Artificial Intelligence

Classical Planning: Planning via Inference

Jiamou Liu
The University of Auckland

Recap: Classical Planning

Classical planning seeks a path from the initial state to a goal through a finite, deterministic, fully-observable search space. We describe a classical planning task using PDDL (STRIPS) syntax:

PDDL domain description:

- States (predicates)
- Action scheme a
  - Parameters
  - Preconditions $Precond(a)$
  - Effects $Effect(a)$: $Add(a)$, $Del(a)$

PDDL problem description:

- Initial state I
- Goal g

Our aim. Building a classic planner

Challenge. Understand the task's relations with search and logic.

Main paradigms:

- Search-based (covered in the last lecture)
    - Forward (Progression) planning
    - Backward (Regression) planning
- Logic-based (to be covered in this lecture)
    - Propositional logic-based planning: SATPlan
    - Logic programming-based planning: Prolog planner

Logic-based Planning

Planning v.s. Logic:

- States can be expressed as logical sentences.
- Actions can be expressed as logical rules that describe the effects thereby capturing state transitions.
- Goal is true only if a sequence of actions are true, triggering a sequence of state transitions that link the initial state with the goal.
- Planning task can be expressed as a knowledge base:

$$\Phi = \textit{initial state} \wedge \; \textit{action descritions} \; \wedge \textit{goal}$$

- Planning is equivalent to checking satisfiability of Φ, i.e., finding an interpretation π such that

$$\pi \models \Phi$$

	Logic	**Planning**
States	Logic sentences	Logic sentences
Actions	Logic rules	Preconditions/effects
Goals	Logical sentences	Logical sentence
Plan	Satisfying interpretation	Sequence of actions

Propositional Logic-based Planner: SATPlan

Recall: Propositional satisfaction problem (SAT)

INPUT A set of propositions Φ

OUTPUT Find an interpretation (that determines the truth values of atomic propositions) π such that $\pi \models \Phi$; *failure* if such an π does not exist.

SAT Solvers[a]**:** DPLL algorithm, local search, etc.

E.g. $(p \vee \neg q) \wedge (\neg p \vee q)$ is satisfiable by $\pi(p) = 1$ and $\pi(q) = 1$.
$(p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee q)$ is not satisfiable.

[a] Beyond the scope of this course.

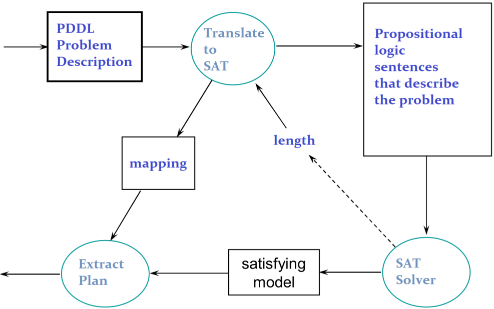
SATPlan[1] finds a plan by converting the problem to a propositional KB Φ:

- A satisfying interpretation of Φ: assign *true* to the actions that are part of a correct plan; *false* to the others.
- If there is no correct plan, Φ is not satisfiable.

[1] We mentioned it in our propositional logic lectures.

Main components of SATPlan:

1. TranslateToSAT: Translate a PDDL description into a proposition KB.
2. SATSolver: Feed this propositions to a SAT solver.
   - If the sentence is unsatisfiable, then there is not valid plan.
   - If a satisfying interpretation is found, then the goal can be achieved.
3. ExtractPlan: If the goal can be achieved, extract action variables at each time $1 \leq i \leq t$ to form a plan.



- Index time steps by $t = 0, 1, 2, 3, \ldots$.
- A proposition needs to have a finite length.
- Set a hyperparameter T_{max} to bound the length of the plan.

A bounded planning problem is a pair $(problem, t)$ where

- *problem* is a planning problem; t is a positive integer.
- A solution is a correct plan for *problem* that has length t.

- Index time steps by $t = 0, 1, 2, 3, \ldots$.
- A proposition needs to have a finite length.
- Set a hyperparameter $T_{\max}$ to bound the length of the plan.

A bounded planning problem is a pair $(problem, t)$ where

- *problem* is a planning problem; t is a positive integer.
- A solution is a correct plan for *problem* that has length t.

SATPlan procedures: Iterative deepening

for $t = 0, 1, 2, \ldots, T_{\max}$

- TranslateToSAT: Encode $(problem, t)$ as a set of propositions Φ
- SATSolver: Solve SAT on Φ
- **if** Φ is satisfiable **then**

  ExtractPlan: Construct a plan from the satisfying interpretation

  end if

end for

Question. How to translate a PDDL description to SAT?

We illustrate the process using a concrete example.

Example. [spare tire]

$$
\begin{aligned}
&Action(Remove(obj), PRECOND\colon AtAxle(obj),\\
&\quad EFFECT\colon \neg AtAxle(obj) \wedge AtGround(obj))\\
&Action(PutOn(obj), PRECOND\colon AtGround(obj) \wedge \neg AtAxle(Flat),\\
&\quad EFFECT\colon \neg AtGround(obj) \wedge AtAxle(obj))\\
&Init(AtAxle(Flat) \wedge AtGround(Spare))\\
&Goal(AtAxle(Spare))
\end{aligned}
$$

Question. How to translate a PDDL description to SAT?

We illustrate the process using a concrete example.

Example. [spare tire]

$$
\begin{aligned}
& Action(Remove(obj), PRECOND: AtAxle(obj), \\
& \quad EFFECT: \neg AtAxle(obj) \wedge AtGround(obj)) \\
& Action(PutOn(obj), PRECOND: AtGround(obj) \wedge \neg AtAxle(Flat), \\
& \quad EFFECT: \neg AtGround(obj) \wedge AtAxle(obj)) \\
& Init(AtAxle(Flat) \wedge AtGround(Spare)) \\
& Goal(AtAxle(Spare))
\end{aligned}
$$

Main steps of TranslateToSAT:

1. **Preparation:** propositionalise actions.
2. **Code initial state**
3. **Code goal**
4. **Precondition axioms**
5. **Successor-state axioms**
6. **Action exclusion axioms**

- **Step i). Propositionalise actions:** Instantiate each action:

$$
\begin{aligned}
Action(Remove(Flat), PRECOND&: AtAxle(Flat),\\
EFFECT&: \neg AtAxle(Flat) \wedge AtGround(Flat))\\
Action(Remove(Spare), PRECOND&: AtAxle(Spare),\\
EFFECT&: \neg AtAxle(Spare) \wedge AtGround(Spare))\\
Action(PutOn(Flat), PRECOND&: AtGround(Flat) \wedge \neg AtAxle(Flat),\\
EFFECT&: \neg AtGround(Flat) \wedge AtAxle(Flat))\\
Action(PutOn(Spare), PRECOND&: AtGround(Spare) \wedge \neg AtAxle(Flat),\\
EFFECT&: \neg AtGround(Spare) \wedge AtAxle(Spare))
\end{aligned}
$$

Create a new proposition for every ground term, ground action, and time step $i = 0, \ldots, t$:

$$Remove\_Flat^i, AtAxle\_Flat^i, AtGround\_Flat^i, Remove\_Spare^i,$$

$$AtAxle\_Spare^i, AtGround\_Spare^i, PutOn\_Flat^i, \ldots, Puton\_Spare^i$$

The translation consists of the following:

- **Step ii). Initial state:** Assert propositions for every positive literal appear/not appear in the initial state.

$$AtAxle\_Flat^0 \wedge AtGround\_Spare^0 \wedge \neg AtGround\_Flat^0 \wedge \neg AtAxle\_Spare^0$$

- **Step iii). Goal:** For every variable in the goal, replace the literal that contain the variable with a disjunction over constants.

$$AtAxle\_Spare^t$$

- **Step iv). Precondition axioms:** For each ground action A, add the axiom $A^{i+1} \rightarrow Precond(A)^i$ for $i = 0, 1, \ldots, t-1$.

$$\begin{aligned} Remove\_Flat^{i+1} &\rightarrow AtAxle\_Flat^i \\ Remove\_Spare^{i+1} &\rightarrow AtAxle\_Spare^i \\ PutOn\_Flat^{i+1} &\rightarrow (AtGround\_Flat^i \wedge \neg AtAxle\_Flat^i) \\ PutOn\_Spare^{i+1} &\rightarrow (AtGround\_Spare^i \wedge \neg AtAxle\_Flat^i) \end{aligned}$$

- **Step v). Successor-state axioms:** For each positive literal F, add $i = 1, \ldots, t$

$$F^i \leftrightarrow ActionCausesF^i \vee (F^{i-1} \wedge \neg ActionCausesNotF^i),$$

where *ActionCausesF* is a disjunction of all the ground actions that have F in their add list, and *ActionCausesNotF* is a disjunction of all the ground actions that have F in their delete list.

$$AtAxle\_Flat^i \leftrightarrow (PutOn\_Flat^i \vee (AtAxle\_Flat^{i-1} \wedge \neg Remove\_Flat^i)),$$

$$AtGround\_Flat^i \leftrightarrow (Remove\_Flat^i \vee (AtGround\_Flat^{i-1} \wedge \neg PutOn\_Flat^i)),$$

$$AtAxle\_Spare^i \leftrightarrow (PutOn\_Spare^i \vee (AtAxle\_Spare^{i-1} \wedge \neg Remove\_Spare^i)),$$

$$AtGround\_Spare^i \leftrightarrow (Remove\_Spare^i \vee (AtGround\_Spare^{i-1} \wedge \neg PutOn\_Spare^i)),$$

- **Step vi). Action exclusion axioms:** Actions are not taken at the same time. For any $i = 1, \ldots, t$

$$\neg Remove\_Flat^i \vee \neg Remove\_Spare^i$$

$$\neg Remove\_Flat^i \vee \neg PutOn\_Flat^i$$

$$\neg Remove\_Flat^i \vee \neg PutOn\_Spare^i$$

$$\neg Remove\_Spare^i \vee \neg PutOn\_Flat^i$$

$$\neg Remove\_Spare^i \vee \neg PutOn\_Spare^i$$

$$\neg PutOn\_Flat^i \vee \neg PutOn\_Spare^i$$

A satisfying interpretation:

The following atoms are `true` (the rest are `false`).

- Time step 0 (initial state):

$$AtAxle\_Flat^0, AtGround\_Spare^0,$$

- Time step 1:

$$Remove\_Flat^1, AtGround\_Flat^1, AtGround\_Spare^1,$$

- Time step 2:

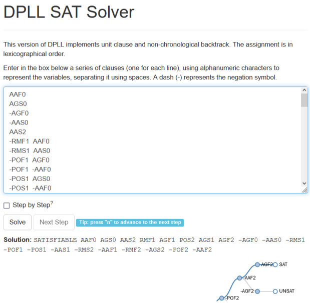
$$PutOn\_Spare^2, AtAxle\_Spare^2, AtGround\_Flat^2.$$

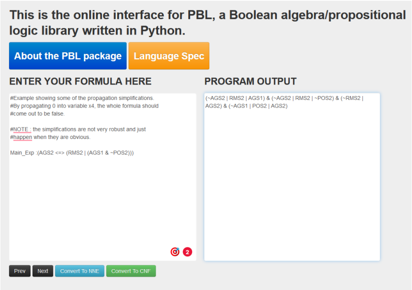
Thus a plan is found with length $t = 2$:

$$1.Remove(Flat), \qquad 2.PutOn(Spare).$$

Online demo: We can manually input a SAT and construct a plan.

- **Online DPLL SAT solver:** `https://www.inf.ufpr.br/dpasqualin/d3-dpll/`
  Input a set of propositions in conjunctive normal form.
  Solve SAT and find a satisfying interpretation.
- **Python Boolean library (PBL) which implements a CNF converter:** `https://github.com/tyler-utah/PBL`





Disadvantage: A planning problem usually requires a large propositional KB.

TranslateToSAT needs to create

- $(T_{\max} + 1) \times |Obj|^{Args_P}$ new atomic propositions for each predicate symbol, and
- $T_{\max} \times |Obj|^{Args_A}$ new atomic propositions for each action schema,

where $|Obj|$ is the set of constants, $Args_P$ is the maximum arity of a predicate, $Args_A$ is the maximum arity of an action scheme.

Advantages: Speed

- Utilising efficient domain-independent heuristic for propositional logic reasoning
- Taking advantage of mature SAT solver such as DPLL, which is highly optimised.
- Fixed structure in classical planning domain and problem means that further optimisation is possible.

Logic Programming-base Planner

Idea: Write a planner in Prolog

- PDDL and logic programs are both descriptive languages.
- Instead of using propositional logic, we write a logic program that defines valid plans.
- **Challenge:** Translating PDDL to Prolog.
- **Assumption:** Initial state, goal, and preconditions do not contain negative literals.

Advantage:

- Taking advantage of Prolog interpreter which is highly optimised.
- We are essentially developing a "PDDL interpreter":
  - Define domain-independent Prolog rules to for planning
  - Describe PDDL domain using Prolog facts.

We need to define how Prolog can code the following **main components (Domain independent):**

1. Predicates
2. States
3. Actions
4. Plan
5. Goal
6. Change State: State transitions (current state, plan, next state).
7. Conditions met: Whether a state satisfies the precondition of an action.
8. Goal met: Whether a state satisfies the goal.

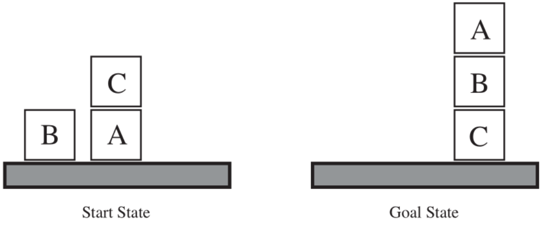
We next present a Prolog planner that is developed by Luger, Stubblefield, and Davis at UNM at `https://www.cs.unm.edu/~luger/ai-final/code/PROLOG.planner.html`. Other Prolog planner is also possible[2].

---

[2] See *Prolog: Programming for artificial intelligence*, by I. Bratko.

Example. [Blocks world domain]

- Predicates: $onTable(x)$, $on(x, y)$, $clear(x)$
- Action $moveToTable(x, y)$
  - Preconditions: $clear(x) \wedge on(x, y)$
  - Effects: $clear(y) \wedge onTable(x) \wedge \neg on(x, y)$
- Action $moveToBlock1(x, y, z)$
  - Preconditions: $clear(x) \wedge clear(z) \wedge on(x, y)$
  - Effects: $clear(y) \wedge on(x, z) \wedge \neg clear(z) \wedge \neg on(x, y)$
- Action $moveToBlock2(x, y)$
  - Preconditions: $clear(x) \wedge clear(y) \wedge onTable(x)$
  - Effects: $on(x, y) \wedge \neg clear(y) \wedge \neg onTable(x)$



- **1. Predicate:** A predicate is represented by a function symbol.
  - terms represent literals.
  - ground terms represent ground literals.

  E.g. `on(X,Y), clear(X), ontable(a), clear(a)`

- **2. States:** A state is a set of ground terms.

  E.g. `[ontable(b), on(c,a), ontable(a), clear(b), clear(a)]`

- **3. Actions:** An action name is represented by a function symbol.
  - Precondition: a set of terms
  - Effects: a set of Del and Add terms.

  E.g.

```
act(movetotable(X, Y), [clear(X), on(X,Y)],
        [del(on(X,Y)), add(clear(Y)), add(ontable(X))]).
```

- **4. Plan:** A plan is represented by a list of ground actions.

  E.g. `[movetotable(c,a), movetoblock2(b,c), movetoblock1(a,b)]`

- **5. Goal:** A goal is represented by a set of ground terms.

  E.g. `[ontable(c), on(a,b), on(b,c), clear(a)]`

- **6. Change state** Parameters: S (current state), $Effects$, S' (new state)
  - **Case 1.** If $Effects$ is empty, then new state $S' = S$.

    ```
    change_state(S, [], S).
    ```
  - **Case 2.** If the first effect is in *Add*, then add it to the state.

    ```
    change_state(S,[add(P)|T],S_new) :- change_state(S,T,S2),
                                        add_to_set(P, S2, S_new), !.
    ```
  - **Case 3.** If the first effect is in *Del*, then remove it from the state.

    ```
    change_state(S,[del(P)|T],S_new) :- change_state(S,T,S2),
                                     remove_from_set(P, S2, S_new), !.
    ```
- **7. Condition met**

  ```
  conditions_met(Precond, State) :- subset(Precond, State).
  ```
- **8. Goal met**

  ```
  goal_met(State, Goal) :- equal_set(State, Goal).
  ```

Note. Here we use a Prolog implementation of set data structure with operations such as `subset`, `add_to_set`, `remove_from_set` and `equal_set`.

Progression planning on Prolog:

```
plan(State, Goal, _, Plan) :-  goal_met(State, Goal),
            write('actions are'), nl,
            reverse_print_stack(Plan).

plan(State, Goal, Been_list, Plan) :-
            act(Name, Preconditions, Effects),
            conditions_met(Preconditions, State),
            change_state(State, Effects, Child_state),
            not(member_state(Child_state, Been_list)),
            stack(Child_state, Been_list, New_been_list),
            stack(Name, Plan, New_plan),
            plan(Child_state, Goal, New_been_list, New_plan),!.

go(S, G) :- plan(S, G, [S], []).
```

Prolog domain specification:

```
%domain definition
act(movetoblock2(X, Y), [clear(X), clear(Y), ontable(X)],
        [del(ontable(X)), del(clear(Y)), add(on(X,Y))]).

act(movetoblock1(X, Y, Z), [clear(X), clear(Z), on(X,Y)],
        [del(clear(Z)), del(on(X,Y)), add(clear(Y)), add(on(X,Z))]).

act(movetotable(X, Y), [clear(X), on(X,Y)],
        [del(on(X,Y)), add(clear(Y)), add(ontable(X))]).

%problem definition
%initial and goal states
test :- go([ontable(a), ontable(b), clear(b), clear(c), on(c,a)],
        [clear(a),on(a,b),on(b,c),ontable(c)]).
```

A run-through:

1. Start state: `[ontable(a), ontable(b), clear(b), clear(c), on(c,a)]`

   `Goal:  [clear(a),on(a,b),on(b,c),ontable(c)]`

2. `movetoblock1(c,a,b)`:

   `State: [ontable(a),ontable(b),clear(c),clear(a),on(c,b)]`

3. `movetotable(c,b)`:

   `State: [ontable(a),ontable(b), clear(c),clear(a),clear(b),ontable(c)]`

4. `movetoblock2(b,a)`:

   `State: [ontable(a),clear(c),clear(b),ontable(c),on(b,a)]`

5. `movetoblock1(b,a,c)`:

   `State: [ontable(a),clear(b),ontable(c),clear(a),on(b,c)]`

6. `movetoblock2(a,b)`:

   `State: [ontable(c),clear(a),on(b,c),on(a,b)]`

Note. This is a progression planner.

Summary of The Topic

The following are the main knowledge points covered:

- **Logic-based Planning:**
  - Making use of well-developed and highly-optimised logic inference mechanisms.
  - Reducing planning to the satisfaction problem.
- **Propositional logic-based planner:** SATPlan
  - Main components: TranslateToSAT, SATSolver, ExtractPlan
  - Translate PDDL to propositional KB
- **Logic programming-based planner**: Prolog planner
  - States and goal expressed as sets.
  - Progression planner implementation.