# Artificial Intelligence Planning

## Advanced Heuristics

**Artificial Intelligence Planning**

- **Advanced Heuristics**

# The FF Planner

- performs forward state-space search (A* / EHC)
- relaxed problem heuristic ($h^{FF}$)
  - – construct relaxed problem: ignore delete lists
  - – solve relaxed problem (in polynomial time)
    - chain forward to build a relaxed planning graph
    - chain backward to extract a relaxed plan from the graph
  - – use length of relaxed plan as heuristic value
- pruned search with helpful actions

**The FF Planner**

•a state-of-the-art planner that uses an efficient and accurate heurstic

# Overview

- Simple Planning Graph Heuristics
- Pattern Database Heuristics
- The FF Planner

**Overview**

**➢Simple Planning Graph Heuristics**

**•Pattern Database Heuristics**

**•The FF Planner**

# Forward State-Space Search with A*

- A* is optimally efficient: For a given heuristic function, no other algorithm is guaranteed to expand fewer nodes than A*.
- room for improvement: use better heuristic function!

**Forward State-Space Search with A***

•**A* is optimally efficient: For a given heuristic function, no other algorithm is guaranteed to expand fewer nodes than A*.**

- •all planning algorithms seen so far use search
- •given an admissible heuristic and the need for a minimal length plan, we cannot do better than A*
- •caveats: only have non-admissible heuristic; do not need optimal solution; not enough memory

•**room for improvement: use better heuristic function!**

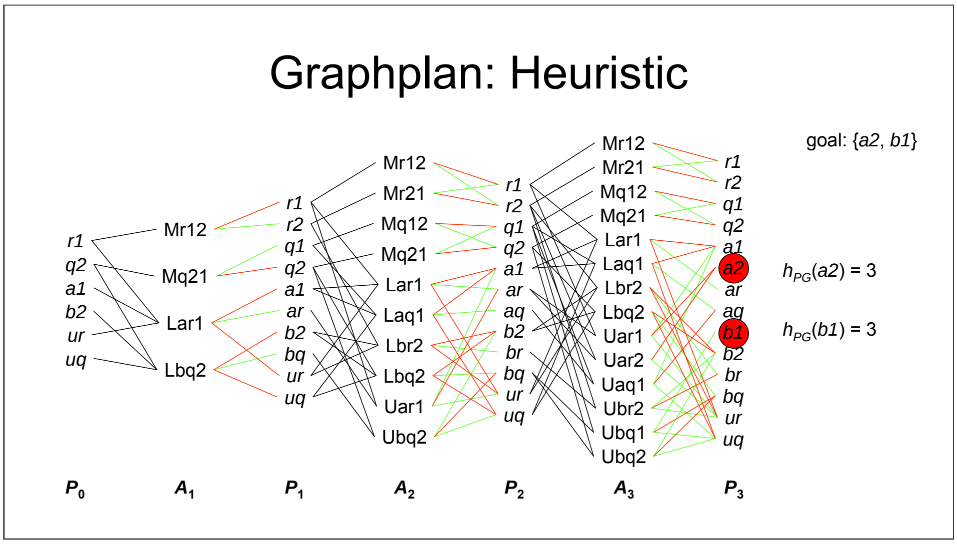
- •perfect heuristic uses linear time and memory
- •often: expensive but more accurate heuristic works better

# Planning Graph Heuristics

- basic idea: use reachability analysis as a heuristic for forward search
  - $P = (A, s_i, g)$ be a propositional planning problem and $G = (N, E)$ the corresponding planning graph
  - $g = \{g_1, \ldots, g_n\}$
  - $g_k$, $k \in [1,n]$, is reachable from $s_i$ if there is a proposition layer $P_g$ such that $g_k \in P_g$
  - in proposition layer $P_m$: if $g_k$ not in $P_m$ then $g_k$ not reachable in $m$ steps
- define (admissible) $h_{PG}(g_k) = m$ for reachable $\{g_k\}$

**Planning Graph Heuristics**

- **basic idea: use reachability analysis as a heuristic for forward search**
  - **$P = (A, s_i, g)$ be a propositional planning problem and $G = (N, E)$ the corresponding planning graph**
  - **$g = \{g_1, \ldots, g_n\}$**
  - **$g_k$, $k \in [1,n]$, is reachable from $s_i$ if there is a proposition layer $P_g$ such that $g_k \in P_g$**
    - reverse statement:
  - **in proposition layer $P_m$: if $g_k$ not in $P_m$ then $g_k$ not reachable in $m$ steps**
    - look for first proposition layer in which $g_k$ appears
- **define $h_{PG}(g_k) = m$ for reachable $g_k$**
  - works only for single goal condition
  - inaccurate if multiple actions from preceding layers are required (but need at least one action from each layer)




**Graphplan: Heuristic**

•**goal: {*a2*, *b1*}**

- goal consists of two propositions

•***hPG*(*a2*) = 3**

- first proposition layer in which *a2* holds

•***hPG*(*b1*) = 3**

- first proposition layer in which *b1* holds

# Multiple Goal Conditions

- $P = (A,s_i,g)$, $g = \{g_1,\ldots,g_n\}$
- option 1: take the maximum
  - still admissible
  - bad accuracy, especially for independent goals
- option 2: add the values
  - no longer admissible
  - inaccurate for dependent goals

**Multiple Goal Conditions**

•**$P = (A,s_i,g)$, $g = \{g_1,\ldots,g_n\}$**

•**option 1: take the maximum**

  •**still admissible**

  •**bad accuracy, especially for independent goals**

•**option 2: add the values**

  •**no longer admissible**

  •**inaccurate for dependent goals**

    •example dependency between at and occupied relations

•note: no mutex relations required

# Overview

- Simple Planning Graph Heuristics
- Pattern Database Heuristics
- The FF Planner

**Overview**

**•Simple Planning Graph Heuristics**

**➢Pattern Database Heuristics**

**•The FF Planner**

## Sub-Problems and Heuristics

| | | |
|---|---|---|
| * | 2 | 4 |
| * | | * |
| * | 3 | 1 |

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | * |
| * | * | * |

cost of the optimal solution of sub-problem
≤ cost of the optimal solution of complete problem

**Sub-Problems and Heuristics**

**•cost of the optimal solution of sub-problem ≤ cost of the optimal solution of complete problem**

•sub-problem here: move tiles 1 to 4 into their correct positions

•but: must compute heuristic; search is expensive

•size of “abstract” search space is smaller

## Pattern Databases

- idea: pre-compute and store the solution costs for all possible sub-problems in database
- computing heuristic = DB lookup
- construct DB by searching backwards from the goal state and recording costs
  - very expensive operation, but needs to be computed only once

**Pattern Databases**

**•idea: pre-compute and store the solution costs for all possible sub-problems in database**

**•computing heuristic = DB lookup**

**•construct DB by searching backwards from the goal state and recording costs**

**•very expensive operation, but needs to be computed only once**

•size of DB: depends on sub-problem

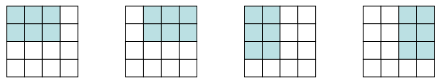
•for 8-puzzle: permutations of *-tiles irrelevant: saving factor 4! = 24 over search space size

•permutations irrelevant, but moves do count towards solution cost

•results achieved: pattern databases give better heuristic values than e.g. Manhattan distance

## Choosing Patterns

- choose such that pattern DB fits into memory (and still leaves space for search algorithm)
- exploit symmetry and use composite heuristic



**Choosing Patterns**

**•choose such that pattern DB fits into memory (and still leaves space for search algorithm)**

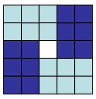
•patterns for 8-puzzle: irrelevant as whole search space fits into memory; different for 15/24-puzzles

**•exploit symmetry and use composite heuristic**

•symmetry: example: position of 6 tiles in 15-puzzle can be re-used in 8 sub-problems

## Disjoint Pattern Databases

- Can we add the values instead of taking the maximum? – No, because the solutions to the different sub-problems share moves.
- *idea: record just the cost of moving the non-*-tiles in the pattern DB*
- sum is admissible heuristic if patterns do not overlap



patterns in the 24-puzzle

**Disjoint Pattern Databases**

**•Can we add the values instead of taking the maximum? – No, because the solutions to the different sub-problems share moves.**

•no, if we still want an admissible heuristic

•***idea: record just the cost of moving the non-*-tiles in the pattern DB***

**•sum is admissible heuristic if patterns do not overlap**

•picture: 24-puzzle with pattern consisting of 6 tiles, non-overlapping, with symmetric re-usability

•disjoint pattern DBs currently state of the art (for 24-puzzle);

•not applicable to every problem yet (e.g. Rubik's cube);

# Planning with Pattern Databases

- divide set of all state propositions into mutually exclusive (disjoint) groups: $G_1 \ldots G_k$
- construct abstract problem spaces
  - modified goals: goals from even groups + goals from odd groups
  - modify operators: intersect preconditions/effects with corresponding groups
- construct pattern database

- result:
  - heuristic computes in constant time (hash table lookup)
  - pattern database is disjoint
  - pattern database slow to compute, but reusable
  - reusability is limited (e.g. cannot change goal or increase number of containers)

**Planning with Pattern Databases**

•**divide set of all state propositions into mutually exclusive (disjoint) groups**

- •example: r1 and r2 in same group
- •usually several ways to do this
- •need additional symbol "true" for groups that may not hold

•**construct abstract problem spaces**

- •divide groups, e.g. even and odd groups
- •**modified goals: goals from even groups + goals from odd groups**
- •**modify operators: intersect preconditions/effects with corresponding groups**

•**construct pattern database**

- •use breadth-first backward search in abstract spaces
- •note: search tree in abstract space shrinks exponentially

•**result:**

- •heuristic computes in constant time (hash table lookup)
- •pattern database is disjoint
- •pattern database slow to compute, but reusable
- •reusability is limited (e.g. cannot change goal or increase number of containers)

## Overview

- Simple Planning Graph Heuristics
- Pattern Database Heuristics
- The FF Planner

**Overview**

**•Simple Planning Graph Heuristics**

**•Pattern Database Heuristics**

**➢The FF Planner**

# The FF Planner

- performs forward state-space search (A* / EHC)
- relaxed problem heuristic ($h^{FF}$)
  - construct relaxed problem: ignore delete lists
  - solve relaxed problem (in polynomial time)
    - chain forward to build a relaxed planning graph
    - chain backward to extract a relaxed plan from the graph
  - use length of relaxed plan as heuristic value
- pruned search with helpful actions

**The FF Planner**

- **performs forward state-space search (A* / EHC)**
  - EHC: commit first to better state; does not work well if state space has dead ends
- **relaxed problem heuristic ($h^{FF}$)**
  - **construct relaxed problem: ignore delete lists**
    - Joerg's example: have a beer, drink the beer, have the beer in tummy, still have a beer!
  - **solve relaxed problem (in polynomial time)**
    - **chain forward to build a relaxed planning graph**
    - **chain backward to extract a relaxed plan from the graph**
  - **use length of relaxed plan as heuristic value**
- **pruned search with helpful actions**
  - use information gained during the computation of the heuristic value

## Relaxed Planning Problem: Example

- move(*r*,*l*,*l'*)
  - precond: at(*r*,*l*), adjacent(*l*,*l'*)
  - effects: at(*r*,*l'*), ¬at(r,*l*)
- load(*c*,*r*,*l*)
  - precond: at(*r*,*l*), in(*c*,*l*), unloaded(*r*)
  - effects: loaded(*r*,*c*), ¬in(*c*,*l*), ¬unloaded(*r*)
- unload(*c*,*r*,*l*)
  - precond: at(*r*,*l*), loaded(*r*,*c*)
  - effects: unloaded(*r*), in(*c*,*l*), ¬loaded(*r*,*c*)

**Relaxed Planning Problem: Example**

**•move(*r*,*l*,*l'*)**

**•precond: at(*r*,*l*), adjacent(*l*,*l'*)**

**•effects: at(*r*,*l'*), ¬at(r,*l*)**

•robot now in two places

**•load(*c*,*r*,*l*)**

**•precond: at(*r*,*l*), in(*c*,*l*), unloaded(*r*)**

**•effects: loaded(*r*,*c*), ¬in(*c*,*l*), ¬unloaded(*r*)**

•container now in two places

**•unload(*c*,*r*,*l*)**

**•precond: at(*r*,*l*), loaded(*r*,*c*)**

**•effects: unloaded(*r*), in(*c*,*l*), ¬loaded(*r*,*c*)**

•container again in two places

# Computing $h^{FF}$: Relaxed Planning Graph

**function** computeRPG($A$,$s_i$,$g$)
  $F_0 \leftarrow s_i$; $t \leftarrow 0$
  **while** $g \nsubseteq F_t$ **do**
    $t \leftarrow t+1$
    $A_t \leftarrow \{a \in A \mid \text{precond}(a) \subseteq F_t\}$
    $F_t \leftarrow F_{t-1}$
    **for all** $a \in A_t$ **do**
      $F_t \leftarrow F_t \cup \text{effects}^+(a)$
    **if** $F_t = F_{t-1}$ **then return** failure
  **return** $[F_0, A_1, F_1, \ldots, A_t, F_t]$

**Computing $h^{FF}$: Relaxed Planning Graph**

- **function computeRPG($A$,$s_i$,$g$)**
  - arguments: propositional planning problem (again)
- **$F_0 \leftarrow s_i$; $t \leftarrow 0$**
- **while $g \nsubseteq F_t$ do**
- **$t \leftarrow t+1$**
- **$A_t \leftarrow \{a \in A \mid \text{precond}(a) \subseteq F_t\}$**
- **$F_t \leftarrow F_{t-1}$**
- **for all $a \in A_t$ do**
- **$F_t \leftarrow F_t \cup \text{effects}^+(a)$**
- **if $F_t = F_{t-1}$ then return failure**
- **return $[F_0, A_1, F_1, \ldots, A_t, F_t]$**
- similar to planning graph expansion
  - no mutex relations needed
  - stops when goal first appears

# Computing $h^{FF}$: Extracting a Relaxed Plan

**function** extractRPSize($[F_0, A_1, F_1, \ldots, A_k, F_k]$, $g$)
  **if** $g \nsubseteq F_k$ **then return** failure
  $M \leftarrow \max\{\text{firstlevel}(g_i, [F_0, \ldots, F_k]) \mid g_i \in g\}$
  **for** $t \leftarrow 0$ **to** $M$ **do**
    $G_t \leftarrow \{g_i \in g \mid \text{firstlevel}(g_i, [F_0, \ldots, F_k]) = t)$
  **for** $t \leftarrow M$ **to** 1 **do**
    **for all** $g_t \in G_t$ **do**
      **select** $a$ : firstlevel($a$, $[A_1, \ldots, A_t]$) = $t$ **and** $g_t \in$ effects$^+(a)$
      **for all** $p \in$ precond($a$) **do**
        $G_{\text{firstlevel}(p,\, [F0, \ldots, Fk])} \leftarrow G_{\text{firstlevel}(p,\, [F0, \ldots, Fk])} \cup \{p\}$
  **return** number of selected actions

**Computing $h^{FF}$: Extracting a Relaxed Plan**

- **function extractRPSize($[F_0, A_1, F_1, \ldots, A_k, F_k]$, $g$)**
    - arguments: planning graph and goal
- **if $g \nsubseteq F_k$ then return failure**
- **$M \leftarrow \max\{\text{firstlevel}(g_i, [F_0, \ldots, F_k]) \mid g_i \in g\}$**
    - function firstlevel: computes level in PG where proposition first appears
- **for $t \leftarrow 0$ to $M$ do**
- **$G_t \leftarrow \{g_i \in g \mid \text{firstlevel}(g_i, [F_0, \ldots, F_k]) = t)$**
    - start with goals in level where they first appear
- **for $t \leftarrow M$ to 1 do**
- **for all $g_t \in G_t$ do**
- **select $a$ : firstlevel($a$, $[A_1, \ldots, A_t]$) = $t$ and $g_t \in$ effects$^+(a)$**
    - commit to selected action (no backtracking)
- **for all $p \in$ precond($a$) do**
- **$G_{\text{firstlevel}(p,\, [F0, \ldots, Fk])} \leftarrow G_{\text{firstlevel}(p,\, [F0, \ldots, Fk])} \cup \{p\}$**
    - sub-goals in levels where they first appear
- **return number of selected actions**
- runs in polynomial time

# FF: Result

- heuristic is not admissible, but quite accurate

- "Almost all current successful satisficing planners use variations of (some of) these [ideas introduced in FF]!"

**FF: Result**

**•heuristic is not admissible, but quite accurate**

•returned plan not guaranteed to be optimal

**•"Almost all current successful satisficing planners use variations of (some of) these [ideas introduced in FF]!"**

•satisficing: type of planner we have looked at

•successful: in research; most practical planners still based on HTN paradigm

## Overview

- Simple Planning Graph Heuristics
- Pattern Database Heuristics
- The FF Planner

**Overview**

**•Simple Planning Graph Heuristics**

**•Pattern Database Heuristics**

**•The FF Planner**