

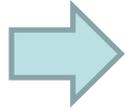
Multiple Path Pruning, Iterative Deepening

CPSC 322 – Search 7

Textbook § 3.7.1-3.7.3

January 26, 2011

Lecture Overview



Some clarifications & multiple path pruning

- Recap and more detail: Iterative Deepening

Clarifications for the A* proof

- Defined two **lemmas** about **prefixes** x of a solution path s
 - (I called the prefix pr , but a 2-letter name is confusing; let's call it x instead)
- Clarifications:
 - "**Lemma**":
proven statement, stepping stone in larger proof
 - "**Prefix**" x of a path s :
subpath starting from the same node as s
 - E.g. $s=(a,c,z,e,d)$, short $aczed$
 - All prefixes x : $a, ac, acz, acze, aczed$
 - E.g. not a prefix: $ab, ace, acezd$ (order is important!)

Prefixes

- Which of the following are prefixes of the path **aiiscool**?

aicool

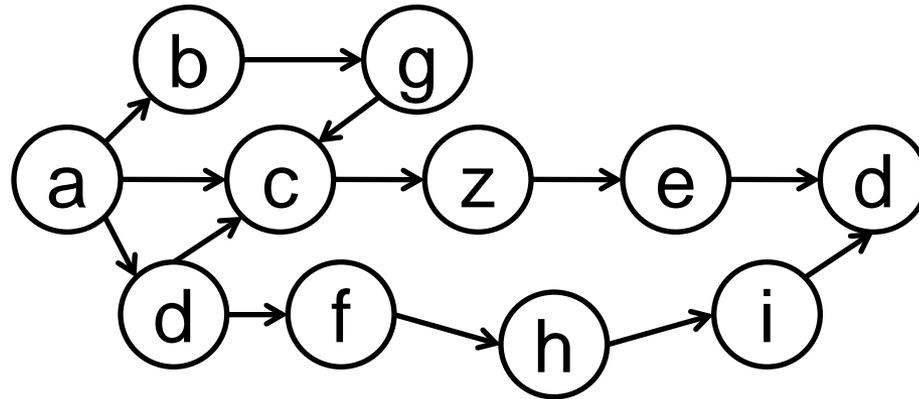
ai

aii

aisc

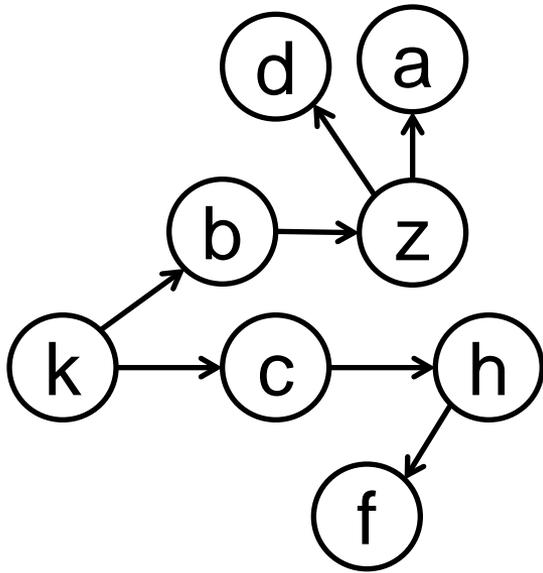
- ai and aii
- aiisc is different from aisc !
 - The optimal solution won't have a cycle if all path costs are > 0

Recap: A* admissibility

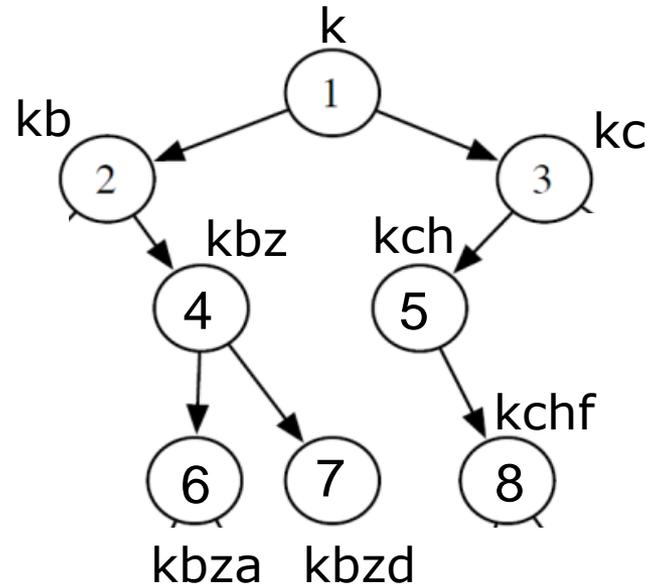


- f_{\min} : = cost of **optimal solution path s** (e.g. $s=aczed$)
 - Cost is unknown but finite if a solution exists
- **Lemmas for prefix x of s** (exercise: prove at home)
 - Has cost $f(x) \leq f_{\min}$ (due to admissibility)
 - Always one such x on the frontier (by induction)
- Used these Lemmas to prove:
A* only expands paths x with $f(x) \leq f_{\min}$
- Then we're basically done!
 - Only finite number of such paths (\Rightarrow completeness)
 - Solution with cost $> f_{\min}$ won't be expanded (\Rightarrow optimality)

Clarification: state space **graph** vs search **tree**



State space graph.

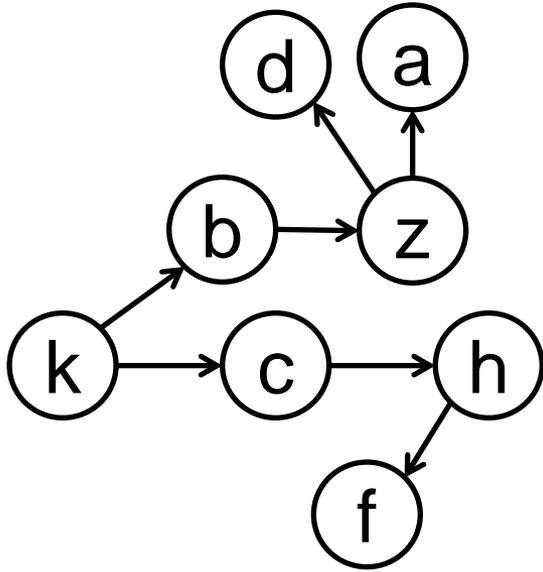


Search tree.

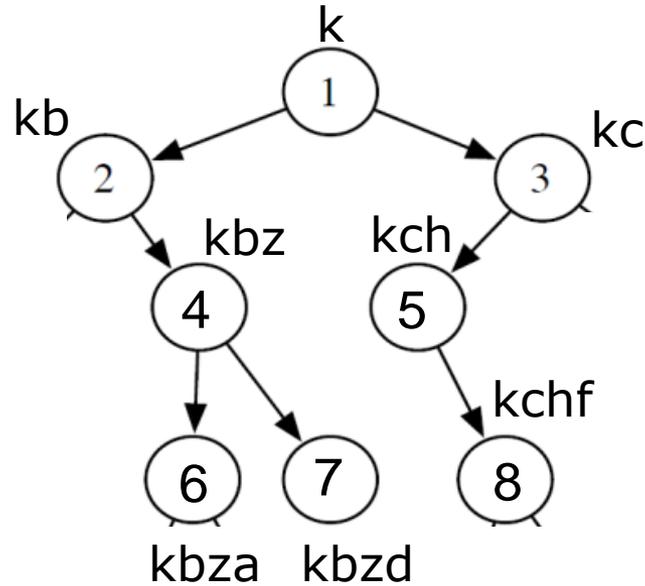
Nodes in this tree correspond to **paths in the state space graph**

If there are no cycles, the two look the same

Clarification: state space **graph** vs search **tree**



State space graph.



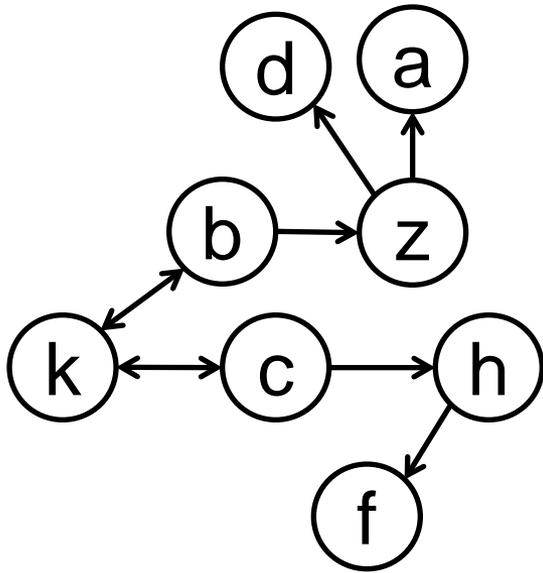
Search tree.

What do I mean by the numbers in the search tree's nodes?

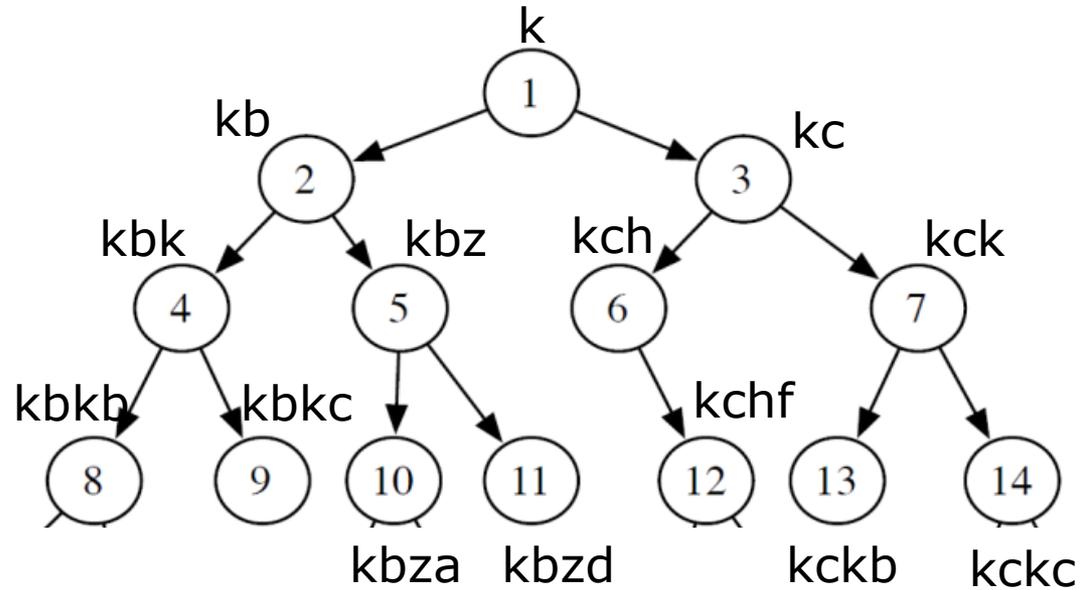
Node's name

Order in which a search algo. (here: BFS) expands nodes

Clarification: state space **graph** vs search **tree**



State space graph.

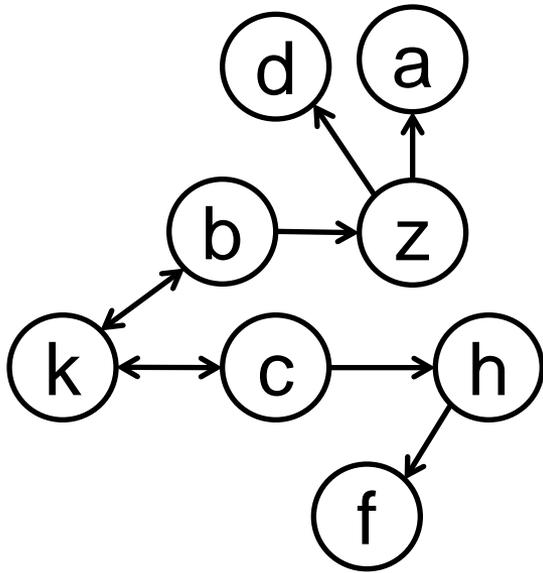


Search tree.

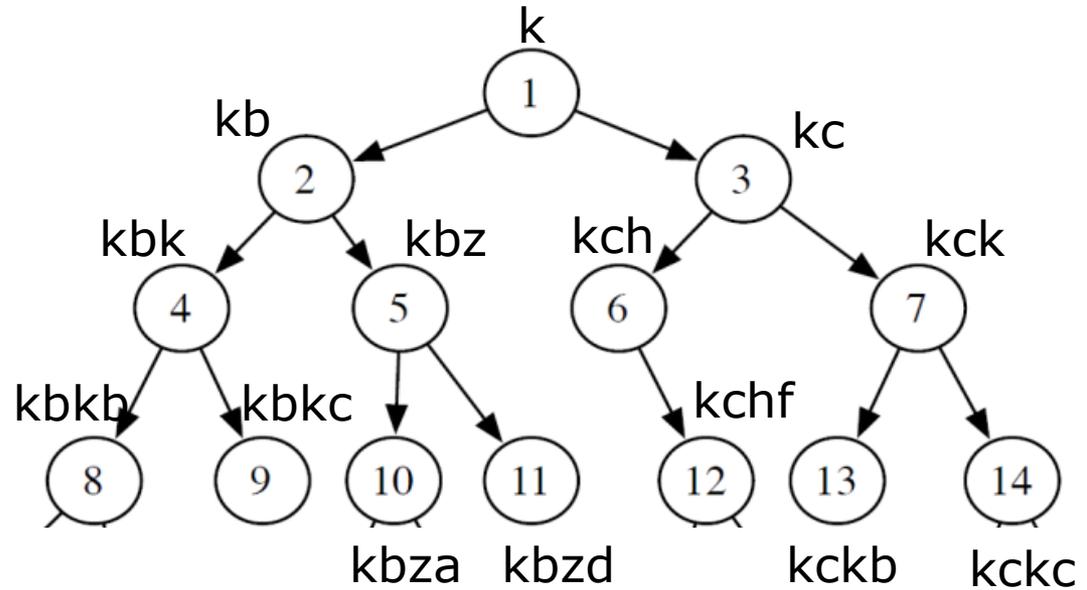
(only first 3 levels, of BFS)

- If there are cycles, the two look very different

Clarification: state space **graph** vs search **tree**



State space graph.



Search tree.

(only first 3 levels, of BFS)

What do nodes in the search tree represent in the state space?

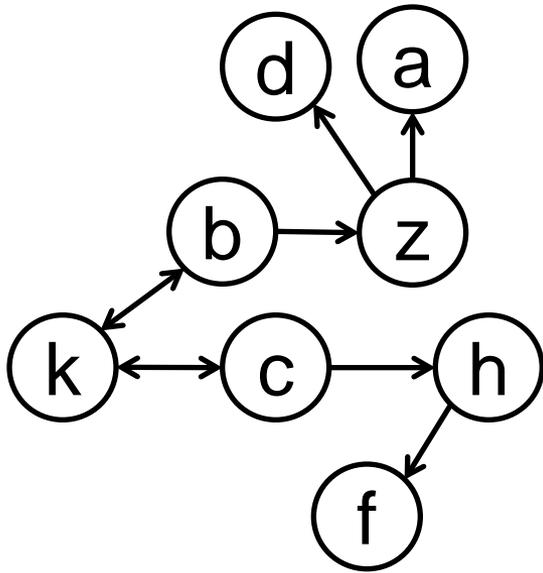
nodes

edges

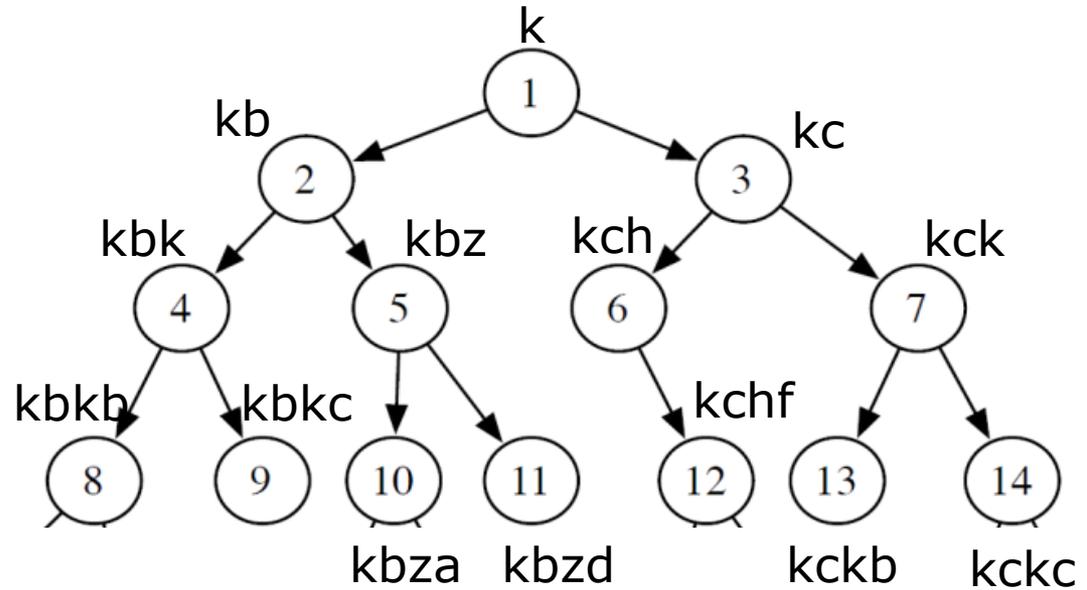
paths

states

Clarification: state space **graph** vs search **tree**



State space graph.



Search tree.

(only first 3 levels, of BFS)

What do edges in the search tree represent in the state space?

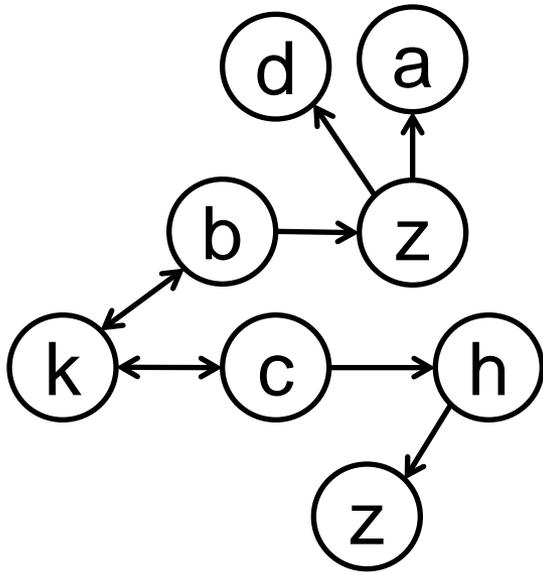
nodes

edges

paths

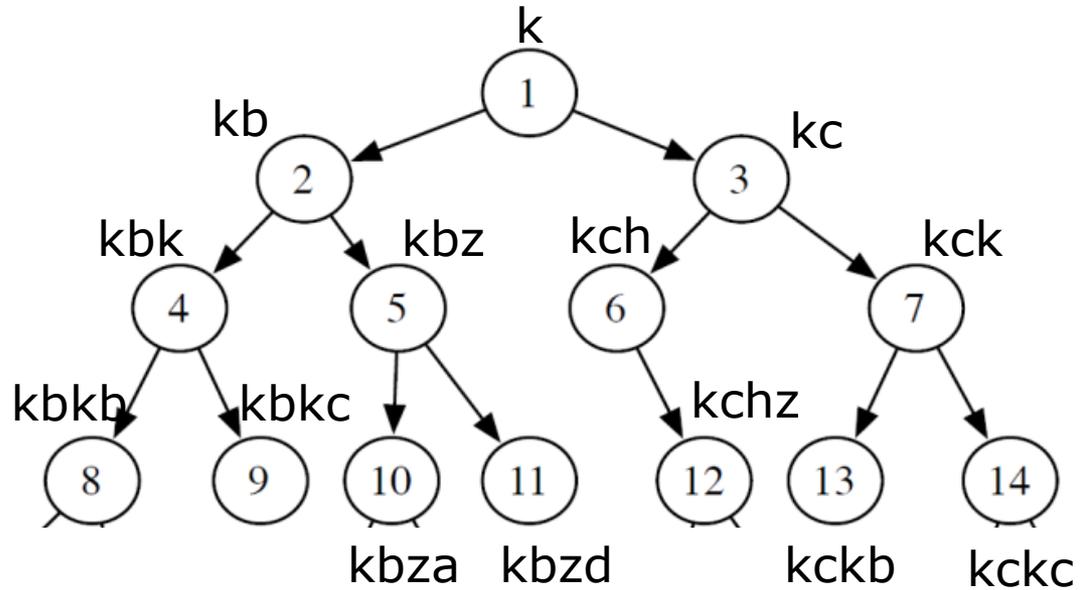
states

Clarification: state space **graph** vs search **tree**



State space graph.

May contain cycles!



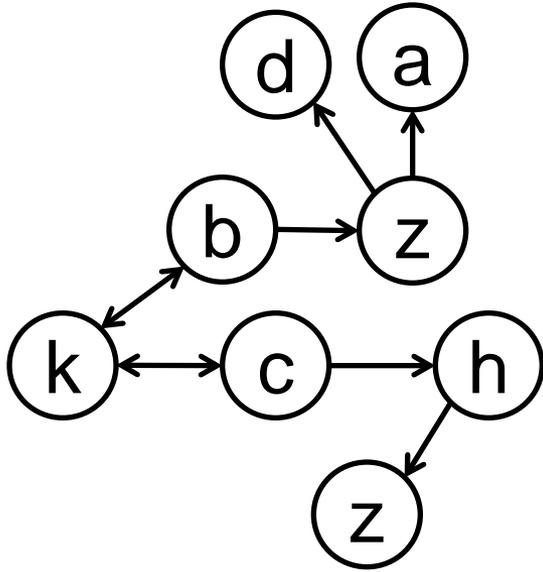
Search tree.

Nodes in this tree correspond to **paths in the state space graph**

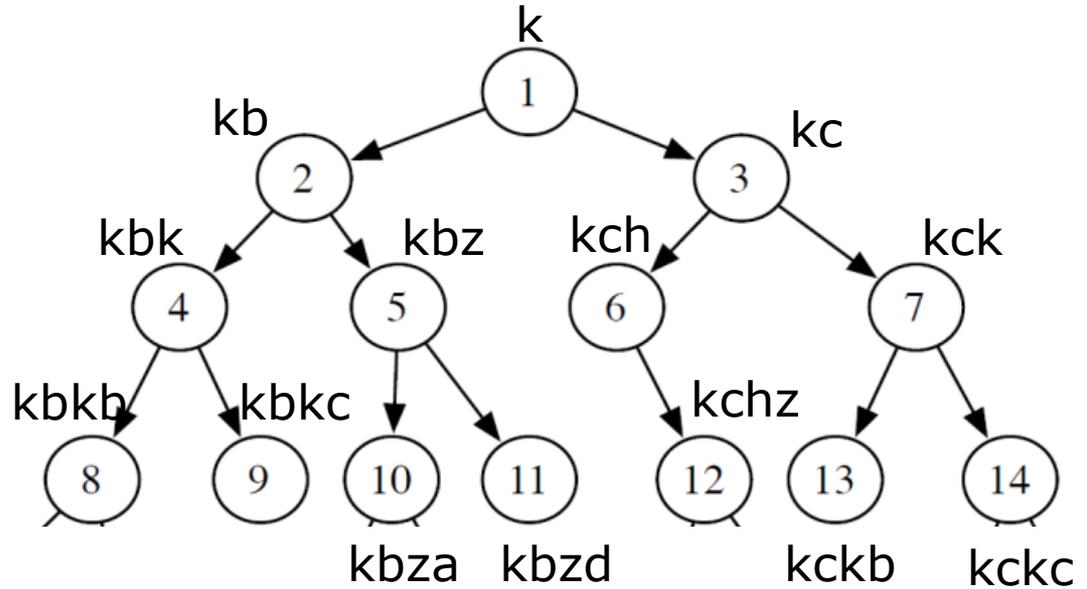
(if multiple start nodes: forest)

Cannot contain cycles!

Clarification: state space **graph** vs search **tree**



State space graph.



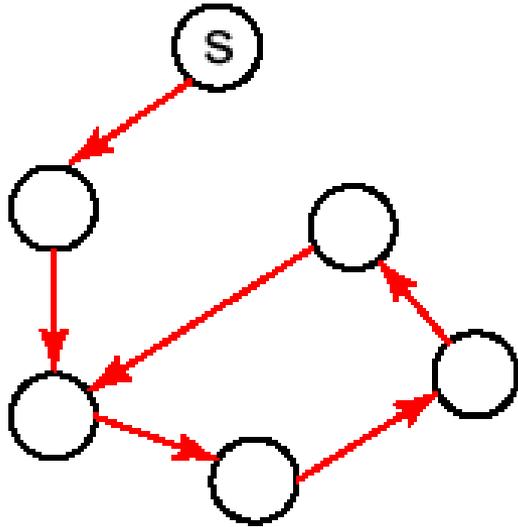
Search tree.

Nodes in this tree correspond to **paths in the state space graph**

Why don't we just eliminate cycles?

Sometimes (but not always) we want multiple solution paths

Cycle Checking: if we only want optimal solutions

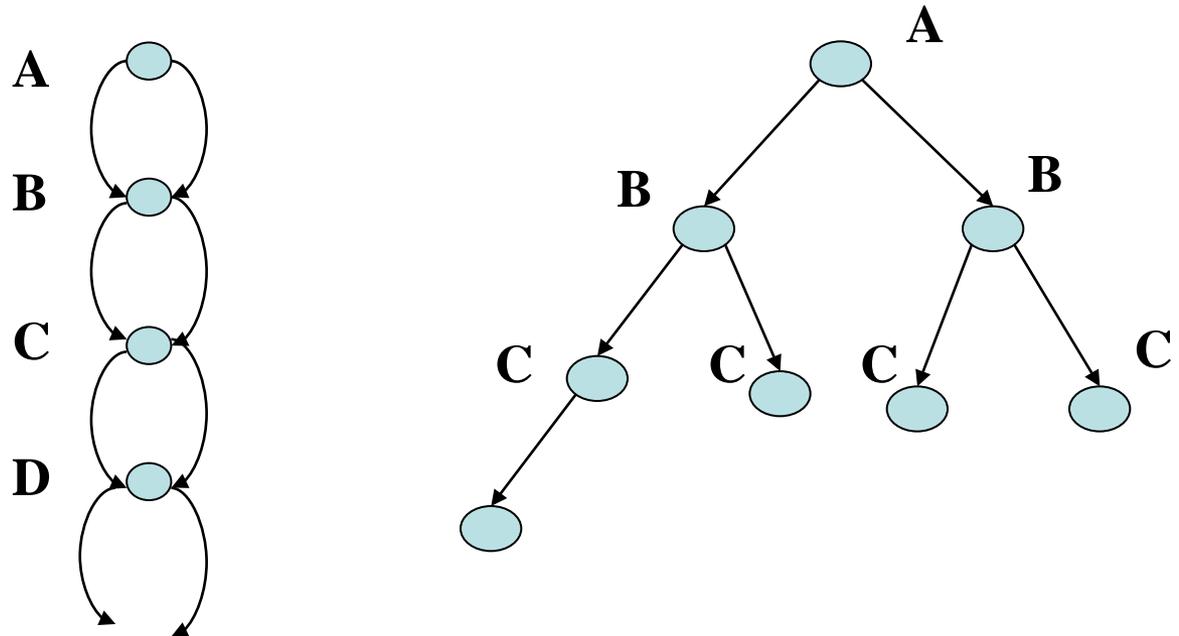


- You can **prune** a node n that is on the path from the start node to n .
- This pruning cannot remove an optimal solution \Rightarrow **cycle check**

- Using depth-first methods, with the graph explicitly stored, this can be done in constant time
 - Only one path being explored at a time
- Other methods: cost is linear in path length
 - (check each node in the path)

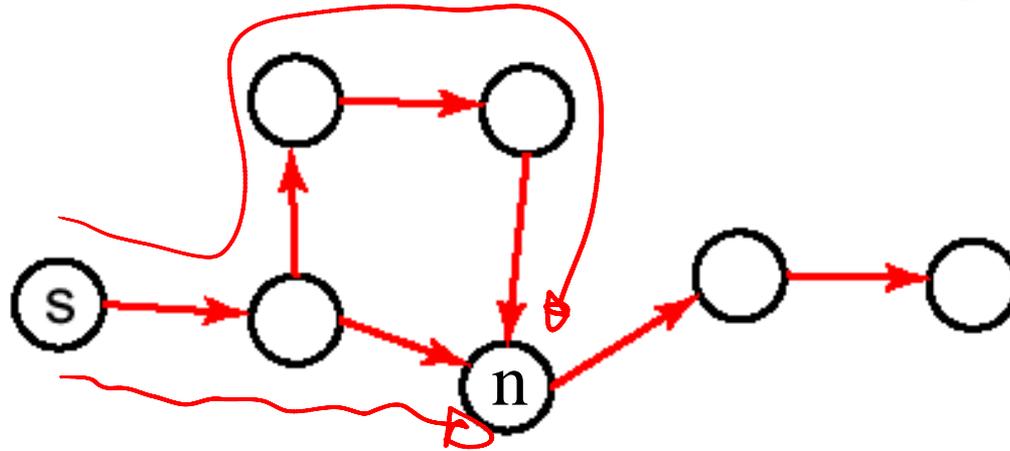
Size of search space vs search tree

- With cycles, search tree can be **exponential** in the state space
 - E.g. state space with 2 actions from each state to next
 - With $d + 1$ states, search tree has depth d



- **2^d possible paths through the search space
=> exponentially larger search tree!**

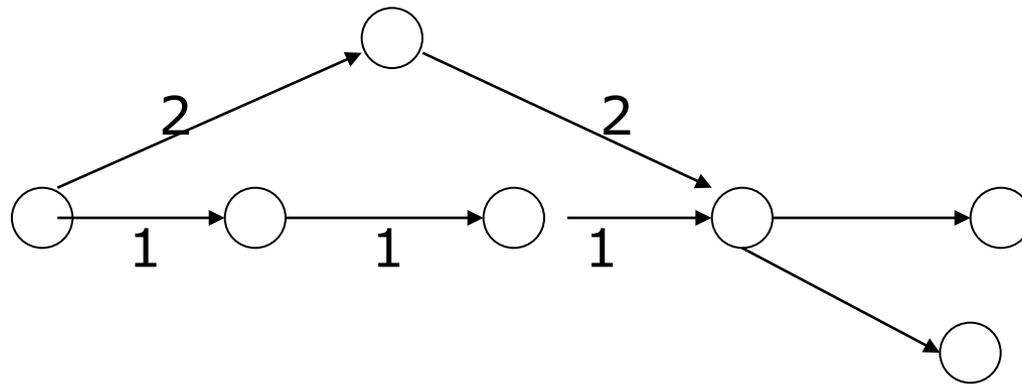
Multiple Path Pruning



- If we only want one path to the solution
- Can prune path to a node n that has already been reached via a previous path
 - Store $S := \{\text{all nodes } n \text{ that have been expanded}\}$
 - For newly expanded path $p = (n_1, \dots, n_k, n)$
 - Check whether $n \in S$
 - Subsumes cycle check
- Can implement by storing the path to each expanded node

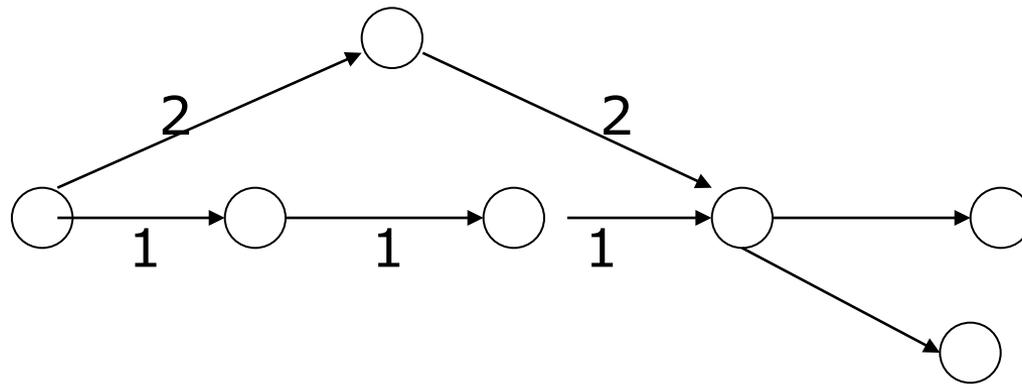
Multiple-Path Pruning & Optimal Solutions

- Problem: what if a subsequent path to n is shorter than the first path to n , and we want an optimal solution ?
- Can remove all paths from the frontier that use the longer path. (these can't be optimal)



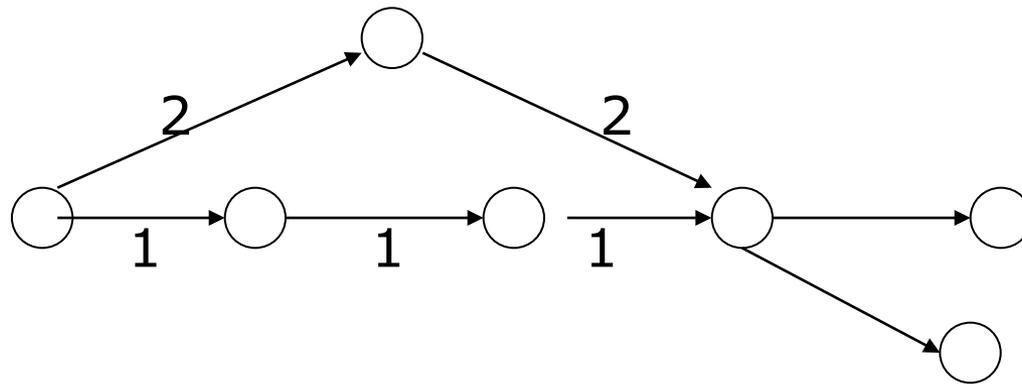
Multiple-Path Pruning & Optimal Solutions

- Problem: what if a **subsequent path to n is shorter** than the first path to n , and we want just the optimal solution ?
- Can **change the initial segment** of the paths on the frontier to use the shorter path



Multiple-Path Pruning & Optimal Solutions

- Problem: what if a **subsequent path to n is shorter** than the first path to n , and we want just the optimal solution ?
- Can **prove that this can't happen** for an algorithm



- Which of the following algorithms always find the shortest path to nodes on the frontier first?

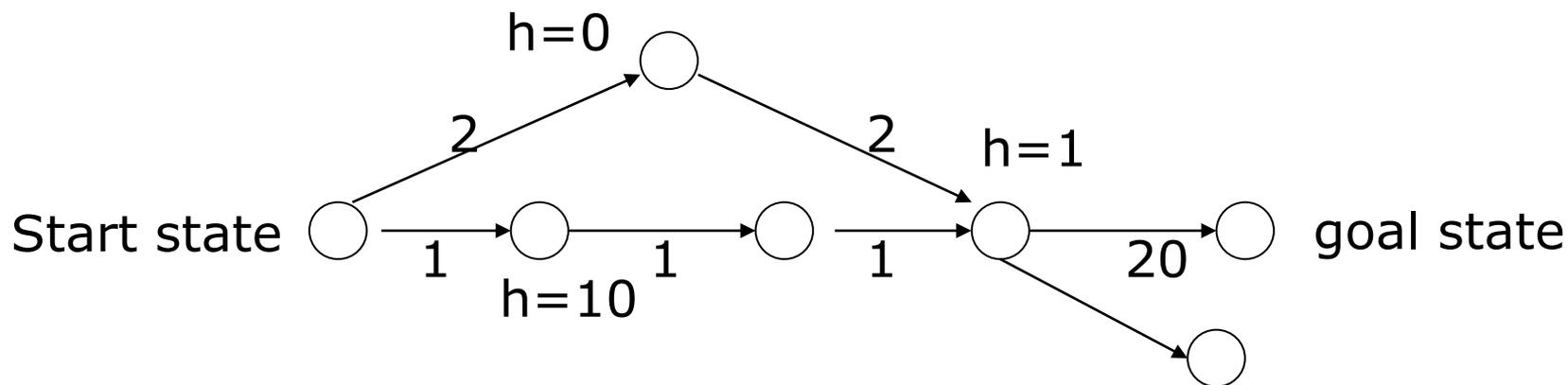
Least Cost Search First

A*

Both of the above

None of the above

- Which of the following algorithms always find the shortest path to nodes on the frontier first?
 - Only Least Cost First Search (like Dijkstra's algorithm)
 - For A* this is only guaranteed for nodes on the optimal solution path
- Example: A* expands the upper path first
 - Special conditions on the heuristic can recover the guarantee of LCFS

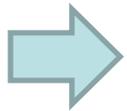


Summary: pruning

- Sometimes we don't want pruning
 - Actually want multiple solutions (including non-optimal ones)
- Search tree can be exponentially larger than search space
 - So pruning is often important
- In DFS-type search algorithms
 - We can do cheap cycle checks: $O(1)$
- BFS-type search algorithms are memory-heavy already
 - We can store the path to each expanded node and do multiple path pruning

Lecture Overview

- Some clarifications & multiple path pruning



Recap and more detail: Iterative Deepening

Iterative Deepening DFS (short IDS): Motivation

Want **low space complexity** but completeness and optimality

Key Idea: re-compute elements of the frontier rather than saving them

	Complete	Optimal	Time	Space
DFS	N (Y if no cycles)	N	$O(b^m)$	$O(mb)$
BFS	Y	Y	$O(b^m)$	$O(b^m)$
LCFS (when arc costs available)	Y Costs > 0	Y Costs >=0	$O(b^m)$	$O(b^m)$
Best First (when h available)	N	N	$O(b^m)$	$O(b^m)$
A* (when arc costs and h available)	Y Costs > 0 h admissible	Y Costs >=0 h admissible	$O(b^m)$	$O(b^m)$

Iterative Deepening DFS (IDS) in a Nutshell

- Depth-bounded depth-first search: **DFS on a leash**
 - For depth bound d , ignore any paths with longer length:
 - Not allowed to go too far away \Rightarrow backtrack (“fail unnaturally”)
 - Only finite # paths with length $\leq d \Rightarrow$ **terminates**
 - What is the memory requirement at depth bound d ? (it is DFS!)
 - m =length of optimal solution path
 - b =branching factor

$$O(b^m)$$

$$O(mb)$$

$$O(bd)$$

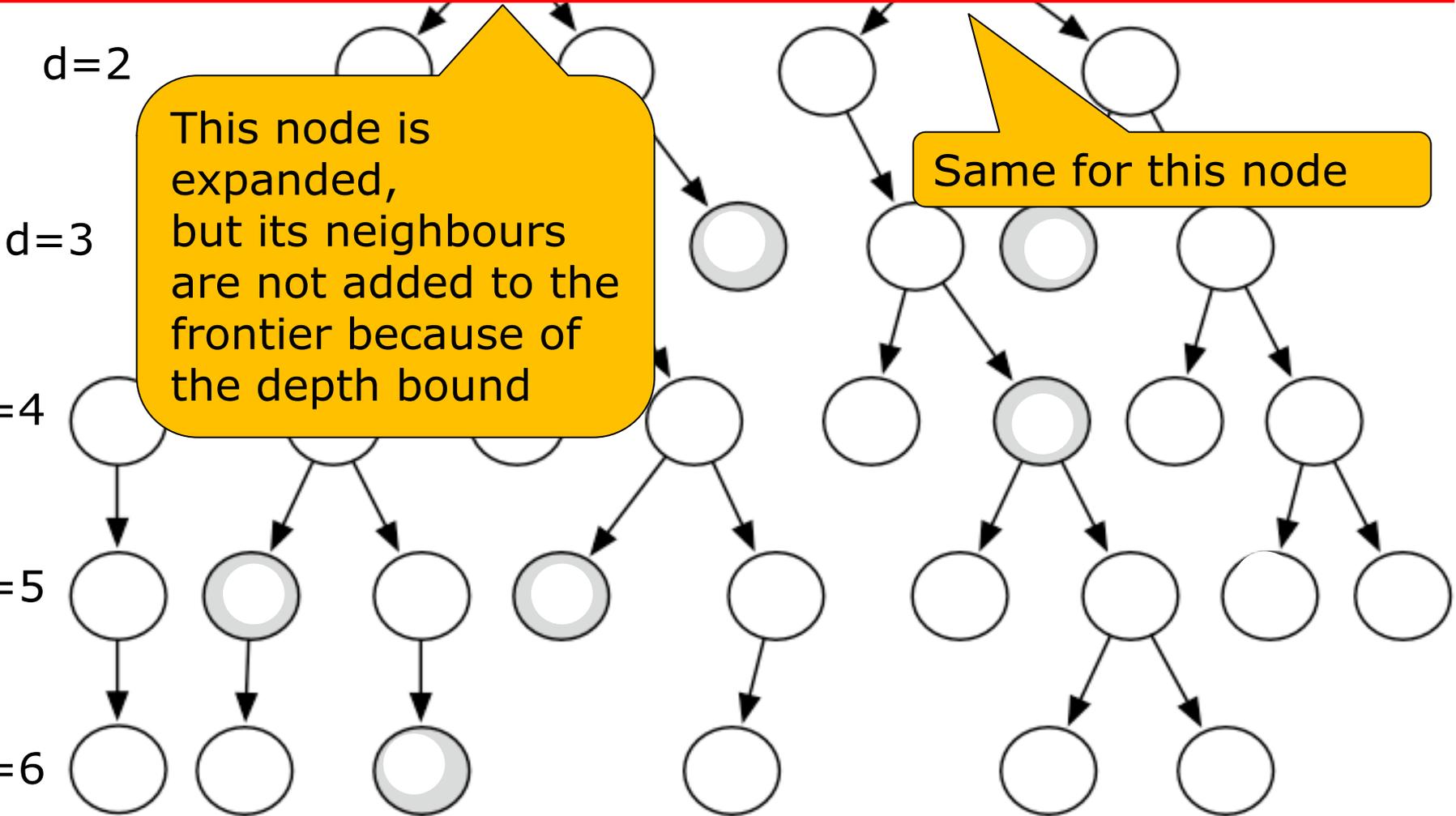
$$O(dm)$$

- $O(bd)$! It's a DFS, up to depth d .
- Progressively increase the depth bound d
 - Start at 1
 - Then 2
 - Then 3
 - ...
 - Until it finds the solution at depth m

Iterative Deepening DFS, depth bound = 1

Numbers in nodes: when expanded?

Depth d=1



d=2

d=3

This node is expanded, but its neighbours are not added to the frontier because of the depth bound

Same for this node

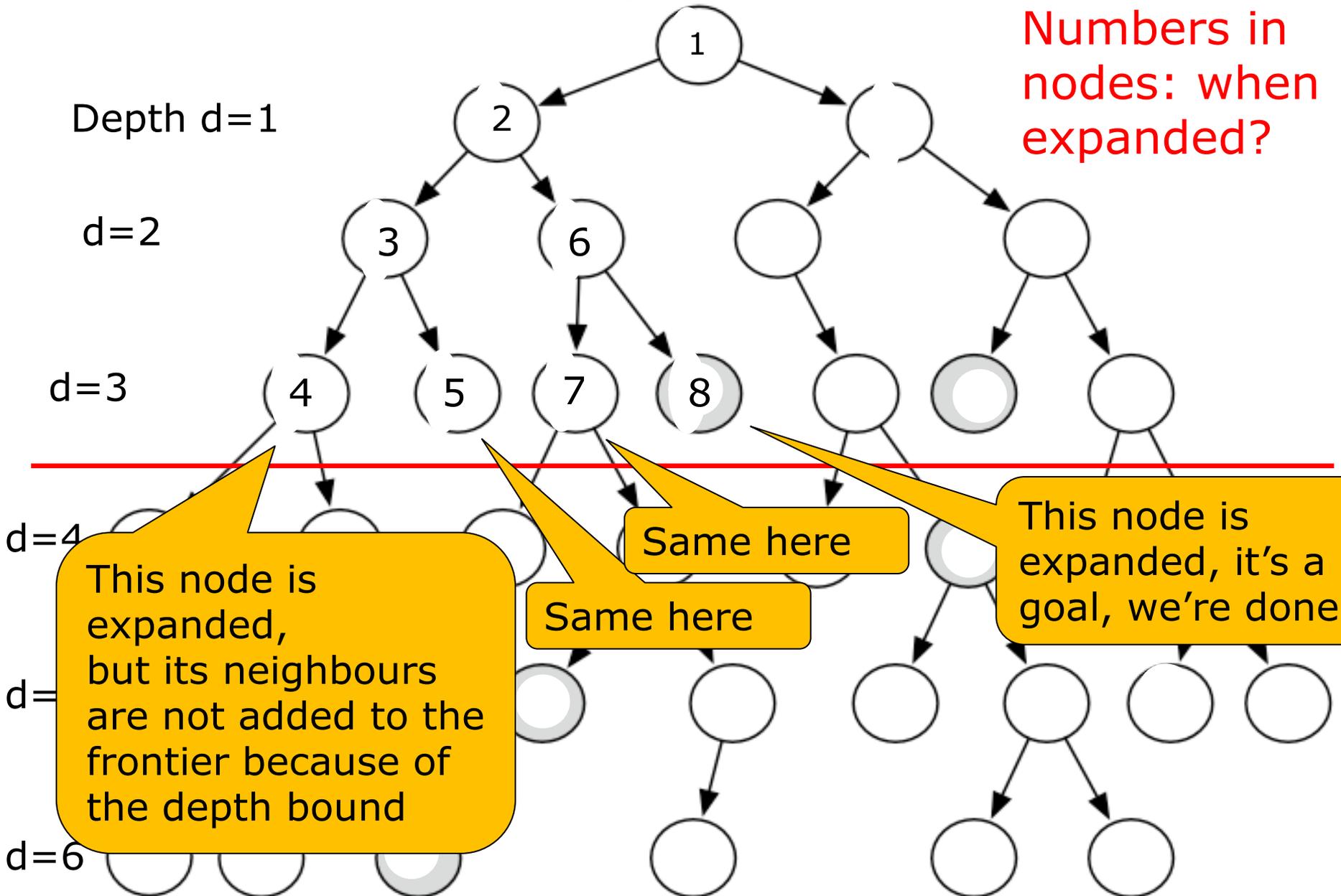
d=4

d=5

d=6

Iterative Deepening DFS, depth bound = 3

Numbers in nodes: when expanded?



Analysis of Iterative Deepening DFS (IDS)

- Space complexity

$O(b^m)$

$O(m^b)$

$O(bm)$

$O(b+m)$

- DFS scheme, only explore one branch at a time

- Complete?

Yes

No

- Only finite # of paths up to depth m , doesn't explore longer paths

- Optimal?

Yes

No

- Proof by contradiction

(Time) Complexity of IDS

The solution is at depth m , branching factor b

Total # of paths generated:

$$\leq b^m + (2 b^{m-1}) + (3 b^{m-2}) + \dots + mb$$

We only expand paths at depth m once

We only expand paths at depth $m-2$ three times

We only expand paths at depth $m-1$ twice

We expand paths at depth 1 m times (for every single depth bound)

(Time) Complexity of IDS

From there on, it's just math:

Total # paths generated by IDS

$$\leq b^m + (2 b^{m-1}) + (3 b^{m-2}) + \dots + mb$$

$$= b^m (1 b^0 + 2 b^{-1} + 3 b^{-2} + \dots + m b^{1-m})$$

$$= b^m \left(\sum_{i=1}^m i b^{1-i} \right) = b^m \left(\sum_{i=1}^m i (b^{-1})^{i-1} \right)$$

$$\leq b^m \left(\sum_{i=0}^{\infty} i (b^{-1})^{i-1} \right) = b^m \left(\frac{1}{1-b^{-1}} \right)^2 = b^m \left(\frac{b}{b-1} \right)^2 \in O(b^m)$$

If $b > 1$

Geometric progression: for $|r| < 1$: $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$

$$\frac{d}{dr} \sum_{i=0}^{\infty} r^i = \sum_{i=0}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}$$

Conclusion for Iterative Deepening

- Even though it redoes what seems like a lot of work
 - Actually, compared to how much work there is at greater depths, it's not a lot of work
 - Redoes the first levels most often
 - But those are the cheapest ones
- Time Complexity $O(b^m)$
 - Just like a single DFS
 - Just like the last depth-bounded DFS
 - That last depth bounded DFS **dominates the search complexity**
- Space complexity: $O(bm)$
- Optimal
- Complete

(Heuristic) Iterative Deepening: IDA*

- Like Iterative Deepening DFS
 - But the “depth” bound is measured in terms of the f value
 - f-value-bounded DFS: DFS on a f-value leash
 - IDA* is a bit of a misnomer
 - The only thing it has in common with A* is that it uses the f value $f(p) = \text{cost}(p) + h(p)$
 - It does NOT expand the path with lowest f value. It is doing DFS!
 - But f-value-bounded DFS doesn't sound as good ...
- If you don't find a solution at a given f-value
 - Increase the bound:
to the minimum of the f-values that exceeded the previous bound
- Will explore all nodes with f value $< f_{\min}$ (optimal one)

Analysis of Iterative Deepening A* (IDA*)

- Complete and optimal? Same conditions as A*
 - h is admissible
 - all arc costs > 0
 - finite branching factor
- Time complexity: $O(b^m)$
 - Same argument as for Iterative Deepening DFS
- Space complexity:

$O(b^m)$

$O(m^b)$

$O(bm)$

$O(b+m)$

- Same argument as for Iterative Deepening DFS

Search methods so far

	Complete	Optimal	Time	Space
DFS	N (Y if no cycles)	N	$O(b^m)$	$O(mb)$
BFS	Y	Y	$O(b^m)$	$O(b^m)$
IDS	Y	Y	$O(b^m)$	$O(mb)$
LCFS (when arc costs available)	Y Costs > 0	Y Costs >=0	$O(b^m)$	$O(b^m)$
Best First (when h available)	N	N	$O(b^m)$	$O(b^m)$
A* (when arc costs and h available)	Y Costs > 0 h admissible	Y Costs >=0 h admissible	$O(b^m)$	$O(b^m)$
IDA*	Y (same cond. as A*)	Y	$O(b^m)$	$O(mb)$

Learning Goals for today's class

- Define/read/write/trace/debug different search algorithms
 - In more detail today: [Iterative Deepening](#),
New today: [Iterative Deepening A*](#)
 - Apply basic properties of search algorithms:
 - completeness, optimality, time and space complexity
-

Announcements:

- New practice exercises are out: see WebCT
 - Heuristic search
 - Branch & Bound
 - Please use these! (Only takes 5 min. if you understood things...)
- Assignment 1 is out: see WebCT

Learning Goals for search

- **Identify** real world examples that make use of deterministic, goal-driven search agents
- **Assess** the size of the search space of a given search problem.
- **Implement** the generic solution to a search problem.
- **Apply** basic properties of search algorithms:
 - completeness, optimality, time and space complexity
- **Select** the most appropriate search algorithms for specific problems.
- **Define/read/write/trace/debug** different search algorithms
- **Construct** heuristic functions for specific search problems
- **Formally prove** A* optimality.
- **Define optimally** efficient

Coming up: Constraint Satisfaction Problems

- Read chapter 4
- Get busy with assignment 1