**02180**

**Introduction to Artificial Intelligence**

**Heureka 2017**

*Loïs Lano (s166388)*

*and*

*Sandra Griffon (s166341)*

Abstract:

Introduction to artificial intelligence project which aim at applying heuristic search methods to route finding and inference engine for propositional logic problems.

Content list:

1. Introduction
2. The algorithms: A\* and RBFS
3. Backward chaining algorithm and frontier
4. Graph building
5. Route Finding
6. The problem’s specifications
7. Creation of a knowledge base
8. Results
9. Inference Engine for Propositional Logic
10. The problem’s specifications
11. Creation of a knowledge base
12. Results
13. Conclusion
14. Introduction

This exercise aim at applying heuristic search methods on a map and on a knowledge base of clauses. Two kinds of algorithms are employed: A\* and RBFS. The language used for the project implementation is C++. The main challenge is to provide an algorithm that can be reused to tackle two different problems: route finding in a map and inference engine for propositional logic.

1. The algorithms: A\* and RBFS
2. Backward chaining algorithm and frontier

First, these two algorithms are both implemented by the backward chaining algorithm. The difference is the sorting that happens in the frontier.

A\* is an adaptation of the breadth first Dijkstra algorithm with a heuristic. The frontier is sorted when implemented as a *priority\_queue*. Therefore, an element with high priority is handled before an element with low priority. The priority depends on the distance between the start and current points.

RBFS is a depth first algorithm with a heuristic so the frontier is implemented with *stacks.* The first element in is also the first element out of the frontier.

As said before both algorithms are instances of the backward chaining algorithm but with different kinds of frontiers. Hence the use of templates is practical in order to be able to use backward chaining for different algorithms as well as different kinds of states. Each problem is represented by a graph where vertices are states and edges are the possible transitions between states. The vertices have a method *isStart* that indicates, through the graph, if that state is the starting state, the state to reach in the backward chaining algorithm.

The backward chaining algorithm needs an initial state to begin with, which is added to the frontier during the initialisation. Then, and until the frontier is empty, its top element is picked up at each iteration of the algorithm. If this element is the final one (denoted by *isStart)*, the algorithm returns *true*. Otherwise, the successors of the current vertex are added to the frontier if they haven’t been visited yet. The current vertex is removed from the frontier and added to the set of explored vertices. If the frontier gets empty before reaching a final element, then the problem has no solution, and the algorithm returns *false*.

In the backward chaining algorithm template, the *typenames* *State* and *Frontier* are used. *Frontier* is the type of structure we want for the frontier (for instance: *priority\_queue, stack* or *queue*), depending on the kind of algorithm we need. *State* is the type of vertices that are in our graph, depending on the problem we want to solve.

1. Graph building

A map is represented by a graph made of crossings and roads, whereas a knowledge base is represented by a graph made of clauses and if-rules. Here, we decided to use templates again, in order to be able to define different type of vertices and edges. A general edge is a structure that inherits from the specific edge structure corresponding to the problem we are dealing with. It can be a road, that has a name and a distance, or an if-rule. Similarly, a general vertex is a structure that inherits from the specific vertex structure used for that problem. For instance, it can be a crossing with coordinates or a clause containing a set of literals. The vertex method *successors* returns the set of accessible vertices as a vector. A vertex also has the method *edgeTo*, which returns the edge from that vertex to its successor. Edges are stored in vertices as vectors. Finally, vertices can have a *parent* vertex, used to find the shortest path of the solution, and a *weight* which is the distance of the shortest path from the initial state to the current vertex.

All things considered, the graph can be built with the set of all the vertices in the problem. The starting vertex of a graph can be set via the method *startAt*. The graph method *makePath* returns the path from the starting point to the initial state found by the backward chaining algorithm instance. Hence, this method can only be used after the algorithm is run, because the parents of every vertex have to be known in order to deduce a path.

1. Route Finding
2. The problem’s specifications

The route-finding problem consists of finding the shortest path from one street crossing to another. In this section, a concrete problem is tackled. A map is a graph where the vertices are street crossings and the edges are the actual streets, or roads, between crossings.

Therefore, we need a *Crossing* structure to specify the vertex type. It contains coordinates (x and y) and a few methods for convenience. We also need a *Road* structure to specify the edge type. It contains the name of the street and its distance, or length. We can then define the *Map* type as a graph with vertices inheriting from crossings and edges inheriting from roads. The algorithm is then usable for route-finding problems.

1. Creation of a map

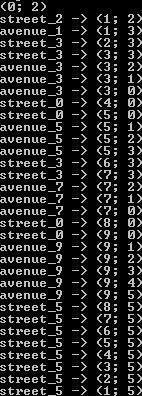
A map is still needed to run the algorithm. We have a text file containing all the necessary information to build the map. For instance, the first line of the file, *“0 0 street\_0 1 0”* represents a road: the first and last couples of integers are the coordinates of both ends. The street name is in the middle. Every such line describes only a one-way road, so the first pair is the starting point, and the other one, the road’s end. Two-way streets have to be described by two lines in the file. For instance, in the same file, we can also find the line *“1 0 street\_0 0 0”.* That means the road *street\_0* is two-way.

The file is read twice. A first time to build the set of vertices, where each new pair of integers represent a new *Crossing* object, and a second time to build the edges for each vertex. The length of each street is also computed for initializing the corresponding edge. In the end, we have the complete vertex and edge sets that we can use to create the graph.

1. Results

After setting the starting point in the map, we can run the chosen algorithm and get back the path it found*.*

We ran the RBFS algorithm to find the shortest path form (0; 2) to (5; 1)



This path is obviously not the shortest one, which should be:

0 2 avenue\_0 0 1

0 1 street\_1 1 1

1 1 street\_1 2 1

2 1 street\_1 3 1

3 1 street\_1 4 1

4 1 street\_1 5 1

However, since the heuristic hasn’t been implemented, this result is not surprising: it is effectively the result of a depth-first search.

For the A\* algorithm, we faced another problem. The algorithm finds a short path, but it invents edges that do not exist. The algorithm found a solution, but the path seems to be broken.



1. Inference Engine for Propositional Logic
2. The problem’s specifications

The problem addressed is that of finding a proof of some goal clause from a knowledge base. The knowledge base used is a list of if-rules between clauses, where clauses are sets of literals in disjonction.

Here, the vertices are clauses, and the edges are if-rules. The *Clause* structure describes vertices. It contains a set of literals. The *IfRule* structure describes the edges and contains the “name” of the rule. Similarly to the previous part, vertices inherit from clauses and edges inherit from if-rules.

1. Creation of a knowledge base

Again, we have a text file containing the knowledge base. An if-rule is represented as a line *“a if b c”.* Here, the head of the if rule is the clause “a” and the tail contains both clauses “b” and “c”. The rule’s name is simply *“a if b c”.*

The file is again read twice: the first time to get the list of clauses, and the second time to build the if-rules of each clause. In the end, we get the complete vertex and edge sets that make up a *KnowledgeBase.*

1. Results

We can run the algorithms the same way we did in the previous problem. After setting the starting point of the KnowledgeBase, which is basically the empty clause, we can run the chosen algorithm and get back a path.

The issue is that the program crashes during the backward chaining algorithm, even though this algorithm worked fine previously. Thus, we do not have any result to show.

1. Conclusion

In order to be able to tackle several kinds of problems, templates and inheritance have proved very useful. The algorithm is general and can work on any problem involving search, as long as the proper structures are defined to represent the concrete problem.

During the project, many problem were faced, both on a conceptual level and on the implementation details, which led to long debugging sessions. The heuristic for the RBFS algorithm hasn’t been implemented due to a lack of time. So the RBFS algorithm is in fact a depth first algorithm. This would have been a good improvement over what we are currently able to do. For maps, the heuristic would make use of the euclidean distance between visited vertices and the end goal of the algorithm (which is the starting point of the graph). In the inference engine problem, it would make use of the similarity between visited clauses and the clause to be proved, for example by looking at the number of literals present in both clauses.