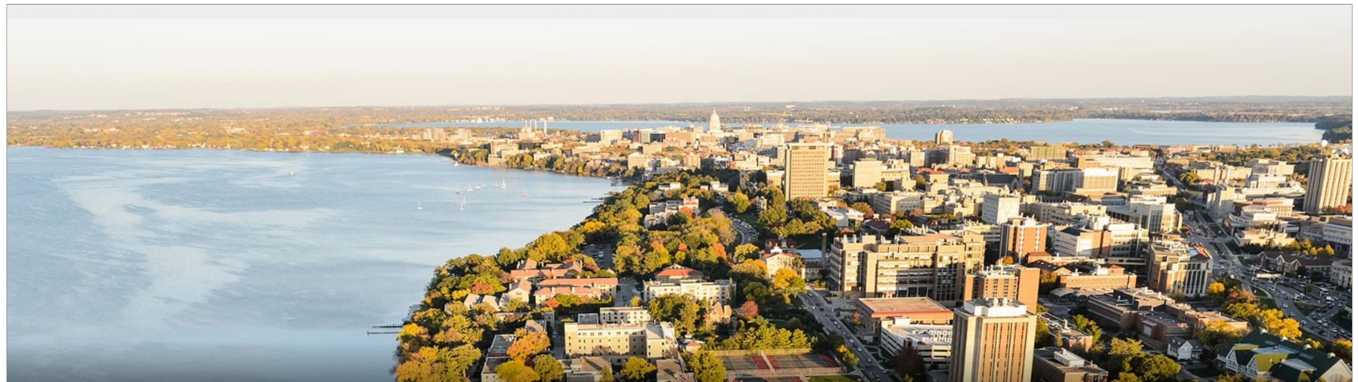


CS 540 Introduction to Artificial Intelligence
Informed Search

Yingyu Liang
University of Wisconsin-Madison

Nov 16, 2021

Based on slides by Fred Sala

Outline

- Uninformed continued
- A* Search
  - Heuristic properties, stopping rules, analysis

General State-Space Search Algorithm

```
function general-search(problem, QUEUEING-FUNCTION)
 ;; problem describes the start state, operators, goal test, and
 ;;   operator costs
 ;; queueing-function is a comparator function that ranks two states
 ;; general-search returns either a goal node or "failure"

 nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
 loop
   if EMPTY(nodes) then return "failure"
   node = REMOVE-FRONT(nodes)
   if problem.GOAL-TEST(node.STATE) succeeds then return node
   nodes = QUEUEING-FUNCTION(nodes, EXPAND(node,
                              problem.OPERATORS))
   ;; succ(s)=EXPAND(s, OPERATORS)
   ;; Note: The goal test is NOT done when nodes are generated
   ;; Note: This algorithm does not detect loops
 end
```

The general framework for search algorithms.

Input: problem description and also an implementation of the fringe

First put the initial state into the fringe then go to loops

In each iteration:
check if the fringe is empty, if so output failure.
Otherwise get a node from the fringe, and test if it’s the goal state.
If yes, then claim success.
If no, get the successors and put them into the fringe.

Recall the bad space complexity of BFS

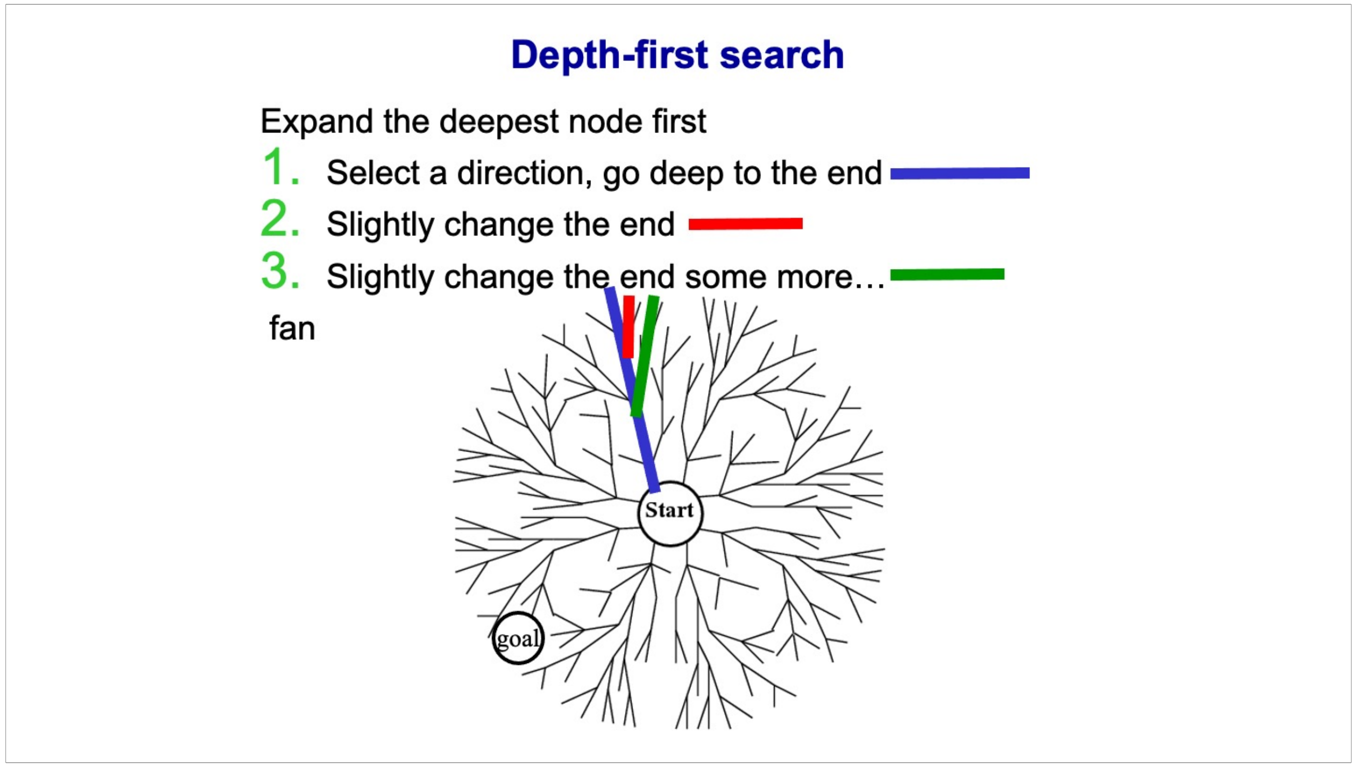
Four measures of search algorithms:

- Completeness (not finding all goals): find a goal.
- Optimality: yes if edges cost 1 (more gene ally positive non-decreasing with depth), no otherwise.
- Time comple): goal is the last node at radius d.
  - Have to nodes at radius d.
  - $b + b^2 + \ldots + b^d \sim O(b^d)$
- Space complexity (bad, see the Figure)
  - Back points for all generated nodes $O(b^d)$
  - The queue (smaller, but still $O(b^d)$)

Solution: Uniform-cost search

Solution: Depth-first search

Two drawbacks of BFS.
Not optimal for non-uniform cost: addressed by UCS
Bad space complexity: addressed by DFS



DFS expands the deepest node first (compared to: DFS expands the shallowest node first)

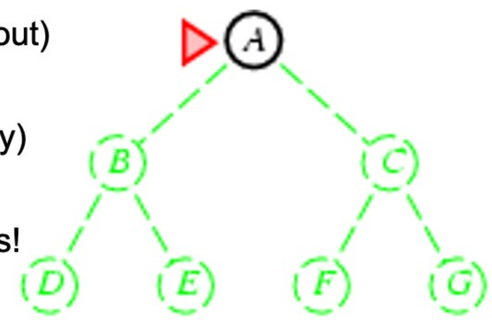
The execution of the algorithm intuitively is like going along a direction: going deeper and deeper (because of expanding the deepest node first) until the end;
if still doesn't get the goal state, step back a bit and slightly change the end (expanding the current deepest node which is one step back along the path);
if still doesn't get the goal state, step back a bit more and slightly change the end
…

It's like a fan swinging across the tree.

Depth-first search (DFS)

Use a stack (First-in Last-out)

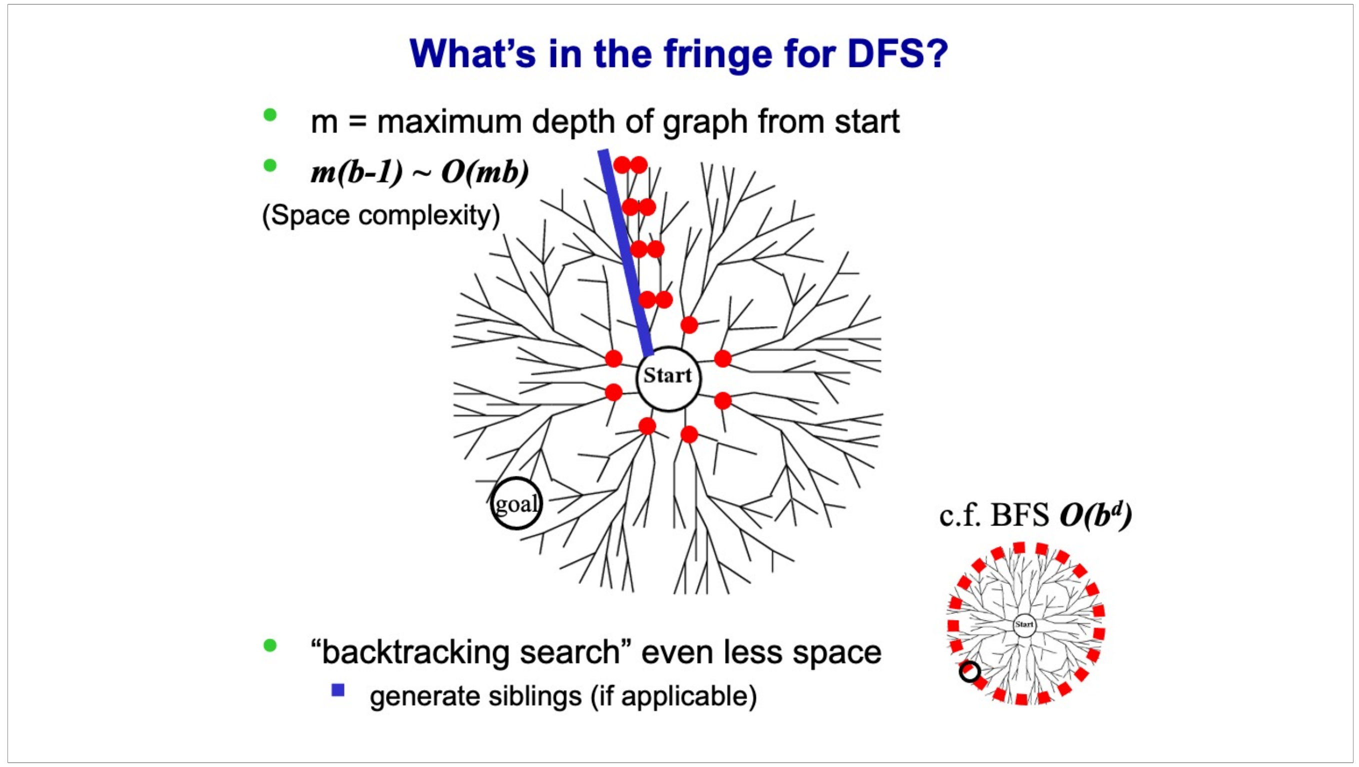
1. push(Initial states)
2. While (stack not empty)
3. s = pop()
4. if (s==goal) success!
5. T = succs(s)
6. push(T)
7. endWhile



stack (fringe)
[] ⇔

Pseudocode similar to BFS (except using stack instead of queue).

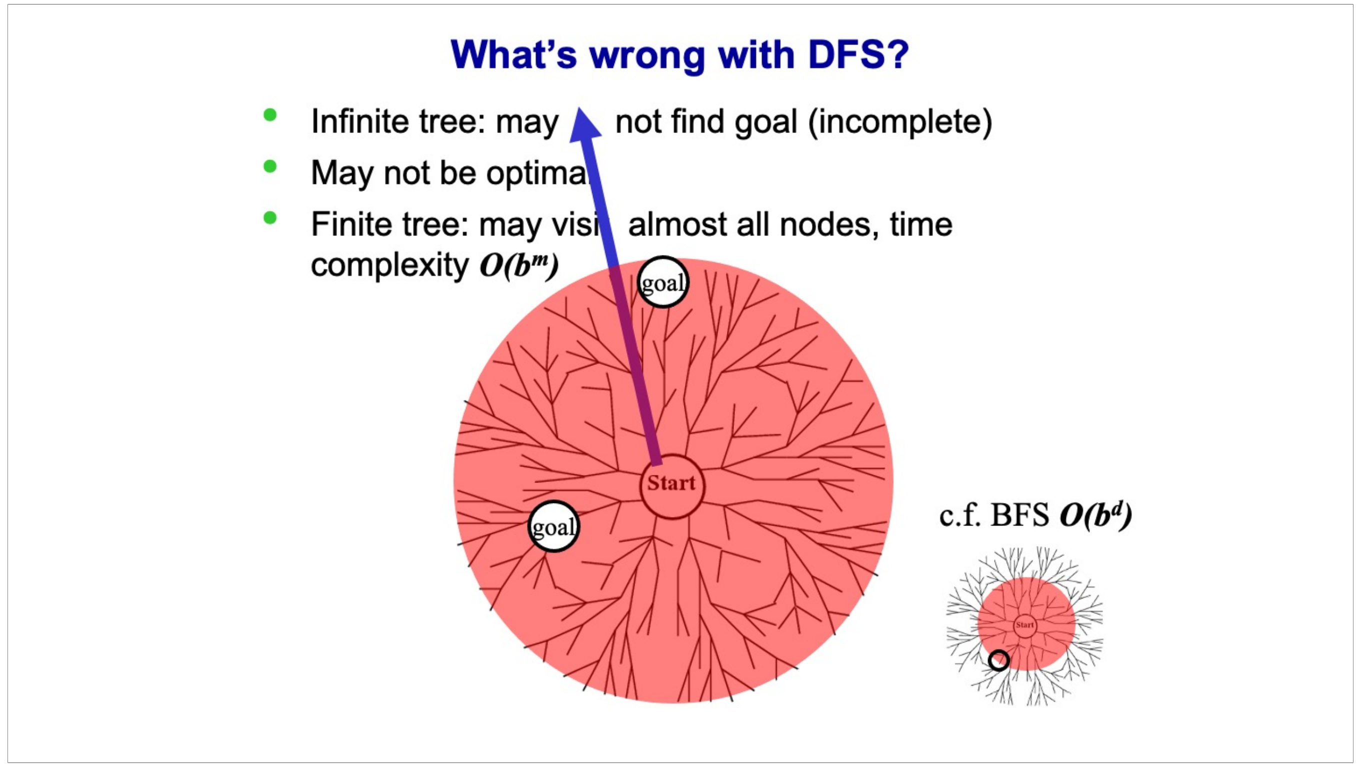
Example:
First put A into the stack.
Iterations, node popped, stack at the end of the iteration (left means going to be popped). Tie breaking: left node has higher priority
1: A, [B C]
2: B, [D E C]
3: D, [E C]
4: E, [C]
5: C, [F G]
6: F, [G]
7: G



Performance of DFS: good in space complexity

The fringe contains the children of the nodes along the path, which is in the order of m*b, where b is the branching factor. A significant win over BFS.

Can be further reduced by backtracking trick (not required in this course)



However, DFS has bad performance in the other 3 aspects.

Incomplete: it can go to the wrong direction which has no goal but is infinite, then it gets in an infinite loop
Not optimal: can shoot in the direction of a suboptimal goal and thus find that goal first
Time complexity: can be infinite on an infinite tree; even on finite tree, it can visit all nodes before reaching the goal which takes time of order b^m

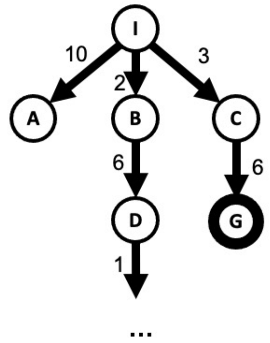
Performance of search algorithms on trees

b: branching factor (assume finite)   d: goal depth   m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$

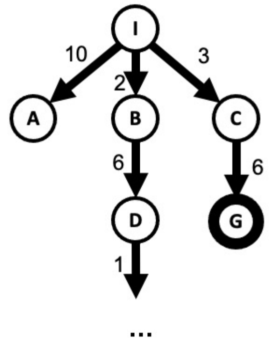
1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C^* is the best goal path cost.

Qu2-1: You are running DFS in the state space graph below. DFS expands nodes left to right. G is the goal state. The state space graph is infinite (the path after D does not terminate). What is the behavior of DFS?



1. Get stuck in an infinite loop
2. Return A
3. Return G
4. Return “failure”

Qu2-1: You are running DFS in the state space graph below. DFS expands nodes left to right. G is the goal state. The state space graph is infinite (the path after D does not terminate). What is the behavior of DFS?



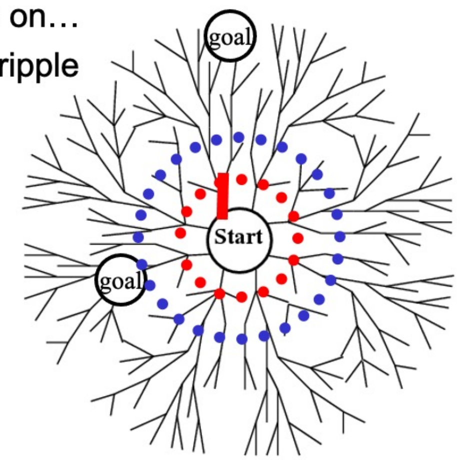
1. Get stuck in an infinite loop
2. Return A
3. Return G
4. Return "failure"

First put I into the stack.
Iteration 1: pop I, put ABC into the stack
Iteration 2: pop A, no successor
Iteration 3: pop B, add D to the stack
Iteration 4: pop D, add the next node along the path to the stack
And it goes deeper and deeper infinitely along the middle path.

How about this?

1. DFS, but stop if path length > 1.
2. If goal not found, repeat DFS, stop if path length > 2.
3. And so on…

fan within ripple



DFS: good space complexity compared to BFS but bad in the other aspects

Can combine the two to get the best of both: run in stages; across stages like BFS; within each stage run DFS.

In stage t: do DFS only on nodes at most t steps away from the initial state. That is equivalent to considering a truncated tree with nodes at most t steps from the initials state, and then run DFS on that truncated tree.

Each stage t is like a ripple of radius t (like that in BFS); within the stage, run DFS which acts like a fan within the ripple.

Iterative deepening

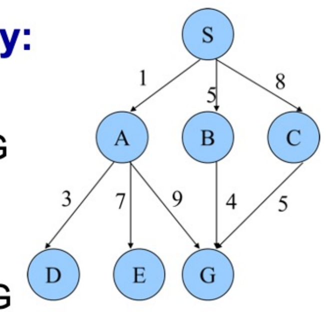
- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
  - Time complexity like BFS
- Preferred uninformed search method

Performance: because each stage we have a finite truncated tree, we avoid the bad aspects of DFS.

Complete: if there is a goal d steps away from the initial state, then within d stages, we must be able to find it.
Optimal when edge costs are uniform: the first time the truncated tree includes a goal state, it must be the optimal goal state.
Small space complexity: the space needed is like that of DFS
Time complexity: stage t may visit all nodes t steps away from the initial state, so has a runtime of order b^t ; we must succeed within d stages if there is a goal d steps away from the initial state. So the total run time is in the order of b^1 + b^2 + … + b^d = O(b^d), similar in order to BFS

So this is the preferred method for uninformed search

Nodes expanded by:



- Breadth-First Search: S A B C D E G
  Solution found: S A G
- Uniform-Cost Search: S A D B C E G
  Solution found: S B G (This is the only uninformed search that worries about costs.)
- Depth-First Search: S A D E G
  Solution found: S A G
- Iterative-Deepening Search: S A B C S A D E G
  Solution found: S A G

IDS on the example (tie breaking: expand left node first)

Stage 1:
First put S into the fringe
Iteration: node expanded, fringe at the end of the iteration
1: S, [A B C]
2: A, [B C] Note that we only consider nodes within 1 step from S, pretending A has no successors.
3: B, [C]
4: C, []

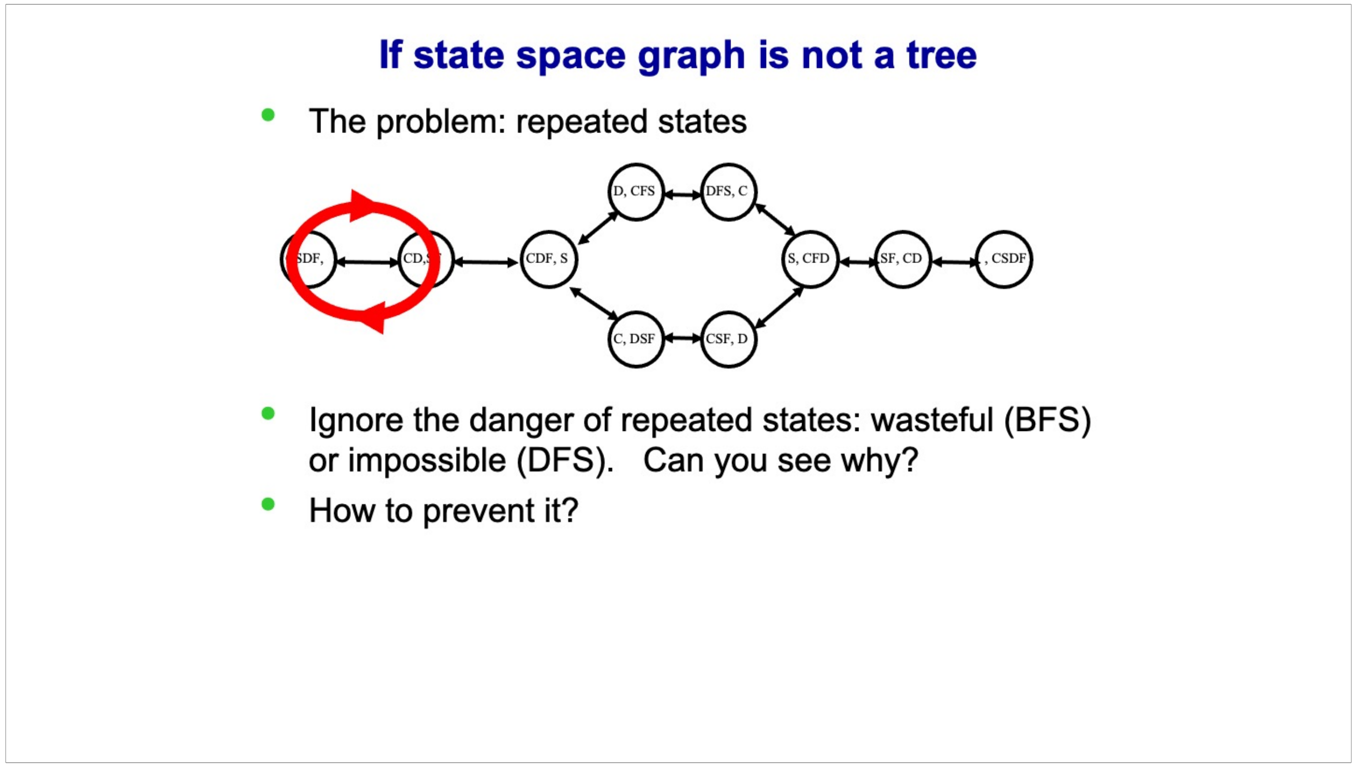
Stage 2:
First put S into the fringe
1: S, [A B C]
2: A, [D E G B C] Note that we now consider nodes within 2 steps from S, ie, including all nodes.
3: D, [E G B C]
4: E, [G B C]
5: G, [B C]. Claim success and return the path SAG

Performance of search algorithms on trees

b: branching factor (assume finite)   d: goal depth   m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$
Iterative deepening	Y	Y, if [1]	$O(b^d)$	$O(bd)$

1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C^* is the best goal path cost.



We have been talking about search on trees which have no loops. If there is a loop then we may revisit an already expanded node.

Consider DFS on the given graph (assuming tie-breaking by expanding left nodes first).

First put (CSDF,) into the fringe.
Iteration 1: pop (CSDF,), put (CD, SF) into the fringe
Iteration 2: pop (CS, SF), put (CSDF,) and (CDF, S) into the fringe
Iteration 3: pop (CSDF,) ….
Get infinite loop

If state space graph is not a tree

- We have to remember already-expanded states (CLOSED).
- When we take out a state from the fringe (OPEN), check whether it is in CLOSED (already expanded).
  - If yes, throw it away.
  - If no, expand it (add successors to OPEN), and move it to CLOSED.

The idea is simple: keep a CLOSED set which memorizes all nodes already expanded; check when get a stage from the fringe. (Can also check at the time point when we generate successors)

Applied to the previous example.

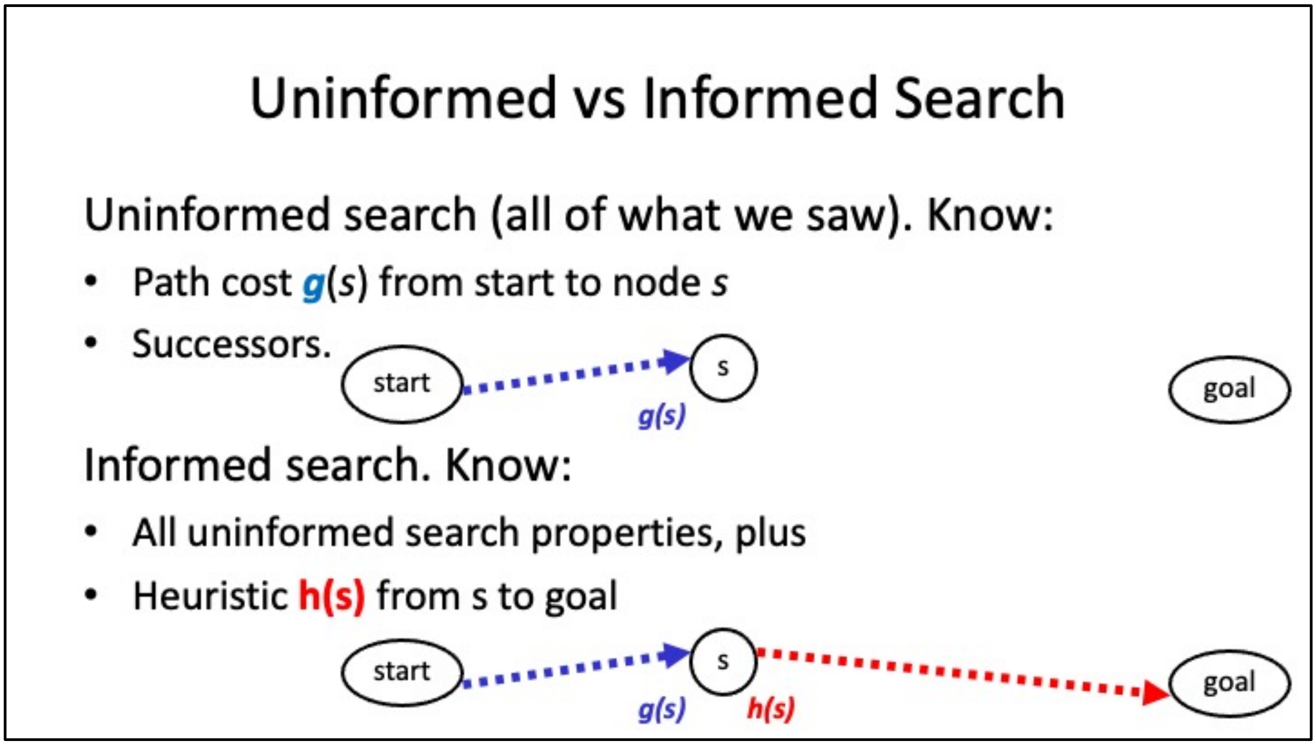
First put (CSDF,) into the fringe. CLOSED set is empty.
Iteration 1: pop (CSDF,), put (CD, SF) into the fringe. CLOSED=[CSDF,)]
Iteration 2: pop (CS, SF), put (CSDF,) and (CDF, S) into the fringe. CLOSED=[CSDF,), (CS, SF)]
Iteration 3: pop (CSDF,). Note that it’s in CLOSED, so throw away.
Iteration 4: pop (CDF, S), ….
Can avoid the infinite loop

What you should know

- Problem solving as search: state, successors, goal test
- Uninformed search
  - Breadth-first search
    - Uniform-cost search
  - Depth-first search
  - Iterative deepening

- Can you unify them using the same algorithm, with different priority functions?
- Performance measures
  - Completeness, optimality, time complexity, space complexity

Summary:
The search framework
Several uninformed search methods
Unified pseudocode for them; key difference: how to pick a node from the fringe to expand first
Performance measure; iterative deepening is the preferred method due to its good performance.

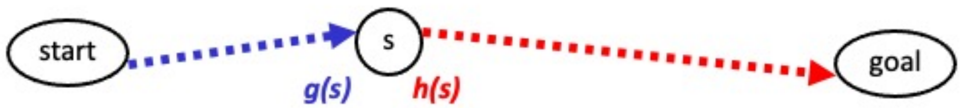


Key difference: knows an additional function h(s), which can be regarded as an estimation of the cost from a state to the goal (or one of the goal states).

Informed Search

Informed search. Know:

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

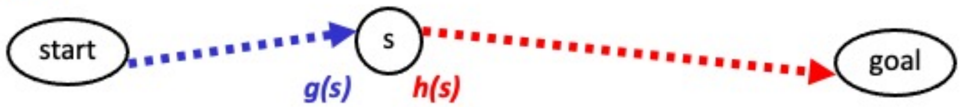


- Use information to **speed up search.**

Using the Heuristic

Back to uniform-cost search

- We had the priority queue
- Expand the node 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

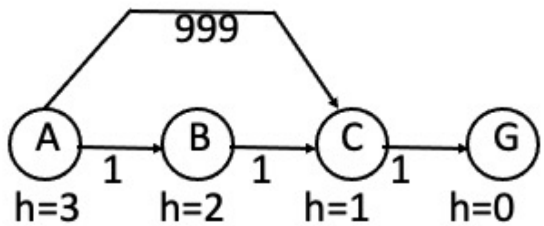
Recall in UCS: we pick the node with the lowest cost g(s). This is using the first half cost. Now we have h(s) giving an estimation of the second-half-cost, we can think of ways to use it.

There are several approaches: which works and under what conditions?

Attempt 1: Best-First Greedy

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

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



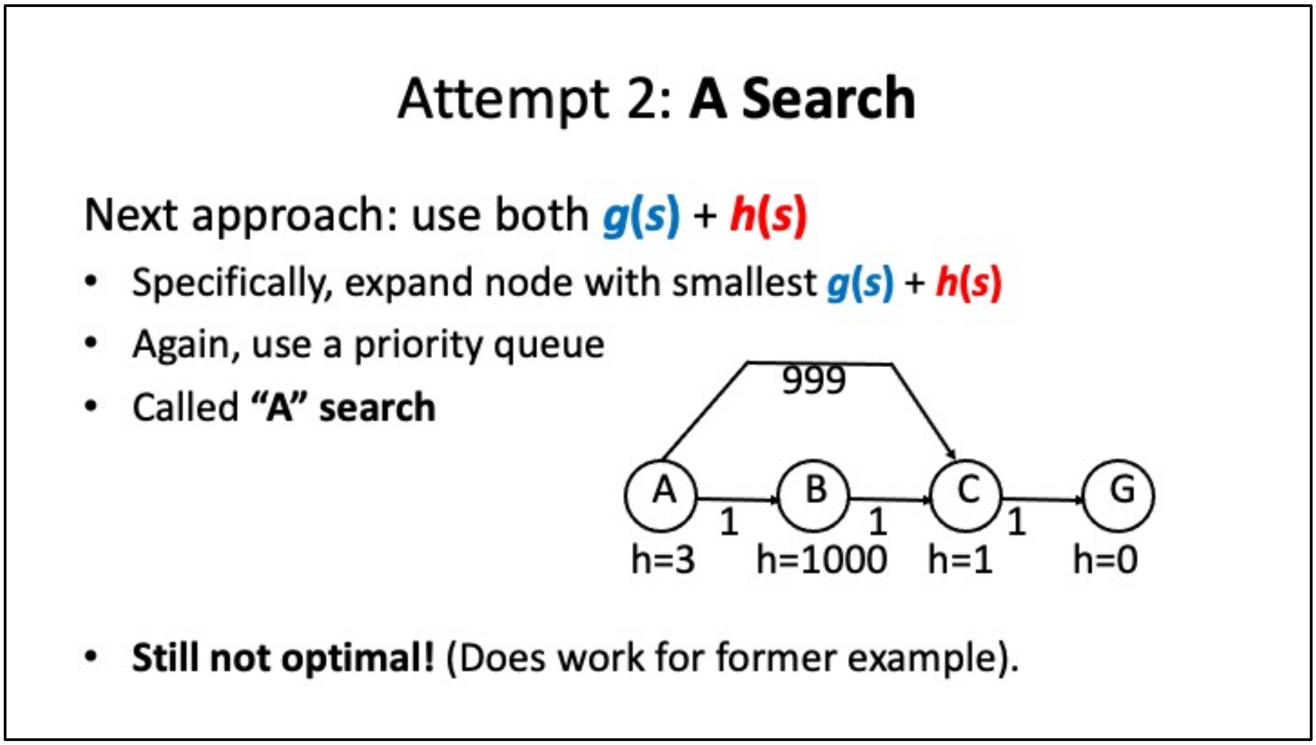
- Not optimal! **Get** A → C → G. **Want**: A →B → C → G

Attempt: use only h as the priority.

Can lead to trouble: because g is not considered at all, then may pick a path that has a large g, which is suboptimal.

Example:
Iteration: node expanded, fringe at the end
1: (A, 3), [(B,2), (C,1)]
2: (C,1), [(B,2), (G,0)]
3: (G,0) claim success and return the path ACG.
ACG has cost 1000, much larger than the optimal cost by ABCG.

This is because the edge AC has a huge cost, which is not taken into account.



Natural fix: use g+h. Called A search.

Can fix the issue in the example on the previous slide. But may still be optimal in other cases, when h is very inaccurate.

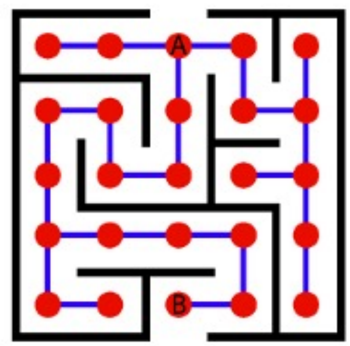
Example on this slide:
Iteration: node expanded, fringe at the end
1: (A, 3), [(B,1001), (C,1000)]
2: (C,1000), [(B,1001), (G,1000)]
3: (G,1000) claim success and return the path ACG.
ACG has cost 1000, much larger than the optimal cost by ABCG.

This is because the h value of B is huge, very inaccurate estimation of the true cost (the true cost from B to G is only 2).

Attempt 3: **A* Search**

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

- Demand that $0 \le h(s) \le h^*(s)$, the actual cost
- If heuristic has this property, "admissible"
  - Optimistic! Never over-estimates
- Still need $h(s) \ge 0$
  - Negative heuristics can lead to strange behavior
- This is **A* search**

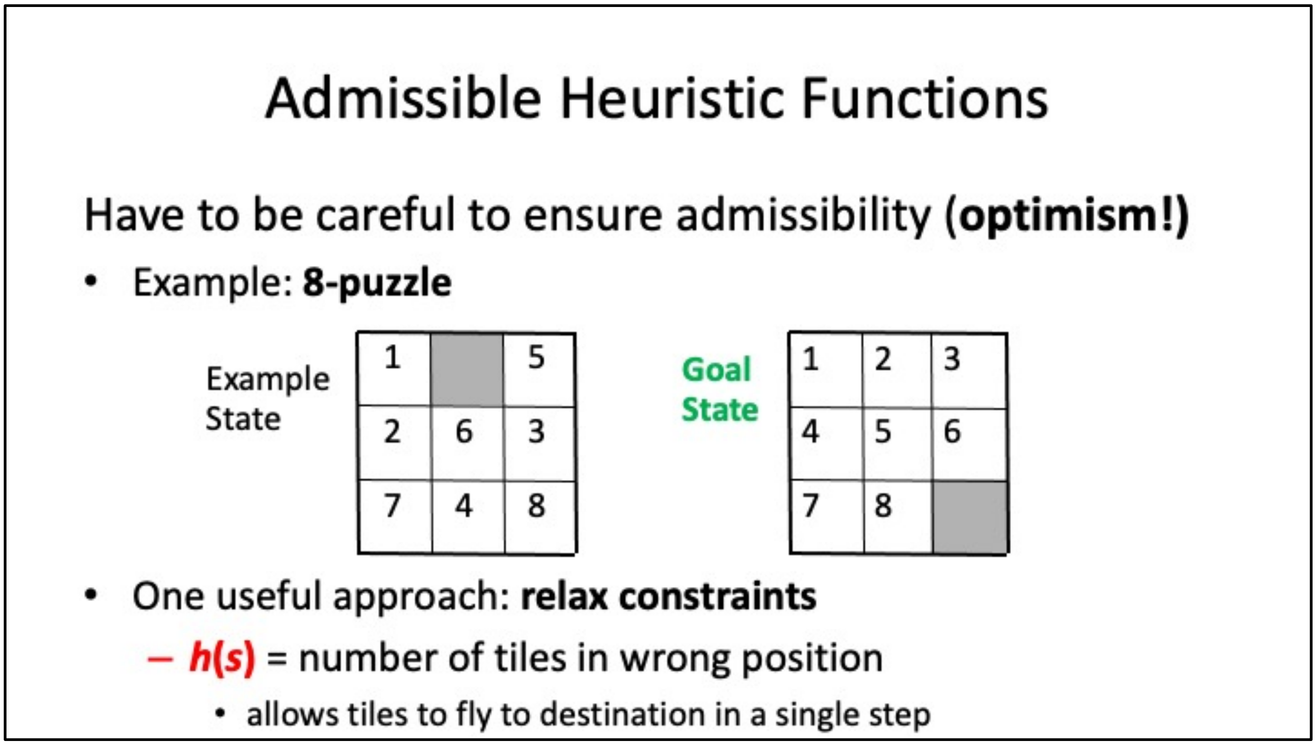


V. Batoćanin

Then we add an requirement on the h: it should be a rough estimation of the true value h*.

Formally, it's within 0 and the true value on any node s: admissible.

A* search = A search + admissible heuristic function



How can we design an admissible heuristic without knowing the actual cost h*?

Useful principle: relax the constraints on the successor function in the search problem; compute the cost in the relaxed problem. Relaxing the constraints is like adding edges to the search graph, which can only introduce more paths and thus will not increase the cost.

Example:
Original 8-puzzle only allow to move the tiles around the blank space.
Relax: allow the corresponding number to fly to the destination. (In the example state, allow to fly 2 to the blank space)
If we keep flying in this way, the steps to reach the goal state is #tiles in wrong position. Let that be the heuristic. It’s now clear that it’s admissible: nonnegative; and at most the true cost since in the original search problem each wrong-position tile need to be moved at least once to get to the goal state.

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

- 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) $h(s) = \max(2, h^*(s))$
(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) $h(s) = \mathrm{sqrt}(h^*(s))$ 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

Dominance: h_2 dominates h_1 if for all states s,

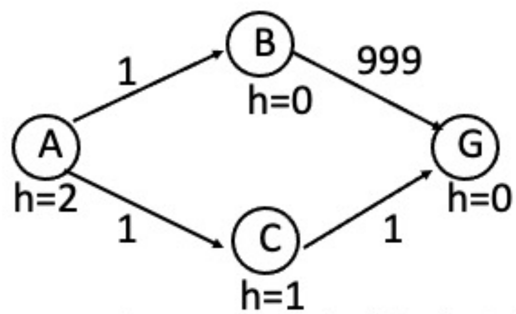
$$h_1(s) \leq h_2(s) \leq h^*(s)$$

- **Idea**: we want to be as close to h^* as possible
  - But not over!
- **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**?

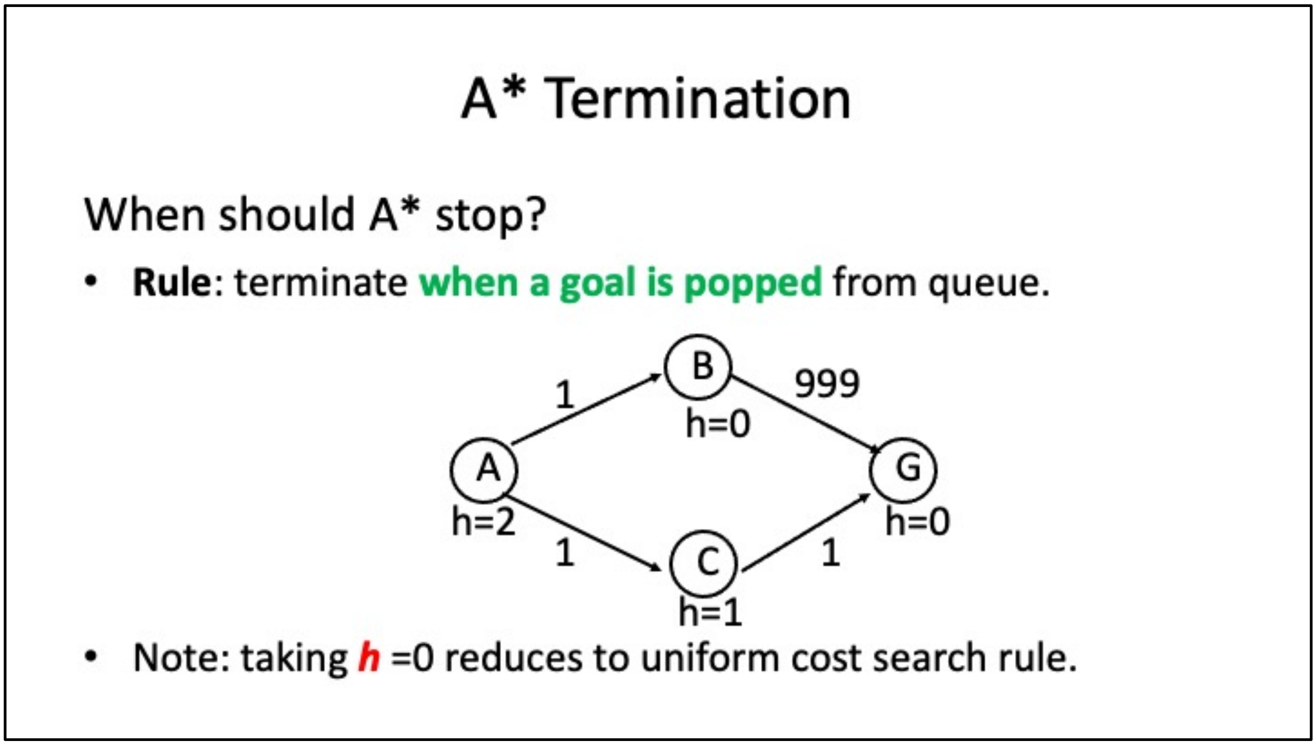
- One idea: as soon as we reach goal state?



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

If we stop and return as soon as we generate a goal state, can return a suboptimal path.

Example:
Iteration: node expanded, fringe at the end
1: (A, 2), [(B,1), (C,2)]
2: (B,1), [(C,2), (G,1000)]. Stop and return
The path obtained is ABG which is suboptimal. This is due to that we haven’t considered the last step (BG has a huge cost 999).



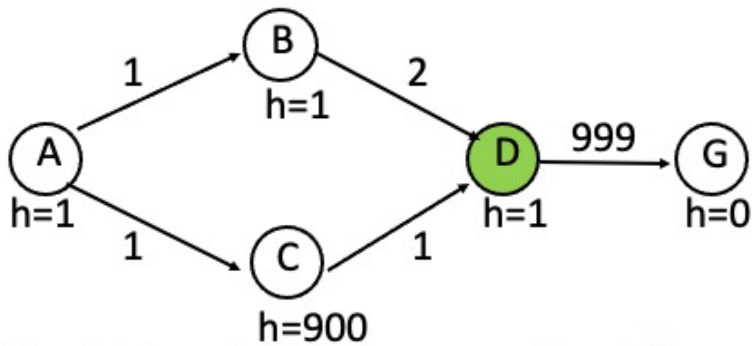
If we return only when we pop the goal from the fringe, then can solve the issue. (Also this is consistent with what we did in uninformed search.)

Example:
Iteration: node expanded, fringe at the end
1: (A, 2), [(B,1), (C,2)]
2: (B,1), [(C,2), (G,1000)]
3: (C,2), [(G, 2)] Here we generate another copy of G (going from A to C to G), which has a smaller cost 2, than the old copy (G,1000). We can keep only the lower cost copy.
4: (G,2).
Return the path ACG.

It also shows that we should compare the new copy with the old copy, when we revisit an already expanded state.

A* Revisiting Expanded States

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



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

In the general case, need to keep a CLOSED set

Example:
Iteration: node expanded, fringe at the end, CLOSED set at the end
1: (A, 1), [(B,2), (C,901)], [(A,1)]
2: (B,2), [(D,4), (C,901)], [(A,1), (B,2)]
3: (D,4), [(C,901), (G, 1002)], [(A,1), (B,2), (D,4)]
4: (C,901), [(G,1002), (D,3)], [(A,1), (B,2), (D,3)] Note that in iteration 4, we find out that D has been expanded but the new copy has a lower cost, so still process it; also put the new copy to the CLOSED set so that later can use it filter new copies with cost >= 3; the old copy in the CLOSED set can be removed or kept and here we remove it

5: (D, 3), [(G,1001)], [(A,1), (B,2), (D,3), (C,901)] Note that in iteration 5, we find out that G has been generated but the new copy has a lower cost.
6, (G,1001). Claim success and return the path ACDG.

A* Full Algorithm

1. Put the start node S on the priority queue, called 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 (trace 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 estimate 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.

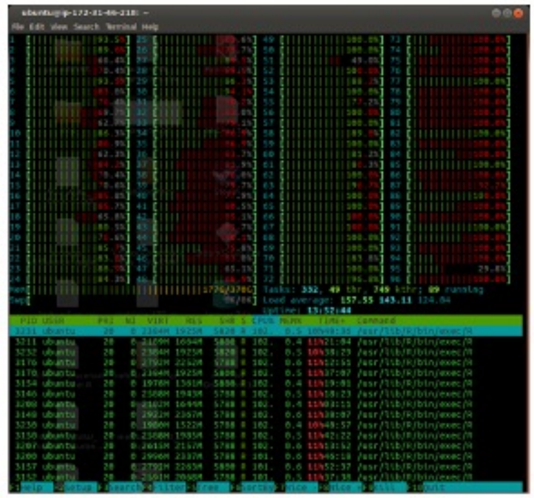
Differences from how we handle repeated states in uninformed search

1. In the uninformed search slide, we don't consider cost. (but we can also consider cost for uninformed search if needed)
2. In the uninformed search slide the check is done when a node is picked from the fringe; here the check is done when a node is generated by the successor function. (but here we can also perform the check when a node is picked from the fringe)

A* Analysis

Some properties:

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



It’s guaranteed to terminate. But it can use lots of memory since it may keep all states before termination. Can use some other tricks to alleviate this issue.

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

Summary

- Informed search: introduce heuristics
  - Not all approaches work: best-first greedy is bad
- A* algorithm
  - Properties of A*, idea of admissible heuristics

Acknowledgements: Adapted from materials by Jerry Zhu, Anthony Gitter, and Fred Sala (University of Wisconsin-Madison).