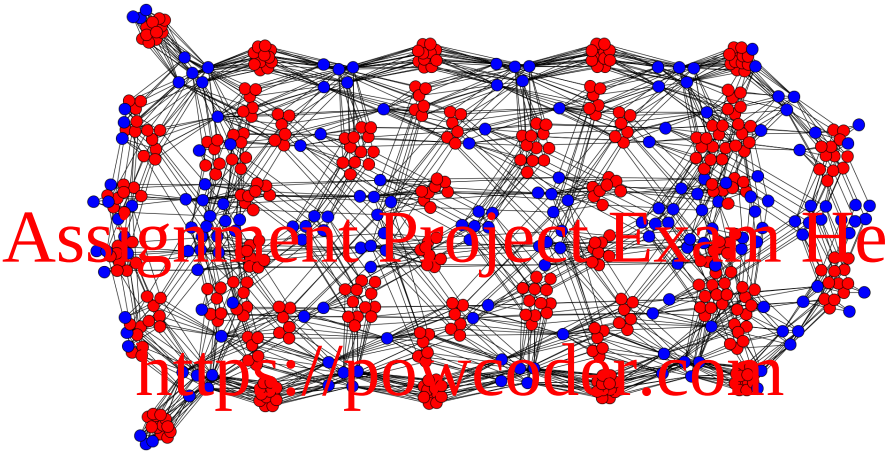
COMP3620/6320 Artificial Intelligence

Assignment 3: SAT-Based Planning

The Australian National University
Semester 1, 2022

Due date: Friday 27 May 2022 at 6pm



1 Background

SAT-based planning is a powerful approach to solve planning problems that relies on unrolling a propositional logic theory over time and checking whether or not a parallel plan exists.

Early SAT encodings of planning problems were generated by hand, but now SAT based planners provides automatic grounding for the actions, reachability analysis (to prune provably unreachable propositions or actions), and automatic translation to the corresponding formula.

Today SAT-based planning techniques do best when solving planning problems that allow for a high degree of parallelism and where the number of time steps to reach a goal state is not very high.

In this assignment you will implement various automatic translations of STRIPS planning instances (supplied as PDDL files) into SAT. The tedious part of the work has already been done (grounding the PDDL into objects representing STRIPS actions and propositions, generating a *plangraph* that computes 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.

The figure at the top of this handout is a graphical representation of the precondition and effect clauses in a five-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 Unix-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 and Mac machines. If you are using Windows 10, we suggest doing this assignment inside the Windows Subsystem for Linux.

If you have trouble with the supplied binaries `gringo` and `precosat`, you can obtain other binaries and source from `https://potassco.org/` 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:

```
python3 planner.py DOMAIN PROBLEM EXPNAME HORIZON [options]
```


where

- `DOMAIN` is the PDDL domain file.
- `PROBLEM` is the PDDL problem file.
- `EXPNAME` is an arbitrary string used to store temporary files of the experiment.
- `HORIZONS` is used to set a maximum number of time steps. Different options can be selected:
  - 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 for the horizons 1, 5, and 7.
  - If the `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.

  See below on how to select the query strategy.
- `-o OUTPUT` specifies the file in which the resulting plan is stored (default: None).
- `-q QUERY` specifies the query strategy to be used: either `fixed` (default) or `ramp`.
- `-p PLANGRAPH` is a boolean specifying whether graphplan preprocessing is used or not (default: `false`).
- `-l PGCONS` specifies what constraints should be included in encodings from the plangraph (default: `both`). Note that this option only works when you turn plangraph on with `-p true`):
  - `fmutex` includes just the fluent mutex axioms (Exercise 8).
  - `reachable` includes just reachable action axioms (Exercise 7).
  - `both` includes both sets of axioms.
- `-x EXECSEM` specifies the execution semantics (default: `parallel`):
  - `serial` means that at most one action can be executed per time step (Exercise 5).

  - `parallel` means that multiple actions per time step can be selected as long as any order is a valid one (Exercise 6).

- `-e ENCODING` specifies the CNF encoding to be used (default: `basic`). You can also select `logistics` to activate the advanced exercise.
- `-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 (in seconds) for each run of the SAT solver (default: `None`).
- `-d DBGCNF` is a boolean that specifies if the system should generate a CNF file annotated with variable names for you to use to debug your encodings. If set to true, it outputs a `.cnf_dbg` file into the `tmp_files` directory. If there's an error in your implementation and you're not sure what the problem is, turn this flag to true and manually examine the debug file. (default: `false`).
- `-r REMOVETMP` is a boolean that specifies whether the system has to remove the temporary files generated (default: `false`).

Here's an example command:

```
python3 planner.py benchmarks/miconic/domain.pddl benchmarks/miconic/problem01.pddl miconic1 4
```

Some hints and implementation notes:

- The important information to help you write your encodings is located in `strips_problem.py`. There you will find the data structures used to represent the STRIPS planning problems.

- The directory `benchmarks` contains planning problems in PDDL format. Planning problems come in two pieces: a **domain** describing the model of the actions and a **problem** file describing the initial state, goal and objects of your interest.
- Use the small problems to test your code as the big ones can take centuries to be solved as long as you do not have a good action encoding or specific domain knowledge.

- The solver won't think a correct plan has been found until Exercise 9 has been completed.
- The comments in `cnf_encodings/basic.py` provide further instructions on how to answer Exercise 1–9. Please read them carefully.
- You can use the results in `benchmarks.md` to **check your implementation**.

3 From Actions to CNF formulas

The first part of this assignment is on the generation of a CNF encoding for your STRIPS problem. The CNF has to encode the set of possible state transitions up to a maximum horizon. Please have a look at the lecture slides for more details, before starting the assignment.

We are going to encode grounded STRIPS planning problems. Such a problem is a tuple $< P, A, I, G >$ where P is the set of propositions, A is a set of actions (with their preconditions and effects), $I \subseteq P$ the initial state expressed in the closed world form, and $G \subseteq P$ is a set of atoms that needs to be true at the end of the plan execution.

The system only allows you to add CNF clauses to your encodings. So, you will need to translate the planning axioms to implement into clauses (on paper, in your head, etc.) and then write code which generates and adds these clauses. Here is a brief recap of the main transformation steps that you need to do in order to turn any formula into a CNF representation:

1. Re-write all $(A \leftrightarrow B)$ as $(A \rightarrow B) \wedge (B \rightarrow A)$.
2. Re-write all $(A \rightarrow B)$ as $(\neg A \vee B)$.

3. Translate the formula into NNF (negation normal form) by pushing negations "inwards", so that there is no negation next to a symbol other than propositional symbols (actions and fluents). This will involve applying the double negation elimination and De Morgan's Laws. For example, $\neg(A \wedge B)$ becomes $(\neg A \vee \neg B)$ and $\neg(A \vee B)$ becomes $(\neg A \wedge \neg B)$.
4. Distribute over disjunctions. For example, $(A \wedge B) \vee (C \wedge D)$ becomes $(A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$.
5. Finally, remove $False$ and $\neg True$ literals from clauses.
6. Remove clauses with $True$ and $\neg False$ literals.

Note that you don't need to write any code to transform axioms to CNF. You just need to encode the final CNF formulae directly in your Python code.

4 Exercises

4.1 Exercise 1: Action and Fluent Variables (5 marks)

For this question you need to implement the propositional variables which will be used to represent your encoding. Where k is the planning horizon, 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
- $a@t$ 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

For example, if you have 3 propositions and 4 actions, and you are looking for a plan which has a maximum length of 10, you will have $(3 \times 11) + (4 \times 10) = 73$ variables.

Add these variables in the method `make_variables` in `cnf_encodings/basic.py`. In the code, each variable is represented internally with an integer. For example, proposition 1 in step 0 might be represented with the integer 1, while its negation is represented with the integer -1.

4.2 Exercise 2: Initial State and Goal Axioms (5 marks)

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

- All propositions that are contained in the initial state and only these must hold at step 0 (Closed World Assumption):

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

- The goal condition must be true after k steps:

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

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

4.3 Exercise 3: Precondition and Effect Axioms (5 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, we will consider any action that adds and deletes the same proposition to be *invalid*. Such actions are weeded out during the grounding process.

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

4.4 Exercise 4: Explanatory Frame Axioms (10 marks)

These clauses state that 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 $t < k$ then:

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

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

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

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

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

4.5 Exercise 5: Serial Mutex Axioms (10 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.

Note that since serial mutex axioms remove parallelism which typically leads to encodings that are less efficient to solve, they are not normally used in SAT planners unless one really wants to produce sequential plans.

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

4.6 Exercise 6: Interference Mutex Axioms (10 marks)

These clauses ensure that two actions a and a' cannot be executed in parallel at a step t if they interfere. The clauses have the same form as those generated in Q5, but they only apply to inteferring actions.

Two actions a and a' interfere if there is a proposition p, such that $p \in EFF^{-}(a)$ and $p \in PRE(a')$ or vice versa. 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, take the necessary precautions so to avoid adding duplicate clauses.

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

4.7 Exercise 7: Reachable Action Axioms (5 marks)

The planner computes and uses the *plangraph* to further improve the encoding. 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.

4.8 Exercise 8: Fluent Mutex Axioms (5 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.

4.9 Exercise 9: Extracting a Plan (5 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 its time step indices.

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

4.10 Exercise 10: Using control knowledge in SAT (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-independent planning is hard! Of course, in general classical planning is PSPACE-complete, but even solving bounded length instances is NP-complete.

However, 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 in questions 1-6, and 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 compilation of arbitrary LTL into SAT. LTL formulae can be proven to be useful to find plans quickly, see this reference for more details.

With good control knowledge many problems can become easier to solve, at the expense of generality, optimality and sometimes, even completeness when plans exist (the specified control knowledge may well be conveying constraints on action execution that cannot be satisfied). On the other hand, control knowledge can also be used to obtain plans of better quality by constraining the search space only to seek for solutions conforming to specific constraints. 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 or to produce better plans on the average, or both.

Control Knowledge in Logistics

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 (or cities), consisting of locations. Trucks can move packages around the locations within each region, but airplanes are needed 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 assume that the actions and propositions in the Problem instance come from this planning domain (see `benchmarks/logistics`) for details.

For example in this domain, control knowledge rules can be used to restrict 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 $t + 1$.

If you need to determine the type of an object or what cities locations are in, you can use string matching against the names of objects (e.g. `package1`, `city4-1`). Have a look inside the `.pddl` files for more details.

You will be marked based on the correctness and inventiveness of the control knowledge you devise, as well as its effectiveness in conjunction with the basic encoding (exercises 1-6) without *plangraph* mutexes. 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, and use the flag `-p false` to disable *plangraph* mutexes. The execution semantics and options will work as normal. For example:

```
python3 planner.py benchmarks/logistics/domain.pddl benchmarks/logistics/problem05.pddl logistic1 13 -e
logistics -p false
```

The way you provide your solution can be both corroborated by theoretical results (by providing guarantee on the way the control rules limit the search space) or empirically by showing evidence that the strategy you are proposing is effective for the planner to scale up (increase the number of Logistics

problems solved or coverage) or/and produce on the average better plans (plan length) with comparable computational effort. Our evaluation is not merely quantitative but also and more importantly qualitative. You should motivate your decisions and highlight both their advantages and/or limitations (if any).

Remember to put comments in your code and in `report.pdf` to explain your approaches.

4.11 Exercise 11: Understanding Planning Problems (20 Marks)

In this exercise you need to carry out an experimental analysis on a set of benchmark domains. The empirical part in planning is a fundamental step in the development of a planning system as what is really interesting is the behavior of the system on common situations, rather than in general. There could in fact be no general guarantees as planning is still a PSPACE-Complete problem.

In this exercise you will study the various encoding techniques developed. In particular, you need to consider serial vs parallel planning. For each type, there are four sub-configurations:

- No fluent mutex or reachable action axioms: `-p false`);
- Only fluent mutex axioms: `-p true -l mutex`);
- Only reachable action axioms: `-p true -l reachable`);
- Both fluent mutex and reachable action axioms: `-p true -l both`).

You need to select three domains from the benchmark folder (blocks, depot, logistics, mprime, pipesworld, and rovers). For each domain, you must understand and evaluate the implications of the various configurations on the total time spent and the quality of the produced plans. Each problem instance corresponds to solving a number of bounded planning problems. The number of time steps starts from 0 and keeps going until a plan is found.

Note that there are 8 configurations to consider. We require you to select **3 domains for which you will try to solve the first 10 instances**. This task amounts at solving 240 planning problems, so we highly recommend you to have some kind of python or bash script to launch the experiments. If you set a 100-second timeout, you will gather all the experimental results in 6 hours in the worst case. To set the timeout, you can use the timeout command in Linux. For example:

```
timeout 100 python3 -u planner.py benchmarks/depot/domain.pddl benchmarks/depot/problem05.pddl depot_temp
1:30:1 -x parallel -l both -p false -q ramp | tee depot-1-05.log
```

The `tee` command writes the output to a log file. You need Python's -u argument for `tee` to work. For Mac, replace `timeout` with `gtimeout`. If `gtimeout` is not yet installed, run `brew install coreutils`.

It is up to you to figure out the best way of explaining this behaviour. Make use of graphs and tables if needed. Marks will be awarded according to the quality of the presentation, the thoroughness of details, and depth of analysis.

Another thing to consider is the query strategy which decides what is the sequence of horizons to try in order to find a plan. Is the serial query strategy always the right answer? Can we do something more intelligent?

Put your experimental results and discussion in `report.pdf`. Also submit a script called `run_experiments.py` or `run_experiments.sh` that can be used to reproduce the numbers in your report.

And that's the end of Assignment 3 :-)