Report 4

Sobota Dominik, Mierzwa Michael

October 21st, 2016.

Table of Contents

1. Introduction

Report 4 is designed to test our abilities to

- Implement an algorithm and analyze performance
- Writing and modifying of algorithms
- Writing reports

1.1 Task Division

The Task Division is a description of the work divided between the two authors of the report as well as the programming contributions that each of them made.

Michael:

- A* Modifications
- A* count
- RTA count
- Test Cases
- Report sections 1,2,3,5

Dominik:

- createEdges
- h_m and h_e
- RTA Modifications
- Test Cases
- Report sections 2,4

2. report4.pl

2.1 General Explanation

Report 4 first and foremost is the run predicate which takes in an Algorithm name, an H_functor, a grid size, a start node, an end node, and returns a path.

The thing that the run predicate does upon being searched, is to do one of two things:

1) Remove all previous grid_size from the database, then retract all successor functions s. It Then asserts (assertz) the grid_size as the given grid size, and calls a helper predicate createEdges to create the successor functions for a given grid size
2) Same as previous without removing grid_size or successor functions

Both of these options are in an if statement because if we fail to retract, then it will return false (meaning we did not have any rules to retract) and then go to the 2) option. The Key reason that this is important is that it removes any conflicting statements from previous calls to run. Simply put, cleaning the database.

This same technique is applied to the H_functor.

```
( retract(h_functor(_)) ; true ),  asserta(h_functor(H_functor))
```

This sections shows that it tries to remove any previous h_functor rules before asserting it's own. Note that the if statement follows the format “something ; true” meaning that it tries to execute the retract, but because of “;true” will still run if there is nothing to retract. This exact format is also applied to asserting the goal rule, but the difference is that after asserting goal, there is a cut.

The cut after asserting goal serves one main purpose, to ensure that once the database is done being populated, nothing can go and modify it to see if entering other values into the database could result in true. The cut makes it so that once passing it, any rule that fails will simply result in false.

Finally we assert Predicate =.. [Algorithm, Start, Path], and call it to execute our run, please note that at the end we added a cut to the end of these statements to ensure only one answer, removing the cut would allow the viewing of more answers.

2.2 createEdges

createEdges(GridSize).

createEdges asserts all the edge predicates, s(State, NewState, Cost), which will be used by the algorithms to compute the path to the goal, where GridSize represents the maximum column and row number of the grid. In order to assert the correct edges associated with each grid cell, we chose to traverse the grid cell by cell, asserting the proper edges in their respective places. This was done by creating createEdges(GridSize, CurrentRow, CurrentCol), which keeps tracks of the current cell we are looking at, and in there asserting the proper edges.
The successor function s(State, NewState, Cost) is the edge, where NewState is the next cell to travel to, and cost is the cost of getting into the next cell. The cell is then shifted to the next cell in the row via incrementing the current row and calling createEdges() with the new row number. Once at the maximum column, the next shift will increment the row number instead and reset the column number to 1. This is done until we reach the bottom right corner(where row and column are both equal to GridSize).

There are 9 cases to consider:

.

1	2	2	2	2	3
4	5	5	5	5	6
4	5	5	5	5	6
4	5	5	5	5	6
4	5	5	5	5	6
7	8	8	8	8	9

Figure 1
createEdges cases visualized
for a 6x6 grid

Top Row

1. **Upper Left Corner**
   The robot can move left or right from this point, as it is at the cell 1-1, therefore it cannot move left or above it as 1 is the minimum row and column number.

2. **Middle of first row**
   The robot can move left, right and below the cell.

3. **Upper Right Corner**
   The robot can move left or below.

Middle Rows

4. **First column, not the first or last row**
   The robot can only move right, above and below.

5. **In the middle of any row excluding the first row and the last row**
   The robot can move in all directions: right, left, above and below.

6. **Last column, not the first or last row**
   The robot can move left, above and below.

Bottom Row

7. **Bottom Left corner**
   The robot can move above and to its right.

8. **Middle of bottom row**
   The robot can move left, right and above.

9. **Bottom Right Corner**
   The robot can move left and above.

2.3 difficulty

difficulty(RowNumber, ColNumber, Cost).

difficulty asserts the Cost of entering a cell, by the formula given in the specifications.

$$Difficulty = \mathrm{mod}(row3 + column3,17)$$

The Cost is the result of the formula, which is then asserted as the edge cost for that edge.

2.4 h_m and h_e

The two heursitics to used in this report are:

1) h_e(State,H).

Which defines the euclidian distance from the robot's current cell to the goal cell.

$$ED = ((X1 - X2)2 + (Y1 - Y2))\text{^}(1/2)$$

2) h_m(State,H).

Which defines the Manhattan distance from the robots current cell to the goal cell.

$$MD = abs(X1 - X2) + abs(Y1 - Y2)$$

When run is called, the specific huersitc is specified and then asserted into the database via h_fucntor(H_functor).
The goal state is obtained when run is called as well, as it is asserted into the database via goal(Goal).

These huersitc values are used by the algorithms to determine the path the robot should take.

2.5 Test Cases

There were a variety of test cases used to validate our program. Most of the test cases come in sets of 4, each one of them testing the possible combinations of the astar and rta search algorithms with our two heuristics h_e and h_m.

The first set of tests is the “simpN” tests where N is replaced with the test number. These tests ensure that given a very short path, where the path is both the optimal and the shortest, it is found by both algorithms and heuristics

The second set “arbGoalN” checks if the path from the goal to the goal is only one node, which is the goal node.

The next set of tests are “countN” tests. These tests do not work for rta due to an improper implementation of update_count within rta. We test astar with a the same sample grid from “simpN” and then assert that after running the count is 4. Please note that the current implementation of count retains the count value of the previous run of the query, and is removed upon starting a new query, this functionality is tested by having two count related tests one after the other.

The given examples at the top of report4 are defined as “exampleN” and simply ensure that the given outline of test cases function properly.

The next set of 20 tests follows the format of “num_testN” where num is the current grid size in english, and N is one of the four tests in the set. Each of these tests uses atar, rta, h_m, and h_m in combination to ensure they work for large grid sizes. Note that “num” only goes up to nine due to prolog's stack not being large enough for a 10x10 grid in our implementation.

The next set of tests follow the format “failN” where fail numbers 1-4 test that an out of bound grid fails, and fail numbers 5-8 test that incorrect paths are not accepted. Please note that tests 3 and 4 are commented out due to the current implementation of rta causing infinite recursion on them. Also note that count does not function properly with these fail based tests, that is because count is only retraced when the search algorithm is called, if that fails, count will never be removed, resulting in multiple count values at the end of the set of tests.

.

3. A* Search

3.1 Removed and Modified

The first thing added to f12_3_Astar.pl is a count rule that is asserted as 0 at the beginning of the search (it also has (retract(count(_)); true) in front of it to ensure only one count rule is present). This is later used in the newly added update_count(Succ) rule. What update_count does is that when called it looks at the list Succ and proceeds to copy and remove the previous count, then add the length of Succ to the saved copy of count, then assert the new count to be the sum of the old count and the length of Succ. update_count is only called in case two where you expand leaf nodes.

The other core difference between the given Astar and the modified is the removal of the old heuristic rule of h(N, H). This is because in order to allow our program to use a heuristic that is defined by the query we have to use univ operators. What we add is

```
h_functor(H_functor),  Heuristic =.. [H_functor, N, H],
        call(Heuristic),
```

This first takes the H_functor from the run query and then ensures it is the functor used, we then use the univ operator to create a new heuristic that follows the same format as the removed one, but instead of calling h(N,H) we call H_functor(N,H). We then call this to simulate the removed rule.

3.2 Implementation problems

The main implementation problem was with count. The main problem I had run into was simply that count did not exist when I tried to call it from update count. Frustrated, I proceeded to modify f12_3_Astar_count.pl to use the given heuristic and then I changed report 4 to use it instead of Astar. Only then did count work. After some troubleshooting I realized that I had never asserted count at the beginning of the run, after doing the correct modifications f12_3_Astar.pl was able to properly use count.

4. RTA Search

4.1 Removed and Modified

Given the rta file, we modified it to work with our predicates defined in report4.pl. In order to allow the algorithm to use any heuristic function, we had to change the heuristic code in the file f13_10_RTA.pl.

In order to change the heuristics of the file, we modified the previous h() heuristic and removed it in order to call the specific h_functor that is given in run.
As run asserts the specific heuristic functor needed for the algorithm, we extract it using h_functor(), and then used the univ operator in order to create a predicate of the form h_e(State,H) or h_m(State,H).
For example the code below shows that Heuristic will contain will be able to return the cost of the ChildNode depending on which functor it is given. I

```
/*** Before ***/
        h(ChildNode, H1)
/***After****/
        h_functor(H_functor),
                Heuristic =.. [H_functor, ChildNode, H1], call(Heuristic),
```

In order to produce only one solution, we introduced the cut in the run predicate, after rta(Start,Path) is called. The first solution shown is the cheapest and shortest path that the algorithm produces, while the succeeding paths are longer and are more expensive
Note that even though it is the shortest path that rta produces, it is not always the shortest possible path.

4.2 Implementation problems

We were unsuccessful in implementing the counter for the rta algorithm We had the update_count predicate working correctly, as it would update the count of nodes and asserts the current count via the count() term. We could not find the proper places in f13_10_RTA.pl to place the update_count in order to update the count.

.

5. Efficiency

5.1 A* efficiency

The key thing about Astar is the fact that it must find the shortest path first, this ends up making it generating many paths and exploring many nodes in order to ensure this feature. That is why the core feature that allows it to find more correct paths, the heuristic, should be as optimal as possible. That is why we have to look at the heuristics presented and determine which allows Astar to be more efficient.

Given an example grid of 5x5 with a start at 1-1 and a goal at 5-5 we find that:

- h_e generates 774 nodes
- h_m generates 691 nodes

These numbers suggest that h_m (the Manhattan distance) is more effective, but why? This is probably because of the fact that the cost of h_e is higher on certain nodes.

Take a look at the paths from h_m and h_e

- h_e generates [1-1,2-1,3-1,3-2,3-3,3-4,4-4,4-5,5-5].
- h_m generates [1-1,2-1,3-1,3-2,3-3,4-3,4-4,4-5,5-5].

Notice that the point where the two differ is at 3-3. Why is 3-3 special? It is because h_e looks at more paths because of how it is defined, while h_m simply choses the path with the least overall cost. This difference in path checking does not happen very often, in fact in only starts to appear with a grid size of at least four, but as the paths to check get longer, h_e make astar consider more paths than it needs to, while h_m does not.

This means that h_m is a better heuristic for Astar.

5.2 RTA efficiency

Given that we did not properly implement count we can only speculate the run times of h_m and h_e on the RTA search algorithm. Given that RTA searches for the fastest path, it does not consider nearly as many nodes and simply generates a solution. Since the difference between h_m and h_e took more than a hundred nodes to show, it may not even be noticeable in the RTA algorithm.

However, even though the amount of counted nodes would be smaller in RTA, h_m should still be more efficient as grid size increases.

5.3 Conclusion

Astar and RTA are both search algorithms that depend very much on the heuristics given to them, that is why this reports concludes that overall, h_m, otherwise known as the Manhattan distance as a heuristic is the more efficient heuristic for these searches.

End of Report