIProject1/AIProject1/bin/Debu...
Destination Reached.
Solution is:
BAKIRKÖY - "ZİRAAT ODASI" => 71T
"ZİRAAT ODASI" - ÇARŞI => 72T
ÇARŞI - "ATAKÖY 2.KISIM" => 72T
"ATAKÖY 2.KISIM" - "GEZİ YOLU" => 72YT
"GEZİ YOLU" - YEŞİLYURT => 96T
YEŞİLYURT - "ATATÜRK HAVALİMANI" => 96T
"ATATÜRK HAVALİMANI" - "KÜLTÜR ÜNİVERSİTESİ" => YH-1
"KÜLTÜR ÜNİVERSİTESİ" - "ORGANİZE SAN.ŞİTESİ" => 79B
"ORGANİZE SAN.ŞİTESİ" - AYUANSARAY => 78BE
AYUANSARAY - FESHANE => 33ES
FESHANE - FENER => 33ES
FENER - TEPEBAŞI => 55T
TEPEBAŞI - TAKSİM => 32T
TAKSİM - KABATAŞ => TTK1
```

Since BFS is not optimal, the solution is not applicable to real life.

Note: If a null exception occurs **after** the solution is found, then it is due to the multiple existence of a stop name with different ids.

## Depth-First Search

Since there are loops in the data set graph, tree-search dfs will generate infinite loops while increasing depth at each step, therefore will not terminate at all. For this reason, tree-search dfs is not implemented. Because graph-search dfs eliminates visited nodes by keeping them in the explored list, it eventually terminates given the graph is not infinitely large. Different than DFS, BFS uses stack instead of queue for frontiers. Until the stack becomes empty, or a solution is found, the `currentNode`'s children is pushed to the stack if they are not already explored or in the frontier. If

during the loop, no nodes are pushed to the frontier, i.e. the deepest node is reached, the top element is popped from stack and the loop continues.

### Complexity Analysis

Consider every node has a branching factor of  $b$ . Since only the child nodes of a given node is pushed onto the stack until no unique nodes are left to push, i.e. reached to bottom at depth  $m$ , the amount of nodes in the stack are at most  $b \cdot m$ . Therefore BFS has a time and space complexity of  $O(bm)$ .

Below is the resulting path of the same source and destination, Bakirkoy to Kabatas, using Graph-Search BFS.

You can see a missing link between lines 9-10. The route “MITHATPASA” to “PTT LOJMANLARI” is not specified. This is due to the existence of multiple “MITHATPASA” stops with different ids. Other than that, algorithm works fine. As it is seen from the output as well, the solution is not optimal, thus not applicable to real life.

## A\* Algorithm

A\* algorithm is not implemented due to awful usage of time **but it was given thought**. Different than DFS, A\* algorithm uses a priority queue based on the evaluated costs.  $f(n) = g(n) + h(n)$  where  $f(n)$  is the total cost from node  $n$  to destination,  $g(n)$  is the distance (actual cost) to get to node  $n$  and  $h(n)$  is the straight line distance to the destination node. In this particular Istanbul Route Finding case,  $g(n)$  is the distance between two stops calculated by using latitude and longitude and  $h(n)$  is calculated by the next stop's coordinates and the destination stop's coordinates. Now that each node has a total cost value determined by these distances, they can be prioritized. A min heap data structure would be used to implement this priority queue. At each step, the lowest cost path is chosen from the frontiers and nodes are expanded according to this. Therefore in the end, the optimal path (lowest cost) is found, ignoring the multiple existence of stops with the same name but different ids.

## Complexity Analysis

Since all nodes in the same level will not necessarily be expanded before the next level due to guided search, let's say that the effective branching factor is  $b^e$ . Consider the maximum depth that the solution is found  $d$ , the running time complexity becomes  $O(b^{eb})$ .