Automated (AI) Planning

Planning tasks & Search

Carmel Domshlak

State-space search

- state-space search: one of the big success stories of AI
- many planning algorithms based on state-space search (we'll see some other algorithms later, though)
- will be the focus of this and the following topics
- we assume prior knowledge of basic search algorithms
  - uninformed vs. informed
  - systematic vs. local
- background on search: Russell & Norvig, Artificial Intelligence – A Modern Approach, chapters 3 and 4

Satisficing or optimal planning?

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 state-space search

How to apply search to planning? $\rightsquigarrow$ many choices to make!

Choice 1: Search direction

- progression: forward from initial state to goal
- regression: backward from goal states to initial state
- bidirectional search

Planning by state-space search

How to apply search to planning? ⇝ many choices to make!

Choice 2: Search space representation

- search nodes are associated with states
- search nodes are associated with sets of states

Planning by state-space search

How to apply search to planning? ⇝ many choices to make!

Choice 3: Search algorithm

- uninformed search:
  depth-first, breadth-first, iterative depth-first, . . .
- heuristic search (systematic):
  greedy best-first, A^*, Weighted A^*, IDA^*, . . .
- heuristic search (local):
  hill-climbing, simulated annealing, beam search, . . .

Planning by state-space search

How to apply search to planning? ⇝ many choices to make!

Choice 4: Search control

- heuristics for informed search algorithms
- pruning techniques: invariants, symmetry elimination, helpful actions pruning, . . .

Search-based satisficing planners

FF (Hoffmann & Nebel, 2001)

- search direction: forward search
- search space representation: single states
- search algorithm: enforced hill-climbing (informed local)
- heuristic: FF heuristic (inadmissible)
- pruning technique: helpful actions (incomplete)

⇝ one of the best satisficing planners

Search-based optimal planners

Fast Downward + h^{HHH} (Helmert, Haslum & Hoffmann, 2007)

- search direction: forward search
- search space representation: single states
- search algorithm: A* (informed systematic)
- heuristic: merge-and-shrink abstractions (admissible)
- pruning technique: none

$\rightsquigarrow$ one of the best optimal planners

Our plan for the next lectures

Choices to make:

1. search direction: progression/regression/both
   ⇝ this chapter
2. search space representation: states/sets of states
   ⇝ this chapter
3. search algorithm: uninformed/heuristic; systematic/local
   ⇝ this chapter
4. search control: heuristics, pruning techniques
   ⇝ following chapters

Planning by forward search: progression

Progression: Computing the successor state $\mathit{app}_o(s)$ of a state s with respect to an operator o.

Progression planners find solutions by forward search:

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

pro: very easy and efficient to implement

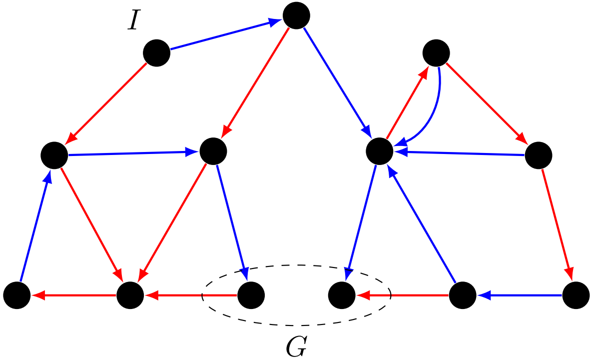
Search space representation in progression planners

Two alternative search spaces for progression planners:

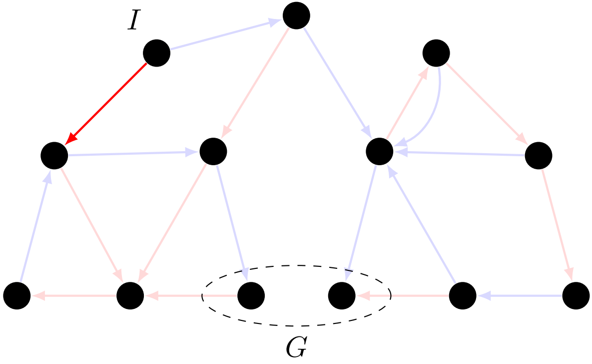
1. search nodes correspond to states
    - when the same state is generated along different paths, it is not considered again (duplicate detection)
    - pro: fast
    - con: memory intensive (must maintain closed list)
2. search nodes correspond to operator sequences
    - different operator sequences may lead to identical states (transpositions)
    - pro: can be very memory-efficient
    - con: much wasted work (often exponentially slower)

⇝ first alternative usually preferable

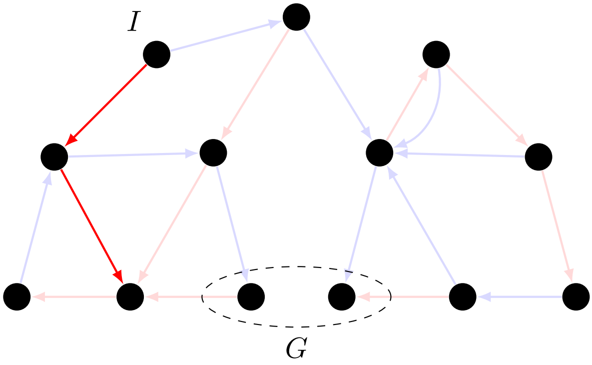
Progression planning example (depth-first search)



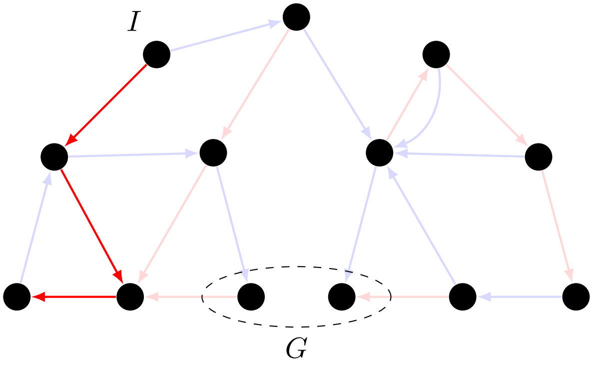
Progression planning example (depth-first search)



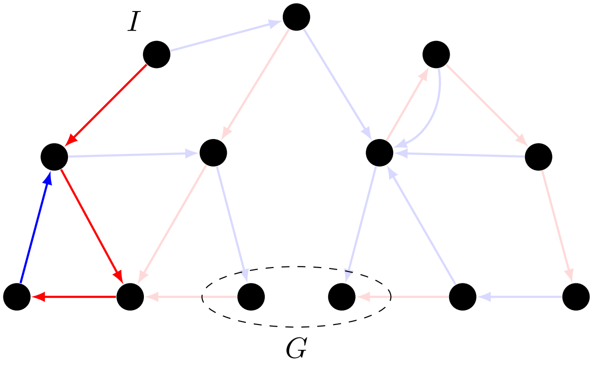
Progression planning example (depth-first search)



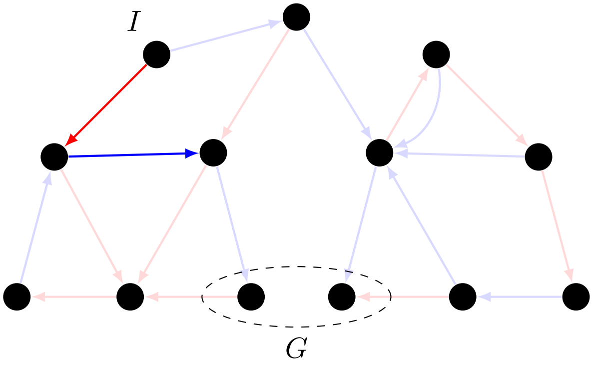
Progression planning example (depth-first search)



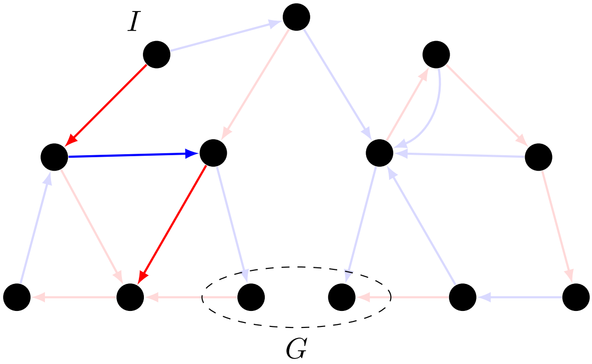
Progression planning example (depth-first search)



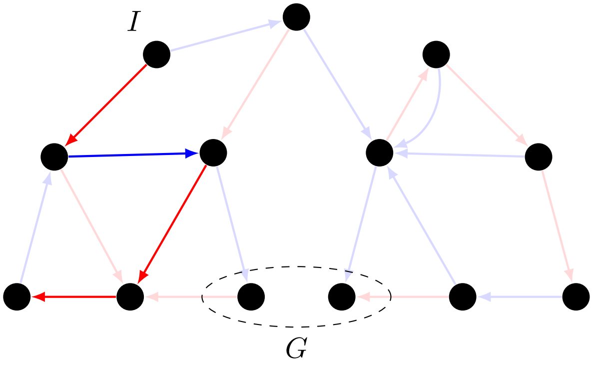
Progression planning example (depth-first search)



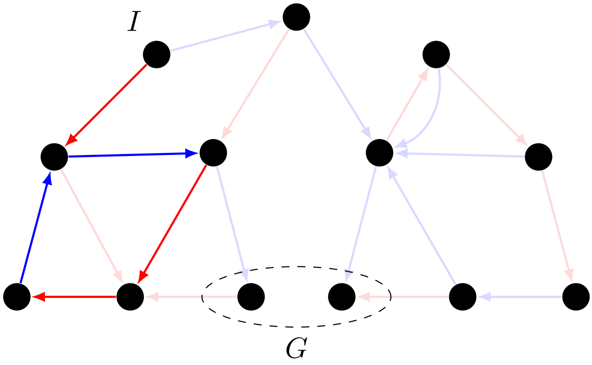
Progression planning example (depth-first search)



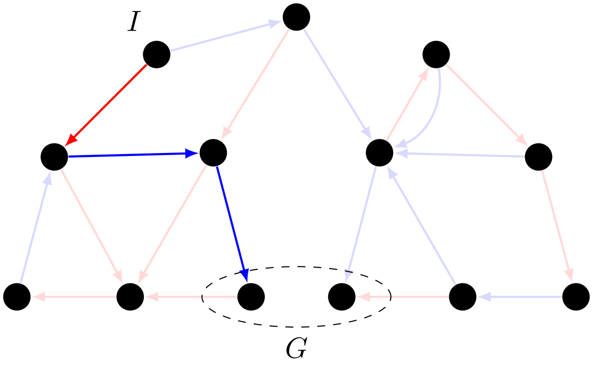
Progression planning example (depth-first search)



Progression planning example (depth-first search)



Progression planning example (depth-first search)



Forward search vs. backward search

Going through a transition graph in forward and backward directions is not symmetric:

- forward search starts from a single initial state;
  backward search starts from a set of goal states
- when applying an operator o in a state s in forward direction, there is a unique successor state s';
  if we applied operator o to end up in state s',
  there can be several possible predecessor states s

$\rightsquigarrow$ most natural representation for backward search in planning associates sets of states with search nodes

Planning by backward search: regression

Regression: Computing the possible predecessor states $\mathit{regr}_o(S)$ of a set of states S with respect to the last operator o that was applied.

Regression planners find solutions by backward search:

- start from set of goal states
- iteratively pick a previously generated state set and regress it through an operator, generating a new state set
- solution found when a generated state set includes the initial state

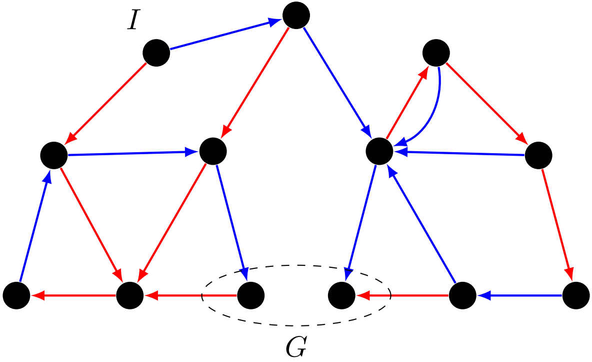
Pro: can handle many states simultaneously
Con: basic operations complicated and expensive

Search space representation in regression planners

identify state sets with logical formulae:

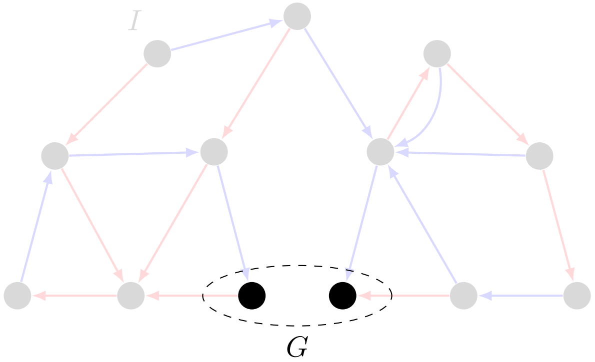
- search nodes correspond to state sets
- each state set is represented by a logical formula: ϕ represents $\{s \in S \mid s \models \phi\}$
- many basic search operations like detecting duplicates are NP-hard or coNP-hard

Regression planning example (depth-first search)



Regression planning example (depth-first search)

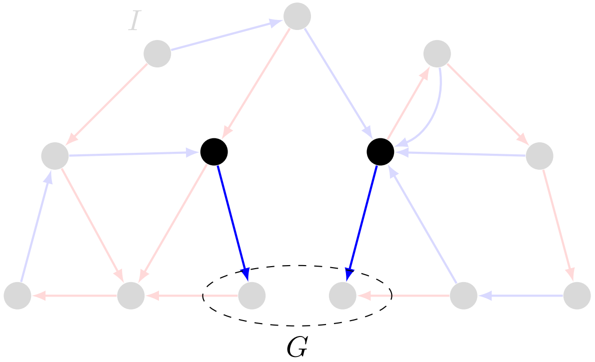
G



Regression planning example (depth-first search)

$\phi_1 = \textit{regr}_{\color{blue}\longrightarrow}(G)$

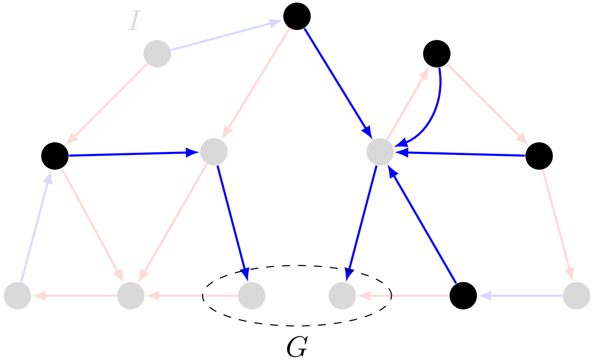
$\phi_1 \xrightarrow{\quad} G$



Regression planning example (depth-first search)

$$\phi_1 = \mathit{regr}_{\longrightarrow}(G)$$
$$\phi_2 = \mathit{regr}_{\longrightarrow}(\phi_1)$$

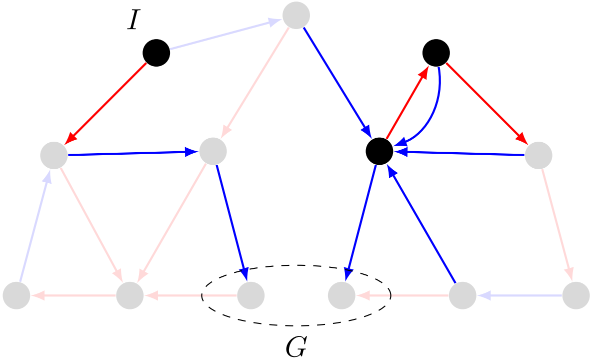
$$\phi_2 \longrightarrow \phi_1 \longrightarrow G$$



Regression planning example (depth-first search)

$$\phi_1 = \mathit{regr}_{\longrightarrow}(G)$$
$$\phi_2 = \mathit{regr}_{\longrightarrow}(\phi_1)$$
$$\phi_3 = \mathit{regr}_{\longrightarrow}(\phi_2), I \models \phi_3$$

$$\begin{tikzcd}\phi_3 \arrow[r] & \phi_2 \arrow[r] & \phi_1 \arrow[r] & G\end{tikzcd}$$



Regression for STRIPS planning tasks

Definition (STRIPS planning task)

A planning task is a STRIPS planning task if all operators are STRIPS operators and the goal is a conjunction of literals.

Regression for STRIPS planning tasks is very simple:

- Goals are conjunctions of literals $l_1 \wedge \cdots \wedge l_n$.
- First step: Choose an operator that makes some of $l_1, \ldots, l_n$ true and makes none of them false.
- Second step: Remove goal literals achieved by the operator and add its preconditions.
- $\rightsquigarrow$ Outcome of regression is again conjunction of literals.

Choices to make:

1. search direction: progression/regression/both
   ⇝ above
2. search space representation: states/sets of states
   ⇝ above
3. search algorithm: uninformed/heuristic; systematic/local
   ⇝ this chapter
4. search control: heuristics, pruning techniques
   ⇝ next chapters

Search

- Search algorithms are used to find solutions (plans) for transition systems in general, not just for planning tasks.
- Planning is one application of search among many.

Required ingredients for search

A general search algorithm can be applied to any transition system for which we can define the following three operations:

- init$()$: generate the initial state
- is-goal(s): test if a given state is a goal state
- succ(s): generate the set of successor states of state s, along with the operators through which they are reached (represented as pairs $\langle o, s' \rangle$ of operators and states)

Together, these three functions form a search space (a very similar notion to a transition system).

Search for planning: progression

Let $\Pi = \langle V, I, O, G\rangle$ be a planning task.

Search space for progression search

states: all states of Π (assignments to V)

- init() $= I$
- succ(s) $= \{\langle o, s'\rangle \mid o \in O, s' = \mathit{app}_o(s)\}$
- is-goal(s) $= \begin{cases} \textbf{true} & \text{if } s \models G \\ \textbf{false} & \text{otherwise} \end{cases}$

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)

Classification: what works where in planning?

uninformed vs. heuristic search:

- For satisficing planning, heuristic search vastly outperforms uninformed algorithms on most domains.
- For optimal planning, the difference is less pronounced. An efficiently implemented uninformed algorithm is not easy to beat in most domains. (But doable! We'll see that later.)

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.

Uninformed search algorithms

Less relevant for planning, yet not irrelevant

Popular uninformed systematic search algorithms:

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

Popular uninformed local search algorithms:

- random walk

Heuristic search algorithms: systematic

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

Popular systematic heuristic search algorithms:

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

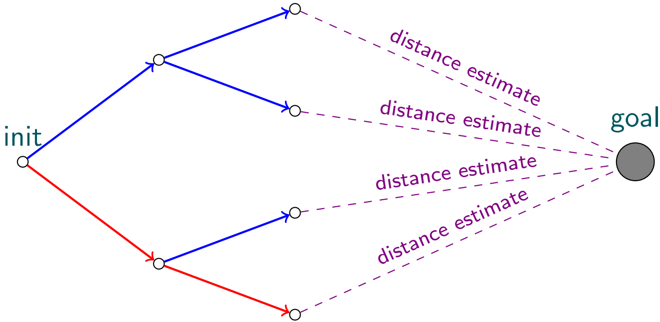
Heuristic search algorithms: local

- Heuristic search algorithms are the most common and overall most successful algorithms for classical 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 $h(\sigma)$ is called the heuristic estimate or heuristic value of heuristic h for node σ. It is supposed to estimate the distance from σ to the nearest goal node.

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 ($=$ be correct and complete).
- 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, $\dots$)
- For other search algorithms, "it works well in practice" is often as good an analysis as one gets.

Heuristics applied to nodes or states?

- Most texts apply heuristic functions to states, not nodes.
- This is slightly less general than our definition:
  - Given a state heuristic h, we can define an equivalent node heuristic as $h'(\sigma) := h(\mathit{state}(\sigma))$.
  - The opposite is not possible. (Why not?)
- There is good justification for only allowing state-defined heuristics: why should the estimated distance to the goal depend on how we ended up in a given state s?
- We call heuristics which don't just depend on $\mathit{state}(\sigma)$ pseudo-heuristics.
- In practice there are sometimes good reasons to have the heuristic value depend on the generating path of σ

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 $h^*(\sigma) = \infty$ for all $\sigma \in \Sigma$ with $h(\sigma) = \infty$
- 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 $h(\sigma) \leq h(\sigma') + 1$ for all nodes $\sigma, \sigma' \in \Sigma$ such that σ' is a successor of σ

Relationships?

Greedy best-first search

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():
    $\sigma = open$.pop-min()
    if $state(\sigma) \notin closed$:
        $closed := closed \cup \{state(\sigma)\}$
        if is-goal(state($\sigma$)):
            return extract-solution($\sigma$)
        for each $\langle o, s \rangle \in$ succ($state(\sigma)$):
            $\sigma' :=$ make-node($\sigma, o, s$)
            if $h(\sigma') < \infty$:
                $open$.insert($\sigma'$)
return unsolvable
```

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)

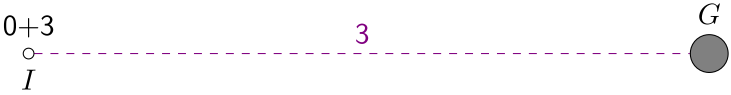
A*

A* (with duplicate detection and reopening)

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

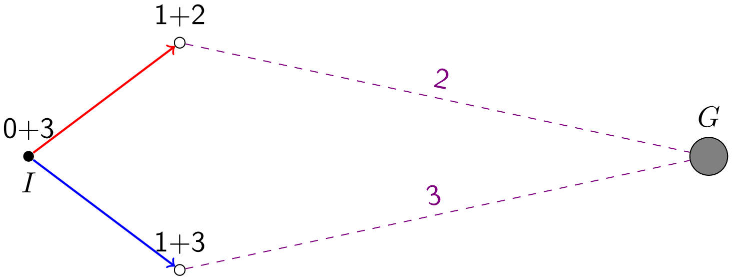
A* example

Example



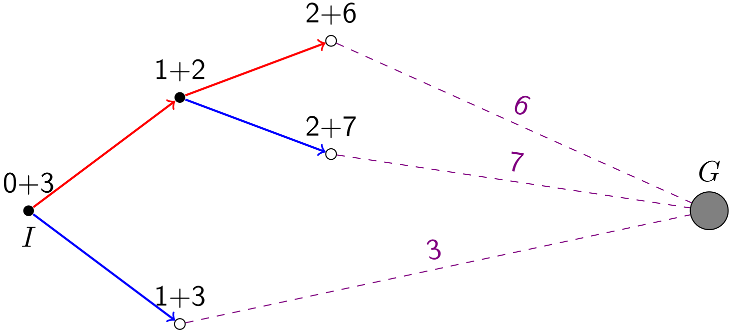
A* example

Example



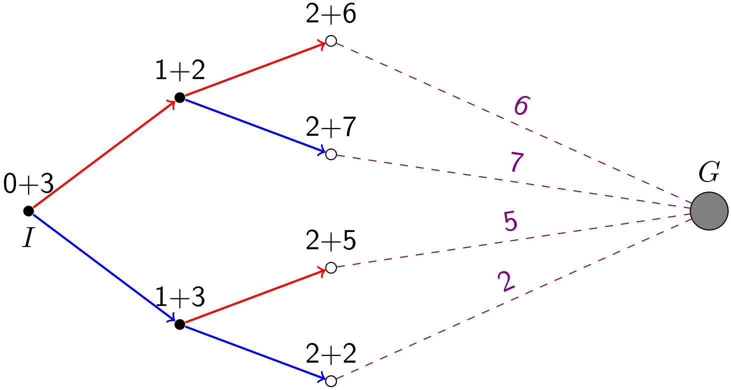
A* example

Example



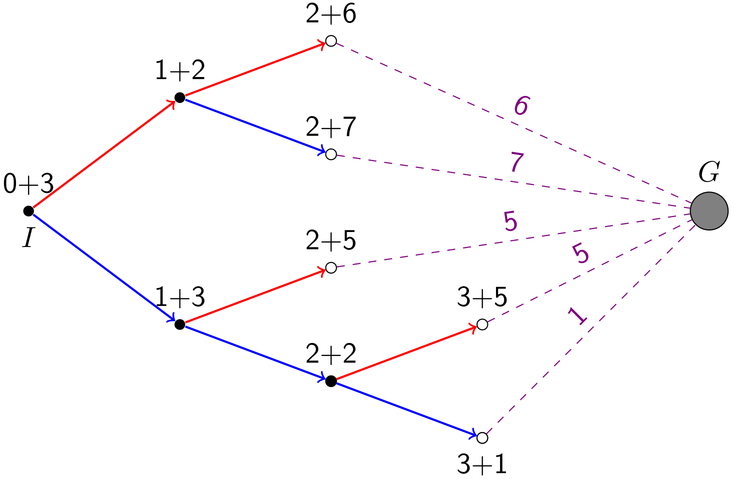
A* example

Example



A* example

Example



Terminology for A*

- f value of a node: defined by $f(\sigma) := g(\sigma) + h(\sigma)$
- generated nodes: nodes inserted into *open* at some point
- expanded nodes: nodes σ popped from *open* for which the test against *closed* and *distance* succeeds
- reexpanded nodes: expanded nodes for which *state*$(\sigma) \in$ *closed* upon expansion (also called reopened nodes)

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 if h is admissible and/or consistent (even without duplicate detection)
- never reopens nodes if h is consistent

Implementation notes:

- in the heap-ordering procedure, it is considered a good idea to break ties in favour of lower h values
- can simplify algorithm if we know that we only have to deal with consistent heuristics
- common, hard to spot bug: test membership in *closed* at the wrong time

Weighted A*

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():
    $\sigma$ = open.pop-min()
    if state($\sigma$) $\notin$ closed or $g(\sigma) <$ distance(state($\sigma$)):
        closed := closed $\cup$ {state($\sigma$)}
        distance($\sigma$) := $g(\sigma)$
        if is-goal(state($\sigma$)):
            return extract-solution($\sigma$)
        for each $\langle o, s \rangle \in$ succ(state($\sigma$)):
            $\sigma'$ := make-node($\sigma, o, s$)
            if $h(\sigma') < \infty$:
                open.insert($\sigma'$)
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

Hill-climbing

Hill-climbing

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

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

Enforced hill-climbing

Enforced hill-climbing: procedure improve

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

⇝ breadth-first search for more promising node than σ_0

Enforced hill-climbing (ctd.)

Enforced hill-climbing

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

- 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