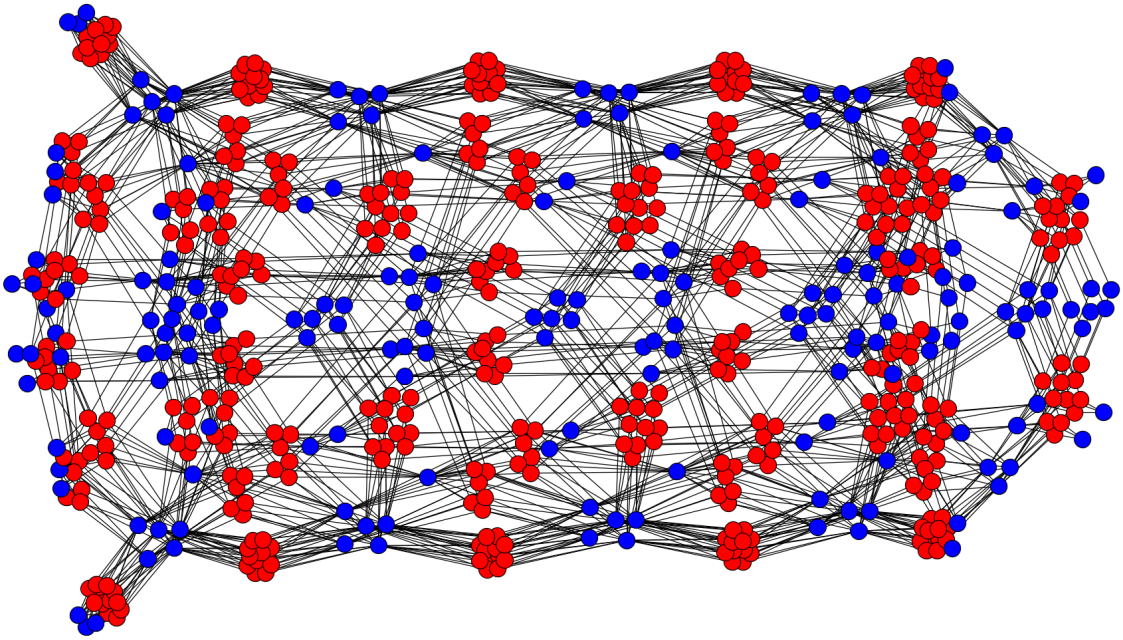
COMP3620/6320 Artificial Intelligence

Assignment 3: SAT-Based Planning

The Australian National University

Semester 1, 2014



1 Background

SAT-based planning is a powerful approach for solving certain automated planning problems. The key ideas behind this assignment were developed in the 90s, mostly by Kautz and Selman, as a way to leverage the growing power of SAT solvers.

Early SAT encodings of planning problems were generated by hand, but within a few years the Planning-Graph was being used as the basis of automatic translations of planning into SAT. (Throughout this assignment the Planning-Graph is also referred to as the plangraph). To this day, SAT-based planning techniques excel in solving certain planning problems, particularly those which allow a high degree of parallel action execution.

In this assignment you will implement various automatic translations of STRIPS planning instances (supplied as PDDL files) into SAT. We have done a lot of the hard work for you, in grounding the PDDL into objects representing STRIPS actions and propositions, generating a plangraph and fluent mutex relationships, calling the SAT solver with

the encodings you generate, and validating the resulting plans. This should allow you to focus on the fun parts, that is generating CNF encodings of planning problems and interpreting the solutions found by the SAT solver.

Part 1 of the assignment asks you to create variables and clauses to represent a basic SAT encoding of planning. Part two asks you to extend your encoding from Part 1 with some clauses which represent domain-specific control knowledge for a `logistics` domain. Finally, in Part 3 you have the chance to explore advanced SAT-based planning techniques, either from the SAT-planning literature or of your own devising.

The picture at the top of this assignment is a representation of the precondition and effect clauses in a 5 step encoding of a small logistics problem. The blue nodes represent fluents and the red nodes represent actions. There is an arc between an action and a fluent if they appear in a precondition or effect clause together.

2 Preliminary: The Planning System

In this assignment you will complete parts of a SAT-based planning system implemented in Python. This system uses some pre-compiled binaries for grounding the planning problem and solving the SAT instances your encodings will create. The planner takes a planning problem specified as a domain PDDL file and a problem PDDL file. It then uses the selected encoding (and other options) to generate and solve CNF SAT instances with planning horizons chosen by the selected query (evaluation) strategy. If one of these instances is satisfiable, the system extracts and attempts to validate a plan from the satisfying assignment returned by the SAT solver.

Due to the dependence on pre-compiled binaries for grounding and SAT solving and Linux specific system calls to run these binaries and manage the temporary files created by the system, the system is only guaranteed to work on x64 Linux installations with recent Python 2.x installations. The system may work on 64 bit Mac installations. It will NOT work on Windows without a lot of work. We are not supporting any platform other than Linux.

If you do not have a suitable Linux installation, we suggest using a virtual machine, working over SVN, or working in the labs. Even better, just install Linux along side whatever OS you normally use. It is very difficult to do lots computer science or use most research software, like SAT solvers or planners, without Linux.

If you have trouble with the supplied binaries `gringo` and `precosat`, you can obtain other binaries, and source from `http://potassco.sourceforge.net/` and `http://fmv.jku.at/precosat/`.

To display details about how to run the system use `python planner.py -h`. The planner is run with the command:

```
python planner.py DOMAIN PROBLEM EXPNAME HORIZON [options]
```

For example,

```
python planner.py benchmarks/logistics/domain.pddl
   benchmarks/logistics/problem01.pddl logistics01 10
```

In detail, the arguments are:

- `DOMAIN` - the PDDL domain;
- `PROBLEM` - the PDDL problem;
- `EXPNAME` - an arbitrary string used to name temporary files;
- `HORIZONS` - this option specifies the planning horizons (maximal planning steps) to use and depends on the chosen query strategy:
  - If the `fixed` query strategy is chosen, then the horizon should be a list of planning horizons separated by `:` characters. For example, `1:5:7` would plan at the horizons 1, 5, and 7.
  - If `ramp` query strategy is chosen, then the horizon should be three numbers `start:end:step` – the starting horizon, end horizon, and horizon step size. For example, `2:8:2` would plan at the horizons 2, 4, 6, and 8.

  Note that planning stops as soon as the system finds a plan.
- `-o OUTPUT` specifies an file which the found plan can optionally be written to (default None);
- `-q QUERY` specifies the query strategy (evaluation strategy) to use. See the description of the `HORIZONS` option for details (default `fixed`).
- `-p PLANGRAPH` specifies if the plangraph should be computed (default `true`);
- `-l PGCONS` specified what constraints should be included in encodings from the plangraph (default `both`):
  - `fmutex` includes just the fluent mutex axioms;
  - `reachable` includes just reachable action axioms;
  - `both` includes both sets of clauses.
- `-x EXECSEM` specifies the execution semantics (default `parallel`):

  - `serial` means that at most one action can be executed at each step in valid plans;
  - `parallel` means that actions cannot be executed in parallel if they have inconsistent effects or interfere.

- `-e ENCODING` specifies which encoding to use from the encodings in the `cnf_encodings` subdirectory, one of `basic`, `logistics`, or `advanced` (default `basic`);
- `-s SOLVER` selects the SAT solver to use. There is only one installed with the system currently, so ignore this option;
- `-t TIMEOUT` specifies an optional timeout for each run of the SAT solver. The system exits if the SAT solver times out (default `None`);
- `-d DBGCNF` specifies if the system should generate an CNF file annotated with variable names for you to use to debug your encodings (default `false`);
- `-r REMOVETMP` specifies if the system should remove the temporary files which it creates. Set this to false if you want to look at these files. Debuging CNFs will never be removed (default `false`).

All of the code you will have to write for this assignment is contained in the files:

- `cnf_encodings/basic.py`
- `cnf_encodings/logistics_control.py`
- `advanced_explanation.pdf` and either `cnf_encodings/advanced.py` or a separate folder with the name `advanced` which contains your planning system if you made changes outside of `advanced.py`.

The other important file to reads is `strips_problem.py` as this contains the data structures representing the STRIPS problem which you need to represent in CNF.

The subdirectory `benchmarks` contains a number of STRIPS benchmark problems on which you can test your system. These problems are specified in PDDL. Test your system on the small instances from these domains as debugging will be much harder on large problems. Do not expect your system to be able to solve all of the supplied problems. It will likely run out of memory on the larger problems. An ongoing part of SAT-based planning research is to find novel encodings which require less time and memory to solve. Maybe you can come up with something good in the last part of this assignment!

3 Part1: Basic Encoding (25 marks)

For this exercise you will write code to automatically generate the variables and clauses representing a basic encoding of STRIPS planning problems into SAT. We strongly suggest that you carefully read the SAT-based planning section of the course lecture slides

before you proceed here. To do this you will complete methods in the `BasicEncoding` class in the file `cnf_encodings/basic.py`.

For all of the following definitions, let $\langle P, A, s_0, g \rangle$ be a (grounded) STRIPS planning problem, where P is the set of state propositions, A is the set of ground actions, $s_0 \subseteq P$ is the initial state and $g \subseteq P$ is the goal. Let k be the planning horizon.

The system only allows you to add clauses to your encodings. So, you will need to translate these planning axioms into clauses (on paper, in your head, etc.) and then write code which generates and adds these clauses. This can be achieved with the following steps:

1. Translate all $(A \leftrightarrow B)$ into $(A \rightarrow B) \wedge (B \rightarrow A)$;
2. Translate all $(A \rightarrow B)$ into $(\neg A \vee B)$;
3. Translate the formula into NNF (negation normal form) by pushing negations into the atoms (actions and fluents). Remember, double negation elimination and De-Morgans Laws – i.e. $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$ and $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$.
4. Distribute over disjunctions – i.e. $(A \wedge B) \vee (C \wedge D)$ becomes $(A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$.
5. Finally, remove $False$ and $\neg True$ literals and remove clauses with $True$ and $\neg False$ literals.

See the comments in `cnf_encodings/basic.py` for ideas about how to proceed.

3.1 Q1: Action and Fluent Variables (2 marks)

For this question you need to make codes for the propositional variables for your encoding. The variables you need to generate are:

- $p@t$ for each proposition $p \in P$ and $t \in \{0, k\}$
  $p@t$ is a *fluent* denoting that p holds at step t – e.g. on(A, B)@3, $\neg\text{on(A, B)@0}$
- $a@t$ for each $a \in A$ and $t \in \{0, k-1\}$
  $a@t$ is an *action fluent* denoting that a occurs at step t – e.g. stack(A, B)@2, $\neg\text{stack(A, B)@1}$

Add these variables in the method `make_variables` in `cnf_encodings/basic.py`.

3.2 Q2: Initial State and Goal Axioms (2 marks)

For this question and all subsequent questions in Part 1, you need to use the propositional variables you made in Q1 in clauses representing various axioms.

- All propositions that are true in the initial state must hold at step 0 and only those (closed world assumption):

$$\bigwedge_{p \in s_0} p@0 \wedge \bigwedge_{p \notin s_0} \neg p@0$$

- All goal propositions must hold at step k (open world assumption):

$$\bigwedge_{p \in g} p@k$$

Add these clauses in the method `make_initial_state_and_goal_axioms` in `cnf_encodings/basic.py`.

3.3 Q3: Precondition and Effect Axioms (4 marks)

For each action $a \in A$, if a occurs at step t then:

- its preconditions must be true at step t:

$$a@t \rightarrow \bigwedge_{p \in \text{PRE}(a)} p@t$$

- Its positive effects are true at step $t + 1$:

$$a@t \rightarrow \bigwedge_{p \in \text{EFF}^+(a)} p@t{+}1$$

- Its negative effects are false at step $t + 1$:

$$a@t \rightarrow \bigwedge_{p \in \text{EFF}^-(a)} \neg p@t{+}1$$

In this planning system, no action will ever add and delete the same proposition.

Add these clauses in the method `make_precondition_and_effect_axioms` in `cnf_encodings/basic.py`.

3.4 Q4: Explanatory Frame Axioms (4 marks)

These clauses state the only way a fluent can change truth value is via the execution of an action that changes it.

For each proposition $p \in P$, if p occurs at step $t > 0$ then:

- If a fluent becomes true, then an action must have added it:

$$(\neg p@t \wedge p@t{+}1) \rightarrow \bigvee_{\substack{a \in A \\ p \in \text{EFF}^+(a)}} a@i$$

- If a fluent becomes false, then an action must have deleted it:

$$(p@t \wedge \neg p@t{+}1) \rightarrow \bigvee_{\substack{a \in A \\ p \in \text{EFF}^-(a)}} a@i$$

Add these clauses in the method `make_explanatory_frame_axioms` in `cnf_encodings/basic.py`.

3.5 Q5: Serial Mutex Axioms (4 marks)

These clauses prevent any actions whatsoever from being executed in parallel. For each pair of actions $(a, a') \in A$, where $a \neq a'$ and both a and a' occurs at step t:

- The actions cannot occur in parallel:

$$\bigwedge_{a,a' \in A^2, a \neq a'} \neg a@t \vee \neg a'@t$$

You should notice that some actions cannot be executed in parallel, even without adding explicit mutex clauses. To get full marks for this question, only add mutex clauses for pairs of actions which are not already ruled out by inconsistent effects.

Add these clauses in the method `make_serial_mutex_axioms` in `cnf_encodings/basic.py`.

3.6 Q6: Interference Mutex Axioms (4 marks)

These clauses ensure that two actions a and a' can not be executed in parallel at a step t if they interfere. The clauses are the same as generated in Q6, except only as specified below.

Two actions a and a' interfere if there is a proposition p, such that $p \in EFF^-(a)$ and $p \in PRE(a')$. To get full marks, you should not add clauses for interfering actions if their parallel execution is already prevented by effect clauses due to inconsistent effects. Also, careful not to add duplicate clauses.

Add these clauses in the method `make_interference_mutex_axioms` in `cnf_encodings/basic.py`.

3.7 Q7: Reachable Action Axioms (1 marks)

A side-effect of computing the plangraph is getting sound bounds on the first level at which actions can be executed. For each action a, if we know that a cannot be executed before step t, then we can add the following clause for each step $t' < t$:

$$\neg a@t'$$

Look in the method `make_reachable_action_axioms` in `cnf_encodings/basic.py` to see how to get this reachability information.

3.8 Q8: Fluent Mutex Axioms (2 marks)

Another side-effect of computing the plangraph is obtaining a set of fluent mutex relationships. These tell us that certain pairs of propositions cannot both be true at a given step.

These clauses are not needed for correctness, but in some cases they can make planning much more efficient!

Assert these mutex relationships with clauses along the lines of those for the action mutex relationships in Questions 5 and 6.

See method `make_fluent_mutex_axioms` in `cnf_encodings/basic.py` for details.

3.9 Q9: Extracting a Plan (2 marks)

Once the SAT solver has found a CNF instance to be satisfiable it returns a satisfying assignment to the variables in this instance. As you created these variables, you are in a position to interpret this satisfying assignment and build a plan from it!

This is as simple as finding the true action variables and inserting the corresponding actions into a plan in an order which is consistent with their steps.

See the method `build_plan` in `cnf_encodings/basic.py` for details.

4 Part 2: Domain-Specific Control Knowledge (20 Marks)

If you have successfully completed the first part of this assignment and tested the planning system on a number of larger benchmark problem instances, you will probably agree that domain-specific planning is **hard**! Of course, in general classical planning is PSPACE-complete, but even solving bounded length instances is NP-complete.

For some planning domains, such as `Pipesworld`, even bounded length plan existence is NP-complete, but for some domains there are domain-specific procedures to find (usually sub-optimal) plans in polynomial time. For example, in `Blocksworld` we can find a plan which is guaranteed to be within a factor of 2 of the length of an optimal plan by simply stacking all blocks onto the table and then re-stacking them correctly.

In this part of the assignment you are going to leverage the hard work you have already done. You are going to add some constraints on top of the basic encoding you developed, which represent domain-specific control knowledge for the `Logistics` domain (`benchmarks/logistics`).

Often control knowledge for planning problems is based on LTL (Linear Temporal Logic) and you might get inspired by studying this. However, we do not expect you to implement an automatic complication of arbitrary LTL into SAT. However, this could be a project for part 3 for a *very* keen student (See Chapter 6 of Nathan's PhD thesis if you are interested, it is in the `papers` directory).

With good control knowledge many problems become much easier to solve. Hopefully, once you have completed this part of the assignment, your planner will be able to solve larger instances of the `Logistics` domain than was possible with the basic encoding alone.

4.1 Q10: Logistics Control Knowledge (20 marks)

The `Logistics` domain is about finding a plan to use trucks and planes to move packages around. Each package starts at some location and must be moved to some other goal location. There is a set of regions, consisting of locations. Trucks can move packages around the locations within each region, but airplanes are need to move packages between regions.

The STRIPS `Logistics` domain abstracts away some of the complexities of this problem, but still leaves an interesting and challenging planning domain.

For this exercise you will implement some additional planning constraints in the file `cnf_encodings/logistics_control.py` specifically for the `Logistics` problem. You can to assume that the actions and propositions in the Problem instance come from this planning domain (see `benchmarks/logistics`) for details.

Your constraints should make solving the problem easier. This may be at the cost of optimality. That is, your additional constraints may rule out some solutions to make planning easier – for example, by restricting the way trucks, packages and airplanes can move – but they should preserve SOME solution (the problems might be very easy to solve if you added a contradiction, but wholly uninteresting!).

As an example rule to get you started, you could assert that if a package was at its destination, then it cannot leave. That is you could iterate over the goal of the problem to find the propositions which talk about where the packages should end up and make some constraints asserting that if one of these propositions is true at step t then it must still be true at step $t + 1$.

You will be marked based on the correctness, inventiveness, and effectiveness of the control knowledge you devise. You should aim to make at least three different control rules. Feel free to leave in (but comment out) rules which you abandon if you think they are interesting and want us to look at them.

Use the flag `-e logistics` to select this encoding when running the planner. The execution semantics and plangraph options will work as normal.

5 Part3: Advanced Techniques

There has been a lot of research in SAT-based planning since the early work of Kautz and Selman. Significant work has gone into finding good query strategies and encoding techniques and also in extending the approach to more expressive types of planning.

In this part of the assignment, we want you to explore this literature and either implement one of the more advanced ideas which has been tried before, or better yet come up with your own idea! It is not essential that whatever you implement works better than the basic encoding, the idea is that you explore and try something interesting.

5.1 Q11: Try an Advanced Technique (5 marks + 5 bonus marks)

For this exercise, you should either implement your own encoding taking ideas from the literature if you wish and reusing as much of `cnf_encodings/basic.py` as you like. Write this encoding in `cnf_encodings/advanced.py`.
Alternatively you can feel free to change the planning system in some other way or use it to solve some real world problem. If you want to do one of these latter two options, talk to us first! We can probably save you a lot of time.

Whatever you decide to do, some of the marks will be awarded for writing a brief report outlining your ideas and showing how your encoding or technique compares to the basic encoding on a selection of the benchmark problems (there is no need to experiment on the large problems if neither approach can solve them). Write this report in `advanced_explanation.pdf` (please do not submit anything other than a PDF for this report).

Here are some ideas for things you might like to do:

- Implement simple operator splitting and axiom factoring (only works for serial plans) [Ernst et. al, 1997; Robinson, 2012 (Thesis, Chapter 4)];
- Implement precise operator splitting and factoring (works for both serial and parallel plans) [Robinson, 2012 (Thesis, Chapter 4); Robinson et al; 2009];
- Implement an efficient encoding of mutual exclusion axioms [Rintanen, 2006; Rintanen et al., 2006];
- Implement an $\exists$-step or Relaxed $\exists$-step encoding, where a set of actions can be executed at a step in at least one order [Rintanen et. al., 2006; Wehrle and Rintanen, 2007];
- Implement constraints to enforce the process semantics, where actions must occur as early as possible [Rintanen et. al., 2006];
- Add a MaxSAT solver to the system and make it find plans with the fewest number of actions for a fixed horizon (or some other optimisation criteria such as maximally satisfying preferences [Giunchiglia and Maratea, 2011]);
- Modify the system to try different query (evaluation) strategies (and produce a report on their effectiveness) [Rintanen 2004];
- Write an automatic encoding of control knowledge specified in some input language (like LTL) [Robinson, 2012 (Thesis, Chapter 6)];
- Taylor the system to solve some real-world problem;
- Come up with your own idea!

The referenced papers (and other interesting SAT-based planning papers) can be found in the `papers` subdirectory.

Marks will be awarded for creativity, effort, and the correctness of what you implement.