Automated Action Planning

Background

Carmel Domshlak

Automated Action Planning
— Background

Course prerequisites

Course prerequisites:

- computational complexity theory: decision problems, reductions, NP-completeness
- foundations of AI: search, heuristic search
- propositional logic: syntax and semantics

A VERY brief guide to NP-hardness

Imagine that ...

You just got a new job. Congratulations!

You are asked to develop an efficient algorithm for determining whether or not a given set of specifications for a new XXX component can be met, and if so, constructing a design that meets them

What is efficient: $O(poly(n))$ vs. $O(exp(n))$

	$n = 10$	20	30	40	50
n	.00001 sec	.00002 sec	.00003 sec	.00004 sec	.00005 sec
n^2	.0001 sec	.0004 sec	.0009 sec	.0016 sec	.0025 sec
n^3	.001 sec	.008 sec	.027 sec	.064 sec	.125 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min
2^n	.001 sec	1.0 sec	17.9 min	12.7 days	35.7 years
3^n	.059 sec	58 min	6.5 years	3855 cent	2×10^8 cent

A VERY brief guide to NP-hardness

Imagine that ...

You just got a new job. Congratulations!

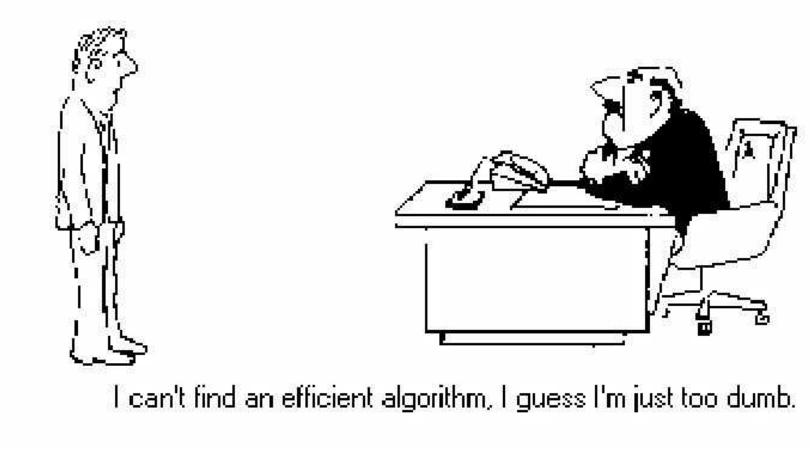
You are asked to develop an efficient algorithm for determining whether or not a given set of specifications for a new XXX component can be met, and if so, constructing a design that meets them

Bad news

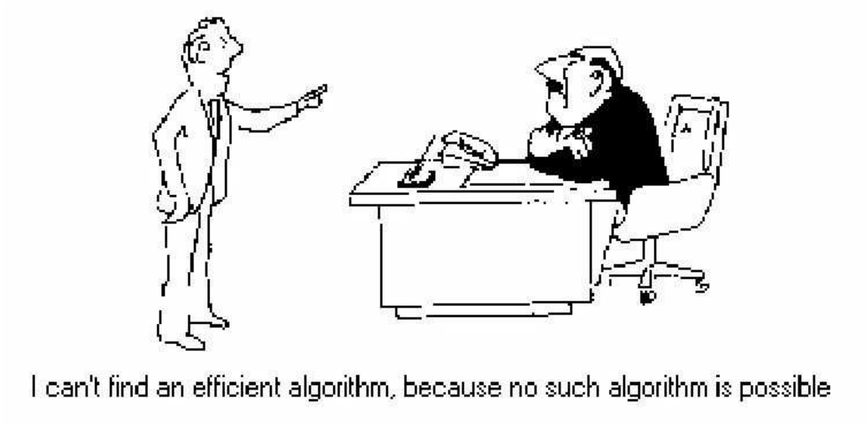
A year after you still have no algorithm that is substantially more efficient than searching through all possible designs ...

What to do?

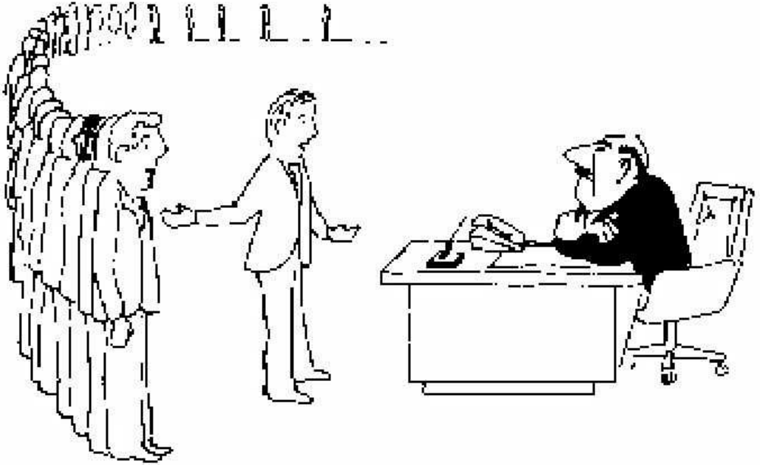
You don't want to ...



You would like to say, but ...



Today you can say



I can't find an efficient algorithm, but neither can all these famous people.

Some important complexity classes

A class of problems is in ...

- **P** if any problem in the class can be solved in polynomial time
- **NP** if, for any problem in the class, some solutions can be verified in polynomial time
- **NPC** if (informally) it is one of the "hardest" problem classes in NP

Obviously, P$\subseteq$NP, and it is not likely that P$=$NP.
However, no proof so far for P$\neq$NP!

NP-complete problem classes

Definition: NP-completeness

A decision (yes/no) problem class C is NP-complete if:

1. C is in NP, and
2. Every problem class in NP is reducible to C in polynomial time.

Definition: Reducibility

A problem class K is reducible to C if there is a polynomial-time deterministic algorithm (reduction) that transforms any problem $k \in K$ into a problem $c \in C$ such that the answer to c is Yes if and only if the answer to k is Yes.

NP-complete problem classes

Definition: NP-completeness

A decision (yes/no) problem class C is NP-complete if:

1. C is in NP, and
2. Every problem class in NP is reducible to C in polynomial time.

- To prove that an NP problem class C is NP-complete it is sufficient to show that an already known NP-complete problem class K reduces to C.
- A problem satisfying condition (2) is said to be NP-hard, whether or not it satisfies condition (1).
- Bottom line: if we had a polynomial time algorithm for C, we could solve all problems in NP in polynomial time.

Still, what to do?

The needs to solve a problem won't disappear overnight simply because the problem is known to be NP-hard, but knowing the problem is NP-complete does provide valuable information

- The search for an efficient exact algorithm should certainly be accorded "low priority"

Dealing with NP-hard problems

- Less relevant to our course:
  - approximation algorithms
  - probabilistic algorithms
- More relevant to our course:
  - efficient algorithms for interesting subclasses (special cases) of the general problem
  - relaxing the problem so that a fast algorithm will meet most of the problem's original properties
  - algorithms that do not guarantee to run quickly, but seem likely to do it "most of the time"

State-space search

- state-space search: one of the big success stories of AI
- different classes of search algorithms
  - uninformed vs. informed
  - systematic vs. local
- many planning algorithms based on state-space search
- 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, . . .

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.

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 states vs. search nodes

In search, one distinguishes:

- search states s $\rightsquigarrow$ states (vertices) of the transition system
- search nodes σ $\rightsquigarrow$ 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:

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

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

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).

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.

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):
  $\sigma :=$ **new** node
  $state(\sigma) := s$
  $parent(\sigma) :=$ undefined
  $action(\sigma) :=$ undefined
  $g(\sigma) := 0$
  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
    $state(\sigma') := s$
    $parent(\sigma') := \sigma$
    $action(\sigma') := o$
    $g(\sigma') := g(\sigma) + 1$
    return σ'

Procedure extract-solution

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

Procedure extract-solution

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

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

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 $\langle o, s \rangle \in$ 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()))
$closed := \emptyset$
while not $queue$.empty():
    $\sigma = queue$.pop-front()
    if $state(\sigma) \notin closed$:
        $closed := closed \cup \{state(\sigma)\}$
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each $\langle o, s \rangle \in$ succ($state(\sigma)$):
            σ' := make-node(σ, o, s)
            $queue$.push-back(σ')
return unsolvable

Random walk

Random walk

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

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
- ...

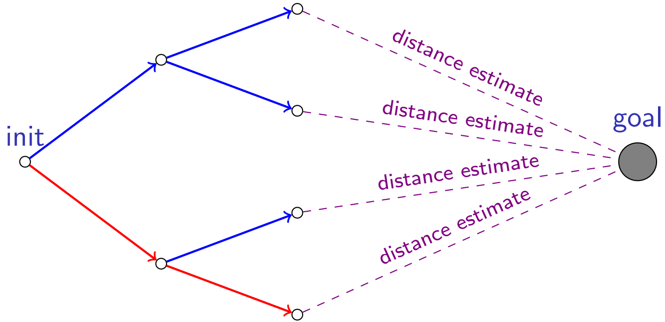
Heuristic search algorithms: local

- 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.

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 $h'(\sigma) := h(state(\sigma))$.
  - 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 $state(\sigma)$ pseudo-heuristics.
- In practice there are sometimes good reasons to have the heuristic value depend on the generating path of σ

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 (σ ↦ h(σ))
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    σ = open.pop-min()
    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)
            if h(σ′) < ∞:
                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)

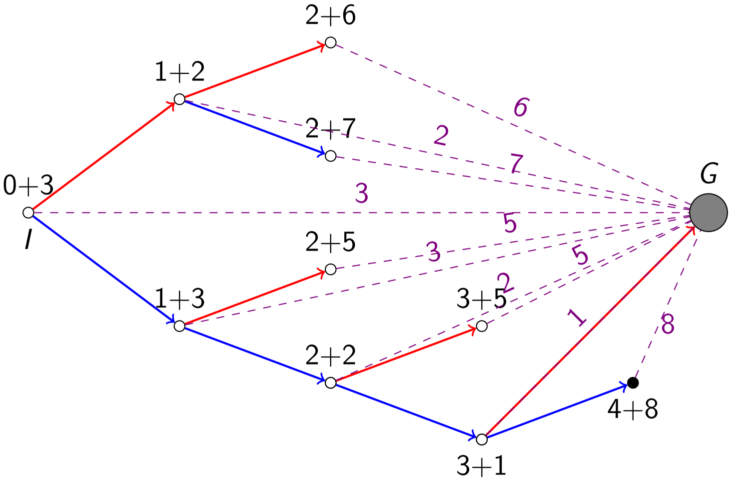
A*

A* (with duplicate detection and reopening)

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

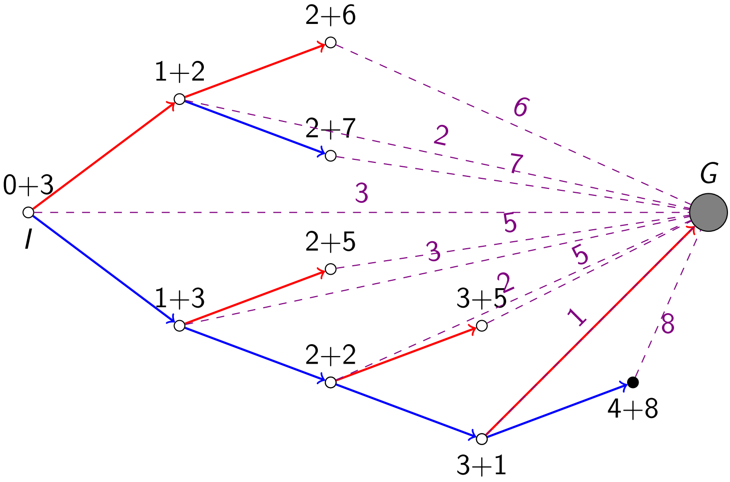
A* example

Example



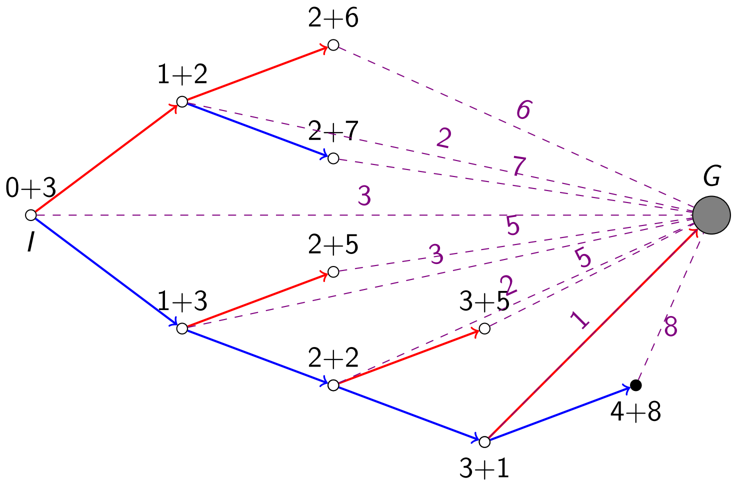
A* example

Example



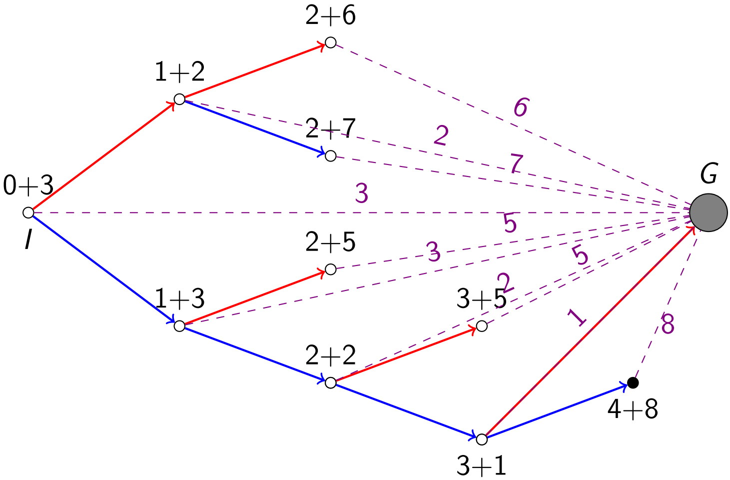
A* example

Example



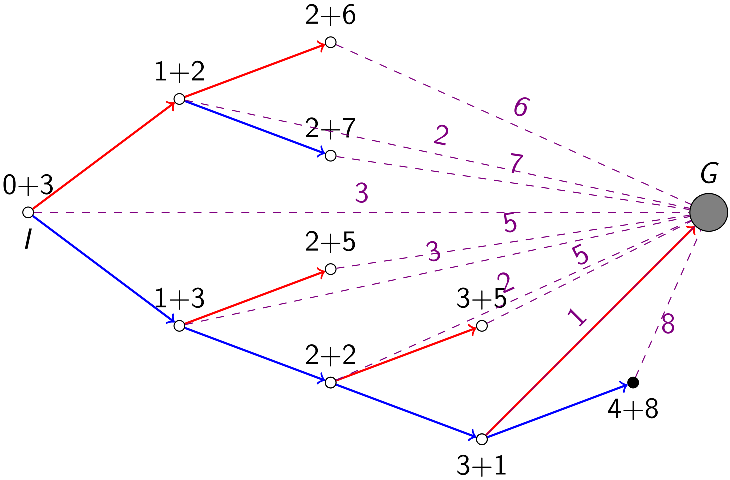
A* example

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 $\mathit{state}(\sigma) \in \mathit{closed}$ 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 (σ ↦ g(σ) + W · 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
```

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

$\sigma :=$ make-root-node(init())
while not is-goal(state(σ)):
    $\sigma :=$ 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

Logical representations of state sets

- n state variables with m values induce a state space consisting of m^n states (2^n states for n Boolean state variables)
- a language for talking about *sets of states (valuations of state variables)*: propositional logic
- logical connectives $\approx$ set-theoretical operations

Syntax of propositional logic

Let P be a set of atomic propositions ($\sim$ state variables).

1. For all $p \in P$, p is a propositional formula.
2. If ϕ is a propositional formula, then so is $\neg\phi$.
3. If ϕ and ϕ' are propositional formulae, then so is $\phi \vee \phi'$.
4. If ϕ and ϕ' are propositional formulae, then so is $\phi \wedge \phi'$.
5. The symbols $\bot$ and $\top$ are propositional formulae.

The implication $\phi \rightarrow \phi'$ is an abbreviation for $\neg\phi \vee \phi'$.
The equivalence $\phi \leftrightarrow \phi'$ is an abbreviation for $(\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$.

Semantics of propositional logic

A valuation of P is a function $v : P \to \{0, 1\}$. Define the notation $v \models \phi$ for valuations v and formulae ϕ by

1. $v \models p$ if and only if $v(p) = 1$, for $p \in P$.
2. $v \models \neg\phi$ if and only if $v \not\models \phi$
3. $v \models \phi \vee \phi'$ if and only if $v \models \phi$ or $v \models \phi'$
4. $v \models \phi \wedge \phi'$ if and only if $v \models \phi$ and $v \models \phi'$
5. $v \models \top$
6. $v \not\models \bot$

Propositional logic terminology

- A propositional formula ϕ is satisfiable if there is at least one valuation v so that $v \models \phi$. Otherwise it is unsatisfiable.
- A propositional formula ϕ is valid or a tautology if $v \models \phi$ for all valuations v. We write this as $\models \phi$.
- A propositional formula ϕ is a logical consequence of a propositional formula ϕ', written $\phi' \models \phi$ if $v \models \phi$ for all valuations v with $v \models \phi'$.
- Two propositional formulae ϕ and ϕ' are logically equivalent, written $\phi \equiv \phi'$, if $\phi \models \phi'$ and $\phi' \models \phi$.

Propositional logic terminology (ctd.)

- A propositional formula that is a proposition p or a negated proposition $\neg p$ for some $p \in P$ is a literal.
- A formula that is a disjunction of literals is a clause. This includes unit clauses l consisting of a single literal, and the empty clause $\bot$ consisting of zero literals.

Normal forms: NNF, CNF, DNF

Formulae vs. sets

sets		formulae
those $\frac{2^n}{2}$ states in which p is true		$p \in P$
$E \cup F$		$E \vee F$
$E \cap F$		$E \wedge F$
$E \setminus F$	(set difference)	$E \wedge \neg F$
$\overline{E}$	(complement)	$\neg E$
the empty set $\emptyset$		$\bot$
the universal set		$\top$

question about sets	question about formulae
$E \subseteq F$?	$E \models F$?
$E \subset F$?	$E \models F$ and $F \not\models E$?
$E = F$?	$E \models F$ and $F \models E$?