CS 540 Introduction to Artificial Intelligence

Search II: Informed Search

University of Wisconsin-Madison

Spring 2023

Announcements

Homeworks:

- Homework 8 released today; due Tuesday April 18

Class roadmap:

Tuesday, April 11	Informed Search
Thursday, April 13	Advanced Search
Tuesday, April 18	Games I
Thursday, April 20	Games II
Tuesday, April 25	Reinforcement Learning I

Announcements

Homeworks:

- Homework 8 released today; due Tuesday April 18

Class roadmap:

Tuesday, April 11	Informed Search
Thursday, April 13	Advanced Search
Tuesday, April 18	Games I
Thursday, April 20	Games II
Tuesday, April 25	Reinforcement Learning I

Practice questions on search and neural networks on Canvas.

Today’s Goals

Today’s Goals

- Finish and review of uninformed search strategies.

Today's Goals

- Finish and review of uninformed search strategies.
- Understand the difference between uninformed and informed search.

Today's Goals

- Finish and review of uninformed search strategies.
- Understand the difference between uninformed and informed search.
- Introduce A* Search
  - Heuristic properties, stopping rules, analysis

Today's Goals

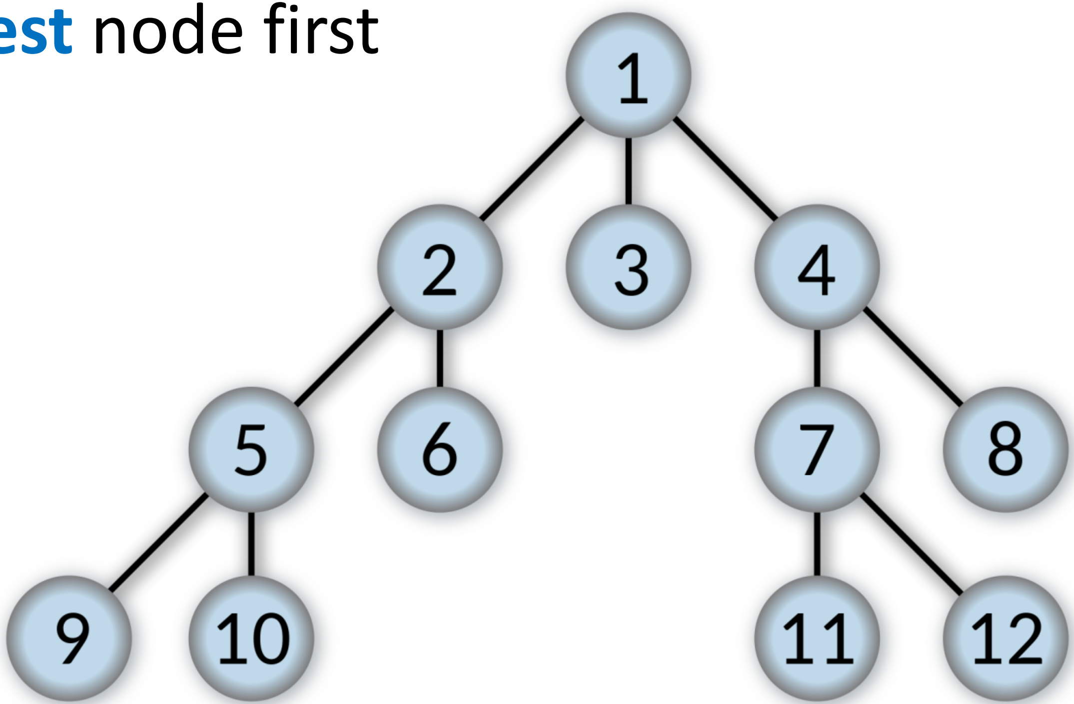
- Finish and review of uninformed search strategies.
- Understand the difference between uninformed and informed search.
- Introduce A* Search
  - Heuristic properties, stopping rules, analysis
- Extensions: Beyond A*
  - Iterative deepening, beam search

Breadth-First Search

Recall: expand **shallowest** node first

Breadth-First Search

Recall: expand **shallowest** node first

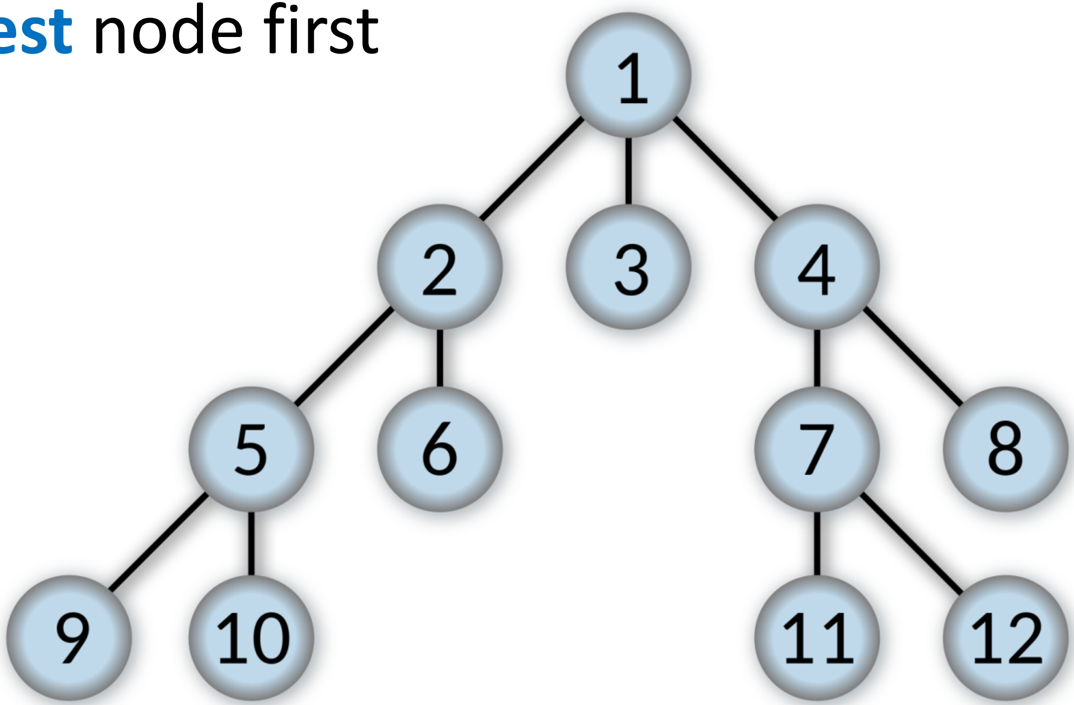


Wiki

Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue

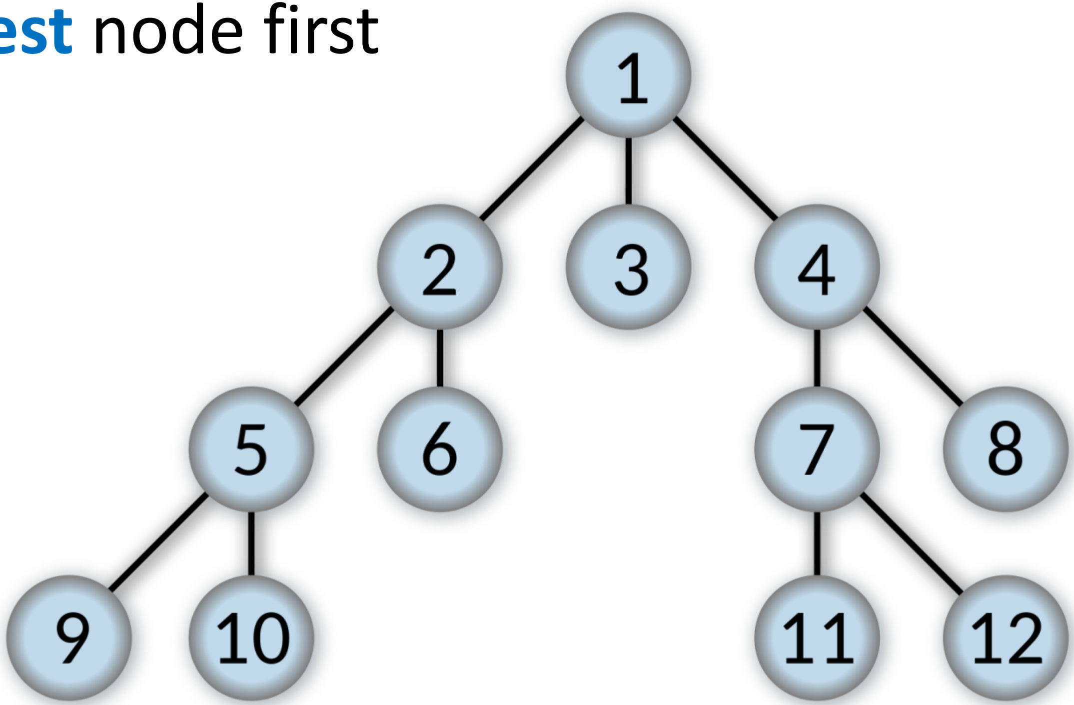


Wiki

Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:

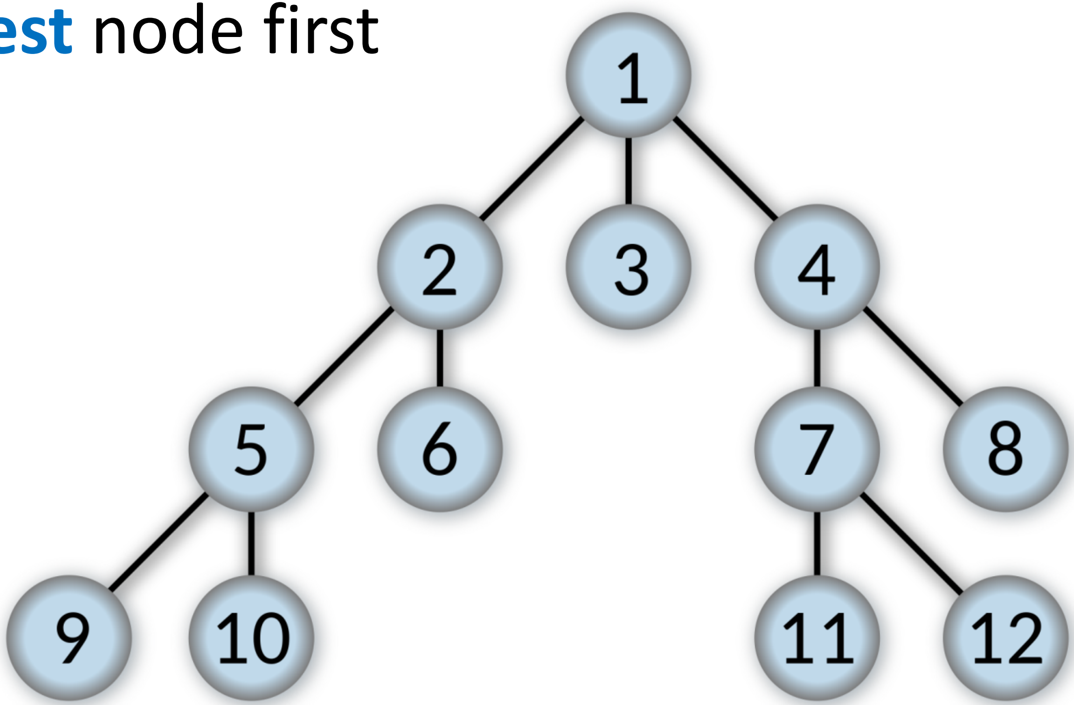


Wiki

Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:
  - Complete

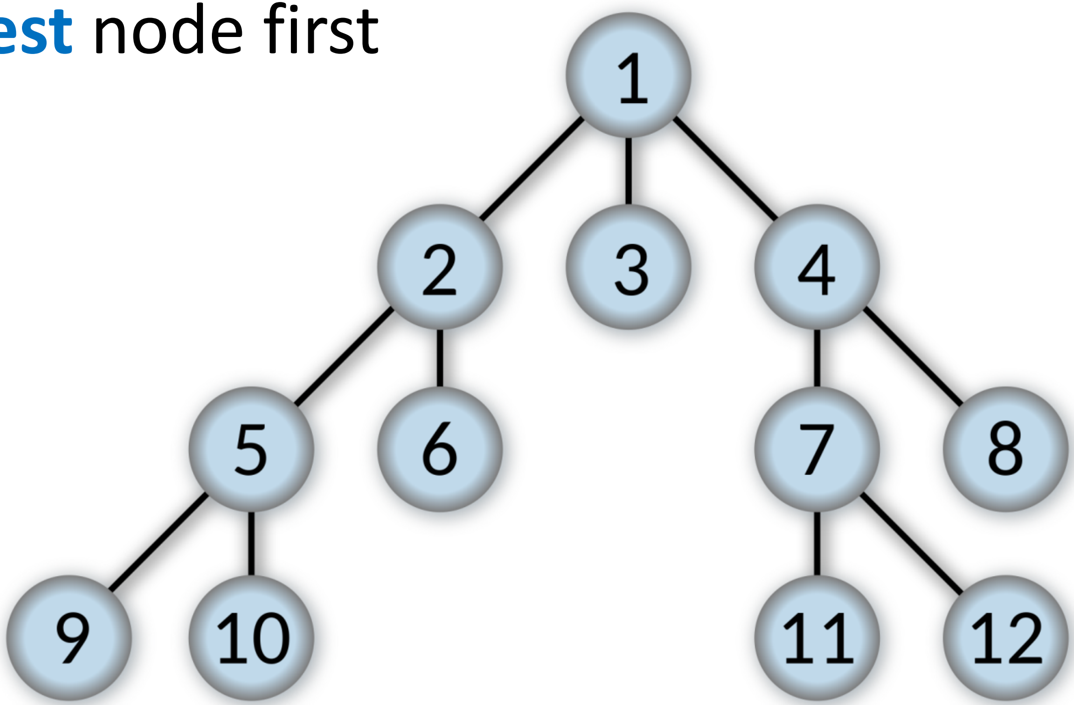


Wiki

Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)

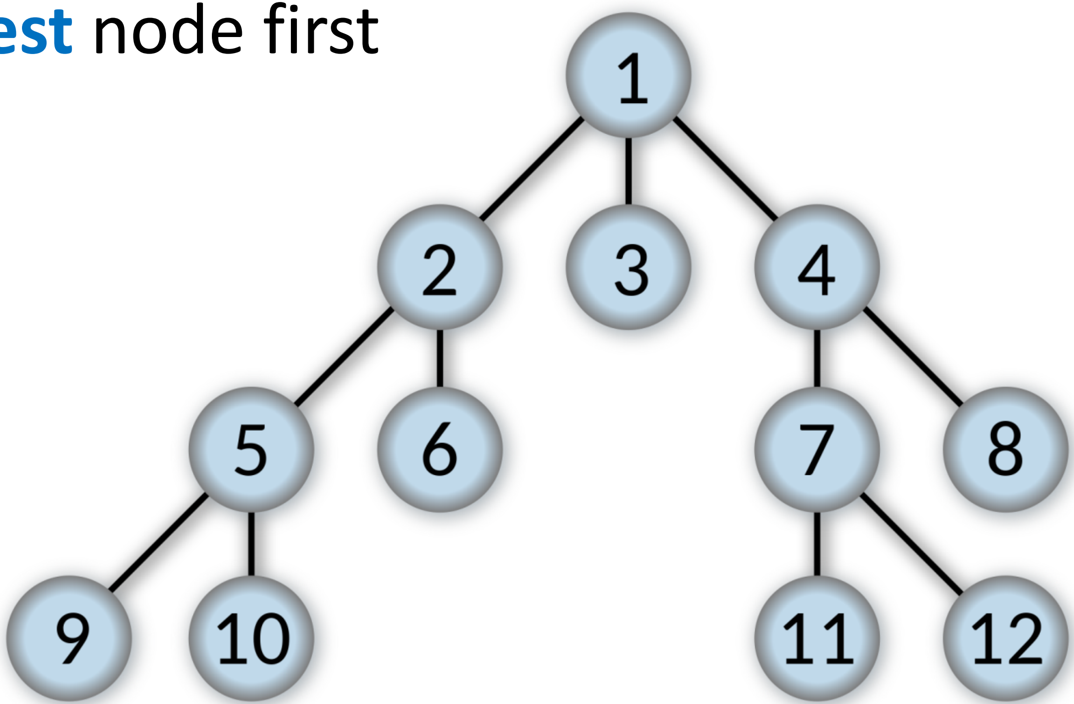


Wiki

Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time O(b^d)



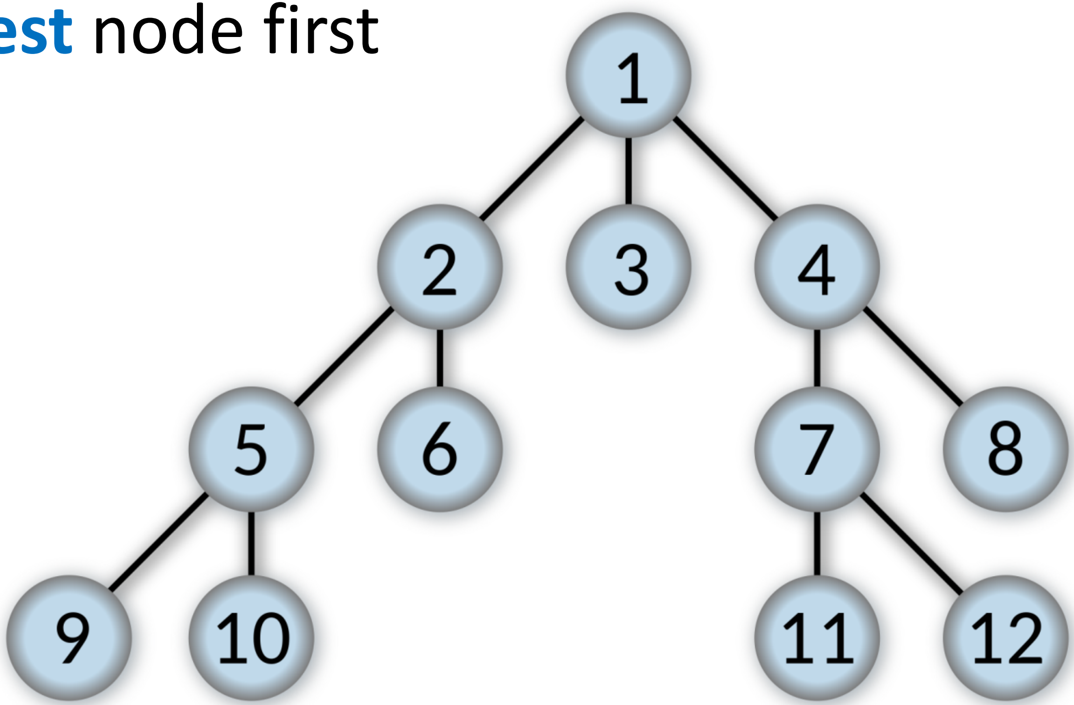
Wiki

Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time $O(b^d)$

Depth



Wiki

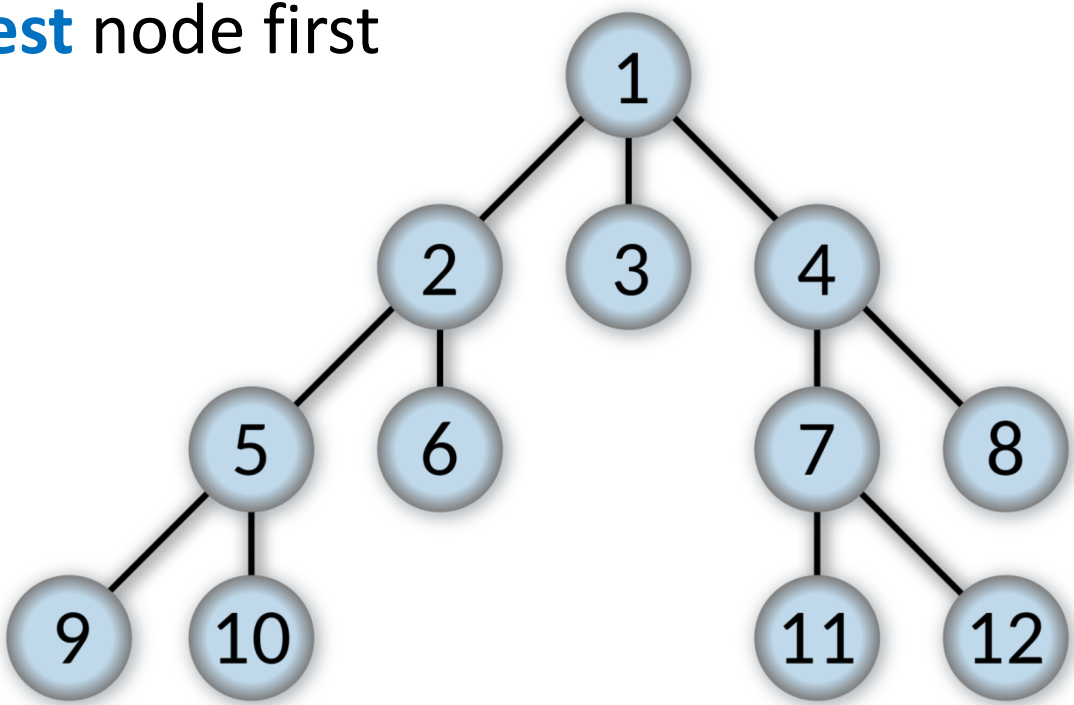
Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time $O(b^d)$

Depth

Branching Factor



Wiki

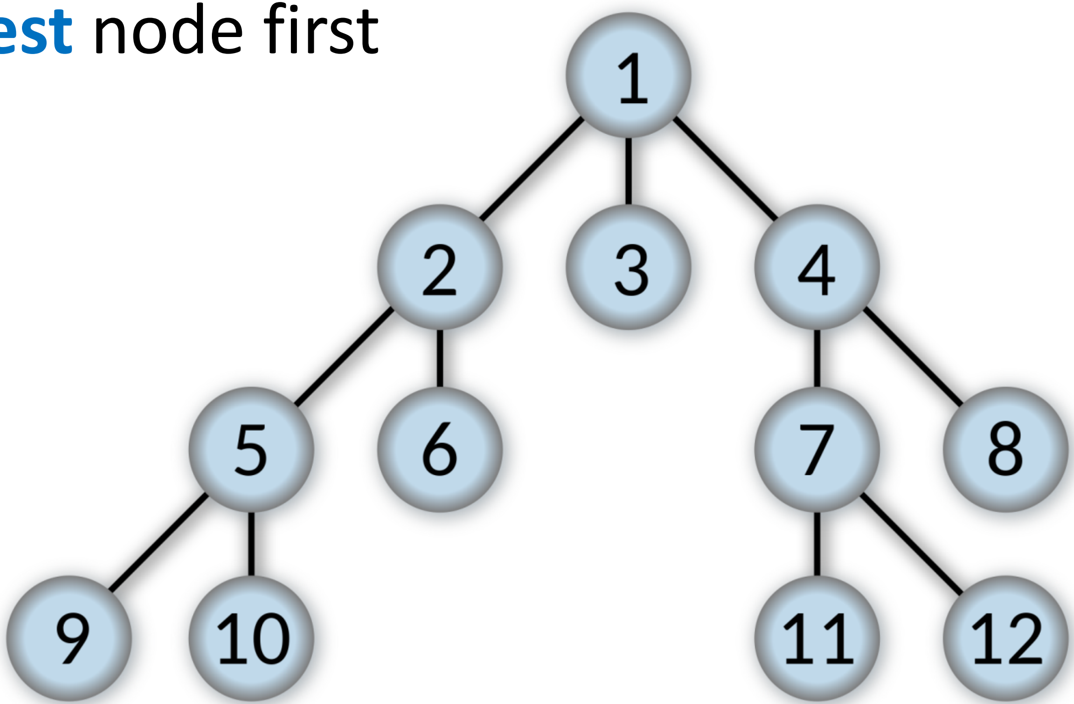
Breadth-First Search

Recall: expand **shallowest** node first

- Data structure: queue
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time $O(b^d)$

    Depth

    Branching Factor
  - Space $O(b^d)$



Wiki

Uniform Cost Search

Like BFS, but keeps track of cost

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue
- **Properties**:

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue
- **Properties**:
  - Complete

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue
- **Properties**:
  - Complete
  - Optimal (if weight lower bounded by ε)

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue
- **Properties**:
  - Complete
  - Optimal (if weight lower bounded by ε)
  - Time $O(b^{C^*/\varepsilon})$

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue
- **Properties**:
  - Complete
  - Optimal (if weight lower bounded by ε)
  - Time O($b^{C^*/\varepsilon}$)
  - Space O($b^{C^*/\varepsilon}$)

Credit: DecorumBY

Uniform Cost Search

Like BFS, but keeps track of cost

- Expand least cost node
- Data structure: priority queue
- **Properties**:
  - Complete
  - Optimal (if weight lower bounded by ε)
  - Time $O(b^{C^*/\varepsilon})$
  - Space $O(b^{C^*/\varepsilon})$

    C* is optimal path cost to goal.

    ϵ is cost of edge with smallest cost.

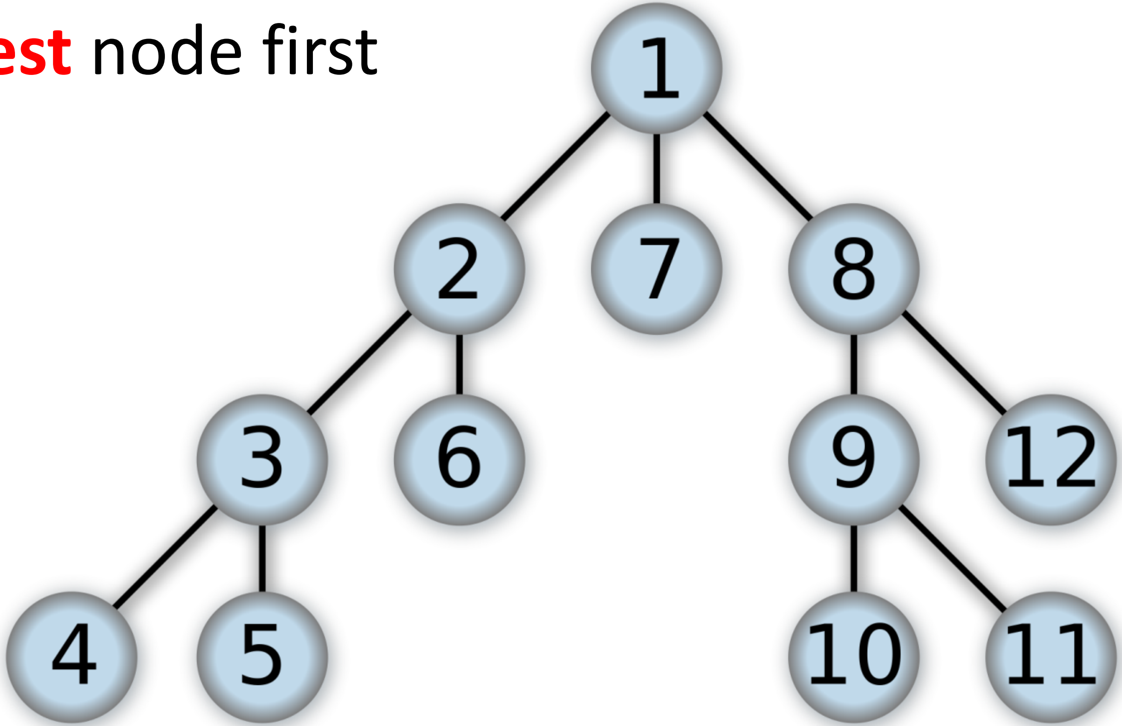
Credit: DecorumBY

Depth-First Search

Recall: expand **deepest** node first

Depth-First Search

Recall: expand **deepest** node first

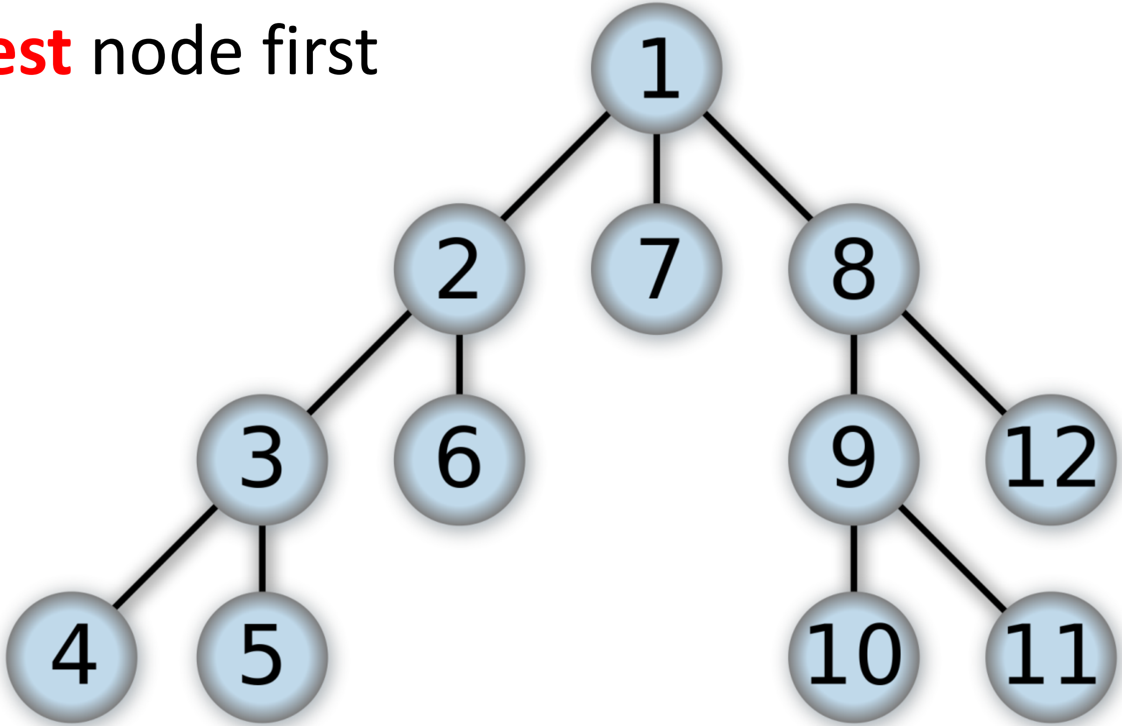


Wiki

Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack

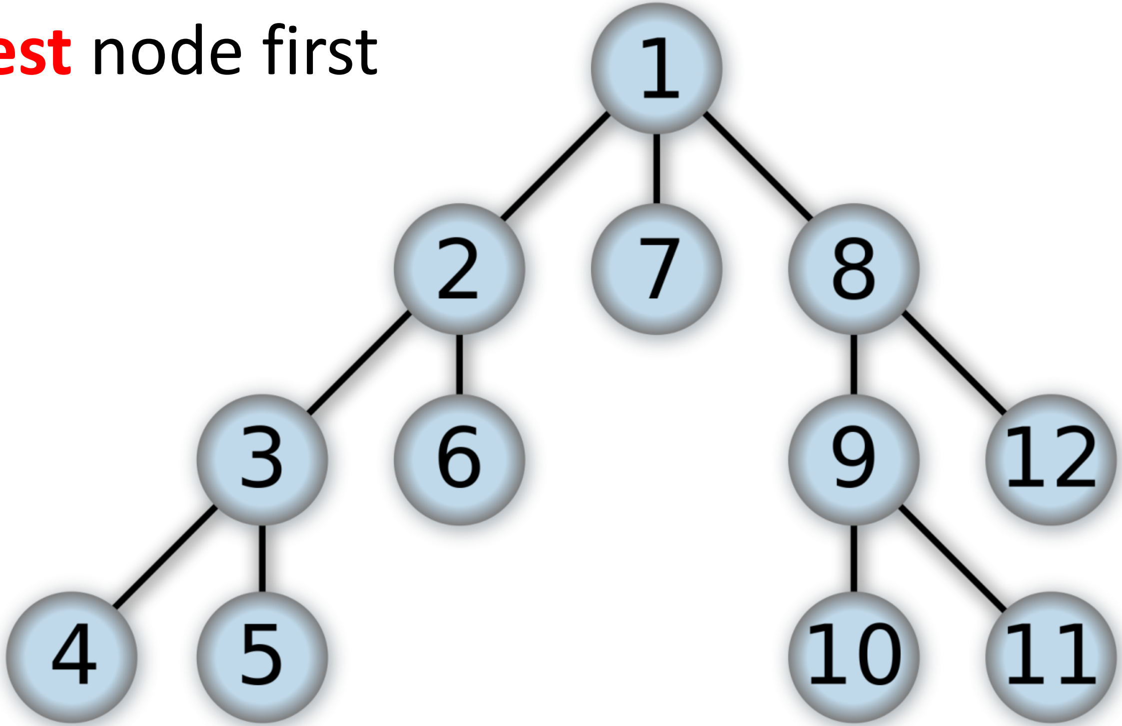


Wiki

Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack
- **Properties**:

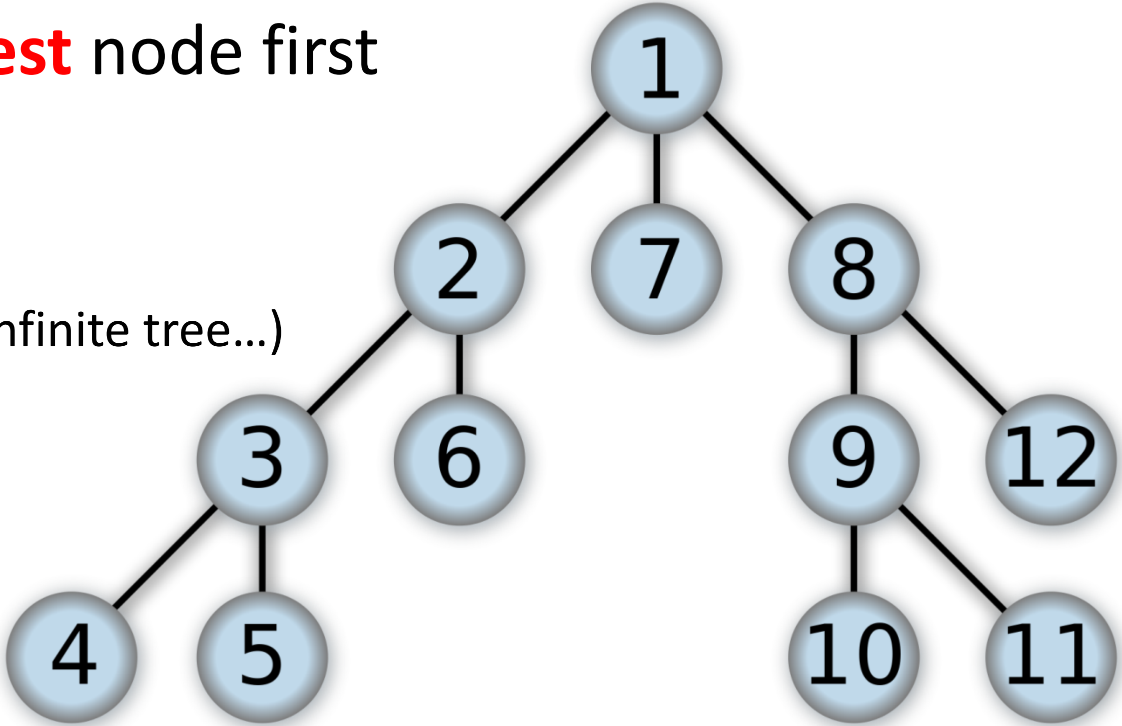


Wiki

Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack
- **Properties**:
  - Incomplete (stuck in infinite tree...)

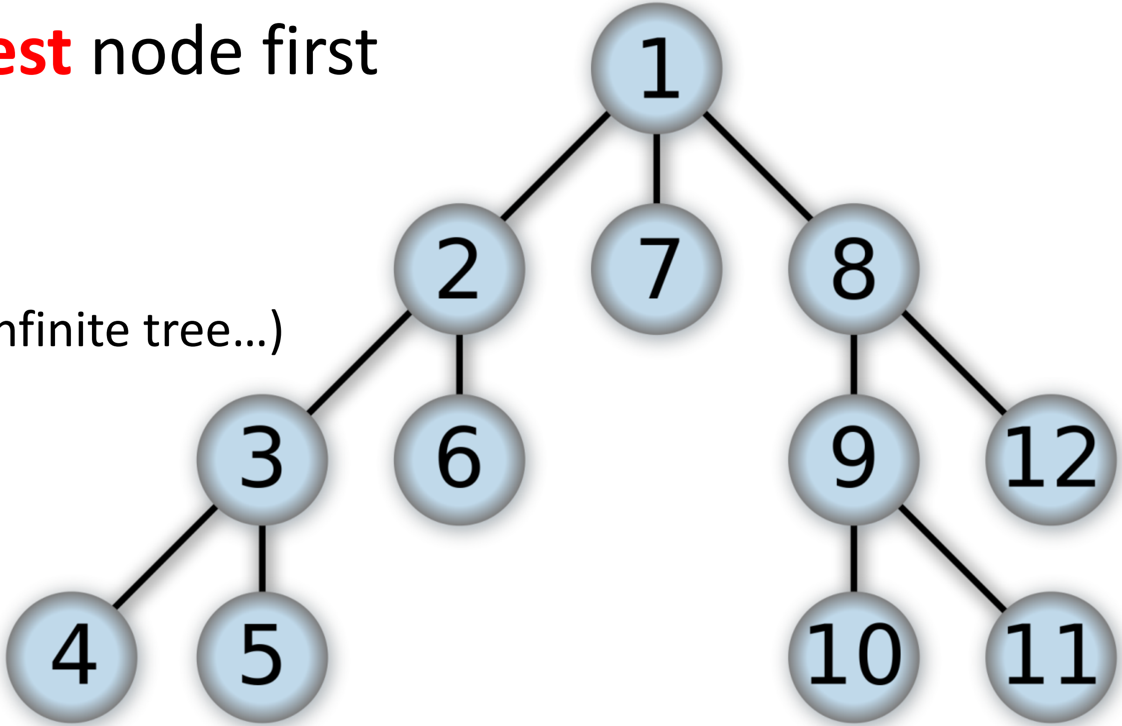


Wiki

Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack
- **Properties**:
  - Incomplete (stuck in infinite tree...)
  - Suboptimal

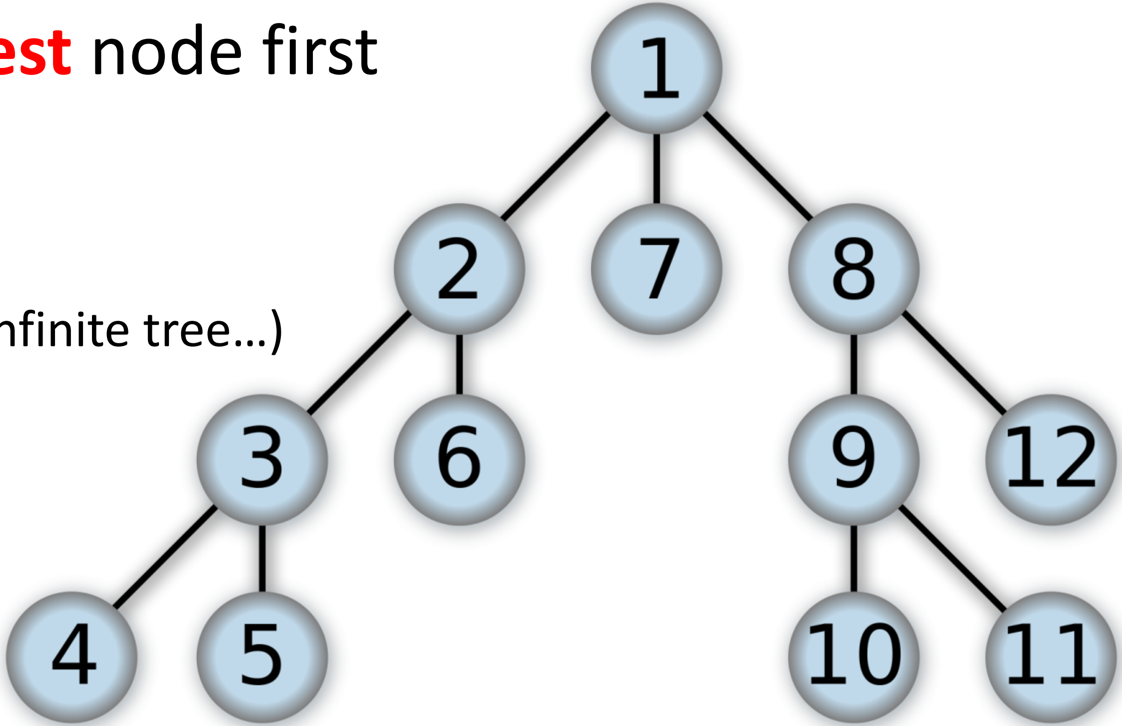


Wiki

Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack
- **Properties**:
  - Incomplete (stuck in infinite tree…)
  - Suboptimal
  - Time O(b^m)



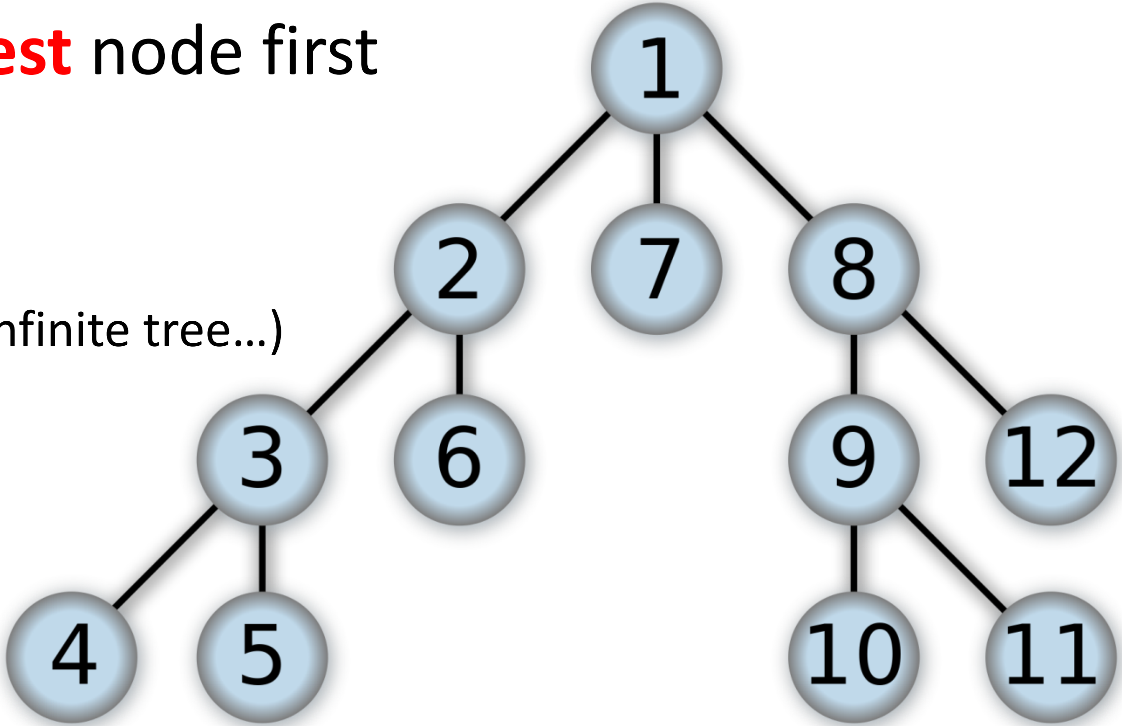
Wiki

Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack
- **Properties**:
  - Incomplete (stuck in infinite tree...)
  - Suboptimal
  - Time O(b^m)

Max Depth



Wiki

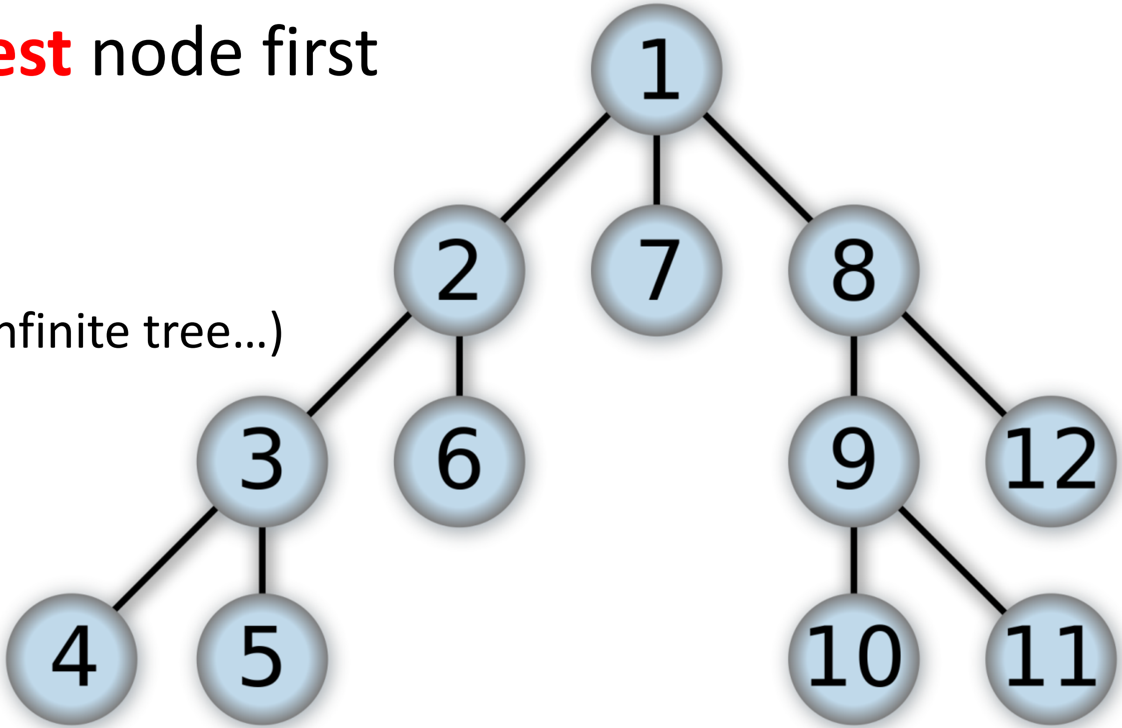
Depth-First Search

Recall: expand **deepest** node first

- Data structure: stack
- **Properties**:
  - Incomplete (stuck in infinite tree...)
  - Suboptimal
  - Time $O(b^m)$

Max Depth

  - Space $O(bm)$



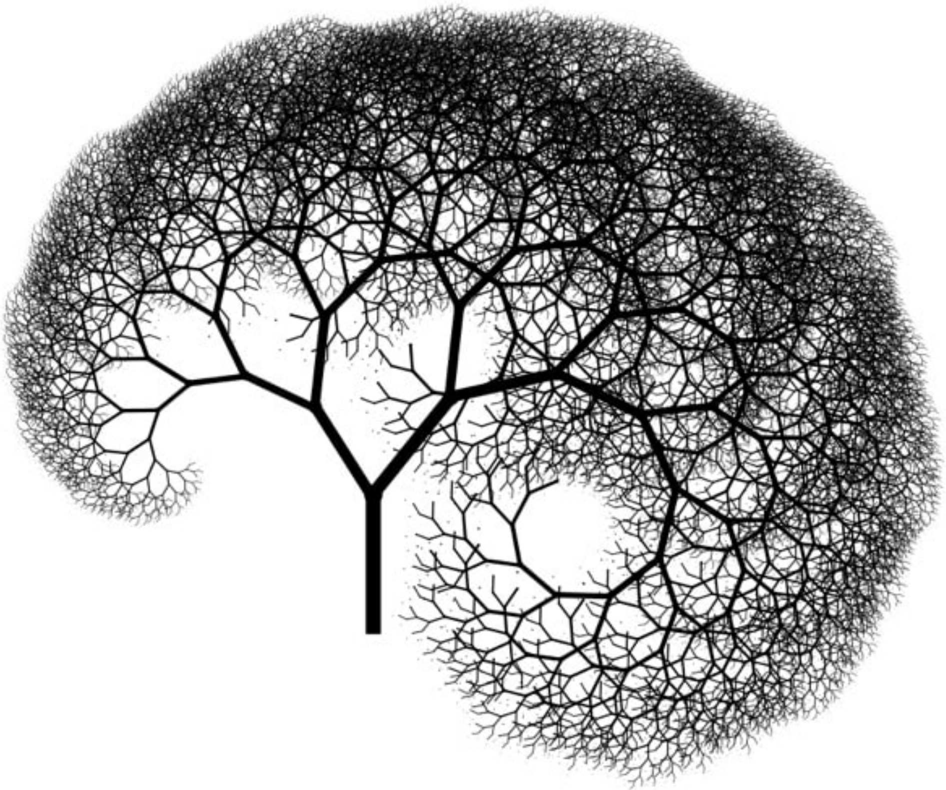
Wiki

Iterative Deepening DFS

Repeated limited DFS

Iterative Deepening DFS

Repeated limited DFS

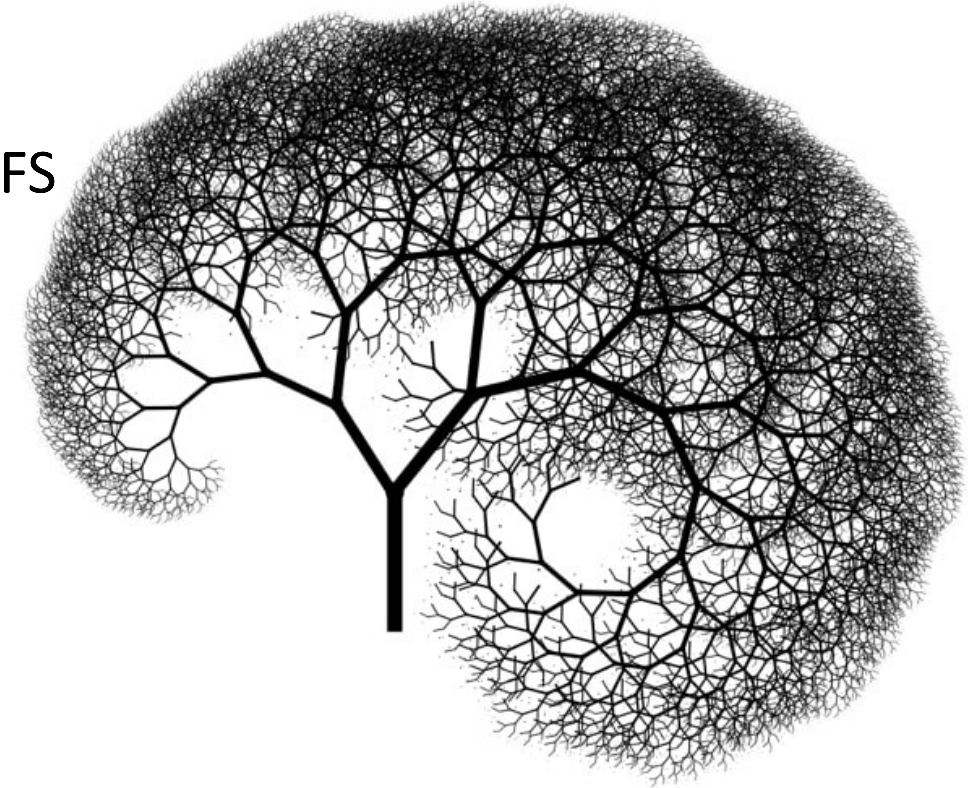


Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS

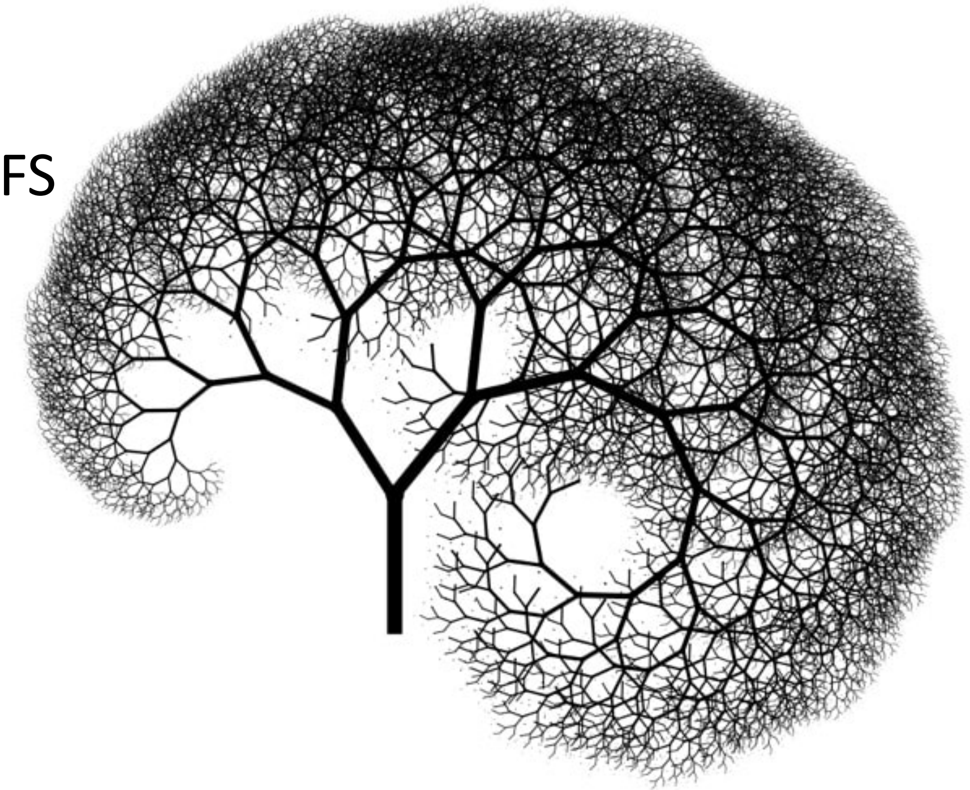


Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:

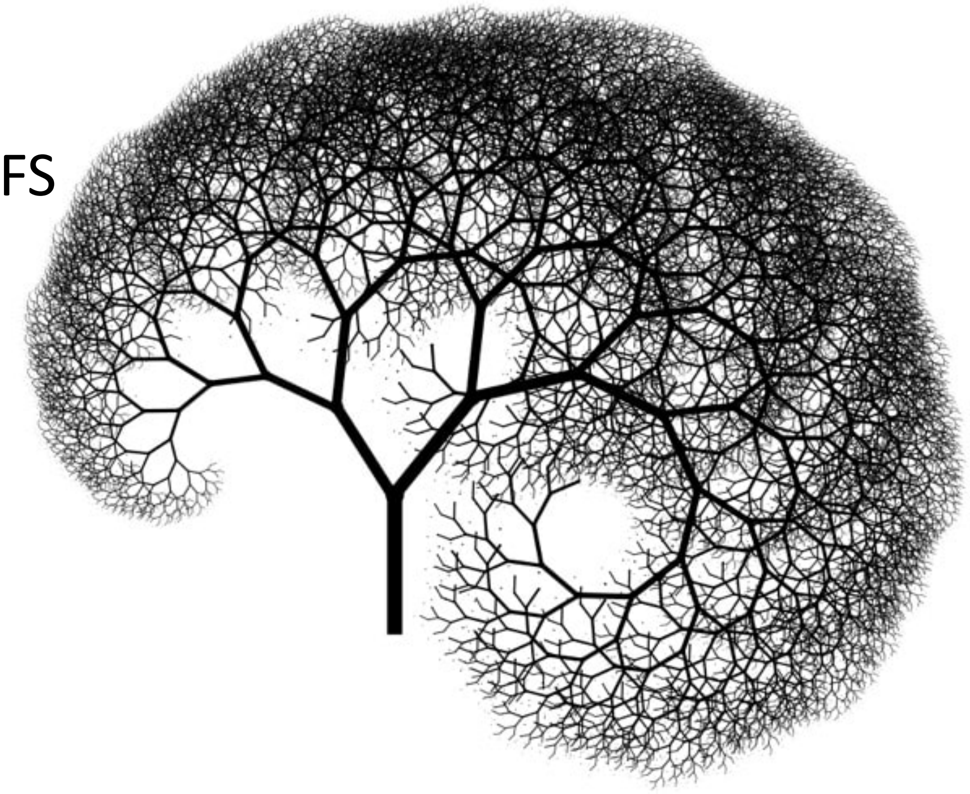


Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:
  - Complete

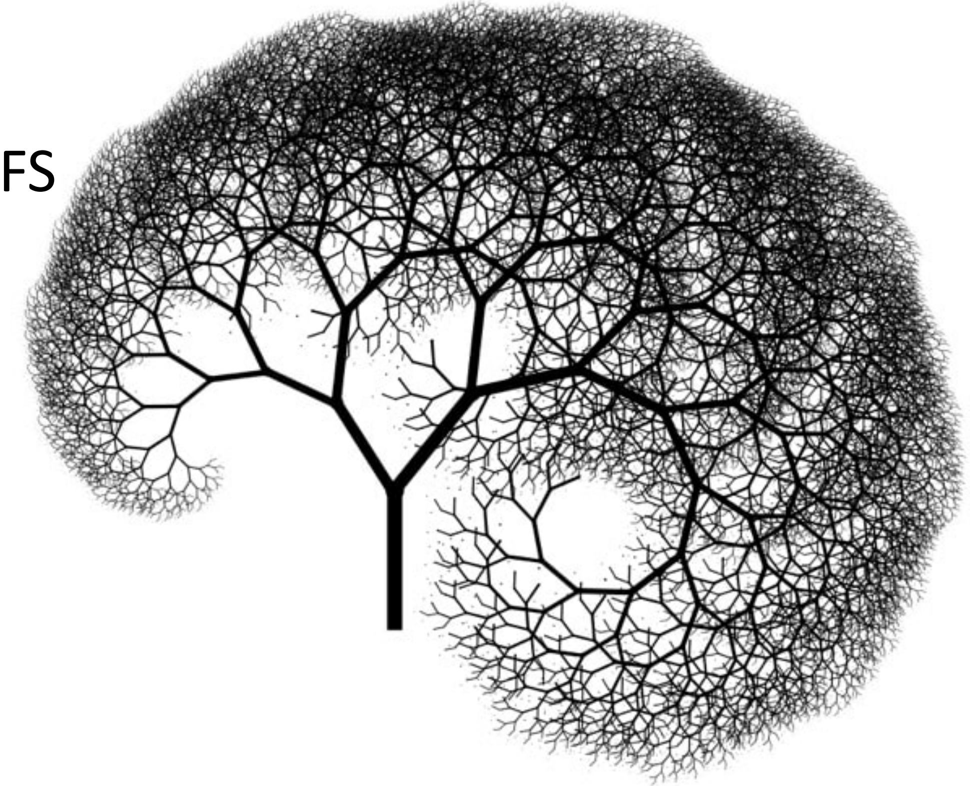


Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)

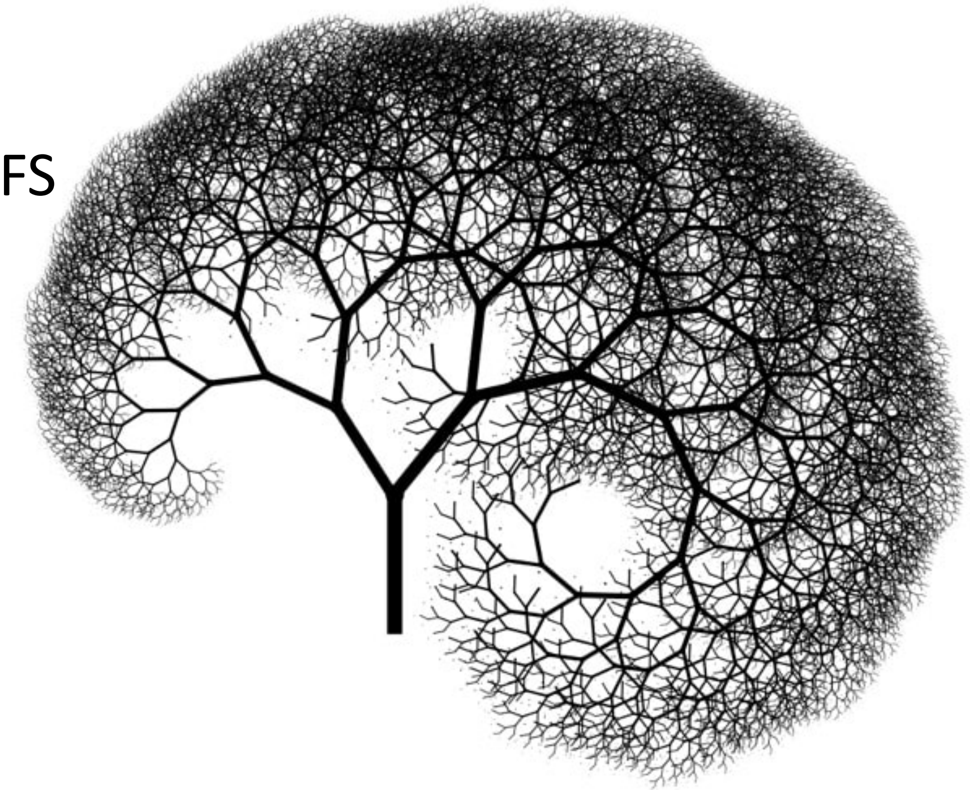


Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time $O(b^d)$

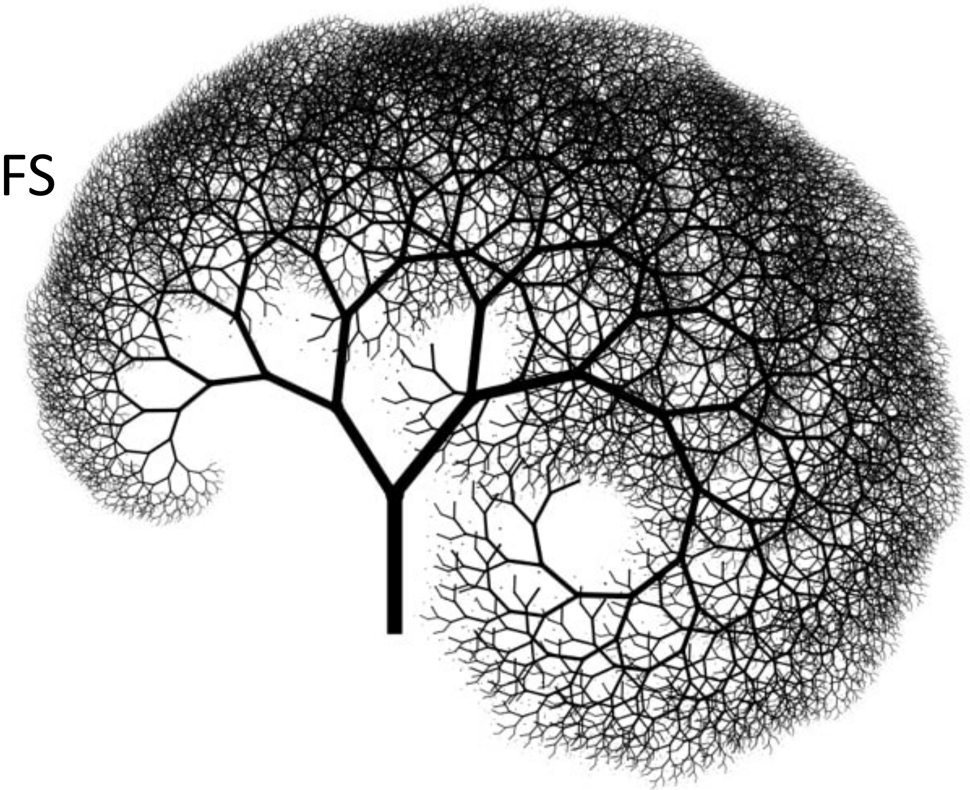


Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time O(b^d)
  - Space O(bd)



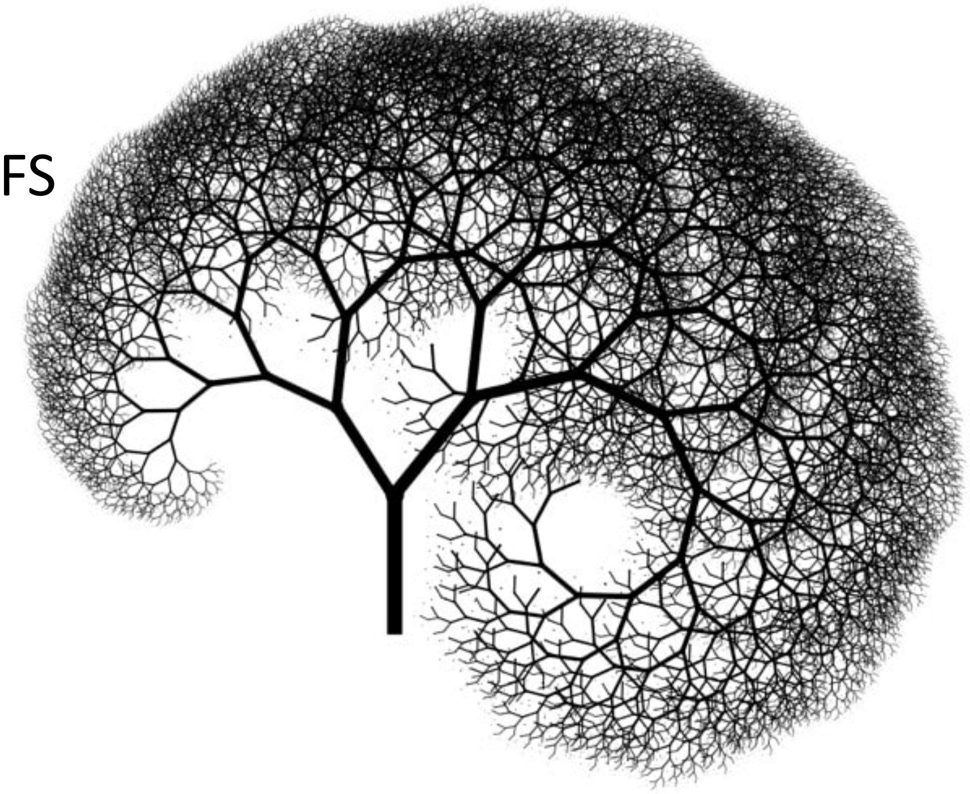
Fractalsaco

Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time O(b^d)
  - Space O(bd)

A good option!



Fractalsaco

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s*.
- Successors.

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s*.
- Successors.

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*
- Successors.

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*
- Successors.

Informed search. Know:

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*
- Successors.

Informed search. Know:

- All uninformed search properties, plus

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*
- Successors.

Informed search. Know:

- All uninformed search properties, plus
- Heuristic **h(s)** from s to goal.

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*
- Successors.

Informed search. Know:

- All uninformed search properties, plus
- Heuristic **h(s)** from s to goal.

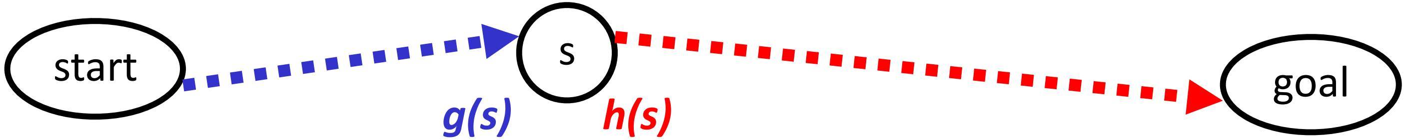
Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g(s)*** from start to state *s.*
- Successors.

Informed search. Know:

- All uninformed search properties, plus
- Heuristic **h(s)** from s to goal.



Informed Search

Informed search. Know:

Informed Search

Informed search. Know:

- All uninformed search properties, plus

Informed Search

Informed search. Know:

- All uninformed search properties, plus
- Heuristic ***h*(*s*)** from *s* to goal.

Informed Search

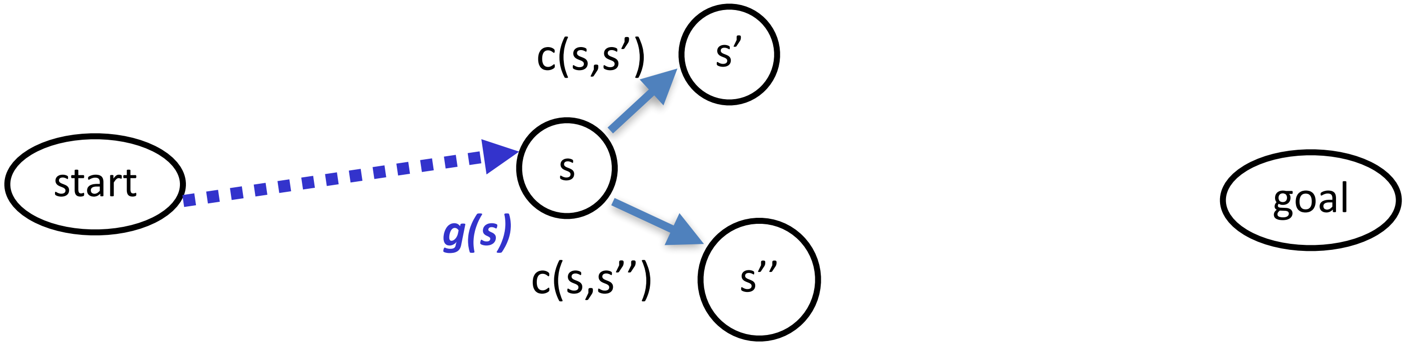
Informed search. Know:

- All uninformed search properties, plus
- Heuristic ***h*(*s*)** from *s* to goal.

Informed Search

Informed search. Know:

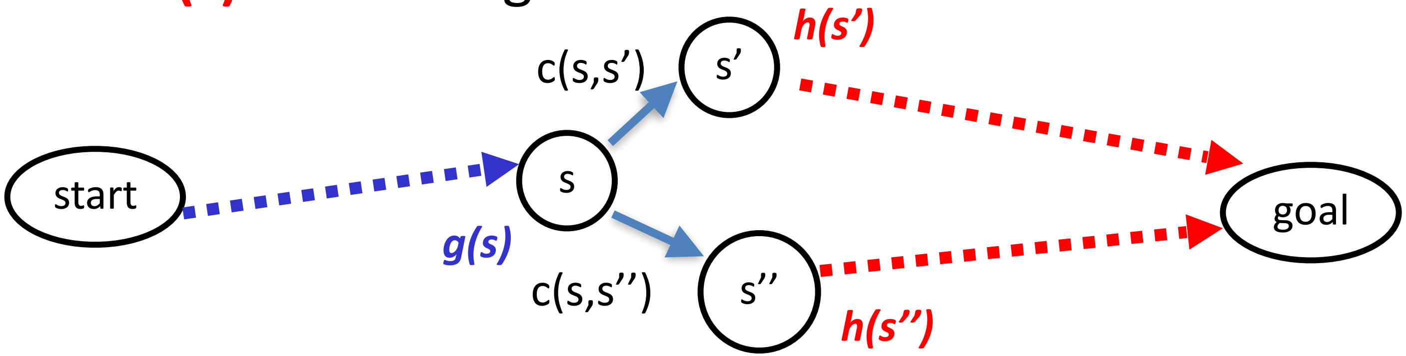
- All uninformed search properties, plus
- Heuristic ***h(s)*** from *s* to goal.



Informed Search

Informed search. Know:

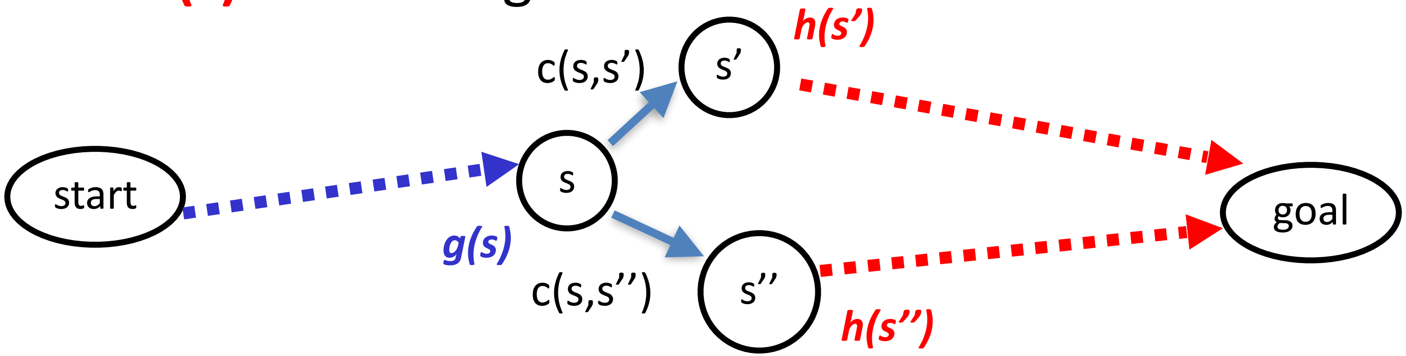
- All uninformed search properties, plus
- Heuristic ***h*(*s*)** from *s* to goal.



Informed Search

Informed search. Know:

- All uninformed search properties, plus
- Heuristic ***h*(*s*)** from *s* to goal.



- Goal: **speed up search.**

Using the Heuristic

Recall uniform-cost search

Using the Heuristic

Recall uniform-cost search

- We store potential next states with a priority queue

Using the Heuristic

Recall uniform-cost search

- We store potential next states with a priority queue
- Expand the state with the smallest $g(s)$

Using the Heuristic

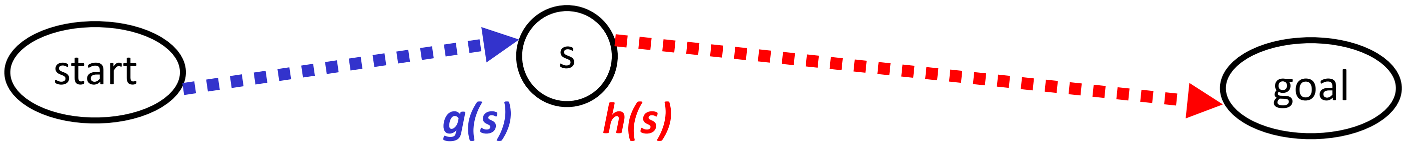
Recall uniform-cost search

- We store potential next states with a priority queue
- Expand the state with the smallest $g(s)$
  - $g(s)$ "first-half-cost"

Using the Heuristic

Recall uniform-cost search

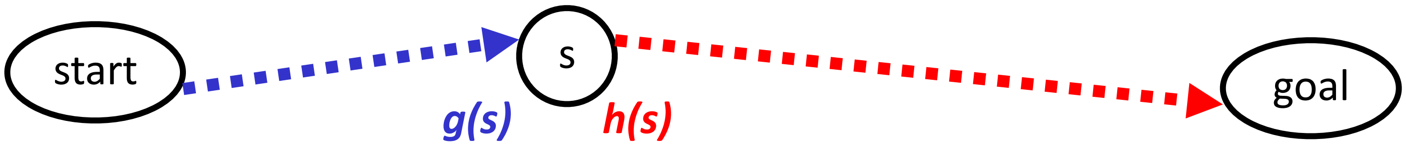
- We store potential next states with a priority queue
- Expand the state with the smallest ***g*(*s*)**
  - ***g*(*s*)** "first-half-cost"



Using the Heuristic

Recall uniform-cost search

- We store potential next states with a priority queue
- Expand the state with the smallest ***g*(*s*)**
  - ***g*(*s*)** "first-half-cost"

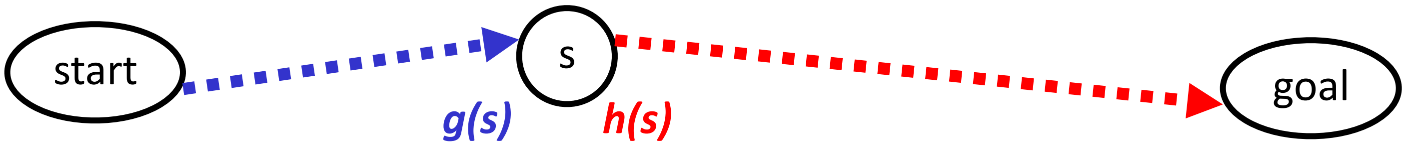


- Now let's use the heuristic ("second-half-cost")

Using the Heuristic

Recall uniform-cost search

- We store potential next states with a priority queue
- Expand the state with the smallest ***g(s)***
  - ***g(s)*** "first-half-cost"



- Now let's use the heuristic ("second-half-cost")
  - Several possible approaches: let's see what works

Attempt 1: Best-First Greedy

One approach: just use ***h*(*s*)** alone

Attempt 1: Best-First Greedy

One approach: just use $h(s)$ alone

- Specifically, expand the state with smallest $h(s)$

Attempt 1: Best-First Greedy

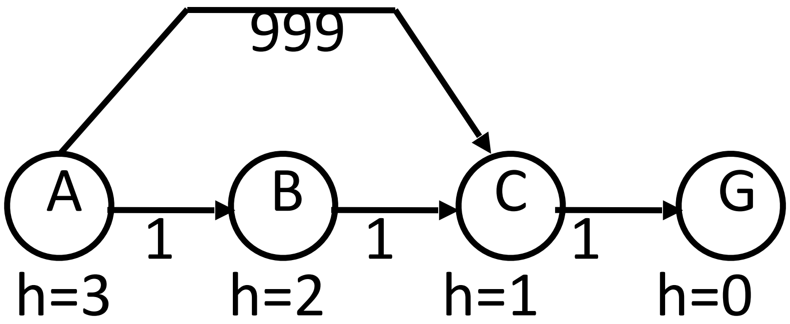
One approach: just use ***h*(*s*)** alone

- Specifically, expand the state with smallest ***h*(*s*)**
- This isn't a good idea. Why?

Attempt 1: Best-First Greedy

One approach: just use ***h*(*s*)** alone

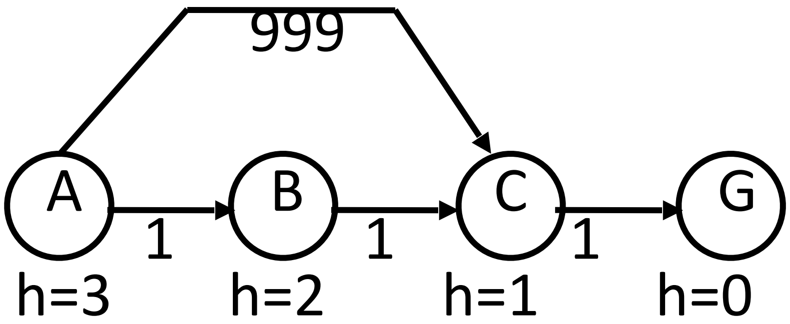
- Specifically, expand the state with smallest ***h*(*s*)**
- This isn't a good idea. Why?



Attempt 1: Best-First Greedy

One approach: just use ***h*(*s*)** alone

- Specifically, expand the state with smallest ***h*(*s*)**
- This isn't a good idea. Why?



- Not optimal! **Get** A → C → G. **Want**: A →B → C → G

Attempt 2: **A Search**

Next approach: use both $g(s) + h(s)$

Attempt 2: **A Search**

Next approach: use both $g(s) + h(s)$

- Specifically, expand state with smallest $g(s) + h(s)$

Attempt 2: **A Search**

Next approach: use both $g(s) + h(s)$

- Specifically, expand state with smallest $g(s) + h(s)$
- Again, use a priority queue

Attempt 2: **A Search**

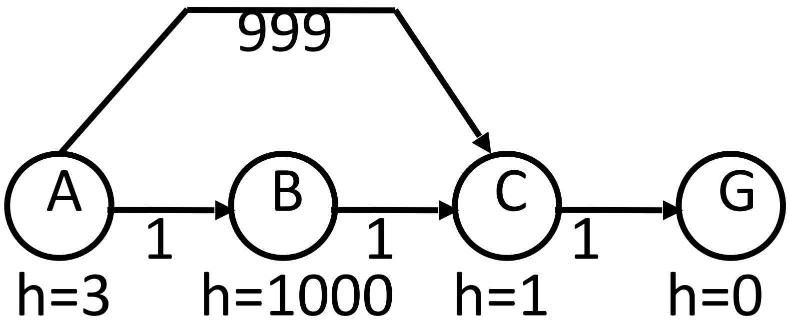
Next approach: use both $g(s) + h(s)$

- Specifically, expand state with smallest $g(s) + h(s)$
- Again, use a priority queue
- Called **"A" search**

Attempt 2: **A Search**

Next approach: use both ***g*(*s*)** + ***h*(*s*)**

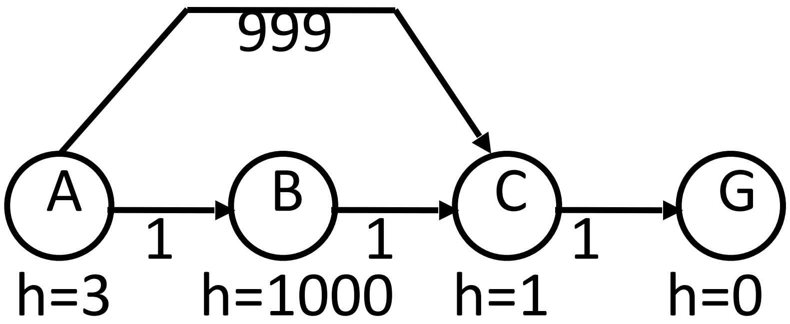
- Specifically, expand state with smallest ***g*(*s*)** + ***h*(*s*)**
- Again, use a priority queue
- Called **"A" search**



Attempt 2: **A Search**

Next approach: use both ***g*(*s*)** + ***h*(*s*)**

- Specifically, expand state with smallest ***g*(*s*)** + ***h*(*s*)**
- Again, use a priority queue
- Called **"A" search**



- **Still not optimal!** (Does work for former example).

Attempt 3: **A* Search**

Same idea, use $g(s)$ + $h(s)$, with one **requirement**

Attempt 3: **A* Search**

Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that ***h*(*s*)** ≤ ***h***(*s*)** where **h*(s)** is true cost from s to goal.

Attempt 3: **A* Search**

Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that ***h*(*s*)** ≤ ***h***(*s*)** where **h*(s)** is true cost from s to goal.
- If heuristic has this property, it is called "admissible"

Attempt 3: **A* Search**

Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that ***h*(*s*)** ≤ ***h***(*s*)** where **h*(s)** is true cost from s to goal.
- If heuristic has this property, it is called "admissible"
  - Optimistic! Never over-estimates

Attempt 3: **A* Search**

Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that ***h*(*s*)** ≤ ***h**(*s*)** where **h*(s)** is true cost from s to goal.
- If heuristic has this property, it is called “admissible”
  - Optimistic! Never over-estimates
- Still need ***h*(*s*)** ≥ 0

Attempt 3: **A* Search**

Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that ***h*(*s*)** ≤ ***h***(*s*)** where **h*(s)** is true cost from s to goal.
- If heuristic has this property, it is called "admissible"
  - Optimistic! Never over-estimates
- Still need ***h*(*s*)** ≥ 0
  - Negative heuristics can lead to strange behavior

Attempt 3: **A* Search**

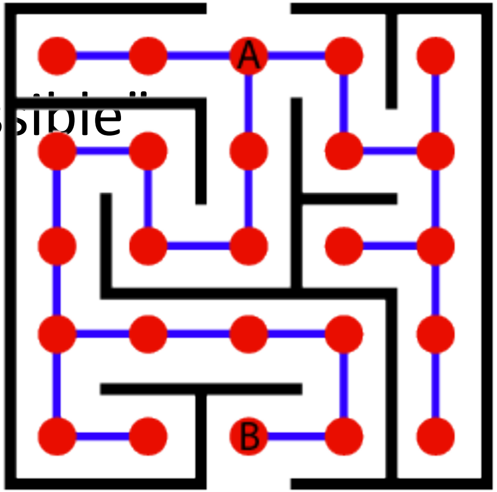
Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that $h(s) \leq h^*(s)$ where $h^*(s)$ is true cost from s to goal.
- If heuristic has this property, it is called "admissible"
  - Optimistic! Never over-estimates
- Still need $h(s) \geq 0$
  - Negative heuristics can lead to strange behavior
- This is **A* search**

Attempt 3: **A* Search**

Same idea, use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

- Demand that ***h*(*s*)** ≤ ***h**(*s*)** where **h*(s)** is true cost from s to goal.
- If heuristic has this property, it is called "admissible"
  - Optimistic! Never over-estimates
- Still need ***h*(*s*)** ≥ 0
  - Negative heuristics can lead to strange behavior
- This is **A* search**



V. Batoćanin

Attempt 3: **A* Search**

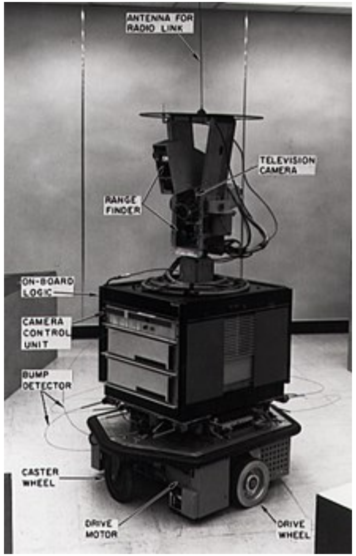
Origins: robots and planning

Animation: finding a path around obstacle

Credit: Wiki

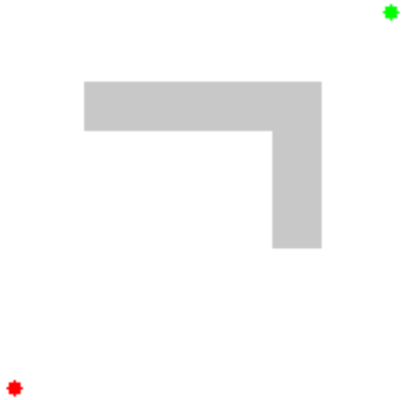
Attempt 3: **A* Search**

Origins: robots and planning



Shakey the Robot, 1960's

Credit: Wiki

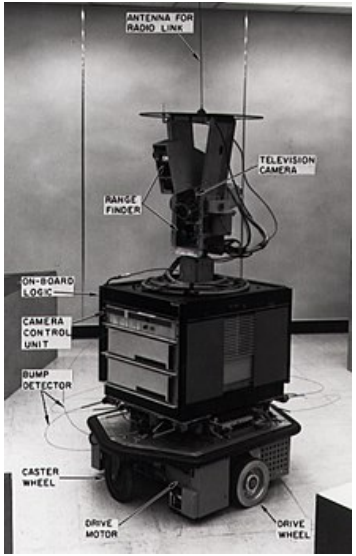


Animation: finding a path around obstacle

Credit: Wiki

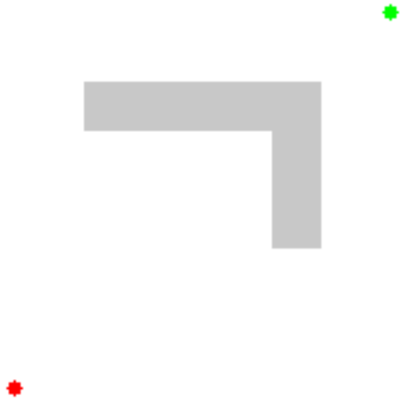
Attempt 3: **A* Search**

Origins: robots and planning



Shakey the Robot, 1960’s

Credit: Wiki



Animation: finding a path around obstacle

Credit: Wiki

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!)**

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!)**

- Example: **8 Game**

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!)**

- Example: **8 Game**

Example State

1		5
2	6	3
7	4	8

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!)**

- Example: **8 Game**

Example State

1		5
2	6	3
7	4	8

Goal State

1	2	3
4	5	6
7	8	

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!**)

- Example: **8 Game**

Example State

1		5
2	6	3
7	4	8

Goal State

1	2	3
4	5	6
7	8	

- One useful approach: **relax constraints**

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!**)

- Example: **8 Game**

Example State

1		5
2	6	3
7	4	8

Goal State

1	2	3
4	5	6
7	8	

- One useful approach: **relax constraints**
  - ***h*(*s*)** = number of tiles in wrong position

Admissible Heuristic Functions

Have to be careful to ensure admissibility (**optimism!**)

- Example: **8 Game**

Example State

1		5
2	6	3
7	4	8

Goal State

1	2	3
4	5	6
7	8	

- One useful approach: **relax constraints**
  - ***h*(*s*)** = number of tiles in wrong position
    - allows tiles to fly to destination in a single step

Break & Quiz

Q 1.1: Consider finding the fastest driving route from one US city to another. Measure cost as the number of hours driven when driving at the speed limit. Let ***h*(*s*)** be the number of hours needed to ride a bike from city s to your destination. ***h*(*s*)** is

- A. An admissible heuristic
- B. Not an admissible heuristic

Break & Quiz

Q 1.1: Consider finding the fastest driving route from one US city to another. Measure cost as the number of hours driven when driving at the speed limit. Let ***h*(*s*)** be the number of hours needed to ride a bike from city s to your destination. ***h*(*s*)** is

- A. An admissible heuristic
- **B. Not an admissible heuristic**

Break & Quiz

Q 1.1: Consider finding the fastest driving route from one US city to another. Measure cost as the number of hours driven when driving at the speed limit. Let ***h***(***s***) be the number of hours needed to ride a bike from city s to your destination. ***h***(***s***) is

- A. An admissible heuristic No: riding your bike takes longer.
- **B. Not an admissible heuristic**

Break & Quiz

Q 1.2: Which of the following are admissible heuristics?

(i) $h(s) = h^*(s)$

(ii) $h(s) = \max(2, h^*(s))$

(iii) $h(s) = \min(2, h^*(s))$

(iv) $h(s) = h^*(s)-2$

(v) $h(s) = \mathrm{sqrt}(h^*(s))$

- A. All of the above
- B. (i), (iii), (iv)
- C. (i), (iii)
- D. (i), (iii), (v)

Break & Quiz

Q 1.2: Which of the following are admissible heuristics?

(i) $h(s) = h^*(s)$

(ii) $h(s) = \max(2, h^*(s))$

(iii) $h(s) = \min(2, h^*(s))$

(iv) $h(s) = h^*(s)-2$

(v) $h(s) = \text{sqrt}(h^*(s))$

- A. All of the above
- B. (i), (iii), (iv)
- **C. (i), (iii)**
- D. (i), (iii), (v)

Break & Quiz

Q 1.2: Which of the following are admissible heuristics?

(i) $h(s) = h^*(s)$

(ii) $h(s) = \max(2, h^*(s))$ No: $h(s)$ might be too big

(iii) $h(s) = \min(2, h^*(s))$

(iv) $h(s) = h^*(s)-2$ No: $h(s)$ might be negative

(v) $h(s) = \text{sqrt}(h^*(s))$ No: if $h^*(s) < 1$ then $h(s)$ *is bigger*

- A. All of the above
- B. (i), (iii), (iv)
- **C. (i), (iii)**
- D. (i), (iii), (v)

Heuristic Function Tradeoffs

Heuristic Function Tradeoffs

Dominance: h_2 dominates h_1 if for all states *s*,

Heuristic Function Tradeoffs

Dominance: h_2 dominates h_1 if for all states s,

$$h_1(s) \leq h_2(s) \leq h^*(s)$$

Heuristic Function Tradeoffs

Dominance: h_2 dominates h_1 if for all states s,

$$h_1(s) \leq h_2(s) \leq h^*(s)$$

- **Idea**: we want to be as close to h^* as possible
  - But not over! **Must under-estimate true cost.**

Heuristic Function Tradeoffs

Dominance: h_2 dominates h_1 if for all states *s*,

$$h_1(s) \leq h_2(s) \leq h^*(s)$$

- **Idea**: we want to be as close to h^* as possible
  - But not over! **Must under-estimate true cost.**
- **Tradeoff**: being very close might require a very complex heuristic, expensive computation
  - Might be better off with cheaper heuristic & expand more nodes.

A* Termination

When should A* **stop**?

A* Termination

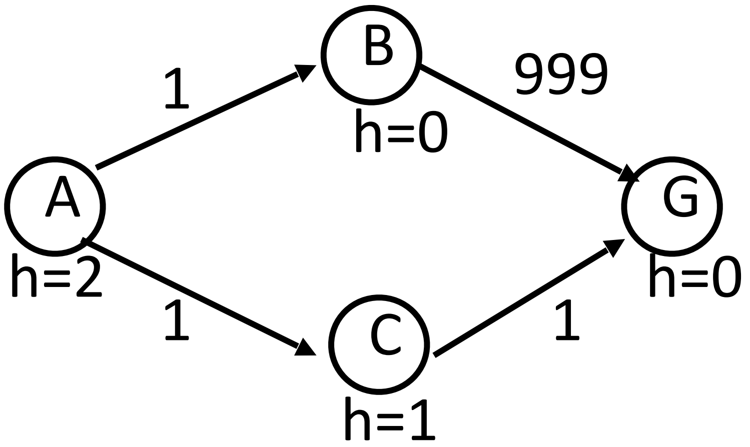
When should A* **stop**?

- One idea: as soon as we reach goal state?

A* Termination

When should A* **stop**?

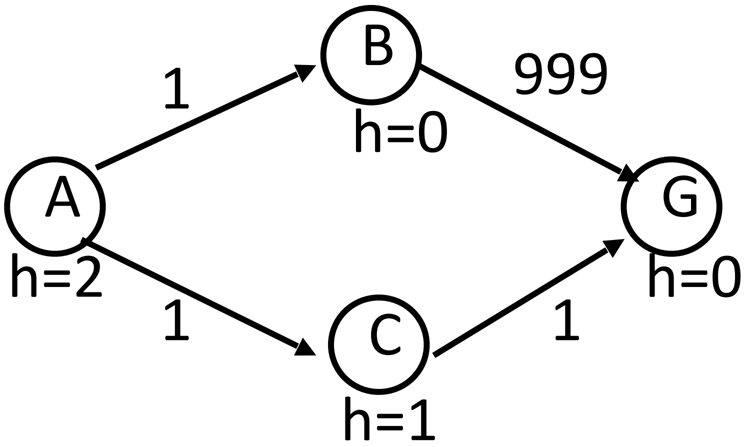
- One idea: as soon as we reach goal state?



A* Termination

When should A* **stop**?

- One idea: as soon as we reach goal state?



- ***h*** is admissible, but note that we get A →B → G (**cost 1000**)!

A* Termination

When should A* **stop**?

A* Termination

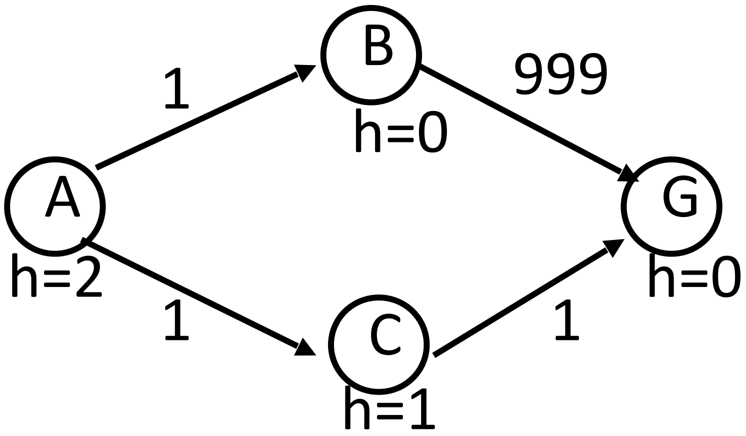
When should A* **stop**?

- **Rule**: terminate **when a goal is popped** from queue.

A* Termination

When should A* **stop**?

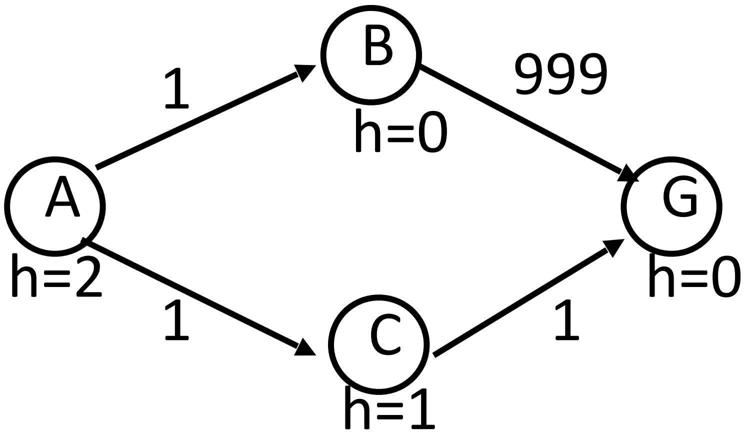
- **Rule**: terminate **when a goal is popped** from queue.



A* Termination

When should A* **stop**?

- **Rule**: terminate **when a goal is popped** from queue.



- Note: taking ***h*** =0 reduces to uniform cost search rule.

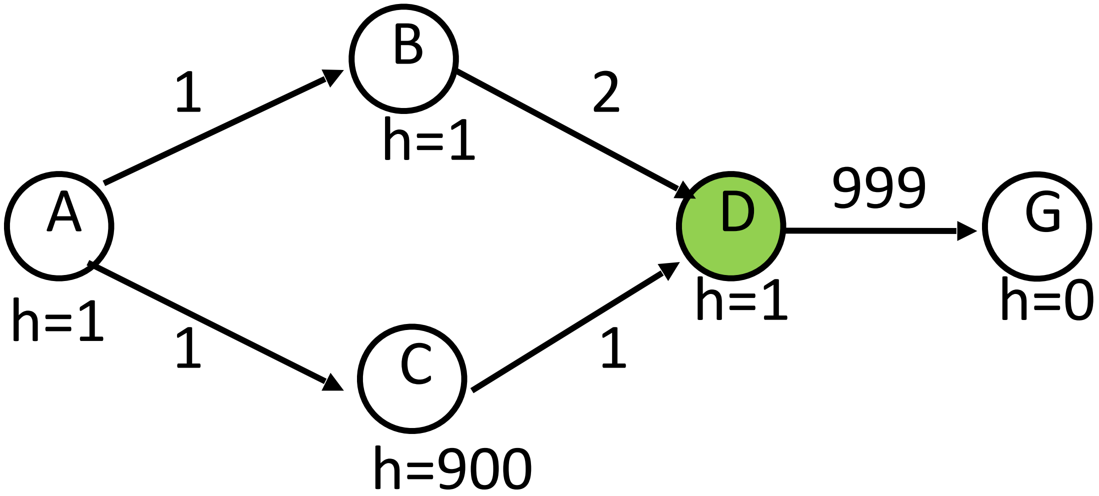
A* Revisiting Expanded States

A* Revisiting Expanded States

Possible to revisit an expanded state, get a shorter path:

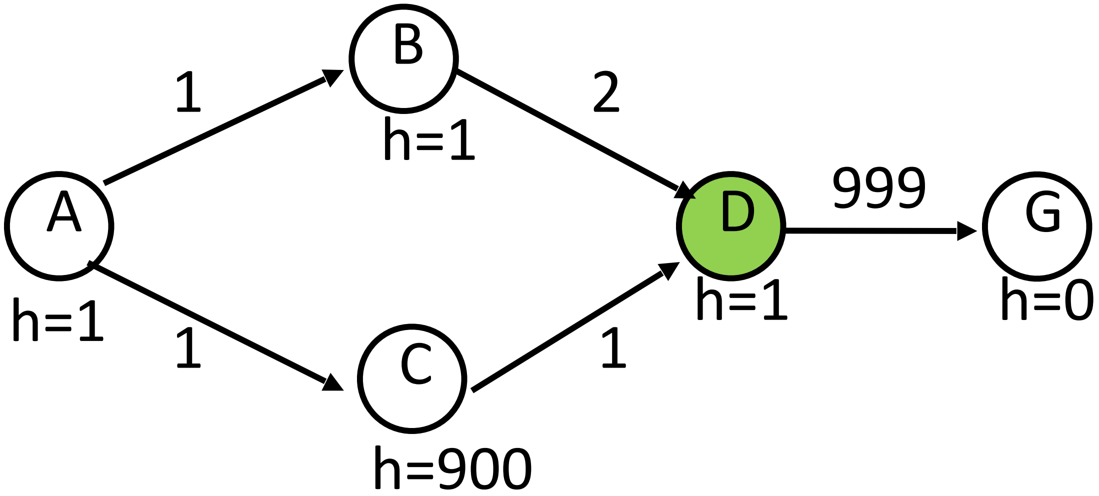
A* Revisiting Expanded States

Possible to revisit an expanded state, get a shorter path:



A* Revisiting Expanded States

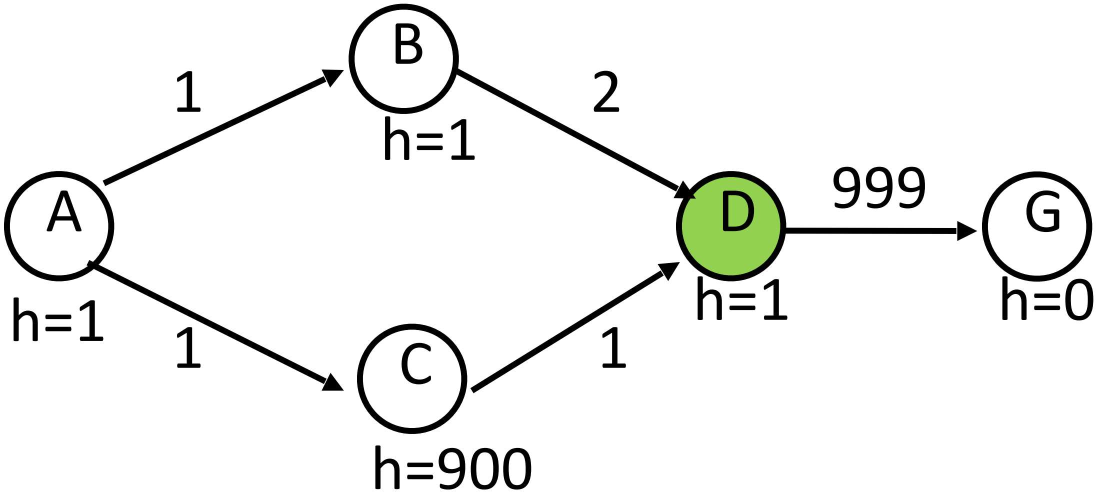
Possible to revisit an expanded state, get a shorter path:



- Put D back into priority queue, smaller **g**+**h.**

A* Revisiting Expanded States

Possible to revisit an expanded state, get a shorter path:



- Put D back into priority queue, smaller **g**+**h.**
- **Note:** uninformed search methods will not revisit expanded states.

A* Full Algorithm

1. Put the start state S on the priority queue. We call the priority queue OPEN
2. If OPEN is empty, exit with failure
3. Remove from OPEN and place on CLOSED a node n for which f(n) is minimum (note that f(n)=g(n)+h(n))
4. If n is a goal node, exit (recover path by tracing back pointers from n to S)
5. Expand n, generating all successors and attach to pointers back to n. For each successor n' of n
    1. If n' is not already on OPEN or CLOSED compute h(n'), g(n')=g(n)+ c(n,n'), f(n')=g(n')+h(n’), and place it on OPEN.
    2. If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n'. If so, then:
        1. Redirect pointers backward from n' along path yielding lower g(n').
        2. Put n' on OPEN.
    3. If g(n') is not lower for the new version, do nothing.
6. Goto 2.

A* Full Algorithm

1. Put the start state S on the priority queue. We call the priority queue OPEN

2. If OPEN is empty, exit with failure

States we have already expanded

3. Remove from OPEN and place on CLOSED a node n for which f(n) is minimum (note that f(n)=g(n)+h(n))

4. If n is a goal node, exit (recover path by tracing back pointers from n to S)

5. Expand n, generating all successors and attach to pointers back to n. For each successor n' of n

   1. If n' is not already on OPEN or CLOSED compute h(n'), g(n')=g(n)+ c(n,n'), f(n')=g(n')+h(n’), and place it on OPEN.
   2. If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n'. If so, then:
      1. Redirect pointers backward from n' along path yielding lower g(n').
      2. Put n' on OPEN.
   3. If g(n') is not lower for the new version, do nothing.

6. Goto 2.

A* Analysis

Some properties:

A* Analysis

Some properties:

- Terminates!

A* Analysis

Some properties:

- Terminates!
- A* can use **lots of memory**:

A* Analysis

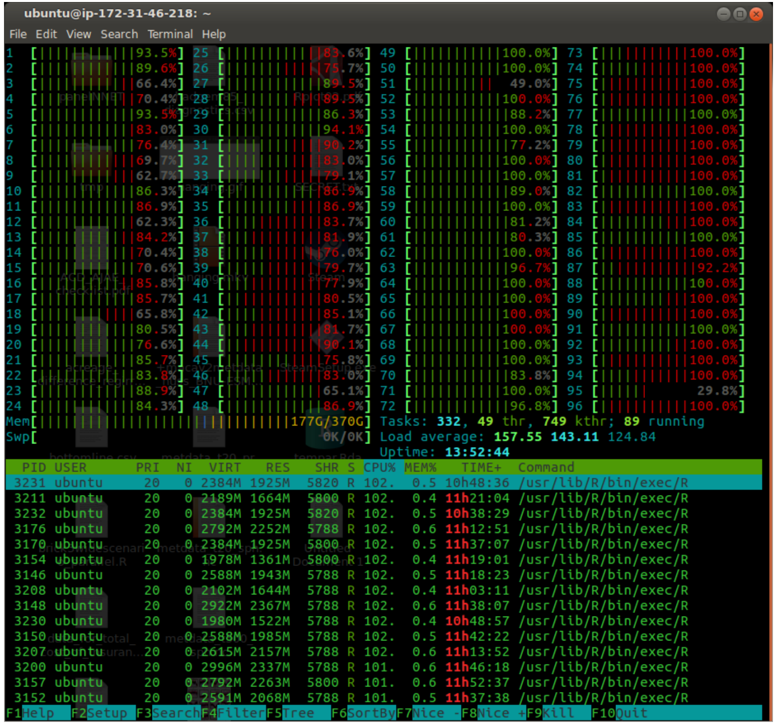
Some properties:

- Terminates!
- A* can use **lots of memory**:
  - O(# states).

A* Analysis

Some properties:

- Terminates!
- A* can use **lots of memory**:
    - O(# states).
- Will run out on large problems.

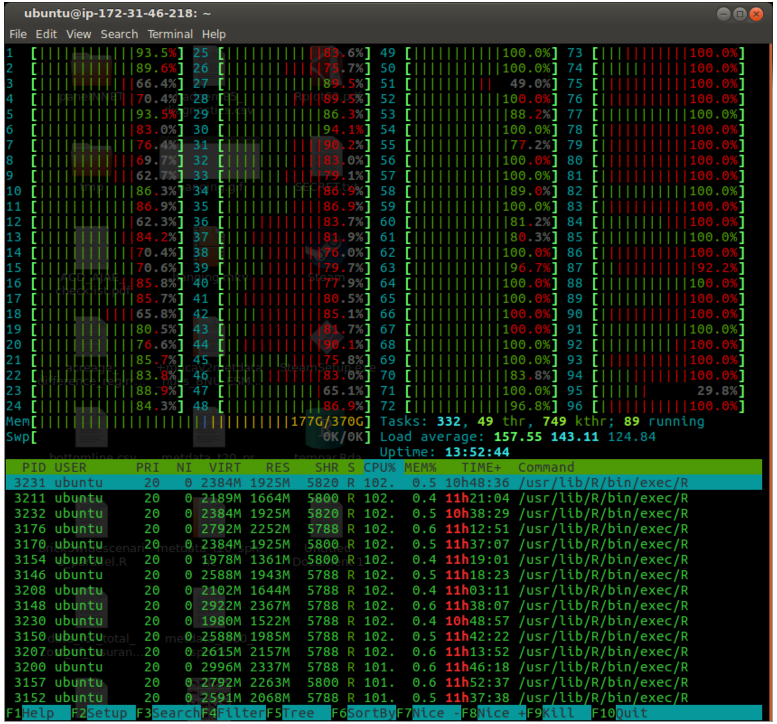


A* Analysis

Some properties:

- Terminates!
- A* can use **lots of memory**:
  - O(# states).
- Will run out on large problems.

- Next, we will consider some alternatives to deal with this.



Break & Quiz

Q 2.1: Consider two heuristics for the 8 puzzle problem. h_1 is the number of tiles in wrong position. h_2 is the l_1/Manhattan distance between the tiles and the goal location. How do h_1 and h_2 relate?

- A. h_2 dominates h_1
- B. h_1 dominates h_2
- C. Neither dominates the other

Break & Quiz

Q 2.1: Consider two heuristics for the 8 puzzle problem. h_1 is the number of tiles in wrong position. h_2 is the l_1/Manhattan distance between the tiles and the goal location. How do h_1 and h_2 relate?

- **A. h_2 dominates h_1**
- B. h_1 dominates h_2
- C. Neither dominates the other

Break & Quiz

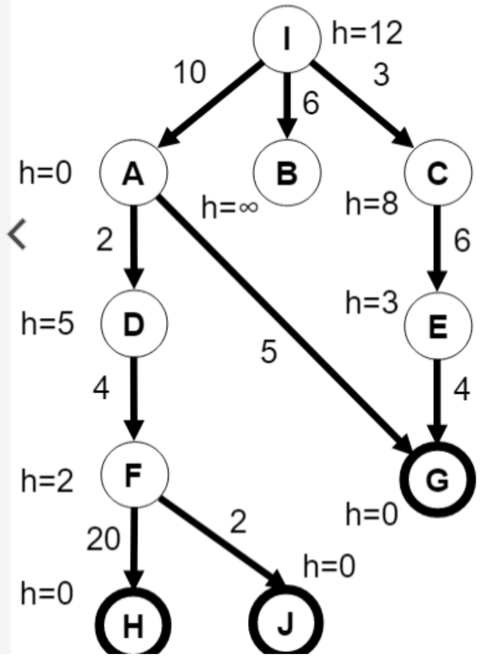
Q 2.1: Consider two heuristics for the 8 puzzle problem. ***h_1*** is the number of tiles in wrong position. ***h_2*** is the l_1/Manhattan distance between the tiles and the goal location. How do ***h_1*** and ***h_2*** relate?

- **A. *h_2* dominates *h_1***
- B. ***h_1*** dominates ***h_2*** (No: ***h_1*** is a distance where each entry is at most 1, ***h_2*** can be greater)
- C. Neither dominates the other

Break & Quiz

Q 2.2: Consider the state space graph below. Goal states have bold borders. ***h*(*s*)** is show next to each node. What node will be expanded by A* after the initial state I?

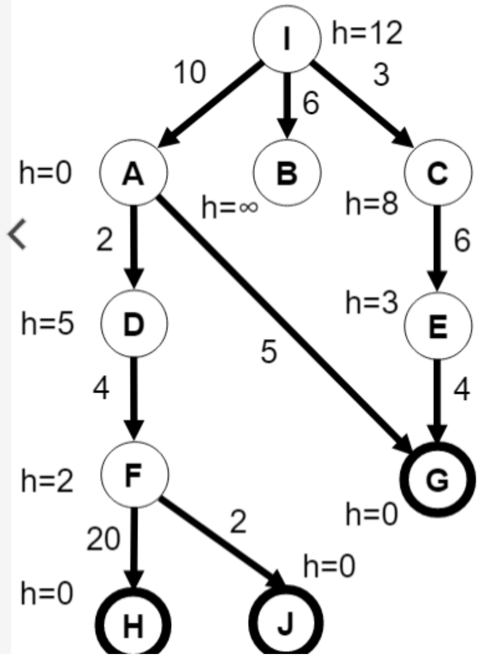
- A. A
- B. B
- C. C



Break & Quiz

Q 2.2: Consider the state space graph below. Goal states have bold borders. ***h*(*s*)** is show next to each node. What node will be expanded by A* after the initial state I?

- **A. A**
- B. B
- C. C



IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

- Bound the memory in search.

IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

- Bound the memory in search.
- At each phase, don't expand any node with $g(s) + h(s) > k$,

IDA*: Iterative Deepening A*

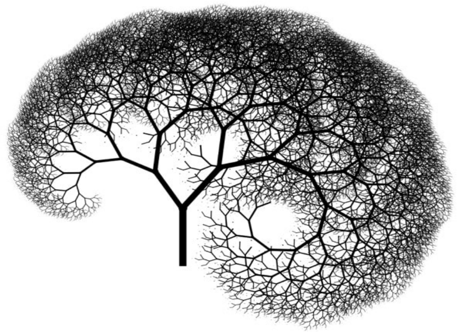
Similar idea to our earlier iterative deepening.

- Bound the memory in search.
- At each phase, don't expand any node with $\boldsymbol{g(s)} + \boldsymbol{h(s)} > k$,
  - Assuming integer costs, do this for k=0, then k=1, then k=2, and so on

IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

- Bound the memory in search.
- At each phase, don't expand any node with $g(s) + h(s) > k$,
  - Assuming integer costs, do this for k=0, then k=1, then k=2, and so on



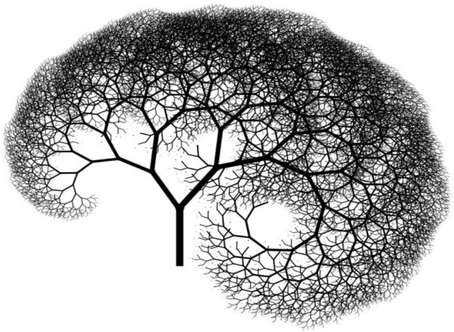
Fractalsaco

IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

- Bound the memory in search.
- At each phase, don't expand any node with $g(s) + h(s) > k$,
  - Assuming integer costs, do this for k=0, then k=1, then k=2, and so on

- Complete + optimal, might be costly time-wise



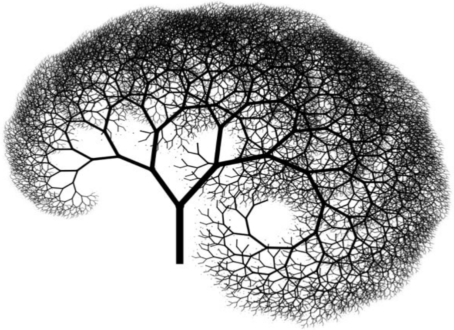
Fractalsaco

IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

- Bound the memory in search.
- At each phase, don't expand any node with $g(s) + h(s) > k$,
  - Assuming integer costs, do this for k=0, then k=1, then k=2, and so on

- Complete + optimal, might be costly time-wise
  - Revisit many nodes

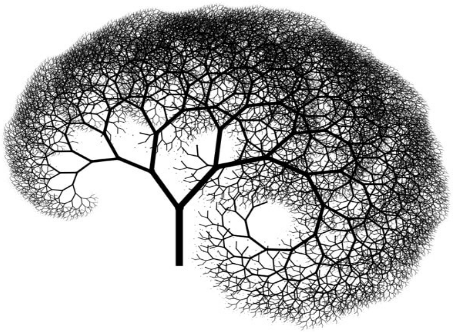


Fractalsaco

IDA*: Iterative Deepening A*

Similar idea to our earlier iterative deepening.

- Bound the memory in search.
- At each phase, don't expand any node with $g(s) + h(s) > k$,
  - Assuming integer costs, do this for k=0, then k=1, then k=2, and so on
- Complete + optimal, might be costly time-wise
  - Revisit many nodes
- Lower memory use than A*



Fractalsaco

IDA*: Properties

How many restarts do we expect?

IDA*: Properties

How many restarts do we expect?

- With integer costs, optimal solution C^*, at most C^*

IDA*: Properties

How many restarts do we expect?

- With integer costs, optimal solution C^*, at most C^*

What about non-integer costs?

IDA*: Properties

How many restarts do we expect?

- With integer costs, optimal solution C^*, at most C^*

What about non-integer costs?

- Initial threshold k. Use the same rule for non-expansion

IDA*: Properties

How many restarts do we expect?

- With integer costs, optimal solution C^*, at most C^*

What about non-integer costs?

- Initial threshold k. Use the same rule for non-expansion
- Set new k to be the min $g(s) + h(s)$ for non-expanded nodes

IDA*: Properties

How many restarts do we expect?

- With integer costs, optimal solution C^*, at most C^*

What about non-integer costs?

- Initial threshold k. Use the same rule for non-expansion
- Set new k to be the min $\boldsymbol{g(s)} + \boldsymbol{h(s)}$ for non-expanded nodes
- Worst case: restarted for each state

Beam Search

General approach (beyond A* too)

Beam Search

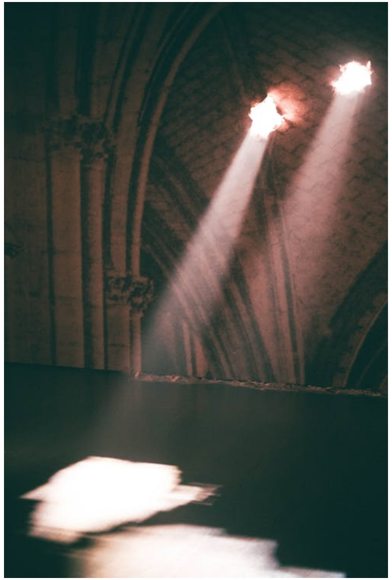
General approach (beyond A* too)

- Priority queue with fixed size *k*; beyond *k* nodes, **discard**!

Beam Search

General approach (beyond A* too)

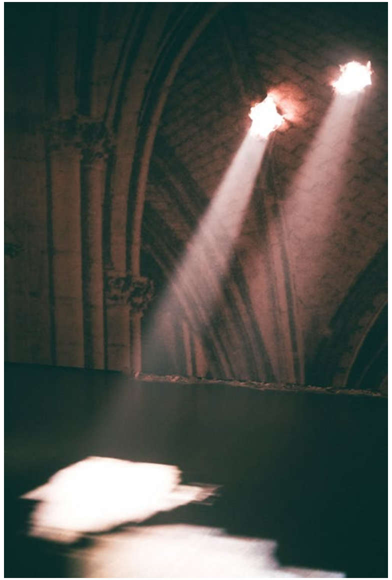
- Priority queue with fixed size *k*; beyond *k* nodes, **discard**!



Beam Search

General approach (beyond A* too)

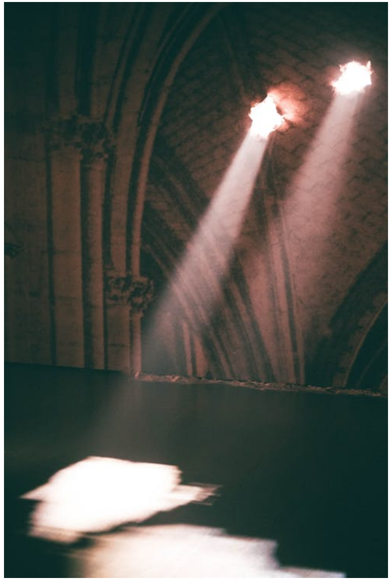
- Priority queue with fixed size *k*; beyond *k* nodes, **discard**!
- **Upside**: good memory efficiency



Beam Search

General approach (beyond A* too)

- Priority queue with fixed size *k*; beyond *k* nodes, **discard**!
- **Upside**: good memory efficiency
- **Downside**: not complete or optimal

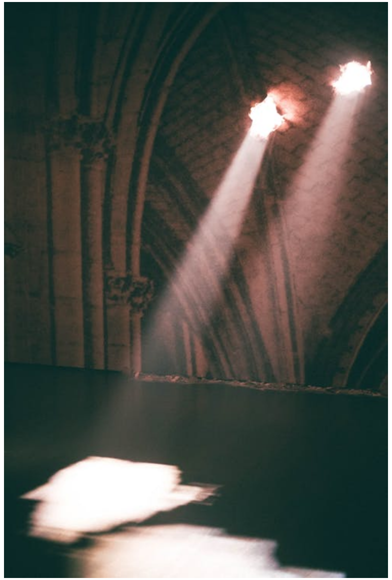


Beam Search

General approach (beyond A* too)

- Priority queue with fixed size *k*; beyond *k* nodes, **discard**!
- **Upside**: good memory efficiency
- **Downside**: not complete or optimal

Variation:



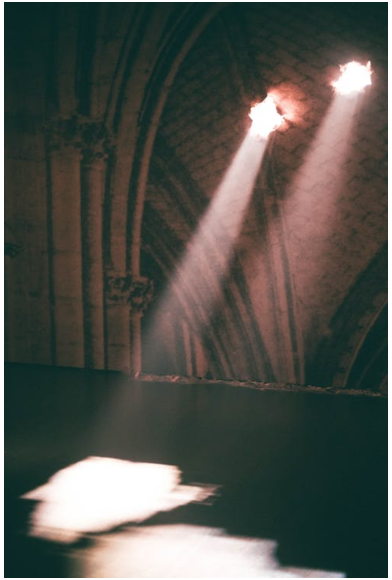
Beam Search

General approach (beyond A* too)

- Priority queue with fixed size *k*; beyond *k* nodes, **discard**!
- **Upside**: good memory efficiency
- **Downside**: not complete or optimal

Variation:

- Priority queue with nodes that **are at most ε worse** than best node.



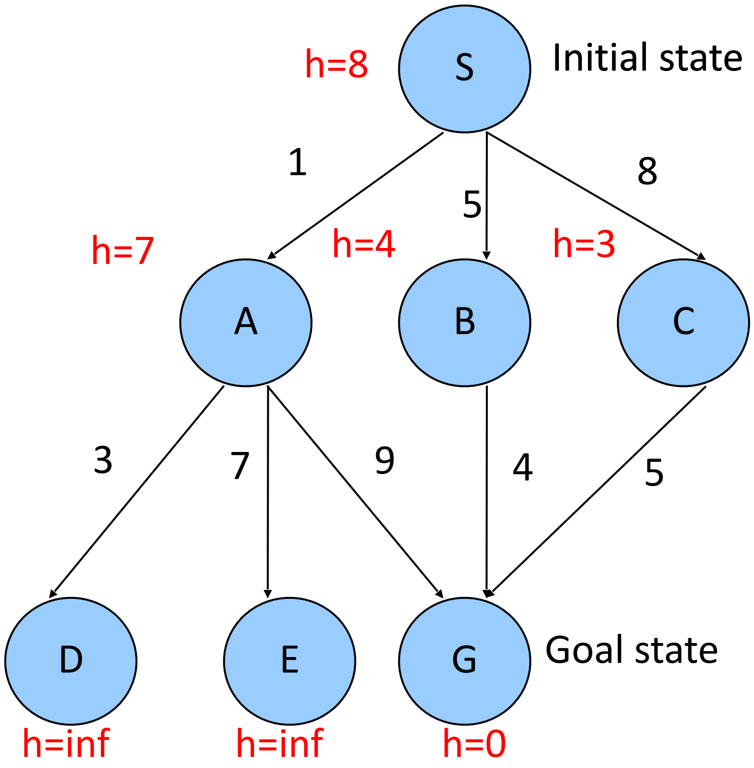
Recap and Examples

Recap and Examples

Example for A*:

Recap and Examples

Example for A*:



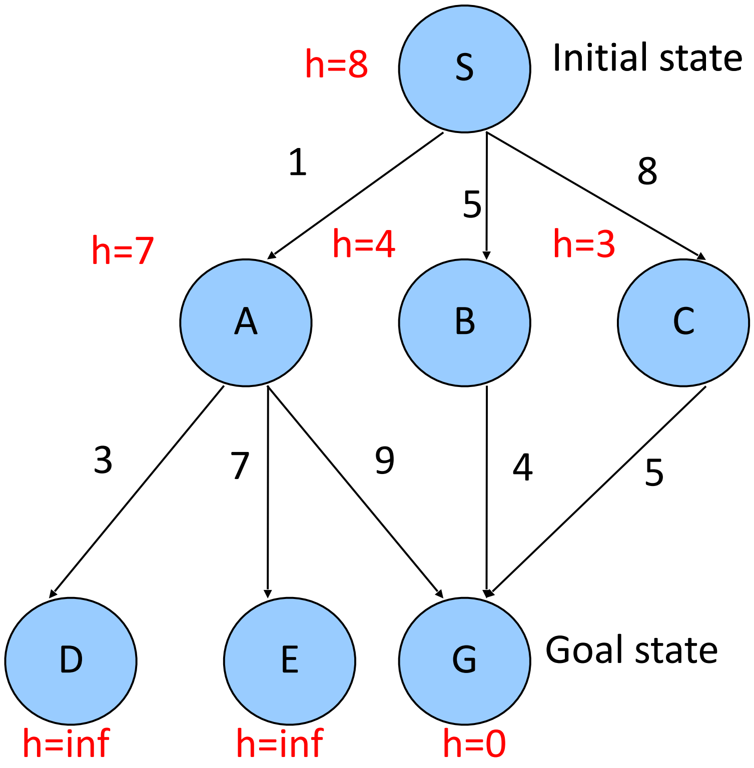
Recap and Examples

Recap and Examples

Example for A*:

Recap and Examples

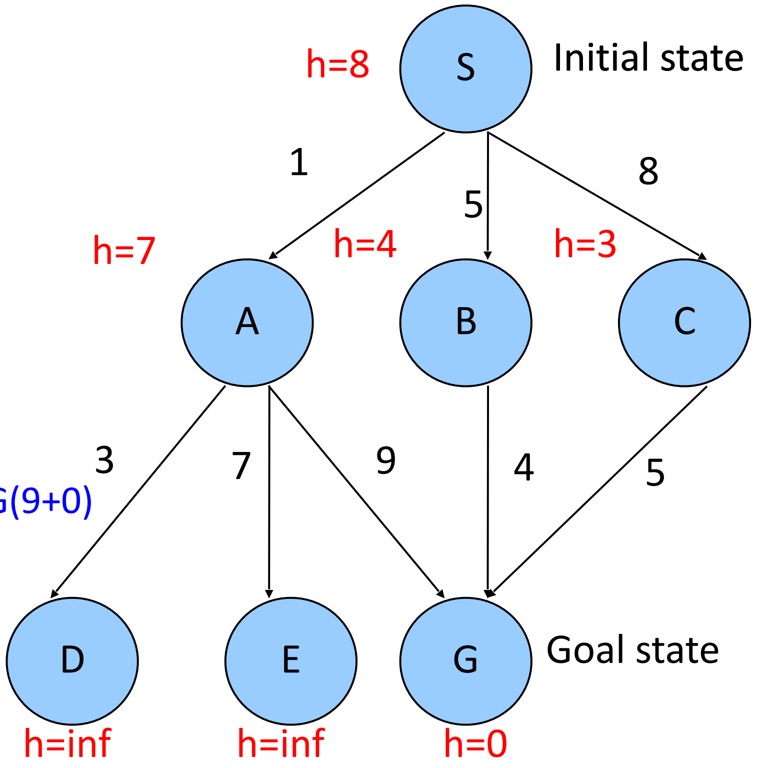
Example for A*:



Recap and Examples

Example for A*:

OPEN	CLOSED
S(0+8)	-
A(1+7) B(5+4) C(8+3)	S(0+8)
B(5+4) C(8+3) D(4+inf) E(8+inf) G(10+0)	S(0+8) A(1+7)
C(8+3) D(4+inf) E(8+inf) G(9+0)	S(0+8) A(1+7) B(5+4)
C(8+3) D(4+inf) E(8+inf)	S(0+8) A(1+7) B(5+4) G(9+0)

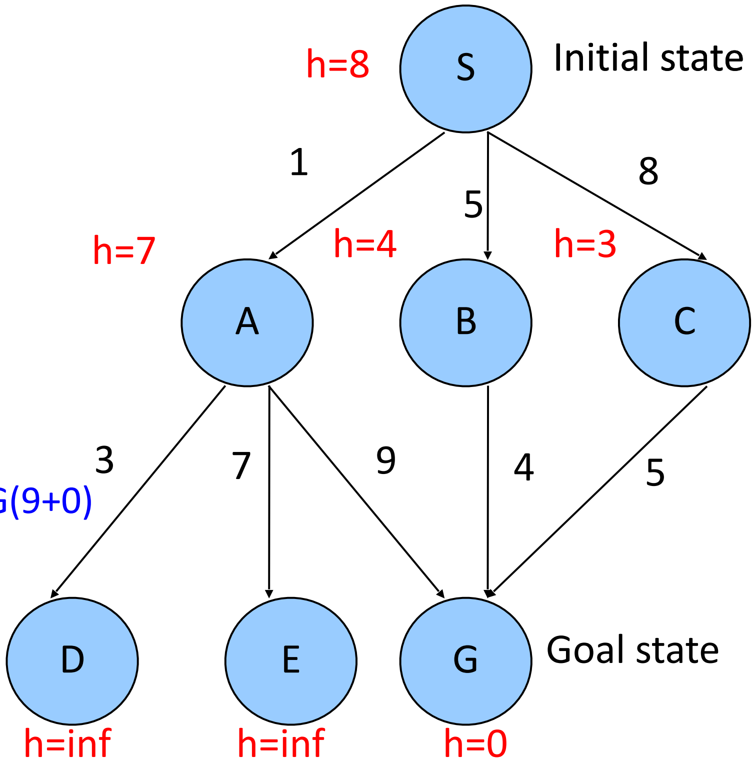


Recap and Examples

Example for A*:

OPEN	CLOSED
S(0+8)	-
A(1+7) B(5+4) C(8+3)	S(0+8)
B(5+4) C(8+3) D(4+inf) E(8+inf) G(10+0)	S(0+8) A(1+7)
C(8+3) D(4+inf) E(8+inf) G(9+0)	S(0+8) A(1+7) B(5+4)
C(8+3) D(4+inf) E(8+inf)	S(0+8) A(1+7) B(5+4) G(9+0)

G → B → S



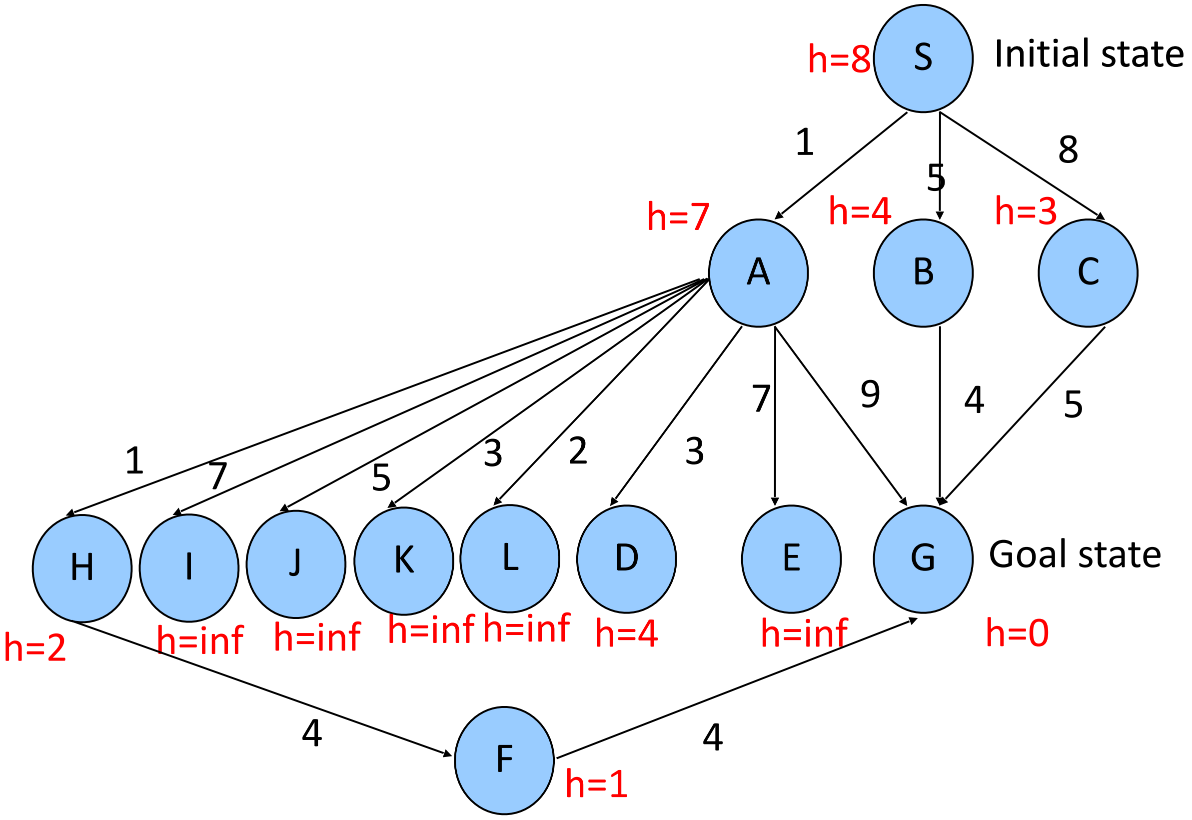
Recap and Examples

Recap and Examples

Example for IDA*:

Recap and Examples

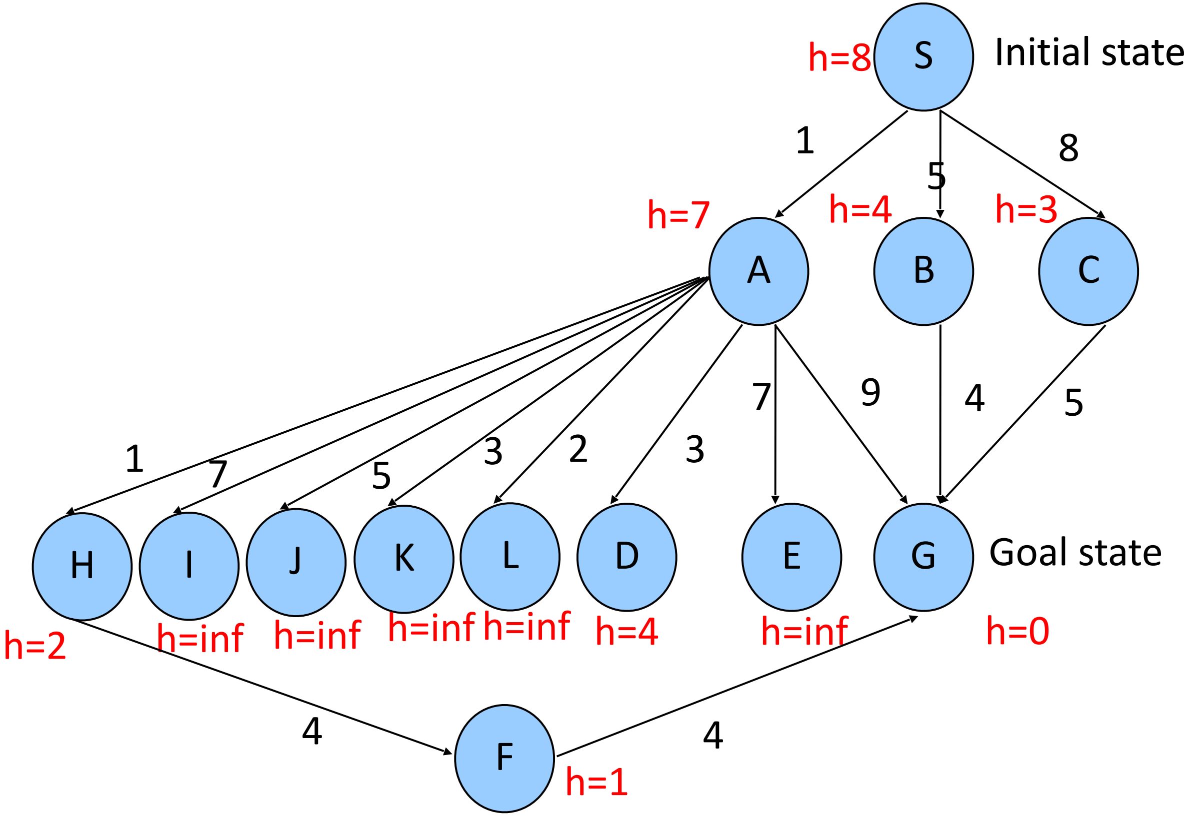
Example for IDA*:



Recap and Examples

Example for IDA*:

Threshold = 8

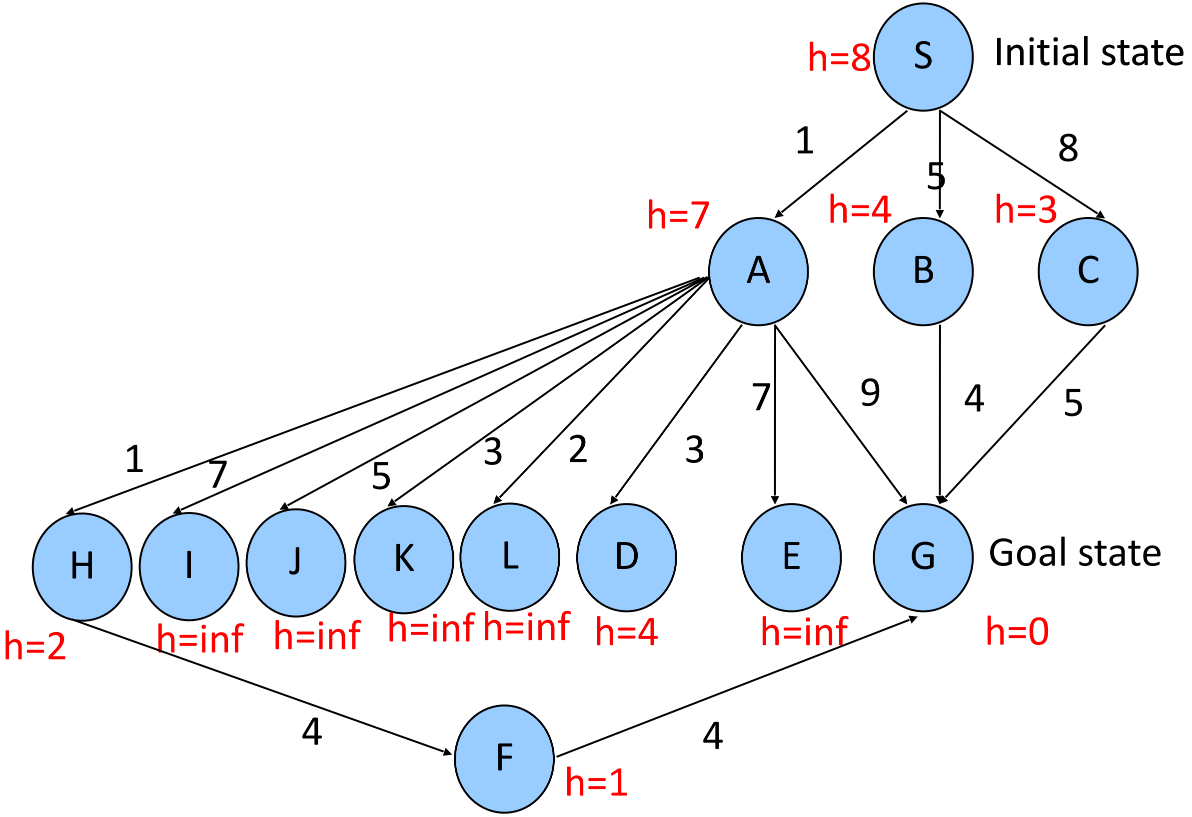


Recap and Examples

Example for IDA*:

Threshold = 8

PATH PREFIX	OPEN
-	S(0+8)
S	A(1+7)
S A	H(2+2) D(4+4)
S A H	D(4+4) F(6+1)
S A H F	D(4+4)
S A D	



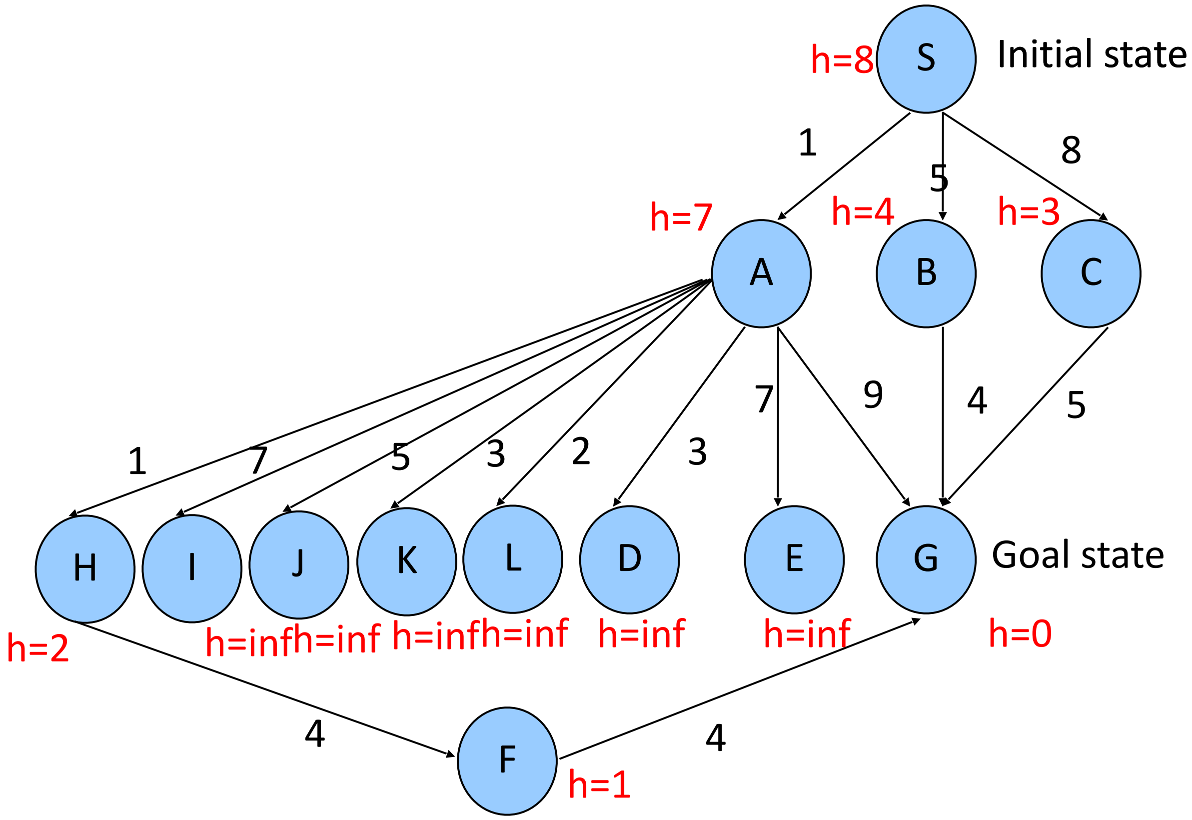
Recap and Examples

Recap and Examples

Example for IDA*:

Recap and Examples

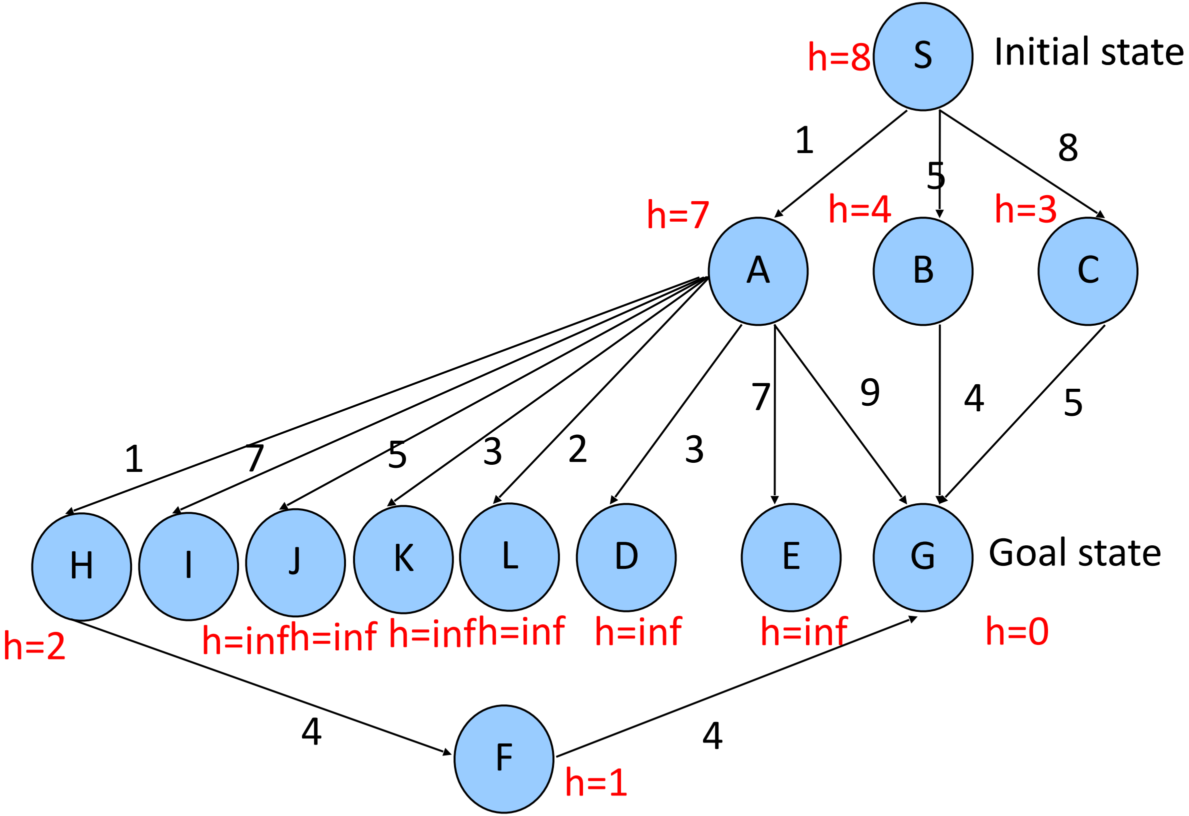
Example for IDA*:



Recap and Examples

Example for IDA*:

Threshold = 9

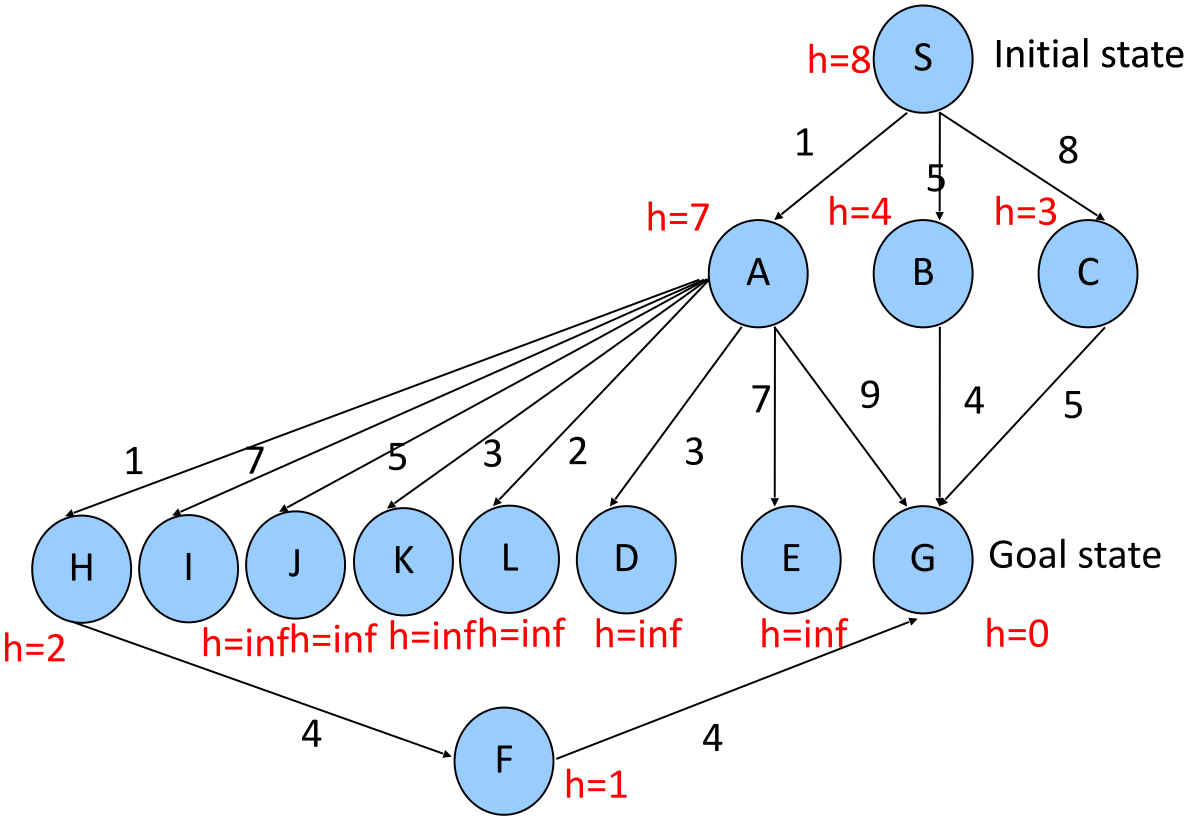


Recap and Examples

Example for IDA*:

Threshold = 9

PREFIX	OPEN
-	S(0+8)
S	A(1+7) B(5+4)
S A	B(5+4) H(2+2) D(4+4)
S A H	B(5+4) D(4+4) F(6+1)
S A H F	B(5+4) D(4+4)
S A D	B(5+4)
S B	G(9+0)
S B G	



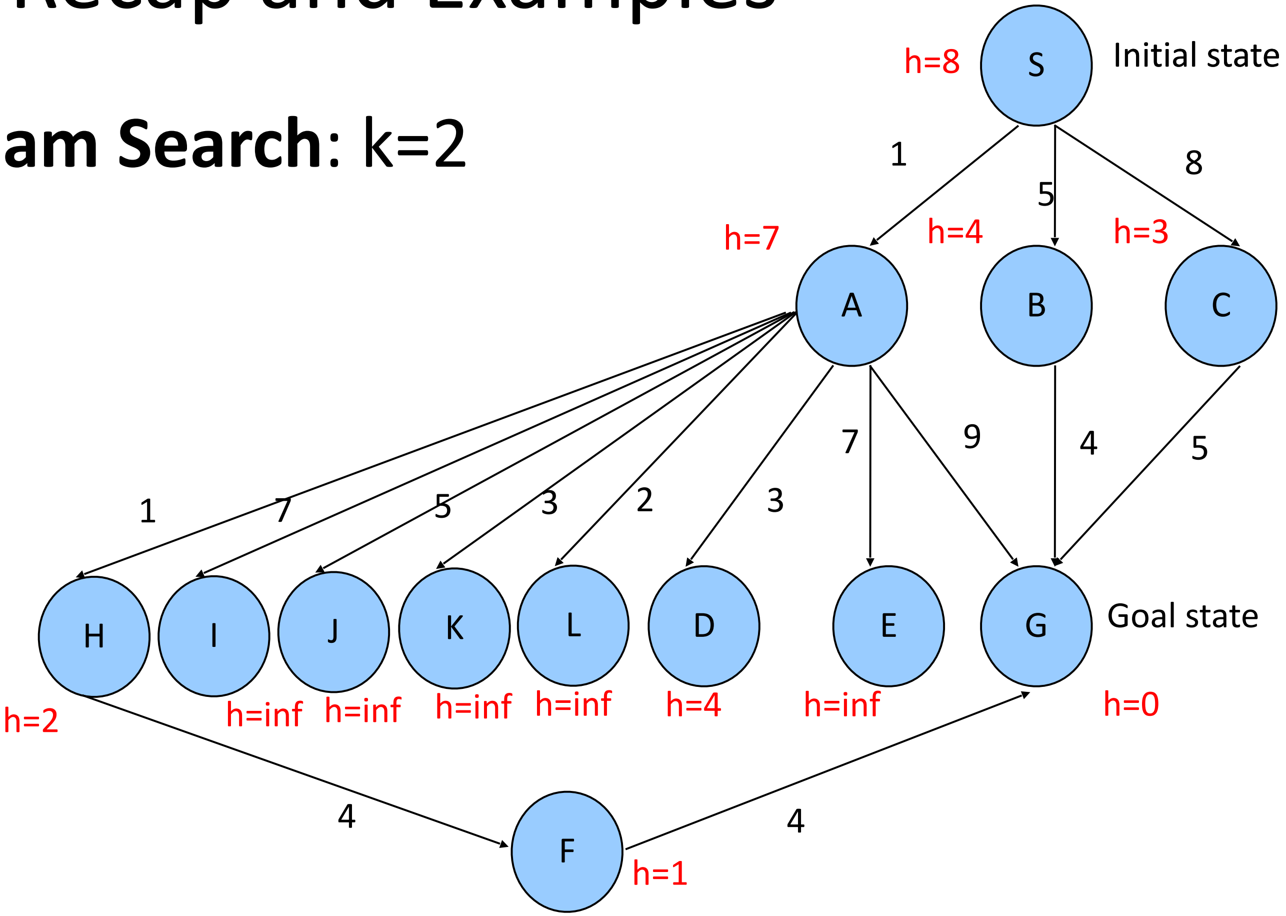
Recap and Examples

Recap and Examples

Example for **Beam Search**: k=2

Recap and Examples

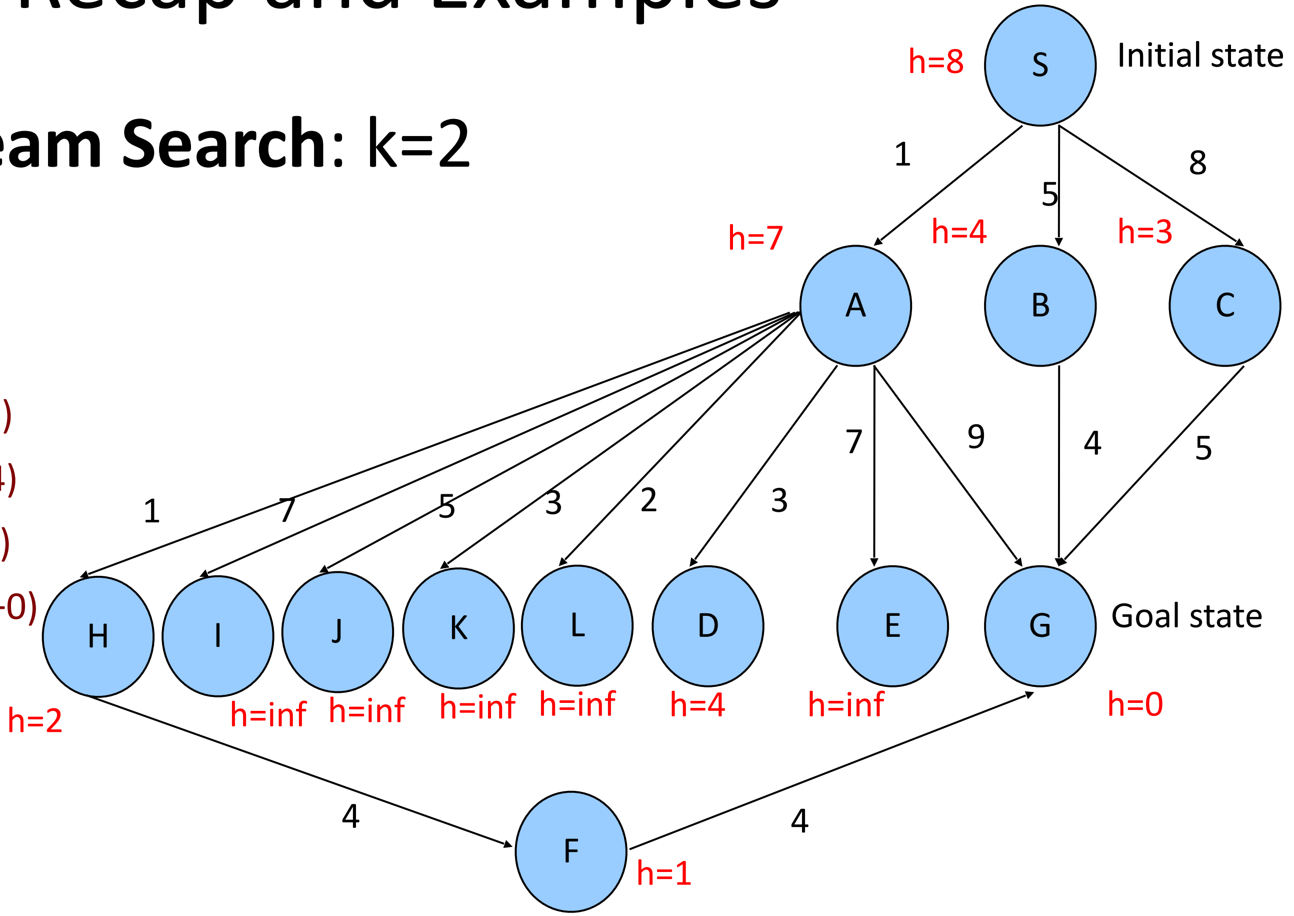
Example for **Beam Search**: k=2



Recap and Examples

Example for **Beam Search**: k=2

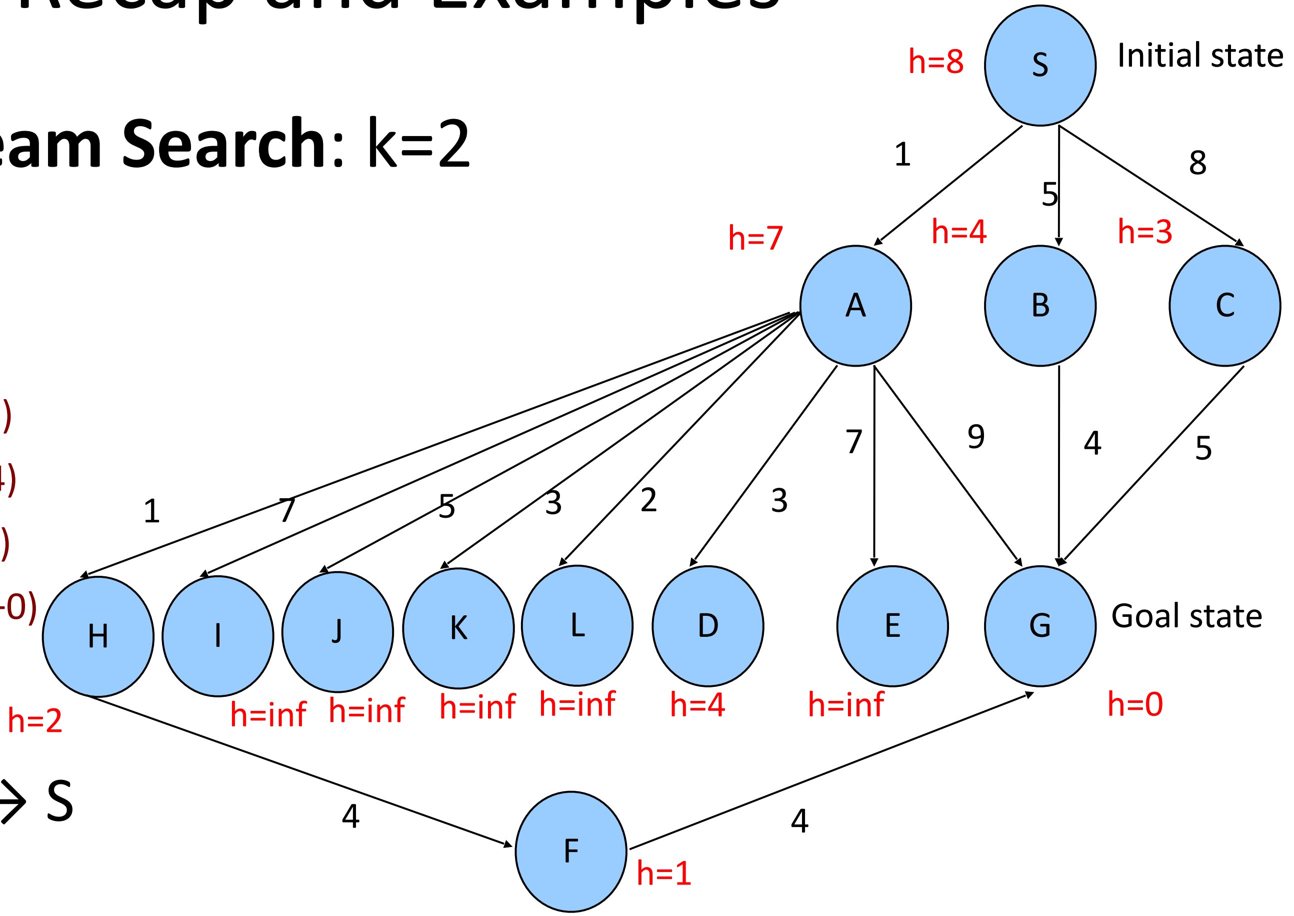
CURRENT	OPEN
-	S(0+8)
S	A(1+7) B(5+4)
A	H(2+2) D(4+4)
H	D(4+4) F(6+1)
F	D(4+4) G(10+0)
D	G(10+0)
G	



Recap and Examples

Example for **Beam Search**: k=2

CURRENT	OPEN
-	S(0+8)
S	A(1+7) B(5+4)
A	H(2+2) D(4+4)
H	D(4+4) F(6+1)
F	D(4+4) G(10+0)
D	G(10+0)
G	



G → F → H → A → S

Not optimal!

Summary

- Informed search: introduce heuristics
  - Not all approaches work: best-first greedy is bad
- A* algorithm
  - Properties of A*, idea of admissible heuristics
- Beyond A*
  - IDA*, beam search. Ways to deal with space requirements.

Acknowledgements: Adapted from materials by Jerry Zhu (University of Wisconsin).