Sequential Decision Making and Reinforcement Learning

(SDMRL)

Supervised Learning vs. Model Bases Planning for Long-Term Decision-Making

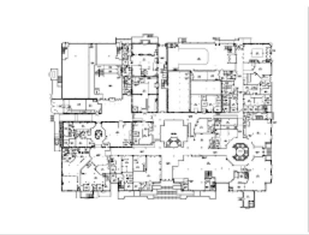
Sarah Keren

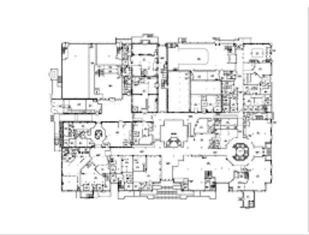
The Taub Faculty of Computer Science
Technion - Israel Institute of Technology

Agenda

- Model-based / model-free spectrum
- Supervised Learning for long-term decision-making
- Model-based planning for long-term decision-making

Model-free vs. Model-based Decision-making





How to come up with 'good' policies?

How to come up with 'good' policies?

How to come up with 'good' policies?
If we want to avoid collision?

How to come up with 'good' policies?
If we want to avoid collision?
If we want to turn left at the next intersection?

How to come up with 'good' policies?
If we want to avoid collision?
If we want to turn left at the next intersection?
If we want to buy apples and make it on time for dinner but have fuel for about 30 mins?

An Agent

An agent typically maintains one or more of these components:

- **Model:** agent's representation of the environment
- **Value function:** how good is each state and/or action
- **Policy:** agent's behaviour function
  - **Deterministic Policy**: a mapping $\pi : \Omega \rightarrow \mathcal{A}$ from states/observations to actions.
  - **Stochastic Policy**: a mapping $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ from state and action pairs to to the probability $\pi(a|s)$ of taking action a when in state s.

Return

- Expected return for episodic tasks:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T = \sum_{k=0}^{T} R_{t+k+1}$$

- Expected return for continuing tasks:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

when γ is the discount factor.

- Returns at successive time steps are related to each other:

$$G_t = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \ldots) = R_{t+1} + \gamma G_{t+1}$$

Value functions estimate how 'good' it is for the agent to be in a given state or how good it is to perform a given action in a given state.

"How good" is defined in terms of expected return.

Since the rewards the agent can expect to receive in the future depend on what actions it will take, value functions are defined with respect to particular policies.

State-Value Function for Policy π:

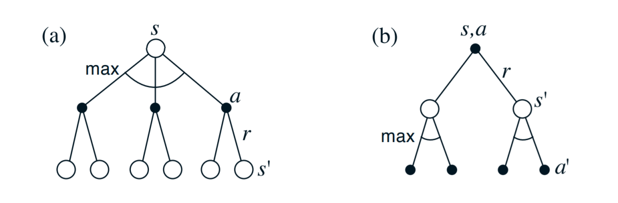
$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

Action-Value Function for Policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]$$

What is G_t ?

Value Functions



Backup diagrams for $v_\pi(s)$ (left) and $q_\pi(s,a)$ (right).

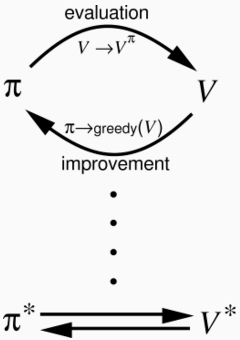
Since it may not be practical to keep separate averages for each state individually v_π and q_π are often represented as parameterized functions (with fewer parameters than states).

How to find an optimal policy ?

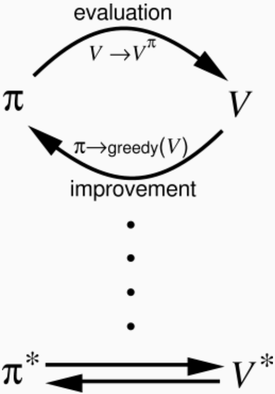
Recipes for Control Approaches

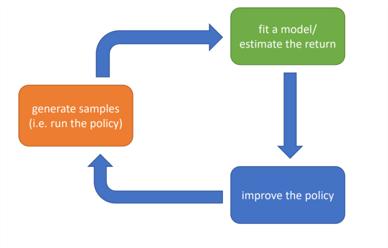
Policy Evaluation and Policy Update

- Prediction / Evaluation: evaluate the future given a policy
- Control/ Update / Improvement: optimize the policy.



Control Approaches Skeleton





Left image by Sutton and Barto. Right image By Sergey Levine

Model Based vs. Model Free

- For this distinction, a **model** typically refers to the transition function $\mathcal{P}$ and the reward function $\mathcal{R}$.
- A model free approach only maintains a policy or value function, but no model.
- A model-based approach maintains a policy or value function and a model.

https://www.davidsilver.uk/wp-content/uploads/2020/03/intro_RL.pdf

Value vs. Policy Based

- **Value Based**:
  - No Policy (Implicit)
  - Value Function
- **Policy Based**
  - Policy
  - No Value Function
- **Combined approach (a.k.a Actor-Critic)**
  - Policy
  - Value Function

On Policy vs. Off Policy

According to Sutton and Barto 2018

- **On-policy** methods evaluate and improve the policy based on the policy that is used to make decisions.
- **Off-policy** methods evaluate or improve a policy different from that used to generate the data.
- We distinguish between the **behavior policy**, according to which an agent interacts with the environment and **target policy** the policy the agent is trying to learn that will optimize its utility.
- In on-policy methods behavior policy == target policy. In off-policy methods they are different.

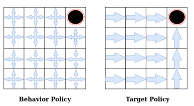
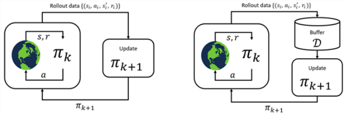


Image from *https://towardsdatascience.com/*

- **Online** iteratively collecting data (a.k.a. experiences) by interacting with the environment.
- **Offline** utilize previously collected data, without additional online data collection.
  - resembles the standard supervised learning.
  - make it possible to turn large datasets into powerful decision making engines.
  - Batch Reinforcement Learning, behavioral cloning



From Or Rivlin *https://towardsdatascience.com/the-power-of-offline-reinforcement-learning-5e3d3942421c*

Planning and Replanning

- **Planning**: the task of coming up with a sequence of actions that will achieve a goal.
  - **Sensorless (conformant) planning:** constructing sequential plans to be executed without perception
  - **Conditional (contingent) planning**: constructing a conditional plan with different branches for different contingencies that could happen.
  - **Continuous planning:** a planner designed to persist over a lifetime.
- **Execution monitoring and replanning**: constructing a plan, but monitoring its execution and generating a new plan when necessary.

from *https://www.cpp.edu/~ftang/courses/CS420/notes/planning.pdf*

Solution Approaches

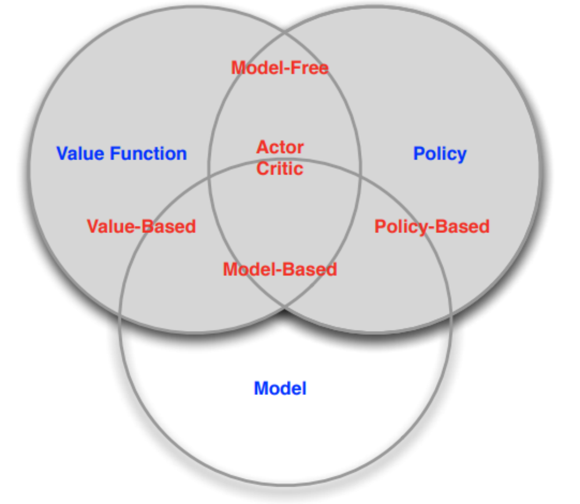


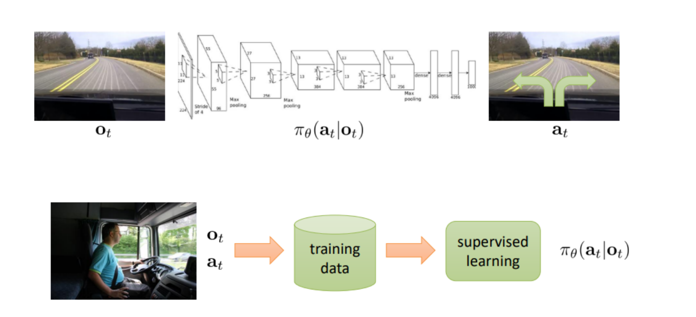
image by David Silver

Approaches to Control

- Supervised learning
- Model-Based Planning
- Monte-Carlo methods
- Temporal-Difference methods
- Combined approaches

Decision-making as Supervised Learning

Behavior Cloning: Example



Behavioral Cloning

- The objective is to learn a policy by mimicking the behavior demonstrated by an expert.
- A model (policy) $\pi_\theta(s)$ is trained to minimize the discrepancy between its predicted actions and the expert's actions over the collected dataset.

Behavioral cloning

Can be broken down into the following steps:

- **Data Collection:** Gather a dataset of state-action pairs (s_i, a_i), where s_i represents the state and a_i represents the action taken by the expert at that state.
- **Model Selection:** Choose a model to approximate the expert's policy. Typically, a neural network $\pi_\theta(s)$ parameterized by θ.

Ideas for managing the learning process?

- **Loss Function:** measures the difference between the expert's actions and the actions predicted by the model. For example, mean squared error (MSE) or cross-entropy loss for discrete actions.

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|a_i - \pi_\theta(s_i)\|^2$$

where N is the number of state-action pairs in the dataset.

- **Training:** Optimize the model parameters θ by minimizing the loss function (e.g., using gradient descent)

$$\theta^* = \arg\min_\theta L(\theta)$$

- **Policy Execution:** Use the trained model $\pi_{\theta^*}(s)$ as the policy to predict actions for new states during execution.

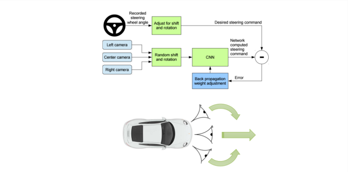
Limitations?

DAgger (Dataset Aggregation)

1. train $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ from human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{a}_1, \ldots, \mathbf{o}_N, \mathbf{a}_N\}$
2. run $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, \ldots, \mathbf{o}_M\}$
3. Ask human to label $\mathcal{D}_\pi$ with actions $\mathbf{a}_t$
4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$

https://jonathan-hui.medium.com/rl-imitation-learning-ac28116c02fc
https://bair.berkeley.edu/blog/2017/10/26/dart/

Applications



Bojarski et al. '16, NVIDIA

Model-based Planning

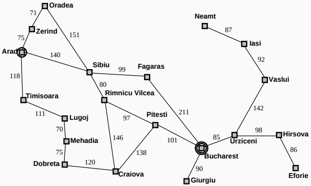
Problem-solving agents

Restricted form of general agent

def Simple-Problem-Solving-Agent (*problem*):
  state, some description of the current world state
  seq, an action sequence, initially empty
 state ← UPDATE-STATE(*state*, *percept*)
 if *seq* is empty **then**
  seq ← SEARCHFORSOLUTION(*problem*)
 action ← SELECTACTION(*seq*, *state*)
 seq ← REMAINDER(*seq*, *action*)
 return *action*

- Offline problem solving; solution executed “eyes closed.”
- Based on a model of the environment and its dynamics

Examples

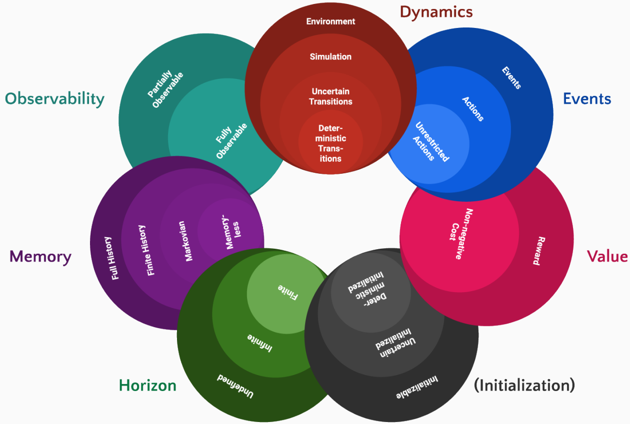

Oradea
71
Zerind
151
75
Arad
140
118
Timisoara
111
Lugoj
70
Mehadia
75
Dobreta
120
Craiova
Sibiu
99
Fagaras
80
Rimnicu Vilcea
97
Pitesti
146
138
211
101
Bucharest
90
Giurgiu
85
Urziceni
98
Hirsova
86
Eforie
142
Vaslui
92
Iasi
87
Neamt


traffic sign
recognition
surround view ON
STOP
STOP

Characterized by:

- A set of **states** $\mathcal{S}$ a system can be in.
  - a state is a full assignment to the set of **variables** (features) $\mathcal{X}$.
- **Actions** $\mathcal{A}$ change the values of certain variables.
- **Reward Function** $\mathcal{R}$ sets a numeric signal passed from the environment (can represent cost)
  - used to signal the objective
  - some domains have **goal conditions** such that the agent should reach **goal states** that satisfy is(e.g., 'be at Austin').
- **Objective:** find a **policy** that drives the initial state into a goal state or that maximizes the expected accumulated reward.
- Language is **generic** and not domain specific.
- **Complexity:** Even in the simplest setting it is NP-hard; i.e., exponential in the number of variables in the worst case.

Many Forms of Planning



https://github.com/fteicht/icaps24-skdecide-tutorial/tree/main

Planning for Deterministic Domains

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.

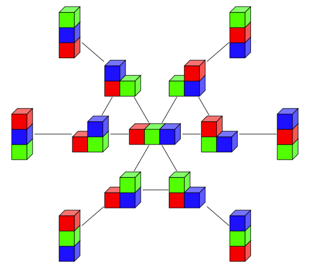
Definition (deterministic transition system)

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	1356437369358855817 3



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

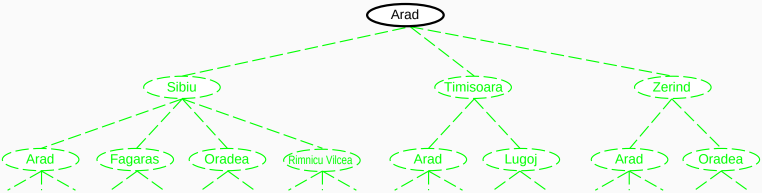
- State-space search: one of the big success stories of AI
- Must carefully distinguish two different problems:
  - satisficing planning: any solution is OK
    (although shorter solutions typically preferred)
  - optimal planning: plans must have shortest possible length
- Both are often solved by search, 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:

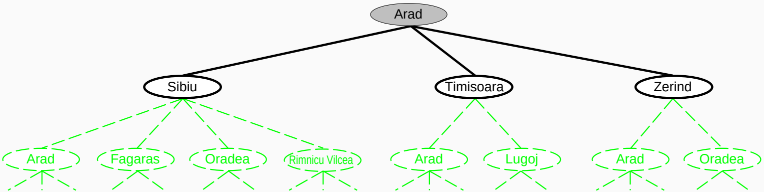
- start from initial state
- iteratively pick a previously generated state and progress it through an action, generating a new state
- solution found when a goal state generated



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

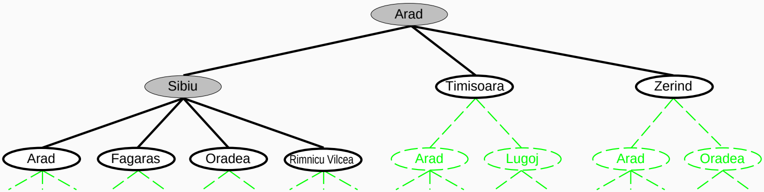


Planning by forward search: progression

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)

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?

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 ∞)

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
- ...

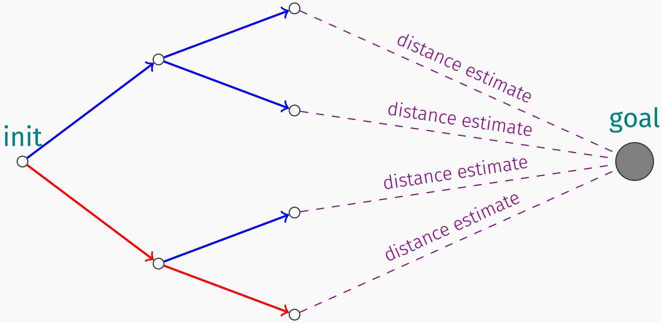
Heuristic search algorithms: local

- 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?

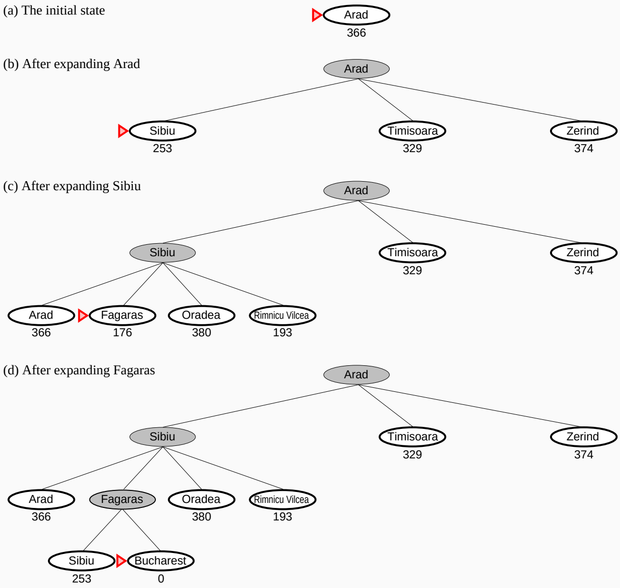
Greedy best-first search (with duplicate detection)

open := **new** min-heap ordered by $(\sigma \mapsto h(\sigma))$
open.insert(make-root-node(init()))
closed := $\emptyset$
while not *open*.empty():
    σ = *open*.pop-min()
    if *state*(σ) $\notin$ *closed*:
        closed := *closed* $\cup$ {*state*(σ)}
        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

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)

A*

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 estimated the expected distance from goal

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

Key points:

- As long as the heuristic is admissible then A* 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)

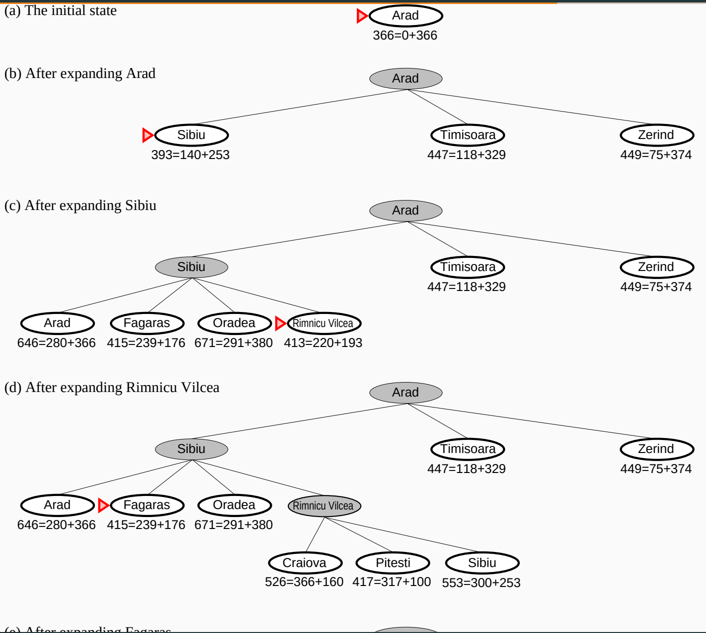
open := **new** min-heap ordered by $(\sigma \mapsto f(\sigma) = g(\sigma) + h(\sigma))$
open.insert(make-root-node(init()))
closed := $\emptyset$
distance := $\emptyset$
while not *open*.empty():
    σ = *open*.pop-min()
    if *state*(σ) $\notin$ *closed* **or** $g(\sigma) <$ *distance*(*state*(σ)):
        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*(one node maintained per state)

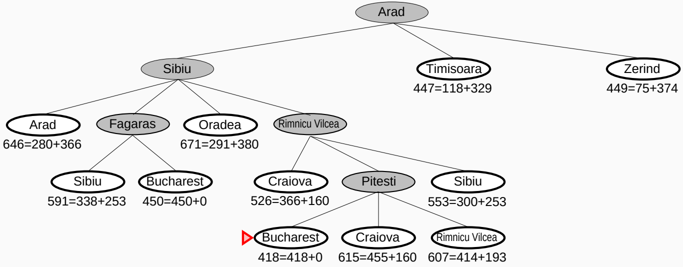
A* (with duplicate detection and reopening)

```
open := new min-set-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* 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

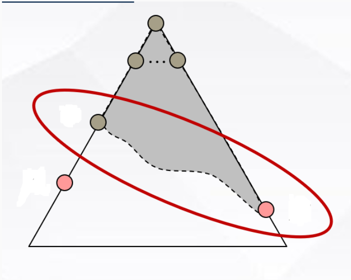
A* by example


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*(σ) $\in G$.

- 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 (σ ↦ g(σ) + W · 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(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
```

Properties of weighted A*

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:

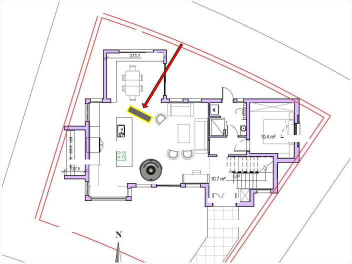
- one of the three most commonly used algorithms for satisficing planning
- for $W > 1$, can prove similar properties to A*, replacing optimal with bounded suboptimal: generated solutions are at most a factor W as long as optimal ones

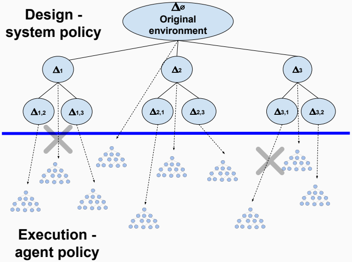
Algorithm 1 Best First Search (BFS)

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





Best First Design

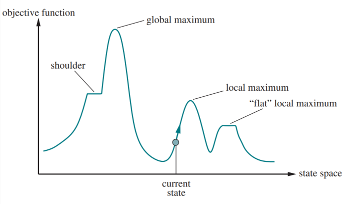
Algorithm 2 Best First Design (BFD)

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
  $n_{cur} = ExtractMax(\text{OPEN})$
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

$\sigma :=$ make-root-node(init())
forever:
  if is-goal(state(σ)):
    return extract-solution(σ)
  $\Sigma' :=$ { make-node(σ, o, s) | $\langle o, s \rangle \in$ succ(state(σ)) }
  $\sigma :=$ 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

Enforced hill-climbing: procedure improve

def *improve*(σ_0):
    queue := **new** fifo-queue
    queue.push-back(σ_0)
    closed $:= \emptyset$
    while not *queue*.empty():
        $\sigma =$ *queue*.pop-front()
        if *state*(σ) $\notin$ *closed*:
            closed := *closed* $\cup$ {*state*(σ)}
            if $h(\sigma) < h(\sigma_0)$:
                return σ
            for each $\langle o, s\rangle \in$ succ(*state*(σ)):
                $\sigma' :=$ 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:

- 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

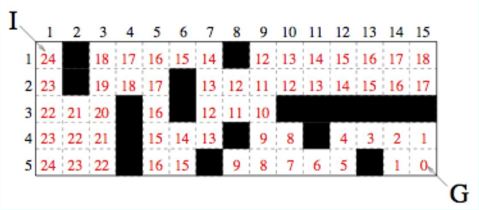
(Admissible) heuristic functions obtained as (optimal) cost functions of relaxed problems

Examples

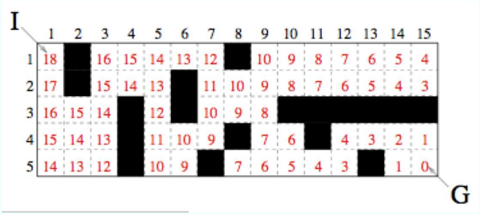
- Euclidian distance in Path Finding
- Manhattan distance in N-puzzle
- Spanning Tree in Traveling Salesman Problem
- Shortest Path in Job Shop Scheduling

Example

True distance h^* for different search states

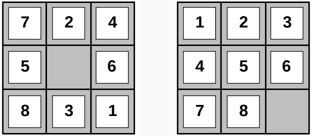


Manhattan distance is based on the relaxation that ...?



Example

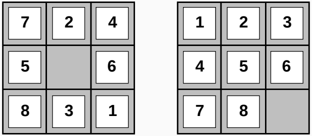
Example



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 can be used for heuristic estimations.

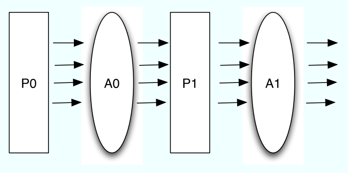
Different possibilities:

- 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 $G \subseteq P_i$

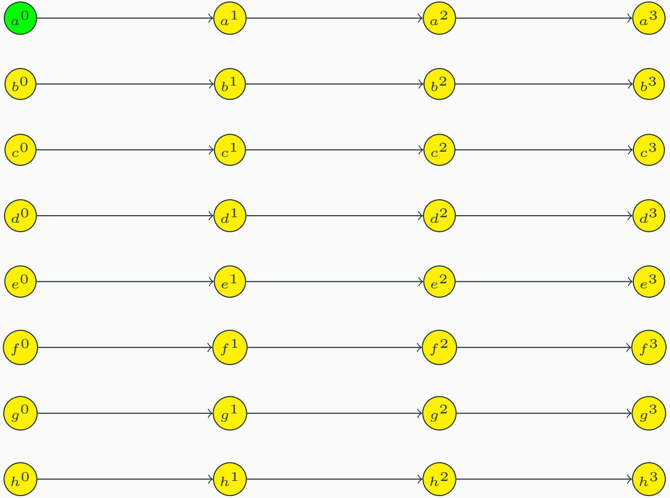
Running example

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

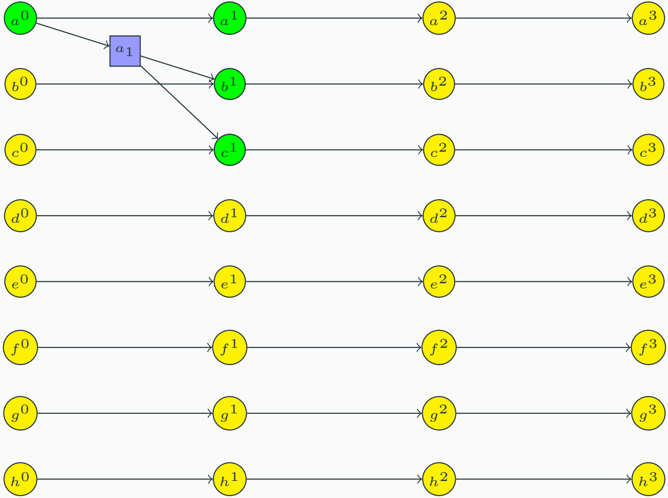
Running example

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

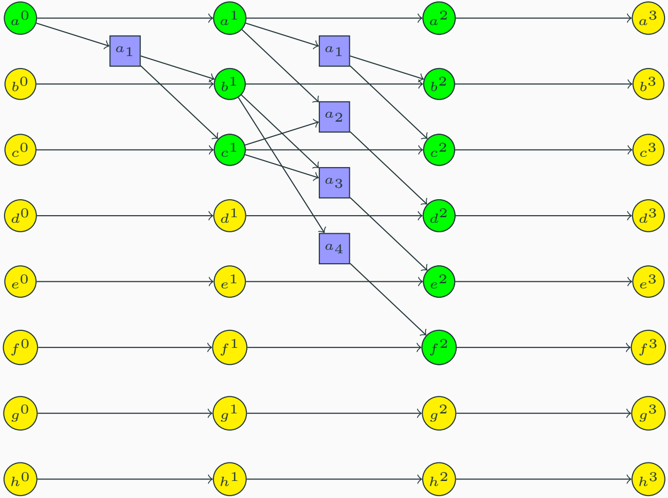
Running example: Relaxed planning graph



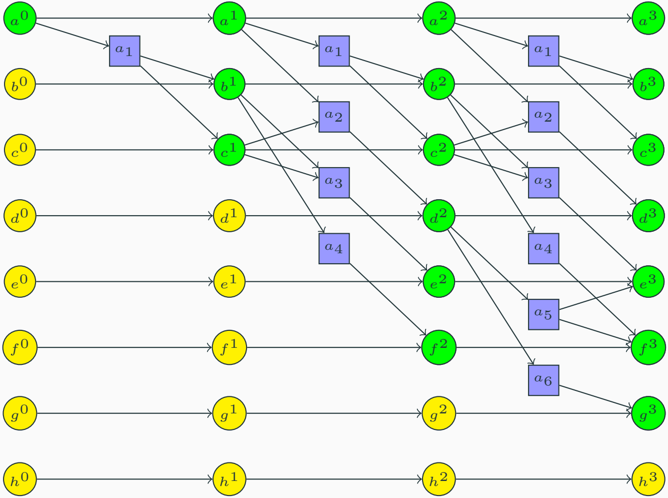
Running example: Relaxed planning graph



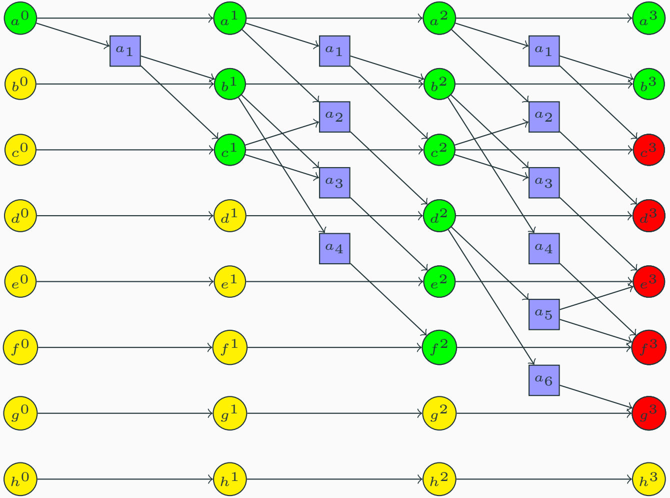
Running example: Relaxed planning graph



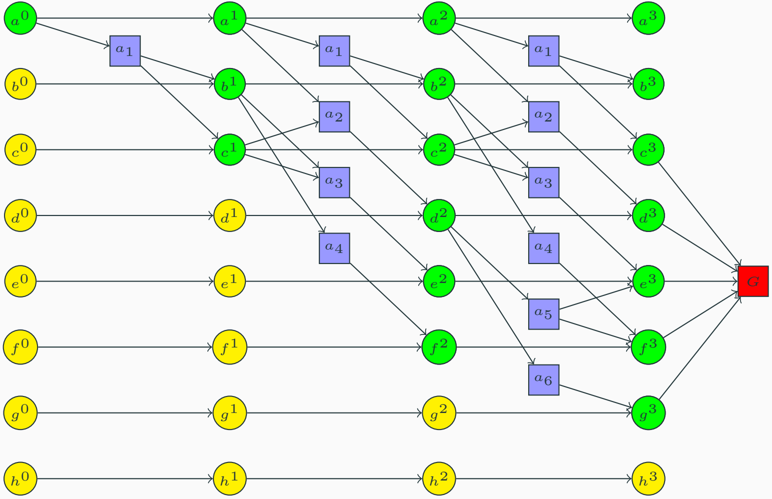
Running example: Relaxed planning graph



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

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

Recap

- Spectrum of approaches to control
- Various characteristics (model-free model-based, offlineonline, etc)
- Policy extraction as supervised learning
- Model-based planning in deterministic and full observable domains