AI Planning for Autonomy

2. Search Algorithms

Basic Stuff You're Gonna Need to Search for a Solution
Where To Search Next?

Chris Ewin & Tim Miller

THE UNIVERSITY OF MELBOURNE

With slides by Nir Lipovetsky

Basic State Model: Classical Planning

Ambition:

Write one program that can solve all classical search problems.

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

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

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

$\rightarrow$ Different **models** and **controllers** obtained by relaxing assumptions in **blue** . . .

Solving the State Model: Path-finding in graphs

Search algorithms for planning exploit the **correspondence** between **(classical) states model** $\mathcal{S}(P)$ and **directed graphs**:

- The **nodes** of the graph represent the **states** s in the model
- The edges (s, s') capture corresponding transition in the model with same cost

In the **planning as heuristic search** formulation, the problem P is solved by **path-finding** algorithms over the **graph** associated with model $\mathcal{S}(P)$

Classification of Search Algorithms

Blind search vs. heuristic (or informed) search:

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

Systematic search vs. local search:

- Systematic search 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.

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

What works where in planning?

Blind search vs. heuristic search:

- For satisficing planning, heuristic search vastly outperforms blind algorithms pretty much everywhwere.
- For optimal planning, heuristic search also is better (but the difference is less pronounced).

Systematic search vs. local search:

- For satisficing planning, there are successful instances of each.
- For optimal planning, systematic algorithms are required.

What works where in planning?

Blind search vs. heuristic search:

- For satisficing planning, heuristic search vastly outperforms blind algorithms pretty much everywhwere.
- For optimal planning, heuristic search also is better (but the difference is less pronounced).

Systematic search vs. local search:

- For satisficing planning, there are successful instances of each.
- For optimal planning, systematic algorithms are required.

→ Here, we cover the subset of search algorithms most successful in planning. Only some Blind search algorithms are covered. (refer to Russel & Norvig Chapters 3 and 4 for that).

Agenda

Search Terminology

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

Search Terminology

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

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

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

Search Terminology

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

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

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

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

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

Search Terminology

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

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

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

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

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

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

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

World States vs. Search States

Reminder: Search Space for Classical Search

A (classical) search space is defined by the following three operations:

- start(): Generate the start (search) state.
- is-target(s): Test whether a given search state is a target state.
- succ(s): Generates the successor states (a, s') of search state s, along with the actions through which they are reached.

World States vs. Search States

Reminder: Search Space for Classical Search

A (classical) search space is defined by the following three operations:

- start(): Generate the start (search) state.
- is-target(s): Test whether a given search state is a target state.
- succ(s): Generates the successor states (a, s') of search state s, along with the actions through which they are reached.

Search states $\neq$ world states:

- Progression:

World States vs. Search States

Reminder: Search Space for Classical Search

A (classical) search space is defined by the following three operations:

- start(): Generate the start (search) state.
- is-target(s): Test whether a given search state is a target state.
- succ(s): Generates the successor states (a, s') of search state s, along with the actions through which they are reached.

Search states $\neq$ world states:

- Progression: Yes, search states = world states.
- Regression:

World States vs. Search States

Reminder: Search Space for Classical Search

A (classical) search space is defined by the following three operations:

- start(): Generate the start (search) state.
- is-target(s): Test whether a given search state is a target state.
- succ(s): Generates the successor states (a, s') of search state s, along with the actions through which they are reached.

Search states $\neq$ world states:

- Progression: Yes, search states = world states.
- Regression: No, search states = sets of world states, represented as conjunctive sub-goals.

$\rightarrow$ We consider progression in the entire course, unless explicitly stated otherwise. We use "s" to denote world/search states interchangeably.

Search States vs. Search Nodes

- **Search states** s: States (vertices) of the search space.
- **Search nodes** σ: Search states, plus information on where/when/how they are encountered during search.

Search States vs. Search Nodes

- **Search states** s: States (vertices) of the search space.
- **Search nodes** σ: Search states, plus information on where/when/how they are encountered during search.

What is in a search node?

Different search algorithms store different information in a search node σ, but typical information includes:

- *state*(σ): Associated search state.
- *parent*(σ): Pointer to search node from which σ is reached.
- *action*(σ): An action leading from *state*(*parent*(σ)) to *state*(σ).
- $g(\sigma)$: Cost of σ (cost of path from the root node to σ).

For the root node, *parent*(σ) and *action*(σ) are undefined.

Criteria for Evaluating Search Strategies

Guarantees:

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

Optimality: Are the returned solutions guaranteed to be optimal?

Criteria for Evaluating Search Strategies

Guarantees:

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

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

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

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

Criteria for Evaluating Search Strategies

Guarantees:

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

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

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

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

Typical state space features governing complexity:

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

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

Agenda

Before We Begin, ctd.

Blind search strategies we'll discuss:

- Breadth-first search. Advantage: time complexity.
  Variant: Uniform cost search.
- Depth-first search. Advantage: space complexity.
- Iterative deepening search. Combines advantages of breadth-first search and depth-first search. Uses depth-limited search as a sub-procedure.

Before We Begin, ctd.

Blind search strategies we'll discuss:

- Breadth-first search. Advantage: time complexity.
  Variant: Uniform cost search.
- Depth-first search. Advantage: space complexity.
- Iterative deepening search. Combines advantages of breadth-first search and depth-first search. Uses depth-limited search as a sub-procedure.

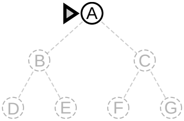
Blind search strategy we won't discuss:

- Bi-directional search. Two separate search spaces, one forward from the initial state, the other backward from the goal. Stops when the two search spaces overlap.

Breadth-First Search: Illustration and Guarantees

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

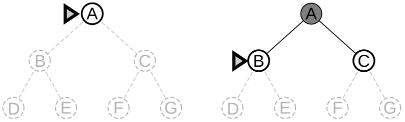
Illustration:



Breadth-First Search: Illustration and Guarantees

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

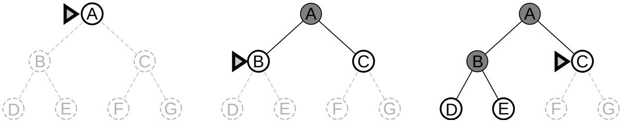
Illustration:



Breadth-First Search: Illustration and Guarantees

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

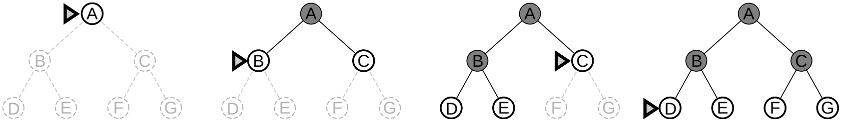
Illustration:



Breadth-First Search: Illustration and Guarantees

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

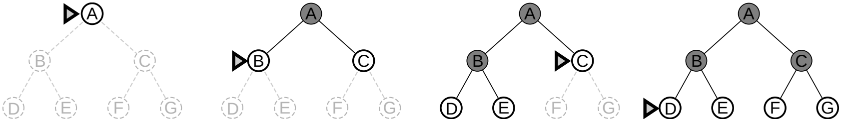
Illustration:



Breadth-First Search: Illustration and Guarantees

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

Illustration:



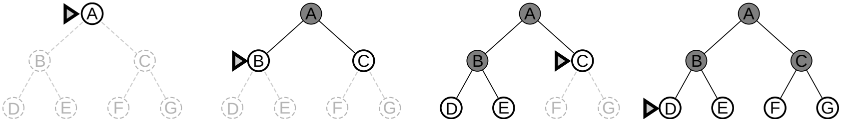
Guarantees:

- Completeness?

Breadth-First Search: Illustration and Guarantees

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

Illustration:



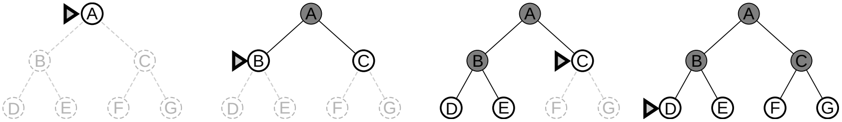
Guarantees:

- Completeness? Yes.
- Optimality?

Breadth-First Search: Illustration and Guarantees

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

Illustration:



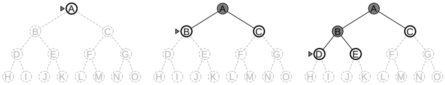
Guarantees:

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

Depth-First Search: Illustration

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

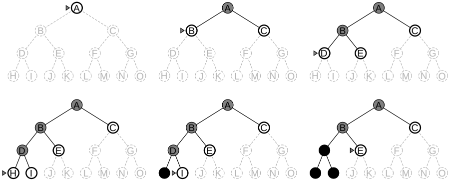
Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Illustration

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

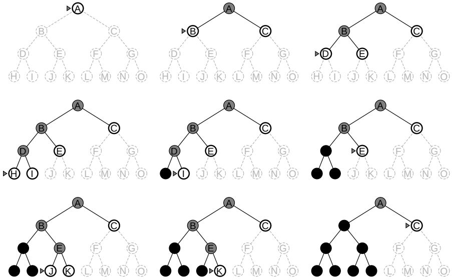
Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Illustration

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

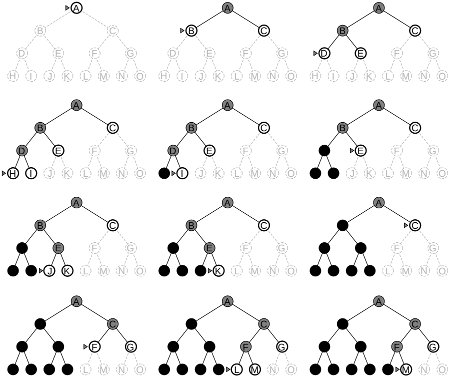
Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Illustration

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

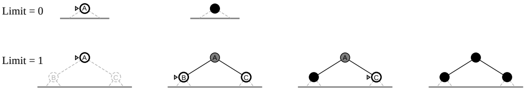
Illustration: (Nodes at depth 3 are assumed to have no successors)



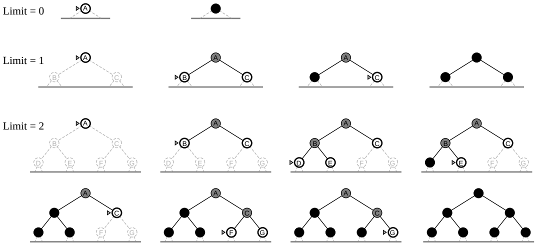
Iterative Deepening Search: Illustration

Limit = 0

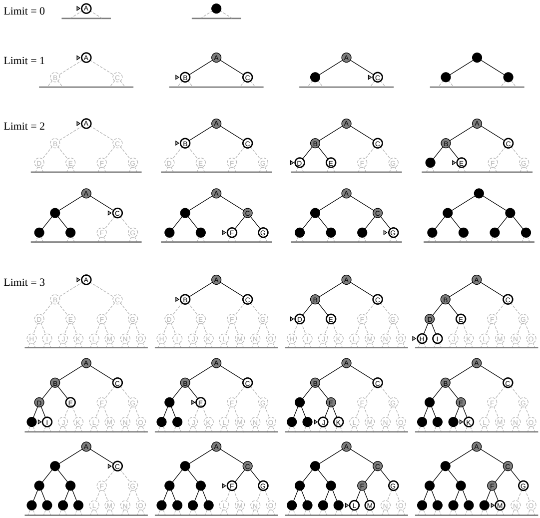
Iterative Deepening Search: Illustration



Iterative Deepening Search: Illustration



Iterative Deepening Search: Illustration



Iterative Deepening Search: Guarantees and Complexity

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

Agenda

Heuristic Search Algorithms: Systematic

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

Systematic heuristic search algorithms:

- Greedy best-first search.
  → One of 3 most popular algorithms in satisficing planning.
- Weighted A^*.
  → One of 3 most popular algorithms in satisficing planning.
- A^*.
  → Most popular algorithm in optimal planning. (Rarely ever used for satisficing planning.)
- IDA^*, depth-first branch-and-bound search, breadth-first heuristic search, . . .

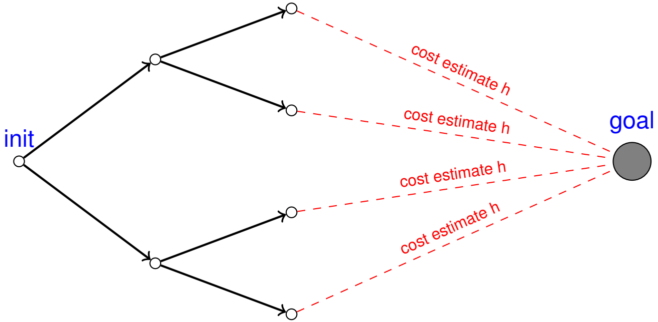
Heuristic Search Algorithms: Local

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

Local heuristic search algorithms:

- Hill-climbing.
- Enforced hill-climbing.
  → One of 3 most popular algorithms in satisficing planning.
- Beam search, tabu search, genetic algorithms, simulated annealing, . . .

Heuristic Search: Basic Idea



→ Heuristic function h estimates the cost of an optimal path to the goal; search gives a preference to explore states with small h.

Heuristic Functions

Heuristic searches require a heuristic function to estimate remaining cost:

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

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

Heuristic Functions: Discussion

What does it mean to "estimate remaining cost"?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to "work" (= be correct and complete).

  $\rightarrow h$ is *any* function from states to numbers . . .

Heuristic Functions: Discussion

What does it mean to “estimate remaining cost”?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to “work” (= be correct and complete).

  $\rightarrow$ h is *any* function from states to numbers ...
- Search performance depends crucially on “how well h reflects h^*”!!

  $\rightarrow$ This is informally called the informedness or quality of h.

Heuristic Functions: Discussion

What does it mean to “estimate remaining cost”?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to “work” (= be correct and complete).

  $\rightarrow$ h is *any* function from states to numbers . . .
- Search performance depends crucially on “how well h reflects h^*”!!

  $\rightarrow$ This is informally called the informedness or quality of h.
- For some search algorithms, like A^*, we can *prove* relationships between formal quality properties of h and search efficiency (mainly the number of expanded nodes).

Heuristic Functions: Discussion

What does it mean to "estimate remaining cost"?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to "work" (= be correct and complete).

  $\rightarrow$ h is *any* function from states to numbers . . .
- Search performance depends crucially on "how well h reflects h^*"!!

  $\rightarrow$ This is informally called the informedness or quality of h.
- For some search algorithms, like A^*, we can *prove* relationships between formal quality properties of h and search efficiency (mainly the number of expanded nodes).
- For other search algorithms, "it works well in practice" is often as good an analysis as one gets.

$\rightarrow$ We will analyze in detail approximations to one particularly important heuristic function in planning: h^+.

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h . . ."

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h . . . "

Any other property of h that search performance crucially depends on?

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h . . . "

Any other property of h that search performance crucially depends on?

". . . and on the computational overhead of computing h!!"

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h . . . "

Any other property of h that search performance crucially depends on?

". . . and on the computational overhead of computing h!!"

Extreme cases:

- $h = h^*$: Perfectly informed; computing it = solving the planning task in the first place.
- $h = 0$: No information at all; can be "computed" in constant time.

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h ..."

Any other property of h that search performance crucially depends on?

"... and on the computational overhead of computing h!!"

Extreme cases:

- $h = h^*$: Perfectly informed; computing it = solving the planning task in the first place.
- $h = 0$: No information at all; can be "computed" in constant time.

$\rightarrow$ Successful heuristic search requires a good trade-off between h's informedness and the computational overhead of computing it.

$\rightarrow$ This really is what research is all about! Devise methods that yield good estimates at reasonable computational costs.

Properties of Heuristic Functions

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

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

Properties of Heuristic Functions

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

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

$\rightarrow$ **Relationships?**

Properties of Heuristic Functions

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

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

$\rightarrow$ **Relationships?**

Proposition. *Let Π be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$, and let h be a heuristic for Π. If h is consistent and goal-aware, then h is admissible. If h is admissible, then h is goal-aware. If h is admissible, then h is safe. No other implications of this form hold.*

Properties of Heuristic Functions

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

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

$\rightarrow$ **Relationships?**

Proposition. *Let Π be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$, and let h be a heuristic for Π. If h is consistent and goal-aware, then h is admissible. If h is admissible, then h is goal-aware. If h is admissible, then h is safe. No other implications of this form hold.*

Proof. $\rightarrow$ **Exercise, perhaps.**

Agenda

Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

open $:=$ **new** priority queue ordered by ascending $h(\textit{state}(\sigma))$
open.insert(make-root-node(init()))
closed $:= \emptyset$
while not *open*.empty():
    $\sigma :=$ *open*.pop-min() /* get best state */
    if $\textit{state}(\sigma) \notin \textit{closed}$: /* check duplicates */
        $\textit{closed} := \textit{closed} \cup \{\textit{state}(\sigma)\}$ /* close state */
        if is-goal(state(σ)): **return** extract-solution(σ)
        for each $(a, s') \in$ succ($\textit{state}(\sigma)$): /* expand state */
            $\sigma' :=$ make-node(σ, a, s')
            if $h(\textit{state}(\sigma')) < \infty$: *open*.insert($\sigma'$)
return unsolvable

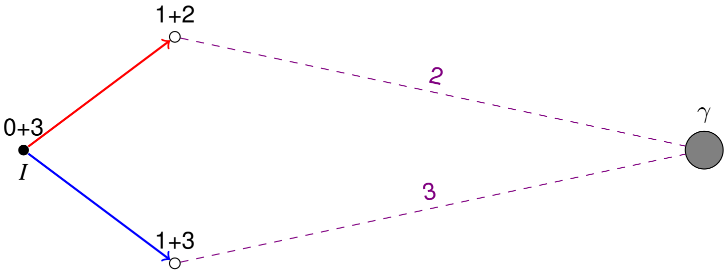
A^*

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

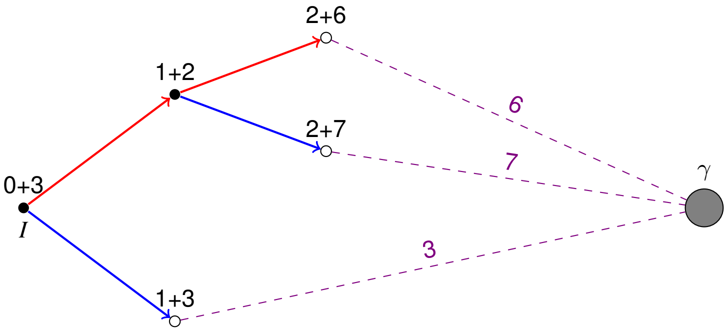
open $:=$ **new** priority queue ordered by ascending $g(\mathit{state}(\sigma)) + h(\mathit{state}(\sigma))$
open.insert(make-root-node(init()))
closed $:= \emptyset$
best-g $:= \emptyset$/* maps states to numbers */
while not *open*.empty():
    σ := *open*.pop-min()
    if $\mathit{state}(\sigma)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

A^*: Example

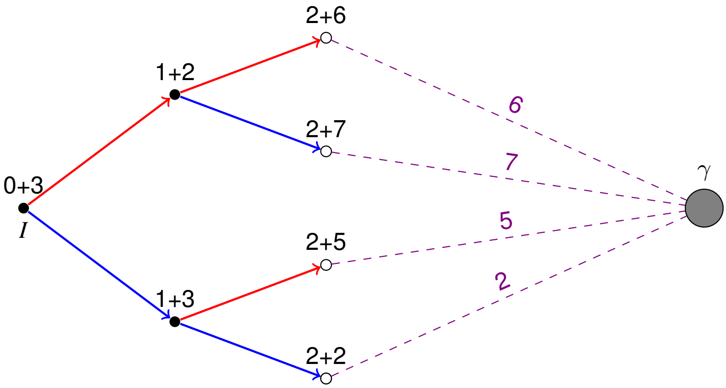
A^*: Example



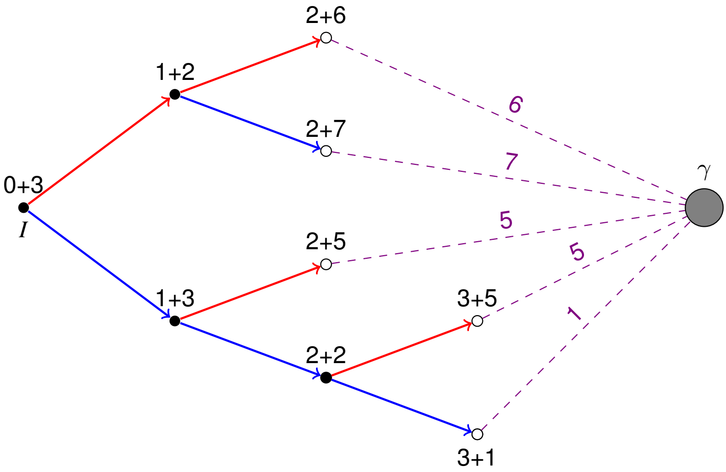
A^*: Example



A^*: Example



A^*: Example



A^*: Example