Sequential Decision Making and Reinforcement Learning

(SDMRL)

Model Based Planning

Sarah Keren

The Taub Faculty of Computer Science
Technion - Israel Institute of Technology

Agenda

- Planning in deterministic fully observable domains
- Accounting for stochastic action outcomes
- Accounting for partial observability

Do we still need to know about planning?

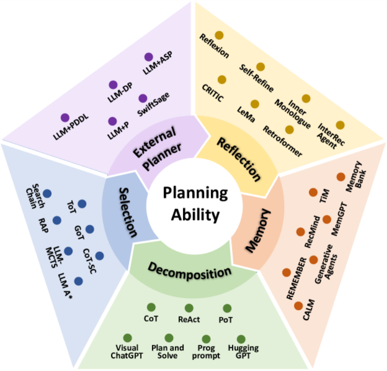


Figure 1: LLms for

https://ar5iv.labs.arxiv.org/html/2402.02716

Planning for Deterministic Domains

Planning and sequential decision making

In the classical planning setting:

Plans (aka solutions) are sequences of moves that transform the initial state into the goal state

What is our task?

1. Find out whether there is a solution
2. Find any solution
3. Find an optimal solution
4. Find near-optimal solution
5. Fixed amount of time, find best solution possible
6. Find solution that satisfy property $\aleph$
   (what is $\aleph$? you choose!)

♠ While all these tasks sound related, they are very different. The best suited techniques are almost disjoint.

Back to deterministic transition systems

A transition system is deterministic if there is only one initial state and all actions are deterministic. Hence all future states of the world are completely predictable.

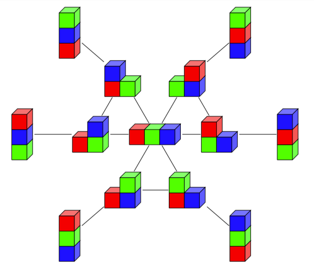
Definition (deterministic transition system)

A deterministic transition system is $\langle S, s_0, A, S_G, c \rangle$ where

- finite state space S
- an initial state $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- applicable actions $A(s) \subseteq A$ for $s \in S$
- a transition function $s' = f(a, s)$ for $a \in A(s)$
- a cost function $c : A^* \to [0, \infty)$

Blocks World

blocks	states
1	1
2	
3	1
4	7
5	50
6	405
7	3763
8	39435
9	459655
...	
19	13564373693588558173



1. Finding a solution is polynomial time in the number of blocks (move everything onto the table and then construct the goal configuration).
2. Finding a shortest solution is NP-hard ...

Planning problems (reminder)

- Route selection (from Arad to Bucharest)
- Solving 15-puzzle (or Rubik's cube, or ...)
- Selecting and ordering movements of an elevator or a crane
- Production lines control
- Autonomous robots
- Crisis management
- ...

State-space search

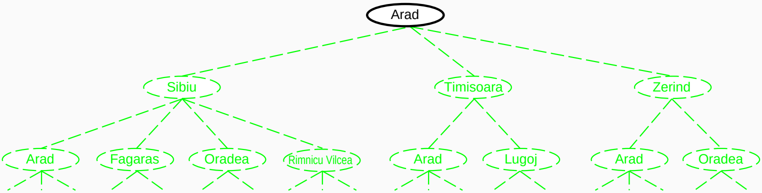
- State-space search: one of the big success stories of AI
- Must carefully distinguish two different problems:
  - satisficing planning: any solution is OK (although shorter solutions typically preferred)
  - optimal planning: plans must have shortest possible length
- Both are often solved by search, but:
  - details are very different
  - almost no overlap between good techniques for satisficing planning and good techniques for optimal planning
  - many problems that are trivial for satisficing planners are impossibly hard for optimal planners

Planning by forward search: progression

Progression: Computing the successor state $f(s,a)$ of a state s with respect to an action a.

Progression planners find solutions by forward search:

- start from initial state
- iteratively pick a previously generated state and progress it through an action, generating a new state
- solution found when a goal state generated

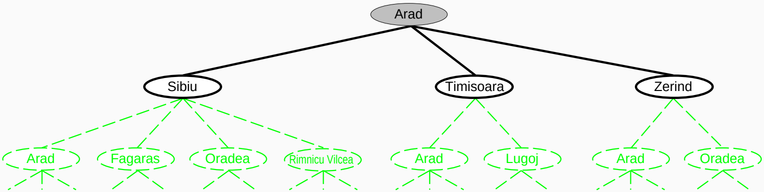


Planning by forward search: progression

Progression: Computing the successor state $f(s, a)$ of a state s with respect to an action a.

Progression planners find solutions by forward search:

- start from initial state
- iteratively pick a previously generated state and progress it through an action, generating a new state
- solution found when a goal state generated

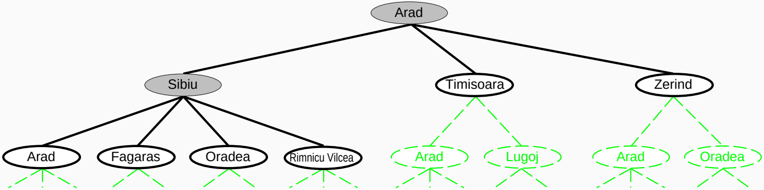


Planning by forward search: progression

Progression: Computing the successor state $f(s,a)$ of a state s with respect to an action a.

Progression planners find solutions by forward search:

- start from initial state
- iteratively pick a previously generated state and progress it through an action, generating a new state
- solution found when a goal state generated



Classification of search algorithms

uninformed search vs. heuristic search:

- uninformed search algorithms only use the basic ingredients for general search algorithms
- heuristic search algorithms additionally use heuristic functions which estimate how close a node is to the goal

systematic search vs. local search:

- systematic 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
- not a black-and-white distinction; there are crossbreeds (e.g., enforced hill-climbing)

Uninformed search algorithms

Popular uninformed systematic search algorithms:

- breadth-first search
- depth-first search
- iterated depth-first search

Popular uninformed local search algorithms:

- random walk

Evaluating Algorithms

Dimensions for evaluation

- completeness: always find a solution if one exists?
- time complexity: number of nodes generated/expanded
- space complexity: number of nodes in memory
- optimality: does it always find a least-cost solution?
- anytime: does the solution improve the more resources are used ?

Time/space complexity measured in terms of

- b maximum branching factor of the search tree
- d depth of the least-cost solution
- m maximum depth of the state space (may be ∞)

Properties of breadth-first search

Complete	Yes (if b is finite)
Time	$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., $\exp(d)$
Space	$O(b^{d+1})$ *(why?)*
Optimal	Yes (if cost = 1 per step); can be generalized *(how?)*

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Heuristic search algorithms: systematic

- Heuristic search algorithms are the most common and overall most successful algorithms for deterministic planning.

Popular systematic heuristic search algorithms:

- greedy best-first search
- A*
- weighted A*
- id-a*
- depth-first branch-and-bound search
- breadth-first heuristic search
- ...

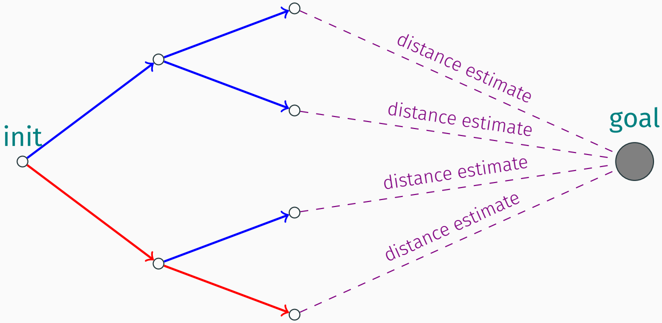
Heuristic search algorithms: local

- Heuristic search algorithms are the most common and overall most successful algorithms for deterministic planning.

Popular heuristic local search algorithms:

- hill-climbing
- enforced hill-climbing
- beam search
- tabu search
- genetic algorithms
- simulated annealing
- ...

Heuristic search: idea



Required ingredients for heuristic search

A heuristic search algorithm requires one more operation in addition to the definition of a search space.

Definition (heuristic function)

Let Σ be the set of nodes of a given search space.

A heuristic function or heuristic (for that search space) is a function $h : \Sigma \to \mathbb{N}_0 \cup \{\infty\}$.

- The value $h(\sigma)$ supposed to estimate the distance from node σ to the nearest goal node.
- Typically: $h(\sigma) \stackrel{\text{def}}{=} h(\mathit{state}(\sigma))$

What exactly is a heuristic estimate?

What does it mean that h “estimates the goal distance”?

- For most heuristic search algorithms, h does not need to have any strong properties for the algorithm to work
- However, the efficiency of the algorithm closely relates to how accurately h reflects the actual goal distance.
- For some algorithms, like A*, we can prove strong formal relationships between properties of h and properties of the algorithm (optimality, dominance, run-time for bounded error, ...)
- For other search algorithms, “it works well in practice” is often as good an analysis as one gets.

Perfect heuristic

Let Σ be the set of nodes of a given search space.

Definition (optimal/perfect heuristic)

The optimal or perfect heuristic of a search space is the heuristic h^* which maps each search node σ to the length of a shortest path from *state*(σ) to any goal state.

Note: $h^*(\sigma) = \infty$ iff no goal state is reachable from σ.

Properties of heuristics

A heuristic h is called

- safe if for all $\sigma \in \Sigma$ $h(\sigma) = \infty$ implies $h^*(\sigma) = \infty$
- goal-aware if $h(\sigma) = 0$ for all goal nodes $\sigma \in \Sigma$
- admissible if $h(\sigma) \leq h^*(\sigma)$ for all nodes $\sigma \in \Sigma$
- consistent if $h(\sigma) \leq h(\sigma') + cost(\sigma, \sigma')$
  for all nodes $\sigma, \sigma' \in \Sigma$ such that σ' is a successor of σ

Relationships?

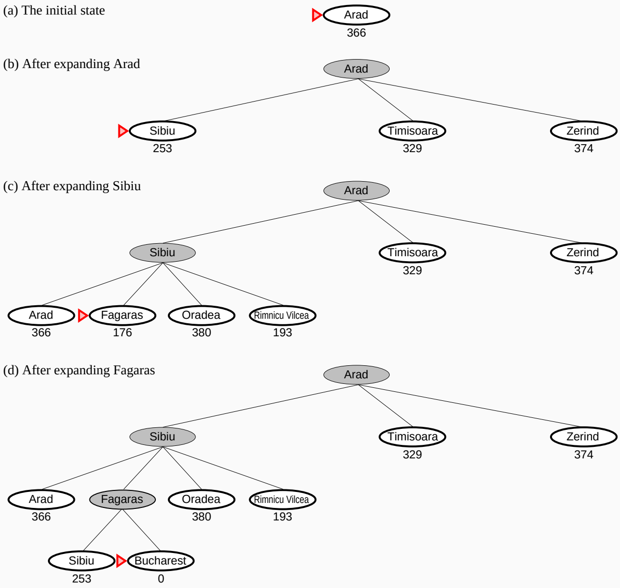
Greedy best-first search (with duplicate detection)

```
open := new min-heap ordered by (σ ↦ h(σ))
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    σ = open.pop-min()
    if state(σ) ∉ closed:
        closed := closed ∪ {state(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ' := make-node(σ, o, s)
            if h(σ') < ∞:
                open.insert(σ')
return unsolvable
```

Planning trip to Bucharest with SLD heuristic

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

GBFS by example



Properties of greedy best-first search

- one of the three most commonly used algorithms for satisficing planning
- complete for safe heuristics (due to duplicate detection)
- suboptimal unless h satisfies some very strong assumptions (similar to being perfect)
- invariant under all strictly monotonic transformations of h (e.g., scaling with a positive constant or adding a constant)

Chooses next node to expand based on $f(n) = g(n) + h(n)$

1. $g(n)$ Distance from start
2. $h(n)$ Heuristic function that estimates the expected distance from goal

A heuristic is *admissible* if it is 'optimisitic': it underestimates the cost to goal.

Key points:

- As long as the heuristic is admissible then A* is guaranteed to return an optimal solution.
- Trade-of between estimation quality and computation cost.
- h = straight-line / Manhattan distance is a good heuristic for motion planning.

A* (with duplicate detection and reopening)

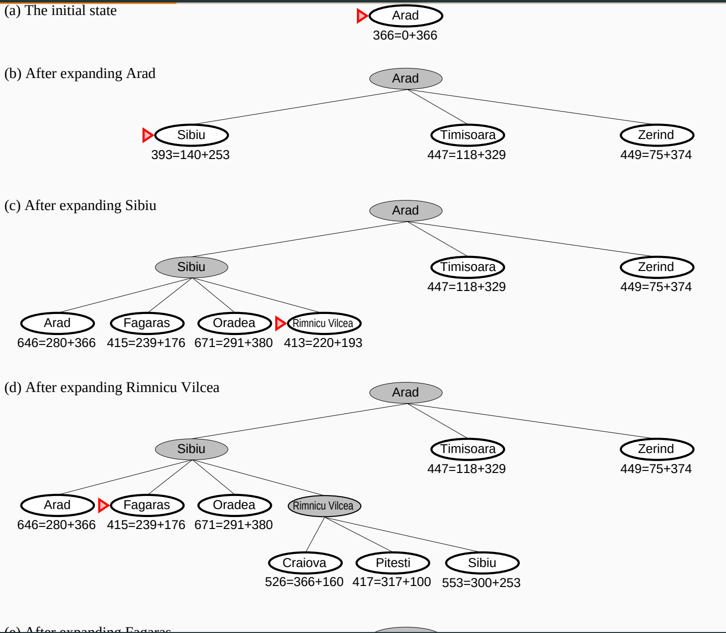
```
open := new min-heap ordered by (σ ↦ f(σ) = g(σ) + h(σ))
open.insert(make-root-node(init()))
closed := ∅
distance := ∅
while not open.empty():
    σ = open.pop-min()
    if state(σ) ∉ closed or g(σ) < distance(state(σ)):
        closed := closed ∪ {state(σ)}
        distance(state(σ)) := g(σ)
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ′ := make-node(σ, o, s)
            if h(σ′) < ∞:
                open.insert(σ′)
return unsolvable
```

A*(one node maintained per state)

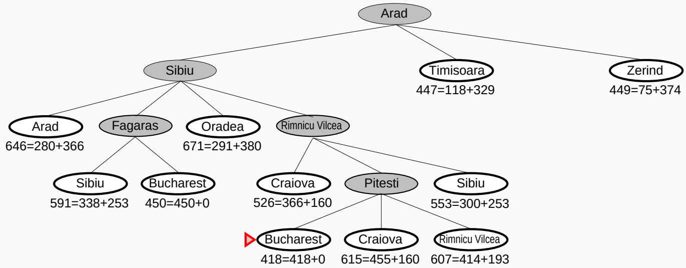
A* (with duplicate detection and reopening)

open := **new** min-set-heap ordered by $(\sigma \mapsto f(\sigma) = g(\sigma) + h(\sigma))$
open.insert(make-root-node(init()))
closed := $\emptyset$
distance := $\emptyset$
while not *open*.empty():
    σ = *open*.pop-min()
    if *state*(σ) $\notin$ *closed* **or** $g(\sigma) <$ *distance*(*state*(σ)):
        closed := *closed* $\cup$ {*state*(σ)}
        distance(*state*(σ)) := $g(\sigma)$
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each $\langle o, s\rangle \in$ succ(*state*(σ)):
            σ' := make-node(σ, o, s)
            if $h(\sigma') < \infty$:
                open.insert(σ')
return unsolvable

A* by example


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* by example

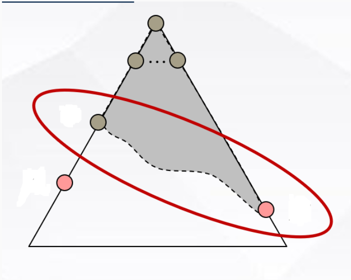


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

Properties of A*

- the most commonly used algorithm for optimal planning
- rarely used for satisficing planning
- complete for safe heuristics
  (even without duplicate detection)
- optimal for admissible heuristics

Optimality of Tree-Search A* with admissible h



Optimality of Tree-Search A* with admissible h

Suppose to the contrary that the algorithm returns a suboptimal plan via *extract-solution*(σ) for some node σ with *state*$(\sigma) \in G$.

- If so, then no optimal goal node σ^* was in the open list (right?)
- If so, then open list contains some unexpanded node σ' on the (optimal) path from root node to σ^*

Optimality of Tree-Search A* with admissible h

Suppose to the contrary that the algorithm returns a suboptimal plan via *extract-solution*(σ) for some node σ with *state*$(\sigma) \in G$.

- If so, then no optimal goal node σ^* was in the open list (right?)
- If so, then open list contains some unexpanded node σ' on the (optimal) path from root node to σ^*

$$
\begin{aligned}
f(\sigma) = \quad & g(\sigma) && \text{since } h(\sigma) = 0 \\
> \quad & g(\sigma^*) && \text{since } \sigma \text{ is suboptimal} \\
\geq \quad & f(\sigma') && \text{since } h \text{ is admissible}
\end{aligned}
$$

$f(\sigma) > f(\sigma')$ $\rightsquigarrow$ contradiction with "σ is dequeued before σ'"

Weighted A* (with duplicate detection and reopening)

open := **new** min-heap ordered by $(\sigma \mapsto g(\sigma) + W \cdot h(\sigma))$
open.insert(make-root-node(init()))
closed := $\emptyset$
distance := $\emptyset$
while not *open*.empty():
    σ = *open*.pop-min()
    if *state*(σ) $\notin$ *closed* **or** $g(\sigma) <$ *distance*(*state*(σ)):
        closed := *closed* $\cup$ {*state*(σ)}
        distance(σ) := $g(\sigma)$
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each $\langle o, s\rangle \in$ succ(*state*(σ)):
            σ' := make-node(σ, o, s)
            if $h(\sigma') < \infty$:
                open.insert(σ')
return unsolvable

The weight $W \in \mathbb{R}_0^+$ is a parameter of the algorithm.

- for $W = 0$, behaves like breadth-first search
- for $W = 1$, behaves like A*
- for $W \to \infty$, behaves like greedy best-first search

Properties:

- one of the three most commonly used algorithms for satisficing planning
- for $W > 1$, can prove similar properties to A*, replacing optimal with bounded suboptimal: generated solutions are at most a factor W as long as optimal ones

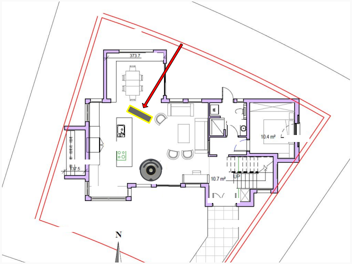
Best First Search

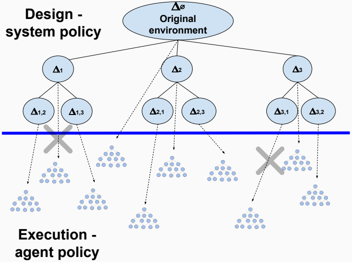
Algorithm 1 Best First Search (BFS)

BFS(P, h)

create OPEN list for unexpanded nodes
put $\langle P.root, h(P.root) \rangle$ in OPEN
$n_{cur} = ExtractMax(OPEN)$ (initial model)
while n_{cur} **do**
  if $IsTerminal(n_{cur})$ **then**
    return $ExtractPath(n_{cur})$ (best solution found - exit)
  end if
  for all $n_{suc} \in GetSuccessors(n_{cur}, P)$ **do**
    put $\langle n_{suc}, h(n_{suc}) \rangle$ in OPEN
  end for
  $n_{cur} = ExtractMax$(OPEN)
end while
return no solution

Best First Design





Admissible heuristics **over-estimate** the value of a modification sequence.

Best First Design

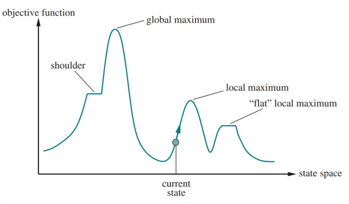
Algorithm 2 Best First Design (BFD)

BFD(δ, h)

create OPEN list for unexpanded nodes
$n_{cur} = \langle design, \vec{m}_{\emptyset} \rangle$ (initial model)
while n_{cur} **do**
  if $IsExecution(n_{cur})$ **then**
    return $n_{cur}.\vec{m}$ (best modification found - exit)
  end if
  for all $n_{suc} \in GetSuccessors(n_{cur}, \delta)$ **do**
    put $\langle\langle design, n_{suc}.\vec{m}\rangle, h(n_{suc})\rangle$ in OPEN
  end for
  if $\Phi_{\sigma}(n_{cur}.\vec{m}) = 1$ **then**
    put $\langle\langle execution, \vec{m}_{new}\rangle, v^{*}(\delta_{\vec{m}_{new}})\rangle$ in OPEN
  end if
  $n_{cur} = ExtractMax($OPEN$)$
end while
return error

Hill-climbing

A local search algorithm that incrementally improves the solution by exploring the neighboring states of the current state.



Hill-climbing

σ := make-root-node(init())
forever:
    if is-goal(state(σ)):
        return extract-solution(σ)
    Σ' := { make-node(σ, o, s) | $\langle o, s \rangle \in$ succ(state(σ)) }
    σ := an element of Σ' minimizing h (random tie breaking)

- can get stuck in local minima where immediate improvements of $h(\sigma)$ are not possible
- many variations: tie-breaking strategies, restarts

Enforced hill-climbing: procedure improve

```
def improve(σ0):
    queue := new fifo-queue
    queue.push-back(σ0)
    closed := ∅
    while not queue.empty():
        σ = queue.pop-front()
        if state(σ) ∉ closed:
            closed := closed ∪ {state(σ)}
            if h(σ) < h(σ0):
                return σ
            for each ⟨o, s⟩ ∈ succ(state(σ)):
                σ' := make-node(σ, o, s)
                queue.push-back(σ')
    fail
```

$\rightsquigarrow$ breadth-first search for more promising node than σ_0

Enforced hill-climbing (ctd.)

Enforced hill-climbing

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

- one of the three most commonly used algorithms for satisficing planning
- can fail if procedure *improve* fails (when the goal is unreachable from σ_0)
- complete for undirected search spaces (where the successor relation is symmetric) if $h(\sigma) = 0$ for all goal nodes and only for goal nodes

Classification: what works where in (deterministic) planning?

uninformed vs. heuristic search:

systematic search vs. local search:

Classification: what works where in (deterministic) planning?

uninformed vs. heuristic search:

- For satisficing planning, heuristic search vastly outperforms uninformed algorithms on most domains.
- For optimal planning, heuristic search typically outperforms uninformed algorithms, but an efficiently implemented uninformed algorithm is not easy to beat in most domains.

systematic search vs. local search:

- For satisficing planning, the most successful algorithms are somewhere between the two extremes.
- For optimal planning, systematic algorithms are required.

Where heuristics come from?

General idea

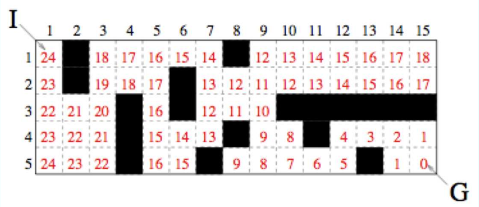
(Admissible) heuristic functions obtained as (optimal) cost functions of relaxed problems

Examples

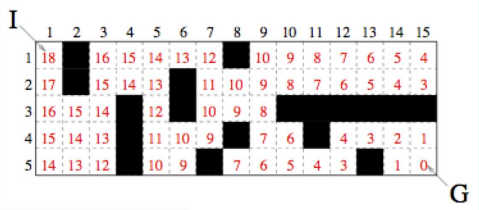
- Euclidian distance in Path Finding
- Manhattan distance in N-puzzle
- Spanning Tree in Traveling Salesman Problem
- Shortest Path in Job Shop Scheduling

Example

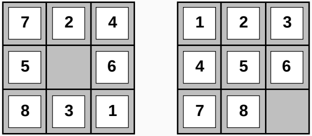
True distance h^* for different search states



Manhattan distance is based on the relaxation that …?



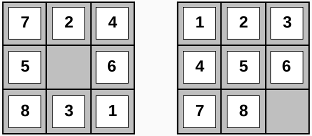
Example



Start State **Goal State**

- A tile can move from square A to square B if A is adjacent to B and B is blank $\rightsquigarrow$ solution distance h^*
- A tile can move from square A to square B if A is adjacent to B $\rightsquigarrow$ manhattan distance heuristic h^{MD}
- A tile can move from square A to square B $\rightsquigarrow$ misplaced tiles heuristic h^{MT}

Example



Start State   Goal State

- A tile can move from square A to square B if A is adjacent to B and B is blank $\rightsquigarrow$ solution distance h^*
- A tile can move from square A to square B if A is adjacent to B $\rightsquigarrow$ manhattan distance heuristic h^{MD}
- A tile can move from square A to square B $\rightsquigarrow$ misplaced tiles heuristic h^{MT}

Here: $h^*(s_0) = ?$, $h^{MD}(s_0) = 14$, $h^{MT}(s_0) = 6$

In general, $h^* \geq h^{MD} \geq h^{MT}$. *(Why?)*

Relaxations as Heuristics

Relaxations can be used for heuristic estimations.

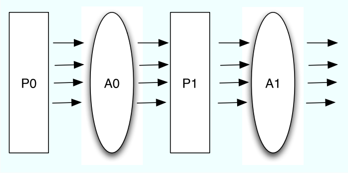
Different possibilities:

- Implement an optimal planner for relaxed planning tasks and use its solution lengths as an estimate, even though it is NP-hard.
  $\rightsquigarrow$ h^+ heuristic -ingore delete effects.
- Do not actually solve the relaxed planning task, but compute an estimate of its difficulty in a different way.
  $\rightsquigarrow$ h_{max} heuristic, h_{add} heuristic
- Compute a solution for relaxed planning tasks which is not necessarily optimal, but "reasonable".
  $\rightsquigarrow$ h_{FF} heuristic

Graphical “interpretation”: Relaxed planning graphs

- Build a layered reachability graph $P_0, A_0, P_1, A_1, \ldots$

$$
\begin{aligned}
P_0 &= \{p \in I\} \\
A_i &= \{a \in A \mid pre(a) \subseteq P_i\} \\
P_{i+1} &= P_i \cup \{p \in add(a) \mid a \in A_i\}
\end{aligned}
$$



- Terminate when $G \subseteq P_i$

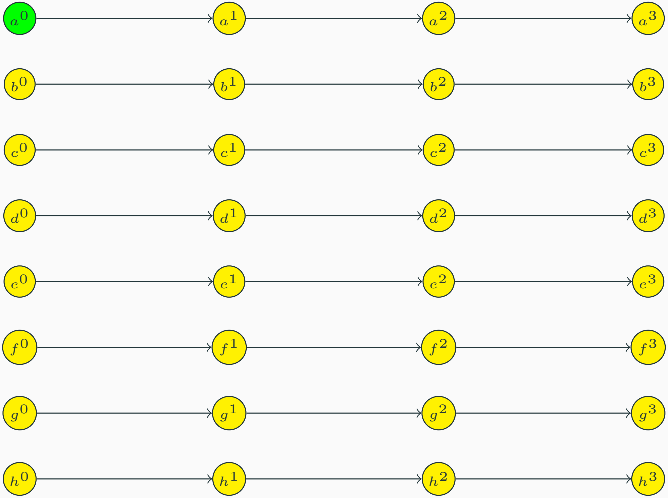
Running example

$$P = \{a, b, c, d, e, f, g, h\}$$
$$I = \{a\}$$
$$G = \{c, d, e, f, g\}$$
$$a_1 = \langle \{a\}, \{b, c\}, \{a\} \rangle$$
$$a_2 = \langle \{a, c\}, \{d\}, \{d\} \rangle$$
$$a_3 = \langle \{b, c\}, \{e\}, \{e, f\} \rangle$$
$$a_4 = \langle \{b\}, \{f\}, \emptyset \rangle$$
$$a_5 = \langle \{d\}, \{e, f\}, \{d\} \rangle$$
$$a_6 = \langle \{d\}, \{g\}, \{b\} \rangle$$

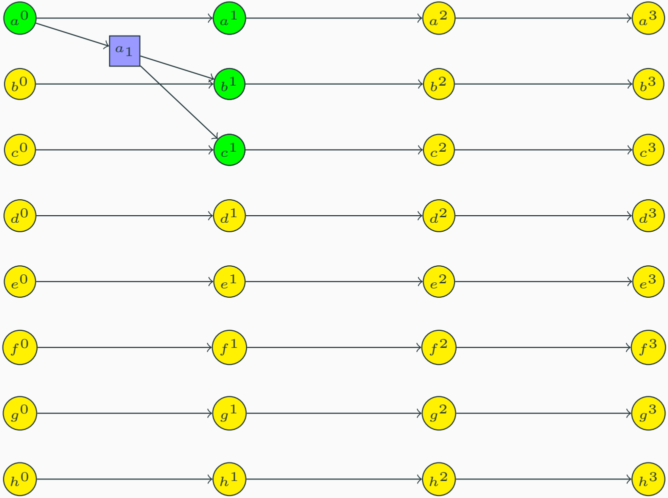
Running example

$$P = \{a, b, c, d, e, f, g, h\}$$
$$I = \{a\}$$
$$G = \{c, d, e, f, g\}$$
$$a_1 = \langle \{a\}, \{b, c\}, \emptyset \rangle$$
$$a_2 = \langle \{a, c\}, \{d\}, \emptyset \rangle$$
$$a_3 = \langle \{b, c\}, \{e\}, \emptyset \rangle$$
$$a_4 = \langle \{b\}, \{f\}, \emptyset \rangle$$
$$a_5 = \langle \{d\}, \{e, f\}, \emptyset \rangle$$
$$a_6 = \langle \{d\}, \{g\}, \emptyset \rangle$$

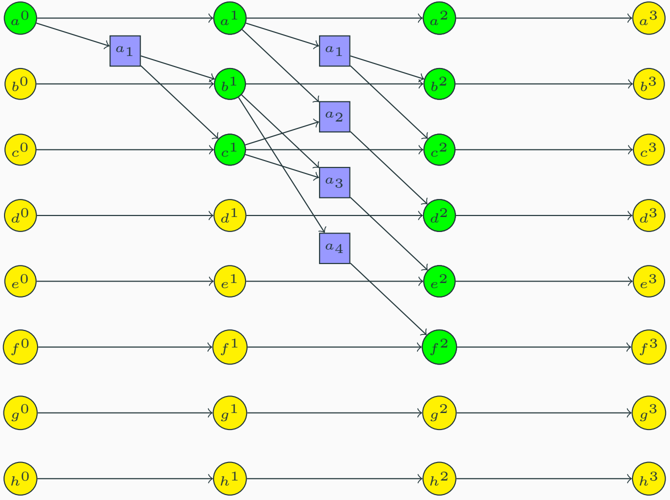
Running example: Relaxed planning graph



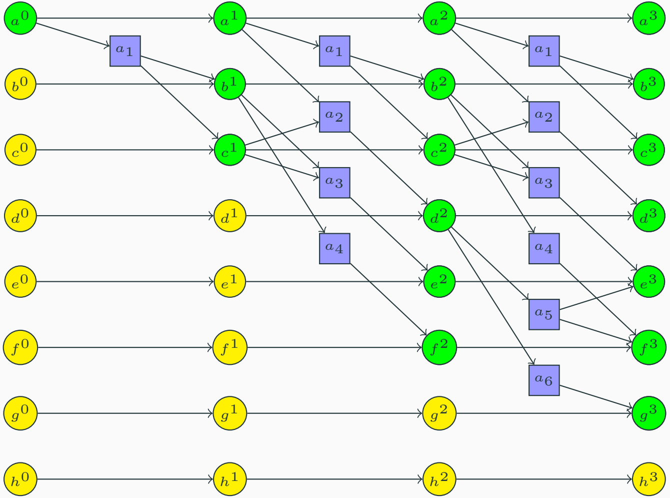
Running example: Relaxed planning graph



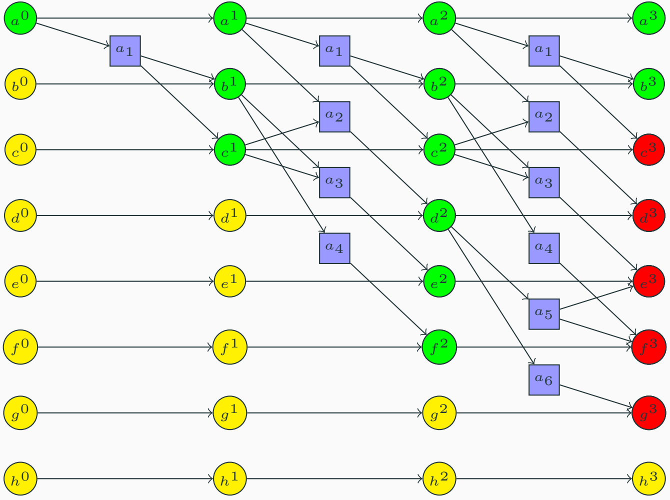
Running example: Relaxed planning graph



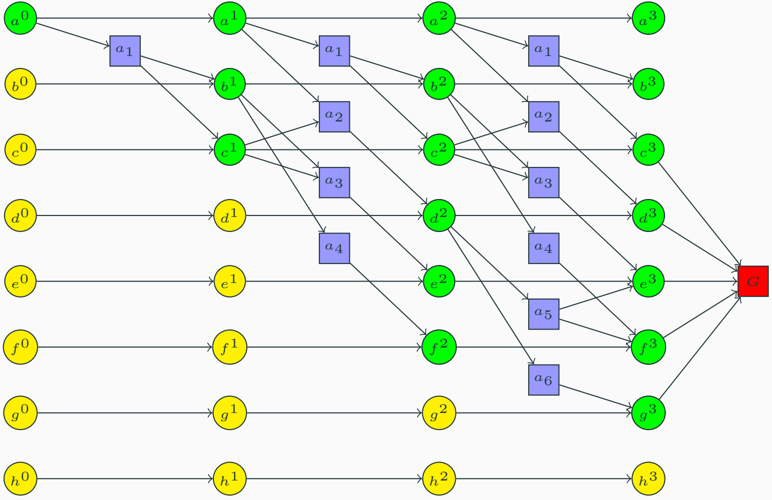
Running example: Relaxed planning graph



Running example: Relaxed planning graph



Running example: Relaxed planning graph



Dominance relation between admissible heuristics

Precision matters

Given two admissible heuristics h_1, h_2, if $h_2(\sigma) \geq h_1(\sigma)$ for all search nodes σ, then h_2 dominates h_1 and is better for optimizing search

Typical search costs (unit-cost action)

$h^*(I) = 14$ BFS ≈ 1,700,000 nodes
A$^*(h_1)$ ≈ 560 nodes
A$^*(h_2)$ ≈ 115 nodes

$h^*(I) = 24$ BFS ≈ 27,000,000,000 nodes
A$^*(h_1)$ ≈ 40,000 nodes
A$^*(h_2)$ ≈ 1,650 nodes

Dominance relation between admissible heuristics

Precision matters

Given two admissible heuristics h_1, h_2, if $h_2(\sigma) \geq h_1(\sigma)$ for all search nodes σ, then h_2 dominates h_1 and is better for optimizing search

Combining admissible heuristics

For any admissible heuristics $h_1, \ldots, h_k$,

$$h(\sigma) = \max_{i=1}^{k}\{h_i(\sigma)\}$$

is also admissible and dominates all individual h_i

Are we done?

General idea

(Admissible) heuristic functions obtained as (optimal) cost functions of relaxed problems

- OK, but heuristic is yet another input to our agent!
- Satisfactory for general solvers?
- Satisfactory in special purpose solvers?

Are we done?

General idea

(Admissible) heuristic functions obtained as (optimal) cost functions of relaxed problems

- OK, but heuristic is yet another input to our agent!
- Satisfactory for general solvers?
- Satisfactory in special purpose solvers?

Towards domain-independent agents

- How to get heuristics automatically?
- Can such automatically derived heuristics dominate the domain-specific heuristics crafted by hand?

Example Heuristics

Planning for Stochastic Domains

What are stochastic domains?
How do we model them?

Remidner: Markov Decision Process

A **Markov Decision Process**(MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where

- $\mathcal{S}$ is a finite set of states, defined over a set of variables $\mathcal{X}$
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix
  $\mathcal{P}^a_{s,s'} = \mathcal{P}[S_{t+1} = s' | S_t = s, A_t = a]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$, and
- **optional:** γ is a discount factor $\gamma \in [0, 1]$ that is used to favor immediate rewards over future rewards.

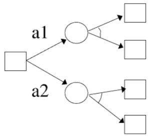
The Markov property: "The future is independent of the past given the present".

Extensions: Infinite and continuous MDPs, partially observable MDPs, undiscounted, average reward MDPs. etc.

And-Or Graphs

- For conditional / probabilistic planning we need to take some action at every state, but must handle every outcome for the action.
- A solution is a conditional plan / policy rather than just a single move.
- For this purpose we use and-or-graphs with two kinds of nodes:
  - **OR node** - specify a selection between actions
  - **AND node** - specify the possible outcomes

Where have you already used an and-or graph ?



Reminder:

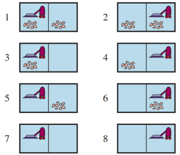


Figure 4.8 The eight possible states of the vacuum world; states 7 and 8 are goal states.

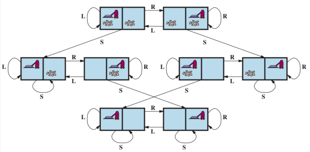
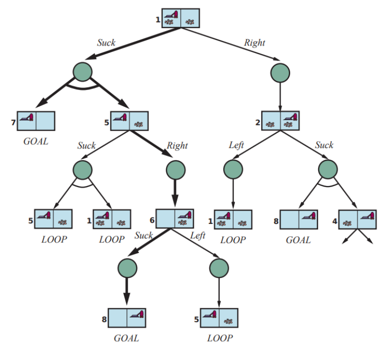


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

From Artificial Intelligence A Modern Approach by Russel and Norvig

And-Or Graphs - The erratic vacuum world

In this version, when Suck is applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too. When applied to a clean square the action sometimes deposits dirt on the carpet.



From Artificial Intelligence A Modern Approach by Russel and Norvig

And-Or Search

function OR-SEARCH(*problem*, *state*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.IS-GOAL(*state*) **then return** the empty plan
  if IS-CYCLE(*path*) **then return** *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) **do**
    $plan \leftarrow$ AND-SEARCH(*problem*, RESULTS(*state*, *action*), [*state*] + *path*])
    if $plan \neq failure$ **then return** [*action*] + *plan*]
  return *failure*

function AND-SEARCH(*problem*, *states*, *path*) **returns** *a conditional plan, or failure*
  for each s_i **in** *states* **do**
    $plan_i \leftarrow$ OR-SEARCH(*problem*, s_i, *path*)
    if $plan_i = failure$ **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** $\ldots$ **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

What happens when we have probabilities associated with action outcomes ?

What happens when the state space is large ?

Any ways to increase efficiency ?

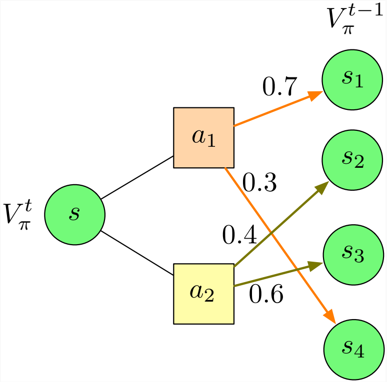
From Artificial Intelligence A Modern Approach by Russel and Norvig

Policy Optimization: Bellman Backups

How can we compute compute $\mathcal{V}_\pi^t(s)$ given $\mathcal{V}_\pi^{t-1}(s)$?

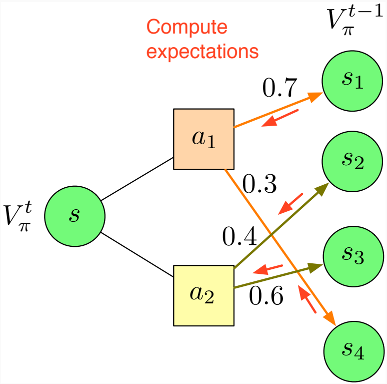
Where are the rewards ?

How can we compute compute $\mathcal{V}_{\pi}^{t}(s)$ given $\mathcal{V}_{\pi}^{t-1}(s)$?

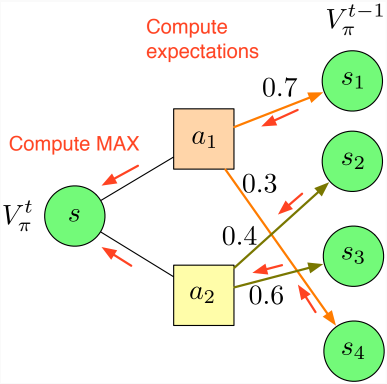


Where are the rewards ?

How can we compute compute $\mathcal{V}_\pi^t(s)$ given $\mathcal{V}_\pi^{t-1}(s)$?



How can we compute compute $\mathcal{V}_\pi^t(s)$ given $\mathcal{V}_\pi^{t-1}(s)$?



The value of a state under an optimal policy must equal the expected return for the best action from that state

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

Figure 2: Richard E. Bellman

The Bellman Optimality Equation

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s,a)$$

The Bellman Optimality Equation

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi*}(s, a)$$

$$= \max_a \mathbb{E}_{\pi*} \left[G_t | S_t = s, A_t = a \right]$$

The Bellman Optimality Equation

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

$$= \max_a \mathbb{E}_{\pi^*} \left[G_t | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a \right]$$

The Bellman Optimality Equation

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

$$= \max_a \mathbb{E}_{\pi^*} \left[G_t | S_t = s, A_t = a\right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a\right]$$

$$= \max_a \mathbb{E} \left[R_{t+1} + \gamma \mathcal{V}^*(S_{t+1}) | S_t = s, A_t = a\right]$$

The Bellman Optimality Equation

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s,a)$$

$$= \max_a \mathbb{E}_{\pi^*} \left[G_t | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E} \left[R_{t+1} + \gamma \mathcal{V}^*(S_{t+1}) | S_t = s, A_t = a \right]$$

$$= \max_{a \in A(s)} \sum_{s',r} p(s', r | s, a) \left[r + \gamma \mathcal{V}^*(s') \right]$$

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

$$= \max_a \mathbb{E}_{\pi^*} [G_t | S_t = s, A_t = a]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a \right]$$

$$= \max_a \mathbb{E} \left[R_{t+1} + \gamma \mathcal{V}^*(S_{t+1}) | S_t = s, A_t = a \right]$$

$$= \max_{a \in A(s)} \sum_{s', r} p(s', r | s, a) \left[r + \gamma \mathcal{V}^*(s') \right]$$

$$Q^*(s, a) = \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A = a \right]$$

The Bellman optimality equation for Q^* is

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s,a)$$

$$= \max_a \mathbb{E}_{\pi^*}\left[G_t | S_t = s, A_t = a\right]$$

$$= \max_a \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

$$= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a\right]$$

$$= \max_a \mathbb{E}\left[R_{t+1} + \gamma \mathcal{V}^*(S_{t+1}) | S_t = s, A_t = a\right]$$

$$= \max_{a \in A(s)} \sum_{s',r} p(s',r|s,a)\left[r + \gamma \mathcal{V}^*(s')\right]$$

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1},a') | S_t = s, A = a\right]$$

$$= \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

The Bellman Optimality Equation

We want a policy but we got values, what to do ?

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

We want a policy but we got values, what to do ?

$$\mathcal{V}^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a)$$

Once we have $\mathcal{V}^*$ or Q^*, it is easy to determine an optimal policy: for each state, there will be one or more actions at which the maximum is obtained according to the Bellman optimality equation.

In many places in the literature you might see this.

$$V(s) = \max_a(R(s,a) + \gamma V(s'))$$

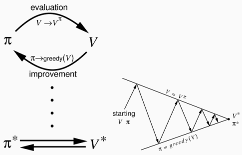
Figure 3: What's missing here ?

Generalized Policy Iteration (GPI)

Most solution approaches can be characterized by two simultaneous, interacting processes:

1. **Policy evaluation -** making the value function consistent with the current policy.
2. **Policy improvement -** making the policy greedy with respect to the current value function.

Generalized Policy Iteration - is the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the processes.



GPI: Policy Iteration and Value Iteration[1]

Policy Iteration

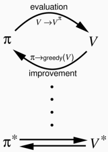
- Given a policy π, we use $\mathcal{V}_\pi$ to improve the policy, and yield $\pi^{'}$. We then compute $\mathcal{V}_{\pi'}$ to yield an even better policy, $\pi^{''}$, etc.
- We can thus obtain a sequence of monotonically improving policies and value functions.

Value Iteration

- Until convergence, iterativly update values based on the value of the best next-state.
- Value iteration is obtained simply by turning the Bellman optimality equation into an update rule.

[1] U and $\mathcal{V}$ used interchangeably

Policy Iteration



function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      $a^* \leftarrow \underset{a \in A(s)}{\text{argmax}}$ Q-VALUE(mdp, s, a, U)
      if Q-VALUE(mdp, s, a^*, U) $>$ Q-VALUE($mdp, s, \pi[s], U$) **then**
        $\pi[s] \leftarrow a^*$; $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

Value Iteration

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s')\right] \tag{1}$$

How can the Bellman optimality equation be used within a solution algorithm ?

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a) \left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s') \right] \tag{1}$$

How can the Bellman optimality equation be used within a solution algorithm ?

$$\mathcal{V}_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a) \left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}_k(s') \right]$$

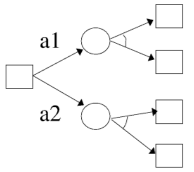
Value Iteration

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s, a, s')$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum relative change in the utility of any state

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow \max_{a \in A(s)}$ Q-VALUE(mdp, s, a, U)
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return U

Complexity?

Alternatives?



- **A*** finds a solution in graphs, and finds a sequence of actions (path) leading from the start state to a goal state.
- **AO*** finds a solution that has a conditional structure and takes the form of a **tree**.

AO*

- AO* is a heuristic search algorithm that can find optimal solutions for acyclic MDPs (e.g., finite-horizon MDPS) without evaluating the entire state space.
- Thus, it provides a useful alternative to dynamic programming algorithms for MDPs such as value iteration and policy iteration.
- Iterares over two steps:
  - Expansion of the current best partial policy
  - Updating the states' value according to the current policy envolope to propogate values from the newly expanded state.
- LAO* (Hansen et al 2001) is a variation of AO* that supports solutions with loops.
- The difference is in the update rule: while in an acyclic graph (and with AO*) updates are done with a single sweep of Bellman backups - LAO* applies VI or PI to update states (Kolobov 2012).

AO* (Nilsson 1980)

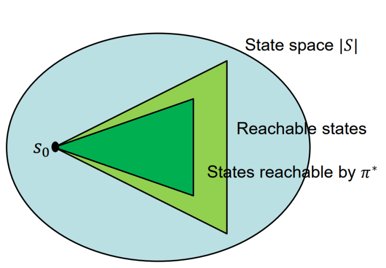


Image by Pascal Poupart 2013.

AO* (Nilsson 1980)

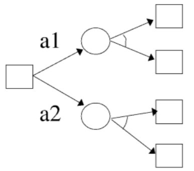
Algorithm 1 AO*

Initialize the graph with a root node n_0.
Set n_0 as the current node.
while goal states are not reached or graph needs refinement **do**
    Expand the most promising unexpanded node n.
    Evaluate n and update its heuristic value $h(n)$.
    if n is a leaf node **then**
        Mark it as solved if it satisfies the goal condition.
    else
        Add child nodes of n to the graph based on its AND/OR structure.
    end if
    Update heuristic values for all ancestors of n:
    for all nodes m in the path to n **do**
        if m is an OR-node **then**
            Update $h(m) \leftarrow \min_{c \in \text{children}(m)} \big(h(c) + \text{cost}(m, c)\big)$.
        else if m is an AND-node **then**
            Update $h(m) \leftarrow \sum_{c \in \text{children}(m)} \big(h(c) + \text{cost}(m, c)\big)$.
        end if
    end for
    Backtrack and mark paths that lead to solved nodes as part of the solution graph.
end while
Return the solution graph.

Heuristics for Stochastic Planning?

Heuristics for Stochastic Planning

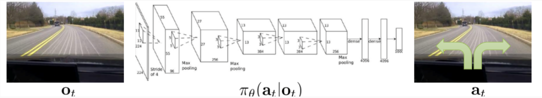
- Many ideas that are relevant to deterministic planning are also relevant here (e.g., delete relaxation)
- Some ideas are specific to stochastic domains
  - **All-outcome Determinization**
    - For each action in the stochastic environment, list all possible outcomes.
    - Each outcome corresponds to a deterministic transition



Planning With Partial Observability

- Can we use a **Markov Decision Process**(MDP) $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ to account for partially observable environments ?

Sometimes, yes.



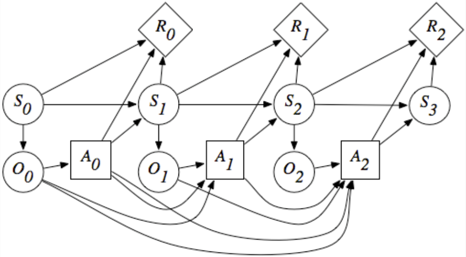
Sometimes, an MDP is not enough.

A **Partially Observable Markov Decision Process**(POMDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \Omega, \mathcal{O}, \beta_0 \rangle$ where

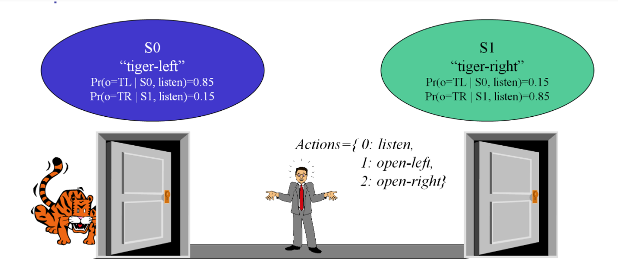
- $\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$ and γ are as for an MDP.
- Ω is a set of observations (observation tokens),
- $\mathcal{O}$ is a sensor function specifying the conditional observation probabilities $\mathcal{O}^o_{s,a} = \mathcal{P}[O_{t+1} = o | S_t = s, A_t = a]$ of receiving observation token $o \in \Omega$ in state s after applying action a [2].
- β_0 the initial belief: a probability distribution over the states such that $\beta_0(s)$ stands for the probability of s being the true initial state.

[2] alternatively: $\mathcal{O}^o_s = \mathcal{P}[o_t = o | S_t = s]$

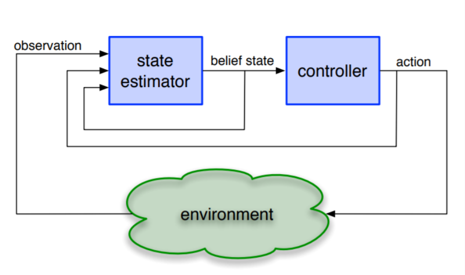
POMDP- graphical form



POMDP example



- A **belief** is a probability distribution over the possible world states such that $b(s)$ stands for the probability that s is the true world state.
- In partially observable domains, we may have a **sensor model / state estimator** represented as a mapping function from what is observed to the actual world state.

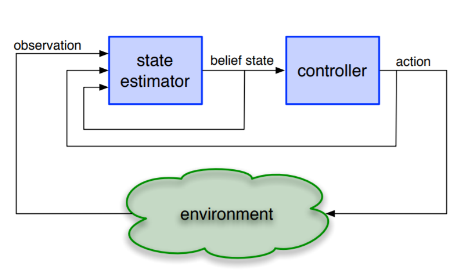


From Kaelbling, L. P., and T. Lozano-Perez. "Integrated Task and Motion Planning in Belief Space" 2013 *https://dspace.mit.edu/bitstream/handle/1721.1/87038/Kaelbling_Integrated%20task.pdf?sequence=1&isAllowed=y*

Planning in Belief Space

Two key challenges when planning in belief space:

- Belief tracking - what is the state of the world ?
- Policy computation - what is the best action to perform ?

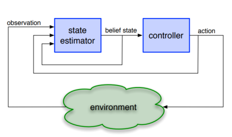


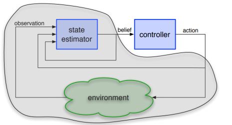
Pineau, Nicholas and Thrun. "A hierarchical approach to POMDP planning and execution." 2001. https://www.cs.mcgill.ca/ jpineau/files/jpineau-icml01.pdf

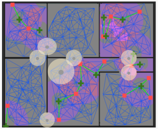
Planning in Belief Space: Solution Approaches

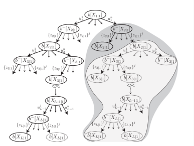
Combinations of different approaches:

- Planning in a **belief MDP**, an MDP with beliefs as states
- Sampling / discretization
- Approximations / relaxations









See work by Vadim Indelman from the Technion, e.g.,

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8793548

Belief Update

When receiving an observation o, the agent updates its belief current belief β using its **belief update function** $\tau : \mathcal{B} \times \Omega \times \mathcal{X} \mapsto \mathcal{B}$ which maps belief $\beta \in \mathcal{B}$ to the new belief.

Belief Update

When receiving an observation o, the agent updates its belief current belief β using its **belief update function** $\tau : \mathcal{B} \times \Omega \times \mathcal{X} \mapsto \mathcal{B}$ which maps belief $\beta \in \mathcal{B}$ to the new belief.

Commonly, a **Bayesian filter** is used:

$$\beta^{o,a} = \frac{\hat{P}(o|s,a)\,\beta(s)}{\int_{s' \in \mathcal{S}} \hat{P}(o|s',a)\,\beta(s')\,ds'} \tag{2}$$

where $\beta(s)$ is the estimated probability that s is the actual world state when the new observation o is emitted.
Discrete version:

$$\beta^{o,a} = \frac{\hat{P}(o|s,a)\,\beta(s)}{\Sigma_{p' \in \mathcal{P}} \hat{P}(o|s',a)\,\beta(s')} \tag{3}$$

Belief Update - Example

Planning with POMDPs

- A policy $\pi : \beta \mapsto \mathcal{A}$ of a POMDP maps the current belief into an action.
- The belief is assumed to be a sufficient statistic and an optimal policy is the solution of a continuous space "belief MDP"
- Some relevant links:
  - *https://people.csail.mit.edu/lpk/papers/aij98-pomdp.pdf*
  - *https://people.eecs.berkeley.edu/~pabbeel/cs287-fa13/slides/pomdps.pdf*
  - *https://cs.brown.edu/research/ai/pomdp/tutorial/pomdp-solving.html*
  - *https://www.youtube.com/watch?v=cTu7mvRE354*

Value iteration for POMDPs

Bellman optimality for MDPS:

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s')\right]$$

Bellman optimality for POMDPs:

Value iteration for POMDPs

Bellman optimality for MDPS:

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s')\right]$$

Bellman optimality for POMDPs:

$$\mathcal{V}^*(\beta) = \max_a \left[\mathcal{R}(\beta,a) + \gamma \sum_o \mathcal{P}(o|a,\beta)\mathcal{V}^*(\tau(\beta,a,o))\right]$$

Value iteration for POMDPs

Bellman optimality for MDPS:

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s')\right]$$

Bellman optimality for POMDPs:

$$\mathcal{V}^*(\beta) = \max_a \left[\mathcal{R}(\beta,a) + \gamma \sum_{o} \mathcal{P}(o|a,\beta)\mathcal{V}^*(\tau(\beta,a,o))\right]$$

Update formula for MDPS:

$$\mathcal{V}_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}_k(s')\right]$$

Value iteration for POMDPs

Bellman optimality for MDPS:

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s')\right]$$

Bellman optimality for POMDPs:

$$\mathcal{V}^*(\beta) = \max_a \left[\mathcal{R}(\beta,a) + \gamma \sum_{o} \mathcal{P}(o|a,\beta)\mathcal{V}^*(\tau(\beta,a,o))\right]$$

Update formula for MDPS:

$$\mathcal{V}_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a)\left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}_k(s')\right]$$

Update formula for POMDPs:

Value iteration for POMDPs

Bellman optimality for MDPS:

$$\mathcal{V}^*(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a) \left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}^*(s')\right]$$

Bellman optimality for POMDPs:

$$\mathcal{V}^*(\beta) = \max_a \left[\mathcal{R}(\beta,a) + \gamma \sum_{o} \mathcal{P}(o|a,\beta) \mathcal{V}^*(\tau(\beta,a,o))\right]$$

Update formula for MDPS:

$$\mathcal{V}_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}(s'|s,a) \left[\mathcal{R}(s,a,s') + \gamma \mathcal{V}_k(s')\right]$$

Update formula for POMDPs:

$$\mathcal{V}_{k+1}(\beta) = \max_a \left[\mathcal{R}(\beta,a) + \gamma \sum_{o \in \Omega} \mathcal{O}(o|a,\beta) \mathcal{V}_k(\tau(\beta,a,o))\right]$$

Value iteration for POMDPs

$$\mathcal{V}_{k+1}(\beta) = \max_a \left[\mathcal{R}(\beta, a) + \gamma \sum_{o \in \Omega} \mathcal{O}(o|a, \beta) \mathcal{V}_k(\tau(\beta, a, o)) \right]$$

Problems?

- Reward is not a function of the belief, but of the state
- While states are discrete - beliefs are continuous (so the space is ∞)

Value iteration for POMDPs

Reward is a function of the state:

$$\mathcal{V}_{k+1}(\beta) = \max_a \left[\mathcal{R}(\beta, a) + \gamma \sum_{o \in \Omega} \mathcal{O}(o|a, \beta) \mathcal{V}_k(\tau(\beta, a, o)) \right]$$

Reward is a function of the state:

$$\mathcal{V}_{k+1}(\beta) = \max_a \left[\mathcal{R}(\beta, a) + \gamma \sum_{o \in \Omega} \mathcal{O}(o|a, \beta) \mathcal{V}_k(\tau(\beta, a, o)) \right]$$

Beliefs are continuous:

Value iteration for POMDPs

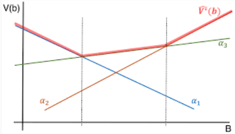
Reward is a function of the state:

$$\mathcal{V}_{k+1}(\beta) = \max_a \left[\mathcal{R}(\beta, a) + \gamma \sum_{o \in \Omega} \mathcal{O}(o|a, \beta) \mathcal{V}_k(\tau(\beta, a, o)) \right]$$

Beliefs are continuous:

Instead of considering beliefs, we consider a finite set of α-vectors that represent beliefs.

$$\mathcal{V}_k(\beta) = max_{\alpha \in \Gamma_k} \alpha \cdot \beta = max_{\alpha \in \Gamma_k} \Sigma_s \alpha(s) \cdot \beta(s)$$



```
function POMDP-VALUE-ITERATION(pomdp, ε) returns a utility function
   inputs: pomdp, a POMDP with states S, actions A(s), transition model P(s′ | s, a),
               sensor model P(e | s), rewards R(s), discount γ
            ε, the maximum error allowed in the utility of any state
   local variables: U, U′, sets of plans p with associated utility vectors α_p

   U′ ← a set containing just the empty plan [], with α_[](s) = R(s)
   repeat
      U ← U′
      U′ ← the set of all plans consisting of an action and, for each possible next percept,
          a plan in U with utility vectors
      U′ ← REMOVE-DOMINATED-PLANS(U′)
   until MAX-DIFFERENCE(U, U′) ≤ ε(1 − γ)/γ
   return U
```

- A high-level sketch of the value iteration algorithm for POMDPs.
- The REMOVE-DOMINATED-PLANS step and MAX-DIFFERENCE test are typically implemented as linear programs.

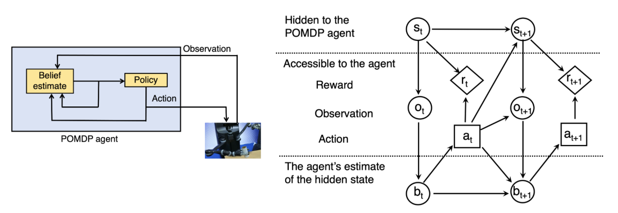
Planning with POMDPs

- SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. Kurniawati et al. 2008 *https://bigbird.comp.nus.edu.sg/m2ap/wordpress/wp-content/uploads/2016/01/rss08.pdf*
- Efficient point-based POMDP planning by approximating optimally reachable belief spaces: Kurniawati (2021): *https://arxiv.org/pdf/2107.07599.pdf*

Planning with POMDPs

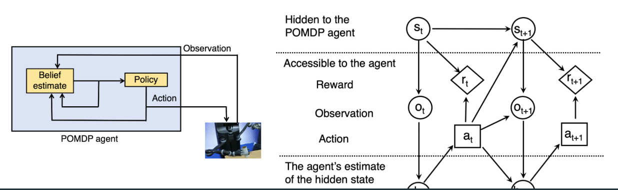
When a POMDP $\langle \mathcal{S}, \mathcal{A}, \mathbb{P}, \mathcal{R}, \gamma, \Omega, \mathcal{O}, \beta_0 \rangle$ is used to represent a robot's task

- the transition function is typically represented as a noisy dynamics function $s' = f(s, a, \eta)$, where $s, s' \in \mathcal{S}$ and $\eta \sim N$ is a noise vector sampled from noise distribution N, while $f(\cdot)$ denotes the system's dynamics.
- Similarly, $\mathcal{O}$ denotes the sensor/ observation function, representing errors and noise in measurement and perception.



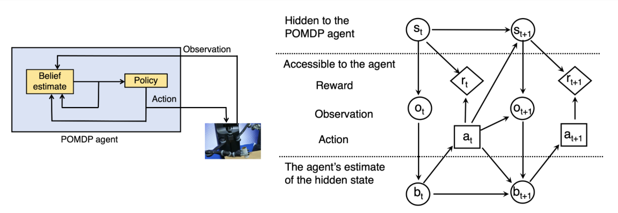
Planning with POMDPs

- POMDP is powerful in its quantification of the non-deterministic effects of actions and partial observability due to errors in sensor measurements and in perception
- The computed policy will balance information gathering and goal attainment.
- But precisely because of this, POMDP is notorious for its high computational complexity and deemed impractical for robotics.
- Until recently, most benchmark problems for POMDPs had less than 30 states and the best algorithms that could solve them took hours.



Planning with POMDPs

- In the past 2 decades, POMDPs solving capabilities have advanced tremendously, thanks to **sampling-based approximate solvers**.
- Although optimality is compromised, robustness and computational efficiency is improved: practical for many realistic robotics problems.



Planning with POMDPs

Algorithm 1 A typical program skeleton for sampling-based POMDP solvers

1: Initialize policy π and a set of sampled beliefs B
   {Generally, B is initialised to contain only a single belief (e.g., the initial belief b_0)}
2: **repeat**
3:   Sample a (set of) beliefs {Some methods sample histories (a history is a sequence of action–observation tuples) rather than beliefs. In POMDPs, beliefs provide sufficient statistics of the entire history [25], and therefore the two provide equivalent information}
4:   Estimate the values of the sampled beliefs
     {Generally, via a combination of heuristics and update / backup operation}
5:   Update π {In most methods, this step is a byproduct of the previous step}
6: **until** Stopping criteria is satisfied

- Key idea: sample a set of representative beliefs and computes optimal policy only for them, thus substantially reducing complexity.
- Which set would be sufficiently representative ?
  - A variety of sampling strategies have been proposed to select the sample set and to estimate the values of the sampled beliefs.
  - Most sampling-based approximate POMDP solvers are **anytime**
  - Some methods compute upper and lower bound estimates of the value functions
  - Can be broadly divided into offline and online.

SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. Kurniawati et al. 2008 *https://bigbird.comp.nus.edu.sg/m2ap/wordpress/wp-content/uploads/2016/01/rss08.pdf*

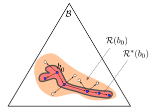


Fig. 1. Belief space $\mathcal{B}$, reachable space $\mathcal{R}(b_0)$, and optimally reachable space $\mathcal{R}^*(b_0)$. Note that $\mathcal{R}^*(b_0) \subseteq \mathcal{R}(b_0) \subseteq \mathcal{B}$.

- Some early POMDP algorithms sample the entire belief space B, using a uniform sampling distribution, such as a grid.
- More recent point-based algorithms sample only $\mathcal{R}(\beta_0)$, the subset of belief points reachable from a given initial point $\beta_0 \in \mathcal{B}$ under arbitrary sequences of actions.

SARSOP

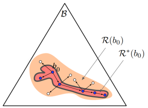


Fig. 1. Belief space $\mathcal{B}$, reachable space $\mathcal{R}(b_0)$, and optimally reachable space $\mathcal{R}^*(b_0)$. Note that $\mathcal{R}^*(b_0) \subseteq \mathcal{R}(b_0) \subseteq \mathcal{B}$.

- SASOP pushes this direction further, by sampling near $\mathcal{R}^*(\beta_0)$, a subset of belief points reachable from β_0 under we**optimal sequences of actions** ($\mathcal{R}^*(\beta_0)$ is usually much smaller than $\mathcal{R}(\beta_0)$).
- Optimality not achievalbe, so approximations of $\mathcal{R}^*(\beta_0)$ are used.
    - Use successive approximations of $\mathcal{R}^*(\beta_0)$ and converge to it iteratively.
    - The algorithm relies on heuristic exploration to sample $\mathcal{R}(\beta_0)$ and improves sampling over time through a simple on-line learning technique.
    - Bounding technique are used to avoid sampling in regions that are unlikely to be optimal
    - This leads to substantial gain in computational efficiency

What if we don't have full access to the model of the environment ?

Recap and what next

- Spectrum of approaches to planning with
  - deterministic and stochastic actions
  - full and partial observability

Do we still need to know about planning?

by Subbarao Kambhampati:
"Human, grant me the serenity to accept the things I cannot learn, the data to learn the things I can, and the wisdom to know the difference."[3]

[3] My addition: and the ability to know what to do about it