Automated (AI) Planning

Autonomous Systems

Carmel Domshlak

What is AI?

Two of somewhat more pragmatic attempts

The study of mental faculties through the use of computational models.
(E. Charniak & D. McDermott)

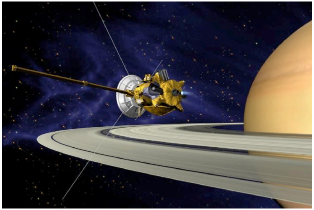
The science concerned with understanding intelligent behavior by attempting to create it in the artificial.
(T. Smithers)

- Intelligent behavior can be considered (postulated?) as ability to solve problems for which the machine has no knowledge of an suitable algorithm

NASA Experience

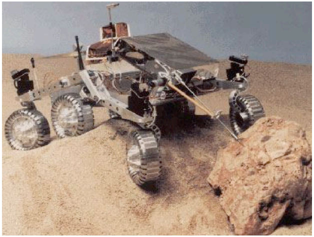
Galileo Jupiter or Cassini Saturn missions

- $1G budget
- Ground crew of 100-300 personnel



Mars micro-rover Sojourne

- $100M budget
- Small (and tired!) ground teams



Sojourne operated for two month, but future robots are expected to operate **much** longer!

NASA Vision

Space-exploring systems should be

- Low-cost and rapid development, low-cost control
- Autonomous operation for long periods of time
- Autonomous operation must guarantee success, given tight deadlines and resource constraints

Utopy?

NASA Vision

Space-explorating systems should be

- Low-cost and rapid development, low-cost control
- Autonomous operation for long periods of time
- Autonomous operation must guarantee success, given tight deadlines and resource constraints

Utopy? Not really. First progress in this direction has been accomplished in 1998 in the scope of the Deep Space One project!

Planning Problems

A sample of problems:

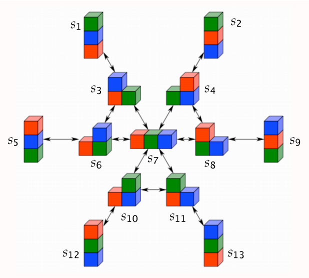
- Solving Rubik's cube (or 15-puzzle, or ...)
- Selecting and ordering movements of an elevator or a crane
- Scheduling of production lines
- Autonomous robots
- Crisis management
- ...

What is in common?

Planning Problems

What is in common?

- All these problems deal with action selection or control
- Some notion of problem state
- (Often) specification of initial state and/or goal state
- Legal moves or actions that transform states into other state



Planning Problems

For now focus on:

- Plans (aka solutions) are sequences of moves that transform the initial state into the goal state
- Intuitively, not all solutions are equally desirable

What is our task?

1. Find out whether there is a solution
2. Find any solution
3. Find an optimal (or near-optimal) solution
4. Fixed amount of time, find best solution possible
5. Find solution that satisfy property $\aleph$ (what is $\aleph$? you choose!)

Planning Problems

What is our task?

1. Find out whether there is a solution
2. Find any solution
3. Find an optimal (or near-optimal) solution
4. Fixed amount of time, find best solution possible
5. Find solution that satisfy property $\aleph$ (what is $\aleph$? you choose!)

- ♠ While all these tasks sound related, they are *very different*. The techniques best suited for each one are almost disjoint.
- In AI planning, (1) is usually assumed not to be an issue. (In contrast, in formal verification this is the central issue.)

Planning and Action Selection in AI

Three approaches in AI (*in general?*) to the problems of action selection or control

- *Learning*: learn control from experience
- *Programming*: specify control by hand
- *Planning*: specify problem by hand, derive control automatically

All three have strengths and weaknesses; approaches not exclusive and often complementary.

Planning is a form of **general problem solving**

Three Key Ingredients of Planning

... and of AI approach to problems in general?

Planning is a form of general problem solving

$$\texttt{Problem} \Longrightarrow \texttt{Language} \Longrightarrow \texttt{Planner} \Longrightarrow \texttt{Solution}$$

1. **models** for defining, classifying, and understanding problems
   - what is a *planning problem*
   - what is a *solution* (*plan*), and
   - what is an *optimal solution*
2. **languages** for representing problems
3. **algorithms** for solving them

Three Key Ingredients of Planning

... and of AI approach to problems in general?

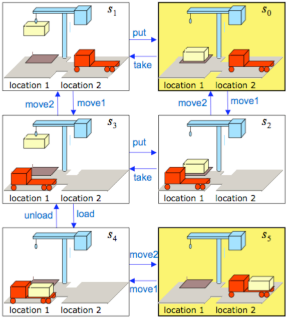
Planning is a form of general problem solving

$$\texttt{Problem} \Longrightarrow \texttt{Language} \Longrightarrow \texttt{Planner} \Longrightarrow \texttt{Solution}$$

1. **models** for defining, classifying, and understanding problems
   - what is a *planning problem*
   - what is a *solution* (*plan*), and
   - what is an *optimal solution*
2. **languages** for representing problems
3. **algorithms** for solving them

State model for Classical AI Planning

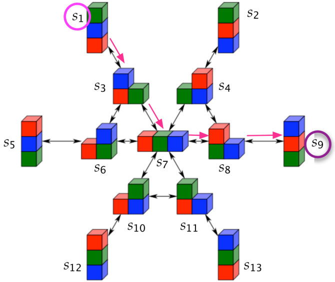
- finite state space S
- an initial state $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- applicable actions $A(s) \subseteq A$ for $s \in S$
- a transition function $s' = f(a, s)$ for $a \in A(s)$
- a cost function $c : A^* \longrightarrow [0, \infty)$



A **solution** is a sequence of applicable actions that maps s_0 into S_G
An **optimal solution** minimizes c

Why planning is difficult?

- Solutions to planning problems are paths from an initial state to a goal state in the transition graph
- Dijkstra's algorithm solves this problem in $O(|V|\log(|V|)+|E|)$
- Can we go home??

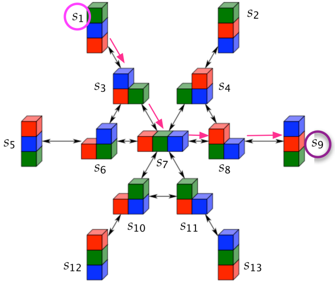


Why planning is difficult?

- Solutions to planning problems are paths from an initial state to a goal state in the transition graph
- Dijkstra's algorithm solves this problem in $O(|V| \log(|V|) + |E|)$
- Can we go home??

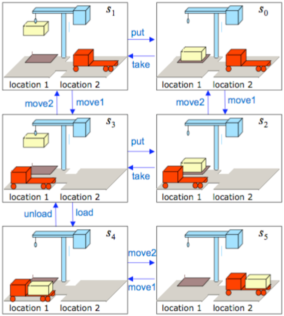
♠ Not exactly ⇒ $|V|$ of our interest is 10^{10}, 10^{20}, 10^{100}, ...

- *But do we need such values of* $|V|$ *?!*



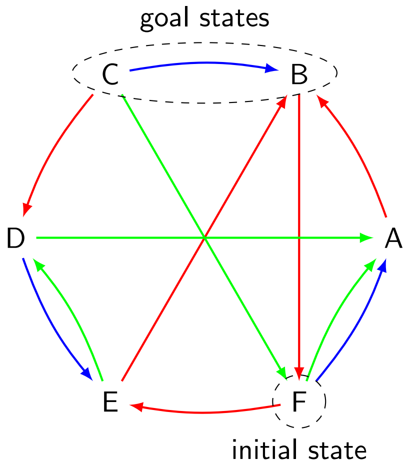
Why planning is difficult?

- Generalize the earlier example:
  - Five locations, three robot carts, 100 containers, three piles
  - $|V| \approx 10^{277}$
- The number of atoms in the universe is only about 10^{87}
  - The state space in our example is more than 10^{109} times as large (uppss ...)



And solving such a problem is not hopeless!

Transition systems



Transition systems

Formalization of the dynamics of the world/application

Definition (transition system)

A transition system is $\langle S, I, \{a_1, \ldots, a_n\}, G \rangle$ where

- S is a finite set of states (the state space),
- $I \subseteq S$ is a finite set of initial states,
- every action $a_i \subseteq S \times S$ is a binary relation on S,
- $G \subseteq S$ is a finite set of goal states.

Definition (applicable action)

An action a is applicable in a state s if sas' for at least one state s'.

Transition systems
Deterministic transition systems

A transition system is deterministic if there is only one initial state and all actions are deterministic. Hence all future states of the world are completely predictable.

Definition (deterministic transition system)

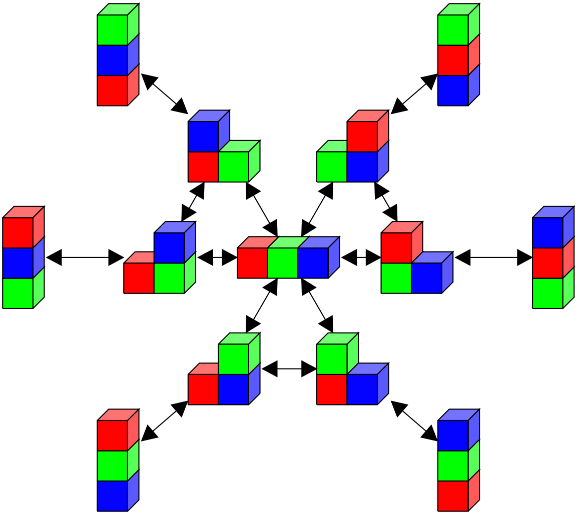
A deterministic transition system is $\langle S, I, O, G \rangle$ where

- S is a finite set of states (the state space),
- $I \in S$ is a state,
- actions $a \in O$ (with $a \subseteq S \times S$) are partial functions,
- $G \subseteq S$ is a finite set of goal states.

Successor state wrt. an action

Given a state s and an action a so that a is applicable in s, the successor state of s with respect to a is s' such that sas', denoted by $s' = app_a(s)$.

Blocks world
The transition graph for three blocks



Blocks world
Properties

blocks	states
1	1
2	3
3	13
4	73
5	501
6	4051
7	37633
8	394353
9	4596553
...	
19	1356437369358817 3

1. Finding a solution is polynomial time in the number of blocks (move everything onto the table and then construct the goal configuration).
2. Finding a shortest solution is NP-complete (for a compact description of the problem).

Deterministic planning: plans

Definition (plan)

A plan for $\langle S, I, A, G \rangle$ is a sequence $\pi = a_1, \ldots, a_n$ of action instances such that $a_1, \ldots, a_n \in A$ and $s_0, \ldots, s_n$ is a sequence of states (the execution of π) so that

1. $s_0 = I$,
2. $s_i = app_{a_i}(s_{i-1})$ for every $i \in \{1, \ldots, n\}$, and
3. $s_n \in G$.

This can be equivalently expressed as

$$app_{a_n}(app_{a_{n-1}}(\ldots app_{a_1}(I) \ldots)) \in G$$

Three Key Ingredients of Planning
… and of AI approach to problems in general?

Planning is a form of general problem solving

$$\texttt{Problem} \Longrightarrow \texttt{Language} \Longrightarrow \texttt{Planner} \Longrightarrow \texttt{Solution}$$

1. **models** for defining, classifying, and understanding problems
   - what is a *planning problem*
   - what is a *solution* (*plan*), and
   - what is an *optimal solution*
2. **languages** for representing problems
3. **algorithms** for solving them

Succinct representation of transition systems

- More compact representation of actions than as relations is often
  - possible because of symmetries and other regularities,
  - unavoidable because the relations are too big.
- Represent different aspects of the world in terms of different state variables. $\rightsquigarrow$ A state is a valuation of state variables.
- Represent actions in terms of changes to the state variables.

State variables

- The state of the world is described in terms of a finite set of finite-valued state variables.

Example

hour: $\{0, \ldots, 23\} = 13$
minute: $\{0, \ldots, 59\} = 55$
location: $\{51, 52, 82, 101, 102\} = 101$
weather: $\{\text{sunny}, \text{cloudy}, \text{rainy}\} = \text{cloudy}$
holiday: $\{\mathsf{T}, \mathsf{F}\} = \mathsf{F}$

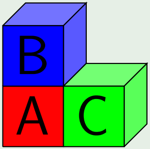
- Any n-valued state variable can be replaced by $\lceil \log_2 n \rceil$ Boolean (2-valued) state variables.
- Actions change the values of the state variables.

Blocks world with state variables

State variables:
location-of-A: {B, C, table}
location-of-B: {A, C, table}
location-of-C: {A, B, table}

Example

$s(\textit{location-of-A}) = \text{table}$
$s(\textit{location-of-B}) = \text{A}$
$s(\textit{location-of-C}) = \text{table}$

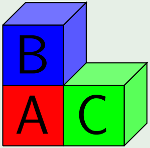


Not all valuations correspond to an intended blocks world state, e. g. s such that $s(\textit{location-of-A}) = \text{B}$ and $s(\textit{location-of-B}) = \text{A}$.

Blocks world with Boolean state variables

Example

$$
\begin{aligned}
s(\textit{A-on-B}) &= 0\\
s(\textit{A-on-C}) &= 0\\
s(\textit{A-on-table}) &= 1\\
s(\textit{B-on-A}) &= 1\\
s(\textit{B-on-C}) &= 0\\
s(\textit{B-on-table}) &= 0\\
s(\textit{C-on-A}) &= 0\\
s(\textit{C-on-B}) &= 0\\
s(\textit{C-on-table}) &= 1
\end{aligned}
$$



Deterministic planning tasks

Definition (deterministic planning task)

A deterministic planning task is a 4-tuple $\Pi = \langle V, I, A, G \rangle$ where

- V is a finite set of state variables,
- I is an initial state over V,
- A is a finite set of actions over V, and
- G is a constraint ($=$ formula) over V describing the goal states.

Notes:

- Unless stated otherwise, G will be a single partial assignment to V
- We will omit the word "deterministic" where it is clear from context.

Mapping planning tasks to transition systems

From every deterministic planning task $\Pi = \langle V, I, A, G \rangle$ we can produce a corresponding transition system $\mathcal{T}(\Pi) = \langle S, I, A', G' \rangle$:

1. S is the set of all valuations of V,
2. $A' = \{R(a) \mid a \in A\}$ where $R(a) = \{(s, s') \in S \times S \mid s' = \mathit{app}_a(s)\}$, and
3. $G' = \{s \in S \mid s \models G\}$.

Planning Languages

Key issue

Models represented **implicitly** in a **declarative language**

Play two roles

- **specification**: concise model description
- **computation**: reveal useful info about problem's *structure*

The SAS Language

A problem in SAS is a tuple $\langle V, A, I, G \rangle$

- V is a finite set of state variables with finite domains $dom(v_i)$
- I is an initial state over V
- G is a partial assignment to V
- A is a finite set of actions a specified via $\mathsf{pre}(a)$ and $\mathsf{eff}(a)$, both being partial assignments to V

- An action a is applicable in a state $s \in dom(V)$ iff $s[v] = \mathsf{pre}(a)[v]$ whenever $\mathsf{pre}(a)[v]$ is specified
- Applying an applicable action a changes the value of each variable v to $\mathsf{eff}(a)[v]$ if $\mathsf{eff}(a)[v]$ is specified.
- Example: 8-puzzle

The STRIPS language
Useful fragment of SAS

A problem in STRIPS is a tuple $\langle P, A, I, G \rangle$

- P stands for a finite set of **atoms** (boolean vars)
- $I \subseteq P$ stands for **initial situation**
- $G \subseteq P$ stands for **goal situation**
- A is a finite set of **actions** a specified via pre(a), add(a), and del(a), all subsets of P

- States are collections of atoms
- An action a is applicable in a state s iff pre$(a) \subseteq s$
- Applying an applicable action a at s results in $s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$

Why STRIPS is interesting

- STRIPS operators are particularly simple, yet expressive enough to capture general planning problems.
- In particular, STRIPS planning is no easier than general planning problems.
- Many algorithms in the planning literature are easier to present in terns of STRIPS .

Three Key Ingredients of Planning

... and of AI approach to problems in general?

Planning is a form of general problem solving

$$\texttt{Problem} \Longrightarrow \texttt{Language} \Longrightarrow \texttt{Planner} \Longrightarrow \texttt{Solution}$$

1. **models** for defining, classifying, and understanding problems
2. **languages** for representing problems
3. **algorithms** for solving them
   - NEXT: algorithms for **classical planning**
     where a significant progress has been recently achieved

More on the Motivation

Planning is a form of general problem solving

$$\texttt{Problem} \Longrightarrow \texttt{Language} \Longrightarrow \texttt{Planner} \Longrightarrow \texttt{Solution}$$

Modeling Time vs. Solution Time and Quality

- specialized methods are typically more efficient (though even that is not necessarily correct), but tend to require lots of programming
- goal in AI problem solving is to **facilitate modeling** and yet provide **efficient solutions**
- this involves **general languages** (*a la* SAS or STRIPS) and thus **language-specific algorithms**

State-space search

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

Satisficing or optimal planning?

Must carefully distinguish two different problems:

- satisficing planning: any solution is OK (although shorter solutions typically preferred)
- optimal planning: plans must have shortest possible length

Both are often solved by search, but:

- details are very different
- almost no overlap between good techniques for satisficing planning and good techniques for optimal planning
- many problems that are trivial for satisficing planners are impossibly hard for optimal planners

Planning by state-space search

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

Choice 1: Search direction

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

Planning by state-space search

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

Choice 2: Search space representation

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

Planning by state-space search

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

Choice 3: Search algorithm

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

Planning by state-space search

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

Choice 4: Search control

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

Planning by forward search: progression

Progression: Computing the successor state $app_o(s)$ of a state s with respect to an operator o.

Progression planners find solutions by forward search:

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

pro: very easy and efficient to implement

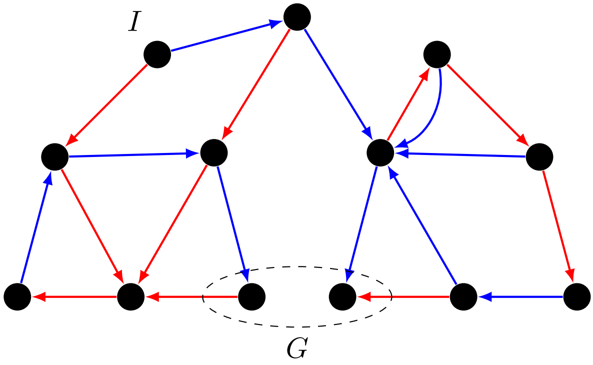
Search space representation in progression planners

Two alternative search spaces for progression planners:

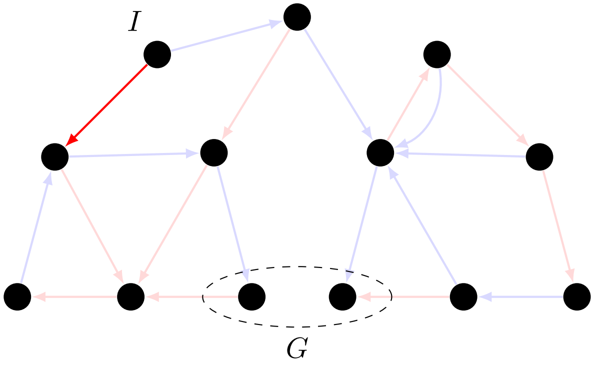
1. search nodes correspond to states
   - when the same state is generated along different paths, it is not considered again (duplicate detection)
   - pro: fast
   - con: memory intensive (must maintain closed list)
2. search nodes correspond to operator sequences
   - different operator sequences may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

$\rightsquigarrow$ first alternative usually preferable

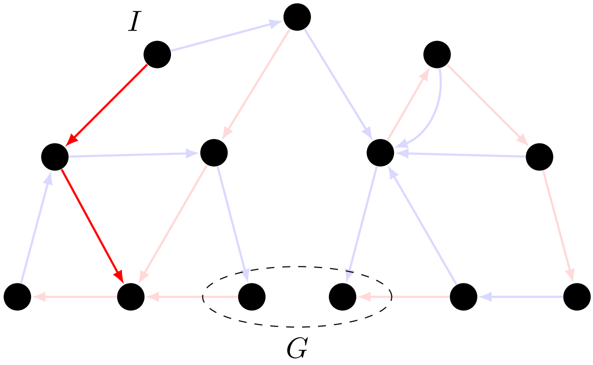
Progression planning example (depth-first search)



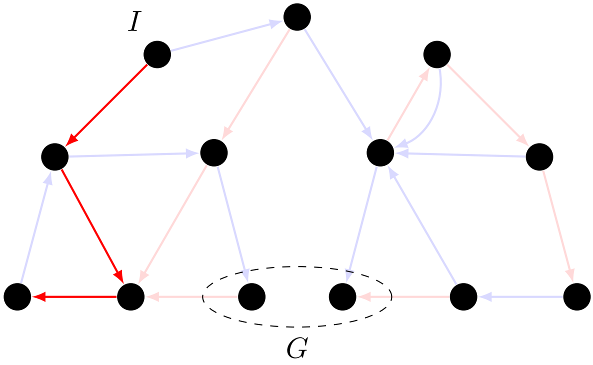
Progression planning example (depth-first search)



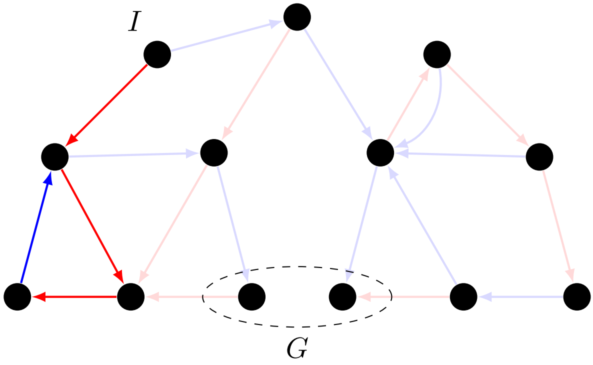
Progression planning example (depth-first search)



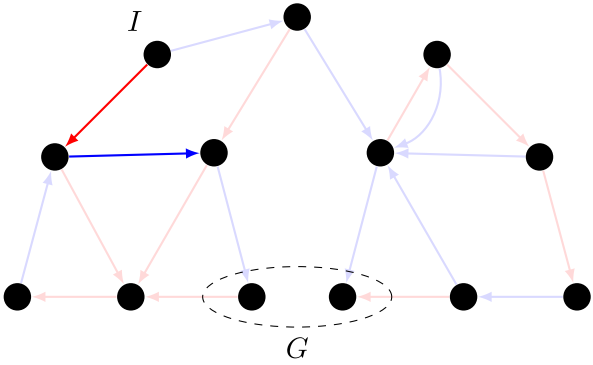
Progression planning example (depth-first search)



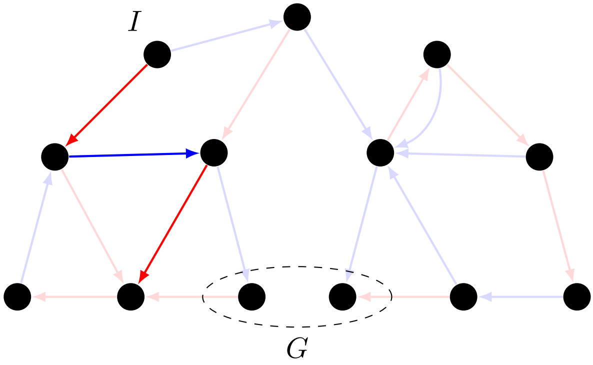
Progression planning example (depth-first search)



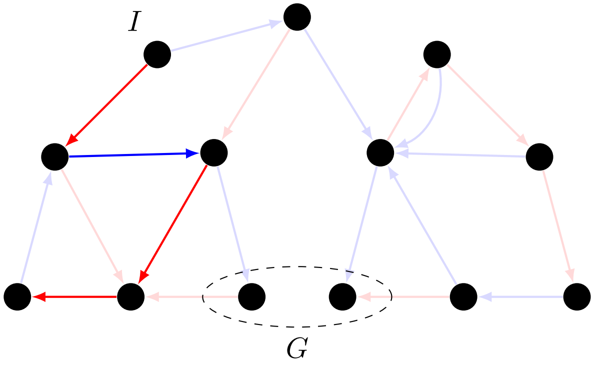
Progression planning example (depth-first search)



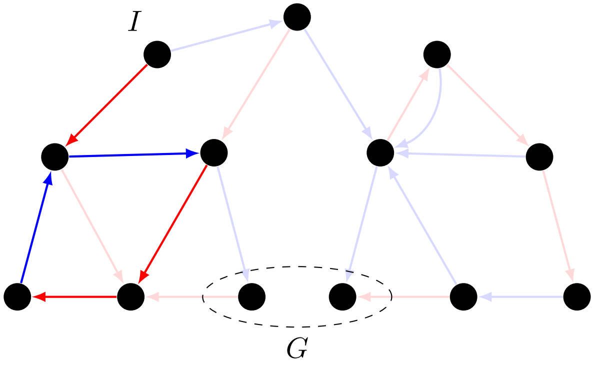
Progression planning example (depth-first search)



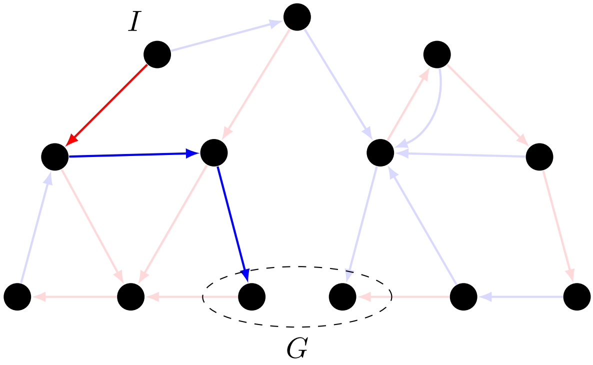
Progression planning example (depth-first search)



Progression planning example (depth-first search)



Progression planning example (depth-first search)



Forward search vs. backward search

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

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

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

Search

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

Required ingredients for search

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

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

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

Search for planning: progression

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

Search space for progression search

states: all states of Π (assignments to V)

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

Classification of search algorithms

uninformed search vs. heuristic search:

- uninformed search algorithms only use the basic ingredients for general search algorithms
- heuristic search algorithms additionally use heuristic functions which estimate how close a node is to the goal

systematic search vs. local search:

- systematic algorithms consider a large number of search nodes simultaneously
- local search algorithms work with one (or a few) candidate solutions (search nodes) at a time
- not a black-and-white distinction; there are crossbreeds (e. g., enforced hill-climbing)

Classification: what works where in planning?

uninformed vs. heuristic search:

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

systematic search vs. local search:

- For satisficing planning, the most successful algorithms are somewhere between the two extremes.
- For optimal planning, systematic algorithms are required.

Uninformed search algorithms

Less relevant for planning, yet not irrelevant

Popular uninformed systematic search algorithms:

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

Popular uninformed local search algorithms:

- random walk

Heuristic search algorithms: systematic

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

Popular systematic heuristic search algorithms:

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

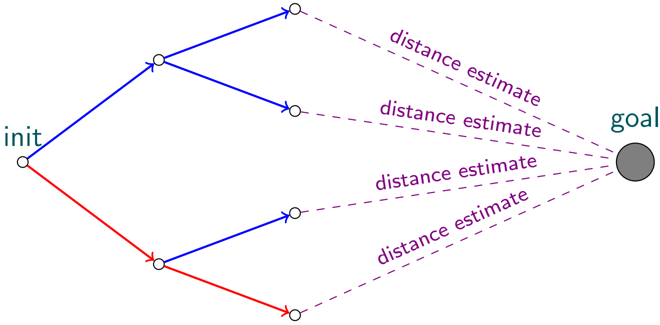
Heuristic search algorithms: local

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

Popular heuristic local search algorithms:

- hill-climbing
- enforced hill-climbing
- beam search
- tabu search
- genetic algorithms
- simulated annealing
- ...

Heuristic search: idea



Required ingredients for heuristic search

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

Definition (heuristic function)

Let Σ be the set of nodes of a given search space.
A heuristic function or heuristic (for that search space) is a function $h : \Sigma \to \mathbb{N}_0 \cup \{\infty\}$.

The value $h(\sigma)$ is called the heuristic estimate or heuristic value of heuristic h for node σ. It is supposed to estimate the distance from σ to the nearest goal node.

What exactly is a heuristic estimate?

What does it mean that h "estimates the goal distance"?

- For most heuristic search algorithms, h does not need to have any strong properties for the algorithm to work ($=$ be correct and complete).
- However, the efficiency of the algorithm closely relates to how accurately h reflects the actual goal distance.
- For some algorithms, like A*, we can prove strong formal relationships between properties of h and properties of the algorithm (optimality, dominance, run-time for bounded error, ...)
- For other search algorithms, "it works well in practice" is often as good an analysis as one gets.

Perfect heuristic

Let Σ be the set of nodes of a given search space.

Definition (optimal/perfect heuristic)

The optimal or perfect heuristic of a search space is the heuristic h^* which maps each search node σ to the length of a shortest path from $state(\sigma)$ 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():
    σ = *open*.pop-min()
    if *state*(σ) $\notin$ *closed*:
        closed := *closed* $\cup$ {*state*(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each $\langle o, s \rangle \in$ succ(*state*(σ)):
            σ' := make-node(σ, o, s)
            if $h(\sigma') < \infty$:
                open.insert(σ')
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*

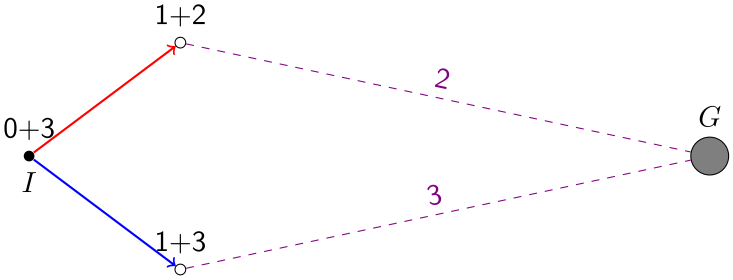
A* (with duplicate detection and reopening)

```
open := new min-heap ordered by (σ ↦ g(σ) + h(σ))
open.insert(make-root-node(init()))
closed := ∅
distance := ∅
while not open.empty():
    σ = open.pop-min()
    if state(σ) ∉ closed or g(σ) < distance(state(σ)):
        closed := closed ∪ {state(σ)}
        distance(σ) := g(σ)
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each ⟨o, s⟩ ∈ succ(state(σ)):
            σ′ := make-node(σ, o, s)
            if h(σ′) < ∞:
                open.insert(σ′)
return unsolvable
```

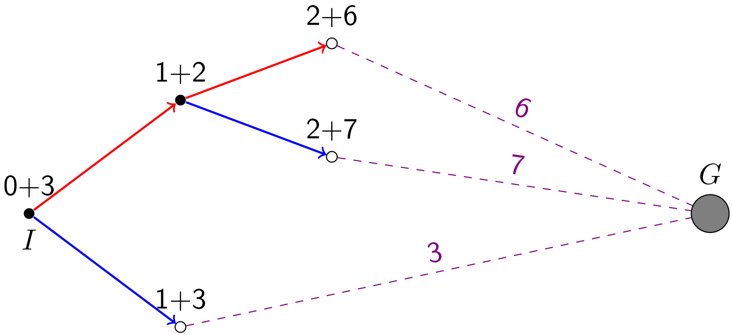
A* example
Example

A* example

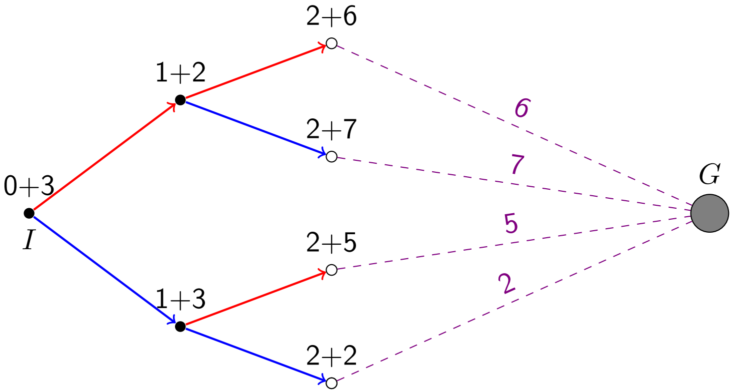
Example



A* example
Example

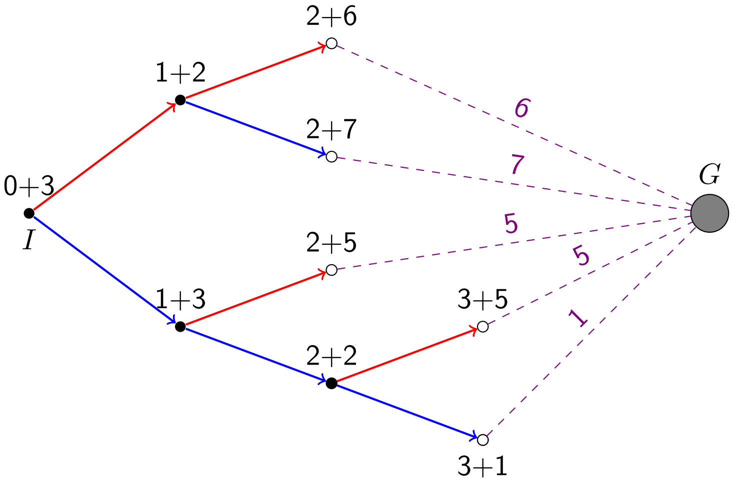


A* example
Example



A* example

Example



Terminology for A*

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

Properties of A*

- the most commonly used algorithm for optimal planning
- rarely used for satisficing planning
- complete for safe heuristics (even without duplicate detection)
- optimal if h is admissible and/or consistent (even without duplicate detection)
- never reopens nodes if h is consistent

Implementation notes:

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

Weighted A*

Weighted A* (with duplicate detection and reopening)

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

Properties of weighted A*

The weight $W \in \mathbb{R}_0^+$ is a parameter of the algorithm.

- for $W = 0$, behaves like breadth-first search
- for $W = 1$, behaves like A*
- for $W \to \infty$, behaves like greedy best-first search

Properties:

- one of the three most commonly used algorithms for satisficing planning
- for $W > 1$, can prove similar properties to A*, replacing optimal with bounded suboptimal: generated solutions are at most a factor W as long as optimal ones

Hill-climbing

Hill-climbing

$\sigma :=$ make-root-node(init())
forever:
  if is-goal(state(σ)):
    return extract-solution(σ)
  $\Sigma' := \{$ make-node(σ, o, s) $\mid \langle o, s \rangle \in$ succ(state(σ)) $\}$
  $\sigma :=$ an element of Σ' 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*(σ_0):
  queue := **new** fifo-queue
  queue.push-back(σ_0)
  closed := $\emptyset$
  while not *queue*.empty():
    σ = *queue*.pop-front()
    if *state*(σ) $\notin$ *closed*:
      closed := *closed* $\cup$ {*state*(σ)}
      if $h(\sigma) < h(\sigma_0)$:
        return σ
      for each $\langle o, s \rangle \in$ succ(*state*(σ)):
        σ' := make-node(σ, o, s)
        queue.push-back(σ')
  fail

$\rightsquigarrow$ breadth-first search for more promising node than σ_0

Enforced hill-climbing (ctd.)

Enforced hill-climbing

σ := make-root-node(init())
while not is-goal(state(σ)):
    σ := improve(σ)
return extract-solution(σ)

- 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