CSC2542
Search+ for Planning
Sheila McIlraith
Department of Computer Science
Winter, 2019
Acknowledgements

Some of the slides used in this course originate with (or are modifications of) slides from colleagues at other universities. I’d like to gratefully acknowledge their contributions and thank them for generously sharing their presentation materials. These include:

- Malte Helmert
- Gabriele Roeger
- Thomas Keller
- Bernard Nebel
- Jussi Rintanen
- P@trick Haslum

The slides in this deck are from Malte Helmert.
Outline

1. Introduction to search algorithms for planning
   - Search nodes & search states
     - Search for planning
     - Common procedures for search algorithms

2. Uninformed search algorithms

3. Heuristic search algorithms
   - Heuristics: definition and properties
   - Systematic heuristic search algorithms
   - Heuristic local search algorithms
Selecting the Right Planning Approach

Choices to make:

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

Today, we describe some popular and/or representative search algorithms, and (the basics of) how they apply to planning.

Most of the search material is covered in the textbook: Russell & Norvig: AI a Modern Approach, if you wish a further reference.
Search states vs. search nodes

In search, one distinguishes:

- **search states** $s \mapsto$ states (vertices) of the transition system
- **search nodes** $\sigma \mapsto$ 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 $\sigma$, but typical information includes:

- **state($\sigma$)**: associated search state
- **parent($\sigma$)**: pointer to search node from which $\sigma$ is reached
- **action($\sigma$)**: an action/operator leading from $\text{state}(\text{parent}(\sigma))$ to $\text{state}(\sigma)$
- **$g(\sigma)$**: cost of $\sigma$ (length of path from the root node)

For the root node, **parent($\sigma$)** and **action($\sigma$)** are undefined.
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.
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).
1 Introduction to search algorithms for planning
   - Search nodes & search states
   - Search for planning
   - Common procedures for search algorithms

2 Uninformed search algorithms

3 Heuristic search algorithms
   - Heuristics: definition and properties
   - Systematic heuristic search algorithms
   - Heuristic local search algorithms
Let \( \Pi = \langle A, I, O, G \rangle \) be a planning task.

**Search space for progression search**

<table>
<thead>
<tr>
<th>states: all states of ( \Pi ) (assignments to ( A ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>init() = ( I )</td>
</tr>
<tr>
<td>succ(s) = { \langle o, s' \rangle \mid o \in O, s' = app_o(s) }</td>
</tr>
<tr>
<td>is-goal(s) = \begin{cases} true &amp; \text{if } s \models G \ false &amp; \text{otherwise} \end{cases}</td>
</tr>
</tbody>
</table>

where \( app_o(s) \) refers to the state resulting from applying operator \( o \) in state \( s \).

* **Note the unfortunate choice of \( A \) to denote the set of atomic propositions, rather than the set of actions.**
Let \( \langle A, I, O, G \rangle \) be a planning task.

### Search space for regression search

**states:** all formulae over \( A \)

- \( \text{init}() = G \)
- \( \text{succ}(\phi) = \langle o, \phi' \rangle \mid o \in O, \phi' = \text{regr}_o(\phi), \phi' \text{ is satisfiable} \)

(modified if splitting is used)

- \( \text{is-goal}(\phi) = \begin{cases} 
  \text{true} & \text{if } I \models \phi \\
  \text{false} & \text{otherwise}
\end{cases} \)

where \( \text{regr}_o(\phi) \) refers to the formula resulting from regressing \( \phi \) over operator \( o \).

Recall that when regressing the search node is only a partial state, often compactly represented by a formula, e.g. \( \phi \).
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)
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.

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.
1. **Introduction to search algorithms for planning**
   - Search nodes & search states
   - Search for planning
   - Common procedures for search algorithms

2. **Uninformed search algorithms**

3. **Heuristic search algorithms**
   - Heuristics: definition and properties
   - Systematic heuristic search algorithms
   - Heuristic local 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.
make-root-node: Create a search node without parent.

```
def make-root-node(s):
    σ := new node
    state(σ) := s
    parent(σ) := undefined
    action(σ) := undefined
    g(σ) := 0
    return σ
```
**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(σ, o, s):
  σ′ := new node
  state(σ′) := s
  parent(σ′) := σ
  action(σ′) := o
  g(σ′) := g(σ) + 1
  return σ′
```
**Procedure extract-solution**

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

```python
def extract-solution(σ):
    solution := new list
    while parent(σ) is defined:
        solution.push-front(action(σ))
        σ := parent(σ)
    return solution
```
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
1 Introduction to search algorithms for planning
   - Search nodes & search states
   - Search for planning
   - Common procedures for search algorithms

2 Uninformed search algorithms

3 Heuristic search algorithms
   - Heuristics: definition and properties
   - Systematic heuristic search algorithms
   - Heuristic local search algorithms
Stepping back for a moment, let’s recall why we’re interested in heuristic search:

- The primary revolution in automated planning in the last decade has been the impressive scale-up in planner performance.
- Most of the gains have been as a direct result of the invention and deployment of powerful reachability heuristics.
- Most, if not all of these reachability heuristics were based on (or can be recast in terms of) the planning graph data structure, and were performed over a relaxation of the planning graph. Planning graphs are a cheap means to obtain informative look-ahead heuristics for search.
- Since their development, these so-called Relaxed Planning Graph (RPG) heuristics have been used for a variety of different types of planners, though the most noteworthy are forward search satisficing or optimizing planners.
Ghallab and Laruelle, 1994 used reachability heuristics for action selection in their partial order planners, IxTeT.

McDermott 1996, 1999 rediscovered the notion of reachability heuristics in the context of UNPOP. UNPOP showed impressive performance, for its day.

Bonet and Geffner, 1999 HSP

Hoffmann and Nebel, 2000 Fast-Forward (FF)

Helmert and Richter, 2004 Fast Downward

Wah, Hsu, Chen, and Huang, 2006 SGPlan

Baier, Bacchus and McIlraith, 2006 HPlan-P

Richter and Westphal, 2008 LAMA

We will return to the details of how these more recent planners achieved their great success, after first reviewing the underlying principles of heuristic search for planning.
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 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

- **Init**
- Distance estimate
- Distance estimate
- Distance estimate
- Distance estimate

- **Goal**
A **heuristic search algorithm** requires one more operation in addition to the definition of a search space.

**Definition (heuristic function)**

Let $\Sigma$ 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 $\sigma$. It is supposed to estimate the distance from $\sigma$ 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 (\( \equiv \) 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*\(^{\dagger}\), we can prove strong formal relationships between properties of \( h \) and properties of the algorithm (optimality, dominance, run-time for bounded error, \( \ldots \))

- For other search algorithms, “it works well in practice” is often as good an analysis as one gets.
Most texts apply heuristic functions to states, not nodes.

This is slightly less general than the definition here:

Given a state heuristic $h$, we can define an equivalent node heuristic as $h'(\sigma) := h(state(\sigma))$.

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 $state(\sigma)$ pseudo-heuristics.

In practice there are sometimes good reasons to have the heuristic value depend on the generating path of $\sigma$ (e.g., the landmark pseudo-heuristic, Richter et al. 2008).
Let $\Sigma$ 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 $\sigma$ 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 $\sigma$. 
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 $\sigma'$ is a successor of $\sigma$
Outline

1. Introduction to search algorithms for planning
   - Search nodes & search states
   - Search for planning
   - Common procedures for search algorithms

2. Uninformed search algorithms

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

- greedy best-first search
- A*
- weighted A*
Greedy best-first search

Greedy best-first search (with duplicate detection)

\[
\begin{align*}
open & := \text{new min-heap ordered by } (\sigma \mapsto h(\sigma)) \\
open.\text{insert}(\text{make-root-node}(\text{init}())) \\
closed & := \emptyset \\
\text{while not } open.\text{empty}(): \\
& \quad \sigma = open.\text{pop-min}() \\
& \quad \text{if } \text{state}(\sigma) \notin closed: \\
& \quad \quad closed := closed \cup \{\text{state}(\sigma)\} \\
& \quad \quad \text{if } \text{is-goal}(\text{state}(\sigma)): \\
& \quad \quad \quad \text{return } \text{extract-solution}(\sigma) \\
& \quad \quad \text{for each } \langle o, s \rangle \in \text{succ}(\text{state}(\sigma)):\ \\
& \quad \quad \quad \sigma' := \text{make-node}(\sigma, o, s) \\
& \quad \quad \quad \text{if } h(\sigma') < \infty: \\
& \quad \quad \quad \quad open.\text{insert}(\sigma') \\
\text{return unsolvable}
\end{align*}
\]
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* is a best-first search algorithm

- it uses a *distance-plus-cost* heuristic function,
  \[ f(x) = g(x) + h(x), \]
  where
  - \( g(x) \) is the cost from the starting node to the current node,
  - \( h(x) \) is the estimated distance to the goal.

- \( h(x) \) is generally admissible – it must not overestimate the distance to the goal. As such, A* can be shown to yield the optimal solution.
A* Algorithm

A* (with duplicate detection and reopening)

\[
\begin{align*}
open & := \textbf{new} \text{ min-heap ordered by } (\sigma \mapsto g(\sigma) + h(\sigma)) \\
open.\text{insert}(\text{make-root-node}(\text{init}())) \\
closed & := \emptyset \\
distance & := \emptyset \\
\textbf{while not } open.\text{empty}(): \\
\quad \sigma & = open.\text{pop-min}() \\
\quad \textbf{if } state(\sigma) \notin closed \textbf{ or } g(\sigma) < distance(state(\sigma)):\ \\
\quad \quad closed & := closed \cup \{state(\sigma)\} \\
\quad \quad distance(\sigma) & := g(\sigma) \\
\quad \textbf{if } is\text{-goal}(state(\sigma)):\ \\
\quad \quad \quad \textbf{return} \ \text{extract-solution}(\sigma) \\
\quad \textbf{for each } \langle o, s \rangle \in \text{succ}(state(\sigma)):\ \\
\quad \quad \quad \sigma' & := \text{make-node}(\sigma, o, s) \\
\quad \quad \quad \textbf{if } h(\sigma') < \infty: \\
\quad \quad \quad \quad open.\text{insert}(\sigma') \\
\textbf{return} \ \text{unsolvable}
\end{align*}
\]
A* example

Example

0 + 3

I

3

G
A* example

Example

Diagram:

- Node I
- Node G
- Edges labeled with numbers:
  - 0+3 from I to another node
  - 1+2 from another node to I
  - 1+3 from I to another node
  - 2 from I to G
  - 3 from another node to G
A* example

Example

![Graph Diagram]

- **I** (Initial State)
- **G** (Goal State)
- Nodes and Edges with Labels:
  - 0 + 3
  - 1 + 2
  - 2 + 6
  - 2 + 7
  - 3
  - 6
  - 7
A* example

Example

Diagram of an A* search algorithm example.
A* example

Example

Diagram with nodes and edges labeled with values.
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 $\sigma$ 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
Weighted A* (with duplicate detection and reopening)

\[ \text{open} := \text{new min-heap ordered by } (\sigma \mapsto g(\sigma) + W \cdot h(\sigma)) \]

\[ \text{open}.\text{insert}(\text{make-root-node}(\text{init}())) \]

\[ \text{closed} := \emptyset \]

\[ \text{distance} := \emptyset \]

\[ \text{while not } \text{open}.\text{empty}() : \]

\[ \sigma = \text{open}.\text{pop-min}() \]

\[ \text{if } \text{state}(\sigma) \notin \text{closed} \text{ or } g(\sigma) < \text{distance}(\text{state}(\sigma)) : \]

\[ \text{closed} := \text{closed} \cup \{ \text{state}(\sigma) \} \]

\[ \text{distance}(\sigma) := g(\sigma) \]

\[ \text{if } \text{is-goal}(\text{state}(\sigma)) : \]

\[ \text{return } \text{extract-solution}(\sigma) \]

\[ \text{for each } \langle o, s \rangle \in \text{succ}(\text{state}(\sigma)) : \]

\[ \sigma' := \text{make-node}(\sigma, o, s) \]

\[ \text{if } h(\sigma') < \infty : \]

\[ \text{open}.\text{insert}(\sigma') \]

\[ \text{return unsolvable} \]
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
1. Introduction to search algorithms for planning
   - Search nodes & search states
   - Search for planning
   - Common procedures for search algorithms

2. Uninformed search algorithms

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

- hill-climbing
- enforced hill-climbing
Hill-climbing

\[ \sigma := \text{make-root-node}(\text{init}()) \]

\textbf{forever}:

\textbf{if} \ is\text{-}goal(state(\sigma)):

\textbf{return} extract\text{-}solution(\sigma)

\[ \Sigma' := \{ \text{make\text{-}node}(\sigma, o, s) \mid \langle o, s \rangle \in \text{succ}(\text{state}(\sigma)) \} \]

\sigma := \text{an element of} \ \Sigma' \ \text{minimizing} \ h \ \text{(random tie breaking)}

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

\[ \sigma := \text{make-root-node}(\text{init}()) \]

while not \( \text{is-goal}(\text{state}(\sigma)) \):
    \[ \sigma := \text{improve}(\sigma) \]

return \( \text{extract-solution}(\sigma) \)

- one of the three most commonly used algorithms for satisficing planning
- can fail if procedure improve fails (when the goal is unreachable from \( \sigma_0 \))
Enforced hill-climbing: procedure `improve`

```python
def improve(σ₀):
    queue := new fifo-queue
    queue.push-back(σ₀)
    closed := ∅
    while not queue.empty():
        σ = queue.pop-front()
        if state(σ) ∉ closed:
            closed := closed ∪ {state(σ)}
            if h(σ) < h(σ₀):
                return σ
            for each ⟨o, s⟩ ∈ succ(state(σ)):
                σ' := make-node(σ, o, s)
                queue.push-back(σ')
    fail
```

⇒ breadth-first search for more promising node than σ₀
KEEP CALM AND MAKE A PLAN