

Principles of AI Planning

6. Planning as search: search algorithms

Bernhard Nebel and Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

November 15th, 2011

Principles of AI Planning

November 15th, 2011 — 6. Planning as search: search algorithms

6.1 Introduction to search algorithms for planning

6.2 Uninformed search algorithms

6.3 Heuristic search algorithms

Introduction

6.1 Introduction to search algorithms for planning

- Search nodes & search states
- Search for planning
- Common procedures for search algorithms

Introduction Nodes and states

Our plan for the next lectures

Choices to make:

1. search direction: progression/regression/both
~> previous chapter
2. search space representation: states/sets of states
~> previous chapter
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.
- ▶ In this chapter, we describe some popular and/or representative search algorithms, and (the basics of) how they apply to planning.
- ▶ Most of this is review of material that should be known (details: Russell and Norvig's textbook).

Search states vs. search nodes

In search, one distinguishes:

- ▶ **search states** $s \rightsquigarrow$ states (vertices) of the transition system
- ▶ **search nodes** $\sigma \rightsquigarrow$ 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/operator leading from $state(parent(\sigma))$ to $state(\sigma)$
- ▶ **g**(σ): cost of σ (length of path from the root node)

For the root node, $parent(\sigma)$ and $action(\sigma)$ are undefined.

Search states vs. planning states

Search states \neq (planning) states:

- ▶ **Search states** don't have to correspond to **states** in the planning sense.
 - ▶ progression: search states \approx **(planning) states**
 - ▶ regression: search states \approx **sets of states** (formulae)
- ▶ Search algorithms for planning where search states are planning states are called **state-space search** algorithms.
- ▶ Strictly speaking, regression is **not** an example of state-space search, although the term is often used loosely.
- ▶ However, we will put the emphasis on progression, which is almost always state-space search.

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**(\cdot): 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 $\Pi = \langle A, I, O, \gamma \rangle$ be a planning task.

Search space for progression search

states: all states of Π (assignments to A)

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

Search for planning: regression

Let $\Pi = \langle A, I, O, \gamma \rangle$ be a planning task.

Search space for regression search

states: all formulae over A (how many?)

- ▶ $\text{init}() = \gamma$
- ▶ $\text{is-goal}(\varphi) = \begin{cases} \text{true} & \text{if } I \models \varphi \\ \text{false} & \text{otherwise} \end{cases}$
- ▶ $\text{succ}(\varphi) = \{ \langle o, \varphi' \rangle \mid o \in O, \varphi' = \text{regr}_o(\varphi), \varphi' \text{ is satisfiable} \}$
(modified if splitting is used)

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.

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.

Common procedures for search algorithms

Before we describe the different search algorithms, we introduce three procedures used by all of them:

- ▶ **make-root-node**: Create a search node without parent.
- ▶ **make-node**: Create a search node for a state generated as the successor of another state.
- ▶ **extract-solution**: Extract a solution from a search node representing a goal state.

Procedure make-root-node

make-root-node: Create a search node without parent.

Procedure make-root-node

```
def make-root-node(s):
   $\sigma := \mathbf{new}$  node
  state( $\sigma$ ) := s
  parent( $\sigma$ ) := undefined
  action( $\sigma$ ) := undefined
  g( $\sigma$ ) := 0
  return  $\sigma$ 
```

Procedure make-node

make-node: Create a search node for a state generated as the successor of another state.

Procedure make-node

```
def make-node( $\sigma$ , o, s):
   $\sigma' := \mathbf{new}$  node
  state( $\sigma'$ ) := s
  parent( $\sigma'$ ) :=  $\sigma$ 
  action( $\sigma'$ ) := o
  g( $\sigma'$ ) := g( $\sigma$ ) + 1
  return  $\sigma'$ 
```

Procedure extract-solution

extract-solution: Extract a solution from a search node representing a goal state.

Procedure extract-solution

```
def extract-solution( $\sigma$ ):
  solution :=  $\mathbf{new}$  list
  while parent( $\sigma$ ) is defined:
    solution.push-front(action( $\sigma$ ))
     $\sigma :=$  parent( $\sigma$ )
  return solution
```

6.2 Uninformed search algorithms

- Breadth-first search without duplicate detection
- Breadth-first search with duplicate detection
- Random walk

Uninformed search algorithms

- ▶ Uninformed algorithms are less relevant for planning than heuristic ones, so we keep their discussion brief.
- ▶ Uninformed algorithms are mostly interesting to us because we can compare and contrast them to related heuristic search algorithms.

Popular uninformed systematic search algorithms:

- ▶ **breadth-first search**
- ▶ depth-first search
- ▶ iterated depth-first search

Popular uninformed local search algorithms:

- ▶ **random walk**

Breadth-first search without duplicate detection

Breadth-first search

```

queue := new fifo-queue
queue.push-back(make-root-node(init()))
while not queue.empty():
    σ = queue.pop-front()
    if is-goal(state(σ)):
        return extract-solution(σ)
    for each ⟨o, s⟩ ∈ succ(state(σ)):
        σ' := make-node(σ, o, s)
        queue.push-back(σ')
return unsolvable

```

- ▶ Possible improvement: **duplicate detection** (see next slide).
- ▶ Another possible improvement: test if σ' is a goal node; if so, terminate immediately. (We don't do this because it obscures the similarity to some of the later algorithms.)

Breadth-first search with duplicate detection

Breadth-first search with duplicate detection

```

queue := new fifo-queue
queue.push-back(make-root-node(init()))
closed := ∅
while not queue.empty():
    σ = queue.pop-front()
    if state(σ) ∉ closed:
        closed := closed ∪ {state(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ' := make-node(σ, o, s)
            queue.push-back(σ')
return unsolvable

```

Breadth-first search with duplicate detection

Breadth-first search with duplicate detection

```

queue := new fifo-queue
queue.push-back(make-root-node(init()))
closed := {}
while not queue.empty():
    σ = queue.pop-front()
    if state(σ) ∉ closed:
        closed := closed ∪ {state(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ' := make-node(σ, o, s)
            queue.push-back(σ')
return unsolvable

```

Random walk

Random walk

```

σ := make-root-node(init())
forever:
    if is-goal(state(σ)):
        return extract-solution(σ)
    Choose a random element ⟨o, s⟩ from succ(state(σ)).
    σ := make-node(σ, o, s)

```

- ▶ The algorithm usually does not find any solutions, unless almost every sequence of actions is a plan.
- ▶ Often, it runs indefinitely without making progress.
- ▶ It can also fail by reaching a **dead end**, a state with no successors. This is a weakness of many local search approaches.

6.3 Heuristic search algorithms

- Heuristics: definition and properties
- Systematic heuristic search algorithms
- Heuristic local search algorithms

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
- ▶ ...

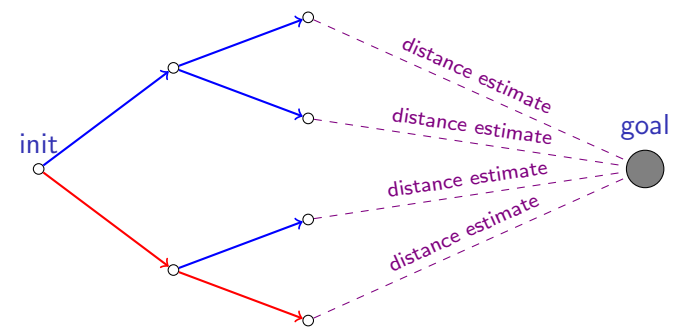
Heuristic search algorithms: local

- ▶ 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 $h : \Sigma \rightarrow \mathbb{N}_0 \cup \{\infty\}$.

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, ...)
- ▶ 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 $h'(\sigma) := h(\text{state}(\sigma))$.
 - ▶ 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 $\text{state}(\sigma)$ **pseudo-heuristics**.
- ▶ In practice there are sometimes good reasons to have the heuristic value depend on the generating path of σ (e. g., the **landmark pseudo-heuristic**, Richter et al. 2008).

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

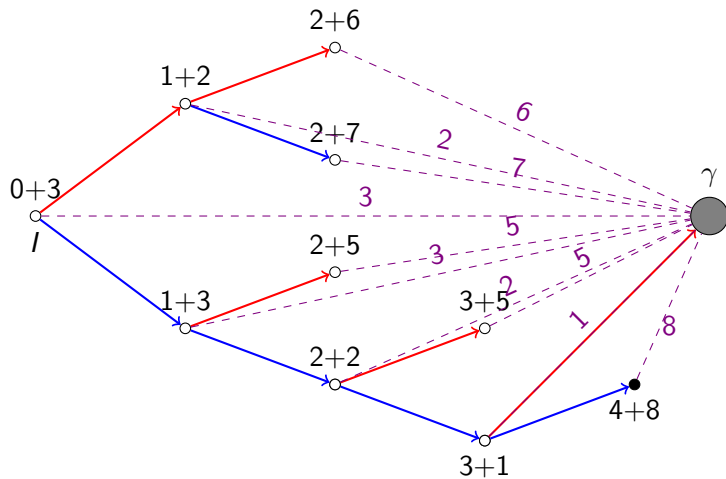
A* (with duplicate detection and reopening)

```

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(state( $\sigma$ )) :=  $g(\sigma)$ 
        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
    
```

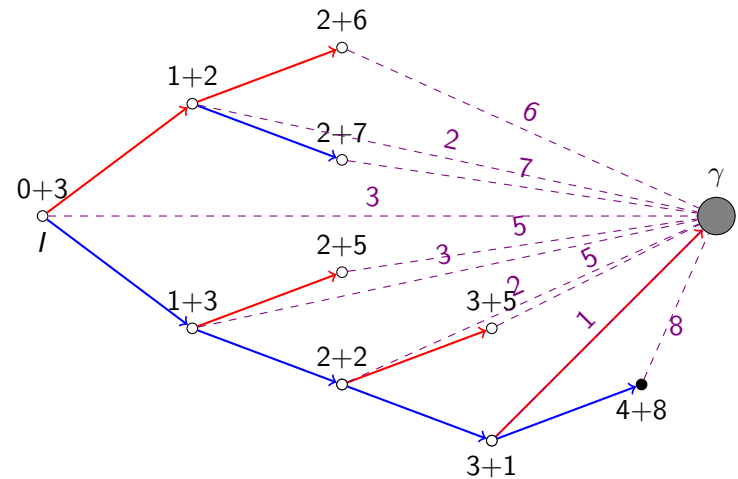
A* example

Example



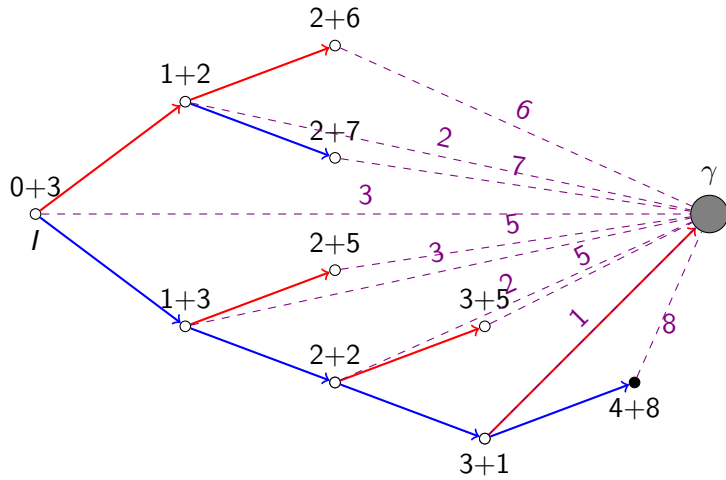
A* example

Example



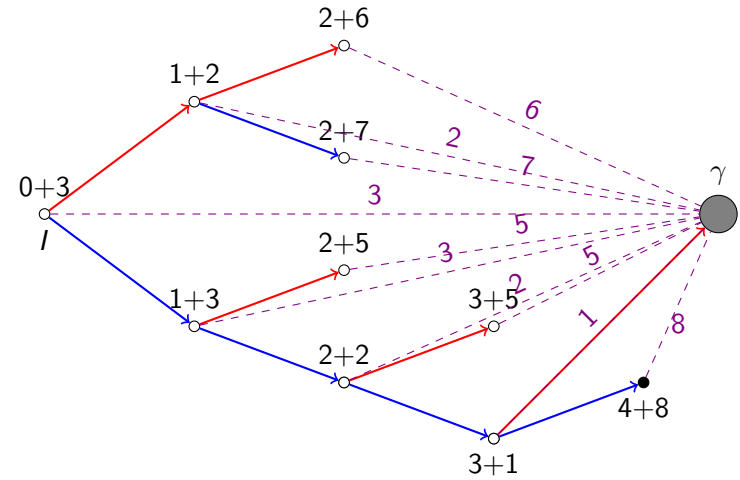
A* example

Example



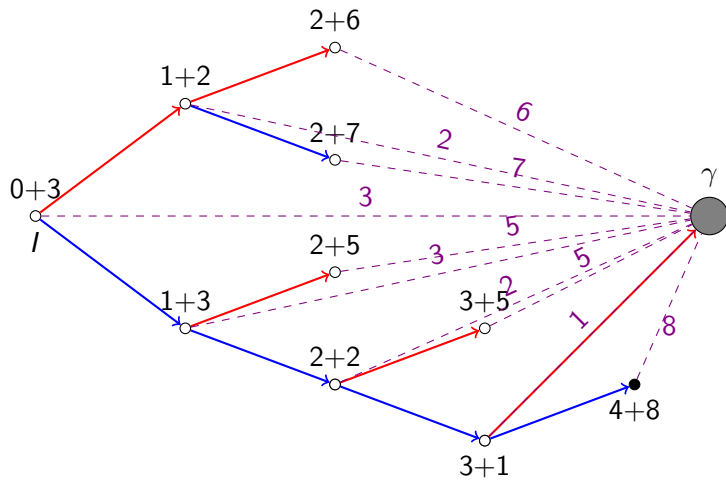
A* example

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 (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 $W \rightarrow \infty$, behaves like greedy best-first search

Properties:

- ▶ one of the 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  $\langle o, s \rangle \in \text{succ}(\text{state}(\sigma))$ :
         $\sigma'$  := make-node( $\sigma, o, s$ )
        queue.push-back( $\sigma'$ )
  fail

```

\rightsquigarrow 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

Summary

- ▶ distinguish: **planning states**, **search states**, **search nodes**
 - ▶ **planning state**: situation in the world modelled by the task
 - ▶ **search state**: subproblem remaining to be solved
 - ▶ In **state-space search** (usually progression search), planning states and search states are identical.
 - ▶ In regression search, search states usually describe sets of states ("subgoals").
 - ▶ **search node**: search state + info on "how we got there"
- ▶ search algorithms mainly differ in **order of node expansion**
 - ▶ **uninformed** vs. **informed** (**heuristic**) search
 - ▶ **local** vs. **systematic** search

Summary (ctd.)

- ▶ **heuristics**: estimators for "distance to goal node"
 - ▶ usually: the more accurate, the better performance
 - ▶ desiderata: **safe**, **goal-aware**, **admissible**, **consistent**
 - ▶ the ideal: **perfect heuristic** h^*
- ▶ most common algorithms for **satisficing planning**:
 - ▶ **greedy best-first search**
 - ▶ **weighted A***
 - ▶ **enforced hill-climbing**
- ▶ most common algorithm for **optimal planning**:
 - ▶ **A***