EECS 3401: Report 4 Documentation

Camillo John (CJ) D'Alimonte
(212754396 - cjdal34)
&
Dinesh Kalia
(213273420 - dinesh49)

November 19, 2015

Contents

1 Abstract

This report summarizes the methodologies used to complete the series of tasks provided in the *Report 4 Specification*. It provides a detailed analysis as to the reasoning behind certain chosen heuristics, the successful implementation of various predicates as well as any difficulties encountered throughout the assigned tasks.

2 Introduction

A mobile robot moves in the xy-plane among various rectangular obstacles aligned on the x and y axes where $x \in \mathbb{R}$ and $y \in \mathbb{R}$. The robot has to plan a collision free path between its current position and its given goal position. Since the robot's travel requires adequate energy, it aims at minimizing the path length, the number of changes in direction and the total number of stops. The costs are outlined below.

- Travel cost is 1 per unit
- A change of direction costs 1 unit of travel distance.
- A start/stop occur between each node and costs 2 units of travel distance.

The robot utilizes the A* and RTA* algorithms to find paths. The distance is calculated using two distinct methods. The formulas for each distance definition is provided below.

Euclidean Distance: $ED = \sqrt{(X1 - X2)^2 + (Y1 - Y2)^2}$
Manhattan Distance: $MD = |(X1 - X2)| + |(Y1 - Y2)|$

3 Design & Implementation

The following subsections outline the specific design decisions made during the implementation process of the program. The Professor had supplied us with various auxiliary files to assist in the development of our program.

3.1 Modification of *Bratko's* A* Algorithm

Bratko's original implementation of the A* Algorithm, as shown in Listing 1, was modified by renaming the starting point as **astar** so that the algorithm was invoked with the call **astar(Start, Path)**. The algorithm was also altered in order to be able to use any **h** predicate out of an unbounded collection.

```
bestfirst( Start, Solution) :-
expand( [], l( Start, 0/0),  9999, _, yes, Solution).
```

Listing 1: Bratko's Implementation of A*

The **hstar(N,H)** predicate was added in order to accomplish the new functionality requested. This predicate replaced the call **h(N,H)** with **hstar(N,H)** in order to be able to get different heuristics, regardless of specifics, for the algorithm. It is also worth noting that the **Solution** argument in **bestfirst(Star, Solution)** has been exchanged with **Path** in accordance with the Professor's instructions. The modifications are shown below, in Listing 2.

```
hstar(N,H) :-
    h_functor(H_functor) ,
    Function =.. [H_functor, N, H],
    call(Function).

astar( Start, Path ) :-
    expand( [], l( Start, 0/0), 9999, _, yes, RP),
    reverse(RP,Path).
```

Listing 2: Modifications to Bratko's Implementation of A*

3.2 Modification of *Bratko's* RTA* Algorithm

The only modification that needed to be made to the original algorithm was the ability to use the **h** predicate out of an unbounded collection. The new predicate is almost identical to the modifications of A* with the only change being the use of a more appropriate name for this particular algorithm. The altered implementation is shown below in Listing 3.

```
rta_star(N,H) :-
    h_functor(H_functor) ,
    Function =.. [H_functor, N, H],
    call(Function).
```

Listing 3: Modifications to Bratko's Implementation of RTA*

3.3 Main Body of Code

The first step in implementing the program was to develop the two distance equations that were used in making the robot move across the maze. The predicates were simply written with respect to the formulas outlined in the *Introduction*. The distance predicates are as follows:

```
euclidian(X,Y,X1,Y1,He) :-
    He is sqrt((X-X1)*(X-X1)+(Y-Y1)*(Y-Y1)).

manhattan(X,Y,X1,Y1,Hm) :-
    Hm is (abs(X-X1) + abs(Y-Y1)).
```

Listing 4: Distance Formulas in Predicate Form

The coordinates of three obstacles were chosen randomly although an emphasis was made to make sure that the obstacles did in fact make sense given the requirements of the program. The hard coded obstacles are listed below.

```
obstacle(2/1, 6/2).
obstacle(5/5, 6/8).
obstacle(4/3, 4/5).
```

Listing 5: Maze Obstacles

The next step was the implementation of the A* and RTA* algorithms. The RTA* algorithm was based off of the **rta** routine in *f13_10_RTA.pl* as well as our course slides. Two new functors,

Hfunc and **Goal** were created. Using the built-in predicate **retract**, the removal of the old h_functor(_) was implemented. **Asserta**, another built-in predicate, was used to add the new functor to the database which allowed the use of either the Euclidean or Manhattan based calculations. The old final goal was removed from the database and a new specified goal was added according to the same principles as **Hfunc**. The final step was to call the rta routine from the *f13_10_RTA.pl* file. The A* algorithm's implementation was identical with the only modification being the final call to the **astar** predicate from the *f12_3_Astar.pl* file.

The **s(State, NewState, Cost)** predicate was one of the most important predicates in the program. It controlled the progress of the robot as it searched for an appropriate path. The goal state had to determine the possibility of moving in all four directions: North, South, East and West. The implementation of the **s** predicate involved the calling of four other specific directional predicates. The main structure is the same throughout all four predicates with the only difference being the directional values (i.e. +1, -1 depending on direction heading). Both the **s** predicate as well as one of its auxiliary predicates, **s_east**, is outlined below.

```
s(Start_X/Start_Y/Start_Dir, End_X/End_Y/End_Dir, Cost) :-

goal(GoalX/GoalY/_),
(
s_north(Start_X/Start_Y/Start_Dir, GoalX/GoalY/_, End_X/End_Y/End_Dir, Cost)
; s_south(Start_X/Start_Y/Start_Dir, GoalX/GoalY/_, End_X/End_Y/End_Dir, Cost)
; s_east(Start_X/Start_Y/Start_Dir, GoalX/GoalY/_, End_X/End_Y/End_Dir, Cost)
; s_west(Start_X/Start_Y/Start_Dir, GoalX/GoalY/_, End_X/End_Y/End_Dir, Cost)
).
```

Listing 6: **s** Predicate

```
s_east(Start_X/Start_Y/Start_Dir, GoalX/_/_, End_X/End_Y/End_Dir, Cost) :-
        GoalX > Start_X,
        east_obs(Start_X/Start_Y, _/Miny, _/Maxy, Boolvalue),
        (
        Boolvalue = false,
        findAllEast(Start_X/Start_Y,GoalX,MaxX),
        End_X = MaxX, End_Y = Start_Y, End_Dir = east,
        (Start_Dir \= east , !, NewCost = 1 ; NewCost = 0),
        Cost is MaxX - Start_X + NewCost
        ;
        Boolvalue = true, End_X = Start_X,
        ( End_Y = Miny, End_Dir = south, Cost is Start_Y - Miny + 1
        ; End_Y = Maxy, End_Dir = north, Cost is Maxy - Start_Y + 1
    )
  ).
```

Listing 7: **s_east** Predicate

The **s_east** predicate checks if there is an obstacle on the X-plane, east of the robot's current position/state. If there is an obstacle, then the robot tries to move along one of its corners (i.e.: travel in the opposite axis and move along the boundary of the obstacle). If there is no obstacle impeding

the robot's path, then the robot continues to travel eastward. The assumption is made that the robot is dimensionless and is able to move between two obstacles if they are right next to each other.

Two auxiliary predicates are used for determining the movement in the east direction: **east_obs** and **findAllEast**. **east_obs** checks whether or not there is an obstacle to the east of the robot. If there is, then set the value of *Boolvalue* to true (false otherwise). **findAllEast** utilizes the built-in **findall** predicate in order to find all the obstacles in the direction (east) of the robot. The **findall** "sums up" the list of obstacles in that specific direction.

The implementation of a **run** predicate allowed for the testing of the program. The **run** predicate was used to find the path from a start node to a final, goal node. If the Algorithm specified is the **astar** routine then use **a_star** routine. If the Algorithm specified was the **rta** routine, then use **rta_alg** routine. The **run** predicate is listed below.

```
run(Algorithm,H_functor,Start,Goal,Path):-
        ((Algorithm = 'astar'), a_star(H_functor,Start,Goal,Path)) ;
        ((Algorithm = 'rta'), rta_alg(H_functor,Start,Goal,Path)).
```

Listing 8: **run** Predicate

4 Assumptions

The report specifications were often ambiguous and open to one's interpretation. An assumption was made regarding the last rule concerning the cost of a path. The requirement read "A start/stop occurs between each node in the path and costs 2 units of travel distance." This was assumed to mean that it costed 1 unit of time to get from the start node to the stop node and another 1 unit of time for changing directions. The sum of these actions resulted in a total travelling distance of 2 units.

5 Testing

Although no PLUnit test cases were written for this specific program, there was a pressing need to determine if the functionality of the implementation was correct. Therefore, command-line testing was used thoroughly to determine if the program behaved as expected while running the four specific algorithms. There were two distinct mazes used for testing purposes, as seen in Figures 1 (a) and (b).

(a)
Maze
1

(b)
Maze
2

Figure 1

5.1 Task 1 - A* Algorithm with Euclidean distance heuristic

The output pertaining to Maze 1 and 2 is listed below. The program behaves as expected and the output is correct.

```
1 ?- run(astar, h_e, 5/4/east, 1/1/_, Path).
Path = [5/4/east, 5/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/3/south, 1/3/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/5/north, 4/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/5/north, 1/5/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/3/south, 4/2/south, 1/2/west, 1/1/south] ;
false.
```

Listing 9: Testing on Maze 1

```
5 ?- run(astar, h_e, 3/1/west, 6/9/_, Path).
Path = [3/1/west, 6/1/east, 6/9/north] ;
Path = [3/1/west, 6/1/east, 6/9/north] ;
Path = [3/1/west, 2/1/west, 2/9/north, 6/9/east] ;
Path = [3/1/west, 2/1/west, 6/1/east, 6/9/north] ;
false.
```

Listing 10: Testing on Maze 2

5.2 Task 2 - A* Algorithm with Manhattan distance heuristic

The output pertaining to Maze 1 and 2 is listed below. The program behaves as expected and the output is correct.

```
2 ?- run(astar, h_m, 5/4/east, 1/1/_, Path).
Path = [5/4/east, 5/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/3/south, 1/3/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/3/south, 4/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/5/north, 4/2/south, 1/2/west, 1/1/south] ;
Path = [5/4/east, 4/4/west, 4/5/north, 1/5/west, 1/1/south] ;
false.
```

Listing 11: Testing on Maze 1

Notice the difference between Listings 4 and 6, specifically in regards to the order of the 4^{th}, 5^{th}, and 6^{th} paths. This contrast in positions resulted from the use of the two different types of distance calculations, the Euclidean and Manhattan distances.

```
6 ?- run(astar, h_m, 3/1/west, 6/9/_, Path).
Path = [3/1/west, 6/1/east, 6/9/north] ;
Path = [3/1/west, 6/1/east, 6/9/north] ;
Path = [3/1/west, 2/1/west, 2/9/north, 6/9/east] ;
Path = [3/1/west, 2/1/west, 6/1/east, 6/9/north] ;
false.
```

Listing 12: Testing on Maze 2

5.3 Task 3 - RTA* Algorithm with Euclidean distance heuristic

The output pertaining to Maze 1 and 2 is listed below. The program behaves as expected and the output is correct.

```
3 ?- run(rta, h_e, 5/4/east, 1/1/_, Path).
Path = [5/4/east, 5/2/south, 1/2/west, 1/1/south] ;
false.
```

Listing 13: Testing on Maze 1

```
7 ?- run(rta, h_e, 3/1/west, 6/9/_, Path).
Path = [3/1/west, 6/1/east, 6/9/north] ;
false.
```

Listing 14: Testing on Maze 2

5.4 Task 4 - RTA* Algorithm with Manhattan distance heuristic

The output pertaining to Maze 1 and 2 is listed below. The program behaves as expected and the output is correct.

```
4 ?- run(rta, h_m, 5/4/east, 1/1/_, Path).
Path = [5/4/east, 5/2/south, 1/2/west, 1/1/south] ;
false
```

Listing 15: Testing on Maze 1

```
8 ?- run(rta, h_m, 3/1/west, 6/9/_, Path).
Path = [3/1/west, 6/1/east, 6/9/north] ;
false.
```

Listing 16: Testing on Maze 2

5.5 Comparison of the Number of Generated Nodes

The A* algorithm features a uniform-cost search and a pure heuristic search to efficiently compute optimal solutions. It is a best-first search algorithm in which the cost associated with a node is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to node n and $h(n)$ is the heuristic estimate or the cost of a path from node n to a goal. Thus, $f(n)$ estimates the lowest total cost of any solution path going through node n.[1] The major drawback of A* however is its memory requirement as it must maintain, at the very minimum, the entire open list of nodes. This is why the number of generated nodes is higher in the test cases run on A* then it is on the

[1] Robin. *A* Algorithm.* December 18, 2009. World of Computing. URL: `http://intelligence.worldofcomputing.net/ai-search/a-star-algorithm.html#.Vk3XKr9UUhU`.

cases running RTA*. RTA* does not find a complete solution until near the goal and thus saves time.

The assumption was made that the term "generated node" meant all the different nodes that were displayed in the paths outputted during testing. The decision was made to add all the generated nodes of the **longest** path which was then recorded and stored in the tables below.

Algorithm	**Heuristic**	**Number of Generated Nodes**
A*	Euclidean Distance	6
A*	Manhattan Distance	6
RTA*	Euclidean Distance	4
RTA*	Manhattan Distance	4

Table 1: Comparison of the Nodes Generated in Maze 1

Algorithm	**Heuristic**	**Number of Generated Nodes**
A*	Euclidean Distance	4
A*	Manhattan Distance	4
RTA*	Euclidean Distance	3
RTA*	Manhattan Distance	3

Table 2: Comparison of the Nodes Generated in Maze 2

6 Difficulties Encountered

Overall, this program was extremely challenging and time consuming. In general terms, the most difficult part of the implementation was probably the development of the auxiliary predicates for the movement patterns in each specific direction. Not only did built-in predicates need to be utilized but a lot of focus was needed to make sure that no wrong movement patterns were being implemented (i.e. moving in the right direction when coming in contact with obstacles). Many mistakes were caught during testing where the robot moved in the wrong direction or in some cases, went right through the hard coded obstacles.

Note: Through out the program, built-in predicates were used that were a little difficult to understand, such as **asserta** and **retract**.

6.1 Counter Predicate

The only functionality that was not implemented was the Counter Predicate. The code for the predicate is below.

```
counter(C) :-
                C_Updated is C + 1,
                nl,
                write('Nodes Generated: '),
                write(C_Updated),
                nl.
```

Listing 17: **counter** Predicate

This code segment is not included in the submitted code. Although this predicate appears to be correct, the location of where to output the Count value was unknown. It was unclear how to output the value when the robot approached an upcoming node, regardless of where any obstacles were. Since the **Counter** predicate is not functioning, it was impossible to find the number of generated nodes computationally. Therefore, an assumption (outlined earlier) was used to try to calculate the number of visited nodes manually.

7 Appendix

This report was jointly authored. Both members worked on the outline of each question together as a pair. The programming of the predicates was jointly done. Dinesh wrote the documentation and comments for each predicate while CJ wrote the test cases for each exercise.

8 Bibliography

Robin. *A* Algorithm*. December 18, 2009. World of Computing. URL: `http://intelligence.worldofcomputing.net/ai-search/a-star-algorithm.html#.Vk3XKr9UUhU`.