Technical University of Denmark

02180 Introduction to Artificial Intelligence 2014

Heureka Project

Authors:

Andreas Hallberg Kjeldsen
s092638@student.dtu.dk

Morten Chabert Eskesen
s133304@student.dtu.dk

May 12, 2014

Abstract

The Heureka Project focuses on graph searching and logical deduction. In this report we will show how to encode data sets for both a map and a knowledge base as directed graphs. Further we will introduce search algorithms for the different data sets, while also describing the heuristics used. We will describe how we optimize our knowledge base and make sure it is consistent. At last we will reflect upon our work, concluding what works and what can be improved.

1 Introduction

This report will focus on our work on the *Heureka Project*, given as an assignment in the DTU course *02180 Introduction to Artificial Intelligence*. The Heureka Project focuses on graph searching, logical deduction and the heuristics.

1.1 Objectives

The two objectives for the project are:

Route Planning
Be able to find a route from one point to another point within a map.

Logical Deduction
Be able to deduce whether a query is satisfied using a knowledge base, either as a direct proof or as a refutation proof.

2 Data Representation

To solve both objectives, we decided to represent our data as graphs. The reasoning behinds this decision, is that a graph can both represent a map and a knowledge base.

2.1 Map

For the route planning problem, we are expecting a source file containing the map data. The source file is read and interpreted, generating a directed graph representing the map. Each node represents a coordinate pair `(X, Y)` indicating either a crossing of two streets or an end of a street. Each edge has a name, a weight and represents a partial street. Edge names are not unique, as a street can consist of multiple edges.

2.1.1 Source File

The source file should be in plaintext, have one entry per line and use the following syntax:

- `X1 Y1 StreetName X2 Y2`
  Where `(X1, Y1)` and `(X2, Y2)` are node coordinates and `StreetName` is the edge name. The weight of the edge is then calculated as the straight line distance from `(X1, Y1)` to `(X2, Y2)`.

2.2 Knowledge Base

For the logical deduction problem, we are expecting a source file containing the knowledge base. The source file is read and interpreted, generating a directed graph representing the knowledge base.

2.2.1 Source File

The source file should be in plaintext, have one entry per line and use one of the following syntaxes:

- `Literal`
  Where `Literal` is the ID of the literal. When a literal is stated alone, it is interpreted as being a fact and always satisfiable.
- `Literal`$_1$ `if Literal`$_2$ `[ ... Literal`$_n$`]`
  Where `Literal`$_1$`, Literal`$_2$ `...` `Literal`$_n$ are the IDs of the literals. `Literal`$_1$ is satisfied if `Literal`$_2$ `...` `Literal`$_n$ are satisfied. There must be at least one literal to depend on, there is no specific upper limit.
- `Literal`$_1$ `Literal`$_2$ `[ ... Literal`$_n$`]`
  Where `Literal`$_1$`, Literal`$_2$ `...` `Literal`$_n$ are the IDs of the literals. Literals listed this way, are being interpreted as a disjunction of the literals. There must be listed at least two literals.

Literals can be stated as negative by prepending their ID with a `!`, example:
`!Literal`$_1$ `if Literal`$_2$ `!Literal`$_3$

2.2.2 Nodes

In the knowledge base, nodes can represent different objects:

Literal Literals are what the knowledge base works with. Literals can be `Positive` and `Negative`, they have a unique ID. A negative literal has a `!` prepended to its ID, which means literal `lit` is a positive literal and literal `!lit` is a negative literal. Literals can be stated as facts, if the positive literal is a fact, the negative literal cannot be, as this would make the knowledge base inconsistent.

Conjunctive Literal Literals can depend on more than one other literal to be satisfied. In this case we make a node representing the conjunction of the depending literals. The ID of the conjunction will be the IDs of the depending literals in sorted order separated by `&`. An edge is then made between the literal and the conjunction, and between the conjunction and the literals it represents. If all the literals of a conjunction are facts, then so will the conjunction be.

Disjunctive Literal In cases where a literal has more than one way to be satisfied, a disjunctive literal can be used. The ID of the disjunction will be the IDs of the literals in sorted order separated by `|`. An edge is made between the literal and the disjunction, and between the disjunction and the literals it represents. If any of the literals in the disjunction are facts, then so will the disjunction be.

The nodes are stored in the graph as either `AND` nodes or `OR` nodes. An `AND` node is satisfied if all nodes it connects to are satisfied. An `OR` node is satisfied if any of the nodes it connects to are satisfied. The scheme for determining the node type is:

$$\text{Node type} = \begin{cases} \texttt{AND} & \text{if } \texttt{Negative} \text{ or } \texttt{Conjunctive} \\ \texttt{OR} & \text{if } \texttt{Positive} \text{ or } \texttt{Disjunctive} \end{cases}$$

2.2.3 Edges

In the knowledge base, edges represent a dependency. If node n_1 has an edge to node n_2 it indicates that n_1 depends on n_2.

2.2.4 Illustration

View the appendix 'Knowledge Base Illustration' (section B), to see an illustration of how the graph looks internally.

3 Route Planning

For the route planning problem, we wanted to focus on finding an optimal route according to distance. Our solution does not take speeding limits into considerations nor routes requiring payment (bridges, ferries etc.).

3.1 Initiating A Search

To initiate a search, a start node and an end node must be specified. The start and end node are identified by giving two street names and then finding the node corresponding to the crossing of the two streets.

3.2 Finding A Route

We chose to use the A* algorithm, with the straight line distance (henceforth *SLD*) as our heuristic function. The SLD heuristic can only be applied if the triangle inequality is satisfied, therefore we assume that the data set supplied satisfies this. Further the SLD heuristic is both admissible and monotonic.

The A* algorithm makes use of a priority queue to keep track of which node to expand next. The priority queue prioritizes the nodes based on their score, where their score is the cost of going to the node + the SLD to the goal node. The priority queue should be able to update values, due to the scenario when a node is found multiple times, the best possible score should be used. To support this, we implemented a priority queue using a binary

heap supported by a dictionary. This gave us the possibility of making constant time look ups, and decrease value updates by $O(n)$ time.

3.3 Example Run

Running our route planner on data set 'One-way Streets' (see A.1.1), using the corner of `SktPedersStraede` and `Larsbjoernsstraede` as starting point, and the corner of `Studiestraede` and `Larsbjoernsstraede` as ending point, gave the following result:

```
SktPedersStraede -> LarslejStraede -> Noerrevoldgade ->
Vestervoldgade -> Studiestraede
```

4 Logical Deduction

For our inference engine, we wanted to be able to support both direct proofs and indirect proof. Our inference engine uses the *Closed World Assumption* when conducting a proof.

4.1 Optimizing The Knowledge Base

Removing unwanted properties of the knowledge base such as inconsistencies and self-evident clauses, while also avoiding duplicate clauses and literals, are ways of optimizing the knowledge base.

4.1.1 Removing Duplicates

We chose that the knowledge base should encode a literal as two nodes within our graph, once as a positive literal and once as a negative literal. Whenever a clause is added to our knowledge base, all the literals the clause consists of, are added to the knowledge base, afterwards the dependencies implied by the clause are created by adding edges between the nodes corresponding to the literals. If a literal has already been added, it will not be added again. Encoding our knowledge base this way, ensures that all literals and clauses will only be added once.

4.1.2 Inconsistency

When adding a clause to the knowledge base, we check for inconsistency. We determine the knowledge base is inconsistent if:

- If both the positive and negative node of a literal are stated as facts.
- If both the positive and negative node of a literal can be determined as being a fact based on their dependencies.

- If any literal at some point depend on both the positive and negative node of another literal.

4.1.3 Self-evident Clauses

If the head of a clause is also stated as one of the dependent literals, the head is removed from the dependent literals. Further if the head is depending on its negated self, the clause would never become true, and is therefore invalid.

4.2 Initiating A Search

The inference engine can answer queries. We interpret the queries as a conjunction of all the literals listed in the query. A query of `"a b c"` would be interpreted as `"a` $\wedge$ `b` $\wedge$ `c"`.

4.3 Searching For Proof

A search for a proof is carried out by having a set of start nodes. Our algorithm will then do a *Depth First Search* (henceforth DFS), starting at all the start nodes. Every time a node is reached that is not listed as satisfied and that has not been visited already, a search for proving that literal is initiated. When a node is reached and all of its dependencies (if any) have already been checked and the node is not a fact, its assumed that the literal is not satisfied. The inference engine always answers `true` or `false` to a query, never `unknown/maybe`.

4.3.1 Direct Proof

When conducting the direct proof, we start the search at the literals from the query. All the literals have to be satisfied for the query to be satisfied.

4.3.2 Indirect Proof

When conducting the indirect proof, we first negate all the literals. Then we start the search at the negated literals. If any of the negated literals are satisfied, the query is not satisfied, otherwise the query is assumed to be satisfied. This can sometimes lead to false positives, as an example querying the knowledge base build from the data set 'Inconclusive Sample' (see A.2.2) for `"a"` would claim that `a` was satisfied, as `!a` isn't satisfied.

4.3.3 Heuristics

We do not use any heuristics when performing our search. This is due to several complications we encountered.

We wanted to give the nodes coordinates, in such a way that if a node were dependent

on another, they were close by. If the nodes had coordinates, that satisfied the triangle inequality we could have used SLD as our heuristic. Unfortunately, we didn't have a proper algorithm for clustering the nodes together, therefore we discarded the idea of using SLD as our heuristic.

We also tried to give the edges between the nodes weights, where the weight was equal to the amount of literals in the end node. Using $h(0)$ as our heuristic function in combination with the edge weights, we could use A* for searching. Doing so would make the inference engine prioritize paths that had the least amount of literals to be checked. We did some benchmarking, but it showed that doing a normal DFS instead was at least as fast, as using A*. Therefore we decided not to include any heuristics in our search.

4.4 Example Run

Running our inference engine on data set 'Simple Sample' (see A.2.1), we get the following results querying for 'q':

```
[Direct] Is q satisfiable?
a is satisfied
a & p is not satisfied
a & b is satisfied
l is satisfied
b is satisfied
b & l is satisfied
m is satisfied
l & m is satisfied
p is satisfied
q is satisfied
[Direct] Yes q is satisfiable

[Refutation] Is q satisfiable?
!a is not satisfied
!a | !p is not satisfied
!l is not satisfied
!b is not satisfied
!b | !l is not satisfied
!m is not satisfied
!l | !m is not satisfied
!p is not satisfied
!q is not satisfied
[Refutation] Yes q is satisfiable
```

5 Conclusion

We have managed to encode the data sets for both problems as graphs, making it possible to use the same code to solve both problems.

5.1 Route Planning

Our route planner works, it's fast and finds an optimal route according to distance. An improvement would be to include speeding limits on the roads, this would make it possible to plan the route according to traveling time instead of distance, or even make a hybrid that would take both into consideration. It would also be advantageous to include bridges, ferries and the like, also stating the cost of using these.

5.2 Logical Deduction

Our inference engine can properly answer queries using both direct proof and indirect proof within the closed world assumption. Though we do optimize our knowledge base by removing some redundant clauses and dependencies, there are still ways to optimize it further (removing circular dependencies etc.).

The way we're conducting our indirect proof, can sometimes result in false positives, this is not satisfactory but we believe its a quirk.

We didn't choose to implement our inference engine using heuristics, but putting more research into finding a proper heuristic function would most likely be beneficial.

Appendices

A Data Sets

Below are some of the data sets we used for testing.

A.1 Route Planning

A.1.1 One-way Streets

```
10 70 Vestervoldgade 20 50
20 50 Vestervoldgade 10 70
20 50 Vestervoldgade 35 35
35 35 Vestervoldgade 20 50

10 70 SktPedersStraede 35 80
35 80 SktPedersStraede 50 90
65 100 SktPedersStraede 50 90

20 50 Studiestraede 45 70
45 70 Studiestraede 70 85

55 55 Vestergade 35 35
80 70 Vestergade 55 55

60 150 Noerregade 65 110
65 110 Noerregade 65 100
65 100 Noerregade 70 85
70 85 Noerregade 80 70

45 70 Larsbjoernsstraede 55 55
45 70 Larsbjoernsstraede 35 80

25 100 TeglgaardsStraede 35 80

50 90 LarslejStraede 35 120

10 70 Noerrevoldgade 25 100
25 100 Noerrevoldgade 10 70
25 100 Noerrevoldgade 35 120
35 120 Noerrevoldgade 25 100
35 120 Noerrevoldgade 60 150
60 150 Noerrevoldgade 35 120
```

A.2 Logical Deduction

A.2.1 Simple Sample

```
q if p
p if l m
m if b l
l if a p
l if a b
a
b
```

A.2.2 Inconclusive Sample

```
a if b
b if c
b c
```

A.2.3 Breakfast

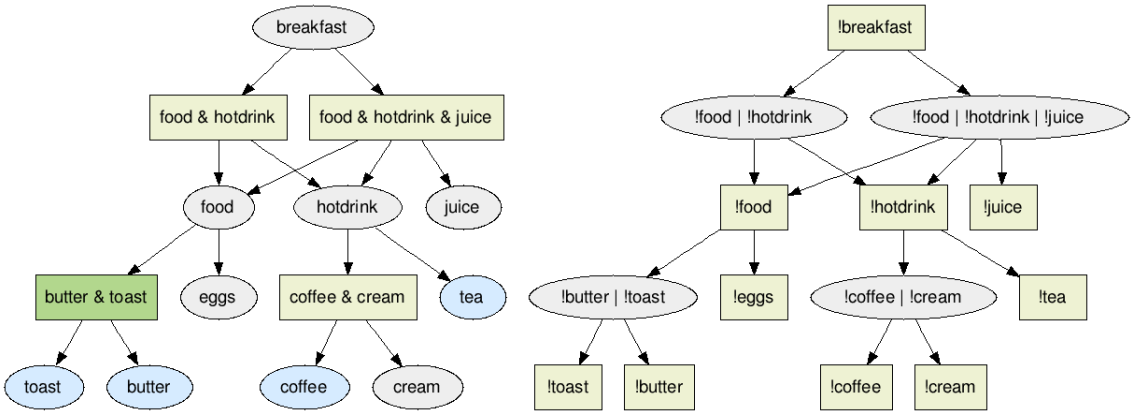
```
breakfast if hotdrink juice food
breakfast if hotdrink food

hotdrink if coffee cream
hotdrink if tea

food if toast butter
food if eggs

coffee
tea
toast
butter
```

B Knowledge Base Illustration

Below is an illustration of the graph used by the knowledge base. The illustration is generated using Graphviz[1]. The shape of the nodes indicate what kind of node they are, rectangular nodes are `AND` nodes, circular nodes are `OR` nodes. The color of the nodes indicate whether or not the nodes are facts, a `grey` or `yellow` node is not known to be a fact, a `blue` or `green` node is known to be a fact.

B.1 Breakfast Example

The example below is build upon the data set 'Breakfast' (see A.2.3).



[1] Graphviz - Graph Visualization Software: `http://www.graphviz.org/`.