Heuristic Search for Classical Planning

B(E)4M36PUI – Artificial Intelligence Planning

Antonín Komenda

AIC, FEE, CTU

March 6, 2023

Outline

Search in Transition Systems
Principles

- big success of symbolic AI[1]
- most automated planners use search algorithms
- not only for planners! – CSP/(D)COP, (PO)MDPs, MCTS (AlphaGo), ...
- required solution properties: soundness and completeness
- categorization of search algorithms[2] by:
  - solution quality: satisficing vs. optimal
  - direction: forward, backward, bidirectional, sampling, ...
  - informativeness: uninformed, informed (heuristic – weak to strong)
  - orderliness: local vs. global (systematic)

[1]Stuart Russell, Peter Norvig: Artificial Intelligence: A Modern Approach (4th Edition). Pearson 2020, ISBN 9780134610993 (Chapters 3 and 4)

[2]Stefan Edelkamp, Stefan Schrödl: Heuristic Search - Theory and Applications. Academic Press 2012, ISBN 978-0-12-372512-7

Search in Transition Systems
Solution Properties

Two required properties on a solution of a problem represented as a search in a transition system:

- soundness(correctness): a found solution (plan) by a search algorithm is sound if the path in the transition system sequentially follows the nodes and transitions (only allowed states and only allowed transitions (action applications) between them) and if defined starts in the designated initial state and ends in one of designated goal states
- completeness: a search algorithm is complete if it guarantees founding a sound solution if such exists

Search in Transition Systems
Solution Quality

Two different problems:

- satificing planning: any plan is a solution (in benchmarks we prefer cheaper solutions)
- optimal planning: only the cheapest plan is a solution (cheapest = shortest for unit cost planning)

Search algorithms can be used for both types, however:

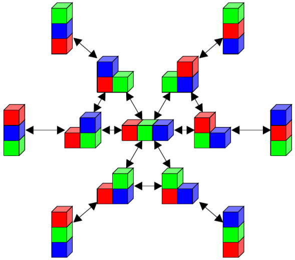
- the differences are non-trivial and often domain (or even problem) specific
- some planning domains/problems are P for satisficing planning, but NP for optimal
- almost no overlap between good techniques for satisficing planning and good techniques for optimal planning (search algorithms, search node to state(s) mapping, heuristics, ...)

Search in Transition Systems

Example

Two different problems:

- satificing planning: any plan is a solution (in benchmarks we prefer cheaper solutions)
- optimal planning: only the cheapest plan is a solution (cheapest = shortest for unit cost planning)



Search Space Representation

Principles

The search nodes does not necessarily correspond to the states one-to-one:

- states are defined in the context of the transition system
- search can visit a state repeatedly (with different path costs)
- search form a tree (repeated visits of the same state are different search nodes!)
- search node is typically a more rich structure then a state (state information, cost to the search node, heuristic value, reference to the parent node, ...)

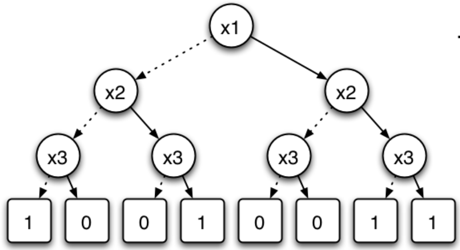
A search node can relate to:

- one state
- set of states – compact representation of set of states (actions applied to sets of states directly)
  - BDDs (Binary Decision Diagrams),
  - ADDs (Algebraic Decision Diagrams) $\rightsquigarrow$ FDR Symbolic Search (SymBA* planner)

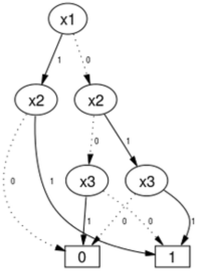
Search Space Representation

BDDs/ADDs Intuition

BDD example (left: Boolean function, right: corresponding BDD):



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



$\rightsquigarrow$

BDDs vs. ADDs are analogical to STRIPS vs. FDR.

Search Direction
Principles

The direction is defined in the context of the initial state and goal state(s):

- forward search (progression): search from the initial state towards the goal(s)
- backwars search (regression): search from the goal state(s) towards the init
- bidirectional search: search simultaneously from the intial state towards the goal(s) and vice versa while searching where the searches meet
- sampling search: syntatic generation of states and multi-parallel search (multiple bidirectional searches in parallel)

Search Direction

Forward search vs. backward search

A planning problem is not symmetric from perspective of forward and backward searches:

- forward search starts from a single initial state; backward search starts from a set of goal states
- recap.: action application function $\mathrm{app}_a(s)$ is non-injective (more actions can end up in the same state), but deterministic (the outcome of $\mathrm{app}_a(s)$ is always the same)

forward application induces a tree of individual states

backward application induces a tree of sets of states

Search Informativeness
Intuition

To improve efficiency of the search we can extract structural information from the domain and/or problem definition.
The extraction can be done on various levels:

- (rarely) from the lifted form (PDDL)
- from STRIPS/FDR representation (FDR contains structural information in the variable domains per se!) $\rightsquigarrow$ most of the domain-independent heuristics
- from expert knowledge (domain-specific heuristics – Manhattan distance, Euclidean distance, ...)

Search Informativeness

Structural Information for Navigation in Transition System

How to decide which state to consider next (structural information for navigating the search):

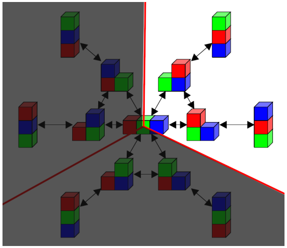
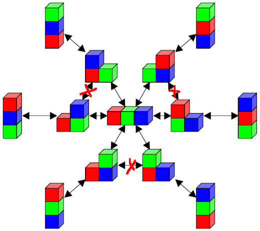
- uninfromed search:
  - depth-first search (DFS)
  - breadth-first search (BFS)
  - iterative deepening (depth-first) search (IDS/IDDFS)
  - ...
- heuristic (informed) search:
  - hill-climbing
  - simulated annealing
  - beam search
  - greedy best-first search (GBFS)
  - A*
  - Weighted A* (WA*)
  - iterative deepening A* (IDA*)
  - ...

Search Informativeness

Structural Information for Elimination in Transition System

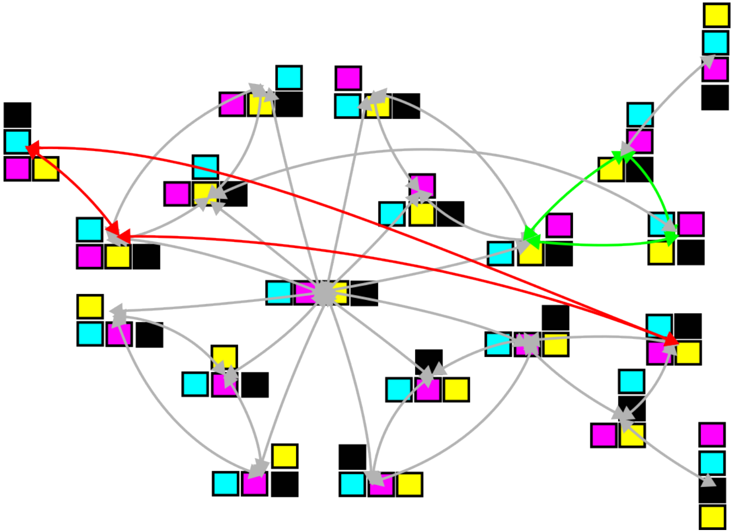
Using the extracted structural information for pruning:

- invariants (e.g., always at least one block on the table, never more then three blocks on each other)
- mutexes (e.g., if red block is on the blue block, blue block cannot be on the red block – cf. STRIPS representation, if one fact is true another fact cannot be true as well, Boolean formulas)
- symmetry elimination (see below left)
- initially prune other then helpful actions (see below right)
- ...


,


Search Informativeness

Structural Information for Elimination in Transition System



Search Orderliness

Systematic vs. Local Search

Scope of the search navigation can vary:

- local search:
  - does not keep information about the whole searched space
  - can unnecessarily repeat work (search the same state repeatedly)
  - usually does not provide complete search (it can miss a solution)
  - usually more computationally efficient
  - algorithms: random walk, hill-climbing, simulated annealing, beam search, ...
- systematic (global) search:
  - keeps information about the searched space (e.g., OPEN, and/or CLOSED lists, marking visited states, ...)
  - complete search
  - usually memory/computationally heavier
  - algorithms: DFS, BFS, IDS, greedy best-first search, A*, Weighted A* , IDA* , ...

Heuristic Functions

Heuristic principles

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, ...)
- For other search algorithms, "it works well in practice" is often as good an analysis as one gets.

Heuristic Functions

Perfect heuritic

Let's assume a transition system $\Sigma = \langle N, E, L, c \rangle$ and a set of goal nodes $G \subseteq N$, then a optimal/perfect heuristicfunction $h^*(n)$ in it:

- maps each node n to the lenght of a cheapest (shortest) path from n to a goal state $n_G \in G$
- $h^*(n) = \infty$ iff no goal state is reachable from n (i.e., n is a dead-end)

Heuristic Functions

Heuristic properties

Let's assume a transition system $\Sigma = \langle N, E, L, c \rangle$ and a set of goal nodes $G \subseteq N$, then a heuristic h is called:

- safe: for all $n \in N : (h(n) = \infty) \implies (h^*(n) = \infty)$
- goal-aware: for all $n_g \in G : h(n_g) = 0$
- admissible for all $n \in N : h(n) \leq h^*(n)$
- consistent[3] for all $(n \rightarrow n') \in E : h(n) \leq c(n \rightarrow n') + h(n')$

[3] Sometimes called monotonous, but what is really monotonic then?

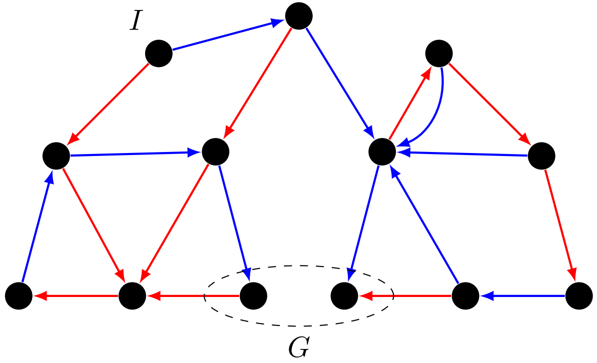
Building blocks of search algorithms
Principles

Seach needs three operations in general transition system $\Sigma = \langle N, E, L, c \rangle$:

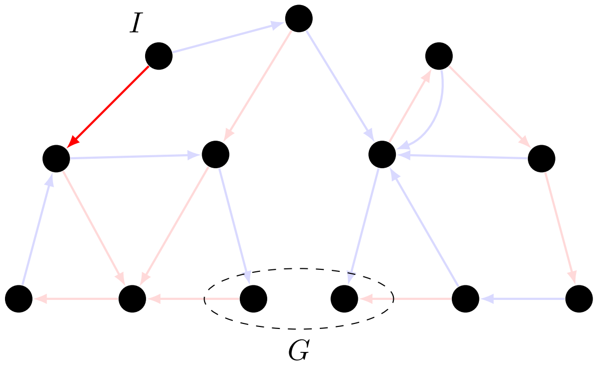
- $\mathrm{init}() : \emptyset \rightarrow N$ generate the initial node
- $\mathrm{is\text{-}goal}(n) : N \rightarrow \{\top, \bot\}$ test if a given state s is a goal node
- $\mathrm{succ}(n) : N \rightarrow \{\langle l, n' \rangle) | nln' \in E\}$: generate a set of successor nodes of a node n, together with the applied transition label, which they were reached with

These three operation form a search space (search nodes vs. transition system nodes).

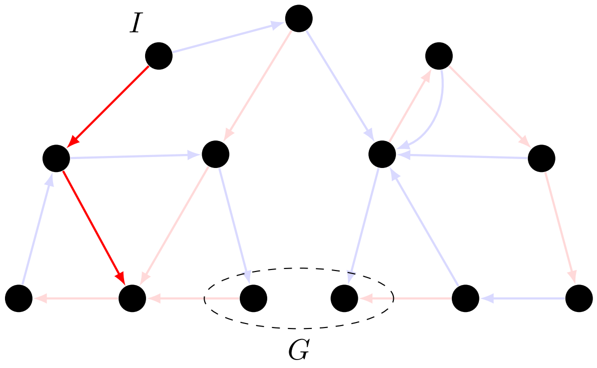
Progression planning example (depth-first search)



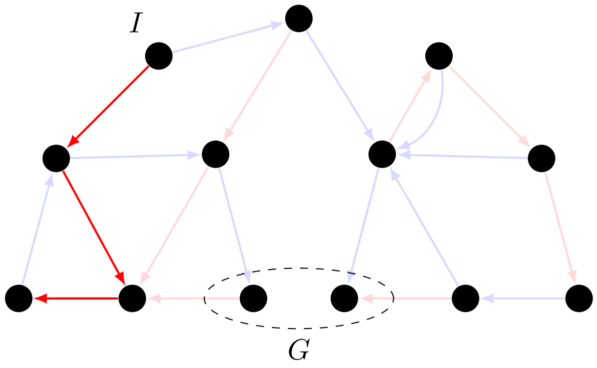
Progression planning example (depth-first search)



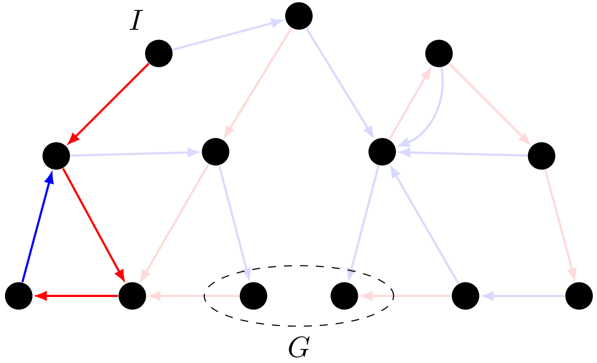
Progression planning example (depth-first search)



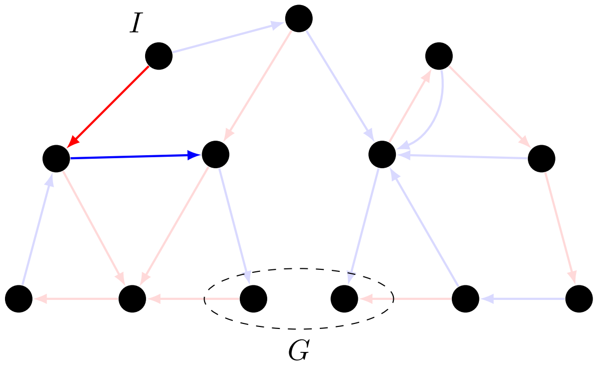
Progression planning example (depth-first search)



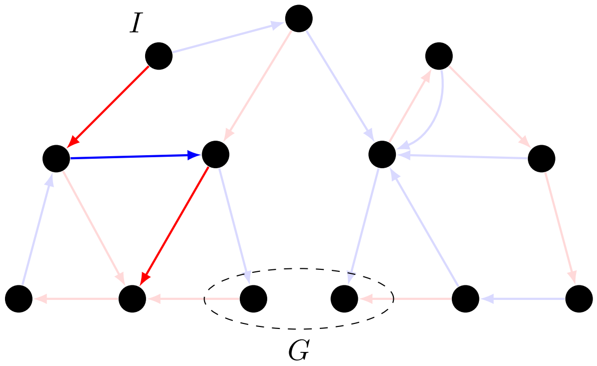
Progression planning example (depth-first search)



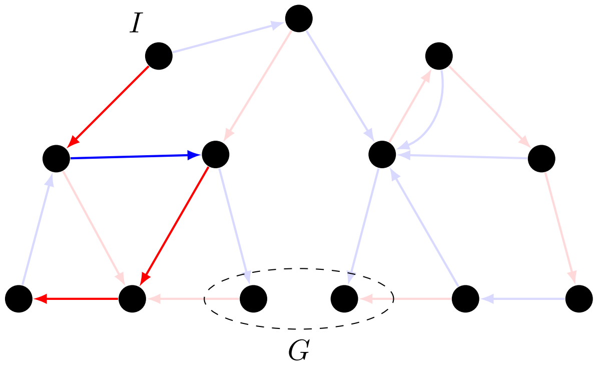
Progression planning example (depth-first search)



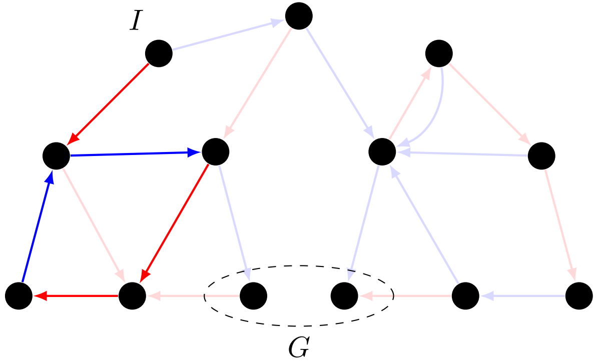
Progression planning example (depth-first search)



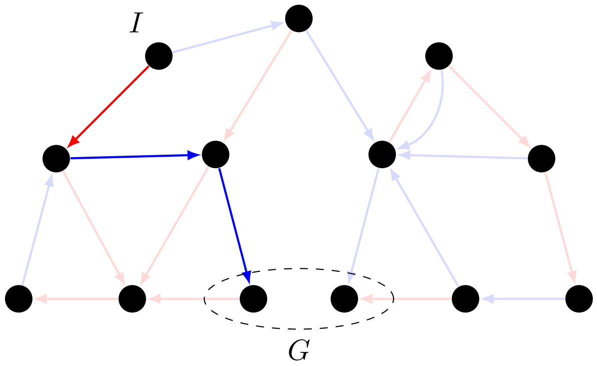
Progression planning example (depth-first search)



Progression planning example (depth-first search)



Progression planning example (depth-first search)



Search algorithms for planning
Principles

Lets assume a planning problem in STRIPS $\Pi = \langle F, A, c, s_I, G\rangle$ and induced transition system $\Sigma = \langle N, E, L, c\rangle$, where $N = S = 2^F, L = A, (n, n') \in E | \mathrm{app}_a(n) = n'$, then we define the search operations as:

- $\mathrm{init}() \mapsto s_I$
- $\mathrm{is\text{-}goal}(s) \mapsto \begin{cases} \top & G \subseteq s \\ \bot & \textit{otherwise} \end{cases}$
- $\mathrm{succ}(s) \mapsto \{\langle a, s'\rangle \,| a \in A, s' = \mathrm{app}_a(s)\}$

Heuristic search algorithms for planning

Principles

Heuristic seach needs four operations in general transition system $\Sigma = \langle N, E, L, c \rangle$:

- $\mathrm{init}() : \emptyset \rightarrow N$ generate the initial node
- $\mathrm{is\text{-}goal}(n) : N \rightarrow \{\top, \bot\}$ test if a given state s is a goal node
- $\mathrm{succ}(n) : N \rightarrow \{\langle l, n' \rangle) | nln' \in E\}$: generate a set of successor nodes of a node n, together with the applied transition label, which they were reached with
- $h(n) : N \rightarrow \mathbb{R}^{0+} \cup \{\infty\}$: the heuristic function (or just heuristic), the value $h(n)$ is called heuristic estimate or heurstic value of heuristic h for node n; it estimates the distance from n to the nearest (cheapest) goal node

A complement function to $h(n)$ is:

- $g_P(n) : N \rightarrow \mathbb{R}^{0+}$: the distance function (or just distance when in context of a state), the value $g_P(n)$ is the accumulated cost of the transitions over a path P from the initial node n_0 to n, i.e., $g_P(n) \mapsto \sum_{i=0}^{k} c(n_i \rightarrow n_{i+1}) | P = n_o l_1 n_1 l_2 n_2 \ldots l_k, n_k = n$

Search algorithms for planning

Algorithm classification

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)

Search algorithms for planning

Search algorithms in the context of 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 planning 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.

Search algorithms for planning

Search algorithms in the context of planning

Most common combination of the search properties for planning:

- solution quality: satisficing vs. optimal
- direction: forward, backward, bidirectional, sampling, ...
- informativeness: uninformed, informed (heuristic – weak to strong)
- orderliness: local vs. global (systematic)
- infromed (heuristic) vs. uninformed (one of the greatest achievements of automated planning are automatically derived domain-independent heuristics)
- global (systematic) vs. local (completeness usually require systematic search)
- forward vs. backward (not so strict, but still prevalent)

Search algorithms for planning

Search algorithms in the context of planning (systematic)

Most popular systematic heuristic forward search algorithms for planning:

- greedy best-first search(FastForward planner)
- A*(SymBA*)
- weighted A*(LAMA planner)
- IDA*
- depth-first branch-and-bound search
- breadth-first heuristic search
- ...

Search algorithms for planning

Search algorithms in the context of planning (local)

Most popular local heuristic forward search algorithms for planning:

- enforced hill-climbing[4] (FastForward planner)
- hill-climbing
- deep learning (technically local, as no strong guarantees)
- beam search
- tabu search
- genetic algorithms
- simulated annealing
- ...

[4] As hill-climbing, but uses BFS for tie breaking.

Search algorithms for planning

Pseudocode (greedy best-first search (GBFS) with duplicate detection)

```
$open$ := **new** empty min-heap ordered by $f(\sigma.s) = h(\sigma.s)$
$open$.insert(search-node($\emptyset, \emptyset$, init()))
$closed$ := **new** empty set
**while not** $open$.empty():
    $\sigma = open$.pop-min()
    **if** $\sigma.s \notin$ closed:
        $closed$.insert($\sigma$)
        **if** is-goal($\sigma.s$):
            **return** path($\sigma$)
        **for each** $\langle a, s' \rangle \in$ succ($\sigma.s$):
            $\sigma'$ := search-node($\sigma, a, s'$)
            **if** $h(\sigma.s) < \infty$:
                $open$.insert($\sigma'$)
**return** $\perp$
```

Search algorithms for planning

Properties (greedy best-first search (GBFS) with duplicate detection)

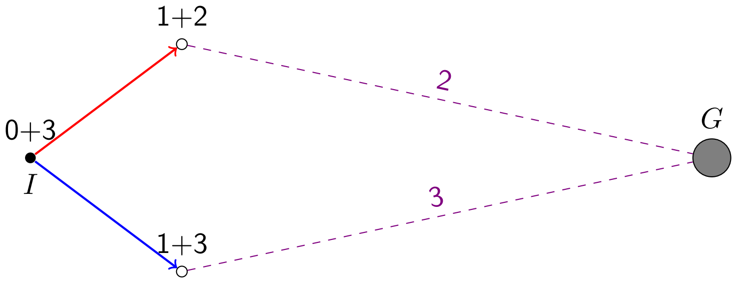
- one of the three most commonly used algorithms for satisficing planning
- assuming STRIPS, therefore $N = S$,thus $\sigma.s = \sigma.n$ (states and transition system nodes are identical, i.e., the search works over the induced transition system)
- requires safe heuristic (**if** $h(\sigma.s) < \infty$)
- complete for safe heuristics and due to duplicate detection
- satisficing (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)

Search algorithms for planning

Pseudocode (A* with duplicate detection and reopening)

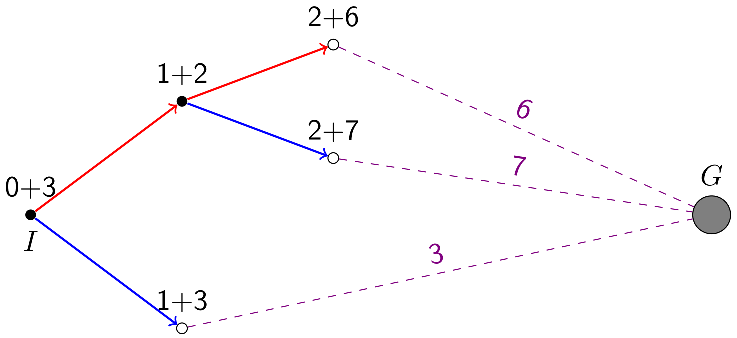
```
open := new empty min-heap ordered by $f(\sigma.s) = g_\sigma(\sigma.s) + h(\sigma.s)$
open.insert(search-node($\emptyset, \emptyset$, init()))
closed := new empty set
distance := new empty map (key, value)
while not open.empty():
    $\sigma$ = open.pop-min()
    if $\sigma.s \notin$ closed or $g_\sigma(\sigma.s) < \mathrm{distance}(\sigma.s)$:
        # $\sigma$ is an expanded node or reexpanded/reopened if $\sigma.s \in$ closed
        closed.insert($\sigma$)
        distance.insert($\sigma$,$g_\sigma(\sigma.s)$)
        if is-goal($\sigma.s$):
            return path($\sigma$)
        for each $\langle a, s' \rangle \in \mathrm{succ}(\sigma.s)$:
            $\sigma'$ := search-node($\sigma, a, s'$)
            if $h(\sigma.s) < \infty$:
                open.insert($\sigma'$) # $\sigma'$ is a generated node
return ⊥
```

A* example
Example

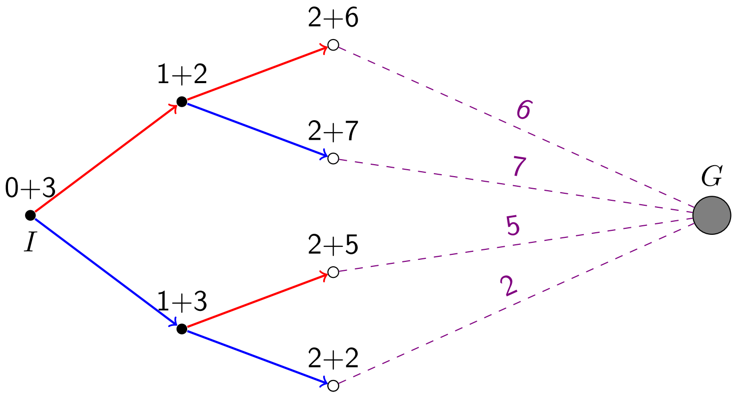


A* example

Example

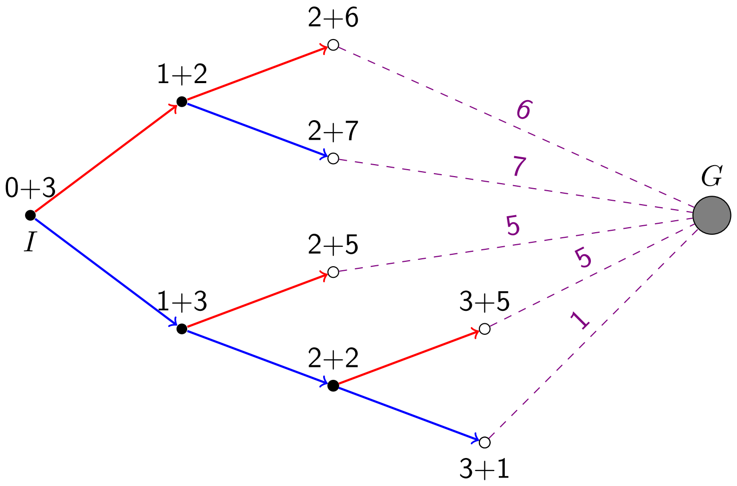


A* example
Example



A* example

Example



Search algorithms for planning

Properties (A* with duplicate detection and reopening)

- 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
- slight abuse of notation, we assume $g_{\sigma}(\sigma.s) = g_{path(\sigma)}(\sigma.s)$

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

Search algorithms for planning

Pseudocode (Weighted A* with duplicate detection and reopening)

```
*open* := **new** empty min-heap ordered by $f(\sigma.s) = g_\sigma(\sigma.s) + h(\sigma.s) \cdot W$
*open*.insert(search-node($\emptyset, \emptyset$,init()))
*closed* := **new** empty set
*distance* := **new** empty map (key, value)
**while not** *open*.empty():
    $\sigma$ = *open*.pop-min()
    **if** $\sigma.s \notin$ closed **or** $g_\sigma(\sigma.s) <$ distance$(\sigma.s)$:
        *closed*.insert($\sigma$)
        *distance*.insert($\sigma$,$g_\sigma(\sigma.s)$)
        **if** is-goal($\sigma.s$):
            **return** path($\sigma$)
        **for each** $\langle a, s' \rangle \in$ succ($\sigma.s$):
            $\sigma'$ := search-node($\sigma, a, s'$)
            **if** $h(\sigma.s) < \infty$:
                *open*.insert($\sigma'$)
**return** $\perp$
```

Search algorithms for planning

Properties (Weighted A* with duplicate detection and reopening)

- The weight $W \in \mathbb{R}^{0+}$ is a parameter of the algorithm
  - for $W = 0$, behaves like breadth-first search ($f(\sigma.s) = g_\sigma(\sigma.s) + 0$)
  - for $W = 1$, behaves like A*
  - for $W \to 1$, behaves like greedy best-first search ($f(\sigma.s) \sim 0 + h(\sigma.s)$)
- 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

Search algorithms for planning

Pseudocode (Hill-climbing)

```
$\sigma = \text{search-node}(\emptyset, \emptyset, \mathrm{init}())$
forever:
    if $\mathrm{is\text{-}goal}(\sigma.s)$:
        return $\text{path}(\sigma)$
    $\Theta = \{\text{search-node}(\sigma, a, s') \mid \langle a, s' \rangle \in \mathrm{succ}(\sigma.s)\}$
    $\sigma :=$ an element of $\Theta$ minimizing $h(\sigma.s)$
```

- various tie breaking strategies
- can easily get stuck in local minima where improvement of $h(\sigma.s)$ is not possible (by one action)
- restarts

Search algorithms for planning

Pseudocode (Enforced hill-climbing)

```
$\sigma_o$ =search-node($\emptyset, \emptyset$,init())
while not is-goal($\sigma_o.s$):
    queue := new fifo-queue
    queue.push-back($\sigma_o$)
    closed := new empty set
    while not queue.empty():
        $\sigma$ =queue.pop-front():
        if $\sigma.s \notin$ closed:
            closed.insert($\sigma$)
            if $h(\sigma.s) < h(\sigma_o.s)$:
                $\sigma_o := \sigma$; break
            for each $\langle a, s' \rangle \in$ succ($\sigma.s$):
                $\sigma'$ := search-node($\sigma, a, s'$)
                queue.push-back($\sigma'$)
    else: return $\bot$
return path($\sigma$)
```

Search algorithms for planning

Properties (Enforced hill-climbing)

- breadth-first search for more promising node than σ_o
- one of the three most commonly used algorithms for satisficing planning
- can fail if procedure improve fails (when the goal is unreachable from σ_o)
- complete for undirected search spaces (where the successor relation is symmetric) if $h(\sigma.s) = 0$ for all goal nodes and only for goal nodes