AI Planning and Autonomy Summary

Classical Planning Model:

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)

Blind Search vs Heuristic Search:

Blind search vs. heuristic (or informed) search:

- Blind search algorithms: Only use the basic ingredients for general search algorithms.
  - *e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID)*
- Heuristic search algorithms: Additionally use heuristic functions which estimate the distance (or remaining cost) to the goal.
  - *e.g., A*, IDA*, Hill Climbing, Best First, WA*, DFS B&B, LRTA*, ...*

Systematic Search vs Local Search:

Systematic search vs. local search:

- Systematic search algorithms: Consider a large number of search nodes simultaneously.
- Local search algorithms: Work with one (or a few) candidate solutions (search nodes) at a time.

  $\rightarrow$ This is not a black-and-white distinction; there are *crossbreeds* (e.g., enforced hill-climbing).

Evaluation Criteria:

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.)

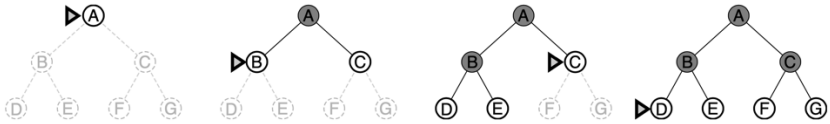
Typical state space features governing complexity:

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

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

BFS:

Illustration:

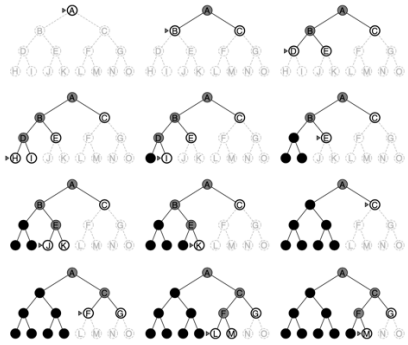


Guarantees:

- 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.

DFS:

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



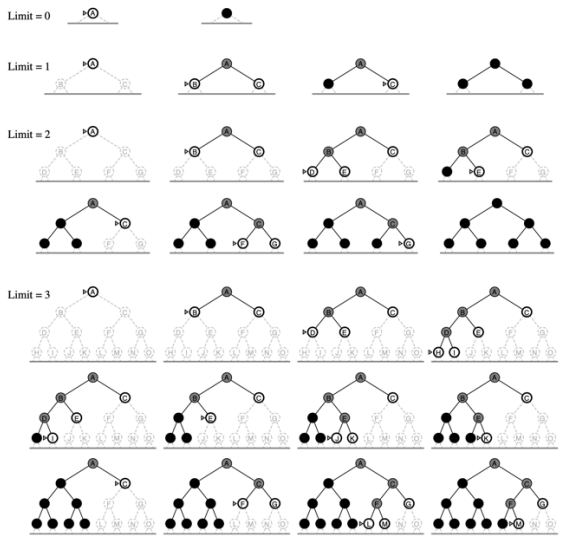
Guarantees:

pollev.com/nirlipo 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.

Iterative Deepening:



Optimality? Yes! Completeness? Yes!

Heuristic Search:

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 $s \xrightarrow{a} s'$.*

→ **Relationships?**

Proposition. *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. If h is consistent and goal-aware, then h is admissible. If h is admissible, then h is goal-aware. If h is admissible, then h is safe. No other implications of this form hold.*

Greedy Best-First Search:

Greedy Best-First Search (with duplicate detection)

```
open := new priority queue ordered by ascending h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    σ := open.pop-min() /* get best state */
    if state(σ) ∉ closed: /* check duplicates */
      closed := closed ∪ {state(σ)} /* close state */
      if is-goal(state(σ)): return extract-solution(σ)
      for each (a, s') ∈ succ(state(σ)): /* expand state */
          σ' := make-node(σ, a, s')
          if h(state(σ')) < ∞: open.insert(σ')
return unsolvable
```

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*:

A* (with duplicate detection and re-opening)

```
open := new priority queue ordered by ascending g(state(σ)) + h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
best-g := ∅ /* maps states to numbers */
while not open.empty():
    σ := open.pop-min()
    if state(σ) ∉ closed or g(σ) < best-g(state(σ)):
      /* re-open if better g; note that all σ' with same state but worse g
         are behind σ in open, and will be skipped when their turn comes */
      closed := closed ∪ {state(σ)}
      best-g(state(σ)) := g(σ)
      if is-goal(state(σ)): return extract-solution(σ)
      for each (a, s') ∈ succ(state(σ)):
          σ' := make-node(σ, a, s')
          if h(state(σ')) < ∞: open.insert(σ')
return unsolvable
```

Properties:

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

Weighted A*:

Weighted A* (with duplicate detection and re-opening)

```
open := new priority queue ordered by ascending g(state(σ)) + W * h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
best-g := ∅
while not open.empty():
    σ := open.pop-min()
    if state(σ) ∉ closed or g(σ) < best-g(state(σ)):
      closed := closed ∪ {state(σ)}
      best-g(state(σ)) := g(σ)
      if is-goal(state(σ)): return extract-solution(σ)
      for each (a, s') ∈ succ(state(σ)):
          σ' := make-node(σ, a, s')
          if h(state(σ')) < ∞: open.insert(σ')
return unsolvable
```

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: Procedure *improve*

```
def improve(σ_0):
    queue := new fifo queue
    queue.push-back(σ_0)
    closed := ∅
    while not queue.empty():
        σ = queue.pop-front()
        if state(σ) ∉ closed:
            closed := closed ∪ {state(σ)}
            if h(state(σ)) < h(state(σ_0)): return σ
            for each (a, s') ∈ succ(state(σ)):
                σ' := make-node(σ, a, s')
                queue.push-back(σ')
    fail
```

Enforced Hill-Climbing

```
σ := make-root-node(init())
while not is-goal(state(σ)):
    σ := 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 $s' \rightarrow s$.

Overview 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	$b \cdot d$

Three Approaches to the Problem:

- **Programming-based:** Specify control by hand
- **Learning-based:** Learn control from experience
- **Model-based:** Specify problem by hand, derive control automatically

Programming-Based Approach:

→ Control specified by programmer, e.g.:

- If Mario finds no danger, then run...
- If danger appears and Mario is big, jump and kill ...
- ...

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

Learning-Based Approach:

→ Learns a controller from experience or through simulation:

- **Unsupervised** (Reinforcement Learning):
  - penalize Mario each time that 'dies'
  - reward agent each time oponent 'dies' and level is finished, . . .
- **Supervised** (Classification)
  - learn to classify actions into good or bad from info provided by teacher
- **Evolutionary**:
  - from pool of possible controllers: try them out, select the ones that do best, and mutate and recombine for a number of iterations, keeping best

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

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 $Pre(o) \subseteq F$

A STRIPS problem $P = \langle F, O, I, G \rangle$ determines **state model** $\mathcal{S}(P)$ where

- the states $s \in S$ are collections of atoms from F. $S = 2^F$
- the initial state s_0 is I
- the goal states s are such that $G \subseteq s$
- the actions a in $A(s)$ are ops in O s.t. $Prec(a) \subseteq s$
- the next state is $s' = s - Del(a) + Add(a)$
- action costs $c(a, s)$ are all 1

PDDL:

A PDDL planning task comes in two pieces:

- The domain file and the problem file.
- The problem file gives the objects, the initial state, and the goal state.
- The domain file gives the predicates and the operators; each benchmark domain has *one* domain file.

Satisfying planning is P and optimal planning NP-complete and both of them are PSPACE-complete in STRIPS.

Relax A Problem Informally:

How To Relax:

- You have a problem, $\mathcal{P}$, whose perfect heuristic h^* you wish to estimate.
- You define a simpler problem, $\mathcal{P}'$, whose perfect heuristic h'^* can be used to estimate h^*.
- You define a transformation, r, that simplifies instances from $\mathcal{P}$ into instances $\mathcal{P}'$.
- Given $\Pi \in \mathcal{P}$, you estimate $h^*(\Pi)$ by $h'^*(r(\Pi))$.

Relax A Problem Formally:

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 $h'^*(\Pi')$.*

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 $s'^+ \supseteq s^+$.*

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 $appl(s^+, \vec{a}^+)$.*

Proof. (i) is trivial. (ii) by induction over the length n of $\vec{a}^+$. Base case $n = 0$ is trivial. Inductive case $n \to n + 1$ follows directly from induction hypothesis and the definition of $appl(.,.)$.

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) $appl(s, a)$.*

Proof. Trivial from the definitions of $appl(s, a)$ and a^+.

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.*

Proof. Prove by induction over the length of $\vec{a}$ that $appl(s, \vec{a}^+)$ dominates $appl(s, \vec{a})$. Base case is trivial, inductive case follows from (ii) above.

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.*

Proof. Soundness: If $\vec{a}^+$ is returned then, by construction, $G \subseteq appl(s, \vec{a}^+)$. Completeness: If "$\Pi_s^+$ is unsolvable" is returned, then no relaxed plan exists for s^+ at that point; since s^+ dominates s, by the dominance proposition this implies that no relaxed plan can exist for s. Termination: Every $a \in A$ can be selected at most once because afterwards $appl(s^+, a^+) = s^+$.

→ 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.

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

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

Optimal Relaxed Planning 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.)

Additive 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 $h^{add}(s) := h^{add}(s, G)$ where $h^{add}(s, g)$ is the point-wise greatest function that satisfies $h^{add}(s, g) =$*

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

Maximum Heuristics:

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 $h^{max}(s, g) =$*

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

Properties of Additive and Maximum Heuristics:

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, $h^{add} \geq h^+$. There exist Π and s so that $h^{add}(s) > h^*(s)$.*

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 $h^{add}(s) = \infty$.*

- 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^*!!

Bellman-Ford for h^{max} and h^{add}:

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

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

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

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

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

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

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

  $i := i + 1$

enddo

Best-Supporter Function:

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 $bs_s^{\text{add}}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\text{add}}(s, pre_a)$.

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).

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.*

Proof. Since $h^+(s) < \infty$ implies $h^{\max}(s) < \infty$, it is easy to see that $bs_s^{\max}$ is closed (details omitted). If $a = bs_s^{\max}(p)$, then a is the action yielding $0 < h^{\max}(s, \{p\}) < \infty$ in the $h^{\max}$ equation. Since $c(a) > 0$, we have $h^{\max}(s, pre_a) < h^{\max}(s, \{p\})$ and thus, for all $q \in pre_a$, $h^{\max}(s, \{q\}) < h^{\max}(s, \{p\})$. Transitively, if the support graph contains a path from fact vertex r to fact vertex t, then $h^{\max}(s, \{r\}) < h^{\max}(s, \{t\})$. Thus there can't be cycles in the support graph and $bs_s^{\max}$ is well-founded. Similar for bs_s^{add}.

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.*

Proof. Order a before a' whenever the support graph contains a path from a to a'. Since the support graph is acyclic, such a sequencing $\vec{a} := \langle a_1, \ldots, a_n \rangle$ exists. We have $p \in s$ for all $p \in pre_{a_1}$, because otherwise $RPlan$ would contain the action $bs(p)$, necessarily ordered before a_1. We have $p \in s \cup add_{a_1}$ for all $p \in pre_{a_2}$, because otherwise $RPlan$ would contain the action $bs(p)$, necessarily ordered before a_2. Iterating the argument shows that $\vec{a}^+$ is a relaxed plan for s.

Relaxed Plan Extraction:

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 $RPlan$

Relaxed Plan Heuristic:

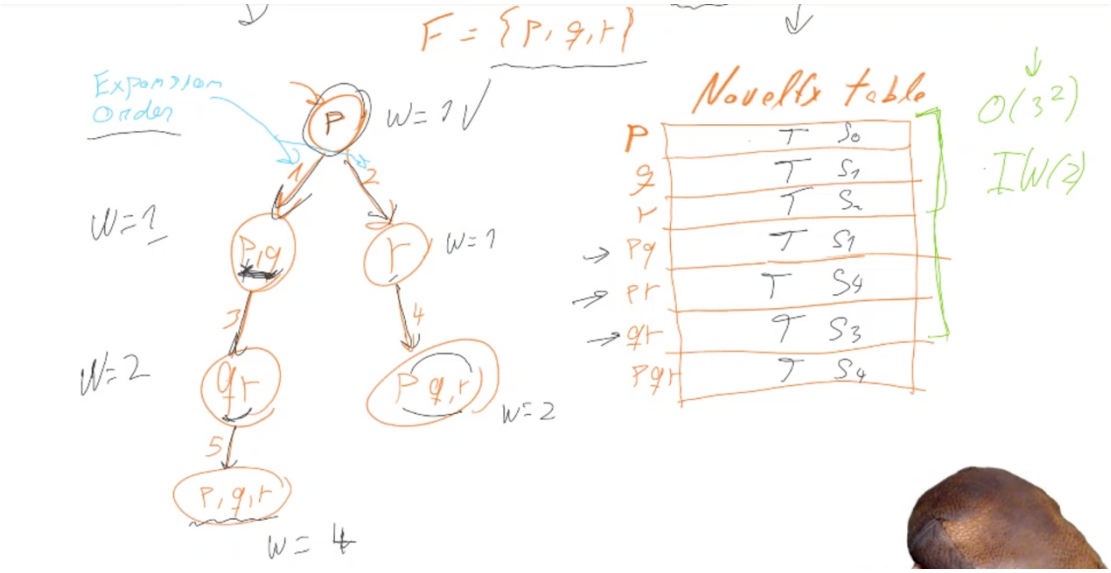
Definition (Relaxed Plan Heuristic). *A heuristic function is called a relaxed plan heuristic, denoted h^{FF}, if, given a state s, it returns ∞ if no relaxed plan exists, and otherwise returns $\sum_{a \in RPlan} c(a)$ where $RPlan$ is the action set returned by relaxed plan extraction on a closed well-founded best-supporter function for s.*

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

Novelty:

Key definition: the **novelty** $w(s)$ **of a state** s is the size of the smallest subset of atoms in s that is true for the first time in the search.

- e.g. $w(s) = 1$ if there is **one** atom $p \in s$ such that s is the first state that makes p true.
- Otherwise, $w(s) = 2$ if there are **two** different atoms $p, q \in s$ such that s is the first state that makes $p \wedge q$ true.
- Otherwise, $w(s) = 3$ if there are **three** different atoms...



Iterated Width:

Algorithm

- *IW*(k) = breadth-first search that prunes newly generated states whose $novelty(s) > k$.
- *IW* is a sequence of calls *IW*(k) for $i = 0, 1, 2, \ldots$ over problem P until problem solved or i exceeds number of variables in problem

Properties

$IW(k)$ expands at most $O(n^k)$ states, where n is the number of atoms.

Serialized Iterated Width:

- ***SIW* uses *IW*** for both **decomposing** a problem into subproblems and for **solving** subproblems
- It's a **blind search** procedure, **no heuristic** of any sort, *IW* **does not even know next goal** G_i "to achieve"

Markov Decision Process:

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** $0 < \gamma < 1$

A policy, π is a function that tells an agent which is the best action to choose in each state. A policy can be *deterministic* or *stochastic*.

A deterministic policy $\pi : S \to A$ is a *mapping* from states to actions. It specifies which action to choose in every possible state. Thus, if we are in state s, our agent should choose the action defined by $\pi(s)$.

A stochastic policy $\pi : S \times A \to \mathbb{R}$ specifies the *probability distribution* from which an agent should select an action. Intuitively, $\pi(s, a)$ specifies the probability that action a should be executed in state s.

Expected Value of Policy:

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

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

Discounted Reward in Bellman Equation:

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

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

Q-Function:

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

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

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.

Complexity: $O(|S|^2|A|)$

Extract Policy in Value Iteration:

$$\operatorname*{argmax}_{a \in A(s)} \sum_{s' \in S} P_a(s'|s)\ [r(s,a,s') + \gamma\ V(s')] \quad \text{or} \quad \operatorname*{argmax}_{a \in A(s)} Q(s,a)$$

Policy Iteration:

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

- Policy Evaluation:

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

- Policy Iteration:

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

$$\pi(s) := \operatorname{argmax}_{a \in A(s)} Q^{\pi}(a,s)$$

Complexity: $O(|S|^3 + |S|^2|A|)$

Partially Observable MDP:

- states $s \in S$
- actions $A(s) \subseteq A$
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- initial belief state b_0
- final belief state b_f
- sensor model given by probabilities $P_a(o|s)$, $o \in Obs$

Offline Planning vs Online Planning:

We saw value iteration and policy iteration in the previous lecture. These are **offline planning** methods, in that we solve the problem offline for all possible states, and then use the solution (a policy) online. In the offline planning approach:

- We can define policies, π, that work from any state in a convenient manner,
- Yet the state space S is usually *far* too big to determine $V(s)$ or π exactly.
- There *are* methods to approximate the MDP by reducing the dimensionality of S, but we will not discuss these until later.

In **online planning**, planning is undertaken immediately before executing an action. Once an action (or perhaps a sequence of actions) is executed, we start planning from the current state. As such, planning and execution are interleaved.

- For each state s visited, many policies π are *evaluated* (partially)
- The quality of each π is approximated by averaging the expected reward of trajectories over S obtained by repeated *simulations* of $r(s,a,s')$.
- The chosen policy $\hat{\pi}$ is then selected and the action $\hat{\pi}(s)$ executed.

Monte Carlo Tree Search:

4 Steps: Selection, Expansion, Simulation, Backpropagation.

- *Select*: Select a single node in the tree that is *not fully expanded*. By this, we mean at least one of its children is not yet explored.
- *Expand*: Expand this node by applying one available action (as defined by the MDP) from the node.
- *Simulation*: From one of the new nodes, perform a complete random simulation of the MDP to a terminating state. This might typically assume that the search tree is finite, however versions for infinitely large trees exist in which we just execute for some time and then estimate the outcome.
- *Backpropagate*: Finally, the value of the node is *backpropagated* to the root node, updating the value of each ancestor node on the way using expected value.

function MCTSSEARCH $M = \langle S, s_0, A, P_a(s'|s), R(s)\rangle$ **returns** action a
  while $current\_time < T$ **do**
    $expand\_node :=$ SELECT$(root)$;
    $children :=$ EXPAND$(expand_node)$;
    $child :=$ CHOOSE$(children)$; – choose a child to simulate
    $reward :=$ SIMULATE$(child)$; – simulate from $child$
    BACKUP$(expand_node, reward)$;
  return $\text{argmax}_a\ Q(s_0, a)$;

Uniform Sampling:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{t=1}^{N(s)} \mathbb{I}_t(s,a) r_t$$

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

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

r_t is the *reward* obtained by the t-th simulation from s.

$\mathbb{I}_t(s,a)$ is 1 if a was selected on the t-th simulation from s, and is 0 otherwise

Multi-Armed Bandit:

- Definition

Imagine that you have N number of slot machines (or poker machines in Australia), which are sometimes called one-armed bandits. Over time, each bandit pays a random reward from an unknown probability distrbution. Some bandits pay higher rewards than others. The goal is to maximize the sum of the rewards of a sequence of lever pulls of the machine.

- Regret: the expected loss due to not choosing the best action.

(Pseudo)–Regret

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

$Q(\pi^*(s), s)$ is the Q-value for the (unknown) optimal policy $\pi^*(s)$,

$N(s)$ is the number of visits to state s,

$\mathbb{I}_i(s,a)$ is 1 if a was selected on the i-th visit from s, and 0 otherwise,

Important: $\mathbb{E}[\sum_t^{N(s)} Q(b,s)\mathbb{I}_t(s,b)] > 0$ for every b.

Exploitation vs Exploration:

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 Confidence Bound:

UCB1 policy $\pi(s)$

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

$Q(s,a)$ 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.

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

UCT exploration policy

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

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

$\rightarrow$ if $Q(s,a) \in [0,1]$ **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).

Value/Policy Iteration vs MCTS:

	Value/policy iteration	MCTS
Cost	Higher cost (exhaustive)	Lower cost (does not solve for entire state space)
Coverage/ Robustness	Higher (works from any state)	Low (works only from initial state or state reachable from initial state)

Q-Learning: offline learning.

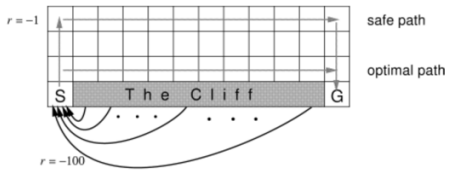
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
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

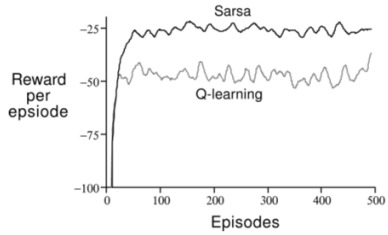
SARSA: online learning.

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: Q-learning learns an optimal path, while SARSA learns a safe path.

- Q-learning will converge to the optimal policy irrelevant of the policy followed, because it is *off-policy*: it uses the greedy reward estimate in its update rather than following the policy such as ϵ-greedy). Using a random policy, Q-learning will still converge to the optimal policy, but SARSA will not (necessarily).
- Q-learning learns an optimal policy, but this can be 'unsafe' or risky *during training*.





Rewards during training

Linear Function Approximation:

- Overall process

The overall process is:

1. For the states, consider what are the features that determine its representation.
2. During learning, perform updates based on the *weights* of *features* instead of states.
3. Estimate $Q(s,a)$ by summing the features and their weights.

- Representation of two vectors

To represent this, we have two vectors:

1. A *feature vector*, $f(s,a)$, which is a vector of $n \cdot |A|$ different functions, where n is the number of state features and $|A|$ the number of actions. Each function extracts the value of a feature for state-action pair (s,a). We say $f_i(s,a)$ extracts the ith feature from the state-action pair (s,a):

$$f(s,a) = \begin{pmatrix} f_1(s,a) \\ f_2(s,a) \\ \dots \\ f_n(s,a) \end{pmatrix}$$

2. A *weight* vector w of size $n \times |A|$: one weight for each feature-action pair. w_i^a defines the weight of a feature i for action a.

- state-action features:

It is straightforward to construct $n \times |A|$ state-pair features from just n state features:

$$f_{ik}(s,a) = \begin{cases} f_i(s) & \text{if } a = a_k \\ 0 & \text{otherwise} \end{cases} \quad 1 \le i \le n, 1 \le k \le |A|$$

This effectively results in $|A|$ different weight vectors: (s,a):

$$f(s,a_1) = \begin{pmatrix} f_{1,a_1}(s,a) \\ f_{2,a_1}(s,a) \\ 0 \\ 0 \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_2) = \begin{pmatrix} 0 \\ 0 \\ f_{1,a_2}(s,a) \\ f_{2,a_2}(s,a) \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_3) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ f_{1,a_3}(s,a) \\ f_{2,a_3}(s,a) \\ \dots \end{pmatrix} \quad \dots$$

- Q-function computation:

Give a feature vector f and a weight vector w, the Q-value of a state is a simple linear combination of features and weights:

$$\begin{aligned} Q(s,a) &= f_1(s,a) \cdot w_1^a + f_2(s,a) \cdot w_2^a + \dots + f_n(s,a) \cdot w_n^a \\ &= \sum_{i=0}^{n} f_i(s,a) w_i^a \end{aligned}$$

- Q-function parameter update (linear approximation):

Q-learning, the update rule is now:

$$w_i^a \leftarrow w_i^a + \alpha[r + \gamma max_a' Q(s', a') - Q(s, a)] f_i(s, a)$$

For SARSA:

$$w_i^a \leftarrow w_i^a + \alpha[r + \gamma Q(s', a') - Q(s, a)] f_i(s, a)$$

- Q-function parameter update (neural network):

The TD update for Q-learning is just:

$$\theta \leftarrow \theta + \alpha[r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)] \nabla_\theta Q(s, a; \theta)$$

Reward Shaping:

The approach to reward shaping is not to modify the reward function, but to just give additional reward for some moves:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \underbrace{F(s, s')}_{\text{additional reward}} + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Potential-based Reward Shaping:

Potential-based reward shaping is a particular type of reward shaping with nice theoretical guarantees. In potential-based reward shaping, F is of the form:

$$F(s, s') = \gamma \Phi(s') - \Phi(s)$$

We call Φ the *potential function* and $\Phi(s)$ is the potential of state s.

Q-function Initialization:

Q-function initialisation is similar to reward shaping: we use heuristics to assign higher values to 'better' states. If we just define $\Phi(s) = V(s)$, then they are equivalent. In fact, if our potential function is *static* (the definition does not change during learning), then Q-function initialisation and reward shaping are equivalent[1].

TD(λ)

We will look at TD(λ), in which λ is the parameter that determines n: the number of steps that we want to look ahead before updating the Q-function. Thus, TD(0) is just 'standard' reinforcement learning, and TD(1) looks one step beyond the immediate reward, TD(2) looks two steps beyond, etc.

Truncated Discounted Reward:

However, we can estimate a two-step return:

$$G_t^2 = r_t + \gamma r_{t+1} + \gamma^3 V(s_{t+2})$$

or three-step return:

$$G_t^3 = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+2})$$

or n-step returns:

$$G_t^n = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \gamma^n V(s_{t+n})$$

If T is the termination step and $t+n > T$, then we just use the full reward.

Q-function Update for n-step RL:

- Truncated discounted reward:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_i$$

- Q-function update:

$$\text{If } \tau + n < T \text{ then } G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$$
$$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$$

n-step SARSA:

n-step Sarsa for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to a fixed given policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t+1$
  | else:
  | Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

Game Theory:

- Normal form

What is Nash equilibrium; pure equilibrium vs mixed equilibrium; strongly dominant strategy vs weakly dominant strategy; expected payoff; indifference of mixed equilibrium.

- Extensive normal form

What is backward induction; Bayesian Game, e.g.

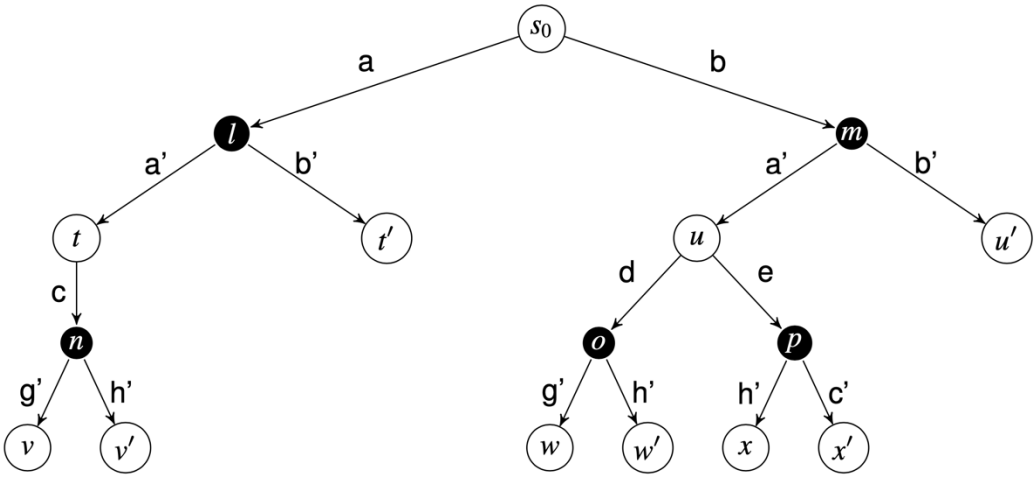
Adversary (P=0.3)

Defender	T1	T2
T1	5, -3	-1, 1
T2	-5, 5	2, -1

Adversary (P=0.7)

Defender	T1	T2
T1	5, -2	-1, 3
T2	-5, 2	2, -2

Game Tree:



MCTS for Games:

- 4 Steps:

1. For 'our' moves, we run *selection* as before, however, we also need to select models for our opponents (more on this in the next slide).
2. In the *expansion* step, instead of expanding all child nodes of an action, we run the simulation forward one step. How do we choose the outcome? We simply choose an action for our opponents!
3. We then simulate as before, and we learn the rewards when we receive them.
4. In the *backpropagation* step, instead of using the Bellman equation to calculate the expected return, we simply use the *average* return. If we simulate each step enough times, the average will converge to the expected return.

- Choose actions for opponents:

1. *Random*: Select a random action. This is easy, but it means that we may end up exploring a lot of actions that will never be taken by a good opponent.
2. *Using a fixed policy*: Hand-code a simple stochastic policy that gives reasonable behaviour of the opponent.
3. *Simultaneously learn a policy*: We *learn* a policy for ourselves and our opponents, and we choose actions for our opponents based on the learnt policies.