Planning as Satisfiability

Alan Fern *

- Review of propositional logic (see chapter 7)
- Planning as propositional satisfiability
- Satisfiability techniques (see chapter 7)
- Combining satisfiability techniques with planning graphs

* Based in part on slides by Stuart Russell and Dana Nau
Planning as Satisfiability

• The “planning as satisfiability” framework lead to large improvements when it was first introduced
  ▲ Especially in the area of optimal planning

• At that time there was much recent progress in satisfiability solvers
  ▲ This work was an attempt to leverage that work by reducing planning to satisfiability

• Planners in this framework are still very competitive in terms of optimal planning
Architecture of a SAT-based Planner

Planning Problem
- Init State
- Goal
- Actions

- Compiler (encoding)
  - CNF
  - Solver (SAT engine/s)
  - Decoder
  - Plan

Propositional formula in conjunctive normal form (CNF)

Increment plan length
If unsatisfiable

- Simplifier (polynomial inference)
  - CNF
  - satisfying model
  - Plan
Propositional Logic

- We use logic as a formal representation for knowledge
- Propositional logic is the simplest logic – illustrates basic ideas

**Syntax:**

- We are given a set of primitive propositions \{P_1, \ldots, P_n\}
  - These are the basic statements we can make about the “world”
- From basic propositions we can construct formulas
  - A primitive proposition is a formula
  - If \( F \) is a formula, \( \neg F \) is a formula (negation)
  - If \( F_1 \) and \( F_2 \) are formulas, \( F_1 \land F_2 \) is a formula (conjunction)
  - If \( F_1 \) and \( F_2 \) are formulas, \( F_1 \lor F_2 \) is a formula (disjunction)
  - If \( F_1 \) and \( F_2 \) are formulas, \( F_1 \implies F_2 \) is a formula (implication)
  - If \( F_1 \) and \( F_2 \) are formulas, \( F_1 \iff F_2 \) is a formula (biconditional)

- Really all we need is negation and disjunction or conjunction, but the other connectives are useful shorthand
  - E.g. \( F_1 \implies F_2 \) is equivalent to \( \neg F_1 \lor F_2 \)
Propositional Logic: CNF

• A literal is either a proposition or the negation of a proposition

• A clause is a disjunction of literals

• A formula is in conjunctive normal form (CNF) if it is the conjunction of clauses
  \[(\neg R \lor P \lor Q) \land (\neg P \lor Q) \land (\neg P \lor R)\]

• Any formula can be represented in conjunctive normal form (CNF)
  \^ Though sometimes there may be an exponential blowup in size of CNF encoding compared to original formula

• CNF is used as a canonical representation of formulas in many algorithms
Propositional logic: Semantics

• A **truth assignment** is an assignment of true or false to each proposition: e.g. \( p_1 = \text{false} \), \( p_2 = \text{true} \), \( p_3 = \text{false} \)

• A formula is either true or false wrt a truth assignment

• The truth of a formula is evaluated recursively as given below: (\( P \) and \( Q \) are formulas)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>( Q )</td>
<td>( \neg P )</td>
<td>( P \land Q )</td>
<td>( P \lor Q )</td>
<td>( P \Rightarrow Q )</td>
<td>( P \iff Q )</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

The base case is when the formulas are single propositions where the truth is given by the truth assignment.
Propositional Satisfiability

• A formula is **satisfiable** if it is true for some truth assignment
  
  e.g.  $A \lor B, \quad C$

• A formula is unsatisfiable if it is never true for any truth assignment
  
  e.g.  $A \land \neg A$

• Testing satisfiability of CNF formulas is a famous NP-complete problem
Propositional Satisfiability

• Many problems (such as planning) can be naturally encoded as instances of satisfiability
• Thus there has been much work on developing powerful satisfiability solvers
  ▲ these solvers work amazingly well in practice
Complexity of Planning Revisited

- Recall that PlanSAT for propositional STRIPS is PSPACE-complete
  
  (Chapman 1987; Bylander 1991; Backstrom 1993, Erol et al. 1994)

- How then it is possible to encode STRIPS planning as SAT, which is only an NP-complete problem?

### Bounded PlanSAT

**Given:** a STRIPS planning problem, and positive integer $n$

**Output:** “yes” if problem is solvable in $n$ steps or less, otherwise “no”

- Bounded PlanSAT is NP-complete
  
  (Chenoweth 1991; Gupta and Nau 1992)

- So bounded PlanSAT can be encoded as propositional satisfiability
Encoding Planning as Satisfiability: Basic Idea

• Bounded planning problem \((P,n)\):
  - \(P\) is a planning problem; \(n\) is a positive integer
  - Find a solution for \(P\) of length \(n\)

• Create a propositional formula that represents:
  - Initial state
  - Goal
  - Action Dynamics

for \(n\) time steps

• We will define the formula for \((P,n)\) such that:
  1) any satisfying truth assignment of the formula represent a solution to \((P,n)\)
  2) if \((P,n)\) has a solution then the formula is satisfiable
Overall Approach

- Do iterative deepening like we did with Graphplan:
  - for $n = 0, 1, 2, ...$,
    - encode $(P, n)$ as a satisfiability problem $\Phi$
    - if $\Phi$ is satisfiable, then
      - From the set of truth values that satisfies $\Phi$, a solution plan can be constructed, so return it and exit

- With a complete satisfiability tester, this approach will produce optimal layered plans for solvable problems
- We can use a GraphPlan analysis to determine an upper bound on $n$, giving a way to detect unsolvability
Example of Complete Formula for \((P,1)\)

\[
\text{at}(r_1,l_1,0) \land \neg\text{at}(r_1,l_2,0) \land \\
\text{at}(r_1,l_2,1) \land \\
\text{move}(r_1,l_1,l_2,0) \Rightarrow \text{at}(r_1,l_1,0) \land \\
\text{move}(r_1,l_1,l_2,0) \Rightarrow \text{at}(r_1,l_2,0) \land \\
\text{move}(r_1,l_2,l_1,0) \Rightarrow \text{at}(r_1,l_2,0) \land \\
\text{move}(r_1,l_2,l_1,0) \Rightarrow \text{at}(r_1,l_1,0) \land \\
\text{move}(r_1,l_2,l_1,0) \Rightarrow \neg\text{at}(r_1,l_2,1) \land \\
\neg\text{move}(r_1,l_1,l_2,0) \lor \neg\text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_1,0) \land \text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_2,0) \land \text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_1,0) \land \neg\text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_2,0) \land \neg\text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{move}(r_1,l_1,l_2,0) \lor \neg\text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_1,0) \land \text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_2,0) \land \text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_1,0) \land \neg\text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_2,0) \land \neg\text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{move}(r_1,l_1,l_2,0) \lor \neg\text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_1,0) \land \text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_2,0) \land \text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_1,0) \land \neg\text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_2,0) \land \neg\text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{move}(r_1,l_1,l_2,0) \lor \neg\text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_1,0) \land \text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \land \\
\neg\text{at}(r_1,l_2,0) \land \text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_1,0) \land \neg\text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \land \\
\text{at}(r_1,l_2,0) \land \neg\text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_2,l_1,0)
\]

Formula has propositions for actions and states variables at each possible timestep

Let's see how to construct such a formula
Fluents (will be used as propositions)

- If plan $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ is a solution to $(P,n)$, then it generates a sequence of states $\langle s_0, s_1, \ldots, s_{n-1} \rangle$.

- A **fluent** is a proposition used to describe what’s true in each $s_i$:
  - $\text{at}(r_1,\text{loc}_1,i)$ is a fluent that’s true iff $\text{at}(r_1,\text{loc}_1)$ is in $s_i$.
  - We’ll use $e_i$ to denote the fluent for a fact $e$ in state $s_i$:
    - e.g. if $e = \text{at}(r_1,\text{loc}_1)$
      then $e_i = \text{at}(r_1,\text{loc}_1,i)$
  - $a_i$ is a fluent saying that $a$ is an action taken at step $i$:
    - e.g., if $a = \text{move}(r_1,\text{loc}_2,\text{loc}_1)$
      then $a_i = \text{move}(r_1,\text{loc}_2,\text{loc}_1,i)$

- The set of all possible action and state fluents for $(P,n)$ form the set of primitive propositions used to construct our formula for $(P,n)$. 

A truth assignment selects a subset of these nodes (the ones assigned the value true)

We want to write down propositional formulas that only allow assignments that correspond to valid plans
Encoding Planning Problems

• We can encode \((P,n)\) so that we consider either layered plans or totally ordered plans
  ▲ an advantage of considering layered plans is that fewer time steps are necessary (i.e. smaller \(n\) translates into smaller formulas)
  ▲ for simplicity we first consider totally-ordered plans

• Encode \((P,n)\) as a formula \(\Phi\) such that
  \[\langle a_0, a_1, \ldots, a_{n-1}\rangle\] is a solution for \((P,n)\)
  if and only if
  \(\Phi\) can be satisfied in a way that makes the fluents \(a_0, \ldots, a_{n-1}\) true

• \(\Phi\) will be conjunction of many formulas
  ▲ We will now consider the types of formulas in
Formulas in $\Phi$

- Formula describing the **initial state**: (let $E$ be the set of possible facts in the planning problem)
  \[
  \bigwedge\{e_0 / e \in s_0\} \land \bigwedge\{\neg e_0 / e \in E - s_0\}
  \]
  Describes the complete initial state (both positive and negative fact)
  
  E.g. \(on(A,B,0) \land \neg on(B,A,0)\)

- Formula describing the **goal**: ($G$ is set of goal facts)
  \[
  \bigwedge\{e_n / e \in G\}
  \]
  says that the goal facts must be true in the final state at timestep $n$
  
  E.g. \(on(B,A,n)\)

- Is this enough?
  
  Of course not. The formulas say nothing about actions.
Formulas in $\Phi$

- For every action $a$ and timestep $i$, formula describing what fluents must be true if $a$ were the $i$'th step of the plan:
  - $a_i \Rightarrow \bigwedge \{ e_i / e \in \text{Precond}(a) \}$, $a$’s preconditions must be true
  - $a_i \Rightarrow \bigwedge \{ e_{i+1} / e \in \text{ADD}(a) \}$, $a$’s ADD effects must be true in $i+1$
  - $a_i \Rightarrow \bigwedge \{ \neg e_{i+1} / e \in \text{DEL}(a) \}$, $a$’s DEL effects must be false in $i+1$

- **Complete exclusion** axiom:
  - For all actions $a$ and $b$ and timesteps $i$, formulas saying $a$ and $b$ can’t occur at the same time
    - $\neg a_i \lor \neg b_i$
  - this guarantees there can be only one action at a time

- Is this enough?
  - The formulas say nothing about what happens to facts if they are not effected by an action
  - This is known as the **frame problem**
Frame Axioms

- **Frame axioms:**
  - Formulas describing what *doesn’t* change between steps $i$ and $i+1$

- Several are many ways to write these
  - Here I show a way that is good in practice

- **explanatory frame axioms**
  - One axiom for every possible fact $e$ at every timestep $i$
  - Says that if $e$ changes truth value between $s_i$ and $s_{i+1}$, then the action at step $i$ must be responsible:

  $$\neg e_i \land e_{i+1} \Rightarrow V\{a_i \mid e \text{ in ADD}(a)\}$$

  If $e$ became true then some action must have added it

  $$e_i \land \neg e_{i+1} \Rightarrow V\{a_i \mid e \text{ in DEL}(a)\}$$

  If $e$ became false then some action must have deleted it
Example

• Planning domain:
  ▶ one robot r1
  ▶ two adjacent locations l1, l2
  ▶ one operator (move the robot)

• Encode \((P,n)\) where \(n = 1\)

  ▶ Initial state: \(\{\text{at}(r1,l1)\}\)
    Encoding: \(\text{at}(r1,l1,0) \land \neg \text{at}(r1,l2,0)\)

  ▶ Goal: \(\{\text{at}(r1,l2)\}\)
    Encoding: \(\text{at}(r1,l2,1)\)

  ▶ Action Schema: see next slide
Example (continued)

- Schema: move(r, l, l’)
  
  PRE: at(r,l)
  ADD: at(r,l’)
  DEL: at(r,l)

Encoding: (for actions move(r1,l1,l2) and move(r1,l2,l1) at time step 0)

move(r1,l1,l2,0) ⇒ at(r1,l1,0)
move(r1,l1,l2,0) ⇒ at(r1,l2,1)
move(r1,l1,l2,0) ⇒ ¬at(r1,l1,1)
move(r1,l2,l1,0) ⇒ at(r1,l2,0)
move(r1,l2,l1,0) ⇒ at(r1,l1,1)
move(r1,l2,l1,0) ⇒ ¬at(r1,l2,1)
Example (continued)

• Schema:  move(r, l, l’)  
  PRE: at(r,l)  
  ADD: at(r,l’)  
  DEL: at(r,l)

• Complete-exclusion axiom:
  \( \neg \text{move}(r_1,l_1,l_2,0) \lor \neg \text{move}(r_1,l_2,l_1,0) \)

• Explanatory frame axioms:
  \( \neg \text{at}(r_1,l_1,0) \land \text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \)
  \( \neg \text{at}(r_1,l_2,0) \land \text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \)
  \( \text{at}(r_1,l_1,0) \land \neg \text{at}(r_1,l_1,1) \Rightarrow \text{move}(r_1,l_1,l_2,0) \)
  \( \text{at}(r_1,l_2,0) \land \neg \text{at}(r_1,l_2,1) \Rightarrow \text{move}(r_1,l_2,l_1,0) \)
Complete Formula for \((P,1)\)

\[
\begin{align*}
[ \text{at}(r_1, l_1, 0) \land \neg \text{at}(r_1, l_2, 0) ] \land \\
\text{at}(r_1, l_2, 1) \land \\
[ \text{move}(r_1, l_1, l_2, 0) \Rightarrow \text{at}(r_1, l_1, 0) ] \land \\
[ \text{move}(r_1, l_1, l_2, 0) \Rightarrow \text{at}(r_1, l_2, 1) ] \land \\
[ \text{move}(r_1, l_1, l_2, 0) \Rightarrow \neg \text{at}(r_1, l_1, 1) ] \land \\
[ \text{move}(r_1, l_2, l_1, 0) \Rightarrow \text{at}(r_1, l_2, 0) ] \land \\
[ \text{move}(r_1, l_2, l_1, 0) \Rightarrow \text{at}(r_1, l_1, 1) ] \land \\
[ \text{move}(r_1, l_2, l_1, 0) \Rightarrow \neg \text{at}(r_1, l_2, 1) ] \land \\
[ \neg \text{move}(r_1, l_1, l_2, 0) \lor \neg \text{move}(r_1, l_2, l_1, 0) ] \land \\
[ \neg \text{at}(r_1, l_1, 0) \land \text{at}(r_1, l_1, 1) \Rightarrow \text{move}(r_1, l_2, l_1, 0) ] \land \\
[ \neg \text{at}(r_1, l_2, 0) \land \text{at}(r_1, l_2, 1) \Rightarrow \text{move}(r_1, l_1, l_2, 0) ] \land \\
[ \text{at}(r_1, l_1, 0) \land \neg \text{at}(r_1, l_1, 1) \Rightarrow \text{move}(r_1, l_1, l_2, 0) ] \land \\
[ \text{at}(r_1, l_2, 0) \land \neg \text{at}(r_1, l_2, 1) \Rightarrow \text{move}(r_1, l_2, l_1, 0) ] \land \\
\end{align*}
\]

Convert to CNF and give to SAT solver.
Extracting a Plan

• Suppose we find an assignment of truth values that satisfies $\Phi$.
  ▲ This means $P$ has a solution of length $n$

• For $i=0,\ldots,n-1$, there will be exactly one action $a$ such that $a_i = true$
  ▲ This is the $i$'th action of the plan.

• Example (from the previous slides):
  ▲ $\Phi$ can be satisfied with $\text{move}(r1,l1,l2,0) = true$
  ▲ Thus $\langle \text{move}(r1,l1,l2,0) \rangle$ is a solution for $(P,0)$
    ■ It's the only solution - no other way to satisfy $\Phi$
Supporting Layered Plans

• **Complete exclusion axiom:**
  - For all actions \( a \) and \( b \) and time steps \( i \) include the formula \( \neg a_i \lor \neg b_i \)
  - this guaranteed that there could be only one action at a time

• **Partial exclusion axiom:**
  - For any pair of incompatible actions (recall from Graphplan) \( a \) and \( b \) and each time step \( i \) include the formula \( \neg a_i \lor \neg b_i \)
  - This encoding will allowed for more than one action to be taken at a time step resulting in layered plans
  - This is advantageous because fewer time steps are required (i.e. shorter formulas)
SAT Algorithms

• Systematic Search (Optional Material)
  ▲ DPLL (Davis Putnam Logemann Loveland)
    backtrack search + unit propagation
  ▲ I won’t cover in class

• Local Search
  ▲ Walksat (Selman, Kautz & Cohen)
    greedy local search + noise to escape minima
DPLL [Davis, Putnam, Loveland & Logemann 1962]

• How to find an assignment of truth values that satisfies \( \Phi \)?
  ▶ Use a satisfiability algorithm

• DPLL is a complete satisfiability solver
  ▶ First need to put \( \Phi \) into conjunctive normal form
    e.g., \( \Phi = D \land (\neg D \lor A \lor \neg B) \land (\neg D \lor \neg A \lor \neg B) \land (\neg D \lor \neg A \lor B) \land A \)
  ▶ Write \( \Phi \) as a set of clauses: (where each clause is a set of literals)
    \( \Phi = \{ (D), (\neg D, A, \neg B), (\neg D, \neg A, \neg B), (\neg D, \neg A, B), (A) \} \)
  ▶ Two special cases:
    ▶ \( \Phi = \{ \} \) (i.e. no clauses) is a formula that’s always true
    ▶ \( \Phi = \{ \ldots, (\), \ldots \} \) (i.e. an empty clause) is a formula that’s always false
  ▶ DPLL simply searches the space of truth assignments, assigning one proposition a value at each step of the search tree
Basic Observations

If literal $L_1$ is true, then clause $(L_1 \lor L_2 \lor \ldots)$ is true
If clause $C_1$ is true, then $C_1 \land C_2 \land C_3 \land \ldots$ has the same value as $C_2 \land C_3 \land \ldots$
Therefore: Okay to delete clauses containing true literals!

If literal $L_1$ is false, then clause $(L_1 \lor L_2 \lor L_3 \lor \ldots)$ has the same value as $(L_2 \lor L_3 \lor \ldots)$
Therefore: Okay to delete shorten containing false literals!
If literal $L_1$ is false, then clause $(L_1)$ is false
Therefore: the empty clause means false!
DPLL

Davis – Putnam – Loveland – Logemann

dpl1(F, literal)
{
    remove clauses containing literal
    shorten clauses containing \(\neg\) literal
    if (F contains no clauses) return true;
    if (F contains empty clause)
        return false;

    ; if F has a clause with single literal then force it it
    ; to be true since it must be the case

    if (F contains a unit or pure L)
        return dpl1(F, L);

    choose V in F;
    if (dpl1(F, \(\neg\) V)) return true;
    return dpl1(F, V);
}

Can initialize by calling dpl1(F, bogus) where bogus is a
literal that is not in F
WalkSat

- Local search over space of **complete** truth assignments

- Start with random truth assignment

- Repeat until stopping condition

  - With probability $P$: flip any variable in any unsatisfied clause
  
  - With probability $(1-P)$: flip best variable in any unsat clause

  - The best variable is the one that when flipped causes the most clauses to be satisfied

  - $P$, the “temperature” controls the randomness of search

- Randomness can help avoid local minima

- If it finds a solution it typically finds it very quickly

- Not complete: does not test for unsatisfiability
Planning Benchmark Test Set

• Extension of Graphplan test set
• blocks world - up to 18 blocks, $10^{19}$ states
• logistics - complex, highly-parallel transportation domain.

Logistics.d:
- 2,165 possible actions per time slot
- $10^{16}$ legal configurations ($2^{2000}$ states)
- optimal solution contains 74 distinct actions over 14 time slots

• Problems of this size never previously handled by general-purpose planning systems
Scaling Up Logistics Planning

Graph showing the log solution time for different logistics planning methods:
- Graphplan
- DP
- DP/Satz
- Walksat

The x-axis represents different logistics planning scenarios labeled as `rocket.a`, `rocket.b`, `log.b`, `log.a`, `log.c`, `log.d`. The y-axis represents the log solution time.
What SATPLAN Shows

• General propositional reasoning can compete with state of the art specialized planning systems
  ▲ New, highly tuned variations of DPLL surprising powerful
  ▲ Radically new stochastic approaches to SAT can provide very low exponential scaling

• Why does it work?
  ▲ More flexible than forward or backward chaining
  ▲ Randomized algorithms less likely to get trapped along bad paths
Discussion

• How well does this work?
  ▲ Created an initial splash but performance can depend strongly on the particular SAT encoding used

• The basic SAT encoding we described does not involve any preprocessing or reasoning about the problem
  ▲ GraphPlan’s power is primarily due to such preprocessing (i.e. the graph construction)

• **Idea**: Combine SAT with some GraphPlan-like pre-processing to get more effective encodings
BlackBox (GraphPlan + SatPlan)

- The BlackBox procedure combines planning-graph expansion and satisfiability checking
- It is roughly as follows:

  - for \( n = 0, 1, 2, \ldots \)
    - Graph expansion:
      - create a “planning graph” that contains \( n \) “levels”
    - Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence
    - If it does, then
      - Encode \((P,n)\) as a satisfiability problem \( \Phi \) by translating the planning graph to SAT
      - If \( \Phi \) is satisfiable then return the solution
Blackbox

Can be thought of as an implementation of GraphPlan that uses an alternative plan extraction technique than the backward chaining of GraphPlan.
Translation of Plan Graph

Fact \implies Act_1 \lor Act_2

Act_1 \implies Pre_1 \land Pre_2

\neg Act_1 \lor \neg Act_2

• Can create such constraints for every node in the planning graph.
• Only involves facts and actions in the graph.
Blackbox Results
Applicability

• When is the BlackBox approach *not* a good idea?
  ▲ when domain too large for propositional planning approaches
  ▲ when long *sequential* plans are needed
  ▲ when solution time dominated by reachability analysis (plan-graph generation), not extraction

• Can download a copy of it