**02180**

**Introduction to Artificial Intelligence**

**Heureka 2017**

*Lois Lano (s166388)*

*and*

*Sandra Griffon (s166341)*

Abstract:

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 kind of algorithm are employed algorithm A\* and RBFS. The language used for the project implementation is c++. The main challenge is to provide an algorithm’s script which can be reuse to tackle two 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, this two algorithms are backward changing algorithm. The difference is the frontier’s sort.

A\* is an adaptation of the breadth first algorithm Dijkstra with a heuristic. So, the frontier is sort thanks to a *priority\_queue*. Therefore, an element with high priority is served before an element with low priority. The priority depends on the distance from the start point and the heuristic.

RBFS is a depth first algorithm with a heuristic so we should sort the frontier thanks to a *stack.* So, the first element in is the first element out of the frontier.

As said before this two algorithm are backward changing algorithm with different kind of frontier. Hence, the use of templates is essential to be able to use backward chaining for different type of algorithm and different type of states. Each problem will be represented thanks to a graph where a state is a vertex and the edges will be the possible transitions between different states. The vertices have a method *isStart()* which ask to the graph is the current state is the starting state. If this method return *true* if this vertex has been chosen as the starting point during the initialisation part of the problem.

The backward chaining algorithm consist in an initial state from a graph. During the initialisation, the initial state is added in the frontier. Then, until the frontier is empty at each iteration of this algorithm, the top element of the frontier is picked up. If this element is the final one, the algorithm return *true*. Else, the successors of the current vertex are added to the frontier if they haven’t been visited yet. Then, the current vertex is retracted from the frontier and added to the vector of explored vertex. If at one time the frontier is empty that mean the problem have no solution, so *false* is returned.

In the backward chaining algorithm template, the *typename* *State* and *Frontier* are used. *Frontier* is a type of sorted vector we want (for instance: *priority\_queue, stack* or *queue*) which depends of the kind of algorithm we are implementing. *State* is the type of vertex which are in our graph which depends on the problem taken up.

1. Graph building

The knowledge base of the problem is represented by a graph which comprise vertices and edges. Here, we decided to use again templates to be able to define different type of vertices and edges. Therefore, a general edge is a structure which inheriting form the specific definition of an edge which depend on the problem. For example, a specific definition of an edge could be the name of a street between two crossing or an IF-rule. Then, a general vertex is a structure which inheriting from the specific definition which also depends on the kind of vertex of the problem we are dealing with. For instance, the specific vertex can be a crossing of a street or a set of clauses. So this general vertex have different methods. The method *successors()* is a vector containing the vertex which can be access form this vertex and a set of edges which permit to go form the current vertex to its successors and the methode *isStart()* which return *true* if this is the final vertex. A vertex also contain a vector of the edges which permit to access its successors, its parent vertex (*parent*) which is the previous vertex visited on 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. It is also a template because we don’t have the specific definition of each element of the graph. Hence, a graph contains the set of all the vertices i.e. possible state. The starting state of a graph can be set thanks to the method *startAt* with a vertex which should be the starting state as argument. Then, the method *isStart* with a vertex as argument permit to know if this vertex is the starting state of the graph. Then the most interesting method is *makePath* which return a path from the starting point to the initial state. This method can be use only after a run of one of the algorithm because the parent of the vertex have to be update to find the path.

1. Route Finding
2. The problem’s specifications

The route-finding problem consist in finding the shortest path to go from one street crossing to another. In this section, a concrete problem is tackle. So, the specific definitions of an edge and a vertex can be specified. A map of a city is a graph where the vertices are the crossing of street and the edges are the street between two crossing.

Therefore, the struct *Crossing* specify the vertices. It is defining by the position x and y. The struct *Road* specify the edges and contain the *name* of the street and the *distance* i.e. the length of the street. Thanks to the template *Crossing* will be inherited by the Vertex structure mentioned in the previous section and *Road* will be inherited by the Edges structure. Then, the algorithm is now usable for a route-finding problem.

1. Creation of a knowledge base

A knowledge base is needed to run the algorithm. So, a text file contains all necessary information to build a graph which represent the map describe in it. For example, *“0 0 street\_0 1 0”* is the first line of the file. The tow first integer are the coordinates of the street start. Then there is the street name. And finally, the two-last integer are the ending point on the street. Every line describes a one-way street. For the two-way street two line will be needed to specify each way. For instance, we can also find in the file *“1 0 street\_0 0 0”* which mean that *street\_0* is a two-way street.

The file is read a first time to get the vertices set of the graph. So for each new integer couple a new *Crossing* will be instantiate. In a second reading of the file we will update each vertex with its neighbour and the edge which permit to go from the current vertex to its successor. During the edges initialisation the length of each street will be compute. At the end of this second lecture we have the complete vertex set which permit to create our graph

1. Results

Before running the algorithm, the starting state should be set. So, after the creation of the graph representing the map we set the starting state thanks to the method *startAt.* And then we will run the chosen algorithm and the path can be get thanks to the method *makePath().*

• choice of heuristic evaluation functions

We choose as a heuristic the Euclidian distance between current vertex and the final vertex.

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

The problem addressed is to find a proof of a goal clause in a knowledge base. The knowledge base use is a list of IF-rule of clauses where clauses are a list of literals which can be related to one another by “AND” or “OR”. Here, the clauses will be the vertex and the IF-rule will make the edges between two classes.

Therefore, the struct *Clause* specify the vertices. It is defining by the position a set of literals which are contained by a clause. The struct *IfRule* specify the edges and contain the *name* of the entire rule. As in the previous part, *Clause* will be inherited by the Vertex structure and *IfRule* will be inherited by the Edges structure of the graph template.

1. Creation of a knowledge base

Here again, the knowledge base is contained in text file. For instance, an if rule is represented as following: *“a if b c”.* In this case, the head of the if rule will contain “a” and the tail will contain “b” and “c” and edgs name will be *“a if b c”.*

As in the previous problem, the file is red a first time to get all the possible clauses in the file. First in each line of the file i.e. in each rule, the world “if” will be separate the rule in two clauses. And then we separate all the clauses in literals, to be able to create the object *Clause*. Then, the file is red a second time to define the clauses successor to every clause. At the end of this second lecture we have the complete vertex set which permit to create a *KwoledgeBase* which is a graph where the vertices are clauses and the edges if-rules.

1. Results

The way of running the algorithms on this part is the same as the previous problem. First the graph s created then the starting state should be set thanks to the method *startAt.* And then we will run the chosen algorithm and the path can be get thanks to the method *makePath().*

• explanation of chosen data representations

~~• elaboration of the crucial parts of your algorithms~~  see algorithm part

• choice of heuristic evaluation functions

• explanation of the data set up for initiating a search

1. Conclusion

To be able to tackle several types of problems, the *template* structure has been massively used. The inheritance of classes has also very useful to define the vertices and edges of the graph representation of the problem.

RESULTS