Automated Planning and Scheduling
Lecture 14: Recap, Q&A
Tomáš Balyo, Dominik Schreiber | February 7, 2019
Outline

- Recap on all topics
- Questions and answers
Planning & Scheduling: Motivation

- Why automated planning and scheduling?

- Planning: Given a current state, a set of possible actions, and a desired state, find a sequence of actions = plan to achieve the desired state.

- Scheduling: Given a collection of actions and restricted resources, decide how to execute all the actions in an efficient manner.
Restrictive assumptions of Planning

Compared to the “Real World” in Classical Planning

- There are finitely many states and actions
- The world state is fully observable, the agent knows the current state
- Actions are deterministic, they only have one outcome
- The world is static, it only changes by the agents actions
- Goal is defined as a set of states
- Plan is defined as a sequence of actions
Three Kinds of Planners

- **Domain–specific**
  - A planner designed and developed for a specific planning domain.
  - Won’t work well or at all for other planning domains.
  - Examples: Path finding algorithms, sokoban puzzle solver

- **Domain–independent**
  - A planner that works on any planning domain (given the restrictions on the previous slide).
  - Correctness and completeness is guaranteed, but performance may be worse than a domain-specific planner on its respective domain.

- **Configurable**
  - Domain independent engine, input includes info about efficient solving.
  - One example of this HTN (Hierarchical Task Network) Planning.
A classical planning problem $\pi = (S, A, s_I, s_G)$ is a tuple where

- $S$ represents the final set of world states
- $A$ represents the final set of actions
- $s_I \in S$ represents the initial state
- $s_G \subset S$ represents the set of goal states

A plan $P = [a_1, a_2, \ldots, a_n]$ is a sequence of actions where $a_i \in A$ that transforms the world state from $s_I$ to a state $s \in s_G$. 
In general far too many states to represent explicitly

We represent states as a set of features

- a set of propositions that are true (PR – Propositional rep.)
- vector of values of finite domain variables (FDR – finite dom. rep.)

Example

- PR propositions: Tr@A, Tr@B, Tr@C, P1@A, P2@A, P1@B, P2@B, P1@C, P2@C, P1-in-Truck, P1-in-Truck
- FDR variables: TruckLocation = \{A|B|C\}, Package1Location = \{A|B|C|T\}, Package2Location = \{A|B|C|T\}
Representation of all possible Actions

- List all the actions – Explicit Representation
  - Can be a huge amount, but most of the time it’s fine
  - Requires some script to generate the list, cannot be done by hand
  - Usually used with FDR

- Operators – Action Templates – Implicit Representation
  - Using objects, types and predicates define Action Templates

Example: operator for loading a package at some location

Types: location, package
Objects: P1, P2 – package, A, B, C – location
Predicates: at(P – package, L – location),
TruckAt(L – location), InTruck(P – package)
Operator:
loadPackage(P – package, L – location) = ({at(P, L),
TruckAt(L)}, {¬ at(P, L), InTruck(P)})
Grounding: Actions from Operators

Types: location, package
Objects: P1, P2 - package, A, B, C - location
Predicates: PackageAt(P - package, L - location), TruckAt(L - location), InTruck(P - package)
Operator:
loadPackage(P - package, L - location) =
(\{PackageAt(P, L), TruckAt(L)\}, \{\neg PackageAt(P, L), InTruck(P)\})

- The operator and objects generate these actions:
loadPackage(P1, A), loadPackage(P1, B), loadPackage(P1, C),
loadPackage(P2, A), loadPackage(P2, B), loadPackage(P2, C),
Theorem.

**PLANSAT** is **PSPACE**-complete.

A proof has two parts:

1. **Show that** **PLANSAT** $\in$ **PSPACE**
   - Propose an algorithm deciding **PLANSAT** which “only” takes polynomial space

2. **Show that** every problem in **PSPACE** can be reduced to **PLANSAT**
   - If a Turing machine on polynomial space can solve a problem, show that we can solve the problem with classical planning
A Turing machine in PDDL: Plan

- Found plan:
  1: transition( p1 p2 s0 s0 zero zero right )
  2: transition( p2 p3 s0 s0 zero zero right )
  3: transition( p3 p4 s0 s0 zero zero right )
  4: transition( p4 p5 s0 s1 one one right )
  5: transition( p5 p6 s1 s1 zero one right )
  6: transition( p6 pb s1 s1 zero one right )
  7: transition( pb p6 s1 s2 blank blank left )
  8: halt( p6 s2 one )

- How can a non-deterministic TM be realized?
- Just add multiple transition atoms for some of the state-symbol-combinations to the initial state
- Nice analogy:
  TM is deterministic ⇔ A forward search through the problem’s state space is a linear path without branches (no decisions necessary!)
Now what?

- **PSPACE**-complete problems are hard. [citation needed]
- What are possible responses to this (too) high complexity?
  1. **Restrict our model** to make it easier (but still useful!)
     - We’ll try on the next slides
  2. **Work out heuristics and/or approximations** that work fine in practice
     - Upcoming lecture(s) on heuristics and search methods
  3. **Drop our model’s generality and develop planning algorithms** which are **specialized** for the problem at hand
     - Partly being done in HTN planning (later lecture)
     - Feel free to take some robotics course covering motion planning
Complexity of Optimal Planning

Theorem.
For each mentioned \texttt{PSPACE}\textsuperscript{-complete} \texttt{PLANSAT} (sub)problem, the corresponding \texttt{PLANMIN} problem is \texttt{PSPACE}\textsuperscript{-complete}, as well.

Theorem.
Take the previous theorem and replace each occurrence of \texttt{PSPACE} with \texttt{NP}. Then the theorem still holds.

Observation: For polynomial \texttt{PLANSAT} subproblems, the corresponding \texttt{PLANMIN} subproblem may still be \texttt{NP-complete}!

- Example: \texttt{PLANMIN}_0 (no preconditions)
State Space Graph

Planning: find a path from AAB to CCC
State Space Graph

- So planning is just path-finding :-)  
- We can use standard path search algorithms  
  - Breadth-first search  
  - Depth-first search  
  - Dijkstra’s algorithm  
  - ...

BUT...

- The graph is astronomically huge  
- Does not fit in the memory  
- We will generate it on the go
Various Search Algorithms

Uninformed search:
- Breadth-first search and depth-first search
- Dijkstra’s algorithm

Informed, heuristic search:
- Best-First and Greedy Best-First search
- A* algorithm
- Enforced Hill Climbing technique
- Backwards and bi-directional search
Admissible heuristics

Admissibility.

A heuristic \( h(s) \) regarding a goal \( g \) is admissible iff \( \forall s : h(s) \leq h^*(s) \), where \( h^*(s) \) is the actual remaining distance to the goal \( g \).

- Our heuristic: \( h_{euc}(s) := d(s, g) = \sqrt{\sum_{a \in g} 1 - [a \in s]} \)
- Is this heuristic admissible (for uniform action costs)?
  Only if no action satisfies more than one goal atom.
- Discussion: Is the heuristic useful?
  - Can be useful if problem has many goals, each of which only take a single action
  - Mostly useless when many actions are required to produce a goal
  - Worst Case: Large planning problems with a single goal; heuristic degenerates to \( h_{euc}(s) = 1 \) for almost all \( s \)
The central paradigm: Relaxation

- We cannot hope to find the true goal distance from all states for our heuristic.
- Instead, fall back to a simplified problem (relaxation)
  - Easier to compute and analyze
  - Provides at least certain bounds for the original problem’s properties

Delete-relaxation [GNT16]

Let $\pi = (P, A, s_i, G)$ a planning problem. Then $\pi_r := (P, A_r, s_i, G)$ is called the delete-relaxation of $\pi$, whereas $A_r = \{a_r \mid a \in A\}$, with

$\text{pre}(a_r) = \text{pre}^+(a)$ and $\text{eff}(a_r) = \text{eff}^+(a)$.

- Delete-relaxed problem never gets harder in any way
The Fast-Forward Heuristic

Fast-Forward heuristic $h^{FF}(s)$ [HN01]

- Build relaxed planning graph $G$ from $s$
  until relaxed goal $g^+$ is satisfied
- Extract an actual (relaxed) solution plan $p$
- Return cost of $p$
Delete-relaxed Landmark heuristics

- Try to find common points which all valid plans share
- How many such points are still missing? ⇒ Use for heuristic

Disjunctive Action Landmarks [HR15]

A disjunctive action landmark is a set of actions $A$ such that each delete-relaxed plan contains some $a \in A$.

- Construct a Justification Graph [HR15]
  - Vertices: Single atoms. Directed edges for each action $a$:
    some precondition of $a$ to each of its effects

- Each graph cut forms a Landmark!
Relaxation by abstraction

- Another relaxation: Abstract state space of some atom pattern $X$
- Example: Consider subspace of pattern $X := \{at(p_1, \cdot), at(p_2, \cdot)\}$
Relaxation by abstraction

- Another relaxation: Abstract state space of some atom pattern $X$
- Example: Consider subspace of pattern $X := \{at(p_1, \cdot), at(p_2, \cdot)\}$
Relaxation by abstraction

- Another relaxation: **Abstract state space of some atom pattern** $X$
- Example: Consider subspace of pattern $X := \{ \text{at}(p1, \cdot), \text{at}(p2, \cdot) \}$
Relaxation by abstraction

- Another relaxation: **Abstract state space of some atom pattern** $X$
- Example: Consider subspace of pattern $X := \{at(p_1, \cdot), at(p_2, \cdot)\}$

Plan search in abstracted state space can be much easier
⇒ Exploit this for a heuristic
Critical Path Heuristics

- Suppose we can apply multiple actions in parallel
- Build causal dependency graph as DAG from initial state to goal
- (At least) one path in DAG is the longest $\Rightarrow$ Critical path

Idea for relaxation: Allow applying $\leq m$ actions in parallel

$\{h^m \mid m = 1, 2, \ldots\}$ heuristics family [GH00]

- Polynomial for a fixed (small) $m$; admissible for all $m$
Lifted Backwards Search

- Goal: decrease the branching factor of search
- Try to ground as little as possible – least commitment strategy

\[
\text{Lifted-backward-search}(O, s_0, g) \\
\quad \pi \leftarrow \text{the empty plan} \\
\quad \text{loop} \\
\quad \quad \text{if } s_0 \text{ satisfies } g \text{ then return } \pi \\
\quad \quad A \leftarrow \{(o, \theta) | o \text{ is a standardization of an operator in } O, \\
\quad \quad \quad \theta \text{ is an mgu for an atom of } g \text{ and an atom of effects } (o), \\
\quad \quad \quad \quad \text{and } \gamma^{-1}(\theta(g), \theta(o)) \text{ is defined}\} \\
\quad \quad \text{if } A = \emptyset \text{ then return failure} \\
\quad \quad \text{nondeterministically choose a pair } (o, \theta) \in A \\
\quad \quad \pi \leftarrow \text{the concatenation of } \theta(o) \text{ and } \theta(\pi) \\
\quad \quad g \leftarrow \gamma^{-1}(\theta(g), \theta(o))
\]

- branching factor will get (significantly) decreased, still need backtracking though
The STRIPS Algorithm

- The original STRIPS algorithm is a lifted version of the algorithm below.

```
Ground-STRIPS(O, s, g)
π ← the empty plan
loop
  if s satisfies g then return π
  A ← \{a | a is a ground instance of an operator in O, and a is relevant for g\}
  if A = ∅ then return failure
  nondeterministically choose any action a ∈ A
  π' ← Ground-STRIPS(O, s, precond(a))
  if π' = failure then return failure
  ;; if we get here, then π' achieves precond(a) from s
  s ← γ(s, π')
  ;; s now satisfies precond(a)
  s ← γ(s, a)
  π ← π . π' . a
```

\[ g_2 = (g - \text{effects}(a_2)) \cup \text{precond}(a_2) \]
\[ π' = \langle a_6, a_4 \rangle \text{ is a plan for precond}(a_2) \]
\[ s = γ(γ(s_0, a_6), a_4) \text{ is a state satisfying precond}(a_2) \]
The STRIPS Algorithm Properties

- STRIPS tries to solve each goal separately
- The current goal is the preconditions of the last selected action
- Works if the goals can be solved in some linear order
- Not optimal, not complete – 
  E.g. cannot solve the register assignment problem
  - swap the values of two variables (registers)
  - registers: R1, R2, R3 values: V0, V1, ..., V5
  - initial state: value(R1, V3), value(R2, V5), value(R3, V0)
  - goal: value(R1, V5), value(R2, V3)
  - operator: assign(r1, v1, r2, v2):
    value(r1, v1), value(r2, v2) -> value(r1, v2)
Plan-Space Planning - Basic Idea

- Backward search from the goal
- Each node of the search space is a *partial plan*
  - A set of partially-instantiated actions
  - A set of constraints
- Make more and more refinements, until we have a solution
- Types of constraints:
  - *precedence constraint:* a must precede b
  - *binding constraints:*
    - inequality constraints, e.g., $v_1 \neq v_2$ or $v \neq c$
    - equality constraints (e.g., $v_1 = v_2$ or $v = c$) and/or substitutions
  - *causal link:*
    - use action $a$ to establish the precondition $p$ needed by action $b$
- How to tell we have a solution: no more *flaws* in the plan
  - Will discuss flaws and how to resolve them
Layers $i$ of possible atoms $P_i$ and potential actions $A_i$

One layer of atoms+actions $\cong$ one time step

- Multiple actions per step allowed when they do not conflict: any ordering must be valid and lead to identical results

- Negative atoms are included as a complementary atom set
For each atom $p$ at each layer, add **persistence action** $nop_p$

- $pre(nop_p) = eff(nop_p) = \{p\}$ (nop = “no operation”)
- Make explicit that an atom remains unchanged between layers
- Also for negative atoms
In addition to atoms $P_i$ and actions $A_i$, maintain sets of conflicts $M_i$:

- Identify pairs of atoms / of actions which logically cannot co-occur
- Remember these as mutually exclusive (mutex)
- Limits possible degree of action parallelism per step
Atom mutex: **Opposite atoms**

- Atom pairs \{p, \bar{p}\} are obviously mutex
- Notation for mutex: \{p, q\} ∈ M_i if p and q are mutex at layer i
- Example: \{t@A, \neg t@A\} ∈ M_1 (even: \{t@A, \neg t@A\} ∈ M_i for all i)
Action mutex: **Conflicting effects**

- Actions $a_1, a_2$ are mutex if an effect of $a_1$ is mutex with an effect of $a_2$
- Example: $\{\text{driveAtoB, driveBtoC}\} \in M_2$ because $\{t@B, \neg t@B\} \in M_2$
Action mutex: Interference between actions

- Actions \{a_1, a_2\} are mutex if an effect of \(a_1\) interferes with a precondition of \(a_2\): \(\exists p \in \text{eff}(a_1) : \neg p \in \text{pre}(a_2)\)
- Example: \text{driveAtoB} deletes \(t@A\) which is needed by \text{loadp1@A} \\
  \Rightarrow \{\text{driveAtoB}, \text{loadp1@A}\} \in M_1
Atom mutex: **Conflicting enabling actions**

- Atoms \( \{p_1, p_2\} \) are mutex if each pair of enabling actions is mutex.
- Example: \( \{t@A, t@B\} \in M_1 \) because \( \{\text{driveAtoB}, \text{nop}_{t@A}\} \in M_1 \)
Atom mutex: Conflicting enabling actions

- Atoms \( \{p_1, p_2\} \) are mutex if each pair of enabling actions is mutex.
- Example: \( \{t@A, t@B\} \in M_1 \) because \( \{\text{driveAtoB}, \text{nop}\}_{t@A} \} \in M_1 \)
- Similarly, \( \{t@B, p1@T\} \in M_1 \) because \( \{\text{driveAtoB}, \text{loadp1@A}\} \in M_1 \)
- Decide if goal can be met at some layer
  - At $P_3$, $p1@C$ and $p2@C$ are both reachable
  - Still, $\{p1@C, p2@C\} \in M_3$ (see illustration for """"proof"""")
  - As a consequence, goal is not satisfiable at $P_3$
    $\Rightarrow$ Expand graph until goals are not mutex any more (?)
The sets of atoms and actions in $P_i$ grow monotonically in $i$; eventually, set of mutexes $M_i$ decreases monotonically in $i$. When atoms, actions, and mutexes reach a fixpoint, no more logical information can be drawn from the graph itself just by expanding it further. If not all goals are contained or if some are still mutex, $\Rightarrow$ 1st termination criterium for unsatisfiability. Else, more layers may be needed (switch example from exercises) until a valid plan is found or unsatisfiability can be shown. Can still find new logical insights $I_i$ by explicit backwards searches from goal atoms to initial state, even after fixpoint is reached. If these insights $I_i$ cease to increase as well after $i$ layers: information fully converged $\Rightarrow$ 2nd termination criterium for unsat.
Planning graph: Properties

- The sets of atoms and actions in $P_i$ grow monotonically in $i$; eventually, set of mutexes $M_i$ decreases monotonically in $i$.
- When atoms, actions, and mutexes reach a fixpoint . . .
  - No more logical information can be drawn from graph itself just by expanding it further.
Planning graph: Properties

- The sets of atoms and actions in $P_i$ grow monotonically in $i$; 
  eventually, set of mutexes $M_i$ decreases monotonically in $i$
- When atoms, actions, and mutexes reach a fixpoint . . . 
  - No more logical information can be drawn from graph itself just by expanding it further
  - If not all goals are contained or if some are still mutex:
    $\Rightarrow$ 1st termination criterium for unsatisfiability
The sets of atoms and actions in $P_i$ grow monotonically in $i$; 
*eventually*, set of mutexes $M_i$ decreases monotonically in $i$

When atoms, actions, and mutexes reach a fixpoint . . .

- No more logical information can be drawn from graph itself just by expanding it further
- If not all goals are contained or if some are still mutex:
  $\Rightarrow$ 1st termination criterium for unsatisfiability
- Else, more layers may be needed (switch example from exercises) until valid plan is found or unsatisfiability can be shown
The sets of atoms and actions in $P_i$ grow monotonically in $i$;
\textit{eventually, set of mutexes $M_i$ decreases monotonically in $i$}

When atoms, actions, and mutexes reach a fixpoint . . .

- No more logical information can be drawn from graph itself just by expanding it further
- If not all goals are contained or if some are still mutex:
  \implies 1st termination criterium for unsatisfiability
- Else, more layers may be needed (switch example from exercises) until valid plan is found or unsatisfiability can be shown

Can still find new logical insights $l_i$ by explicit backwards searches from goal atoms to initial state, even after fixpoint is reached
Planning graph: Properties

- The sets of atoms and actions in $P_i$ grow monotonically in $i$; eventually, set of mutexes $M_i$ decreases monotonically in $i$.
- When atoms, actions, and mutexes reach a fixpoint . . .
  - No more logical information can be drawn from graph itself just by expanding it further.
  - If not all goals are contained or if some are still mutex:
    $\Rightarrow$ 1st termination criterium for unsatisfiability.
  - Else, more layers may be needed (switch example from exercises) until valid plan is found or unsatisfiability can be shown.
- Can still find new logical insights $l_i$ by explicit backwards searches from goal atoms to initial state, even after fixpoint is reached.
- If these insights $l_i$ cease to increase as well after $i$ layers:
  information fully converged $\Rightarrow$ 2nd termination criterium for unsat.
Algorithm 1 Abstract Graphplan

1: \[G := \langle A_0, M_0, P_0 \rangle = \langle \{\}, \{\}, s_0 \rangle\]
2: \[l := 0\]
3: while TRUE do
4: \quad if \[g \in P_l \text{ and } \forall g_1, g_2 \in g: \{g_1, g_2\} \notin M_l\] then
5: \quad \quad result := extractPlan(G)
6: \quad \quad if result \neq FAILURE then return result
7: \quad end if
8: \quad l := l + 1
9: \quad (A_l, M_l, P_l) := expand(G)
10: \quad G := G \cup \langle A_l, M_l, P_l \rangle
11: \quad if \ G \text{ completely converged} \text{ then} \text{ return FAILURE}
12: end while
Towards a SAT Encoding of Planning

General procedure of SAT-based planning:

- Limit encoding of planning problem to at most $n$ steps (actions)
- When unsatisfiable, increase $n$ and try again
  ⇒ Top-level procedure similar to Graphplan, iter. deepening search
SAT Encoding of Planning

Clauses of the encoding:

1. The initial state must hold at $t = 0$: $\forall p \in s_0 : is^0_p$, $\forall p \notin s_0 : \neg is^0_p$

2. At every step, at least one action is applied: $(do_{a_1}^t \lor do_{a_2}^t \lor \ldots do_{a_k}^t)$

3. At every step, at most one action is applied: $\forall a_1 \neq a_2 : (\neg do_{a_1}^t \lor \neg do_{a_2}^t)$

4. If action $a$ is applied at step $t$, then $pre(a)$ hold at step $t$:
   $\forall a \in A : \forall p \in pre^+(a) : (do_a^t \rightarrow is^t_p)$, $\forall p \in pre^-(a) : (do_a^t \rightarrow \neg is^t_p)$

5. If action $a$ is applied at step $t$, then $eff(a)$ hold at step $t + 1$:
   $\forall a \in A : \forall p \in eff^+(a) : (do_a^t \rightarrow is^{t+1}_p)$, $\forall p \in eff^-(a) : (do_a^t \rightarrow \neg is^{t+1}_p)$

6. The goal $g$ holds at step $n$: $\forall p \in g : is^n_p$

7. If atom $p$ changes between steps $t$ and $t + 1$, an action which supports this change must be applied at $t$:
   $\forall p \in P : (is^t_p \land \neg is^{t+1}_p) \rightarrow \bigvee_{a \in sprt(\neg p)} do_a^t$, $(\neg is^t_p \land is^{t+1}_p) \rightarrow \bigvee_{a \in sprt(p)} do_a^t$
Planning as Incremental SAT

- Avoid re-encoding entire problem for each \( n = 0, 1, 2, \ldots \):
  Maintain one single, growing formula

- Incremental SAT Solving: For each \( n \),
  - Add clauses (1)–(5), (7) permanently
  - Assume clauses (6)
    - Literals are considered for one single solving attempt, then dropped
  - Let SAT Solver search for a solution
    - Satisfiable? \( \Rightarrow \) Finished
    - Unsatisfiable? \( \Rightarrow \) Continue

- Solver can learn conflicts from unsatisfiable increments
to speed up subsequent increments

- More in “Practical SAT Solving” lecture

- Implementation: edu.kit.aquaplanning.planners.SimpleSatPlanner
Improvements for SAT Planning

- Better semantics for sets of parallel actions
  - (Naïve version: Purely sequential actions)
  - Foreach step: parallel actions may never be interfering
  - Exists step: parallel set of actions may interfere if \( \exists \) a valid ordering, all preconditions are met in the start, all effects hold in the end
  - Relaxed exists step: not even all preconditions must be met in the start
  - Relaxed relaxed exists step: not even all effects must hold in the end

  \[ \Rightarrow \text{Needed techniques: Enabling graph, implication chains} \]

- Better scheduling of makespans: Increase \( n \) in varying intervals, (pseudo-)parallel execution of solvers on different makespans
Hierarchical Task Network (HTN) planning

- Ovals: non-primitive tasks (expand to new task networks by choosing an appropriate method)
- Rectangles: primitive tasks (correspond to classical actions)
- In addition: Methods establish various constraints between tasks
HTN Planning: Solutions

HTN Solution Plan (semi-formal)

A sequence of actions \( \pi = \langle a_1, \ldots, a_n \rangle \) is a solution plan for a ground HTN planning problem \( \mathcal{P} = (P, A, D, s_0, T_0) \), \( T_0 = (T, C) \), if one of the following alternatives holds:

1. \( T \) is empty, and \( n = 0 \).

2. Achieving a primitive task \( t \in T \) meets the constraints \( C \) in \( s_0 \), its corresponding action \( a_1 \) is applicable in \( s_0 \), and \( \pi' = \langle a_2, \ldots, a_n \rangle \) is a solution plan for \( \mathcal{P}' = (P, A, D, \gamma(s_0, a_1), (T \setminus \{t\}, C)) \).

3. Applying a ground method \( m \) of a compound task \( t \in T \) meets the constraints \( C \) in \( s_0 \), and \( \pi \) is a solution plan for \( \mathcal{P}' = (P, A, D, s_0, (T \setminus \{t\} \cup \text{subtasks}(m), C \cup \text{constraints}(m)) \).
Undecidability of HTN Planning

HTN planning is strictly more expressive than classical planning:

**Theorem.** [EHN94]
Given an HTN planning problem $\mathcal{P}$, it is generally **undecidable** whether $\mathcal{P}$ is solvable (i.e. has a solution plan).

- **Consequence:** Can only have **semi-decidable** planning procedures
  - If a plan exists, it will eventually be found
  - If no plan exists, you may never know

- **Proof:** Model an undecidable problem as an HTN planning problem
  $\Rightarrow$ HTN planning cannot be decidable then!

- “Best” candidate: **Post Correspondence Problem**
HTN can simulate Classical Planning

Given classical planning problem \((S, A, s_i, g)\) with \(|\pi| > 0\) for all plans \(\pi\), find an HTN planning problem solving exactly the same problem.

Tasks: \((\text{mainTask}), (\text{anyAction})\)
Methods: \(m_1, m_2, m_a (a \in A)\)

- \(\text{task}(m_1) = (\text{mainTask}), \)
  \(\text{subtasks}(m_1) = \langle (\text{anyAction}), (\text{mainTask}) \rangle, \text{constraints}(m_1) = \emptyset\)

- \(\text{task}(m_2) = (\text{mainTask}), \)
  \(\text{subtasks}(m_2) = \langle (\text{anyAction}) \rangle, \text{constraints}(m_2) = \emptyset\)

- For each \(a \in A\): \(\text{task}(m_a) = (\text{anyAction}), \)
  \(\text{subtasks}(m_a) = \langle a \rangle, \text{constraints}(m_a) = \emptyset\)

Initial task network \((T, C)\):
\(T = \langle (\text{mainTask}) \rangle, C = \{ ((\text{mainTask}), p) \mid p \in g \} \) (after constraints)
SHOP2 Planner

Most popular HTN planner: SHOP2

Algorithm 2 SHOP2 Planning procedure [NAI+03] (simplified, abstract)

1: $\pi = \langle \rangle$; $(T, C) := \text{initial task network}$
2: while TRUE do
3: \hspace{1em} if $T = \emptyset$ then return $\pi$ \hspace{1em} // everything achieved
4: \hspace{1em} $T' := \{t \in T : \text{there is no } t' \text{ such that } t' \prec t \in C\}$
5: \hspace{1em} if $T' = \emptyset$ then return FAILURE \hspace{1em} // no valid tasks to pick from
6: \hspace{1em} if $\exists t \in T' : t$ is primitive and its action $a$ is applicable in $s$ then
7: \hspace{2em} $T := T \setminus \{t\}$; $\pi := \pi \circ a$; $s := \gamma(s, a)$
8: \hspace{1em} else if $\exists t \in T' : t$ is compound and one of its methods $m$ is applicable in $s$ then
9: \hspace{2em} $T := T \setminus \{t\} \cup \{\text{subtasks}(m)\}$
10: \hspace{2em} $C := C \cup \{\text{constraints}(m)\}$
11: \hspace{1em} else return FAILURE
12: \hspace{1em} end if
13: end while

Alternatives: Plan-space HTN planning, SAT-based HTN planning
Improving Plans

A Plan: \(L(P_1, A), M(A, B), M(B, C), M(C, B), L(P_2, B),\)
\(M(B, C), U(P_1, C), U(P_2, C), M(C, B)\)

- It is **Redundant** – some actions may be removed: \(M(B, C), M(C, B)\)
  and then plan will still be valid.

Another Plan: \(L(P_1, A), M(A, B), M(B, C), U(P_1, C), M(C, B),\)
\(L(P_2, B), M(B, C), U(P_2, C)\)

- It is **Suboptimal** but NOT Redundant
- Cannot be improved by only removing actions (must reorder actions)
Removing Redundant Actions

No redundancy: **Perfectly Justified Plan**

- NP-complete to find (proof idea: 3-SAT reduction)
- Approximation 1: Action Elimination
  - Basic Idea: remove an action and see what happens
  - Overall complexity: $\mathcal{O}(n^2p)$
- Approximation 2: Greedy Action Elimination
  - Identify all sets of redundant actions first, then remove the best such set, repeat until no redundant set is found.
  - Complexity $\mathcal{O}(n^3p)$, good for problems with action cost
- Optimal solution: Encode as MaxSAT
  - SAT encoding where only the plan’s actions exist at each step
  - Soft clauses for omitting each of the actions
Beyond Removing Redundant Actions

Sliding windows, Plan de-ordering, Plan neighborhood graph search
Planning in Parallel

Two kinds of approaches

- **Portfolios** – Diversify and Conquer
  - Run several planners in parallel, each working on the entire problem
  - The planner that first finds the solution stops all the planners (“virtual best planner”)
  - You can run the same planner many times but with different heuristics, search strategies, random seeds, settings, etc
  - **PRO:** no dynamic load balancing required, small communication volume, easy to implement, works well for up 8-16 cores.
  - **CON:** overlapping work, hard to scale up (need many diverse planners)

- **Search Space Splitting** – Divide and Conquer
  - Each planner process works on a distinct subset of search space
  - **PRO:** No overlapping work as in Portfolios, can scale better
  - **CON:** Requires expensive load balancing and lots of communication
Enhancing our problem logic

- From STRIPS to ADL: Logic describing the problem gains more expressive power
  - STRIPS: Pure “sets of literals” – Flat conjunctions
    - Structure of expressions: $F \in \{a, \neg a, F_1 \land F_2\}$
      ($a$: atom; $F_1, F_2$: expressions)
  - ADL (Action Description Language): Function-free First Order Logic
    - $F \in \{a, \neg F_1, F_1 \land F_2, F_1 \lor F_2, \forall x F_1(x), \exists x F_1(x)\}$

- Advanced logical constructs in planning
  - Axioms (PDDL: Derived predicates)
    - (:derived (at-house) (or (at-kitchen)(at-livingroom)))
  - Functions (PDDL: Mainly numeric fluents)
    - (decrease (capacity ?t) (* 2 (weight ?p)))
  - Temporal logic (PDDL: (Temporal) Constraints)
    - (always-within 3 (turbo-gear on) (turbo-gear off))
More Non-classical Planning

Numeric Planning

Temporal Planning

Preference-based Planning

- (:preference)s to express soft plan constraints
- (:constraint)s to express hard plan constraints
- (:metric)s involving action cost, numerics, preference violations
Deterministic Planning: Brief Taxonomy

- Common notions in classical planning
  - STRIPS (preconditions, effects, goals)
  - ADL (equality, disjunctive conditions, quantifications)
  - Axioms / derived predicates

- Hierarchical planning
  - Hierarchical task networks

- Numeric planning
  - Numeric fluents (functions) and conditions
  - Advanced (arithmetic) plan quality metrics

- Temporal planning
  - Durative and concurrent actions

- Preference-based planning
  - Preferences / soft plan constraints
  - Plan trajectories / hard plan constraints
Given:
- A set of jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ to be processed
- A set of machines $\mathcal{M} = \{M_1, \ldots, M_m\}$ to process the jobs
- Various constraints and properties
  - Interference/dependency of jobs
  - Compatibility of machines and jobs
  - Efficiency of a machine for a given (type of) job
  - Preemptiveness of jobs (can be interrupted or not)
  - ...

Various Optimization Criteria

Task:
- Find a **Schedule**, i.e., a mapping of jobs to machines and processing times that satisfies the given constraints and is optimal w.r.t. optimization criteria
Graham Notation

A scheduling problem is described by a triplet: $\alpha | \beta | \gamma$ where

- $\alpha$ describes the machine environment (1-2 entries)
- $\beta$ job characteristics (0-many entries)
- $\gamma$ objective function to be minimized (1 entry)

Examples:

- $1|r_j, pmtn|L_{max}$ polynomially solvable:
  Start with the job $J_j$ with the smallest $r_j$ (break ties by smallest $d_j$); as soon as we reach the $r_j$ of a job $J_j$ with smaller $d_j$ than the current jobs due date, we interrupt the current job and switch to that job $J_j$

- $1|r_j|L_{max}$ NP-complete;
  Branch-and-bound by making use of relaxed problem $1|r_j, pmtn|L_{max}$

- $Pm|prec|C_{max}$ with $m \geq n$: Critical path method
Shop Problems

- The Shop Problems are the most commonly used scheduling problems in practice
  - Each job consists of a set of tasks
  - Each task must be executed on a specific machine
  - There can be precedence relations between the tasks

- The 3 kinds of shop problems
  - Job-shop – the tasks within each job are totally ordered (a job is sequence of tasks), often each resource is used at most once per job.
  - Flow-shop – special case of Job-shop, all jobs have identical tasks in the same order (assembly line production)
  - Open-shop – no precedence relations between the tasks,

Graham notation: $Jm||C_{max}, Fm||C_{max}, Om||C_{max}$ for $m$ machines and optimizing makespan, in general NP-hard

Example: Solving Job Shop problems using Crawford SAT encoding
A Markov Decision Process (MDP) is a stochastic system $\Sigma = (S, A, P)$ with a reward function $R$, a cost function $C$, and a utility function $V$. The solution of a MDP is a policy $\pi^*$ which maximizes $E(\pi^*)$.

- Full observability, but probabilistic state transitions
- More abstract than classical planning: Result is not a plan, but an “online plan generator” $\pi^*$
- Procedures to find optimal solutions, according to Bellman equation:
  - Value iteration: apply recursion until (approx.) fixpoint of utility function
  - Policy iteration: solve systems of linear equations to update policy
  - Problem: Iterating over all states. Alternative: heuristic search.
A Partially Observable Markov Decision Process (POMDP) is a stochastic system $\Sigma = (S, A, P)$ with a finite set of observations $O$ with probabilities $P_a(o|s)$ for $a \in A$, $s \in S$, $o \in O$. It must hold that $\sum_{o \in O} P_a(o|s) = 1$.

Procedure of an acting agent in a POMDP:

1. Start with some initial belief state $b$
2. Based on $b$, apply some action $a$ and make an observation $o$
3. Calculate prior belief state $b_a(s) = \sum_{s' \in S} P_a(s|s') \cdot b(s')$
4. Infer new belief state $b' = P_a(s|o) = \frac{P_a(o|s) \cdot b_a(s)}{P_a(o)}$ (Bayes rule)
5. Back to (2)

POMDP Planning: Heuristic search, or “leave planning towards learning”
# Multi-Agent Path Finding (MAPF)

## Input
- A graph with $n$ vertices, usually a grid with obstacles
- A set of $k$ agents each with a start and goal vertex

## The Problem
- An agent can move or wait in each time step
- Find a path for each agent such that paths do not conflict
- The problem is solved by a centralized solver offline

- Additional constraints: No-swap, no-train, in general k-robustness
- Plan quality: Total makespan (max #steps), sum of costs
- NP-hard in general (some polynomial cases exist)
How to solve MAPF?

- Encode in PDDL and use a generic planner
  - Too slow, does not scale well for large problems
- Polynomial Sub-optimal rule-based algorithms
  - Like push\&swap, push\&rotate, bibox, ...
  - Scale better but the solution quality is usually rather bad
- Search based techniques
  - State space search with A* and some heuristics:
    - Cooperative A*, M*, ...
  - Conflict based search – kinda sorta plan-space-search
  - Work well for many “practical” problems
- Reduction based techniques
  - SAT/MaxSat/ILP/CSP/ASP based MAPF
  - Work best for small sized but complex problem instances
Challenges

- We need real-time planning (very fast reactions)
- Continuous, nondeterministic, partially observable environments
- There are usually other (unpredictable) agents in the world (like the human player)
- How to model properly (how to choose the goals)?

What is used instead

- Reactive techniques (Finite state machines, Behavior Trees)
- Machine learning (very rare, Black and White, Creatures series)
What else to expect in the Exam

- Modeling of planning problems
  - How to do it, what to avoid, . . .
  - Given a planning problem, describe how you could model it with classical planning
  - Given a scenario, pick a planning model / style which is best suited

- Planning in practice
  - Grounding: How it works, what it does
  - Name some popular planner(s) and what makes them special
  - Given some realistic scenario, which combinations of planning techniques would you employ?
    - Optimal planning, plan optimization, planning in parallel, . . .

- Topics covered in the exercises
Questions?
The End

Thank you for participating!
It was fun!
References I


