1 Search Algorithms

Basic State Model: Classical Planning

Ambition:

Write one program that can solve all classical search problems.

State Model $\mathcal{S}(P)$:

- finite and discrete state space S
- a known initial state $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a deterministic transition function $s' = f(a, s)$ for $a \in A(s)$
- positive action costs $c(a, s)$

$\rightarrow$ A **solution** is a sequence of applicable actions that maps s_0 into S_G, and it is **optimal** if it minimizes **sum of action costs** (e.g., # of steps)

Search Terminology

Search node n: Contains a *state* reached by the search, plus information about how it was reached.

Path cost $g(n)$: The cost of the path reaching n.

Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.

Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the *state* s itself is also said to be expanded.

Search strategy: Method for deciding which node is expanded next.

Open list: Set of all *nodes* that currently are candidates for expansion. Also called frontier.

Closed list: Set of all *states* that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in generated states.)

Space Complexity: How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

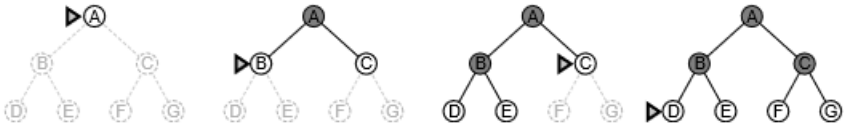
Branching factor b: How many successors does each state have?

Goal depth d: The number of actions required to reach the shallowest goal state.

Breadth-First Search

Strategy: Expand nodes in the order they were produced (FIFO frontier).

Illustration:



Guarantees:

- Completeness? Yes.
- Optimality? Yes, for uniform action costs. Breadth-first search always finds a shallowest goal state. If costs are not uniform, this is not necessarily optimal.

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes? $b + b^2 + b^3 + \cdots + b^d$: In the worst case, the algorithm generates all nodes in the first d layers.
- So the time complexity is $O(b^d)$.
- And if we were to apply the goal test at node-expansion time, rather than node-generation time? $O(b^{d+1})$ because then we'd generate the first $d + 1$ layers in the worst case.

Space Complexity: Same as time complexity since all generated nodes are kept in memory.

Depth-First Search

Strategy: Expand the most recent nodes in (LIFO frontier).

Illustration: (Nodes at depth 3 are assumed to have no successors)

Guarantees:
Quizz@SpeakUp: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality? No. After all, the algorithm just "chooses some direction and hopes for the best". (Depth-first search is a way of "hoping to get lucky".)
- Completeness? No, because search branches may be infinitely long: No check for cycles along a branch!
  → Depth-first search is complete in case the state space is acyclic, e.g., Constraint Satisfaction Problems. If we do add a cycle check, it becomes complete for finite state spaces.

Complexity:

- Space: Stores nodes and applicable actions on the path to the current node. So if m is the maximal depth reached, the complexity is $O(b\,m)$.
- Time: If there are paths of length m in the state space, $O(b^m)$ nodes can be generated. Even if there are solutions of depth 1!
  → If we happen to choose "the right direction" then we can find a length-l solution in time $O(b\,l)$ regardless how big the state space is.

Iterative Deepening Search

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: Optimality? Yes! Completeness? Yes! Space complexity? $O(b\,d)$.

Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in O(b^d)$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in O(b^d)$

Heuristic Functions

Heuristic searches require a heuristic function to estimate remaining cost:

Definition (Heuristic Function). *Let Π be a planning task with state space Θ_Π. A heuristic function, short heuristic, for Π is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. Its value $h(s)$ for a state s is referred to as the state's heuristic value, or h-value.*

Definition (Remaining Cost, h^*). *Let Π be a planning task with state space Θ_Π. For a state $s \in S$, the state's remaining cost is the cost of an optimal plan for s, or ∞ if there exists no plan for s. The perfect heuristic for Π, written h^*, assigns every $s \in S$ its remaining cost as the heuristic value.*

Properties of Heuristic Functions

Definition (Safe/Goal-Aware/Admissible/Consistent). *Let Π be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$, and let h be a heuristic for Π. The heuristic is called:*

- *safe if $h^*(s) = \infty$ for all $s \in S$ with $h(s) = \infty$;*
- *goal-aware if $h(s) = 0$ for all goal states $s \in S^G$;*
- *admissible if $h(s) \leq h^*(s)$ for all $s \in s$;*
- *consistent if $h(s) \leq h(s') + c(a)$ for all transitions $s \xrightarrow{a} s'$.*

Greedy Best-First Search

Properties:

- Complete? Yes, for safe heuristics. (and duplicate detection to avoid cycles)
- Optimal? No.[1]
- Invariant under all strictly monotonic transformations of h (e.g., scaling with a positive constant or adding a constant).

A*

Properties:

- Complete? Yes, for safe heuristics. (Even without duplicate detection.)
- Optimal? Yes, for admissible heuristics. (Even without duplicate detection.)

Weighted A*

The weight $W \in \mathbb{R}_0^+$ is an **algorithm parameter:**

- For $W = 0$, weighted A* behaves like uniform-cost search.
- For $W = 1$, weighted A* behaves like A*.
- For $W \to \infty$, weighted A* behaves like greedy best-first search.

Properties:

- For $W > 1$, weighted A* is bounded suboptimal: if h is admissible, then the solutions returned are at most a factor W more costly than the optimal ones.

Hill-Climbing

Hill-Climbing

```
σ := make-root-node(init())
forever:
  if is-goal(state(σ)):
    return extract-solution(σ)
  Σ' := { make-node(σ, a, s') | (a, s') ∈ succ(state(σ)) }
  σ := an element of Σ' minimizing h /* (random tie breaking) */
```

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this complete or optimal? No.
- Can easily get stuck in local minima where immediate improvements of $h(\sigma)$ are not possible.
- Many variations: tie-breaking strategies, restarts, ...

Enforced Hill-Climbing

Enforced Hill-Climbing

$\sigma :=$ make-root-node(init())
while not is-goal(state(σ)):
    $\sigma :=$ improve(σ)
return extract-solution(σ)

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this optimal? No.
- Is this complete? In general, no. Under particular circumstances, yes. Assume that h is goal-aware.
  $\rightarrow$ Procedure *improve* fails: no state with strictly smaller h-value reachable from s, thus (with assumption) goal not reachable from s.
  $\rightarrow$ This can, for example, not happen if the state space is undirected, i.e., if for all transitions $s \rightarrow s'$ in Θ_Π there is a transition $s' \rightarrow s$.

Properties of Search Algorithms

	DFS	BrFS	ID	A*	HC	IDA*
Complete	No	Yes	Yes	Yes	No	Yes
Optimal	No	Yes*	Yes	Yes	No	Yes
Time	∞	b^d	b^d	b^d	∞	b^d
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$

- Parameters: d is solution depth; b is branching factor
- Breadth First Search (BrFS) optimal when costs are uniform
- A*/IDA* optimal when h is **admissible**; $h \leq h^*$

2 Introduction to Planning

Programming-Based Approach

- Advantage: domain-knowledge easy to express
- Disadvantage: cannot deal with situations not anticipated by programmer

Learning-Based Approach

- Advantage: does not require much knowledge in principle
- Disadvantage: in practice, hard to know which features to learn, and is slow

Model-Based Approach

→ Advantage:

- Powerful: In some applications generality is absolutely necessary
- Quick: Rapid prototyping. 10s lines of problem description vs. 1000s lines of C++ code. (Language generation!)
- Flexible & Clear: Adapt/maintain the description.

Models, Languages, and Solvers

- A planner is a solver over a class of models; it takes a model description, and computes the corresponding controller

$$\textit{Model} \Longrightarrow \boxed{\textit{Planner}} \Longrightarrow \textit{Controller}$$

Strips

- A **problem** in STRIPS is a tuple $P = \langle F, O, I, G \rangle$:
  - F stands for set of all atoms (boolean vars)
  - O stands for set of all operators (actions)
  - $I \subseteq F$ stands for initial situation
  - $G \subseteq F$ stands for goal situation
- Operators $o \in O$ **represented** by
  - the Add list $Add(o) \subseteq F$
  - the Delete list $Del(o) \subseteq F$
  - the Precondition list $Pre(o) \subseteq F$

Decision Problems in Planning

Definition (PlanEx). By PlanEx, we denote the problem of deciding, given a planning task P, whether or not there exists a plan for P.

→ Corresponds to satisficing planning.

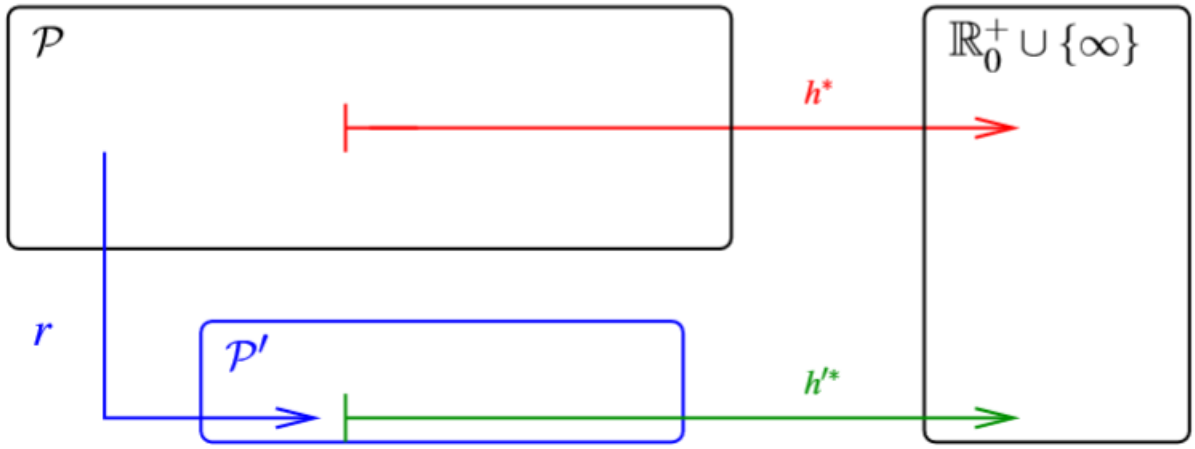
Definition (PlanLen). By PlanLen, we denote the problem of deciding, given a planning task P and an integer B, whether or not there exists a plan for P of length at most B.

→ Corresponds to optimal planning.

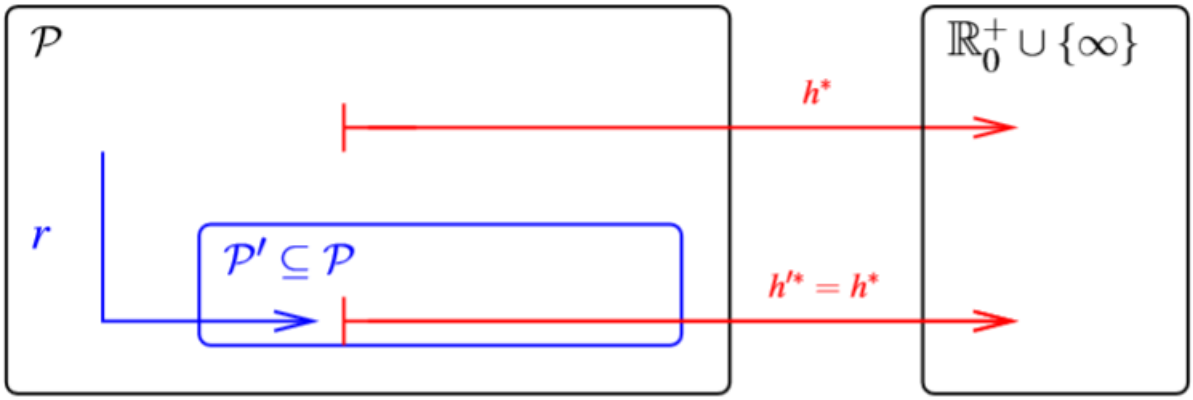
Relaxations

Definition (Relaxation). *Let $h^* : \mathcal{P} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ be a function. A relaxation of h^* is a triple $\mathcal{R} = (\mathcal{P}', r, h'^*)$ where $\mathcal{P}'$ is an arbitrary set, and $r : \mathcal{P} \mapsto \mathcal{P}'$ and $h'^* : P' \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ are functions so that, for all $\Pi \in \mathcal{P}$, the relaxation heuristic $h^{\mathcal{R}}(\Pi) := h'^*(r(\Pi))$ satisfies $h^{\mathcal{R}}(\Pi) \leq h^*(\Pi)$. The relaxation is:*

- *native if $\mathcal{P}' \subseteq P$ and $h'^* = h^*$;*
- *efficiently constructible if there exists a polynomial-time algorithm that, given $\Pi \in \mathcal{P}$, computes $r(\Pi)$;*
- *efficiently computable if there exists a polynomial-time algorithm that, given $\Pi' \in \mathcal{P}'$, computes $h'^*(\Pi')$.*

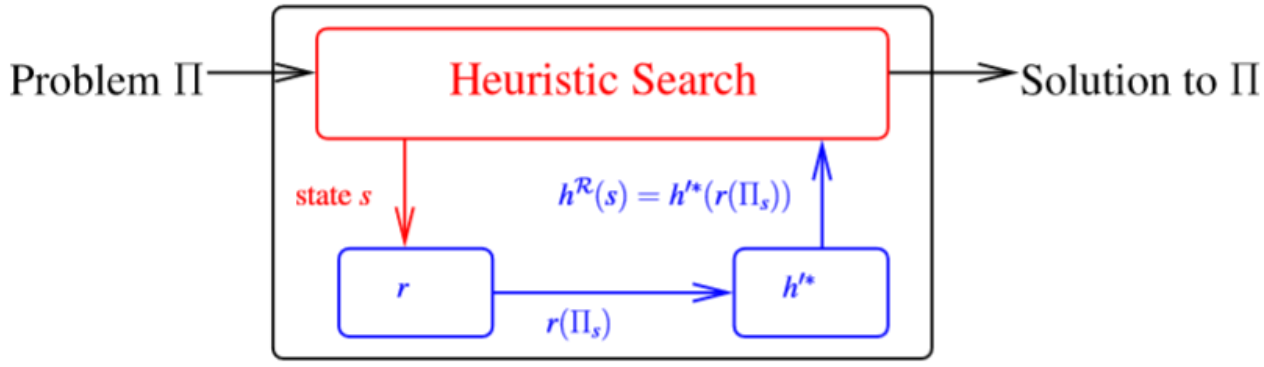


Native Relaxations



How to Relax During Search: Diagram

Using a relaxation $\mathcal{R} = (\mathcal{P}', r, h'^*)$ during search:



→ Π_s: Π with initial state replaced by s, i.e., $\Pi = (F, A, c, I, G)$ changed to (F, A, c, s, G).

→ The task of finding a plan for search state s.

The Delete Relaxation

Definition (Delete Relaxation).

(i) *For a STRIPS action a, by a^+ we denote the corresponding delete relaxed action, or short relaxed action, defined by $pre_{a^+} := pre_a$, $add_{a^+} := add_a$, and $del_{a^+} := \emptyset$.*

(ii) *For a set A of STRIPS actions, by A^+ we denote the corresponding set of relaxed actions, $A^+ := \{a^+ \mid a \in A\}$; similarly, for a sequence $\vec{a} = \langle a_1, \ldots, a_n \rangle$ of STRIPS actions, by $\vec{a}^+$ we denote the corresponding sequence of relaxed actions, $\vec{a}^+ := \langle a_1^+, \ldots, a_n^+ \rangle$.*

(iii) *For a STRIPS planning task $\Pi = (F, A, c, I, G)$, by $\Pi^+ := (F, A^+, c, I, G)$ we denote the corresponding (delete) relaxed planning task.*

→ "+" super-script = delete relaxed. We'll also use this to denote states encountered within the relaxation. (For STRIPS, s^+ is a fact set just like s.)

Definition (Relaxed Plan). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state. An (optimal) relaxed plan for s is an (optimal) plan for Π_s^+. A relaxed plan for I is also called a relaxed plan for Π.*

State Dominance

Definition (Dominance). *Let $\Pi^+ = (F, A^+, c, I, G)$ be a STRIPS planning task, and let s^+, s'^+ be states. We say that s'^+ dominates s^+ if $s'^+ \supseteq s^+$.*

→ For example, on the previous slide, who dominates who? Each state along the relaxed plan dominates the previous one, simply because the actions don't delete any facts.

Proposition (Dominance). *Let $\Pi^+ = (F, A^+, c, I, G)$ be a STRIPS planning task, and let s^+, s'^+ be states where s'^+ dominates s^+. We have:*

(i) *If s^+ is a goal state, then s'^+ is a goal state as well.*

(ii) *If $\vec{a}^+$ is applicable in s^+, then $\vec{a}^+$ is applicable in s'^+ as well, and $appl(s'^+, \vec{a}^+)$ dominates $appl(s^+, \vec{a}^+)$.*

Proposition. *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, let s be a state, and let $a \in A$. Then $appl(s, a^+)$ dominates both (i) s and (ii) $appl(s, a)$.*

$\Longrightarrow$ Optimal relaxed plans admissibly estimate the cost of optimal plans:

Proposition (Delete Relaxation is Admissible). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, let s be a state, and let $\vec{a}$ be a plan for Π_s. Then $\vec{a}^+$ is a relaxed plan for s.*

Greedy Relaxed Planning

Greedy Relaxed Planning for Π_s^+

$s^+ := s$; $\vec{a}^+ := \langle\rangle$
while $G \not\subseteq s^+$ **do**:
  if $\exists a \in A$ s.t. $pre_a \subseteq s^+$ and $appl(s^+, a^+) \neq s^+$ **then**
    select one such a
    $s^+ := appl(s^+, a^+)$; $\vec{a}^+ := \vec{a}^+ \circ \langle a^+ \rangle$
  else return "Π_s^+ is unsolvable" **endif**
endwhile
return $\vec{a}^+$

Proposition. *Greedy relaxed planning is sound, complete, and terminates in time polynomial in the size of Π.*

Using greedy relaxed planning to generate h

- In search state s during forward search, run greedy relaxed planning on Π_s^+.
- Set $h(s)$ to the cost of $\vec{a}^+$, or ∞ if "Π_s^+ is unsolvable" is returned.

$\rightarrow$ Is this heuristic safe? Yes: $h(s) = \infty$ only if no relaxed plan for s exists, which by admissibility of delete relaxation implies that no plan for s exists.

$\rightarrow$ Is this heuristic goal-aware? Yes, we'll have $G \subseteq s^+$ right at the start.

$\rightarrow$ Is this heuristic admissible? Would be if the relaxed plans were optimal; but they clearly aren't. So h isn't consistent either.

h+: The Optimal Delete Relaxation Heuristic

Definition (h^+). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task with state space $\Theta_\Pi = (S, A, c, T, I, G)$. The optimal delete relaxation heuristic h^+ for Π is the function $h^+ : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ where $h^+(s)$ is defined as the cost of an optimal relaxed plan for s.*

Corollary (h^+ is Admissible). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task. Then h^+ is admissible, and thus safe and goal-aware.* (By admissibility of delete relaxation.)

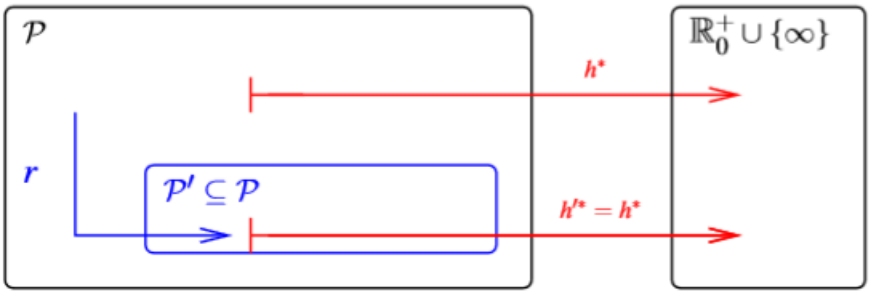
→ To be informed (accurately estimate h^*), a heuristic needs to approximate the *minimum effort* needed to reach the goal. h^+ naturally does so by asking for the cheapest possible relaxed plans.

Definition (Optimal Relaxed Planning). *By PlanOpt$^+$, we denote the problem of deciding, given a STRIPS planning task $\Pi = (F, A, c, I, G)$ and $B \in \mathbb{R}_0^+$, whether there exists a relaxed plan for Π whose cost is at most B.*

→ By computing h^+, we would solve PlanOpt$^+$.

Theorem (Optimal Relaxed Planning is Hard). *PlanOpt$^+$ is* **NP***-complete.*

h+ as a Relaxation Heuristic



where, for all $\Pi \in \mathcal{P}$, $h^*(r(\Pi)) \leq h^*(\Pi)$.

For $h^+ = h^* \circ r$:

- Problem $\mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $\mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic h'^* for $\mathcal{P}'$: Optimal plan cost $= h^*$ on $\mathcal{P}'$.
- Transformation r: Drop the deletes.

→ Is this a native relaxation? Yes.
→ Is this relaxation efficiently constructible? Yes.
→ Is this relaxation efficiently computable? No.

The Additive and Max Heuristics

Definition (h^{add}). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task. The additive heuristic h^{add} for Π is the function $h^{\text{add}}(s) := h^{\text{add}}(s, G)$ where $h^{\text{add}}(s, g)$ is the point-wise greatest function that satisfies $h^{\text{add}}(s, g) =$*

$$\begin{cases} 0 & g \subseteq s \\ min_{a \in A, g \in add_a} c(a) + h^{\text{add}}(s, pre_a) & |g| = 1 \\ \sum_{g' \in g} h^{\text{add}}(s, \{g'\}) & |g| > 1 \end{cases}$$

Definition ($h^{\max}$). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task. The max heuristic $h^{\max}$ for Π is the function $h^{\max}(s) := h^{\max}(s, G)$ where $h^{\max}(s, g)$ is the point-wise greatest function that satisfies $h^{\max}(s, g) =$*

$$\begin{cases} 0 & g \subseteq s \\ min_{a \in A, g \in add_a} c(a) + h^{\max}(s, pre_a) & |g| = 1 \\ \max_{g' \in g} h^{\max}(s, \{g'\}) & |g| > 1 \end{cases}$$

Proposition ($h^{\max}$ is Optimistic). *$h^{\max} \leq h^+$, and thus $h^{\max} \leq h^*$.*

Proposition (h^{add} is Pessimistic). *For all STRIPS planning tasks Π, $h^{\text{add}} \geq h^+$. There exist Π and s so that $h^{\text{add}}(s) > h^*(s)$.*

Proposition ($h^{\max}$ and h^{add} Agree with h^+ on ∞). *For all STRIPS planning tasks Π and states s in Π, $h^+(s) = \infty$ if and only if $h^{\max}(s) = \infty$ if and only if $h^{\text{add}}(s) = \infty$.*

Bellman-Ford for $h^{\max}$ and h^{add}

Bellman-Ford variant computing h^{add} for state s

new table $T_0^{\text{add}}(g)$, for $g \in F$

For all $g \in F$: $T_0^{\text{add}}(g) := \begin{cases} 0 & g \in s \\ \infty & \text{otherwise} \end{cases}$

fn $c_i(g) := \begin{cases} T_i^{\text{add}}(g) & |g| = 1 \\ \sum_{g' \in g} T_i^{\text{add}}(g') & |g| > 1 \end{cases}$

fn $f_i(g) := \min[c_i(g), \min_{a \in A, g \in add_a} c(a) + c_i(pre_a)]$

do forever:

  new table $T_{i+1}^{\text{add}}(g)$, for $g \in F$

  For all $g \in F$: $T_{i+1}^{\text{add}}(g) := f_i(g)$

  if $T_{i+1}^{\text{add}} = T_i^{\text{add}}$ **then** stop **endif**

  $i := i + 1$

enddo

→ Basically the same algorithm works for $h^{\max}$, just change $\sum$ for $\max$

Proposition. *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task. Then the series $\{T_i^{\text{add}}(g)\}_{i=0,\ldots}$ converges to $h^{\text{add}}(s, g)$, for all g.* (Proof omitted.)

Summary of typical issues in practice with h^{add} and $h^{\max}$:

- Both h^{add} and $h^{\max}$ can be computed reasonably quickly.
- $h^{\max}$ is admissible, but is typically far too optimistic.
- h^{add} is not admissible, but is typically a lot more informed than $h^{\max}$.
- h^{add} is sometimes better informed than h^+, but for the "wrong reasons": rather than accounting for deletes, it overcounts by ignoring positive interactions, i.e., sub-plans shared between sub-goals.
- Such overcounting can result in dramatic over-estimates of h^*!!

Relaxed Plan Extraction

→ First compute a best-supporter function bs, which for every fact $p \in F$ returns an action that is deemed to be the cheapest achiever of p (within the relaxation). Then extract a relaxed plan from that function, by applying it to singleton sub-goals and collecting all the actions.

→ The best-supporter function can be based directly on $h^{\max}$ or h^{add}, simply selecting an action a achieving p that minimizes the sum of $c(a)$ and the cost estimate for pre_a.

Relaxed Plan Extraction for state s and best-supporter function bs

$Open := G \setminus s$; $Closed := \emptyset$; $RPlan := \emptyset$
while $Open \neq \emptyset$ **do**:
  select $g \in Open$
  $Open := Open \setminus \{g\}$; $Closed := Closed \cup \{g\}$;
  $RPlan := RPlan \cup \{bs(g)\}$; $Open := Open \cup (pre_{bs(g)} \setminus (s \cup Closed))$
endwhile
return $RPlan$

→ Starting with the top-level goals, iteratively close open singleton sub-goals by selecting the best supporter.

This is fast! Number of iterations bounded by $|P|$, each near-constant time.

But is it correct?
→ What if $g \notin add_{bs(g)}$? Doesn't make sense. → Prerequisite (A).
→ What if $bs(g)$ is undefined? Runtime error. → Prerequisite (B).
→ What if the support for g eventually requires g itself as a precondition? Then this does not actually yield a relaxed plan. → Prerequisite (C).

→ For relaxed plan extraction to make sense, it requires a *closed well-founded* best-supporter function:

Definition (Best-Supporter Function). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state. A best-supporter function for s is a partial function $bs : (F \setminus s) \mapsto A$ such that $p \in add_a$ whenever $a = bs(p)$.*
The support graph of bs is the directed graph with vertices $F \cup A$ and arcs $\{(p, a) \mid p \in pre_a\} \cup \{(a, p) \mid a = bs(p)\}$. We say that bs is closed if $bs(p)$ is defined for every $p \in (F \setminus s)$ that has a path to a goal $g \in G$ in the support graph. We say that bs is well-founded if the support graph is acyclic.

- "$p \in add_a$ whenever $a = bs(p)$": Prerequisite (A).
- bs is closed: Prerequisite (B).
- bs is well-founded: Prerequisite (C).

Definition (Best-Supporters from $h^{\max}$ and h^{add}). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state.*
The $h^{\max}$ supporter function $bs_s^{\max} : \{p \in F \mid 0 < h^{\max}(s, \{p\}) < \infty\} \mapsto A$ is defined by $bs_s^{\max}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\max}(s, pre_a)$.
The h^{add} supporter function $bs_s^{\text{add}} : \{p \in F \mid 0 < h^{\text{add}}(s, \{p\}) < \infty\} \mapsto A$ is defined by $bs_s^{\text{add}}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\text{add}}(s, pre_a)$.

Proposition. *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task such that, for all $a \in A$, $c(a) > 0$. Let s be a state where $h^+(s) < \infty$. Then both $bs_s^{\max}$ and bs_s^{add} are closed well-founded supporter functions for s.*

Proposition. *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, let s be a state, and let bs be a closed well-founded best-supporter function for s. Then the action set $RPlan$ returned by relaxed plan extraction can be sequenced into a relaxed plan $\vec{a}^+$ for s.*

Proposition (h^{FF} is Pessimistic and Agrees with h^* on ∞). *For all STRIPS planning tasks Π, $h^{\mathsf{FF}} \geq h^+$; for all states s, $h^+(s) = \infty$ if and only if $h^{\mathsf{FF}}(s) = \infty$. There exist Π and s so that $h^{\mathsf{FF}}(s) > h^*(s)$.*

4 Markov Decision Processes (MDPs)

MDPs are **fully observable, probabilistic** state models. The most common formulation of MDPs is a *Discounted-Reward* Markov Decision Process:

- a state space S
- initial state $s_0 \in S$
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- **transition probabilities** $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- **rewards** $r(s, a, s')$ positive or negative of transitioning from state s to state s' using action a
- a **discount factor** $0 \leq \gamma \leq 1$

What is different from classical planning? Four things:

- The transition function is no longer deterministic. Each action has a probability of $P_a(s'|s)$ of ending in state s' if a is executed in the state s.
- There are no goals. Each action receives a reward when applied. The value of the reward is dependent on the state in which it is applied.
- There are no action costs. These are modelled as negative rewards.
- We have a *discount factor*.

Expected Value of Policy for Discounted Reward MDPs

For discounted-reward MDPs, optimal solutions maximise the *expected discounted accumulated reward* from the initial state s_0. But what is the expected discounted accumulated reward?

- In Discounted Reward MDPs, the **expected discounted reward from** s is

$$V^{\pi}(s) = E_{\pi}[\sum_{i} \gamma^i \, r(a_i, s_i) \mid s_0 = s, a_i = \pi(s_i)]$$

Thus, $V^{\pi}(s)$ defines the expected value of following the policy π from state s.

Bellman equations (Discounted-Reward MDPs)

The *Bellman equations*, identified by Richard Bellman, describe the condition that must hold for a policy to be optimal. It generalises to problems other than MDPs, but we consider only MDPs here.

For discounted-reward MDPs the Bellman equation is defined recursively as:

$$V(s) = \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \; [r(s,a,s') + \gamma \; V(s')]$$

Thus, V is optimal **if** for all states s, $V(s)$ describes the total discounted reward for taking the action with the highest reward over an indefinite/infinite horizon.

Value Iteration

Value Iteration finds the optimal value function V^* solving the Bellman equations iteratively, using the following algorithm:

- Set V_0 to arbitrary value function; e.g., $V_0(s) = 0$ for all s.
- Set V_{i+1} to result of Bellman's **right hand side** using V_i in place of V:

$$V_{i+1}(s) := \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \; [r(s,a,s') + \gamma \; V_i(s')]$$

This converges exponentially fast to the optimal policy as iterations continue.

Theorem

$V_i \mapsto V^*$ as $i \mapsto \infty$. That is, given an infinite amount of iterations, it will be optimal.

Policy evaluation

The *expected cost* of policy π from s to goal, $V^\pi(s)$, is weighted avg of cost of the possible state sequences defined by that policy times their probability given π.

However, the expected costs $V^\pi(s)$ can also be characterised as a solution to the equation

$$V^\pi(s) = \sum_{s' \in S} P_a(s'|s) \; [r(s,a,s') + \gamma \; V^\pi(s')]$$

where $a = \pi(s)$, and $V^\pi(s) = 0$ for goal states

This set of linear equations can be solved analytically using MATLAB, by a VI-like procedure, or whatever – we do not care for this subject.

The *optimal expected cost* $V^*(s)$ is $\max_\pi V^\pi(s)$ and the *optimal policy* is the $\arg\max$

Policy Iteration

Let $Q^\pi(a, s)$ be the expected cost from s when doing a first and then following the policy π:

$$Q^\pi(a, s) = \sum_{s' \in S} P_a(s'|s) \left[r(s, a, s') + \gamma V^\pi(s') \right]$$

When $Q^\pi(a, s) < Q^\pi(\pi(s), s)$, π *strictly improved* by changing $\pi(s)$ to a

Policy Iteration computes π^* by a sequence of policy evaluations and improvements:

1. Starting with arbitrary policy π
2. Compute $V^\pi(s)$ for all s (policy evaluation)
3. Improve π by setting $\pi(s) := \text{argmax}_{a \in A(s)} Q^\pi(a, s)$ (improvement)
4. If π changed in 3, go back to 2, else *finish*

This algorithm finishes with an optimal π^* after a finite number of iterations, because the number of policies is finite, bounded by $O(|A|^{|S|})$, unlike value iteration, which can theoretically require infinite iterations.

However, each iteration costs $O(|S|^2|A| + |S|^3)$. Empirical evidence suggests that the most efficient is dependent on the particular MDP model being solved.

Monte Carlo Tree Search

The algorithm is online, which means the search is done while the agent is executing, rather than beforehand. Thus, MCTS is invoked every time it visits a new state s. Fundamental features:

1. The *Q-value* for each action a at state s $Q(s, a)$ is approximated using *random simulation*.
   As in the MDP notes, $Q(s, a)$ is the estimated value of taking action a in state s.
2. These estimates of $Q(s, a)$ are used to drive a *best-first strategy*; thus, $Q(s, a)$ is being updated while it is also be used as an heuristic in the search.
3. A *search tree* is built incrementally
4. The search terminates when some pre-defined computational budget is used up, such as a time limit or a number of expanded nodes.
   Therefore, it is an *anytime* algorithm, as it can be terminated at any time and still give an answer.
5. The best performing action a at s is returned; that is $argmax_{a \in A} Q(s, a)$.

→ This is complete if there are *no* dead–ends.

→ This is optimal if if the computational budget, and is perfect if an entire search can be performed (which is unusual – if the problem is that small we should just use value/policy iteration).

The Framework: Monte Carlo Tree Search (MCTS)
The evaluated states are stored in a search tree. The set of evaluated states is incrementally built be iterating over the following four steps:

- *Select*: Given a *tree policy*, select a single node in the tree to assess.
- *Expand*: Expand this node by applying one available action from the node.
- *Simulation*: From the expanded node, perform a complete random simulation to a leaf node. This therefore assumes that the search tree is finite (but version for infinitely large trees exist).
- *Backpropagate*: Finally, the value of the node is *backpropagated* to the root node, updating the value of each ancestor node on the way.

Multi-Armed Bandit

An N-armed bandit is defined by a set of *random variables* $X_{i,k}$ where

- $1 \leq i \leq N$, such that i is the *arm* of the bandit; and
- k the index of the *play* of arm i.

Successive plays $X_{i,1}, X_{j,2}, X_{k,3} \ldots$ are assumed to be independently distributed according to an *unknown* law. That is, we do not know the probability distributions of the random variables.

Intuition: actions a applicable on s are the "arms of the bandit", and $Q(a, s)$ corresponds to the random variables $X_{i,n}$.

Exploration vs. Exploitation

We seek policies π that *minimise regret*.

(Pseudo)–Regret

$$\mathcal{R}_{N(s),b} = Q(\pi^*(s), s)N(s) - \mathbb{E}[\sum_{t}^{N(s)} Q(b, s)\mathbb{I}_t(s, b)]$$

ϵ-**greedy**: ϵ is a number in [0,1]. Each time we need to choose an arm, we choose a random arm with probability ϵ, and choose the arm with max $Q(s, a)$ with probability $1 - \epsilon$. Typically, values of ϵ around 0.05-0.1 work well.

ϵ-**decreasing**: The same as ϵ-greedy, ϵ decreases over time. A parameter α between [0,1] specifies the *decay*, such that $\epsilon := \epsilon.\alpha$ after each action is chosen.

Softmax: This is *probability matching strategy*, which means that the probability of each action being chosen is dependent on its Q-value so far. Formally:

$$\frac{e^{Q(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q(s,b)/\tau}}$$

in which τ is the *temperature*, a positive number that dictates how much of an influence the past data has on the decision.

Upper Condence Bounds (UCB1)

A highly effective (especially in terms of MCTS) multi-armed bandit strategy is the *Upper Confidence Bounds* (UCB1) strategy.

UCB1 policy $\pi(s)$

$$\pi(s) := \underset{a \in A(s)}{\operatorname{argmax}}\ Q(a,s) + \sqrt{\frac{2 \ln N(s)}{N(a,s)}}$$

$Q(a,s)$ is the estimated Q-value.

$N(s)$ is the number of times s has been visited.

$N(s,a)$ is the number of times times a has been executed in s.

→The left–hand side encourages exploitation: the Q-value is high for actions that have had a high reward.
→The right–hand side encourages exploration: it is high for actions that have been explored less.

Upper Condence Trees (UCT)

$$\text{UCT} = \text{MCTS} + \text{UCB1}$$

Kocsis & Szepesvári were the first to treat the selection of nodes to expand in MCTS as a multi-armed bandit problem.

UCT exploration policy

$$\pi(s) := \underset{a \in A(s)}{\operatorname{argmax}}\, Q(a,s) + 2C_p \sqrt{\frac{2 \ln N(s)}{N(s,a)}}$$

$C_p > 0$ is the exploration constant, which determines can be increased to encourage more exploration, and decreased to encourage less exploration. Ties are broken randomly.

→ if $Q(a,s) \in [0,1]$ **and** $C_p = \frac{1}{\sqrt{2}}$ **then** in two-player adversarial games, UCT converges to the well-known Minimax algorithm (if you don't know what Minimax is, ignore this for now and we'll mention it later in the subject).

Reinforcement Learning

In reinforcement learning, there is an MDP model that dictates what happens when actions are performed and what the rewards are (that is, the world behaves like an MDP), **BUT**, the we do not know this information. We must *learn* it.

Assumptions:

- The agent does **not** know the environment (which is an MDP).
- The agent **experiences** the environment by **interacting** with it.
- The environment is **revealed** to agent in the form of a **state** s.
- The agent **influences** the environment by applying actions and then receiving **rewards**.

In short, all we know is what actions the agent can do, how to read/see see the state, and how to receive rewards.

Q-Learning

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Updating the Q-function (line 7) is where the learning happens:

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{old value}} + \overbrace{\alpha}^{\text{learning rate}} \cdot [\underbrace{r}_{\text{reward}} + \overbrace{\gamma}^{\text{discount factor}} \cdot \underbrace{\max_{a'} Q(s',a')}_{\text{estimate of optimal future value}} \overbrace{-Q(s,a)}^{\text{do not count extra } Q(s,a)}]$$

SARSA: On-Policy Reinforcement Learning

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

Q-learning vs. SARSA

On-Policy: Uses the action chosen by the policy for the update!

Off-Policy: Ignores the action chosen by the policy, uses the best action $\text{argmax}_{a'} \mathcal{Q}(s', a')$ for the update!

On-Policy SARSA learns action values relative to the policy it follows, while Off-Policy Q-Learning does it relative to the greedy policy.

- On-policy learning is more appropriate when we want to optimise the behaviour of an agent who learns *while operating in its environment*.

  Imagine if we had 1000 robots that needed to navigate to the other side and each time one fell off the cliff, we lost it; but also we could get each robots Q-function after each episode. Given that the average reward *per trial* is better, this would give us better overall outcomes than Q-learning.
- Off-policy learning is more appropriate when we have the luxury of training our agent offline before it is put into operation.

  That is, imagine if we could run 1000 training episodes on a fake cliff scenario *before* (that is, off-line) we had to move our robots to the other side. In this case, Q-learning would be better because its optimal policy could be followed.