02 Search Algorithms

Table of contents

State Model for Classical Planning

State Model: Classical Planning

State Model $\mathcal{S}(P)$:

- finite and discrete state space S
- a known initial state $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a deterministic transition function $s' = f(a, s)$ for $a \in A(s)$
- positive action costs $c(a, s)$

A solution is a sequence of applicable actions that maps s_0 into S_G, and it is *optimal* if it minimizes *sum of action costs* (e.g., # of steps)

Different models and controllers obtained by relaxing assumptions in blue ...

Solving the State Model: Path-finding in graphs

Search algorithms for planning exploit the correspondence between classical state models $\mathcal{S}(P)$ and directed graphs:

- The nodes of the graph represent the states s in the model
- The edges (s, s') capture corresponding transitions in the model with the same cost

In the *planning as heuristic search* formulation, the problem P is solved by path-finding algorithms over the graph associated with model $\mathcal{S}(P)$

Classification of Search Algorithms

Blind search vs. heuristic (or informed) search:

- Blind search algorithms: Only use the basic ingredients for general search algorithms.
  - e.g., Depth First Search (DFS),
  - Breadth-first Search (BrFS),
  - Uniform Cost (Dijkstra), and
  - Iterative Deepening (ID)
- Heuristic search algorithms: Additionally use heuristic functions which estimate the distance (or remaining cost) to the goal.
  - e.g., A*, IDA*, Hill Climbing, Best First, WA*, DFS B&B, LRTA*, ...

Classification of Search Algorithms (continued)

Systematic search vs. local search:

- Systematic search algorithms: Considers a large number of search nodes simultaneously; maintains an explicit frontier or open list of states still to explore, and often also a closed list of visited states.
- Local search algorithms: Local search usually keeps just one current state (or a small number), and repeatedly moves to a neighbouring state that seems better according to an evaluation function.
- This is not a black-and-white distinction; there are crossbreeds (e.g., enforced hill-climbing).

What works where in planning?

Blind search vs. **heuristic search**:

- For satisficing planning, heuristic search vastly outperforms blind algorithms pretty much everywhere.
- For optimal planning, heuristic search also is better (but the difference is less pronounced).

Systematic search vs. **local search**:

- For satisficing planning, there are successful instances of each.
- For optimal planning, systematic algorithms are required.

We cover a subset of search algorithms most successful in planning. Only some blind search algorithms are covered (refer to Russel & Norvig Chapters 3 and 4).

Search terminology

- Search node n: Contains a state reached by the search, plus information about how it was reached.
- Path cost $g(n)$: The cost of the path reaching n.
- Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.
- Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the state s itself is also said to be expanded.
- Search strategy: Method for deciding which node is expanded next.
- Open list: Set of all nodes that currently are candidates for expansion. Also called frontier.
- Closed list: Set of all states that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

World States versus Search States

Search Space for Classical Search

A classical search space is defined by the following three operations:

- $\mathbf{start}()$: Generate the start (search) state.
- $\mathbf{is\text{-}goal}(s)$: Test whether a given search state is a target (goal) state.
- $\mathbf{succ}(s)$: Generates the successor states (a, s') of search state s, along with the actions through which they are reached.

Search states $\neq$ world states?

- Progression (forward reasoning from initial state)?:
- Regression (backwards reasoning from goal)?::

- We consider progression in the entire course, unless explicitly stated otherwise.
- We use 's' to denote *world* and *search* states interchangeably

Search States versus Search Nodes

- Search states s: are states (vertices) of the search space.
- Search nodes σ: are search states, plus information on where/when/how they are encountered during search (i.e. bookkeeping information).

What is in a search node?

Different search algorithms store different information in a search node σ, but typical information includes:

- $\mathrm{state}(\sigma)$: Associated search state.
- $\mathrm{parent}(\sigma)$: Pointer to search node from which σ is reached.
- $\mathrm{action}(\sigma)$: An action leading from $\mathrm{state}(\mathrm{parent}(\sigma))$ to $\mathrm{state}(\sigma)$.
- $g(\sigma)$: Cost of σ (cost of path from the root node to σ).

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

Criteria for Evaluating Search Strategies

Guarantees:

- Completeness: Is the strategy guaranteed to find a solution when there is one?
- Optimality: Are the returned solutions guaranteed to be optimal?

Computational Complexity:

- Time Complexity: How long does it take to find a solution? (Measured in generated states.)
- Space Complexity: How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

- Branching factor b: How many successors does each state have?
- Goal depth d: The number of actions required to reach the shallowest goal state.

Blind Systematic Search

Blind search versus Informed search

Blind search does not require any input beyond the problem.

Pros and Cons?

Compare with Informed search, which requires as additional input of a heuristic function h (we will cover in next module) that maps states to estimates of their goal distance.

Pros and Cons?

In classical planning, h is generated automatically from the declarative problem description.

Blind search strategies we will cover

Blind search strategies we'll cover:

- Breadth-first search. Advantage: time complexity.

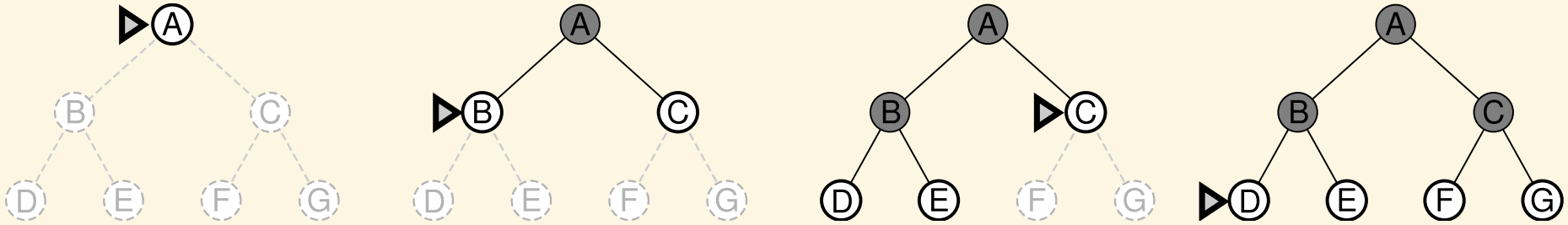
  Variant: Uniform cost search.
- Depth-first search. Advantage: space complexity.
- Iterative deepening search. Combines advantages of breadth-first search and depth-first search. Uses depth-limited search as a sub-procedure.
- Width-based search, in particular Iterated Width (IW)

Blind search strategy we won't cover:

- Bi-directional search. Two separate search spaces, one forward from the initial state, the other backward from the goal. Stops when the two search spaces overlap.

Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).



Guarantees?: **A)** Complete and optimal **B)** Complete but may not be optimal **C)** Optimal but may not be complete **D)** Neither complete nor optimal

- Completeness?
- Optimality?

Breadth-First Search: Time Complexity

Say that b is the maximal branching factor, and d is the goal depth (depth of the shallowest goal state).

- What is the upper bound on the number of generated nodes?
  - So the time complexity is:
- And what if we were to apply the goal test at node-expansion time, rather than node-generation time?

Breadth-First Search: Space Complexity?

Breadth-First Search: Example Data

Settings: $b = 10$; $10{,}000$ nodes/second; $1{,}000$ bytes/node.

Yields data: by inserting values into equations from

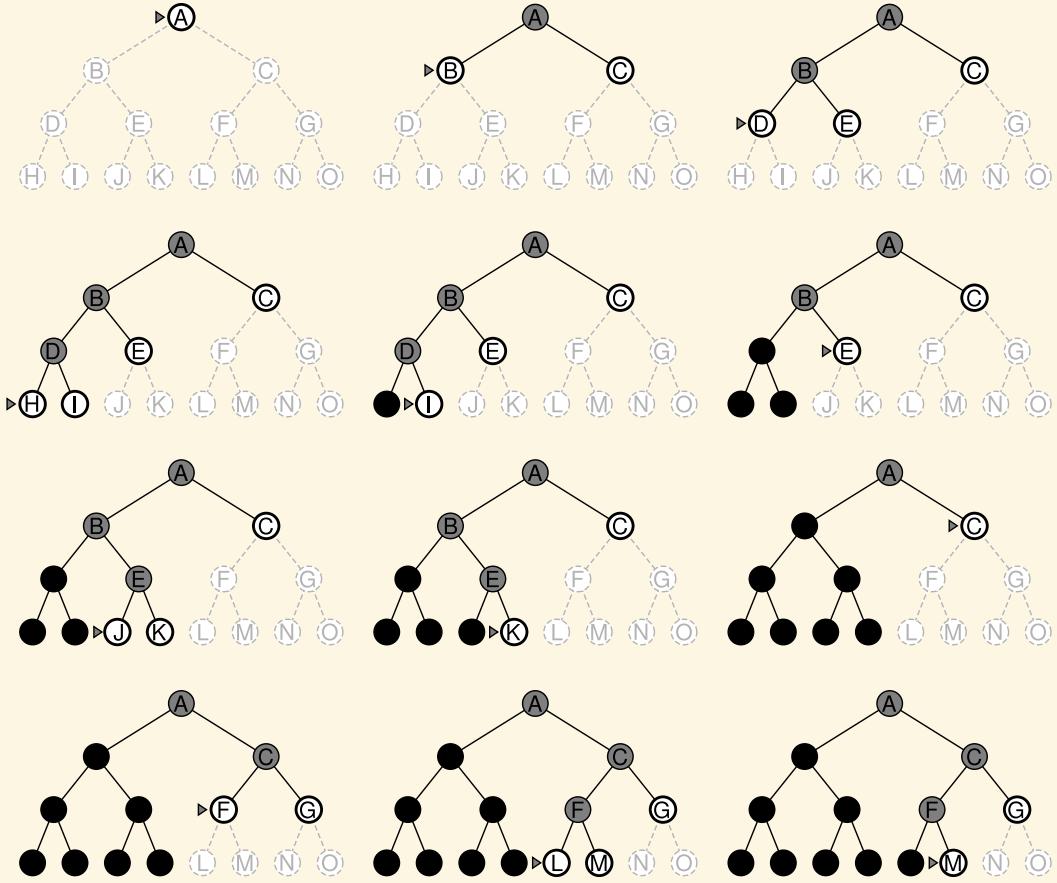
Depth	Nodes	Time	Memory
2	110	0.11 ms	107 KB
4	11,110	11 ms	10.6 MB
6	10^6	1.1 s	1 GB
8	10^8	2 min	103 GB
10	10^{10}	3 h	10 TB
12	10^{12}	13 days	1 PB
14	10^{14}	3.5 years	99 PB

Which is the worse problem - time or memory?

Depth-First Search: Illustration

Strategy: Expand the most recent nodes, LIFO frontier (left to right, top to bottom)

Illustration: Nodes at depth 3 are assumed to have no successors



Depth-First Search: Guarantees

Guarantees?

A) Complete and optimal
B) Complete but may not be optimal
C) Optimal but may not be complete
D) Neither complete nor optimal

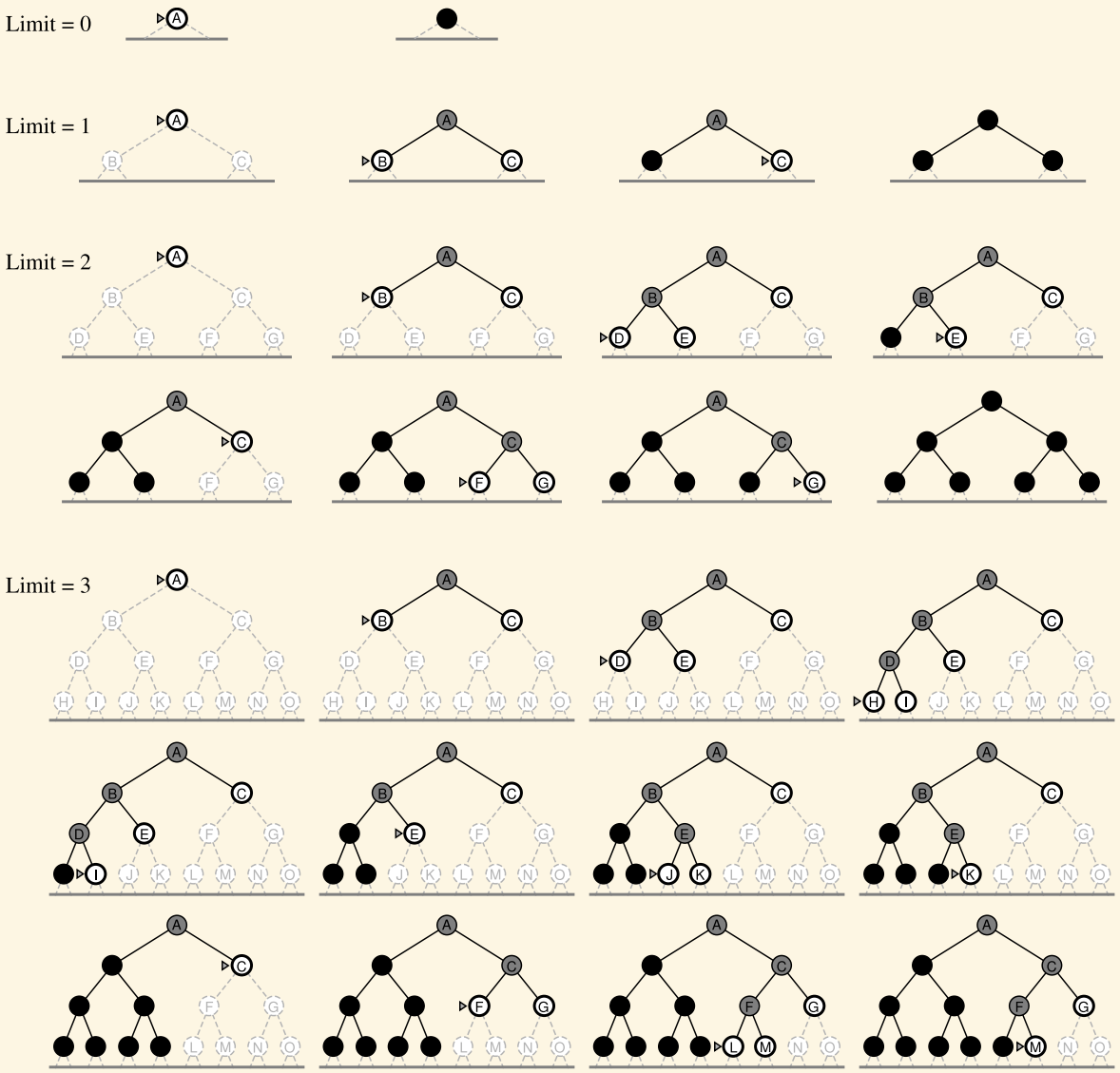
- Optimality?
- Completeness?

Depth-First Search: Complexity

Complexity?

- Space:
- Time:

Iterative Deepening Search: Illustration (by Limit)



Iterative Deepening Search: Guarantees

"Iterative Deepening Search = *Keep doing the same work over again until you find a solution*"

Guarantees:

- Optimality?
- Completeness?
- Space complexity?

Iterative Deepening Search: Time Complexity

Time Complexity?

Width-Based Search

Structure in Planning problems

Planning is computationally complex in the worst case

- PSPACE-complete (see the Appendix)

- However, current planners can *solve* most of the international planning competition (IPC) benchmarks in a few seconds.

Question:

- Can we explain why planners perform well?
  - Can we characterise the structure that separates easy from hard domains?

Answer:

A width of planning exponential in problem width goes a long way to explaining problem *difficulty*:

- Benchmark planning domains have *small width* when *goals* are restricted to *single atoms (boolean variables or facts)*
- Joint goals are easy to serialise into a *sequence* of single goals

Limitations of serialisation?

- Problems with *high atomic width* (however, apparently there are not many practical problems in this class)
- *Multiple-goal* problems that are *not easy to serialise* include, for example, the *Sokoban* game.

Novelty

Definition: Novelty

The *novelty* $w(s)$ of a state s is the size of the smallest subset of atoms (boolean variables or facts) in s that is true for the first time in the search.

- e.g. $w(s) = 1$ if there is *one* atom $p \in s$ such that s is the first state that makes p true.
- Otherwise, $w(s) = 2$ if there are *two* different atoms $p, q \in s$ such that s is the first state that makes $p \wedge q$ true.
- Otherwise, $w(s) = 3$ if there are *three* different atoms...

Iterated Width (IW)

Algorithm

- $IR(k)$ = *breadth-first* search that *prunes* newly generated states whose novelty $(s) > k$.
- IW is a *sequence of calls* $IR(k)$ for $k = 0, 1, 2, \ldots$ over problem P until the problem is solved or k exceeds the number of variables in the problem.

Properties

$IW(k)$ expands at most $O(n^k)$ states, where n is the number of atoms.

Is IW good at Classical Planning?

- IW, while a blind algorithm, is good on planning problems when goals are restricted to *single atoms*.
- The width of benchmark domains is small for such goals.

Our research group tested domains from previous International Planning Competitions (IPCs) benchmark problems.

- For *each instance* with N goal atoms, we *created* N *instances* with a single goal.
- IPC results are remarkably good:

# Instances	IW	ID	$BRFS$	$GBFS + h_{add}$
37921	91%	24%	23%	91%

Why does IW do so well?

Properties

For problems $\Pi \in \mathcal{P}$ where $width(\Pi) = k$:

- $IW(k)$ solves Π in time $O(n^k)$
- $IW(k)$ solves Π *optimally* for problems with uniform cost functions
- $IW(k)$ is *complete* for Π

Theorem

Blocks, Logistics, Gripper, and n-puzzle have a *bounded width* independent of problem *size* and *initial situation*, provided that goals are *single atoms*.

In practice, $IW(k \leq 2)$ solves 88.3% of IPC problems with single goals:

# Instances	$k = 1$	$k = 2$	$k > 2$	Total
37921	37.0%	51.3%	11.7%	88.3%

IW in Classical Planning?

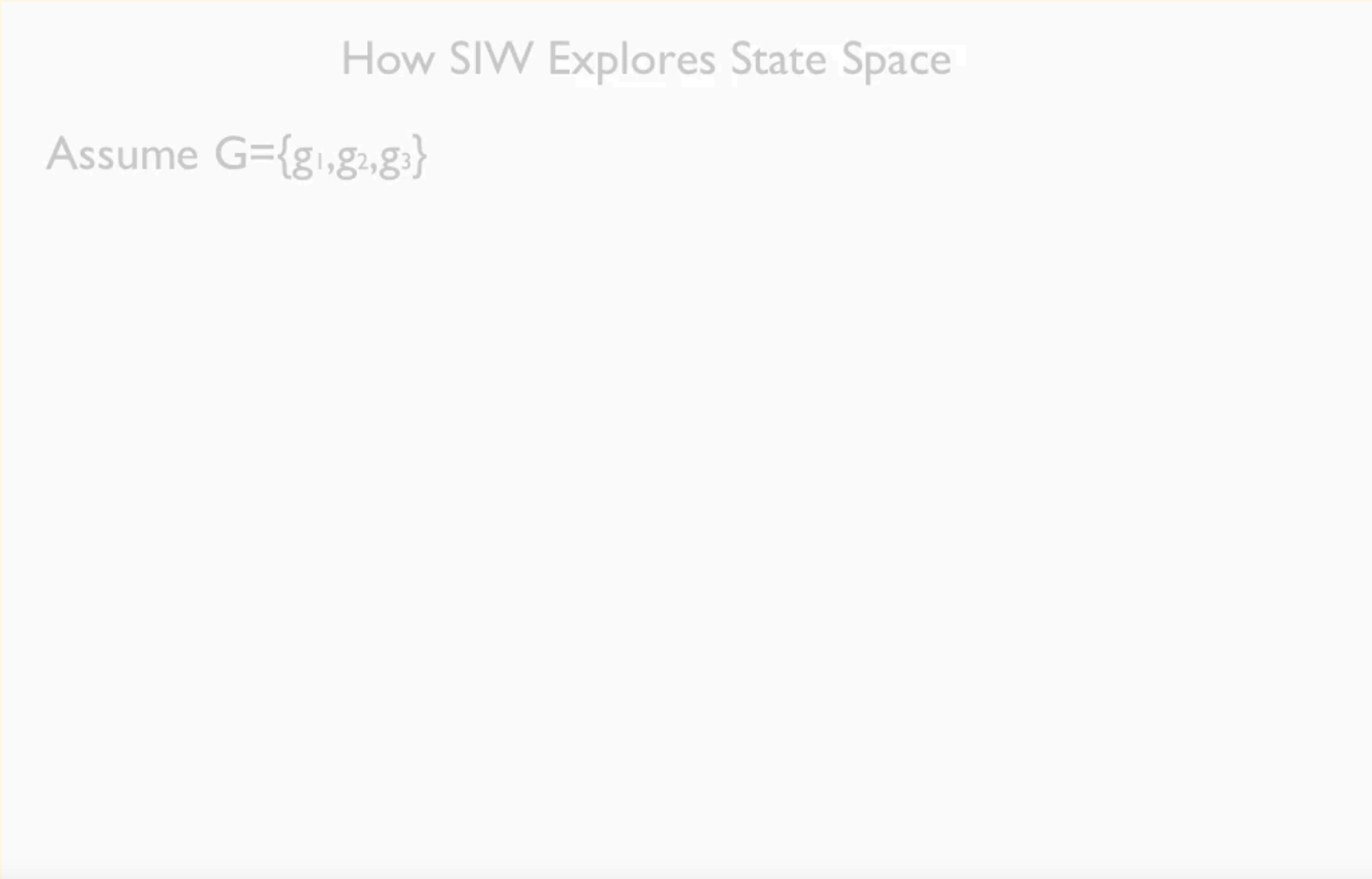
Primary question: IW solves atomic (single atom) goals — how do we extend the blind procedure to multiple atomic goals?

Serialised Iterated Width (SIW)

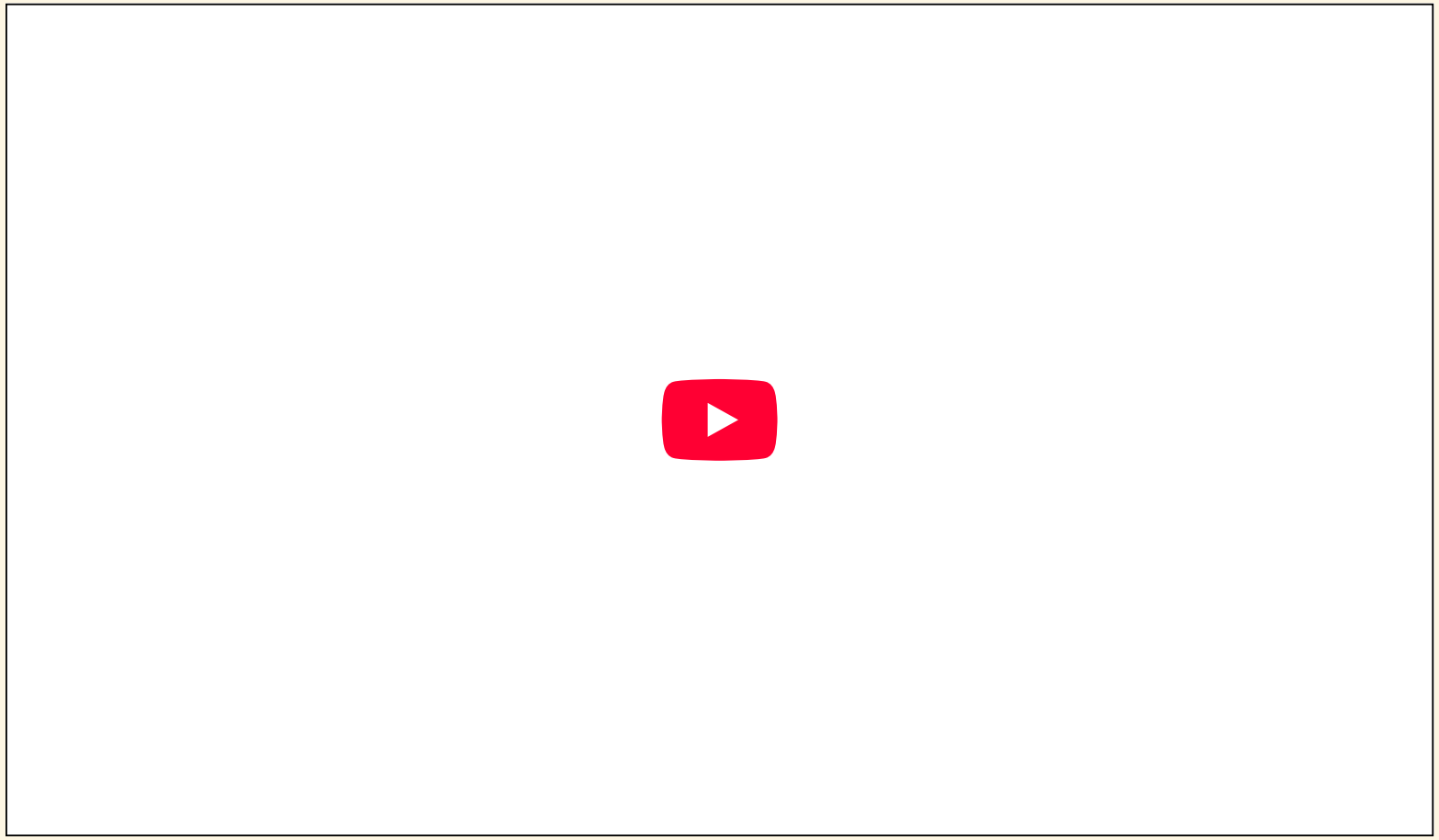
A simple way to use IW for solving real benchmarks P with joint goals is a simple form of hill climbing over the goal set G with $|G| = n$

- This achieves atomic goals one at a time…

How SIW Explores State Space



IW Example: Playing Atari Games



https://www.youtube.com/watch?v=P-603qPMkSg

Serialised Iterated Width (SIW) (continued)

- SIW uses IW for both decomposing a problem into subproblems and solving subproblems.
- It's a *blind search* procedure, as IW does not even know the next goal G_i to achieve.

Blind SIW is better than $GBFS$

- It is even better than GBFS + the *heuristic* h_{add} which we will introduce in the next module

IW Overview

IW: is essentially a sequence of novelty-based pruned breadth-first searches

- **Experiments:** excellent when goals are restricted to atomic goals
- **Theory:** such problems have low width w, and IW runs in time $O(n^w)$

SIW: is essentially IW serialised, used to attain top goals one-by-one

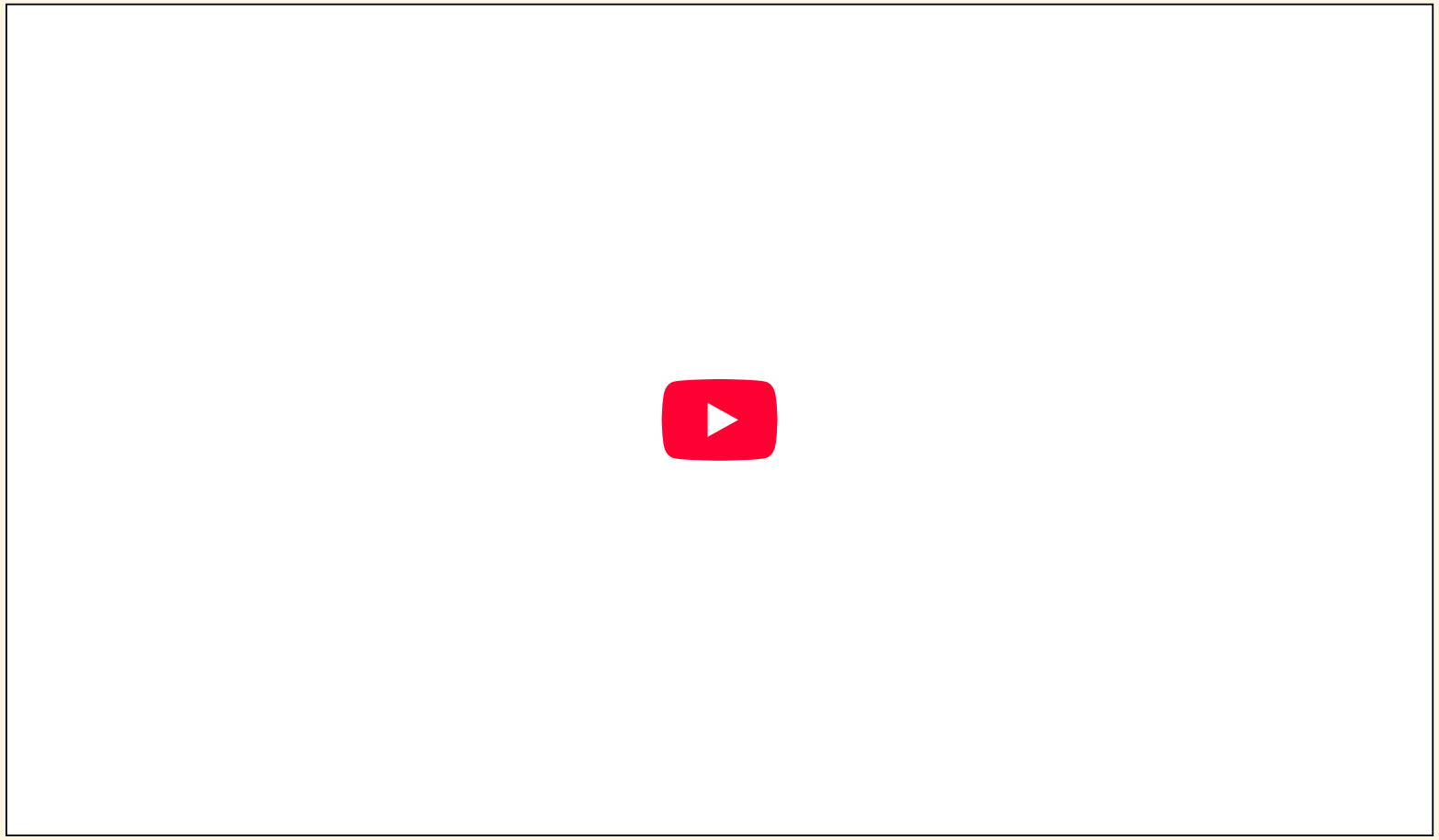
- **Experiments:** faster, better coverage, and much better plans than GBFS with h_{add}
- **Intuition:** goals are easy to [serialise]{style="color:blue"] and have atomic low width w

Heuristic Functions

Heuristic Search Algorithms

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

The Origins of Heuristic Search: Shakey, 1972



https://www.youtube.com/watch?v=GmU7SimFkpU&t=6s

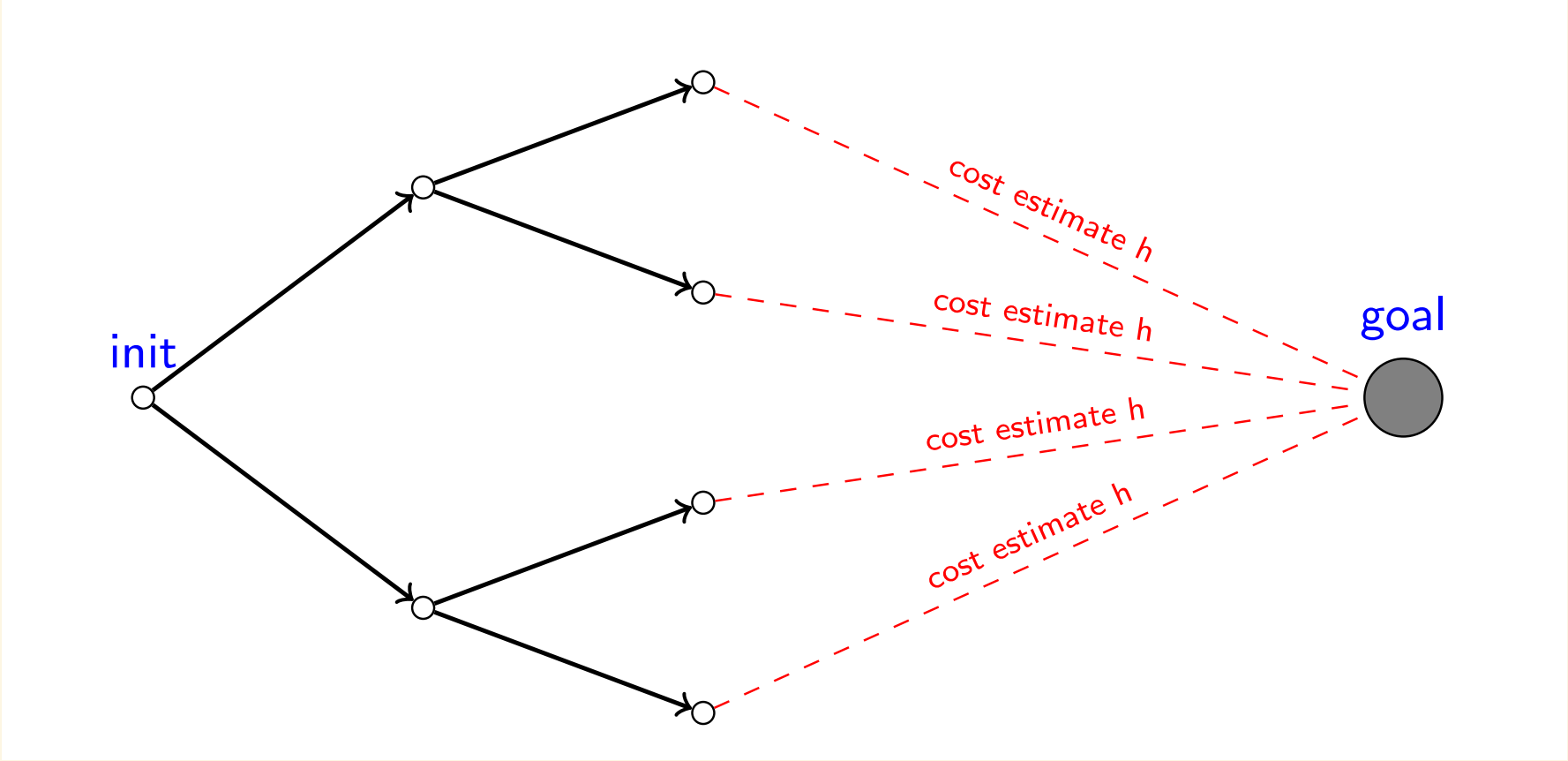
Heuristic search algorithms: Systematic

- Greedy best-first search.
  - One of the three most popular algorithms in *satisficing* planning
- Weighted A*.
  - One of the three most popular algorithms in *satisficing* planning
- A*.
  - Most popular algorithm in *optimal* planning (rarely ever used for satisficing planning)
- IDA*, depth-first branch-and-bound search, breadth-first heuristic search, …

Heuristic search algorithms: Local

- Hill-climbing.
- Enforced hill-climbing.
  - One of the three most popular algorithms in *satisficing* planning
- Other algorithms include beam search, tabu search, genetic algorithms, simulated annealing, etc.

Heuristic Search: Basic Idea



- A heuristic function h estimates the cost of an optimal path to the goal
- Search gives a *preference* to explore states with small h.

Heuristic Functions

Heuristic searches require a heuristic function to estimate remaining cost

Definition: Heuristic Function

Let Π be a planning problem with state space Θ_Π.

- A heuristic function, or *heuristic*, for Π is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$
- Its value $h(s)$ for state s is referred to as the state's heuristic value, or just h-value

Definition: Remaining Cost, h^*

Let Π be a planning problem with state space Θ_Π.

- For a state $s \in S$, the state's remaining cost is the cost of an optimal plan for s, or ∞ if there exists no plan for s.
- The perfect heuristic for Π, written h^*, assigns every $s \in S$ its remaining cost as the heuristic value.

Heuristic Functions: Discussion

What does it mean to estimate remaining cost?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to *work* (meaning being correct and complete).
  - h is *any* function from states to numbers…
- Search *performance* depends crucially on how well h reflects h^*
  - This is informally called the informedness or quality of h.
- For some search algorithms, such as A^*, the relationship between formal quality properties of h and search efficiency can be proven (number of expanded nodes).
- For other search algorithms, *"it works well in practice"* is often as good an analysis as one gets.

We will analyse in a later Module detail approximations to one particularly important heuristic function in planning: h^+.

Heuristic Functions: Discussion (continued)

- Search performance depends crucially on the *informedness* of h

Are there other properties of h that search performance crucially depends on?

What about edge cases?

Important Properties of Heuristic Functions

Definition: Heuristic Function Properties

Let Π be a planning problem with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$, and let h be a heuristic for Π.

- h is safe if for all $s \in S, h(s) = \infty \implies h^*(s) = \infty$
- h is goal-aware if $h(s) = 0$ for all goal states $s \in S^G$
- h is admissible if $h(s) \leq h^*(s)$ for all $s \in S$
- h is consistent if $h(s) \leq h(s') + c(a)$ for all transitions $s \xrightarrow{a} s'$

- Note that $h^*(s)$ is the *perfect* heuristic for state s (optimal cost-to-go), and $h(s)$ is the value *actually* returned by the heuristic function.
- T is the transition relation which specifies which successor states can be reached from which states under which action and L are the action or operator labels.

Relationships between properties of Heuristic Functions

What are the relationship between these properties?

- safety (never discard a state that could lead to a valid solution)
- goal-awareness (assigns 0 to every goal state)
- admissibility (never over estimates true cost to a goal)
- consistency (recognises triangle inequality over transitions)

Informed Systematic Search

Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

$open :=$ **new** priority queue ordered by ascending $h(state(\sigma))$
$open.\,\text{insert}(\text{make-root-node}(init()))$
$closed := \emptyset$
while not $open.\,empty()$:
 $\sigma := open.\,\text{pop-min}()$ /* get best state */
 if $state(\sigma) \notin closed$: /* check for duplicates */
  $closed := closed \cup \{state(\sigma)\}$ /* add state to closed set */
  if $is\text{-}goal(state(\sigma))$: **return** $\text{extract-solution}(\sigma)$
  for each $(a, s') \in succ(state(\sigma))$: /* expand state */
   $\sigma' := \text{make-node}(\sigma, a, s')$
   if $h(state(\sigma')) < \infty$: $open.\,\text{insert}(\sigma')$
return unsolvable

Greedy Best-First Search: Properties

Completeness?

Optimality?

Greedy Best-First Search: Implementation as A^* variant

Depending upon where you do duplicate detection in the Greedy Best First Search (GBFS) loop, it can make GBFS appear as an A* variant as follows:

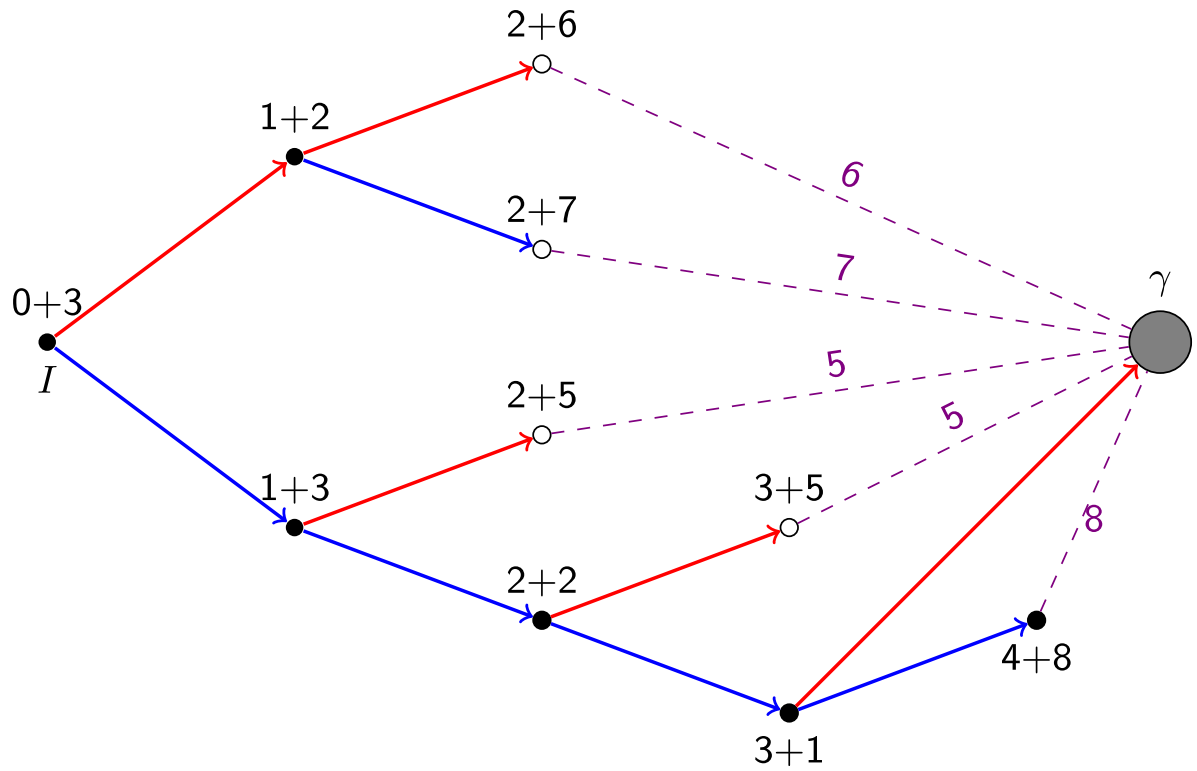
- A priority queue can be implemented as a min heap
  - A min-heap is a data structure for a priority queue where the smallest key is always quick to access.
- Duplicate detection can be done at the state expansion stage of A*
  - Checking duplicates can be done following getting the best state

A^* (GBFS variant)

A^* (with duplicate detection and re-opening)

$open$:= **new** priority queue ordered by ascending $g(state(\sigma)) + h(state(\sigma))$
$open$.insert(make-root-node(init()))
$closed := \emptyset$
$best\text{-}g := \emptyset$ /* maps states to non-negative real numbers */
while not $open$.empty():
 $\sigma := open$.pop-min()
 if $state(\sigma) \notin closed$ or $g(\sigma) < best\text{-}g(state(\sigma))$:
  /* Check duplicates: re-open if better g (note that all σ' with same state but worse g are behind σ in $open$, and will be skipped when their turn comes) */
  $closed := closed \cup \{state(\sigma)\}$
  $best\text{-}g(state(\sigma)) := g(\sigma)$
  if is-goal$(state(\sigma))$: **return** $extract\text{-}solution(\sigma)$
  for each $(a, s') \in$ succ$(state(\sigma))$:
   $\sigma' :=$ make-node(σ, a, s')
   if $h(state(\sigma')) < \infty$: $open.insert(\sigma')$
return unsolvable

A^* (The normal variant): Example



A^*: Terminology

- f-value of a state: defined by $f(s) = g(s) + h(s)$.
- Generated nodes: Nodes inserted into $open$ priority queue at some point.
- Expanded nodes: Nodes σ popped from $open$ priority queue, for which the test against the $closed$ set and the distance succeeds.
- Re-expanded nodes: Expanded nodes for which $state(\sigma) \in closed$ upon expansion (also called re-opened nodes).

A^*: Properties

Completeness?

Optimality?

A^*: Implementation Considerations

- A popular method is to break ties, in the event that $f(s_1) = f(s_2)$, by choosing the node with the smaller h-value.
- If h is admissible and consistent, then A^* never re-opens a state. So if we know that this is the case, we can simplify the algorithm.
- A common, and hard-to-spot bug is as follows: *checking duplicates at the wrong point in the algorithm* (note that the Russell & Norvig text is not very precise about this).
- Note that the implementation shown in this Module is optimised for *readability*, not for *efficiency*!

Question?

Question

If we set $h(s) := 0$ for all s, what does A^* become?

(A) Breadth-first search

(B) Depth-first search

(C) Uniform-cost search

(D) Depth-limited search

Recall that uniform-cost search is essentially Dijkstra (a best-first search that always expands the frontier node with the lowest *path* cost so far).

Weighted A^*

Weighted A^* (with duplicate detection and re-opening)

$open :=$ **new** priority queue ordered by ascending $g(state(\sigma)) + W\ h(state(\sigma))$
$open.\ \text{insert}(\text{make-root-node}(init()))$
$closed := \emptyset$
$best\text{-}g := \emptyset$
while not $open.\ \text{empty}()$:
 $\sigma := open.\ \text{pop-min}()$
 if $state(\sigma) \notin closed$ **or** $g(\sigma) < best\text{-}g(state(\sigma))$:
  $closed := closed \cup \{state(\sigma)\}$
  $best\text{-}g(state(\sigma)) := g(\sigma)$
  if $is\text{-}goal(state(\sigma))$: **return** $\text{extract-solution}(\sigma)$
  for each $(a, s') \in succ(state(\sigma))$:
   $\sigma' := \text{make-node}(\sigma, a, s')$
   if $h(state(\sigma')) < \infty$: $open.\ \text{insert}(\sigma')$
return unsolvable

Weighted A^*: Properties

The weight $W \in \mathbb{R}_0^+$ is an algorithm *parameter*:

- For $W = 0$, weighted A^* behaves like uniform-cost search.
- For $W = 1$, weighted A^* behaves like A^*.
- For $W \to \infty$, weighted A^* behaves like greedy best-first search.

For $W > 1$, weighted A^* is bounded suboptimal*

- if h is admissible, then the solutions returned are at most a factor W more costly than the optimal ones.

*Bounded suboptimal means that the algorithm may return a non-optimal solution, and there's a proven bound on how much worse it can be than optimal.

Local Search Algorithms

Hill-Climbing

Hill-Climbing

$\sigma := make\text{-}root\text{-}node(init())$
forever:
 if $is\text{-}goal(state(\sigma))$:
  return $extract\text{-}solution(\sigma)$
 $\Sigma' := \{ make\text{-}node(\sigma, a, s') \mid (a, s') \in succ(state(\sigma)) \}$
 $\sigma :=$ choose an element of Σ' minimising h /* random tie breaking */

- Makes sense only if $h(s) > 0$ for $s \notin S^G$ (i.e. all non-goal states have a positive heuristic)

Is this complete or optimal?

Enforced Hill-Climbing

Enforced Hill-Climbing: Procedure $improve$

def $improve(\sigma_0)$:
 $queue$:= **new** FIFO queue
 $queue$. push-back(σ_0)
 $closed := \emptyset$
 while not $queue$. empty():
  $\sigma := queue$. pop-front()
  if $state(\sigma) \notin closed$:
   $closed := closed \cup \{state(\sigma)\}$
   if $h(state(\sigma)) < h(state(\sigma_0))$: **return** σ /* If better state is found return it */
   for each $(a, s') \in succ(state(\sigma))$:
    $\sigma' :=$ make-node(σ, a, s')
    $queue$. push-back(σ')
 fail

Is essentially breadth-first search for a state with strictly smaller h-value

- Keep hill climbing, but when stuck, systematically search outward until you find an improvement (less easily gets stuck)

Enforced Hill-Climbing

$\sigma := make\text{-}root\text{-}node(init())$
while not $is\text{-}goal(state(\sigma))$:
 $\sigma := improve(\sigma)$
return $extract\text{-}solution(\sigma)$

Is enforced hill-climbing optimal?

Is enforced hill-climbing complete?

Properties of Search Algorithms

	DFS	BrFS	ID	A^*	HC	IDA*	IW
Complete	No	Yes	Yes	Yes	No	Yes	No
Optimal	No	Yes*	Yes	Yes	No	Yes	No
Time	∞	b^d	b^d	b^d	∞	b^d	$b \cdot n^k$
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$	n^k

- *Parameters:* d is solution depth; b is branching factor (number of applicable actions)
- *Breadth-first search (BrFS)* is optimal when costs are uniform.
- A^*/IDA^* are optimal when h is *admissible*, i.e., $h \leq h^*$
- For IW, n is the number of *features* and k is the *width* parameter.

Conclusion

Questions

Question (Revisited)

If we set $h(s) := 0$ for all s, what does A^* become?

(A) Breadth-first search

(B) Depth-first search

(C) Uniform-cost search

(D) Depth-limited search

Question

If we set $h(s) := 0$ for all s, what does greedy best-first search become?

(A) Breadth-first search

(B) Depth-first search

(C) Uniform-cost search

(D) A), B) and C)

Question

Is informed search always better than blind search?

(A): Yes.

(B): No.

Summary

Distinguish: World states, search states, search nodes.

- World state: Situation in the world modelled by the planning problem.
- Search state: Subproblem remaining to be solved.
  - In progression, world states and search states are identical.
  - In regression, search states are sub-goals describing sets of world states.
- Search node: Search state + information on "how we got there".

Search algorithms mainly differ in order of node expansion:

- Blind vs. heuristic (or informed) search.
- Systematic vs. local search.

Search strategies differ in the order in which they expand search nodes, and in the way they use duplicate elimination.

Criteria for evaluating them include completeness, optimality, time complexity, and space complexity.

- Breadth-first search is optimal but uses exponential space;
- Depth-first search uses linear space but is not optimal;
- Iterative deepening search combines the virtues of both.

Heuristic Functions are estimators for remaining cost.

- Usually, the more informed the search is, the better the performance.
- Desiderata includes safety, goal-awareness, admissibility, and consistency.
- The ideal is a perfect heuristic h^*.

Heuristic Search Algorithms:

- Most common algorithms for satisficing planning are:
  - Greedy best-first search
  - Weighted A^*
  - Enforced hill-climbing
- Most common algorithm for optimal planning are:
  - A^*

Additional Reading

- *Artificial Intelligence: A Modern Approach, Russell and Norvig (Third Edition)*
  - An overview of various search algorithms, including blind searches as well as greedy best-first search and A^*.
  - See Chapter 3: “Solving Problems by Searching” and the first half of Chapter 4: “Beyond Classical Search”.
- Search tutorial in the context of path-finding: http://www.redblobgames.com/pathfinding/a-star/introduction.html