Planning

The Planning problem
Planning with State-space search
Partial-order planning
Planning graphs
Planning with propositional logic
Analysis of planning approaches

What is Planning

Generate sequences of actions to perform tasks and achieve objectives.
- States, actions and goals
- Search for solution over abstract space of plans.
Classical planning environment: fully observable, deterministic, finite, static and discrete.
Assists humans in practical applications
- design and manufacturing
- military operations
- games
- space exploration

Difficulty of real world problems

Assume a problem-solving agent using some search method ...
- Which actions are relevant?
  - Exhaustive search vs. backward search
  - What is a good heuristic functions?
    - Good estimate of the cost of the state?
    - Problem-dependent vs. -independent
  - How to decompose the problem?
    - Most real-world problems are nearly decomposable.

Planning language

What is a good language?
- Expressive enough to describe a wide variety of problems.
- Restrictive enough to allow efficient algorithms to operate on it.
- Planning algorithm should be able to take advantage of the logical structure of the problem.
STRIPS and ADL

General language features

Representation of states
- Decompose the world in logical conditions and represent a state as a conjunction of positive literals.
  - Propositional literals: Poor → Unknown
  - FO-literals (grounded and function-free): At(Plane1, Melbourne) ∧ At(Plane2, Sydney)
- Closed world assumption

Representation of goals
- Partially specified state and represented as a conjunction of positive ground literals
- A goal is satisfied if the state contains all literals in goal.
General language features

Representations of actions
- Action = PRECOND + EFFECT
  Action(Fly(p,from,to))
  PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
  EFFECT: ¬At(p,from) ∧ At(p,to)
- action schema (p, from, to need to be instantiated)
  - Action name and parameter list
  - PRECONDITION (conj of function-free literals)
  - EFFECT (conj of function-free literals and P is True and not P is false)
- Add-list vs delete-list in Effect
  - STRIPS assumption: (avoids representational frame problem)
  - every literal NOT in the effect remains unchanged

Language semantics?

How do actions affect states?
- An action is applicable in any state that satisfies the precondition.
- For FO action schema applicability involves a substitution θ for the variables in the PRECOND.
  `At(P1, JFK) ∧ At(P2, SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)`
  Satisfies: `At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)`
  With θ = `(p/P1, from/JFK, to/SFO)`
  Thus the action is applicable.

Expressiveness and extensions

Strip is simplified
- Important limit: function-free literals
- Allows for propositional representation
- Function symbols lead to infinitely many states and actions

Recent extension: Action Description language (ADL)
Action(Fly(p:Plane, from: Airport, to: Airport),
PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to),
EFFECT: ¬At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
)

Standardization: Planning domain definition language (PDDL)

Example: air cargo transport

Init(`At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, JFK) ∧ At(P2, JFK) ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)`)
Goal(`At(C1, JFK) ∧ At(C2, SFO)`)
Action(`Load(c, p, a)`)
PRECOND: At(c, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
EFFECT: ¬At(c, a) ∧ In(c, p)
Action(`Unload(c, p, a)`)
PRECOND: In(c, p) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
EFFECT: At(c, a) ∧ ¬In(c, p)
Action(`Fly(p, from, to)`)
PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
EFFECT: ¬At(p, from) ∧ At(p, to)

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Load(C2, P2, JFK), Fly(P2, JFK, SFO)]

Example: Spare tire problem

Init(`At(Flat, Axle) ∧ At(Spare, Axle)`)
Goal(`At(Spare, Axle)`)
Action(`PutOn(Spare, Axle)`)
PRECOND: At(Spare, Axle)
EFFECT: ¬At(Flat, Axle) ∧ At(Spare, Axle)
Action(`Remove(Spare, Trunk)`)
PRECOND: At(Spare, Trunk)
EFFECT: ¬At(Spare, Trunk) ∧ At(Spare, Ground)
Action(`PutOn(Spare, Ground)`)
PRECOND: At(Spare, Ground)
EFFECT: ¬At(Spare, Ground) ∧ At(Flat, Axle)
Action(`Remove(Flat, Axle)`)
PRECOND: At(Flat, Axle)
EFFECT: ¬At(Flat, Axle)

This example goes beyond STRIPS: negative literal in pre-condition (ADL description)
Example: Blocks world

Init(\text{On}(A, \text{Table}) \land \text{On}(B, \text{Table}) \land \text{On}(C, \text{Table}) \land \text{Block}(A) \land \text{Block}(B) \land \text{Block}(C) \land \text{Clear}(A) \land \text{Clear}(B) \land \text{Clear}(C))

Goal(\text{On}(A, B) \land \text{On}(B, C))

Action(Move(b, x, y))

\text{PRECOND: } \text{On}(b, x) \land \text{Clear}(b) \land \text{Clear}(y) \land \text{Block}(b) \land (b \neq x) \land (b \neq y) \lor (x \neq y)

\text{EFFECT: } \text{On}(b, y) \land \text{Clear}(x) \land \neg \text{On}(b, x) \land \neg \text{Clear}(y)

Spurious actions are possible: Move(B, C, C)

Planning with state-space search

Both forward and backward search possible

Progression planners
- forward state-space search
- Consider the effect of all possible actions in a given state

Regression planners
- backward state-space search
- To achieve a goal, what must have been true in the previous state.

Progression and regression

Progression algorithm

Formulation as state-space search problem:
- Initial state = initial state of the planning problem
- Literals not appearing are false
- Actions = those whose preconditions are satisfied
- Goal test = does the state satisfy the goal
- Step cost = each action costs 1

No functions ... any graph search that is complete is a complete planning algorithm.
- E.g. A*

Inefficient:
- (1) irrelevant action problem
- (2) good heuristic required for efficient search

Regression algorithm

How to determine predecessors?
- What are the states from which applying a given action leads to the goal?

Actions must not undo desired literals (consistent)
- Often much lower branching factor than forward search.
Heuristics for state-space search

Neither progression or regression are very efficient without a good heuristic.
- How many actions are needed to achieve the goal?
- Exact solution is NP hard, find a good estimate

Two approaches to find admissible heuristic:
- The optimal solution to the relaxed problem.
  - Remove all preconditions from actions
  - The subgoal independence assumption: The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving the subproblems independently.

Partial-order planning

Progression and regression planning are totally ordered plan search forms.
- They cannot take advantage of problem decomposition.
  - Decisions must be made on how to sequence actions on all the subproblems

Least commitment strategy:
- Delay choice during search

Shoe example

Goal(RightShoeOn ∧ LeftShoeOn)
Init()
Action(RightShoe, PRECOND: RightSockOn
  EFFECT: RightShoeOn)
Action(RightSock, PRECOND:
  EFFECT: RightSockOn)
Action(Lef tShoe, PRECOND: LeftSockOn
  EFFECT: LeftShoeOn)
Action(Lef tSock, PRECOND:
  EFFECT: LeftSockOn)

Planner: combine two action sequences:
(1) leftsock, leftshoe
(2) rightsock, rightshoe

Partial-order planning (POP)

Any planning algorithm that can place two actions into a plan without which comes first is a PO plan.

POP as a search problem

States are (mostly unfinished) plans.
- The empty plan contains only start and finish actions.
Each plan has 4 components:
- A set of actions (steps of the plan)
- A set of ordering constraints: A < B (A before B)
- Cycles represent contradictions.
- A set of causal links
  - The plan may not be extended by adding a new action C that conflicts with the causal link. (if the effect of C is ¬p and if C could come after A and before B)
- A set of open preconditions.
  - If precondition is not achieved by action in the plan.

Example of final plan

Actions={Rightsock, Rightshoe, Leftsock, Leftshoe, Start, Finish}
Orderings={Rightsock < Rightshoe; Leftsock < Leftshoe}
Links={Rightsock->Rightsockon -> Rightshoe, Leftsock->Leftsockon-> Leftshoe, Rightshoe->Rightshoeon->Finish, ...}
Open preconditions={}
POP as a search problem

A plan is **consistent** iff there are no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open preconditions is a **solution**.

A partial order plan is executed by repeatedly choosing *any* of the possible next actions.
- This flexibility is a benefit in non-cooperative environments.

Solving POP

Assume propositional planning problems:
- The initial plan contains Start and Finish, the ordering constraint Start < Finish, no causal links, all the preconditions in Finish are open.
- Successor function:
  - picks one open precondition p on an action B and
  - generates a successor plan for every possible consistent way of choosing action A that achieves p.
- Test goal

Enforcing consistency

When generating successor plan:
- The causal link A→p→B and the ordering constraint A < B is added to the plan.
- If A is new also add start < A and A < B to the plan.
- Resolve conflicts between new causal link and all existing actions.
- Resolve conflicts between action A (if new) and all existing causal links.

Process summary

Operators on partial plans
- Add link from existing plan to open precondition.
- Add a step to fulfill an open condition.
- Order one step w.r.t another to remove possible conflicts.

Gradually move from incomplete/vague plans to complete/correct plans.
Backtrack if an open condition is unachievable or if a conflict is irresolvable.

Example: Spare tire problem

```
Init(At(Flat, Axle) ∧ At(Spare, trunk))
Goal(At(Spare, Axle))
Action(Remove(Spare, Trunk))
PRECOND: At(Spare, Trunk)
EFFECT: ¬At(Spare, Trunk) ∧ At(Spare, Ground)
Action(Remove(Flat, Axle))
PRECOND: At(Flat, Axle)
EFFECT: ¬At(Flat, Axle) ∧ At(Flat, Ground)
Action(PutOn(Spare, Axle))
PRECOND: At(Spare, Ground) ∧ ¬At(Flat, Axle)
EFFECT: At(Spare, Axle) ∧ ¬At(Spare, Ground)
Action(LeaveOvernight)
PRECOND: ¬At(Spare, Ground) ∧ ¬At(Flat, Axle)
EFFECT: ¬At(Spare, Axle) ∧ At(Spare, trunk) ∧ ¬At(Flat, Axle)
```

Solving the problem

Initial plan: Start with EFFECTS and Finish with PRECOND.
Solving the problem

Initial plan: Start with EFFECTS and Finish with PRECOND.
Pick an open precondition: At(Spare, Axle)
Only PutOn(Spare, Axle) is applicable
Add causal link: PutOn(Spare, Axle) → Finish
Add constraint: PutOn(Spare, Axle) < Finish

Pick an open precondition: At(Spare, Ground)
Only Remove(Spare, Trunk) is applicable
Add causal link: Remove(Spare, Trunk) → PutOn(Spare, Axle)
Add constraint: Remove(Spare, Trunk) < PutOn(Spare, Axle)

Pick an open precondition: ¬At(Flat, Axle)
LeaveOverNight is applicable
Conflict: LeaveOverNight also has the effect ¬At(Spare, Ground)
To resolve, add constraint: LeaveOverNight < Remove(Spare, Trunk)

Pick an open precondition: At(Spare, Trunk)
Only Start is applicable
Add causal link: Start → Remove(Spare, Trunk)
Conflict: of causal link with effect At(Spare, Trunk) in LeaveOverNight
No re-ordering solution possible.

Remove LeaveOverNight, Remove(Spare, Trunk)
and causal links
Repeat step with Remove(Spare,Trunk)
Add also RemoveFlatAxle and finish
Some details ...

What happens when a first-order representation that includes variables is used?
- Complicates the process of detecting and resolving conflicts.
- Can be resolved by introducing inequality constraint.

CSP's most-constrained-variable constraint can be used for planning algorithms to select a PRECOND.

Planning graphs

"Could"?
- Records only a restricted subset of possible negative interactions among actions.
They work only for propositional problems.
Example:
  Initial: Have(Cake)
  Goal: Have(Cake) \land Eaten(Cake))
  Action(Eat(Cake), PRECOND: Have(Cake)
    EFFECT: \neg Have(Cake) \land Eaten(Cake))
  Action(Bake(Cake), PRECOND: \neg Have(Cake)
    EFFECT: Have(Cake))

Planning graphs

Cake example

Start at level S0 and determine action level A0 and next level S1.
- A0 >> all actions whose preconditions are satisfied in the previous level.
- Connect precond and effect of actions S0 --> S1
- Inaction is represented by persistence actions.
- Conflicts between actions are represented by mutex links.

Cake example

Level S1 contains all literals that could result from picking any subset of actions in A0.
- Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
- S1 defines multiple states and the mutex links are the constraints that define this set of states.
Continue until two consecutive levels are identical: leveled off
- Or contain the same amount of literals (explanation follows later)

Cake example

A mutex relation holds between two actions when:
- 
  - Inconsistent effects: one action negates the effect of another.
  - Interference: one of the effects of one action is the negation of a precondition of the other.
  - Competing needs: one of the preconditions of one action is mutually exclusive with the precondition of the other.

A mutex relation holds between two literals when (inconsistent support):
- If one is the negation of the other OR
- If each possible action pair that could achieve the literals is mutex.

Cake example
PG and heuristic estimation

PG’s provide information about the problem
- A literal that does not appear in the final level of the graph cannot be achieved by any plan.
- Level of appearance can be used as cost estimate of achieving any goal literals = level cost.
- Small problem: several actions can occur
  - Restrict to one action using serial PG (add mutex links between every pair of actions, except persistence actions).
- Cost of a conjunction of goals? Max-level, sum-level and set-level heuristics.

PG is a relaxed problem.

The GRAPHPLAN Algorithm

How to extract a solution directly from the PG

```
function GRAPHPLAN(problem) return solution or failure

    graph ← INITIAL-PLANNING-GRAPH(problem)

    loop do
        if goals all non-mutex in last level of graph then do
            solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
            if solution ≠ failure then return solution
            else if NO-SOLUTION-POSSIBLE(graph) then return failure
            graph ← EXPAND-GRAPH(graph, problem)
        end if
    end loop
```

Example: Spare tire problem

```
Init(At(Flat, Axle) ∧ At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk)
    PRECOND: At(Spare,Trunk)
    EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle)
    PRECOND: At(Flat,Axle)
    EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
Action(PutOn(Spare,Axle)
    PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
    EFFECT: At(Spare,Axle) ∧ ¬At(Spare,Ground))
Action(LeaveOvernight
    PRECOND:
    EFFECT:
```

Initially the plan consist of 5 literals from the initial state and the CWA literals (S0).
Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
Also add persistence actions and mutex relations.
Add the effects at level S1
Repeat until goal is in level S1

EXPAND-GRAPH also looks for mutex relations
- Inconsistent effects
  - e.g. Remove(Spare, Trunk) and LeaveOvernight due to At(Spare,Ground) and not ATM(Spare, Ground)
- Interference
  - e.g. Remove(Flat,Axle) and Leave(Flat, Axle) due to At(Flat, Axle) and not At(Flat, Axle)
- Competing needs
  - e.g. in S1, At(Spare,Axle) and At(Flat,Axle)
- Inconsistent support
  - e.g. in S1, At(Spare,Axle) and At(Flat,Axle)

In S2, the goal literals exist and are not mutex with any other
- Solution might exist and EXTRACT-SOLUTION will try to find it
- EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:
  - Initial state = last level of PG and goal goals of planning problem
  - Actions = select any set of non-conflicting actions that cover the goals in the state
  - Goal = reach level S0 such that all goals are satisfied
  - Cost = 1 for each action.
GRAPHPLAN example

Termination? YES
PG are monotonically increasing or decreasing:
- Literals increase monotonically
- Actions increase monotonically
- Mutexes decrease monotonically
Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off!

Planning with propositional logic

Planning can be done by proving theorem in situation calculus. Here: test the satisfiability of a logical sentence:

\[ \text{initial state} \land \text{all possible action descriptions} \land \text{goal} \]

Sentence contains propositions for every action occurrence.
- A model will assign true to the actions that are part of the correct plan and false to the others
- An assignment that corresponds to an incorrect plan will not be a model because of inconsistency with the assertion that the goal is true.
- If the planning is unsolvable the sentence will be unsatisfiable.

SATPLAN algorithm

\[
\text{function SATPLAN}(\text{problem}, \text{Tmax}) \rightarrow \text{solution or failure} \\
\text{inputs: problem, a planning problem} \\
\text{Tmax, an upper limit to the plan length} \\
\text{for } T = 0 \text{ to } \text{Tmax} \text{ do} \\
\text{cnf, mapping} \leftarrow \text{TRANSLATE-TO_SAT}(\text{problem}, T) \\
\text{assignment} \leftarrow \text{SAT-SOLVER}(\text{cnf}) \\
\text{if } \text{assignment} \text{ is not null then} \\
\text{return } \text{EXTRACT-SOLUTION}(\text{assignment}, \text{mapping}) \\
\text{return failure}
\]

How to determine the time step where the goal will be reached?

- Start at \( T=0 \)
- Assert \( \text{At}(P1, \text{SFO})^0 \land \text{At}(P2, \text{JFK})^0 \)
- Failure. Try \( T=1 \)
- Assert \( \text{At}(P1, \text{SFO})^1 \land \text{At}(P2, \text{JFK})^1 \)
- ...
- Repeat this until some minimal path length is reached.
- Termination is ensured by \( \text{Tmax} \)

\[
\text{cnf, mapping} \leftarrow \text{TRANSLATE-TO_SAT}(\text{problem}, T)
\]

Distinct propositions for assertions about each time step.
- Superscripts denote the time step
  \( \text{At}(P1, \text{SFO})^0 \land \text{At}(P2, \text{JFK})^0 \)
- No CWA thus specify which propositions are not true
  \( \neg \text{At}(P1, \text{SFO})^0 \land \neg \text{At}(P2, \text{JFK})^0 \)
- Unknown propositions are left unspecified.
The goal is associated with a particular time-step
- But which one?
How to encode actions into PL?
- Propositional versions of successor-state axioms:
  \[\text{At}(P1, \text{JFK}) \Rightarrow \neg (\text{Fly}(P1, \text{JFK}, \text{SFO}) \land \neg \text{At}(P1, \text{JFK})) \land \text{Fly}(P1, \text{SFO}, \text{JFK}) \land \text{At}(P1, \text{SFO})]\
- Such an axiom is required for each plane, airport and time step.
- If more airports add another way to travel than additional disjuncts are required.

Once all these axioms are in place, the satisfiability algorithm can start to find a plan.

Multiple models can be found:
They are NOT satisfactory: (for \(T=1\))
\[\text{Fly}(P1, \text{SFO}, \text{JFK}) \land \text{Fly}(P1, \text{JFK}, \text{SFO}) \land \text{Fly}(P2, \text{JFK}, \text{LAX})\]
The second action is infeasible otherwise.
Yet the plan IS a model of the sentence:
\[\text{initial state} \land \text{all possible action descriptions} \land \text{goal}\]

Avoiding illegal actions: pre-condition axioms:
\[\neg (\text{Fly}(P2, \text{JFK}, \text{SFO}) \land \text{Fly}(P2, \text{JFK}, \text{LAX}))\]
Prevents simultaneous actions.
Lost of flexibility since plan becomes totally ordered.
- Restrict exclusion to preconditions.

A plane can fly at two destinations at once:
They are NOT satisfactory: (for \(T=1\))
\[\text{Fly}(P1, \text{SFO}, \text{JFK}) \land \text{Fly}(P2, \text{JFK}, \text{SFO}) \land \text{Fly}(P2, \text{JFK}, \text{LAX})\]
The second action is infeasible otherwise.

Avoid spurious solutions: action-exclusion axioms:
\[\neg (\text{Fly}(P2, \text{JFK}, \text{SFO}) \land \text{Fly}(P2, \text{JFK}, \text{LAX}))\]
Prevents simultaneous actions.

Lost of flexibility since plan becomes totally ordered.
No actions are allowed to occur at the same time:
- Restrict exclusion to preconditions.

Planning is an area of great interest within AI:
- Search for solution
- Constructively prove existence of solution.

Biggest problem is the combinatorial explosion in states.
Efficient methods are under research:
- E.g. divide-and-conquer