Using Heuristic Search to Solve Planning Problems in GROOVE

Erik Snippe
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
e.snippe@student.utwente.nl

ABSTRACT

Planning is a part of artificial intelligence that concerns strategies for solving difficult problems that have no clear algorithmic path to a desired goal. For computer models that generate a big search space it is often difficult to find a path to a specific state because of all the different options given. Planning can help here by handing the strategies to find a path easier and faster. GROOVE is a graph transformation tool which can be used to model problems and then generate the search space for these problems. However, GROOVE currently does not support the functionality of a planning algorithm. In this research we will investigate the impact of a famous state space exploration algorithm on certain problems in GROOVE.

Keywords

Planning, graph transitions, GROOVE, heuristic search, A* algorithm

1. INTRODUCTION

Planning tries to find a path from a given initial state toward a desired goal state using a series of possible given actions. This can be difficult because often the path is not very clear and the amount of options can be quite big. This results in a large search space and requires a strategic approach to find the easiest path toward the goal. Consider the Rubik's Cube for example. You can twist it all day randomly and never get it back in its solved position. For each twist you have 18 possibilities (6 sides and 3 possible new positions for that edge) and it is not very clear which twist will most likely lead to the solved state. A lot of possibilities will only shuffle the colors even further instead of bringing you closer to the desired solved state. There has been some preliminary work in this direction, see Edelkamp and Rensink [1].

1.1 GROOVE

GROOVE [3] is a Graph Transformation tool developed at the University of Twente for modeling graph transitions. By modelling the start situation and the possible transitions, GROOVE can be used to model planning problems. Using this model GROOVE can explore the entire search space in order to find a desired result. To do this it either uses a Depth First Search (DFS) approach or a Breadth-First-Search (BFS)approach. Only BFS guarantees to find the optimal solution.

1.2 Background: GROOVE

Here we will explain the basic functionality of GROOVE using the Ferryman problem. In this problem we have a ferryman who has to transport a cabbage, a goat and a wolf from one bank of the river to the other bank. The problem is that he can only transport them one at a time. Also, should the ferryman leave the goat and the cabbage alone on one riverbank, then the goat will eat the cabbage. In the same way the wolf will eat the goat, if given the chance.

Host Graph

In Figure 1 you can see the initial state of this problem which is represented as a model. The two banks at the center represent the two river banks. It also shows the boat of the ferryman which is "moored" at the left bank. The cabbage, the goat and the wolf are also "on" the left bank. Lastly you can see that the goat likes the cabbage and that the wolf likes the goat.

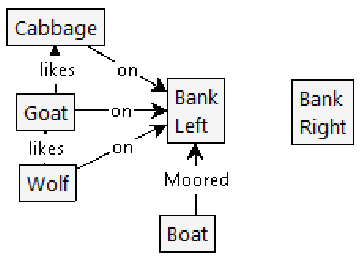

Figure 1. The Graph model for the Initial state of the Ferryman Problem

Graph Rules

Figure 2 shows an example of a Graph Rule for this problem. This transition was called "Transport" and it moves both the boat and one other object from one of the banks to another. The empty block means it can be anything as long as it has an "On" relation with the bank. The dotted line shows a connection that is required for this transition to take place but is also to be removed by this transition. In this case the boat needs to be moored at a bank and the object needs to be on the same bank. The bold lines show a connection that is to be created by this transition. In this case both the boat and the object are to be connected to the other bank with the same relationship. Both banks are connected by a "!=" relation, which basically means it can't be the same bank. The end result of this transition is that the boat and the object are placed on the other bank.

The rest of the rules are designed in a similar fashion.

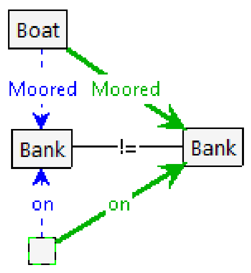

Figure 2. A Graph Rule for the Ferryman Problem

State space

Finally Figure 3 shows the state space for this problem as it is explored by GROOVE. First of all, it shows the initial state as seen in Figure 1 as state S0 at the top of the state space. It also shows the applications of the graph rules as transitions between the several states. The transitions are named after the rule that was applied. The dark states are final states in which GROOVE cannot apply any other rule. Should the state be called “Eat” then that means that one of the objects was eaten, while the “Finished” state means that a solution to the problem was found.

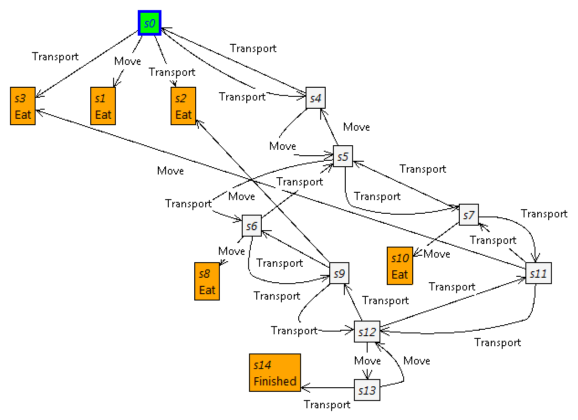

Figure 3. The State space for the Ferryman Problem

2. PROBLEM STATEMENT

As seen in the Ferryman problem GROOVE needs to explore the entire search space in order to solve the problem. In other words there is no intelligent exploration strategy implemented in GROOVE. With planning problems the amount of possible states that need to be explored can grow exponentially to the point where GROOVE cannot handle it. To solve this problem a heuristic search method is required. A heuristic search method will always explore the option that will most likely result in a desired result and will ignore options that will probably not lead to a desired state.

2.1 Research Questions

We aim at answering the following research question:

What heuristic search method is most suited for GROOVE and how do we use it as part of the GROOVE exploration strategy?

In the literature [4] the A* algorithm [2] is often used to tackle planning problems. Using heuristics it estimates the distance between any state and the desired goal state. From the open and unexplored states it will choose the state with the lowest estimated distance and explore that node. Any child state from that state are added to the list of open and unexplored states. This process is repeated until eventually either the goal state is reached or there are no open and unexplored state left. In the latter case this means that the goal cannot be reached. The main problem with this algorithm is writing a good estimation formula for the distance between a given state and a goal state. We intend to implement this algorithm into the GROOVE functionality and then test this impact of this algorithm on GROOVE searches.

3. A* ALGORITHM

The A* algorithm is a heuristic search algorithm that tries to find the shortest path between a start situation and a desired goal situation. It does this by keeping track of two variables for each found sub-state. The first is the shortest distance between the current state and the start state and the second is an estimated distance between the current state and the desired goal state. For as long as the algorithm has not found the goal-state the algorithm will keep on exploring the states preferring the states with the lowest total distance to both the start and the goal state. The pseudocode for the algorithm can be found in the appendix.

The A* algorithm uses a heuristic function to estimate the distance between a given state and the goal-state. For the algorithm to work properly two restrictions have to be placed on this distance function. First of all the distance function needs to be admissible. This means that it should never overestimate the distance towards the goal. It does not matter if the distance is underestimated, which will be the case most of the time, but in order for the A* algorithm to find the shortest path towards the goal this distance should not be over estimated. The second restriction is that the heuristic function should be consistent. This means that the estimated distance of a parent state should always be equal or smaller than the distance between the parent and a child state and the estimated distance of the child. As a result no child node will ever have a smaller total distance (distance from the start and estimated distance to the goal) then its parent. Because of this the start state will always have the lowest total distance and the nodes are always explored in increasing total distance. Should a shortcut be found and the distance from the start be updated for a node then we can know for certain that that node is not yet explored and we should not have to update all the child nodes of the given node. This is because if the node has already been explored then that would mean that the previous total distance is smaller than the total distance of the parent, which should not happen because of the consistency restriction.

4. ADJUSTING A* TO GROOVE

GROOVE uses several exploration strategies to search a the state space. The basic strategies are DFS and BFS. However these strategies only decide in which order the nodes of the search tree are explored. The goal of the search, which can be either exploring the entire state space or halting when a specific goal is reached, is not part of the exploration strategy. When implementing the A* algorithm as a GROOVE strategy we decided to give the user the option to enter a goal as the target for the A* search but the user will have to give the command to GROOVE to stop searching after finding the same state. The goal of the A* search can be chosen from the rules of the GROOVE model and is considered reached if the rule has been applied on the model.

GROOVE has a general template for search strategies which uses two basic functions. First of all there is the next function. This function does the basic exploration of the next node that is to be explored. Should this function find any new child states then it will try to add this node to the search list using the putInPool function. The next function will use the updateAtState function to prepare the next state for exploration. This function will use another function, by the name of getFromPool, to retrieve the next state to be explored. The putInPool and getFromPool functions are the functions from the template that need to be overridden in order to create our own search list for the nodes. Both functions use the GROOVE structure class PoolElement to store the information of a node. Because these PoolElements cannot store the distances we wanted we first created a structure of our own build around the PoolElements so we can store our own information. After that we created a Comparator for our own structures so we could use these two to store them in a Java PriorityQueue.

Next, for updating the distances to the start, which A* does in the later part of the algorithm, we created the updateState function. At first it may seem that this function will recursively visit all the children of all the children of the current node but because of the check at the base of the function it will only check nodes that have already been found and entered into the PriorityQueue. Also because of the consistency requirement discussed earlier we know for certain that the node has not yet been explored. A child node is required to have a bigger total distance than its parent and a node that has already been explored should have a smaller total distance. At this point we left the heuristic to the specific testcases. The pseudocode of the GROOVE implementation can be found in the appendix.

Another point to keep in mind when adjusting the A* algorithm for GROOVE is that the search tree that GROOVE produces is non-weighted. It can take several steps towards the goal but these steps do not have a value added to it. Therefor we decided to count the amount of steps GROOVE has to take to reach the node and weighing them all steps as one.

5. GENERIC HEURISTIC TEMPLATE

Should a programmer want to create his own heuristic to match a problem he wants to solve, then the current framework is suitably adaptable to fill this need. After entering the problem in GROOVE one would have to create a new subclass of the AStarStrategy class which is in the groove.explore.strategy package. There are only two functions that one needs to override for their own heuristic function. The first function is the createGoal function. This function is used to get all the information that is needed from the goal rule, which is given as an argument to the function, and store this information for later use. Different problems may need different information from the goal and this function is meant to extract every that information so it can be used later for calculating the heuristic distances.

The second function that one needs to override is the heuristicDistance function. This function gets a Graph as an argument and is used to evaluate the distance towards the goal for the given graph. Here the information previously stored about the goal state is used. When writing this heuristic function it is advised to keep the admissible and consistency requirements in mind, this to ensure the efficiency of the A* search.

Should the given heuristic not be admissible then it may happen that the goal state is overshot and a less efficient path can be found towards the goal. Should the heuristic function not be consistent then the risk is that a shorter path is found towards a state that has already been explored. This will result in the update of all the children of the node and also its children until potentially half the search tree is updated. This could possibly be very time consuming, especially when it happens often during the search.

It is worth noting that A* will keep on searching for the goal state, independent of how good or bad the heuristic function is written. It will always continue exploring new states until either all the states are explored or the desired goal state has been found. There are just no more guarantees over the efficiency of the search and the path that will be found.

The final thing that must be done when writing a A* strategy is adding the strategy to the GROOVE StrategyEnumerator. This class can be found in the groove.explore package. The strategy can be added to the list similarly to the other existing strategies.

6. EVALUATION

For testing the impact of the A* algorithm we wanted to test it on two different heuristic function. First of all we wanted a heuristic function that was general and could be used on many different problems without the need for extra programming. Next we wanted to create a specialised heuristic function dedicated on a single problem. For the problem that we wanted to test it on we decided on the BoxWorld problem. This is a famous planning problem in which we have a factory hall filled with several stacks of named boxes. We want to re-order these boxes in a predefined order. To do this we have a machine which can take the top box from any one given stack and place it either on another stack or on the floor.

6.1 Generic Heuristic Solution

For the first part of the tests we wanted to create a generic heuristic function that could be used on a variety of problems. For this function we decided to compare the start graph of the goal rule and the current graph and count the similarities between these two graphs. Using the createGoal function we made a list of all the nodes in the goal graph and a list of all the edges the graph has. This information is used in the heuristic function where it counts all the nodes and edges which are in this list but not present in the graph from which the user wants the heuristic distance. We assumed that for every node and/or edge that is not present at least one step has to be made before the edge or goal is present. The heuristic distance would then be the amount of nodes and edges of the goal state that are not yet present.

Should two graphs have the same amount of similar nodes and/or edges we decided that we wanted to favour the node with the most options left. Because of this we decided to count the amount of child nodes the node has and deduct a tenth (0.1) step from the estimated distance for every child node.

It should be noted that this is, most likely, not a proper A* heuristic. We can give no guaranties that this heuristic is admissible and consistent because we can not say how much a single rule of a given problem might do and how much closer it may bring the problem towards its solution. Also there is no guaranty that the goal state is very compatible with this solution. For example, should the goal be that Box C is on the ground in the Boxworld problem then this will either hold true or not and the heuristic does nothing but state whether this truth holds.

Because one can say next to nothing about the problem that the generic heuristic needs to solve we concluded that a proper heuristic is not possible and that we should opt for the next best thing. We wanted to create a heuristic solution that should be on par with BFS ad DFS in most generic cases and preferably better in some.

6.2 Customised Heuristic Function

For the customised heuristic function of this problem we decided to count the amount of boxes that are currently not in the right place. A box is in the right place if and only if the box is on the proper box and that box is also on it's proper place. Boxes that are supposed to be on the ground are considered on their proper place if they are on the ground. For all the boxes which are not on their proper place we will add 1 to the total distance if the box is somewhere in the middle of the stack and 0.9 if the box is on the top. This will make top boxes a bit more favourable and should GROOVE not know exactly what to do then it will start placing more boxes on the ground.

6.3 Experimental Results

For testing the two different implementations, as well as comparing them to DFS and BFS we let GROOVE rearrange a factory hall with varying amounts of boxes using all four algorithms. We tested with 6, 9 and 16 boxes. These are stored in stacks of maximum size 3 for the first two and 4 for the last. For the goal we picked 3 semi-random hall arrangements for the goal per hall size. The criteria we tested the algorithms on are the amount of nodes that GROOVE needs before finding the desired goal, the time it takes to do this and the path length of the found path. The last one is omitted the DFS tests because it will, most likely not produce a very efficient path.

The time of all of the tests is measured in milliseconds and was taken from an average of 5 runs. For these testruns the testcase was tested 6 times in total and the first result was always discarded. We did this because there were several testcases, but not all, which produced a significantly higher time on the first run. We assume that this is because JAVA has to load the settings of the new testcase on the first run and uses this on the later tests.

The biggest problem with large testcases is that GROOVE cannot properly handle very big search spaces. After a certain amount of explored nodes GROOVE will slow down drastically until it almost seems to halt. When the test is aborted at this point GROOVE will report an OutOfMemoryError in the Java Heap space. Because of this problem all tests are aborted after 40.000 nodes have been explored and marked with “Heap Overflow” should the error indeed appear.

Next we decided to test the time efficiency of both new heuristic functions. The best way to do this is by exploring the exact amount of nodes as BFS and DFS. We decided to do this by making a GROOVE exploration without setting a goal state. In this way GROOVE has to explore the entire state space and with that every node. This test is done in a small BoxWorld of 6 boxes in order to minimise the chances of a heap overflow. Because A* needs a goal state to find we enabled a goal rule in all tree tests but because we didn't set the end point GROOVE will continue exploring after that until the entire statespace is explored.

All tests were performed on the personal laptop computer of the author. Results may vary depending on the computer used for testing and tests performed on computers with more memory may be able to tackle bigger problems without the OutOfMemory error seen in these tests. Nevertheless we believe that for comparison these tests suffice.

6.4 Test Results

The results of the first testcases can be found in Tables 1 through 9 in the Appendix. As can be seen in the testcases with only six boxes the A* Algorithm is significantly faster then both DFS and BFS and does this by finding and exploring significantly less nodes then the other two algorithms. Though it should be noted that all three tests were performed within 3 seconds on average and therefore any human testing it will most likely not care much for the difference.

When the scale is increased to 9 boxes a more significant detail comes up. Both BFS and DFS will have to explore more states then the heap could hold on. This while A* still finds results within half a second. When increasing the size to 16 we find that the specialised A* algorithm still has no problems finding a solution whereas DFS, BFS and the generic A* algorithm have their heaps overflown.

From this test we can conclude the following. First of all both the generic and the specialised A* were faster and more efficient then BFS and DFS. Secondly, and more importantly, because of the heap overflows the efficiency of A* is one of its better strengths. Because A* has to explore a lot less states it can tackle bigger problems before it will, most likely, overflow the heap.

During the second testcase we let GROOVE explore the exact same amount of nodes using BFS, DFS and both A* algorithms. The testresults can be found in table 10 in the Appendix. In this test case we can see that when exploring a fixed amount of nodes we can clearly see that A* as a lot slower then the previous methods. This was to be expected but is not much of a problem when A* is the more efficient algorithm. However on this point it should be noted that is a time inefficiency problem with the generic A* algorithm. During the 16 boxes testcases from the previous tests, which caused a heap overflow, did take significantly longer to overflow then BFS and DFS tests. Because time measuring is highly inaccurate when testcases are cut short arbitrarily it should be said that the BFS and DFS were usually cut short after about 40.000 nodes explore which takes about a minute of runtime. The generic A* algorithm was also cut off at around this same amount but took at least 6 minutes to reach this point.

7. CONCLUSIONS

We set out to find and measure the impact of a famous heuristic search algorithm, namely the A* algorithm, on a graph transformation tool like GROOVE by testing it on the famous planning problem BoxWorld. All of our test results proved that the A* search approaches were a lot faster and more efficient than the existing Breadth-First-Search approach and the Depth-First-Search approach. Because of this the A* approach could tackle bigger problems than BFS and DFS before it faced the slowdown problem which GROOVE has.

We also set out to create a good template on which programmers could build their own search strategies. By mapping out the goal state of the search and comparing it with a given graph we managed to create a decent base that was already better at solving the BoxWorld problem than BFS and DFS. Given a different problem a programmer should be able to easily adjust this structure to fit his own needs.

.

8. REFERENCES

[1] S. Edelkamp and A. Rensink. Graph transformation and ai planning. In *Knowledge Engineering Competition (ICKEPS)*, Sep 2007.

[2] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.

[3] A. Rensink. Isomorphism checking in groove. *ECEASST*, 1, 2006.

[4] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

APPENDIX

A. A* ALGORITHM PSEUDOCODE

```
Function A*(start, goal)
   closedset := new Empty Set
   open set := new Set containing start
      node
   start.distToStart := 0
   start.distToFinish :=
      heuristicDistance(start, goal)
   start.distTotal := start.distToFinish
   While (openset is not empty)
   {
      x := node from openset with
         lowest distTotal
      if x == goal
         finish
      remove x from openset
      add x to closedset
      foreach y in neigbours(x)
         tenativeDistToStart =
            x.distToStart +
            distance(x,y)
         if y is not in openset or
            closedset
            add y to openset
            betterTenative := true
         elseif tenativeDistToStart <
            y.distToStart
            betterTenative := true
         else   betterTenative := false
      if betterTenative == true
         y.cameFrom := x
         y.distToStart :=
            tenativeDistToStart
         y.distToFinish :=
            heuristicDistance(y, goal)
         y.distTotal := y.distToStart +
            y.distToFinish
   }
   return pathNotFound
```

B. A* GROOVE PSEUDOCODE

```
class AStarStrategy extends
    ClosingStrategy
{
   HashMap startingDistances =
      Hashmap<Node, int>
   PriorityQueue queue =
      PriorityQueue<AStarStruct>

   //Overriden functions
   void prepare(startstate) {
      super.prepare(startstate);
      startingDistances.add(startState,
         1);
   }

   void next() {
      currstate = updateAtState();
         //global function that give
         the current state
      super.next();
      updateChildren(currstate,
         startingDistances.get(currState)+1);
   }

   putInPool(PoolElement) {
      state = PoolElement.getState();
      queue.offer(new
         AStarStruct(PoolElement,
         HDist(state));
      startingDistances.add(state, 0);
   }

   getFromPool() {
      return queue.first();
   }

   //new functions

   updateChildren(state, tentDist){
      for (all children from state)
         if (startingDistances(child) =
            0 or > tentDist) {
            UpdateChild in queue
            startingDistances.add(child,
               tentDist);
            updateChildren(child,
               tentDist+1);
         }
   }

   void SetGoal(Rule) {
      //sets Goal according to heuristic
   }

   int heuristicDistance(state) {
      //calculate heuristic distance
         for state
   }
}
```

C. TEST RESULTS

Table 1. Test Results: 6 boxes, Case A

Algorithm	Explored States	Time	Path
Generic A*	112 states	93	6
Specialised A*	46 states	15	6
DFS	4032 states	2917	-
BFS	2282 states	1089	6

Table 2. Test Results: 6 boxes Case B

Test	Explored States	Time	Path
Generic A*	117 states	143	6
Specialised A*	41 states	31	6
DFS	3940 states	2409	-
BFS	2491 states	1264	6

Table 3. Test Results: 6 boxes, Case C

Test	Explored States	Time	Path
Generic A*	120 states	103	10
Specialised A*	55 states	28	8
DFS	4038 states	2880	-
BFS	3890 states	2900	8

Table 4. Test Results: 9 boxes, Case A

Test	Explored States	Time	Path
Generic A*	168 states	181	8
Specialised A*	111 states	25	8
DFS	Heap Overflow	-	-
BFS	Heap Overflow	-	-

Table 5. Test Results: 9 boxes, Case B

Test	Explored States	Time	Path
Generic A*	951 states	1229	12
Specialised A*	117 states	41	10
DFS	Heap Overflow	-	-
BFS	Heap Overflow	-	-

Table 6. Test Results: 9 boxes, Case C

Test	Explored States	Time	Path
Generic A*	1147 states	1588	13
Specialised A*	170 states	47	12
DFS	Heap Overflow	-	-
BFS	Heap Overflow	-	-

Table 7. Test Results: 16 boxes, Case A

Test	Explored States	Time	Path
Generic A*	Heap Overflow	-	-
Specialised A*	1320 states	421	26
DFS	Heap Overflow	-	-
BFS	Heap Overflow	-	-

Table 8. Test Results: 16 boxes, Case B

Test	Explored States	Time	Path
Generic A*	Heap Overflow	-	-
Specialised A*	1308 states	381	25
DFS	Heap Overflow	-	-
BFS	Heap Overflow	-	-

Table 9. Test Results: 16 boxes, Case C

Test	Explored States	Time	Path
Generic A*	Heap Overflow	-	-
Specialised A*	862 states	212	21
DFS	Heap Overflow	-	-
BFS	Heap Overflow	-	-

Table 10. Test Results: 6 Boxes, No Goal

Test	Explored States	Time
A* Generic	4051 states	5360
A* Specialised	4051 states	4958
DFS	4051 states	4537
BFS	4051 states	4592