Contents

- Improving Plans
  - Removing Redundant Actions
  - Optimize Sub-plans
  - Plan De-ordering
  - Planning Neighborhood Search

- Planning in Parallel
  - Parallel A*
  - Parallel Portfolios
  - others
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: \( L(P_1, A), M(A, B), M(B, C), M(C, B), L(P_2, B), M(B, C), U(P_1, C), U(P_2, C), M(C, B) \)
  - 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)
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:
- $\text{init}_i = (\{\}, \{T_i, \neg F_i\})$
- $\forall x_i \in c_j : \text{true}_i = (\{T_i\}, \{C_j, L_{i,j}\})$
- $\forall \neg x_i \in c_j : \text{false}_i = (\{F_i\}, \{C_j, L_{i,j}\})$
- $\text{mid} = (\{
\neg F_1, \ldots, \neg F_n\}, \{F_1, \ldots, F_n, \text{all} \ \neg L_{i,j}\})$
Proof

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

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:
- \( \pi = \text{init}_1, \ldots, \text{init}_n, \ \text{mid}, \ \forall \ \text{true}_{i,j}, \ \forall \ \text{false}_{i,j} \)
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 $F$

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: $O(n^2 p)$

```
ActionElimination ($\Pi$, $P$)

\textbf{AE01}
\begin{align*}
& s := s_I \\
& i := 1
\end{align*}
\begin{algorithmic}
\REPEAT
\STATE mark($P[i]$)
\STATE $s' := s$
\FOR{$j := i + 1 \text{ to } |P|$}
\IF{applicable($P[j], s'$)}
\STATE $s' := \text{apply}(P[j], s')$
\ELSE
\STATE mark($P[j]$)
\ENDIF
\ENDFOR
\IF{goalSatisfied($\Pi$, $s'$)}
\STATE $P := \text{removeMarked}(P)$
\ELSE
\STATE unmarkAllActions()
\STATE $s := \text{apply}(P[i], s)$
\STATE $i := i + 1$
\ENDIF
\UNTIL{$i > |P|$}
\RETURN $P$
\end{algorithmic}
```
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

```plaintext
evaluateRemove (Π, P, k)
s := s_1
for i := 1 to k - 1 do
    s := apply(P[i], s)
cost := C(P[k])
for i := k + 1 to |P| do
    if applicable(P[i], s) then
        s := apply(P[i], s)
    else
        cost := cost + C(P[i])
if goalSatisfied(Π, s) then
    return cost
else
    return -1

remove (P, k)
s := s_1
P' := [] // empty plan
for i := 1 to k - 1 do
    s := apply(P[i], s)
P' := append(P', P[i])
for i := k + 1 to |P| do
    if applicable(P[i], s) then
        s := apply(P[i], s)
P' := append(P', P[i])
if goalSatisfied(Π, s) then
    return P'
else
    return -1

greedyActionElimination (Π, P)
repeat
    bestCost := 0
    bestIndex := 0
    for i := 1 to |P| do
        s := apply(P[i], s)
P' := append(P', P[i])
cost := evaluateRemove(Π, P, i)
        if cost ≥ bestCost then
            bestCost := cost
            bestIndex := i
            P := remove(P, bestIndex)
    until bestIndex = 0
return P
```
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.
Encoding Plan Redundancy

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
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
Divide and Conquer – HDA*

- 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 $a$
  - $h(S) = \oplus_{a \in S} c(a)$ where $\oplus$ 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)
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 Year: Temporal Planning and Advanced PDDL Features
I hope you get lots of presents under the [generic] tree :)

Merry 
Generic Holiday