**02180**

**Introduction to Artificial Intelligence**

**Heureka 2017**

*Lois Lano (s166388)*

*and*

*Sandra Griffon (s166341)*

Abstract:

Introduction to artificial intelligence project which aim to apply heuristic search methods on 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 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, these 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 these two algorithms 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 types of algorithm and different types 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. 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 takes as argument an initial state from a graph. During the initialization, the initial state is added to 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 means 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 types 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 crossings or an IF-rule. Then, a general vertex is a structure which inheriting from the specific definition of a vertex depending on the problem. 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 method *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 the specific definition of each element of the graph is not define. 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 allows to know if this vertex is the starting state of the graph. Then the most interesting method is *makePath()* which returns a path from the starting point to the initial state. This method can be used only after a run of one of the algorithm because the parent of the vertex must 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 tackled. 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* specifies the vertices. It is defined 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 two first integers are the coordinates of the street start. Then there is the street name. And finally, the last two integers are the ending point on the street. Every line describes a one-way street. For the two-way street two lines 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 initialization, the length of each street will be compute. At the end of this second lecture we have the complete vertex set which permits 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().*

The RBFS algorithm have been run to find the shortest path form (0,2) to (5,1)

9 7 avenue\_9 9 6

9 6 avenue\_9 9 5

9 5 avenue\_9 9 4

9 4 avenue\_9 9 3

9 3 avenue\_9 9 2

9 2 avenue\_9 9 1

9 1 street\_1 8 1

8 1 street\_1 7 1

7 1 street\_1 6 1

6 1 street\_1 5 1

5 8 avenue\_5 5 7

5 7 avenue\_5 5 6

5 6 avenue\_5 5 5

5 5 avenue\_5 5 4

5 4 avenue\_5 5 3

5 3 street\_3 6 3

6 3 street\_3 7 3

7 3 avenue\_7 7 4

7 4 avenue\_7 7 5

7 5 avenue\_7 7 6

7 6 avenue\_7 7 7

7 7 avenue\_7 7 8

7 8 avenue\_7 7 9

7 9 street\_9 8 9

8 9 street\_9 9 9

9 9 avenue\_9 9 8

9 8 avenue\_9 9 7

0 2 street\_2 1 2

1 2 avenue\_1 1 1

1 1 avenue\_1 1 0

1 0 street\_0 2 0

2 0 street\_0 3 0

3 0 avenue\_3 3 1

3 1 avenue\_3 3 2

3 2 avenue\_3 3 3

3 3 avenue\_3 3 4

3 4 avenue\_3 3 5

3 5 avenue\_3 3 6

3 6 avenue\_3 3 7

3 7 avenue\_3 3 8

3 8 avenue\_3 3 9

3 9 street\_9 4 9

4 9 street\_9 5 9

5 9 avenue\_5 5 8

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

That’s because all the edges value one. Moreover, the heuristic hasn’t been implemented. Thus, all the edges have the same weight and none of them is prior on the other. Therefore, the algorithm will at one time reach the solution but it will not be the shortest one because the algorithm will not have visited this path which is shortest.

For the A\* we faced a real problem, the algorithm finds a short path but it invented edges which doesn’t exist. Or the solution is infinitely long. So, the algorithm finds a solution, but, the printing of the path failed.

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* specifies 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 edges name will be *“a if b c”.*

As in the previous problem, the file is read 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 word “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 allows to create a *KnowledgeBase* 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 is 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().*

The program doesn’t want to compile. During the lecture of the file, the edges cannot be emplaced back of the edges list of the successor clauses of a clause. So, we cannot show you any result at that point because of this error.

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. All in all, the algorithm can work on the two problems.

During the project, many problems were faced and the debugging was quite long especially for memory management. But, we also face difficulty with the library *unordered\_set* which have no hash function for the object created in the project i.e. an *unordered*\_*set* of *Crossing* in the route finding problem. Thus, the heuristic hasn’t been implemented because of time missing. So, the A\* algorithm is a Dijkstra algorithm and the RBFS is a depth first algorithm. So, an amelioration is to implement this heuristic which should be for the map problem the Euclidian distance between the current vertex and the goal vertex. In the clauses inference rule problem, it will be and computation of the clause’s similarity. For instance, if two clauses have the same literals the heuristic would value 0. Then, if there is one literal which is not in the other clause the heuristic between this two clause will increase by one.

Appendix: source listing excerpts

**backwardsChanning.h**

*template<typename State>*

*using breadthfirstSearch = backwardsChaining<State, std::queue<State>>;*

*template<typename State>*

*using depthfirstSearch = backwardsChaining<State, std::stack<State>>;*

*template<typename State>*

*using dijkstraSearch = backwardsChaining<State, std::priority\_queue<State, std::vector<State>, std::greater<State>>>;*

*template<typename State, typename Frontier>*

*bool backwardsChaining<State, Frontier>::operator()(State& initialState) {*

*Frontier frontier;*

*initialState.parent = initialState.selfIterator();*

*frontier.push(initialState);*

*std::unordered\_set<State> explored;*

*explored.insert(initialState);*

*while (!frontier.empty()) {*

*auto current = frontier.top();*

*frontier.pop();*

*if (current.isStart()) {*

*return true;*

*}*

*for (auto& it : current.successors()) {*

*if (explored.find(\*it) == explored.end()) {*

*it->setParent(current);*

*frontier.push(\*it);*

*explored.insert(\*it);*

*}*

*}*

*}*

*return false;*

*}*

**graph.h**

*template<typename Base, typename VertexIterator>*

*struct Edge : public Base {*

*Edge(Base base, VertexIterator end);*

*Edge& operator=(const Edge& other) = default;*

*VertexIterator end;*

*};*

*template<typename VertexBase, typename EdgeBase>*

*struct Graph;*

*template<typename Base, typename EdgeBase>*

*struct Vertex : public Base {*

*using IteratorType = typename std::vector<Vertex>::iterator;*

*using EdgeType = Edge<EdgeBase, IteratorType>;*

*using EdgeSetType = std::vector<EdgeType>;*

*using GraphType = Graph<Base, EdgeBase>;*

*Vertex(GraphType& graph, const Base& base, const std::vector<EdgeType>& edges);*

*Vertex(const Vertex& other) = default;*

*Vertex& operator=(const Vertex& other) {*

*edges = other.edges;*

*parent = other.parent;*

*weight = other.weight;*

*\_successors = other.\_successors;*

*graph = other.graph;*

*return \*this;*

*}*

*auto successors() -> const std::vector<IteratorType>&;*

*bool isStart();*

*auto selfIterator() -> IteratorType;*

*void setParent(Vertex v);*

*auto edgeTo(const IteratorType& successor) -> const EdgeType&;*

*EdgeSetType edges;*

*IteratorType parent;*

*double weight;*

*private:*

*std::vector<IteratorType> \_successors;*

*GraphType& graph;*

*};*

*template<typename VertexBase, typename EdgeBase>*

*struct Graph {*

*using VertexBaseType = VertexBase;*

*using EdgeBaseType = EdgeBase;*

*using VertexType = Vertex<VertexBase, EdgeBase>;*

*using EdgeType = typename VertexType::EdgeType;*

*using VertexSetType = std::vector<VertexType>;*

*using EdgeSetType = typename VertexType::EdgeSetType;*

*using VertexIterator = typename VertexSetType::iterator;*

*using Path = std::vector<VertexIterator>;*

*Graph() = default;*

*template<typename VertexIt, typename EdgeIt>*

*Graph(VertexIt vBegin, VertexIt vEnd, EdgeIt eBegin);*

*auto operator[](const VertexBaseType& vertex) -> VertexIterator;*

*void startAt(const VertexBaseType& vertex);*

*bool isStart(const VertexBaseType& vertex);*

*auto makePath() -> Path;*

*private:*

*VertexSetType vertices;*

*VertexIterator start;*

*};*

**map.h**

*struct Crossing {*

*double x;*

*double y;*

*};*

*struct Road {*

*std::string name;*

*double weight;*

*};*

*using Map = Graph<Crossing, Road>;*

**knowledgebase.h**

*using Literal = std::string;*

*struct Clause {*

*Clause() = default;*

*Clause(const std::string& s);*

*std::unordered\_set<Literal> literals;*

*};*

*struct IfRule {*

*const std::string name;*

*const double weight = 1;*

*};*

*using KnowledgeBase = Graph<Clause, IfRule>;*

**fileReading.cpp**

*Map readMap(const std::string& filename) {*

*std::ifstream file(filename);*

*if(!file) {*

*throw std::runtime\_error("File failed to open.");*

*}*

*std::unordered\_set<Map::VertexBaseType> crossingSet;*

*while (file) {*

*double x1, x2, y1, y2;*

*std::string name;*

*file >> x1 >> y1 >> name >> x2 >> y2;*

*crossingSet.insert(Crossing{x1, y1});*

*}*

*std::vector<Map::VertexBaseType> crossings(crossingSet.begin(), crossingSet.end());*

*using Edge = std::pair<Map::EdgeBaseType, std::vector<Map::VertexBaseType>::difference\_type>;*

*std::vector<std::vector<Edge>> roads(crossings.size());*

*file.clear();*

*file.seekg(0);*

*while (file) {*

*double x1, x2, y1, y2;*

*std::string name;*

*file >> x1 >> y1 >> name >> x2 >> y2;*

*auto i1 = std::distance(crossings.begin(), std::find(crossings.begin(), crossings.end(), Crossing{x1, y1}));*

*auto i2 = std::distance(crossings.begin(), std::find(crossings.begin(), crossings.end(), Crossing{x2, y2}));*

*double distance = std::hypot(x2-x1,y2-y1);*

*roads[i1].emplace\_back(Road{name, distance}, i2);*

*}*

*file.close();*

*return Map(crossings.begin(), crossings.end(), roads.begin());*

*}*

*KnowledgeBase readKnowledgeBase(const std::string& filename) {*

*std::ifstream file(filename);*

*if(!file) {*

*throw std::runtime\_error("File failed to open.");*

*}*

*std::unordered\_set<KnowledgeBase::VertexBaseType> clauseSet;*

*while (file) {*

*std::string line;*

*std::getline(file, line);*

*auto pos = line.find("if");*

*clauseSet.insert(Clause(line.substr(0, pos)));*

*if (pos != std::string::npos) {*

*clauseSet.insert(Clause(line.substr(pos+2)));*

*}*

*}*

*std::vector<KnowledgeBase::VertexBaseType> clauses(clauseSet.begin(), clauseSet.end());*

*clauses.push\_back(Clause()); // Add the empty clause*

*using Edge = std::pair<KnowledgeBase::EdgeBaseType, std::vector<KnowledgeBase::VertexBaseType>::difference\_type>;*

*std::vector<std::vector<Edge>> ifRules(clauses.size());*

*file.clear();*

*file.seekg(0);*

*while (file) {*

*std::string line;*

*std::getline(file, line);*

*auto pos = line.find("if");*

*if (pos != std::string::npos) {*

*clauseSet.insert(Clause(line.substr(pos+2)));*

*}*

*auto i1 = std::distance(clauses.begin(), std::find(clauses.begin(), clauses.end(), Clause(line.substr(0, pos))));*

*if (pos != std::string::npos) {*

*auto i2 = std::distance(clauses.begin(), std::find(clauses.begin(), clauses.end(), Clause(line.substr(pos+2))));*

*ifRules[i1].emplace\_back(IfRule{line}, i2);*

*} else {*

*ifRules[i1].emplace\_back(IfRule{line}, clauses.size()-1);*

*}*

*}*

*file.close();*

*return KnowledgeBase(clauses.begin(), clauses.end(), ifRules.begin());*

*}*

**main.cpp**

*int main()*

*{*

*Map manhattan = readMap("data/manhattan.txt");*

*depthfirstSearch<Map::VertexType> search1;*

*manhattan.startAt({0, 2});*

*std::cout<< "RBFS"<<std::endl;*

*search1(\*manhattan[{5, 1}]);*

*Map::Path path1 = manhattan.makePath();*

*for (auto it = path1.begin(); it+1 != path1.end(); ++it) {*

*auto next = it+1;*

*auto edge = std::find\_if((\*next)->edges.begin(), (\*next)->edges.end(),*

*[it](const Map::EdgeType& e) {*

*return e.end == (\*it)->selfIterator();*

*});*

*std::cout <<(\*it)->x << " "<< (\*it)->y <<" "<< edge->name<<" " <<(\*next)->x << " "<< (\*next)->y <<std::endl;*

*}*

*return search1(\*manhattan[{5, 1}]);*

*}*