Search: Advanced Topics

Computer Science cpsc322, Lecture 9

(Textbook Chpt 3.6)

January, 23, 2009

Lecture Overview

$f(n) = c(n) + h(n)$

- **Recap A***
- **A* Optimal Efficiency**
- Branch & Bound
- A* tricks
- Other Pruning
- Dynamic Programming

Optimal efficiency of *A**

- In fact, we can prove something even stronger about *A**: in a sense (given the particular heuristic that is available) **no search algorithm could do better!**

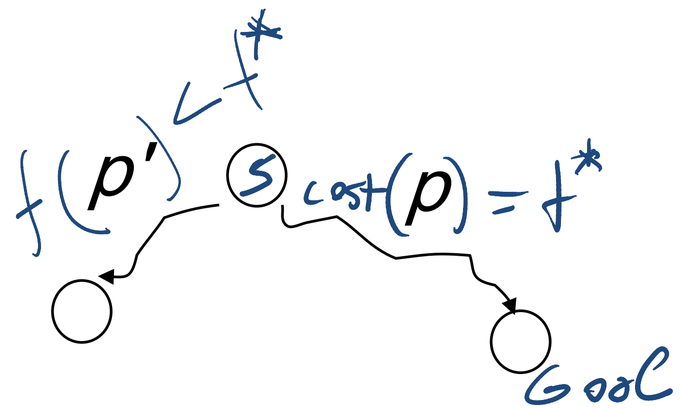
- Optimal Efficiency: Among all optimal algorithms that **start from the same start node** and **use the same heuristic** *h*, *A** expands the minimal number of paths.

Why is A^* optimally efficient?

Theorem: A^* is optimally efficient.

- Let f^* be the cost of the shortest path to a goal.
- Consider any algorithm A'
  - the same start node as A^*,
  - uses the same heuristic
  - fails to expand some path p' expanded by A^*, for which $f(p') < f^*$.
- Assume that A' is optimal.

$f(p') < f^*$

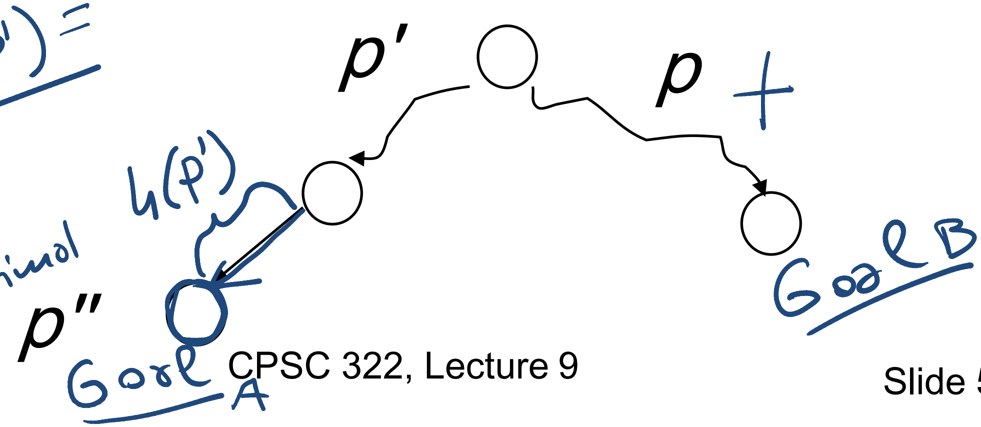


Why is A^* optimally efficient? (cont')

- Consider a different search problem
  - identical to the original
  - on which *h* returns the same estimate for each path
  - except that p' has a child path p'' which is a goal node, and the true cost of the path to p'' is $f(p')$.
  - that is, the edge from p' to p'' has a cost of $h(p')$: the heuristic is exactly right about the cost of getting from p' to a goal.

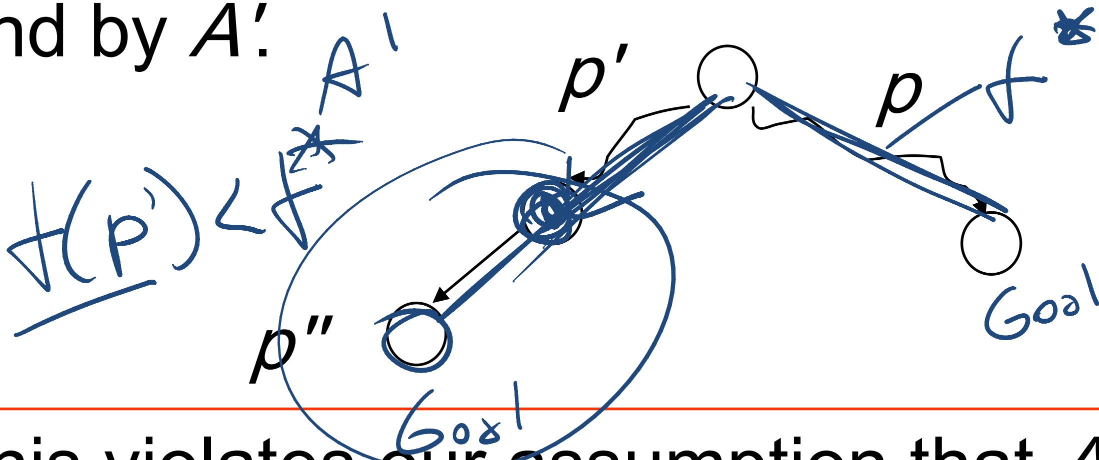
$cost(p'') = cost(p') + h(p') = f(p') < f^*$

so GoalA optimal



Why is A^* optimally efficient? (cont')

- A' would behave identically on this new problem.
  - The only difference between the new problem and the original problem is beyond path p', which A' does not expand.
- Cost of the path to p'' is lower than cost of the path found by A'.



- This violates our assumption that A' is optimal.

Lecture Overview

- Recap A*
- A* Optimal Efficiency
- Branch & Bound
- A^* tricks
- Other Pruning
- Dynamic Programming

Branch-and-Bound Search

- What is the biggest advantage of A*?

$f = c + h$

- What is the biggest problem with A*?

b^m

- Possible Solution:

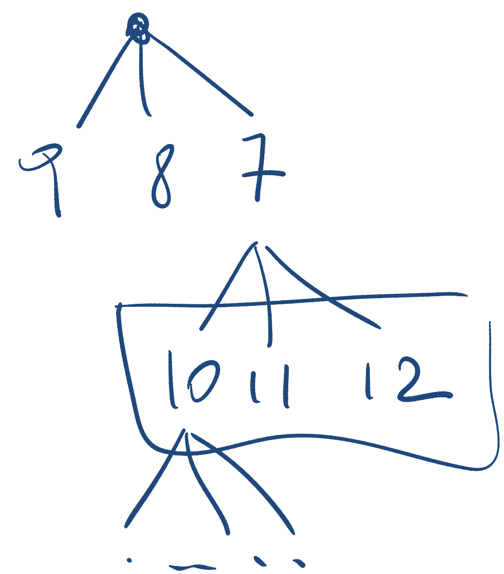
use f

save space

Branch-and-Bound Search Algorithm

- Follow exactly the same search path as **depth-first search**
  - treat the frontier as a stack: expand the most-recently added path first
  - the order in which neighbors are expanded can be governed by some arbitrary node-ordering heuristic

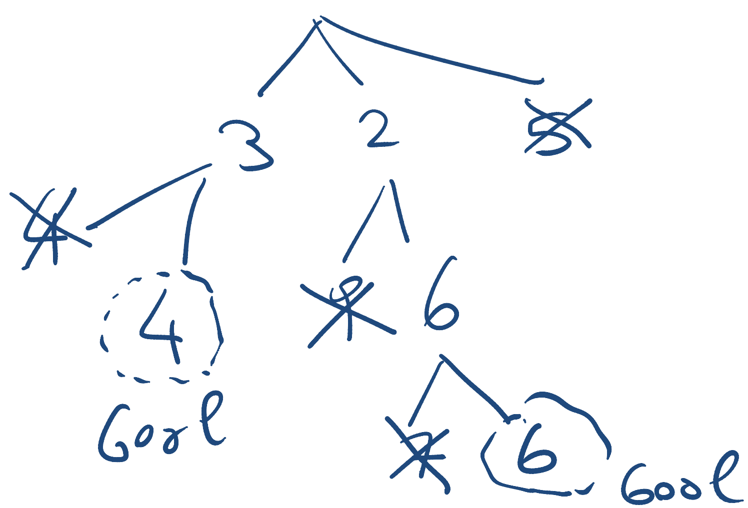
$f = c + h$



Branch-and-Bound Search Algorithm

- Keep track of a lower bound and upper bound on solution cost at each path
  - lower bound: $LB(p) = f(p) = cost(p) + h(p)$
  - upper bound: UB = cost of the best solution found so far.
    - ✓ if no solution has been found yet, set the upper bound to ∞.
- When a path p is selected for expansion:
  - if $LB(p) \geq UB$, remove p from frontier without expanding it (pruning)
  - else expand p, adding all of its neighbors to the frontier

$UB = \infty, 6, 4$



Branch-and-Bound Analysis

- Completeness: no, for the same reasons that DFS isn't complete
  - however, for many problems of interest there are no infinite paths and no cycles
  - hence, for many problems B&B is complete
- Time complexity: $\boxed{O(b^m)}$
- Space complexity: $O(mb)$
  - Branch & Bound has the same space complexity as DFS
  - this is a big improvement over A^*!
- Optimality: yes.

Lecture Overview

- Recap A*
- A* Optimal Efficiency
- Branch & Bound
- A^* tricks
- Pruning Cycles and Repeated States
- Dynamic Programming

Other A^* Enhancements

The main problem with A^* is that it uses exponential space. Branch and bound was one way around this problem. Are there others?

- Iterative deepening A*
- Memory-bounded A^*

(Heuristic) Iterative Deepening – IDA*

B & B can still get stuck in infinite paths

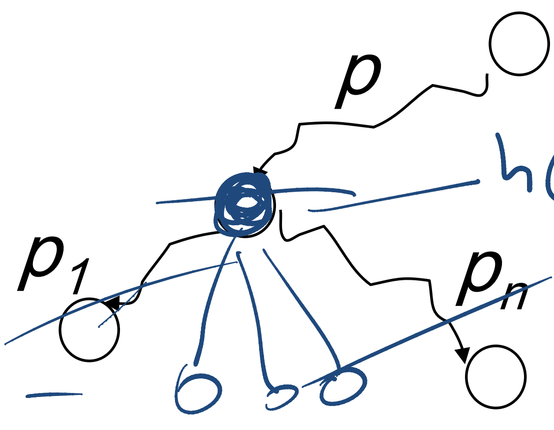
- Search depth-first, but to a fixed depth
  - if you don't find a solution, increase the depth tolerance and try again
  - of course, depth is measured in *f* value

- Counter-intuitively, the asymptotic complexity is not changed, even though we visit paths multiple times (*go back to slides on uninformed ID*)

$\left(\frac{b}{b-1}\right)^2$

Memory-bounded A^*

- Iterative deepening A^* and B & B use a h tiny amount M of memory
- **what if we've got more memory to use?**
- keep as much of the fringe in memory as we can
- if we have to delete something:
  - delete the worst paths (with …highest…f…………)
  - ``back them up'' to a common ancestor

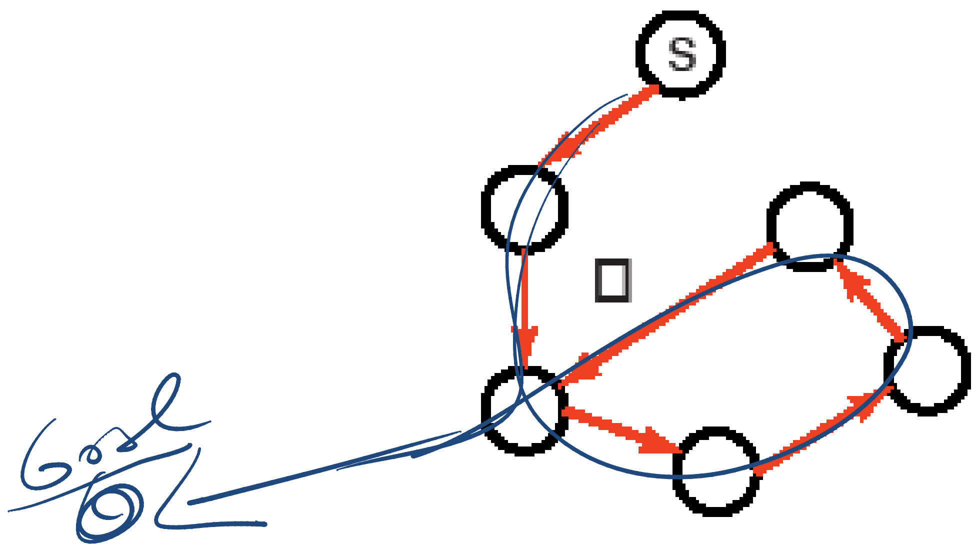


$$h(p) = \min_i \left(\left(cost(p_i) - cost(p) \right) + h(p_i) \right)$$

Lecture Overview

- Recap A*
- A* Optimal Efficiency
- Branch & Bound
- A^* tricks
- Pruning Cycles and Repeated States
- Dynamic Programming

Cycle Checking



linear

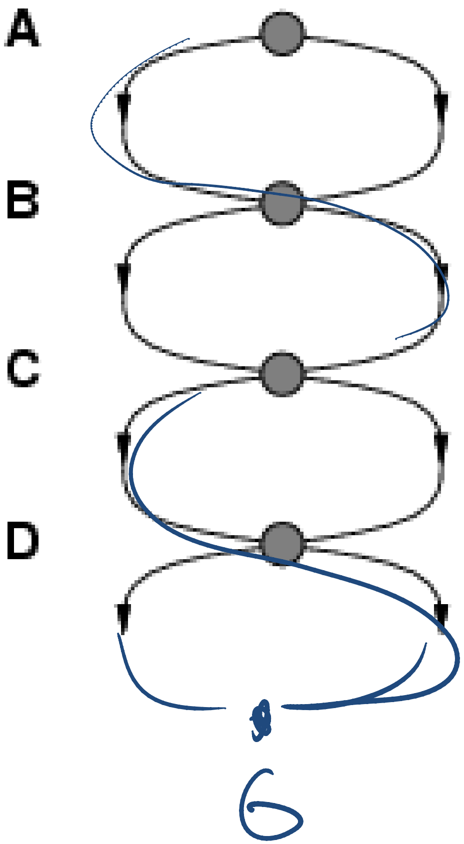
You can prune a path that ends in a node already on the path. This pruning cannot remove an optimal solution.

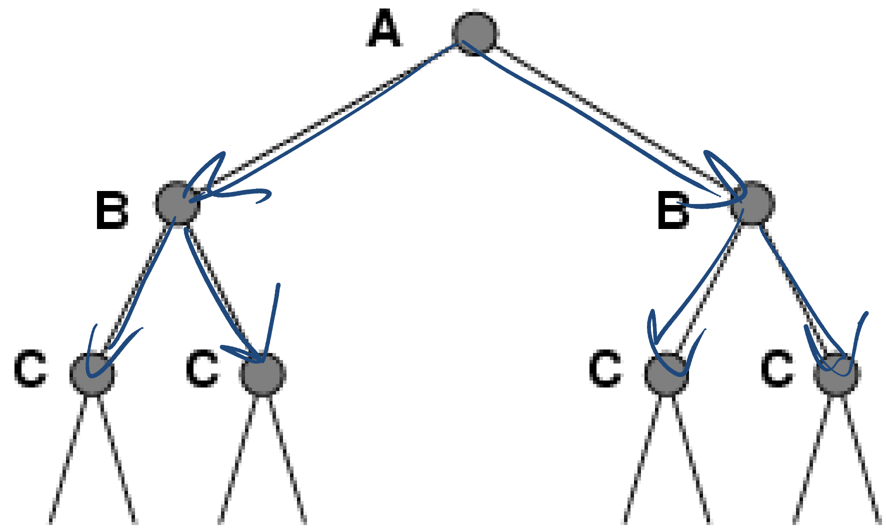
- The ~~cost~~ time is ……linear…… in path length.

$\langle n_1\ n_2 \ldots\ldots\ n_k \rangle$ n_2

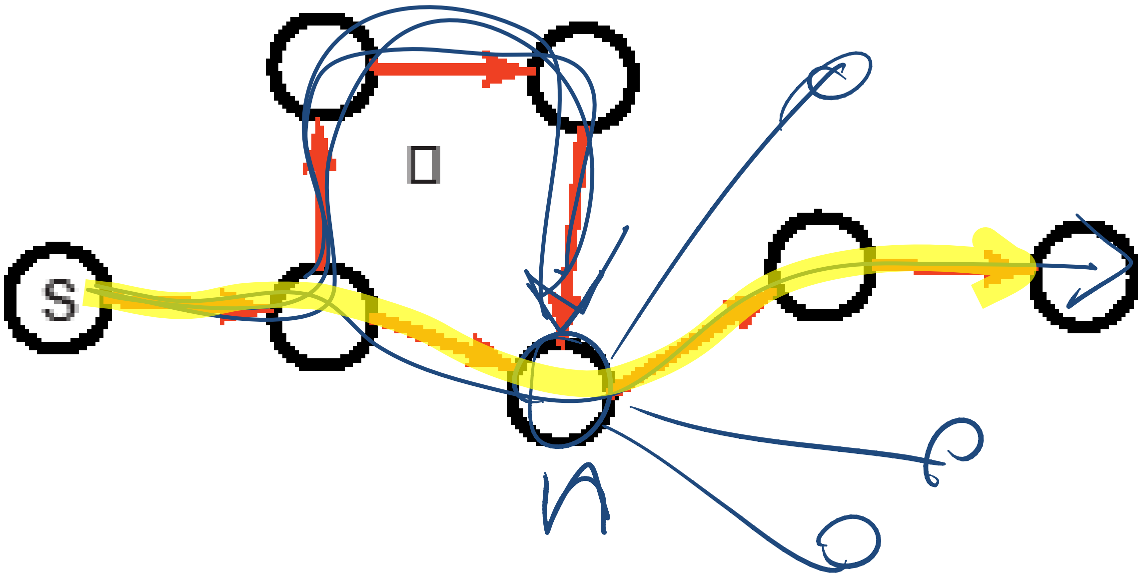
Repeated States / Multiple Paths

Failure to detect repeated states can turn a linear problem into an exponential one!





Multiple-Path Pruning

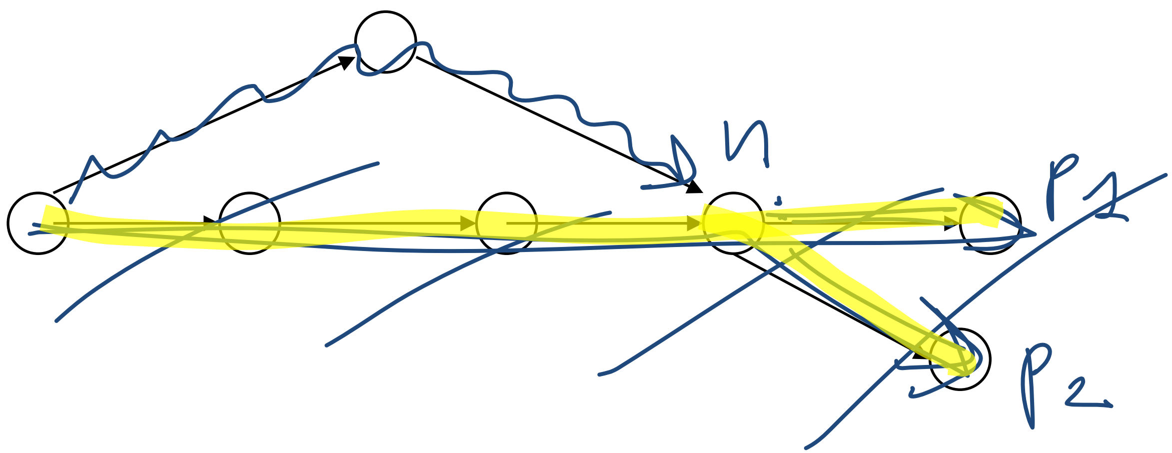


- You can prune a path to node ***n*** that you have already found a path to
- (if the new path is longer – more costly).

Multiple-Path Pruning & Optimal Solutions

Problem: what if a subsequent path to *n* is shorter than the first path to *n* ?

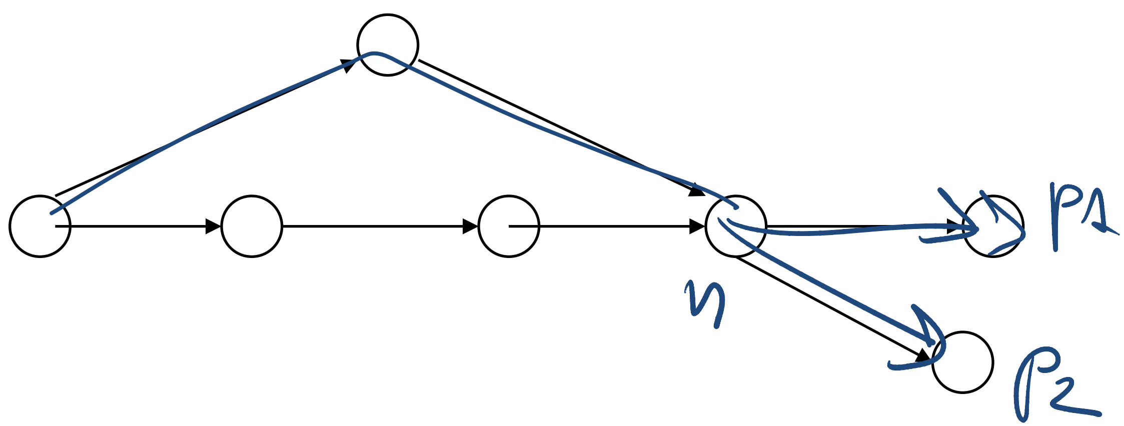
- You can remove all paths from the frontier that use the longer path. (as these can't be optimal)



Multiple-Path Pruning & Optimal Solutions

Problem: what if a subsequent path to *n* is shorter than the first path to *n* ?

- You can change the initial segment of the paths on the frontier to use the shorter path.



Lecture Overview

- Recap A*
- A* Optimal Efficiency
- Branch & Bound
- A^* tricks
- Pruning Cycles and Repeated States
- Dynamic Programming

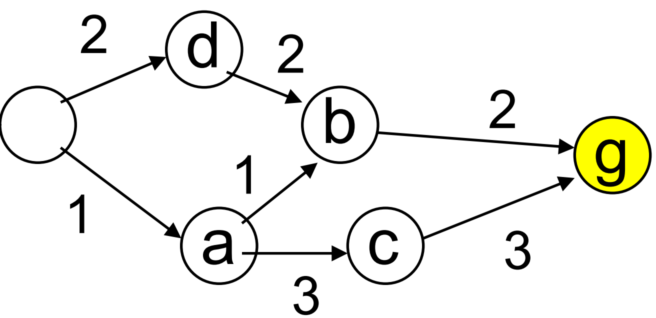
Dynamic Programming

Idea: for statically stored graphs, build a table of *dist(n)* the actual distance of the shortest path from node *n* to a goal.

This is the perfect……..

This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if} \quad is\_goal(n), \\ \min_{\langle n,m\rangle \in A} \left(|\langle n,m\rangle| + dist(m) \right) & \text{otherwise} \end{cases}$$



g

b

c

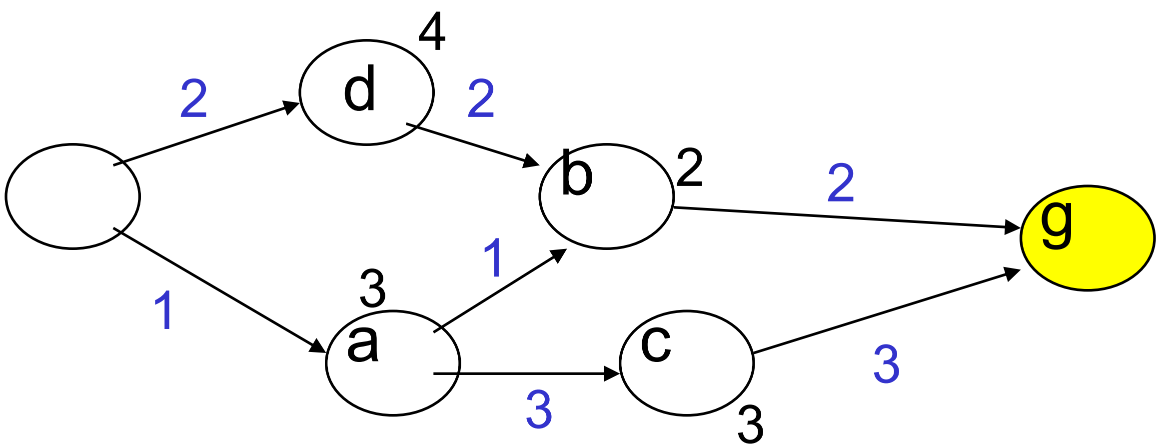
a

Dynamic Programming

This can be used locally to determine what to do.

From each node n go to its neighbor which minimizes

$$\left(\left|\langle n,m\rangle\right| + dist(m)\right)$$



But there are at least two main problems:

- You need enough space to store the graph.
- The *dist* function needs to be recomputed for each goal

Learning Goals for today's class

• Define optimally efficient and formally prove that A* is optimally efficient

•Define/read/write/trace/debug different search algorithms

  •With / Without cost

  •Informed / Uninformed

B & B

IDA* mb

memory bounded

• Pruning cycles and Repeated States

Next class

Recap Search

Start Constraint Satisfaction Problems (CSP)