DTU Department of Mathematics and Computer Science

Program Construction Exercise for
Course 02180 Introduction to Artificial Intelligence 2014

HEUREKA

– exploring heuristic search and logical deduction for AI systems.

1 Introduction

In this HEUREKA[1] exercise you are to design, implement, and test a software system which applies heuristic search methods to
1) route planning in a city map
2) logical deduction in propositional logic on clause form.

The main objectives of the exercise are
a) getting acquainted with heuristic search algorithms
b) learning how to perform logical deduction computationally
c) training modular design of software for re-use.

Generally speaking you should follow the principles and program schemes suggested in the text book. In particular you should aim at designing your program in a modular fashion so that you can re-use the heuristic search program parts developed in the first part to the second part as well.

2 Route Finding

Let us consider route finding in city maps using heuristic search as you may know it from GPS-based systems in cars.

Task: Design, implement and test a program for calculating a good (car/bicycle) travel route between two given street crossings in a city map taking into account one way streets. The initial and goal points are each supposed to be uniquely identified by a pair of street names. The resulting route should be output as a list of street names. Try with the A* algorithm and/or the RBFS algorithm, introducing an appropriate heuristics at your discretion for guiding the search. Observe that it is very easy to equip the recursive RBFS algorithm with trace output so that you can follow the various attempts at finding a best route during the search.

[1] From Greek $\varepsilon\upsilon\rho\eta\kappa\alpha$ meaning "I have found it!".

You are going to be provided with a small data set for a fragment of the Copenhagen city map that you can use for trying your program. The map is modelled as a directed graph, where the nodes are street crossings (given as coordinate pairs) and the arcs are street stretches labelled with the street name. Both way stretches are modelled by two arcs in opposite directions. As a simplification the stretches are assumed straight lines.

Accordingly, the data set simply consists of arcs in the directed graph of streets, an arc being represented as a street name between two coordinate pairs as exemplified:

```
10 70 Vestervoldgade 20 50
20 50 Vestervoldgade 10 70
10 70 SktPedersStraede 35 80
```

etc.

The coordinates are here given in an arbitrarily chosen coordinate system. You can modify/extend the data set it as you please, or apply other ones. You should also compare your route results with route finders available at the net.

3 Inference Engine for Propositional Logic

The second part of this construction exercise addresses construction of a proof system for propositional logic in clausal form. This part of your system is supposed to exploit (re-use) the heuristic search components developed in the first part.

Given as input a systems specification or knowledge base KB in the form of a finite collection of clauses, the prover is to apply the resolution inference rule to deduce new clauses. The input also comprises a conjecture or hypothesis, that is a target clause to be proved through a sequence of deduced clauses originating in the given input clauses.

Basically the HEUREKA prover is to construct the proof as a refutation proof (indirect proof). Recall that logical completeness of resolution in general requires use of refutation. The clause stemming from negation of the original conjecture is used as start clause, and the empty clause then functions as goal for the contradiction-seeking proof search process.

You may also try a setup for conducting a direct proof. In that case the various clauses in the KB are used as start clauses (thus there multiple start states), and the conjecture clause forms the goal state.

3.1 Clauses

A clause C consists of a sequence of literals, L, where a literal is an atomic proposition (identifier) P or its denial $\neg P$. We assume that duplicate literals be removed throughout (so-called factoring, cf. the text book p. 253) in clauses, so that a clause may be conceived of as a set of literals. In particular there is the empty clause representing a contradiction.

If the positive literals of a clause C are called $A_1, A_2, ..., A_k$, and the negative literals (i.e. denials) are called $B_1, B_2, ...B_l$, then it is easy to see that any clause $L_1 \vee L_2 \vee ... \vee L_{l+k}$, may be re-written as

A_1 *or* A_2 *or ...* *or* A_k *if* B_1 *and* B_2 *and ...* *and* B_l

Observe the disjunctions at the left hand side and the conjunctions at the right hand side. This form may be found convenient for human reading – in particular for the common case of definite clauses, that is where $k = 1$. This so-called Kowalski form of clauses may be given a shorthand form, e.g.

$A_1\ A_2\ ...\ A_k$ *if* $B_1\ B_2\ ...\ B_l$

where *if* may be omitted when $l = 0$, or

$A_1\ A_2\ ...\ A_k\ <\ B_1\ B_2\ ...\ B_l$

This latter form may be convenient for input data, once you get used to stating a denial $\neg p$ as the clause "`< p`".

As a simple example (1) consider

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

from which we prove `a` in a few steps.

Example (2): Try to make resolution proofs (direct and refutation) of proposition `a` from

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

Notice that the last clause, `b c`, means $b \vee c$, in turn being logically equivalent to $b \leftarrow \neg c$.

The simplest way of inputting a collection of clauses is to encode the clauses directly as given data in your HEUREKA system for the examples you wish to try.

3.2 Search Space of Resolvent Clauses

The inference process explores a state graph, where each state comprises a clause. For direct proofs the start states correspond to the given clauses of a KB. However, in case of a refutation-proof (proof-by-contradiction) the start state is the clause from the denial of the conjecture. Application of the resolution rule to a clause results in 0, 1 or more immediate successor resolvent clauses. These may be new clauses or they may be recognized as having already been achieved formerly.

We recommend applying the so-called input resolution principle. According to this principle a direct resolution proof takes the form of a sequence of clauses

$$R_0,\ R_1,\ R_2,\ \ldots\ R_n$$

where R_0 is some clause among the given clauses in the KB, and R_{i+1} is formed as resolvent of the previous clause R_i with some input clause from the given clauses, and R_n is the target conjecture clause. In general at each step in forming such a proof sequence there are multiple choices for the input KB clause. Accordingly a sequence like the above represents one branch in exploration of the state space of clauses.

In case of a refutation proof (proof by contradiction) the given clauses KB are extended with the clause(s) from denial of the conjecture. In the above proof sequence, if successful, R_0 is the denied hypothesis clause, and R_n is the empty clause.

Thus the intermediate nodes in the search process comprise a clause, and node expansion consists in application of all of the given input clauses to the currently considered clause to achieve resolvent clauses for the successor nodes. This restricted form of resolution in which one of the considered clauses has to be a clause from the KB is called input resolution. Input resolution offers a much smaller search space than ordinary resolution.

We advise the reader to work through the above small examples by hand calculation.

3.3 Achieving Completeness with Ancestor Resolution

Ordinary unrestricted resolution is logically complete for refutation proofs, but inherently logically incomplete for direct proofs. However, still for the above described input resolution, refutation proving is logically incomplete[2]. This deficiency can be remedied in case of refutation proofs by adding so-called ancestor resolution, cf. the text book p. 356. In an ancestor resolution step in addition to attempting resolving the considered clause R_i, $i > 0$, with all of the input clauses, a resolvent is attempted also by resolving the considered clause with ancestor clauses R_j, $j < i$, in the prospective proof sequence. Thus it is necessary to inspect the parent node and the parent node of the parent node etc. when computing successor nodes of a node. Input resolution extended with ancestor resolution is called linear resolution.

We advise the reader to try working through the above small examples by hand calculation of the search space.

3.4 Representation of Clauses

Clauses should be represented internally so that resolution (including removal of any duplicate literals in the resolvent clause) can be accomplished easily, and so that a convenient clause similarity measure of your suggestion can be computed for the heuristic search.

[2]It is, however, complete for definite clauses.

Appendix: Form of Report

A few words about your coming project report:

The size of the report is *limited to 8 pages* including the front page. In addition you may choose to have at most 5 appendix pages containing e.g. source listing excerpts, test output, figures etc. However, the proper report part should be comprehensible without reading the appendix material.

The front page in your report should contain the following information:

1. 02180 Introduction to Artificial Intelligence 2014
2. Title of exercise
3. Full name and study number of the author(s).

The rest of the front page can be used for a brief abstract and/or a content listing (sections). At the following up to at most 7 pages you should take care to structure your report by appropriate (sub)sectioning. An introduction and a summary/conclusion section are obligatory. Use Danish or English for the report.

In the report you should assume that your reader is familiar with the text book and this project guide, so that there is no need to repeat these sources from scratch. You should focus on

- explanation of chosen data representations,
- elaboration of the crucial parts of your algorithms, and
- choice of heuristic evaluation functions
- explanation of the data set up for initiating a search.

Small exemplifications are important since they are always good for promoting understanding. Try to be as informative as possible. Imagine that the reader could be person who is porting your system from your programming language to another programming language.

There must be a a conclusion telling the development state of your system. Don't hesitate to state what you can do as well as what you may have failed to do; we are in science – not in politics.

If you are copying text paragraphs, algorithms, or figures (from the book, the web, or elsewhere) remember that you have to state clearly the source in each case, e.g. in footnotes or in a reference list.

Enjoy it!

March 2014
Jørgen Fischer Nilsson