Contents

- Finish SAT Planning
- Improving Plans
  - Removing Redundant Actions
  - Optimize Sub-plans
  - Plan De-ordering
  - Planning Neighborhood Search
- Planning in Parallel (maybe)
  - Parallel A*
  - Parallel Portfolios
  - others
Parallel Plans

Definition

A sequence of sets of actions $P = [A_1, \ldots, A_k]$ is a parallel plan for a planning task $\Pi$ if there is an action ordering function $\sqsubseteq$ such that $[\sqsubseteq(A_1) \oplus \cdots \oplus \sqsubseteq(A_k)]$ is a (sequential) plan for $\Pi$, where $\oplus$ denotes the concatenation of sequences. The sets $A_i$ are called parallel steps and $k$ is called the makespan of $P$.

Example:

- $A_1 = \{\text{loadP1, move-A-B}\}$
- $A_2 = \{\text{loadP2, move-B-C}\}$
- $A_3 = \{\text{unloadP1, unloadP2}\}$
Foreach step semantics

Interfering Actions

- A set of atoms $S$ is consistent if $S$ does not contain $x$ and $\neg x$ for some atom $x$.
- Two actions $a_1 = (p_1, e_1)$ and $a_2 = (p_2, e_2)$ do not interfere if:
  - $p_1 \cup p_2$ is consistent (consistent preconditions)
  - $e_1 \cup e_2$ is consistent (consistent effects)
  - $e_1 \cup p_2$ and $e_2 \cup p_1$ are consistent

Proposition

Let $A$ be a set of actions such that $\forall a_i \neq a_j \in A$ the actions $a_i$ and $a_j$ do not interfere. If there is an ordering $\sqsubseteq$ such that $\sqsubseteq(A)$ transforms the state $s_1$ to $s_2$ then all the possible orderings of $A$ transform $s_1$ to $s_2$.

- It is enough to suppress compatible interfering action pairs.
- For each ordering we get a valid plan, hence the name.
Foreach step semantics

A parallel plan $P = [A_1, \ldots, A_k]$ satisfies the $\forall$-Step semantics if
- each action in $A_j$ is applicable in the state $s_j$,
- the effects of all the actions in $A_j$ are applied in $s_{j+1}$,
- each pair of actions in $A_j$ do not interfere.
- $[\sqcap(A_1) \oplus \cdots \oplus \sqcap(A_k)]$ is a valid plan for some ordering function $\sqsubseteq$.

Example:
- $A_1 = \{\text{loadP1}\}$
- $A_2 = \{\text{move-A-B}\}$
- $A_3 = \{\text{loadP2}\}$
- $A_4 = \{\text{move-B-C}\}$
- $A_5 = \{\text{unloadP1, unloadP2}\}$
Exists step semantics

A parallel plan $P = [A_1, \ldots, A_k]$ satisfies the $\forall$-Step semantics if

- each action in $A_j$ is applicable in the state $s_j$,
- the effects of all the actions in $A_j$ are applied in $s_{j+1}$,
- each pair of actions in $A_j$ do not interfere,
- $[\triangledown(A_1) \oplus \cdots \oplus \triangledown(A_k)]$ is a valid plan for some ordering function $\triangledown$.

Example:

- $A_1 = \{\text{loadP1, move-A-B}\}$
- $A_2 = \{\text{loadP2, move-B-C}\}$
- $A_3 = \{\text{unloadP1, unloadP2}\}$

Only 3 steps required
Relaxed Exists step semantics

A parallel plan $P = [A_1, \ldots, A_k]$ satisfies the $\forall$-Step semantics if

- each action in $A_j$ is applicable in the state $s_j$,
- the effects of all the actions in $A_j$ are applied in $s_{j+1}$,
- each pair of actions in $A_j$ do not interfere,
- $[\sqsubseteq(A_1) \oplus \cdots \oplus \sqsubseteq(A_k)]$ is a valid plan for some ordering function $\sqsubseteq$.

Example:

- $A_1 = \{\text{loadP1, move-A-B, loadP2}\}$
- $A_2 = \{\text{move-B-C, unloadP1, unloadP2}\}$

Only 2 steps required!

- What can we drop?
Relaxed Relaxed Exists step semantics

A parallel plan $P = [A_1, \ldots, A_k]$ satisfies the $\forall$-Step semantics if

- each action in $A_j$ is applicable in the state $s_j$,
- the effects of all the actions in $A_j$ are applied in $s_{j+1}$,
- each pair of actions in $A_j$ do not interfere.
- $[\sqsubseteq(A_1) \oplus \cdots \oplus \sqsubseteq(A_k)]$ is a valid plan for some ordering function $\sqsubseteq$.

Example:

- $A_1 = \{\text{loadP1, move-A-B, loadP2, move-B-C, unloadP1, unloadP2}\}$

Only 1 steps required! We probably cannot do better than that :}

Tomáš Balyo, Dominik Schreiber – Planning and Scheduling
How to encode the (relaxed) Exists step semantics into SAT?

Overview of Basic Ideas

- The SAT encoding only approximates the semantics, i.e., the satisfiability of the constructed formula $F_k$ implies the existence of a $k$-step plan (not vice versa).
- The actions are ranked arbitrarily:
  - goal is to guess the order of actions in the final plan
  - heuristic: use cycle–ignoring topological sorting on the enabling graph
  - better ranking heuristics wanted (Master thesis anyone?)
- The encoding allows only lower ranking actions before higher ranking ones in a step.
- The encoding uses implication chains.
How to encode the (relaxed) Exists step semantics into SAT?

Overview of Basic Ideas

- The SAT encoding only approximates the semantics, i.e., the satisfiability of the constructed formula $F_k$ implies the existence of a $k$-step plan (not vice versa).
- The actions are ranked arbitrarily:
  - goal is to guess the order of actions in the final plan
  - heuristic: use cycle–ignoring topological sorting on the enabling graph
  - better ranking heuristics wanted (Master thesis anyone?)
- The encoding allows only lower ranking actions before higher ranking ones in a step.
- The encoding uses implication chains.
Enabling Graph

Definition

The *enabling graph* $G$ for a set of actions $A$ is a directed graph where vertices represent actions and there is an edge $(a, a')$ if $a$ supports $a'$, i.e.,

$$G = (A, \{a \rightarrow a' | a, a' \in A; \text{eff}(a) \cap \text{pre}(a') \neq \emptyset\})$$
The actions are ranked using cycle–ignoring topological sorting on the enabling graph

\[
\text{topologicalRanking}(O)
\]

T1 \hspace{1em} \text{global } lastRank := 0

T2 \hspace{1em} \text{global } visited := \{False, \ldots, False\}

T3 \hspace{1em} \textbf{foreach } a \in O \textbf{ do}

T4 \hspace{1em} \text{rankAction}(a)

\[
\text{rankAction}(a)
\]

R1 \hspace{1em} \text{if } visited[a] = False \text{ then}

R2 \hspace{1em} visited[a] := True

R3 \hspace{1em} \textbf{foreach } s \in \text{supportingActions}(a) \textbf{ do}

R4 \hspace{1em} \text{rankAction}(s)

R5 \hspace{1em} r(a) := lastRank

R6 \hspace{1em} lastRank := lastRank + 1
Encoding – these clauses same as before

1. The initial state must hold at $t = 0$.

   $$\forall p \in s_0 : is^0_p \quad \forall p \notin s_0 : \neg is^0_p$$

2. The goal state must hold at $t = n$.

   $$\forall p \in G : is^n_p$$

3. If atom $p$ changes between steps $t$ and $t + 1$, an action which supports this change must be applied at $t$:

   $$\forall t \in \{0, \ldots, n - 1\}, \forall p \in P :$$

   $$\left( (is^t_p \land \neg is^{t+1}_p) \rightarrow \bigvee_{a \in \text{support}(\neg p)} do^t_a \right)$$

   $$\left( (\neg is^t_p \land is^{t+1}_p) \rightarrow \bigvee_{a \in \text{support}(p)} do^t_a \right)$$
Encoding Clauses – Preconditions

These clauses are added $\forall t \in \{0, \ldots, n-1\}, \forall a \in A$

For Foreach and Exists Step:

4. If action $a$ is applied at step $t$, then $pre(a)$ hold at step $t$:

$$\forall p \in pre^+(a) : (do^t_a \rightarrow is^t_p)$$

$$\forall p \in pre^-(a) : (do^t_a \rightarrow \neg is^t_p)$$

For Relaxed and Relaxed Relaxed Exists Step:

4. If action $a$ is applied at step $t$, then $pre(a)$ hold at step $t$ or some supporting action happens before:

$$\forall p \in pre^+(a) : (do^t_a \rightarrow (is^t_p \lor do^t_{a_1} \lor \ldots \lor do^t_{a_k}))$$

$$\forall p \in pre^-(a) : (do^t_a \rightarrow (\neg is^t_p \lor do^t_{a_1}^{1-p} \lor \ldots \lor do^t_{a_k}^{1-p}))$$

where $a^*_p$ are supports for $p$ with lower rank than $a$. 
Encoding Clauses – Effects

For Foreach, Exists and Relaxed Exists Step:

5 If action \( a \) is applied at step \( t \), then \( \text{eff}(a) \) hold at step \( t + 1 \) or some other action sets in later:

\[
\forall p \in \text{eff}^+(a) : (do_a^t \rightarrow is_p^{t+1})
\]
\[
\forall p \in \text{eff}^-(a) : (do_a^t \rightarrow \neg is_p^{t+1})
\]

For Relaxed Relaxed Exists Step:

5 If action \( a \) is applied at step \( t \), then \( \text{eff}(a) \) hold at step \( t + 1 \) or some other action sets in later:

\[
\forall p \in \text{eff}^+(a) : (do_a^t \rightarrow (is_p^{t+1} \lor do_{a_1}^t \lor \ldots \lor do_{a_k}^t))
\]
\[
\forall p \in \text{eff}^-(a) : (do_a^t \rightarrow (\neg is_p^{t+1} \lor do_{a_1}^t \lor \ldots \lor do_{a_k}^t))
\]

where \( a_p^* \) are supports for \( p \) and \( \neg p \) with higher rank than \( a \).
Dealing with action interference

Easy in Foreach step

- add \( \neg \text{do}_{a_i} \lor \neg \text{do}_{a_j} \) for each pair of interfering actions \( a_i, a_j \)

How about (Relaxed)* Exists step? What could go wrong?

- In Exists Step: one actions destroys the precondition for another action coming later

- In (Relaxed)+ Exists Step:
  
  - one actions destroys the precondition for another action
  - one actions destroys the effect of another action
  - one actions \( a_1 \) sets up the precondition for \( a_2 \) but then some action \( a_3 \) between them destroys it again
  - one actions \( a_1 \) set up the precondition for \( a_2 \) but then some action \( a_3 \) between them destroys it again, but then \( a_4 \) between \( a_3 \) and \( a_2 \) may restore it back ...

  ...

Dealing with action interference

Easy in Foreach step

- add \( \lnot do_{a_i} \lor \lnot do_{a_j} \) for each pair of interfering actions \( a_i, a_j \)

How about (Relaxed)* Exists step?
What could go wrong?

- **In Exists Step**: one actions destroys the precondition for another action coming later

- **In (Relaxed)+ Exists Step**: 
  - one actions destroys the precondition for another action
  - one actions destroys the effect of another action
  - one actions \( a_1 \) sets up the precondition for \( a_2 \) but then some action \( a_3 \) between them destroys it again
  - one actions \( a_1 \) set up the precondition for \( a_2 \) but then some action \( a_3 \) between them destroys it again, but then \( a_4 \) between \( a_3 \) and \( a_2 \) may restore it back ...
We have a chain for each positive and negative atom.

- $h_i$ are helper variables, $a_i = do_{a_i}$, full arrow is implication ($h_i \implies a_j$) dashed arrow is negative implication ($h_i \implies \neg a_j$)
- an action $a$ activates the chain for $\neg p$ if $p \in eff(a)$
- the chain for $\neg p$ deactivates an action $a$ if $p \in pre(a)$
- if one action activates the chain all actions with higher rank that require that atom in their preconditions are disabled.
- $a_1$, $a_2$, $a_3$ and $a_5$ are opponents of the given atom $p$ (they have $\neg p$ as an effect).
- $a_3$ and $a_6$ require $p$ (they have $p$ in their preconditions).
- $a_3$ both requires and opposes $p$
  - Example: $p = truckAt_A$, $a_3 = move(A, B)$
- we need one helper variable $h$ in $p$'s chain for each action that requires $p$
- \(a_1, a_2, a_3\) and \(a_5\) are opponents of the given atom \(p\) (they have \(\neg p\) as an effect).
- \(a_3, a_4\) and \(a_6\) require \(p\) (they have \(p\) in their preconditions).
- \(a_3\) both requires and opposes \(p\)
- \(a_4\) supports \(p\) (has \(p\) as an effect)
- \(a_4\) can break the chain between \(h_3\) and \(h_4\): \(h_3 \implies (h_4 \lor a_4)\)
each involved action gets a helper variable (except the last one)

For each atom \( p \) at each time step \( t \)

1. \( (h_{j-1} \rightarrow h_j) \) for each \( j \) such that \( a_j \) is not a support for \( p \)
2. \( (h_{j-1} \rightarrow (h_j \lor do_{a_j})) \) for each \( j \) such that \( a_j \) is a support for \( p \)
3. \( (do_{a_j} \rightarrow h_j) \) for each \( j \) such that \( a_j \) is an opponent for \( p \)
4. \( (h_{j-1} \rightarrow \neg do_{a_j}) \) for each \( j \) such that \( a_j \) requires for \( p \)
Extracting the Plan

How to get the plan from the satisfying assignment of the formula?

- Get the parallel plan (sequence of sets of actions)
  - check which $do^t_{a_i}$ variables are true in each step
- Turn it into a regular plan
  - order the actions in each step according to the ranking
Extracting the Plan

How to get the plan from the satisfying assignment of the formula?

- Get the parallel plan (sequence of sets of actions)
  - check which $do^t_{a_i}$ variables are true in each step
- Turn it into a regular plan
  - order the actions in each step according to the ranking
The better you rank the actions (predict their ordering)
- the least SAT solving steps you need
- the faster and with less memory you find a plan

For some domains 1 step is enough
- if each actions is required only once
- if you can guess the order of actions

Number of variables/ clauses?
- Homework :)

Future Work:
- better ranking heuristics!
How about chaining “n++” part of the algorithm?

Why?
Runtime Profile

Evaluation times: gripper10

<table>
<thead>
<tr>
<th>time in secs</th>
<th>0</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
<th>350</th>
<th>400</th>
<th>450</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>time points</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Karlsruhe Institute of Technology
Runtime Profile

Evaluation times: blocks22

![Histogram showing evaluation times for blocks22](chart.png)
Scheduling Strategies by Rintanen

- Classical scheduling: solve 1,2,3,... sequentially

- Algorithm A
  - start $n$ solvers in parallel solving 1,2,...,$n$
  - if a formula found unsat continue with the smallest not solved yet
  - can gets past hard UNSAT formulas if $n$ is big enough
  - in the worst case $n$ times slower than sequential
  - bigger formulas – higher memory requirement
  - skipping lengths is ok, i.e., solve 10,20,30,...,10n

- Algorithm B – geometric
  - start $n$ solvers in parallel solving 1,2,...,$n$
  - solver solving step $k + 1$ has a time limit $g$ times less than the time limit of the solver solving $k$, for some constant $g < 1$
  - focus on solving the smaller steps
  - can choose higher $n$ than before
Runtime Profile

Finding a plan for blocks22 with Algorithm B

![Graph showing runtime profile over time points. The x-axis represents time points ranging from 40 to 90, and the y-axis represents time in seconds, ranging from 0 to 45. The graph includes a line and a bar chart illustrating the distribution of runtime over different time points.](image-url)
Algorithm C – exponential
- start $n$ solvers in parallel solving 1,2,4,8,...
- works surprisingly well
- easy to run out of memory
- finds very long plans
- solves dozens of problems unsolved by the previous strategies
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)
Improving Plans

A Plan: $L(P1,A), M(A,B), M(B,C), M(C,B), L(P2,B), M(B,C), U(P1,C), U(P2,C), M(C,B)$

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

Another Plan: $L(P1,A), M(A,B), M(B,C), U(P1,C), M(C,B), L(P2,B), M(B,C), U(P2,C)$

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

- A plan that is not redundant is called a **Perfectly Justified Plan**.

**Hardness**

The problem to find a perfectly justified sub-plan of a given plan is **NP-complete**.

**Proof Idea:**

- Reduce 3-SAT to existence of a proper sub-plan of a given plan.
Proof

Let $\mathcal{F}$ be a 3-SAT CNF formula with $n$ variables and $k$ clauses.

Predicates:
- for each SAT variable $x_i$ two PDDL predicates: $T_i$ and $F_i$
- for each clause $c_i$ one predicate $C_i$
- for each literal $\{x_i, \neg x_i\} \in c_j$ a predicate $L_{i,j}$

Actions:
- $\text{init}_i = (\{\}, \{T_i, \neg F_i\})$
- $\forall x_i \in c_j : \text{true}_i = (\{T_i\}, \{C_j, L_{i,j}\})$
- $\forall \neg x_i \in c_j : \text{false}_i = (\{F_i\}, \{C_j, L_{i,j}\})$
- mid = ($\{\neg F_1, \ldots \neg F_n\}$, $\{F_1, \ldots, F_n, \text{all } \neg L_{i,j}\}$)
Proof

- $\text{init}_i = (\{\}, \{T_i, \neg F_i\})$
- $\forall x_i \in c_j: \text{true}_{i,j} = (\{T_i\}, \{C_j, L_{i,j}\})$
- $\forall \neg x_i \in c_j: \text{false}_{i,j} = (\{F_i\}, \{C_j, L_{i,j}\})$
- mid = ($\{\neg F_1, \ldots, \neg F_n\}$, $\{F_1, \ldots, F_n, \text{all } \neg L_{i,j}\}$)

Initial State:

- $F_1, \ldots, F_n, \neg T_1, \ldots, \neg T_n, \neg C_1, \ldots, \neg C_k, \text{all } L_{i,j}$

Goal:

- $C_1, \ldots C_k, \text{all } L_{i,j}$

Redundant Plan:

- $\pi = \text{init}_1, \ldots, \text{init}_n, \text{mid}, \forall \text{true}_{i,j}, \forall \text{false}_{i,j}$
Proof Conclusion

- Let $\pi' \subset \pi$ be a proper sub-plan of $\pi$ if it exists.
- $\pi'$ does not contain the “mid” action because:
  - if we remove any of the “init” actions “mid” has unsat preconditions
  - if we remove any “true” or “false” action “mid” interferes with the goal
- therefore $\pi'$ must have the form
  - $\pi' = \text{init}_{k_1}, \ldots, \text{init}_{k_m}, \text{some true}_i,j, \text{some false}_i,j$
- The following two statements are equivalent
  - $\pi'$ is a proper sub-plan of $\pi$ that achieves the goal
  - $x_{k_1} = \ldots = x_{k_m} = \text{True}, x_{k_m+1} = \ldots = x_n = \text{False}$ is satisfying assignment for $\mathcal{F}$
- Therefore a proper sub-plan exists if and only the formula is satisfiable.
Action Elimination Algorithm

- Basic Idea: remove an action and see what happens
- Remove an action and all actions that don’t have fulfilled preconditions anymore, check if the goal is still achieved.
- Check complexity for one action: $O(np)$, $n = |P|$, $p$ is the maximum number of preconditions.
- Overall complexity: $O(n^2p)$
Action Elimination Problems

- fly(A, E), fly(E, A), fly(A, B), fly(B, C), fly(C, D), fly(D, E)

Remove These to get a non-optimal but also non-redundant plan

Remove These to get an optimal and non-redundant plan

- The order of removing redundant actions matters
Greedy Action Elimination

Identify all the sets of redundant actions first and then remove the best such set, repeat until no redundant set is found.

Complexity $O(n^3p)$, good for problems with action cost
Removing All Redundant Actions

- It is NP complete, NP tools are justified.
- We will encode plan redundancy to Partial MaxSAT

Partial MaxSAT Definition
- Like in SAT we have Boolean variables and clauses (CNF formula)
- BUT the clauses are divided into 2 categories
  - Hard clauses – must be satisfied at all costs
  - Soft clauses – we would like to satisfy as many as possible
- GOAL: find a truth assignment that satisfies ALL the hard clauses and AS MANY AS POSSIBLE soft clauses.
Removing All Redundant Actions

- It is NP complete, NP tools are justified.
- We will encode plan redundancy to Partial MaxSAT

**Partial MaxSAT Definition**

- Like in SAT we have Boolean variables and clauses (CNF formula)
- BUT the clauses are divided into 2 categories
  - Hard clauses – must be satisfied at all costs
  - Soft clauses – we would like to satisfy as many as possible
- GOAL: find a truth assignment that satisfies ALL the hard clauses and AS MANY AS POSSIBLE soft clauses.
Let $\pi$ be possibly redundant plan for planning problem $\Pi$.

- SAT encoding similar to the simple encoding for plan existence from lecture 6 and 7
  - We know the length of the plan (makespan) already $n = |\pi|$
  - Instead of encoding each possible action at each step we only encode $\pi[i]$ or noop (no operation, empty action) at step $i$.

- all the above mentioned will be Hard clauses.

- we add a soft unit clause for each step that is satisfied if the noop action is taken

- the MaxSAT solver will find a valid sub-plan with as many actions removed as possible.
Beyond Removing Redundant Actions

- Cut out a subsection of the plan
- Construct a planning problem based on the cut:
  - initial state = state before the cut,
  - goal = preconditions of actions after the cut
- Find an optimal plan
- Replace cut with the opt. plan
- Reduced search space, known upper bound, easy
How to choose the cut?

- Simplest but efficient approach – Sliding Windows
  - The cut is defined by a sliding windows of increasing size
  - Windows may or may not overlap
  - Gives a simple “anytime” plan optimization algorithm
  - Eventually the window will be large enough to cover the entire plan
    - then an optimal plan will be found

- More sophisticated approach – find a cut where the difference between start and end is higher than some heuristic estimation
Plan De-ordering

- Decompose the plan into blocks, a block is like a planning problem (has initial state and goals).
- Solve each block optimally individually.
Plan Neighborhood Graph Search

- Standard State space graph, vertex = state, edge = action
- Expand the states in the plan up to some exploration limit $L$
  - Anytime algorithm: keep increasing $L$ and find better plans
- Use a shortest path algorithm (like Dijkstra) to find optimal plan
- Works well as postprocessing for greedy heuristic planners
  - They follow narrow paths in search space guided by their heuristic.
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
  - 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 to 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
Parallel Portfolios

- Original Inspiration: VBS (Virtual Best Solver)
- What if we could somehow guess which planner (heuristic) is best for a given planning problem?
- We cannot, but we can calculate it after trying out everything

![Graph showing time in seconds vs. problems]
Parallel Portfolios

- Original Inspiration: VBS (Virtual Best Solver)
  - What if we could somehow guess which planner (heuristic) is best for a given planning problem?
  - We cannot, but we can calculate it after trying out everything

- If we have many cores we run each planner in parallel, The system will behave like the VBS

- We need a **diverse** collection of planners
  - no problem to find a small number of them
  - difficult if you need hundreds

- This approach dominates the IPC\(^1\) parallel track, where the competition is usually run on 8 core machines.

\(^1\)International Planning Competition
Hash Distributed A* (HDA*) – Basic Idea
- Each search space node (state) is assigned to one process
- Assignment based on a hash function (random but deterministic)

Algorithm: Same as any forward or backwards search but:
- when a new node is generated we do not add it to the open list
- we calculate the hash value and send it to the appropriate process
- we receive nodes from other processes and add to our open list
- we select a node from the open list, expand and repeat

Implementation Details
- Exchange nodes not one by one but in batches
- What hash function to use? – next slide
The Zobrist Hash Function

- Invented in 1970 by Albert L. Zobrist, commonly used in the game tree search community (Checkers, Chess, Go, etc)
- Suppose we want a $k$-bit hash code
  - Assign a random $k$-bit sequence (code) $c(a)$ to each atom $a$
  - $h(S) = \bigoplus_{a \in S} c(a)$ where $\bigoplus$ means XOR
  - basically we just XOR the codes of all true atoms in the state
- useful in sequential setting too (hash visited states for duplicate detection)
Optimality of HDA*

- Even if the heuristic function is admissible, parallel A search may sometimes have to re-open a state in the closed list.
  - For example, a process may receive many identical states with various initial state distance
  - Therefore we cannot stop when we first reach the goal
  - We must make sure that no other process can reach the goal with a cheaper plan
  - If each process confirms that there is no node in their open list with a lower f-value we can terminate.
Other Approaches

- **SAT-Based Planning**
  - Use a parallel SAT solver
  - Compute several makespans in parallel

- **Static Load Balancing – Cube and Conquer**
  - Run a depth limited Breadth first search (or limited A*)
  - Save all the leaf node into a queue
  - Split the nodes in the queue randomly between the processors
  - The processors will continue the search from these nodes
  - The queue should contain around $10^3$ as many nodes as there are processors to achieve good load balancing

- **Hybrid approach – Divide, Diversify and Conquer**
  - Split the search space but allow some overlap
The End

Next Week: Advanced Planning Models
I hope you get lots of presents under the [generic] tree :)}