Searching Strategies

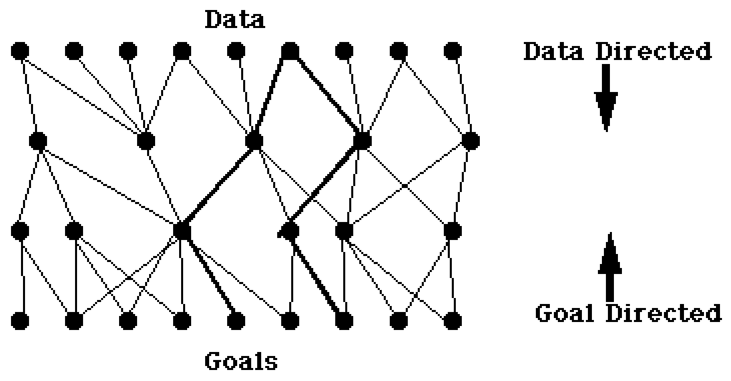
Dr. Isma F. Siddiqui *(isma.farah@faculty.muet.edu.pk)*

A General State-Space Search Algorithm (*Review*)

```
open := {S}; closed :={ };
repeat
    n := select(open);        /* select one node from open for expansion */
   if n is a goal
        then exit with success;  /* delayed goal testing */
   expand(n)
        /* - generate all children of n
           - put these newly generated nodes in open (check duplicates)
           - put n in closed (check duplicates) */
until open = {};
exit with failure
```

Search Directions

- The objective of search procedure is to discover a path through a problem spaces from an initial configuration to a goal state. There are two directions in which a search could proceed:
  - **Forward, from that start states**
  - **Backward, from the goal states**



Forward Search
(Data-directed / Data-Driven Reasoning / Forward Chaining)

This search starts from available information and tries to draw conclusion regarding the situation or the goal attainment. This process continues until (hopefully) it generates a path that satisfies the goal condition.

Backward Search
(Goal directed/driven / Backward Chaining)

This search starts from expectations of what the goal is or what is to happen (hypothesis), then it seeks evidence that supports (or contradicts) those expectations (or hypothesis).

The problem solver begins with the goal to be solved, then finds rules or moves that could be used to generate this goal and determine what conditions must be true to use them.

These conditions become the new goals, **sub goals**, for the search. This process continues, working backward through successive sub goals, until (hopefully) a path is generated that leads back to the facts of the problem.

Goal-driven search uses knowledge of the goal to guide the search.

Use goal-driven search if;

- A goal or hypothesis is given in the problem or can easily be formulated.
- There are a large number of rules that match the facts of the problem and thus produce an increasing number of conclusions or goals. (inefficient)
- Problem data are not given but must be acquired by the problem solver. (impossible)

Data-driven search uses knowledge and constraints found in the given data to search along lines known to be true.

Use data-driven search if:

- All or most of the data are given in the initial problem statement.
- There are a large number of potential goals, but there are only a few ways to use the facts and the given information of a particular problem.
- It is difficult to form a goal or hypothesis.

Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time Complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**

Evaluating Search Strategies

- **Space Complexity**
  - How much space is used by the algorithm? Usually measured in terms of the **maximum size that the “OPEN" list** becomes during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal or highest quality solution among several different solutions? For example, is it the one with minimum cost?

Uninformed vs. Informed Search

- **Uninformed Search Strategies**

  (Blind Search)
  - Breadth-First search
  - Depth-First search
  - Uniform-Cost search
  - Depth-First Iterative Deepening search

Uninformed vs. Informed Search

- **Informed Search Strategies (Heuristic Search)**
  - Hill climbing
  - Best-first search
  - Greedy Search
  - Beam search
  - Algorithm A
  - Algorithm A*

Breadth-First Search (BFS)

Breadth First Search Algorithm:

- In breadth-first search, when a state is examined, all of its siblings are examined before any of its children.
- The space is searched level-by-level, proceeding all the way across one level before going down to the next level.
- BFS is suitable for problems with shallow solutions

Breadth-First Search (BFS)

- **<u>Algorithm outline:</u>**
  - Always select from the **OPEN** the node with the smallest depth for expansion, and put all newly generated nodes into **OPEN**
  - OPEN is organized as **FIFO** (first-in, first-out) list, i.e., a **queue**.
  - Terminate if a node selected for expansion is a goal

Breadth-First Search (BFS)

Properties

- **Complete**
- **Optimal** (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with shortest path length **(shallowest solution)**.
- **Exponential time and space complexity,**

  O(b^d) nodes will be generated, where

  d is the depth of the solution and

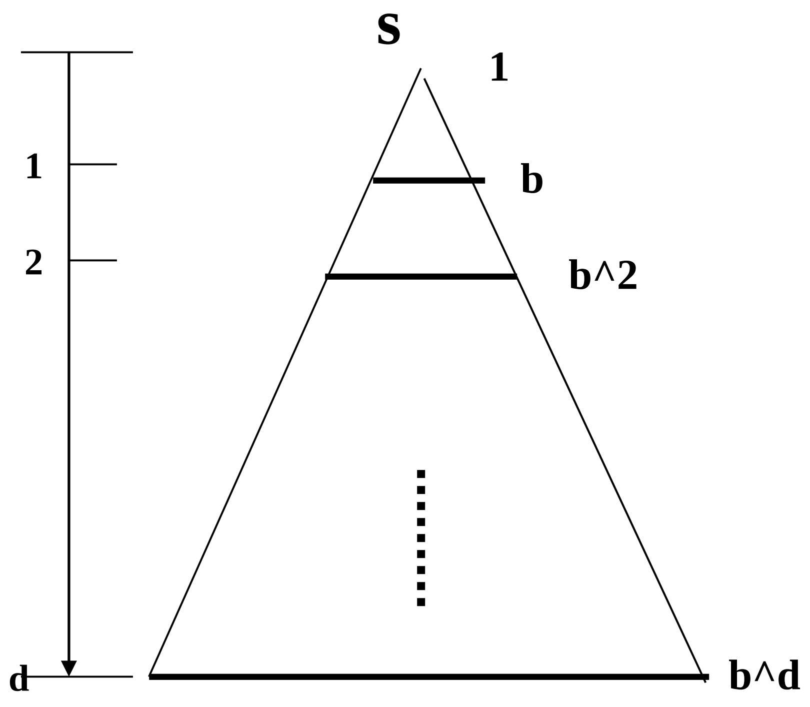
  b is the branching factor (i.e., number of children) at each node.

Breadth-First Search (BFS)

- A complete search tree of depth d where each non-leaf node has b children, has a total of 1 + b + b^2 + ... + b^d = (b^(d+1) - 1)/(b-1) nodes

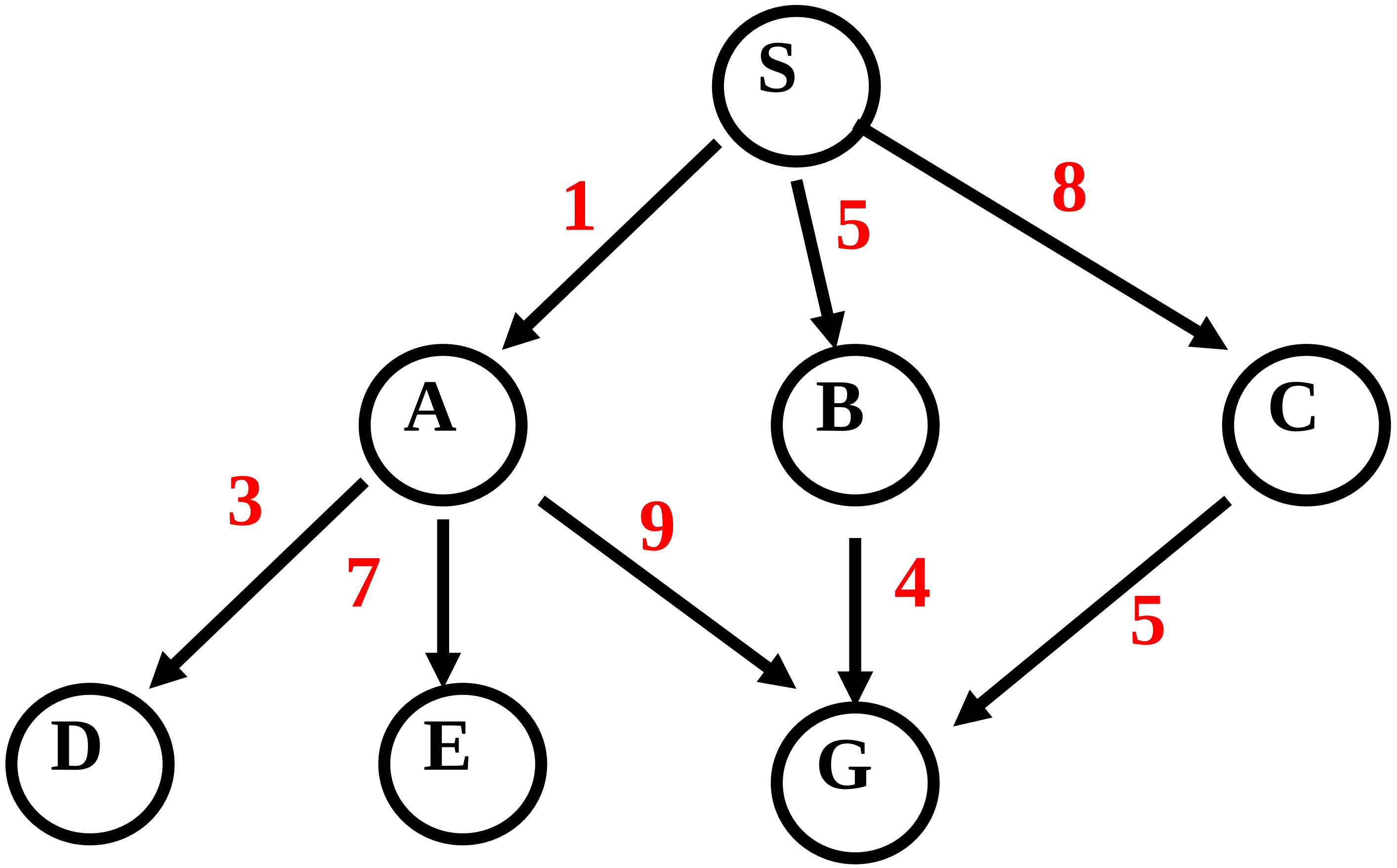
- Time complexity (# of nodes generated): O(b^d)

- Space complexity (maximum length of OPEN): O(b^d)



- For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are 1 + 10 + 100 + 1000 + ... + 10^12 = (10^13 - 1)/9 = O(10^12) nodes in the complete search tree.

Example Illustrating Uninformed Search Strategies

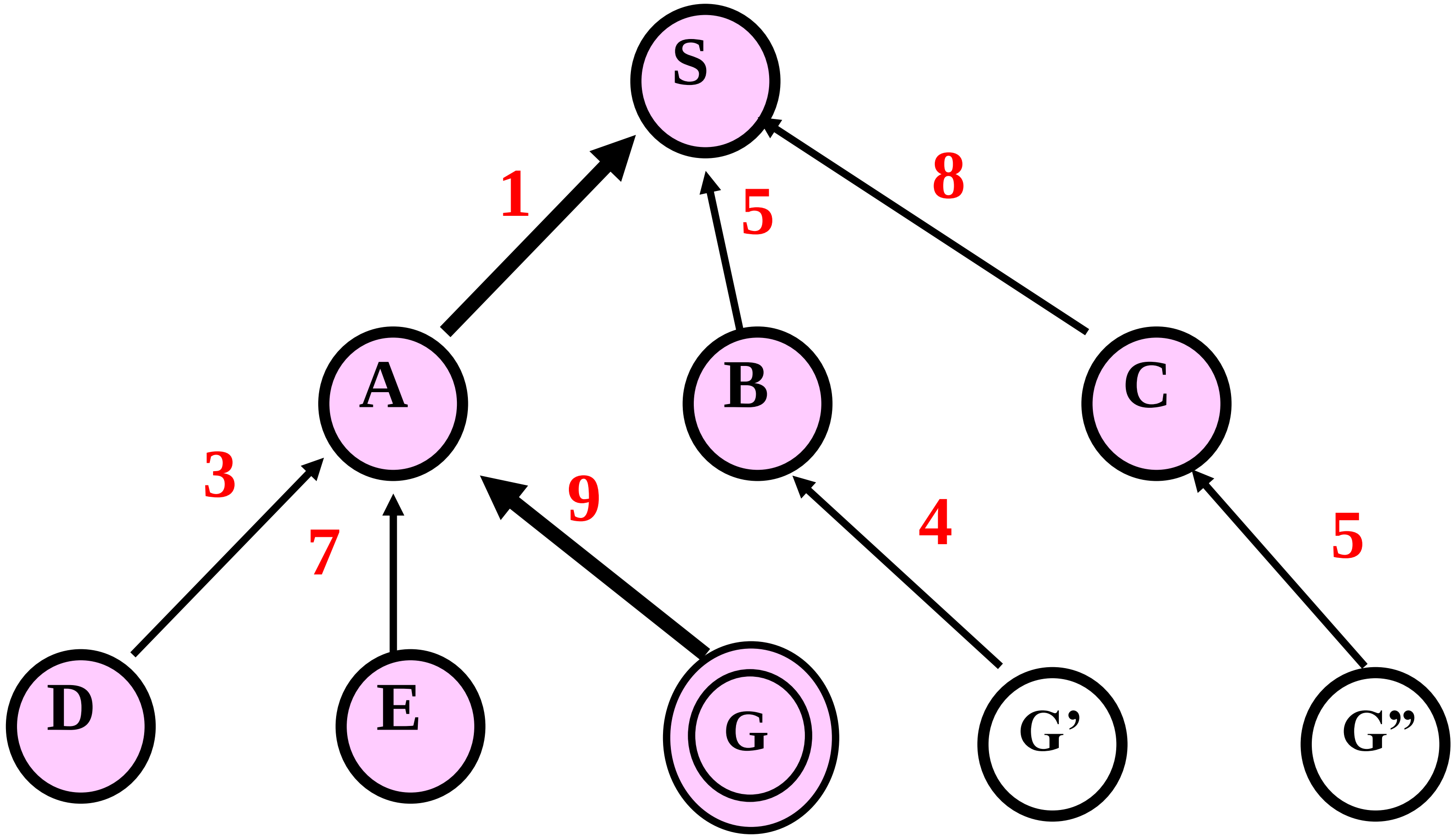


Breadth-First Search

Exp. node	**OPEN list**	**CLOSED list**
	{ S }	{}
S	{ A B C }	{S}
A	{ B C D E G }	{S A}
B	{ C D E G G' }	{S A B}
C	{ D E G G' G" }	{S A B C}
D	{ E G G' G" }	{S A B C D}
E	{ G G' G" }	{S A B C D E}
G	{ G' G" }	{S A B C D E}

- Solution path found is S A G <-- this G also has cost 10
- Number of nodes expanded (including goal node) = 7

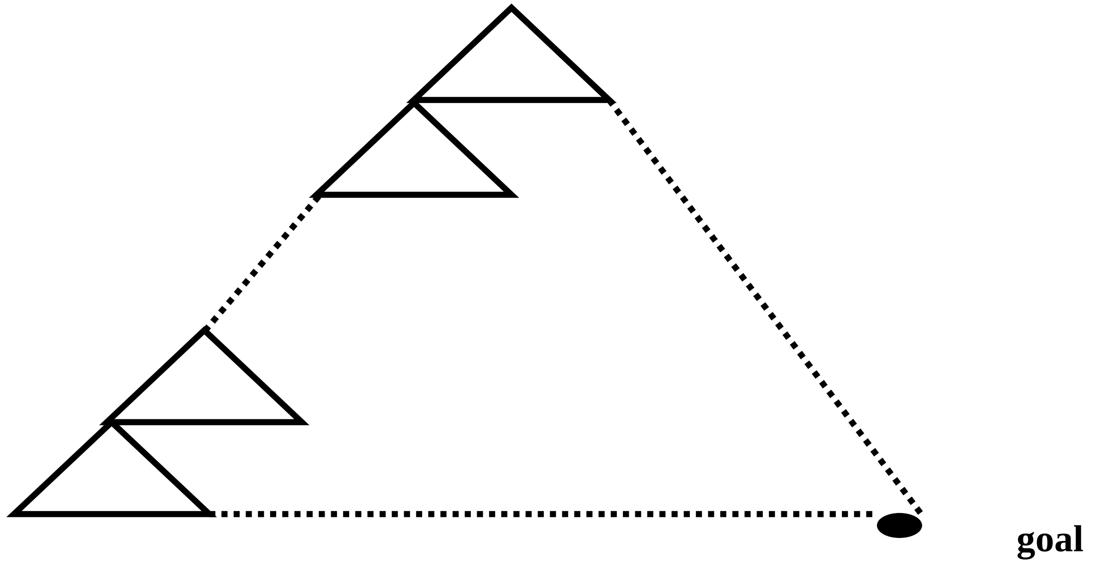
Breadth-First Search



Depth-First Search (DFS)

Depth First Search Algorithm:

- In depth-first search, when a state is examined, all of its children and their descendants are examined before any of its sibling.



Depth-First Search (DFS)

- **Algorithm outline:**
  - Always select from the OPEN the node with the greatest depth for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **LIFO** (last-in, first-out) list.
  - Terminate if a node selected for expansion is a goal.

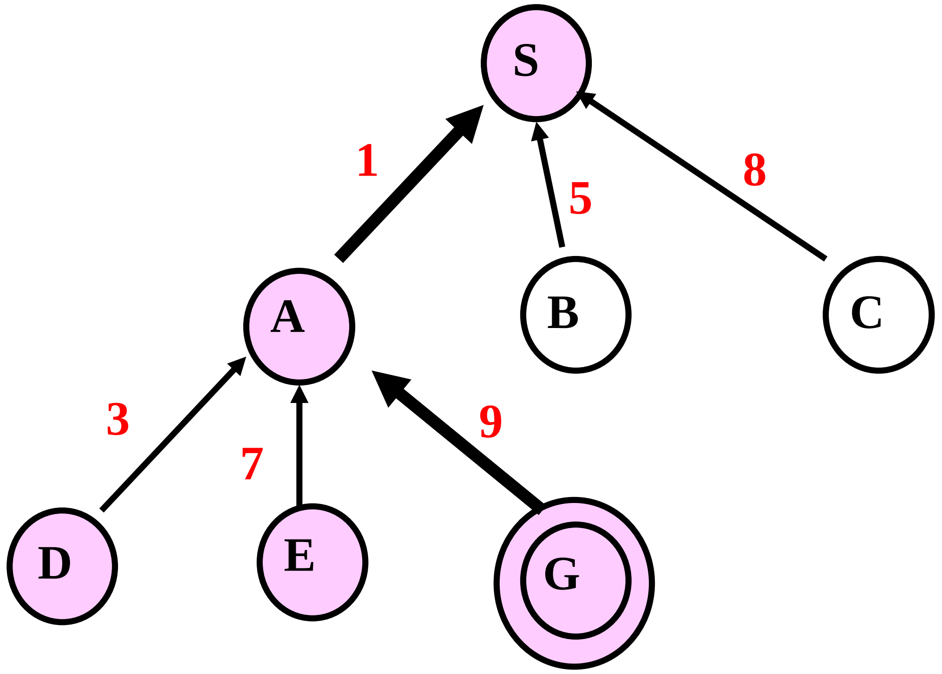
Depth-First Search (DFS)

- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D (How to determine the depth bound?)
- **Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, O(b^d), but only **linear space**, O(bd), required
- Can find **deep solutions quickly** if lucky
- When search hits a dead-end, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "chronological backtracking"

Depth-First Search (DFS)

return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)

exp. node	**OPEN list**	**CLOSED list**
	{ S }	
S	{ A B C }	
A	{ D E G B C}	
D	{ E G B C }	
E	{ G B C }	
G	{ B C }	



Solution path found is S A G <-- this G has cost 10

Number of nodes expanded (including goal node) = 5

Total Search cost = 20

Depth-First Iterative Deepening (DFID)

- BF and DF both have exponential time complexity O(b^d)

  BF is **complete** but has exponential space complexity

  DF has **linear space complexity** but is incomplete

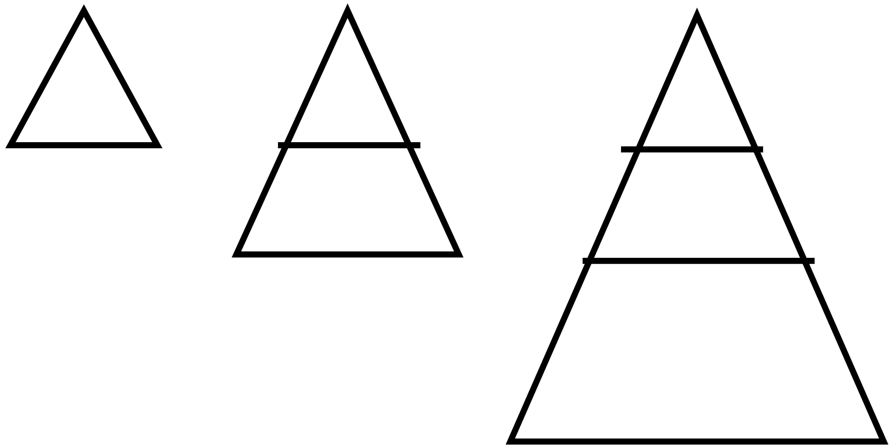
- Space is often a **harder** resource constraint than time
- Can we have an algorithm that
  - Is complete
  - Has linear space complexity, and
  - Has time complexity of O(b^d)

DFID by Korf in 1985.

Depth-First Iterative Deepening (DFID)

DFID Algorithm

It involves repeatedly carrying out DFS on the tree, starting with a DFS limited to depth of one, then DFS of depth two, and so on, until a goal is found.



Depth-First Iterative Deepening (DFID)

- **Complete** (iteratively generate all nodes up to depth ‘d’)
- **Optimal/Admissible** if all operators have the same cost.

  Otherwise, not optimal but does guarantee finding solution of shortest length (like BF).

Depth-First Iterative Deepening (DFID)

- **Linear space complexity:** O(bd), (like DF)
- **Time complexity** is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, O(b^d).

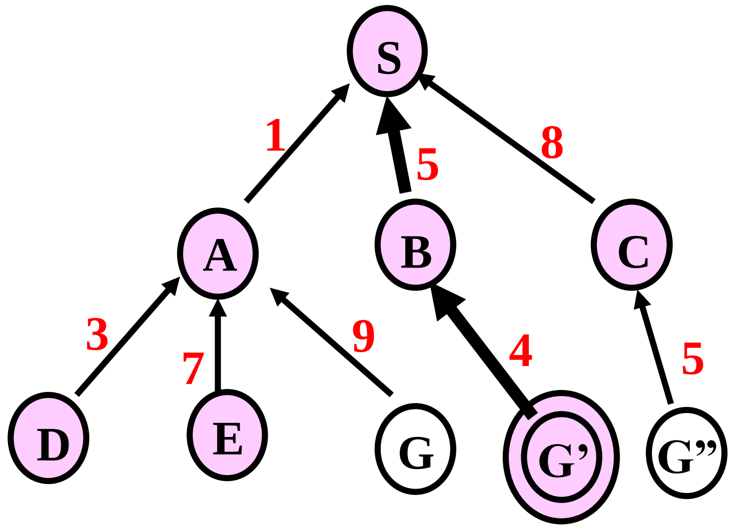
Uniform-Cost Search(UCS)

- Let g(n) = cost of the path from the start node to an open node n
- Algorithm outline:
  - Always select from the OPEN the node with the least g(.) value for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(.) values (in ascending order)
  - Terminate if a node selected for expansion is a goal

Uniform-Cost Search(UCS)

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

Exp. node	**Nodes list**	**CLOSED list**
	{S(0)}	
S	{A(1) B(5) C(8)}	
A	{D(4) B(5) C(8) E(8) G(10)}	
D	{B(5) C(8) E(8) G(10)}	
B	{C(8) E(8) G'(9) G(10)}	
C	{E(8) G'(9) G(10) G''(13)}	
E	{G'(9) G(10) G''(13) }	
G'	{G(10) G''(13) }	



Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

Uniform-Cost Search(UCS)

- **Complete** (if cost of each action is not infinitesimal)
  - The total # of nodes n with g(n) <= g(goal) in the state space is finite
  - If n' is a child of n, then g(n') = g(n) + c(n, n') > g(n)
  - Goal node will eventually be generated (put in OPEN) and selected for expansion (and passes the goal test)
- **Exponential time and space complexity**,
  - worst case: becomes BFS when all arcs cost the same

Uniform-Cost Search(UCS)

- **Optimal/Admissible**
  - Admissibility depends on the goal test being applied when a node is removed from the OPEN list, not when it's parent node is expanded and the node is first generated (delayed goal testing)
  - Multiple solution paths (following different back pointers)
  - Each solution path that can be generated from an open node n will have its path cost
  - >= g(n)
  - When the first goal node is selected for expansion (and passes the goal test), its path cost is less than or equal to g(n) of every OPEN node n (and solutions entailed by n)

When to use what

- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Breadth-First Search**:
  - Some solutions are known to be shallow
- **Uniform-Cost Search**:
  - Actions have varying costs
  - Least cost solution is required

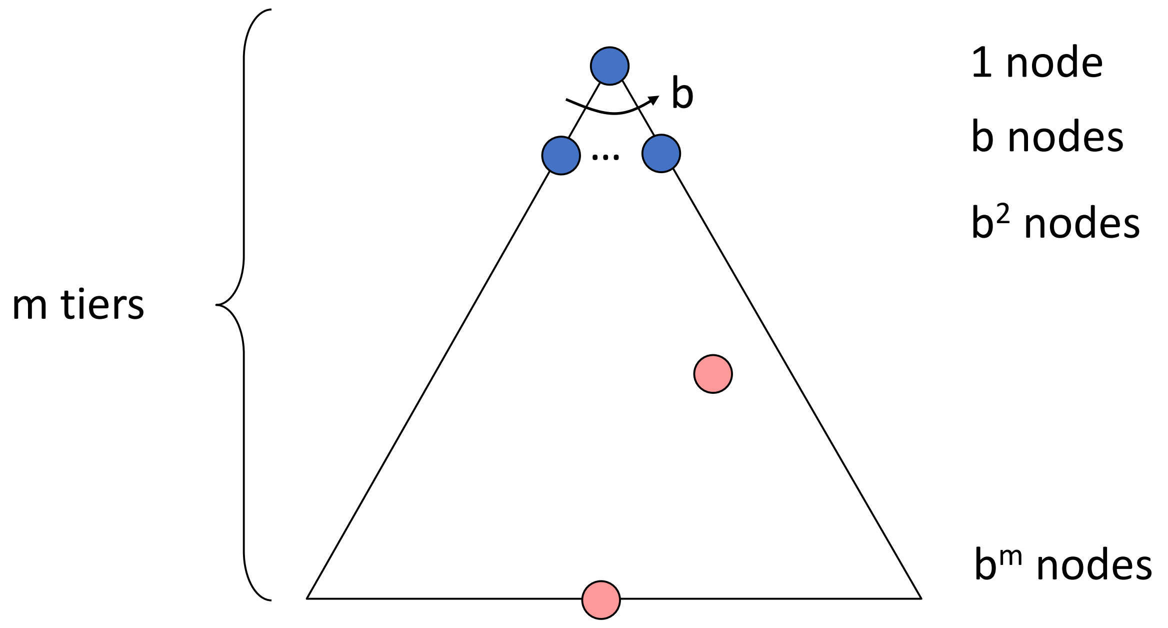
This is the only uninformed search that worries about costs.

Search Algorithm Properties (Review)

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
  - Computational complexity that describes the amount of time it takes to run an algorithm
- Space complexity?
  - Amount of working storage an algorithm needs

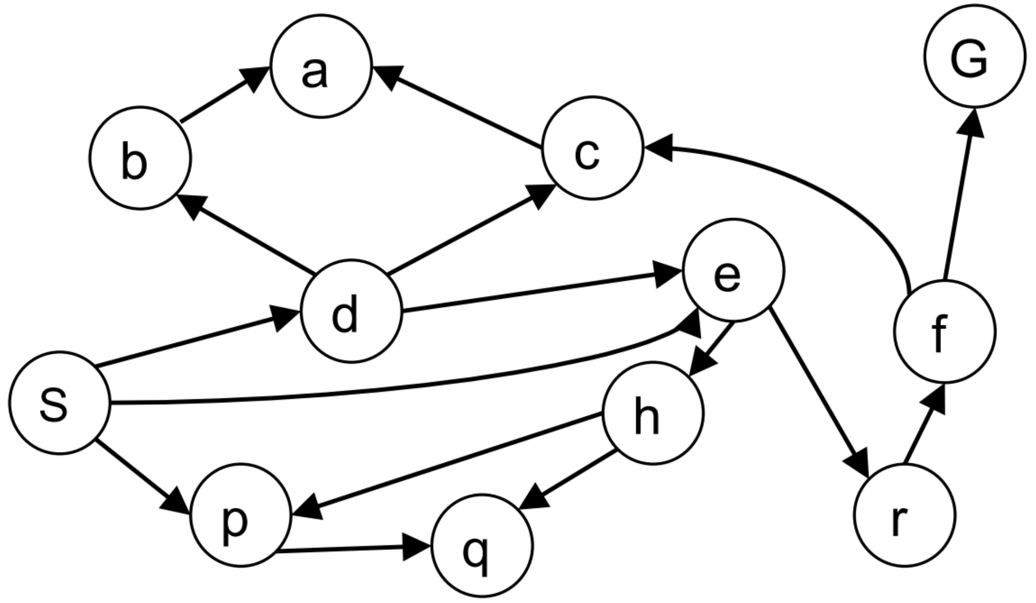
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \dots. b^m = O(b^m)$



Breadth First Search (Review)

- Strategy: expand shallowest (of little depth) node first

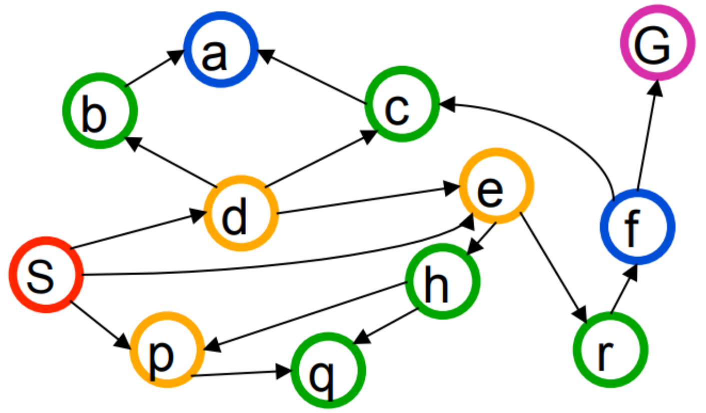


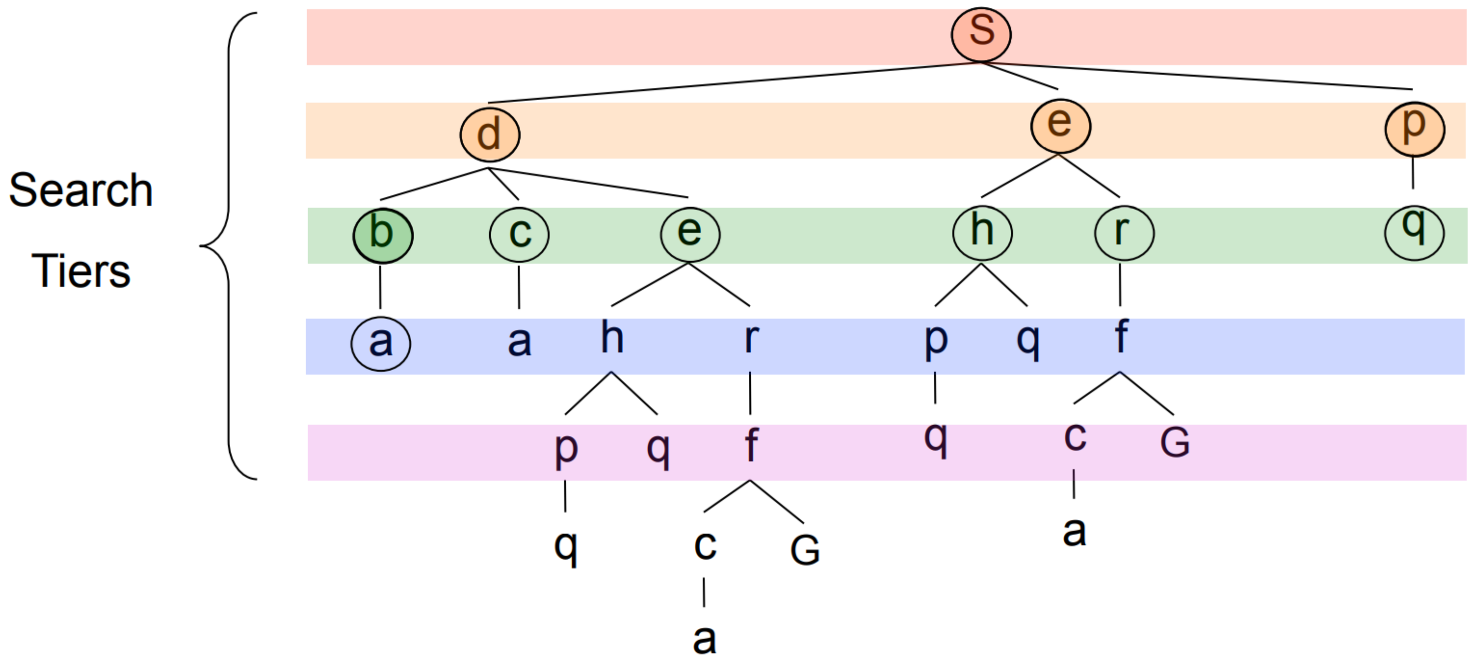
- Implementation: Fringe is a FIFO queue

Breadth First Search (Review)

- Expansion order:

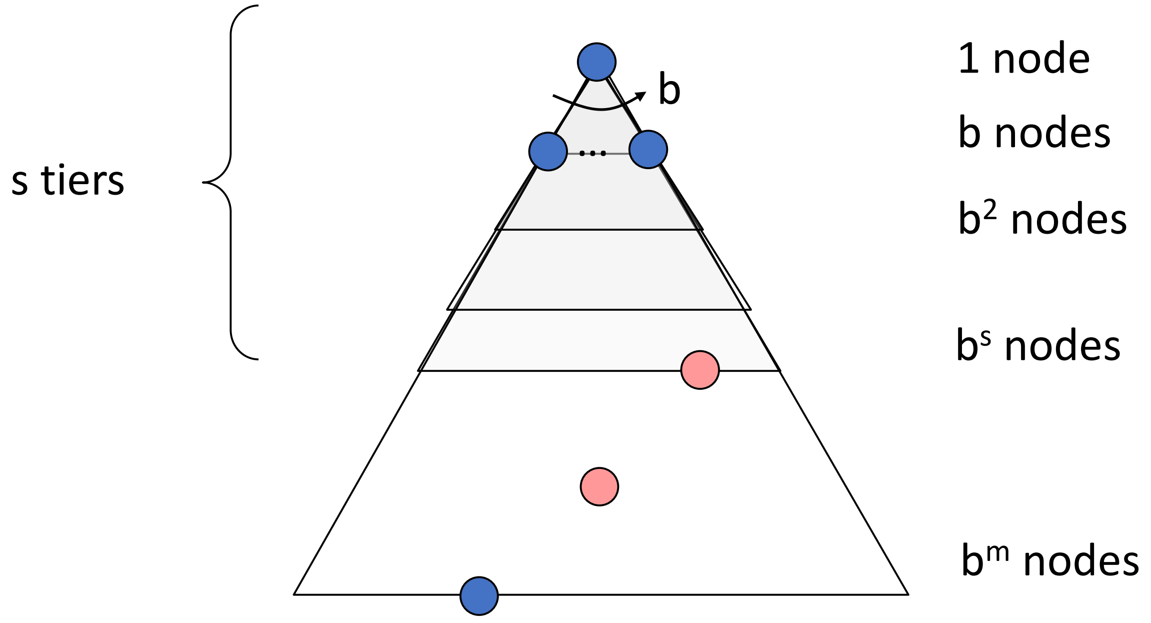
  (S,d,e,p,b,c,e,h,r,q,a,a ,h,r,p,q,f,p,q,f,q,c,G)





Breadth-First Search (BFS) Properties (Review)

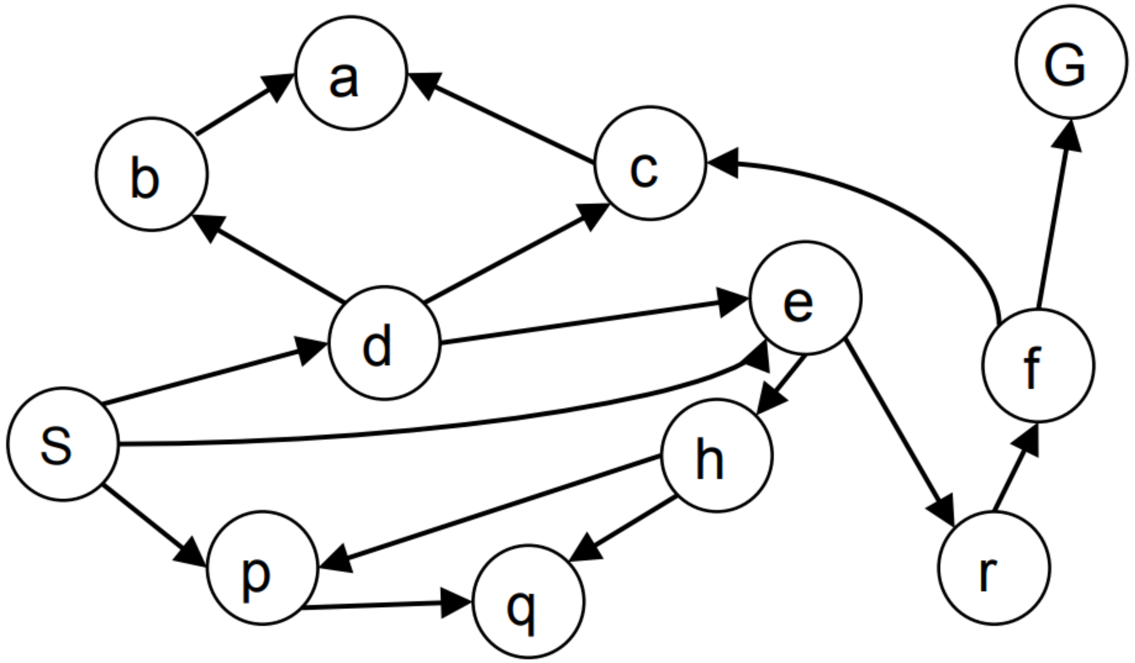
- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



Depth First Search (Review)

- Strategy: expand deepest node first

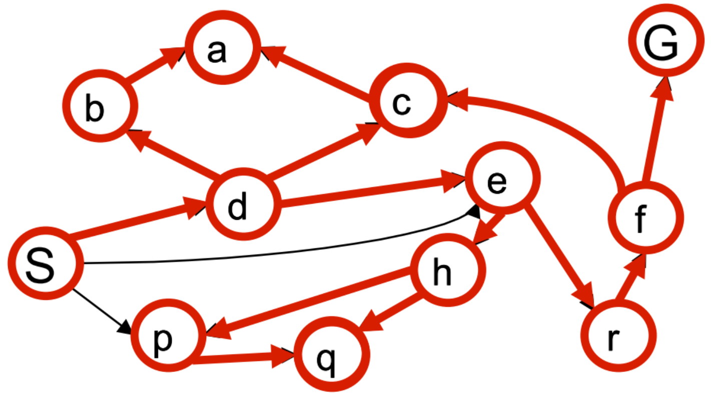
- Implementation: Fringe is a LIFO queue (a stack)

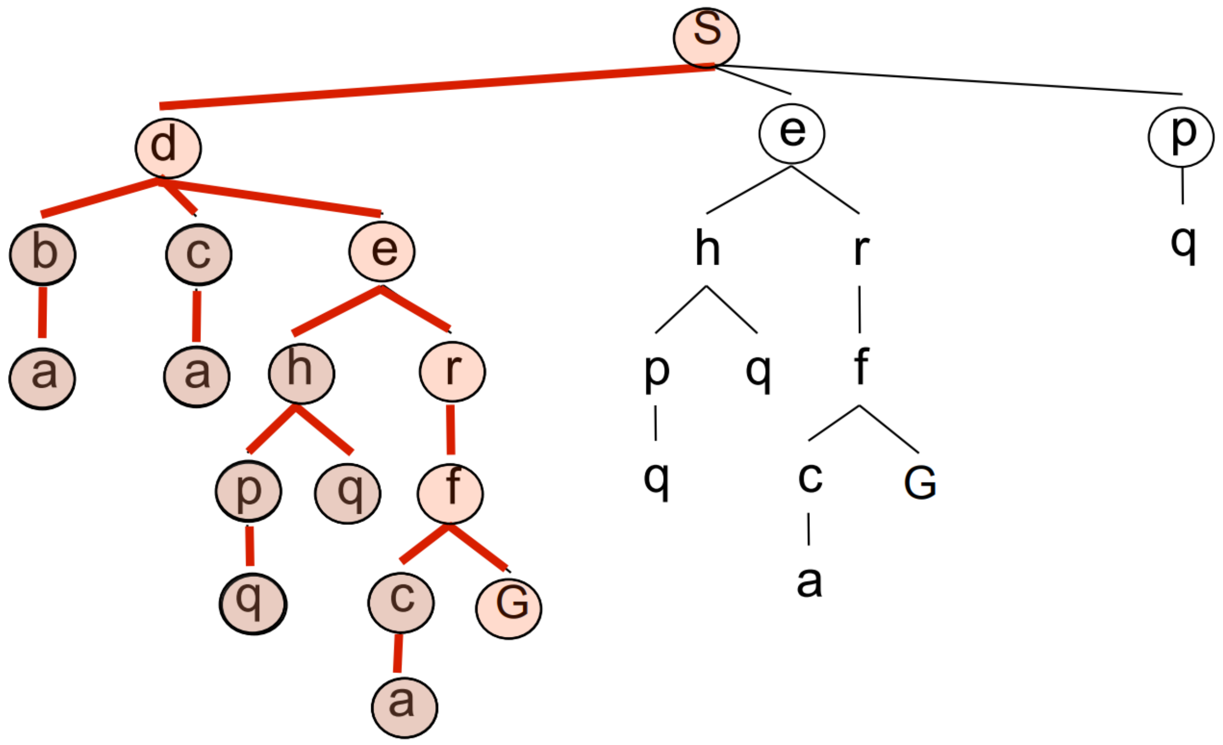


Depth First Search (Review)

- Expansion ordering:

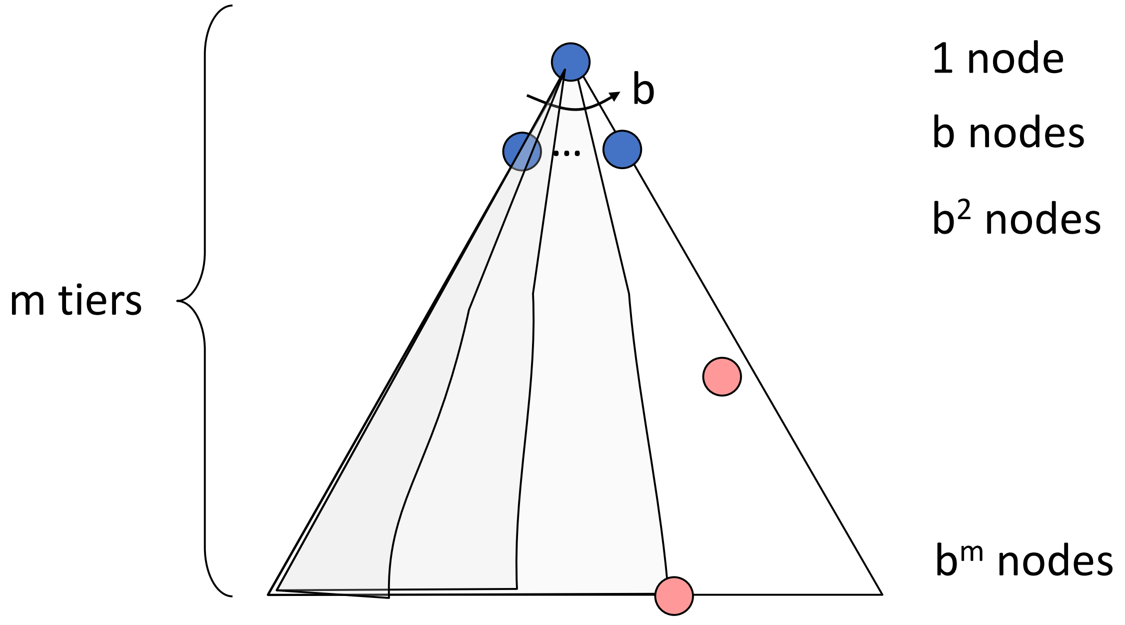
(d,b,a,c,a,e,h,p,q,q,r,f,c,a,G)



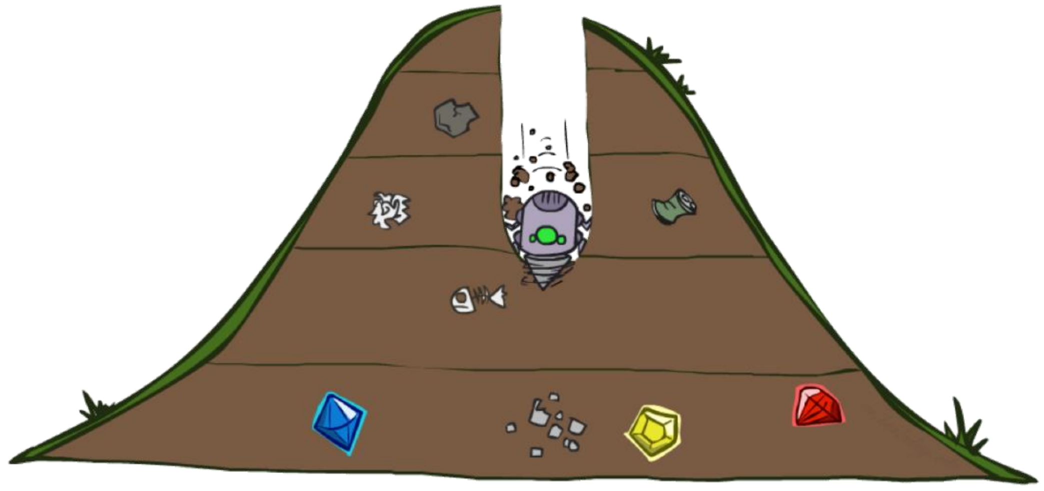


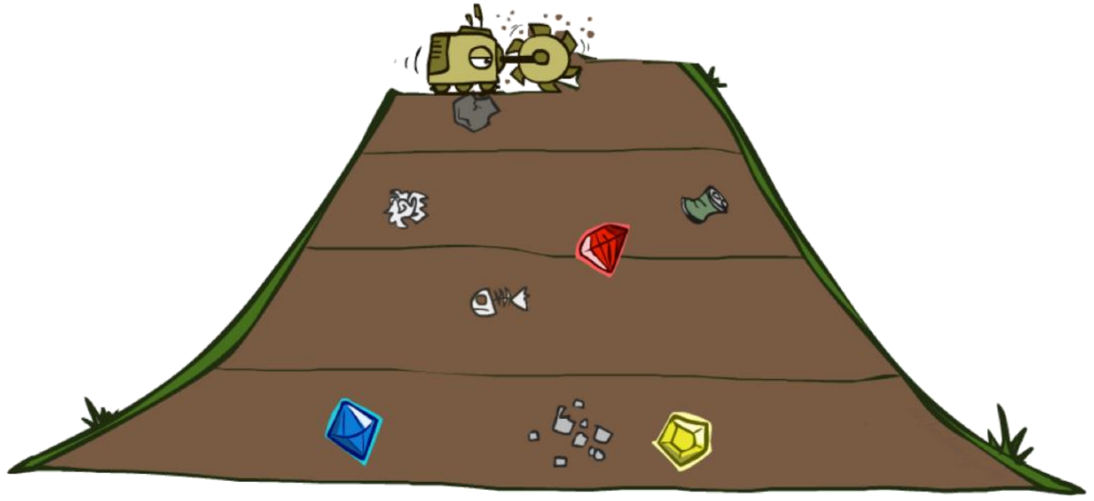
Depth-First Search (DFS) Properties (Review)

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost



BFS Vs. DFS (Review)

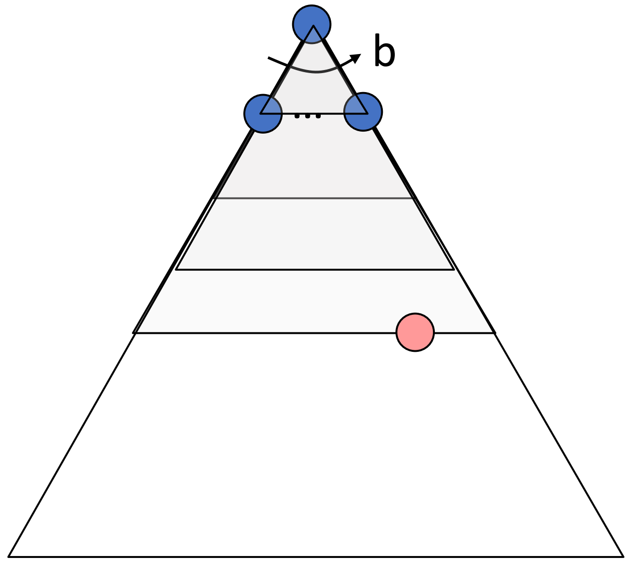




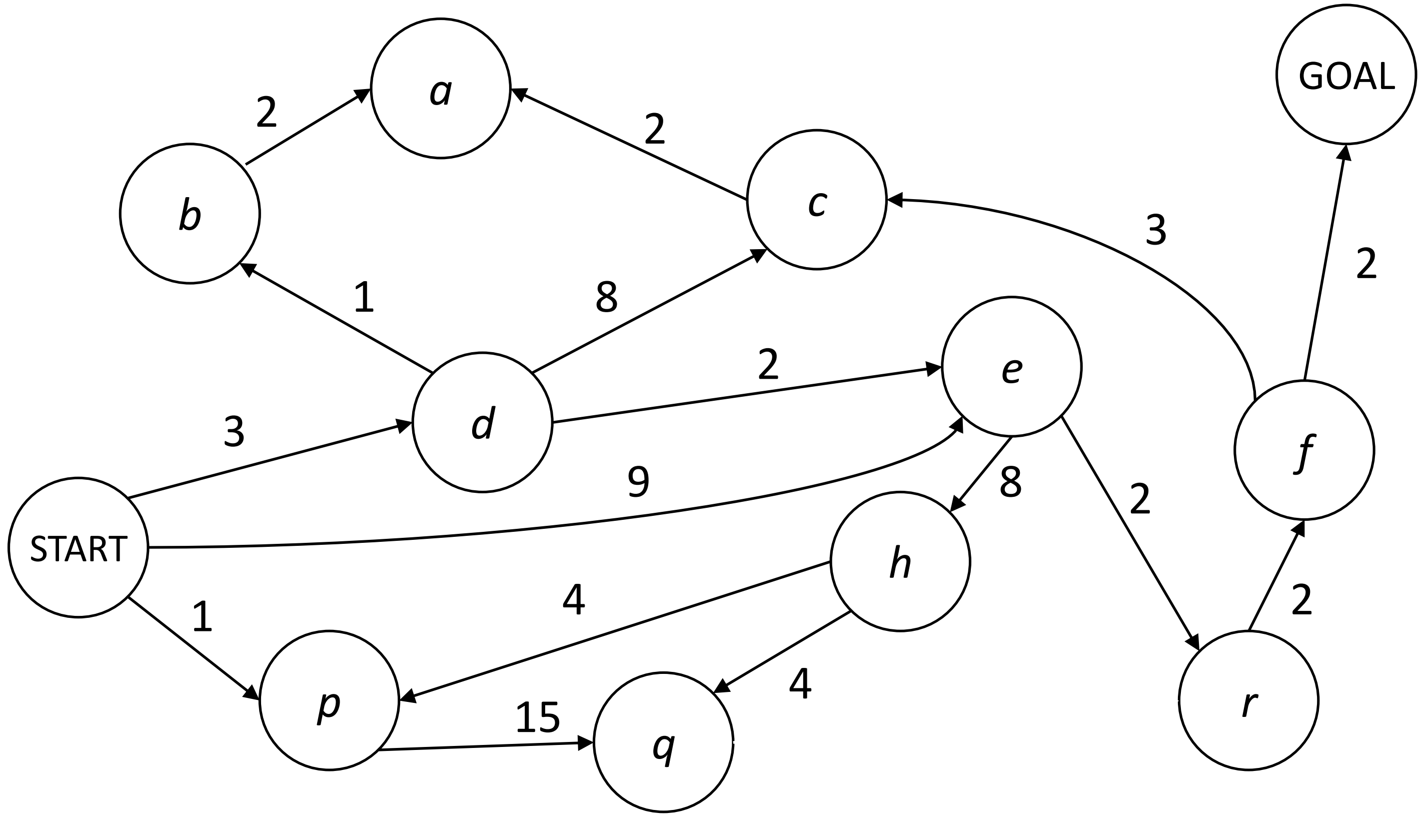
Iterative Deepening

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..

- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!

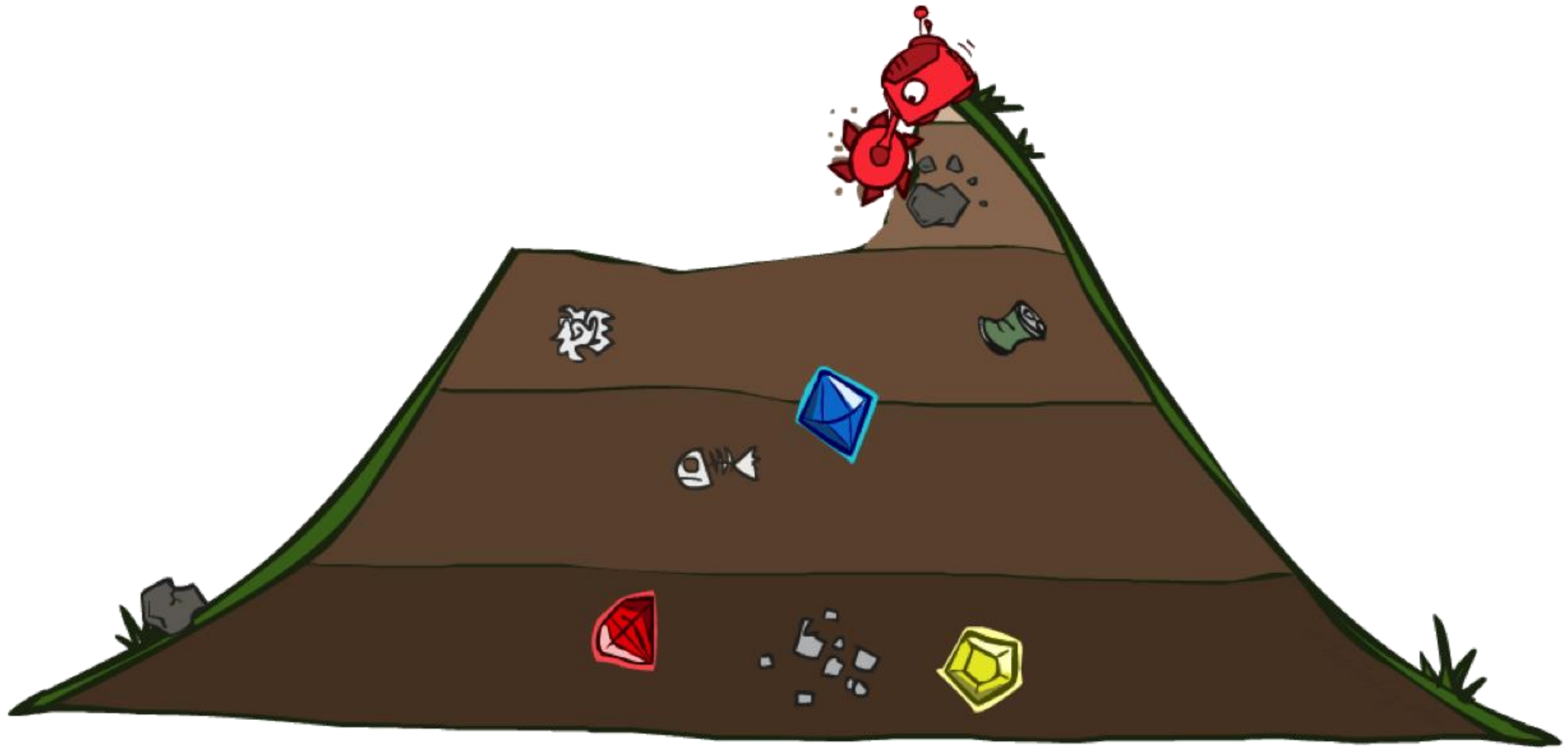


Cost-Sensitive Search



BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.

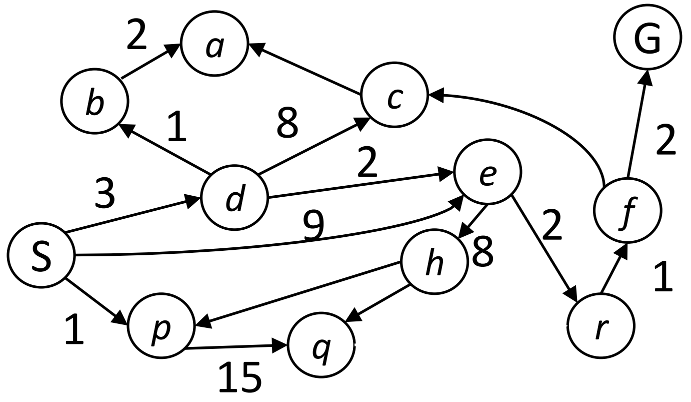
Uniform Cost Search



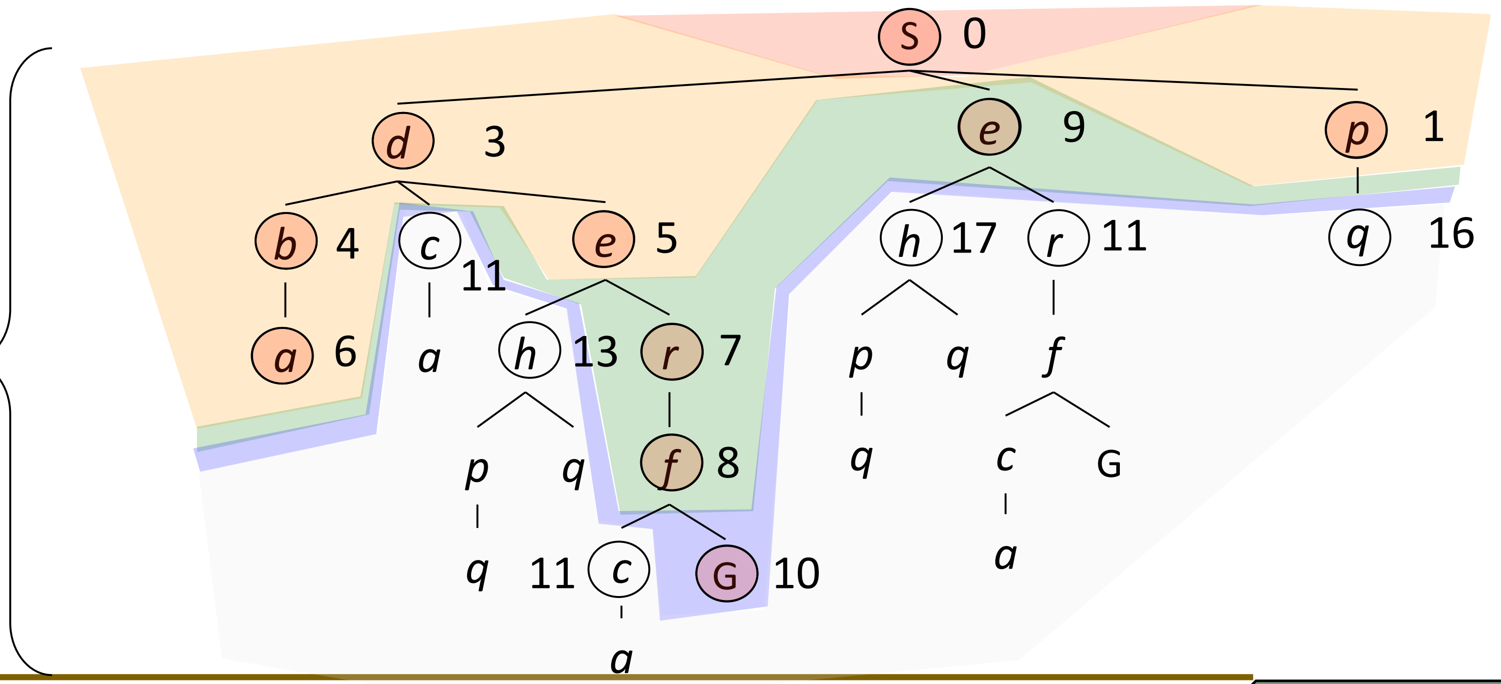
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

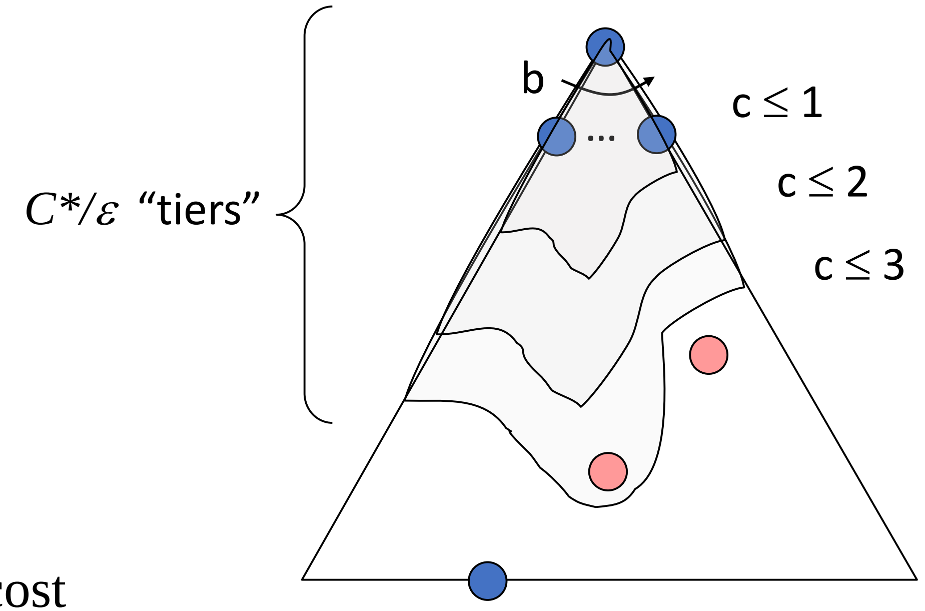


Cost contours (Outline)



Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes!

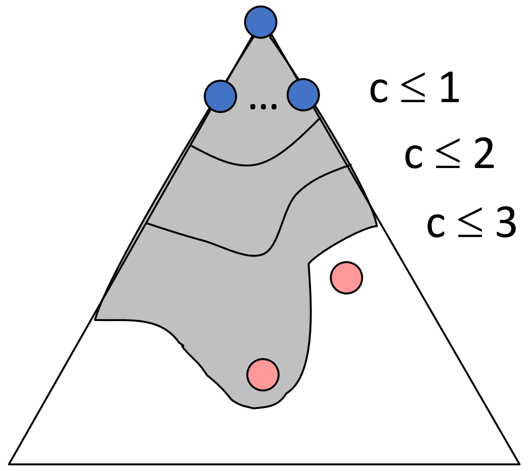


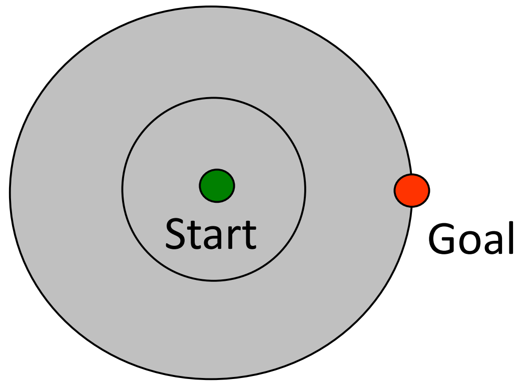
Uniform Cost Issues

- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal!

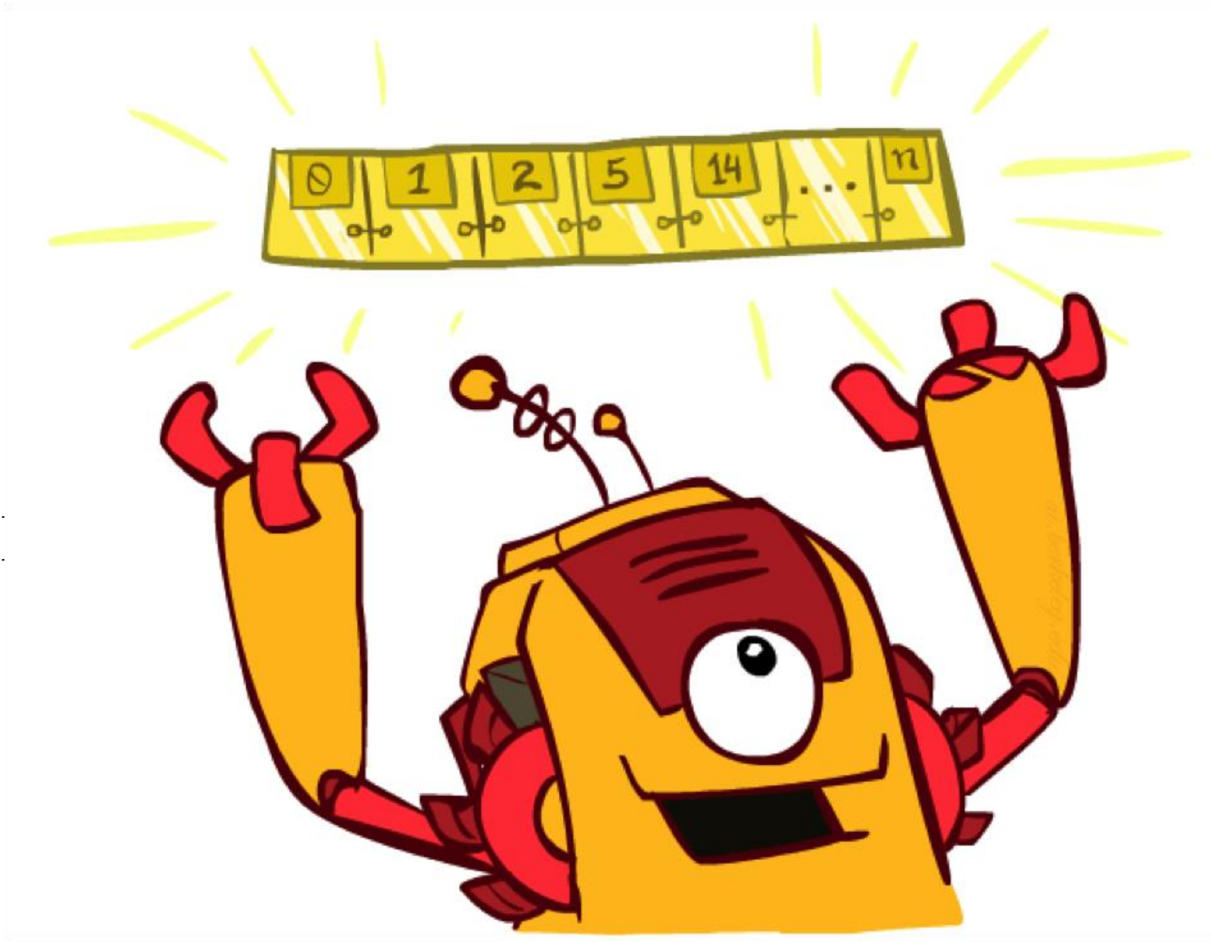
- The bad:
  - Explores options in every "direction"
  - No information about goal location





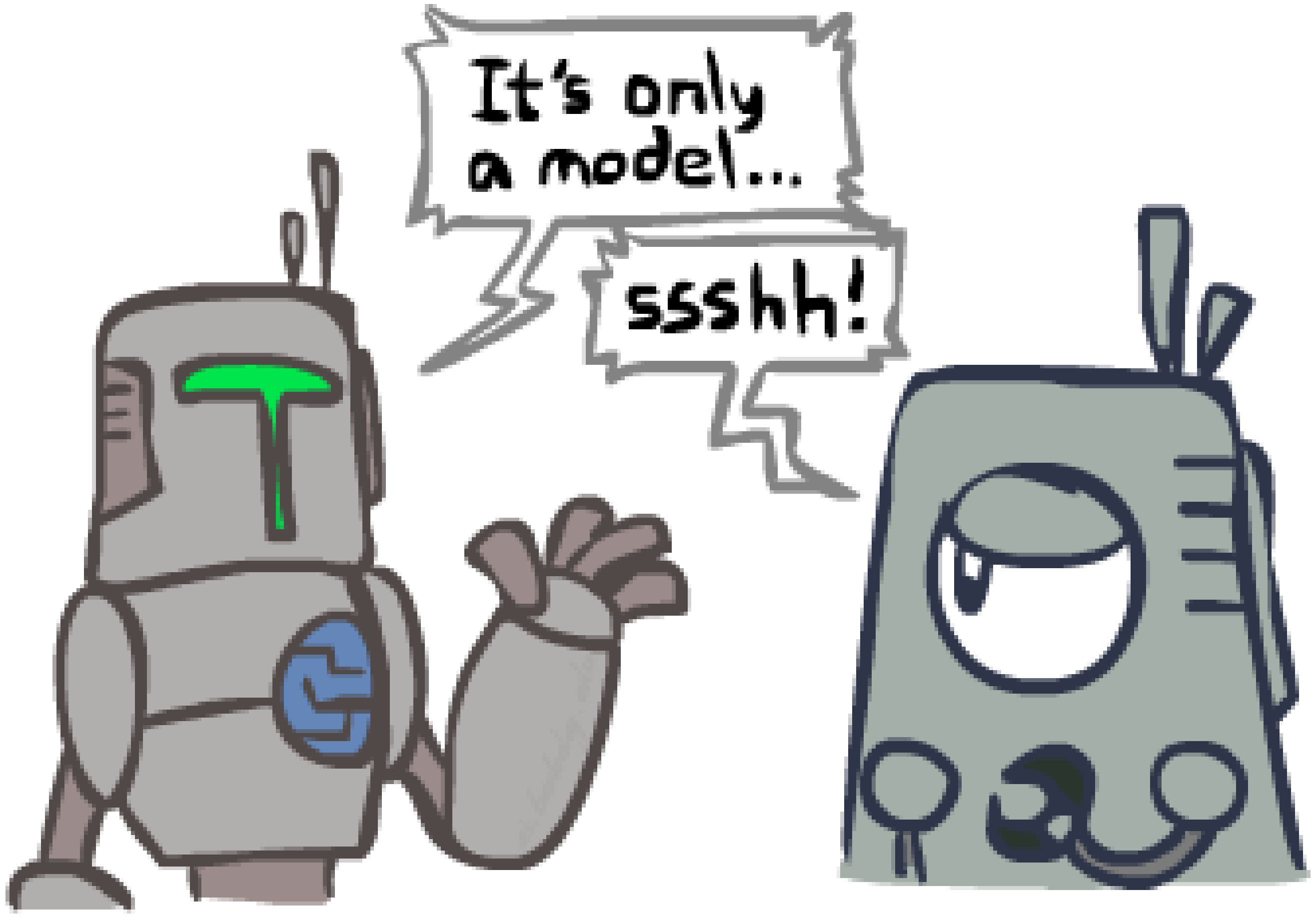
The One Queue

- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actua priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object

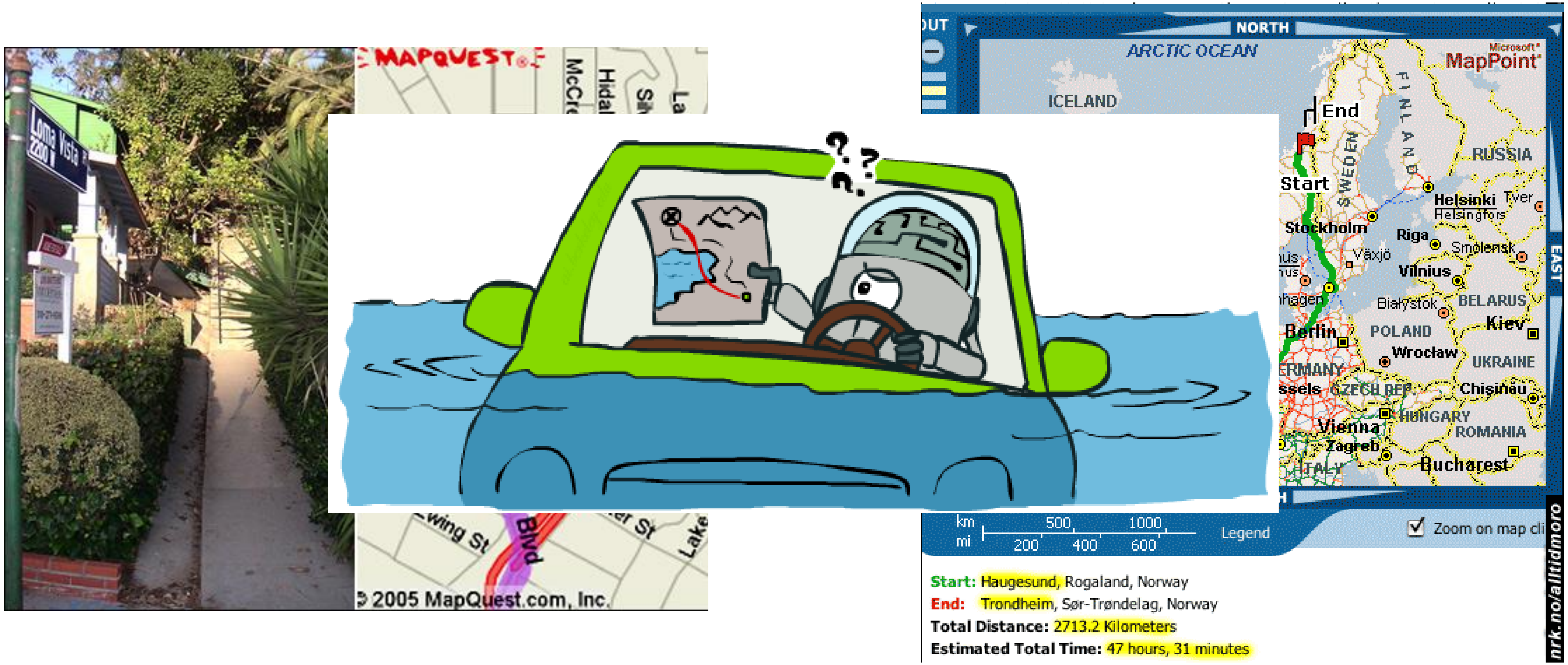


Search and Models

- Search operates over models of the world
    - The agent doesn't actually try all the plans out in the real world!
    - Planning is all "in simulation"
    - Your search is only as good as your models…

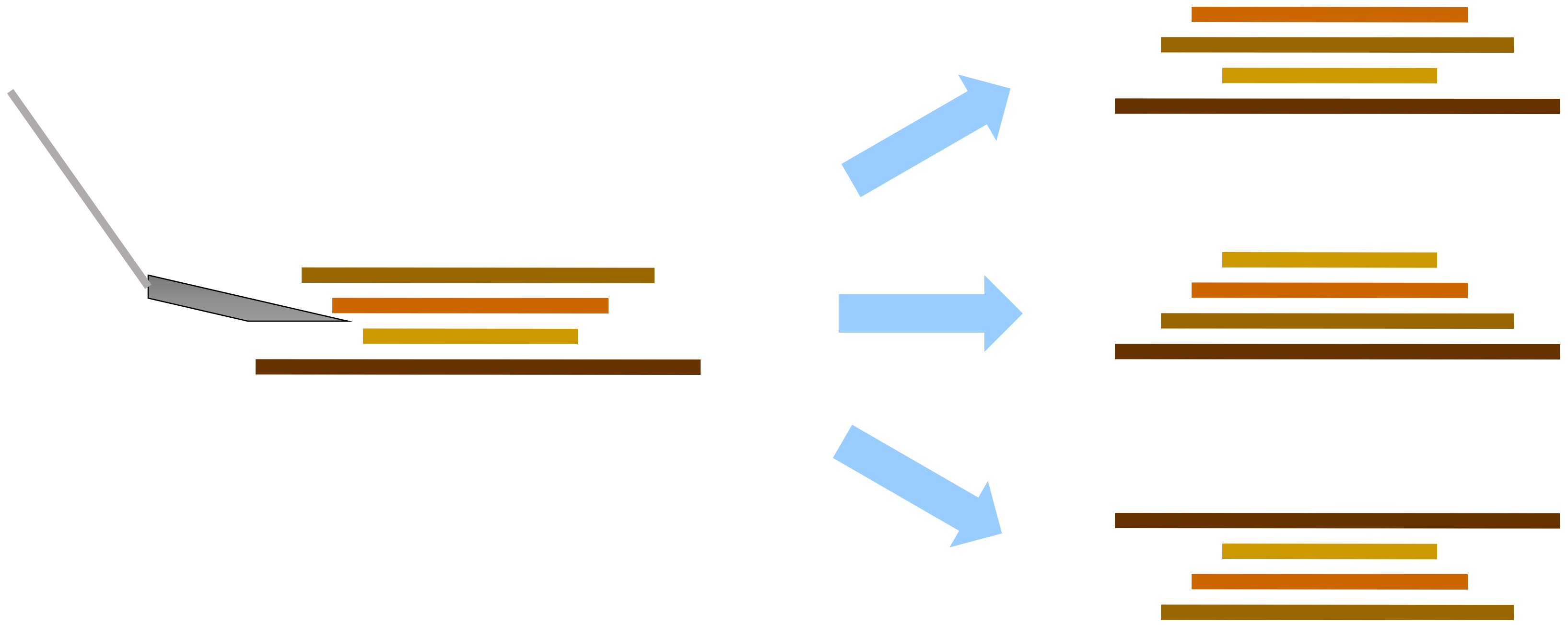


Search Gone Wrong?



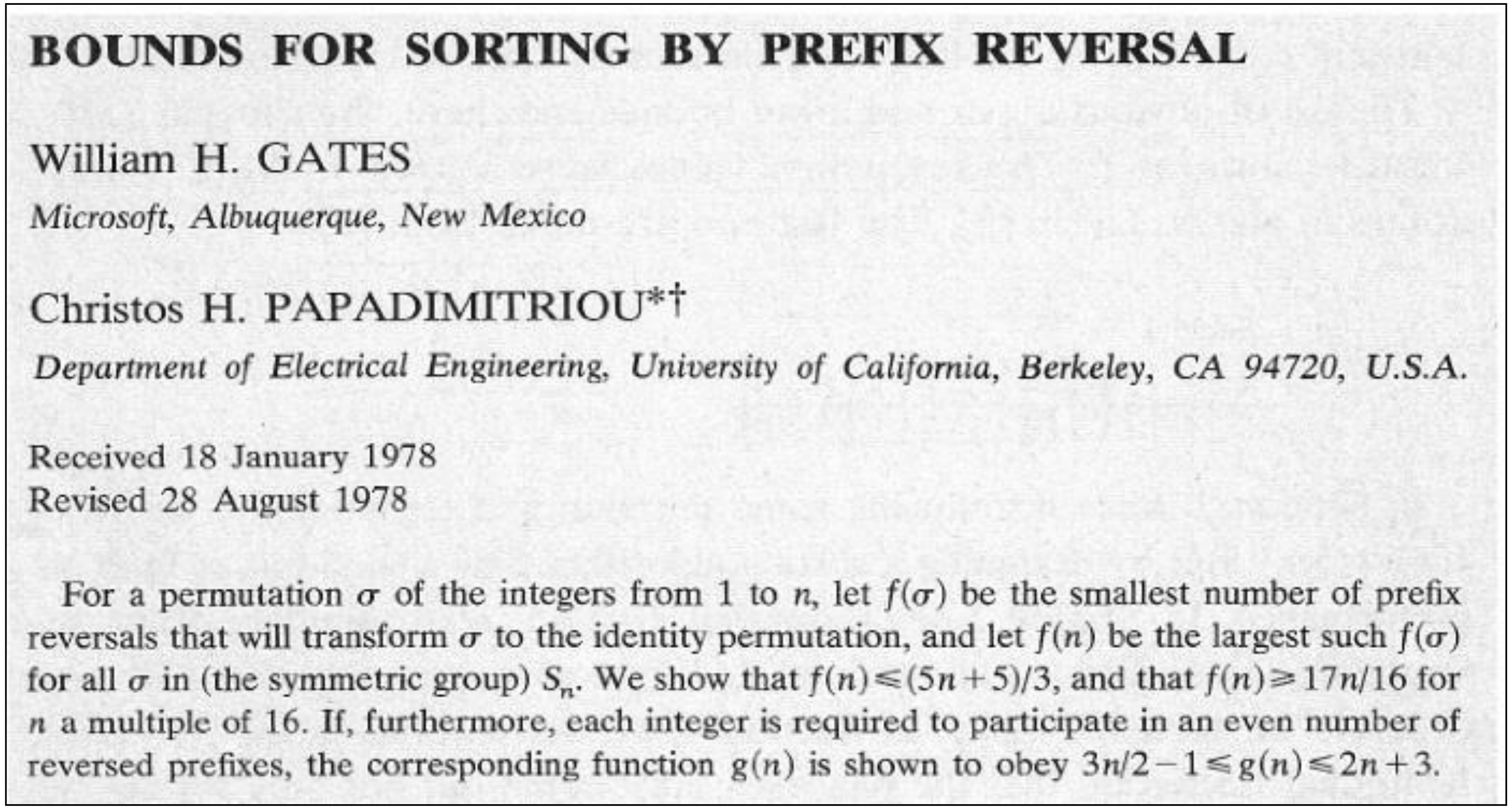
Start: Haugesund, Rogaland, Norway
End: Trondheim, Sør-Trøndelag, Norway
Total Distance: 2713.2 Kilometers
Estimated Total Time: 47 hours, 31 minutes

Example: Pancake Problem



Cost: Number of pancakes flipped

Example: Pancake Problem



BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES

Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†

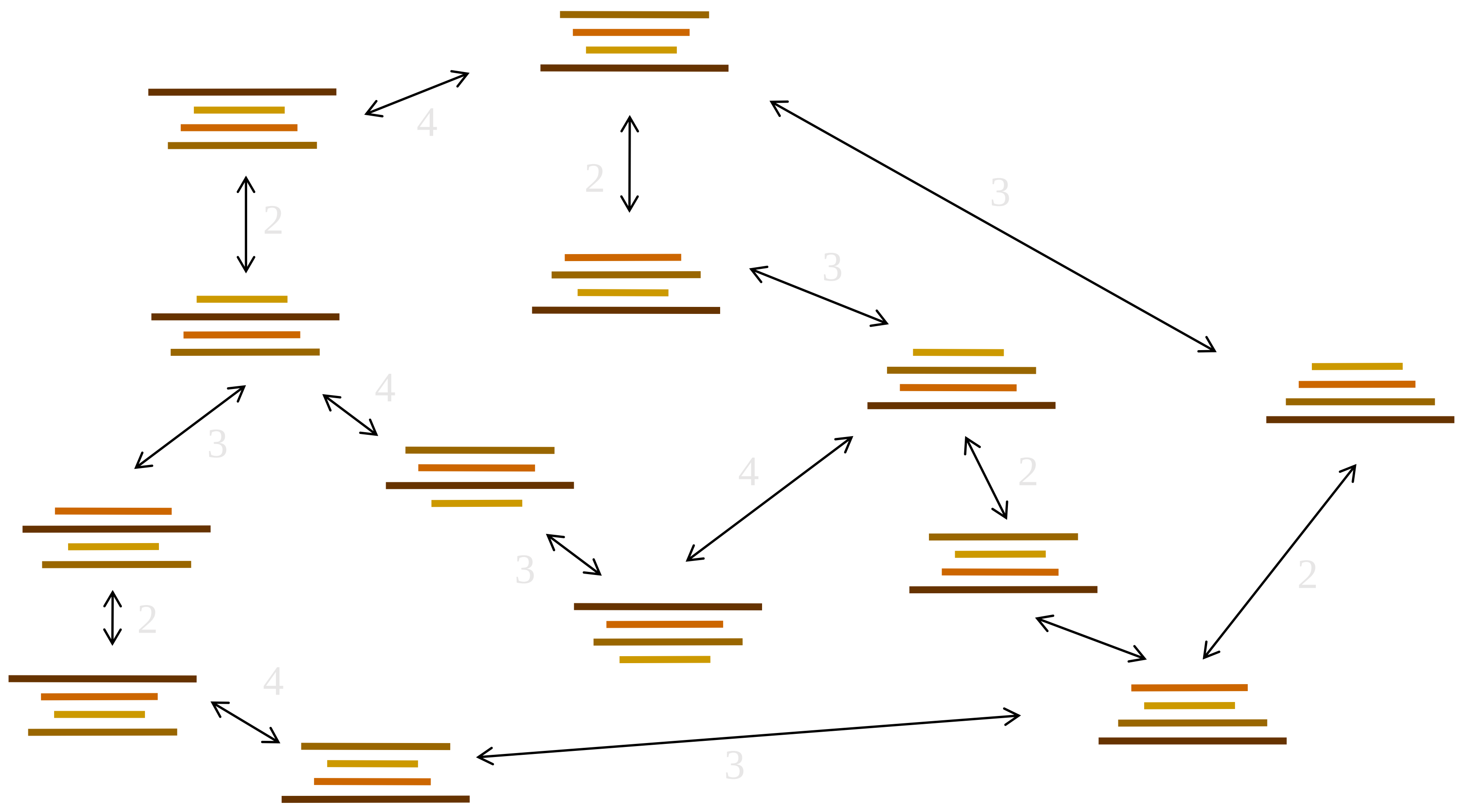
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leq (5n+5)/3$, and that $f(n) \geq 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leq g(n) \leq 2n+3$.

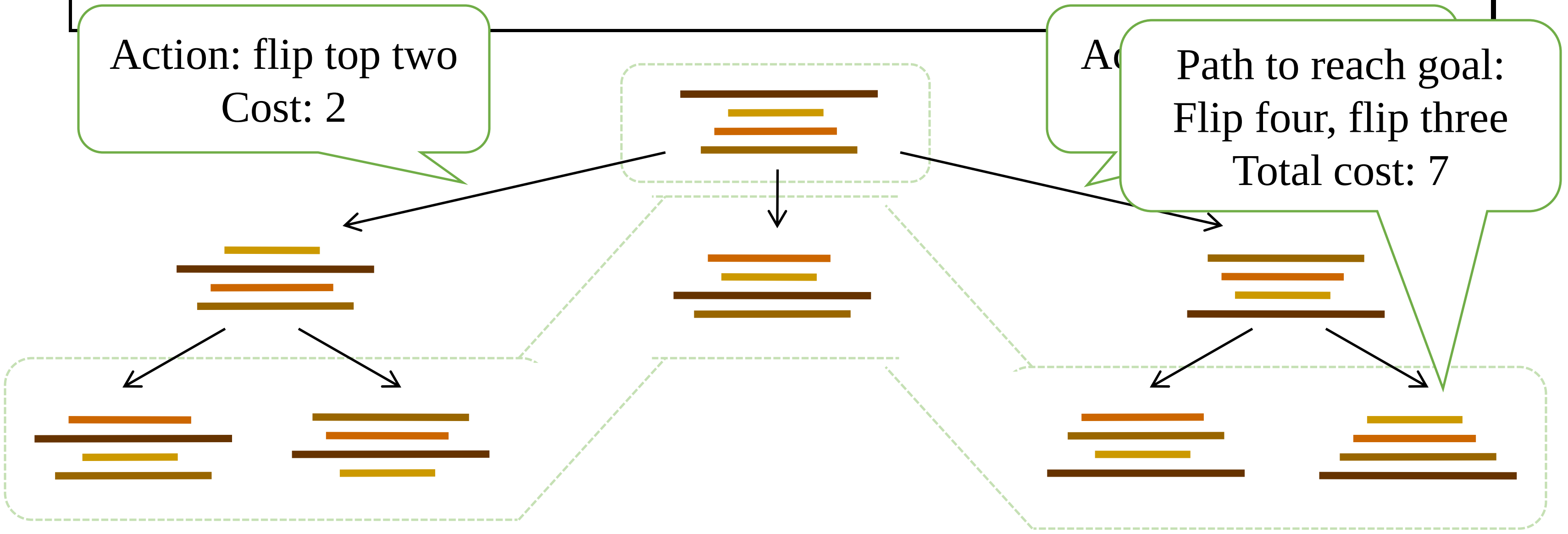
Example: Pancake Problem

State space graph with costs as weights



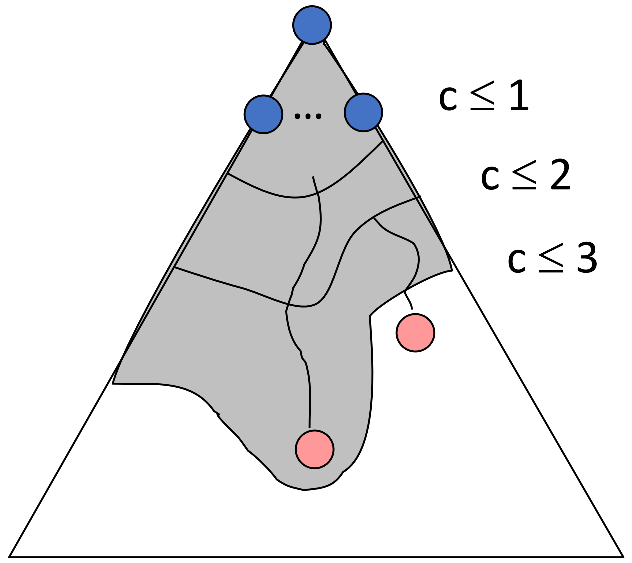
General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
   end
```



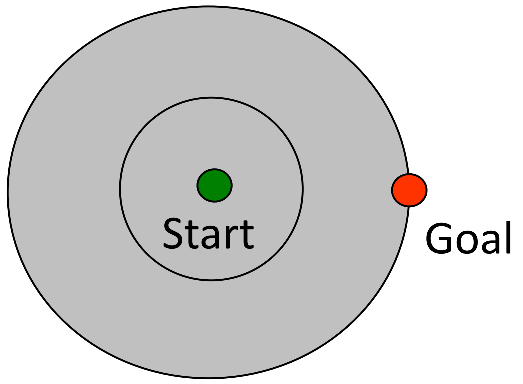
Uniform Cost Search

- Strategy: expand lowest path cost

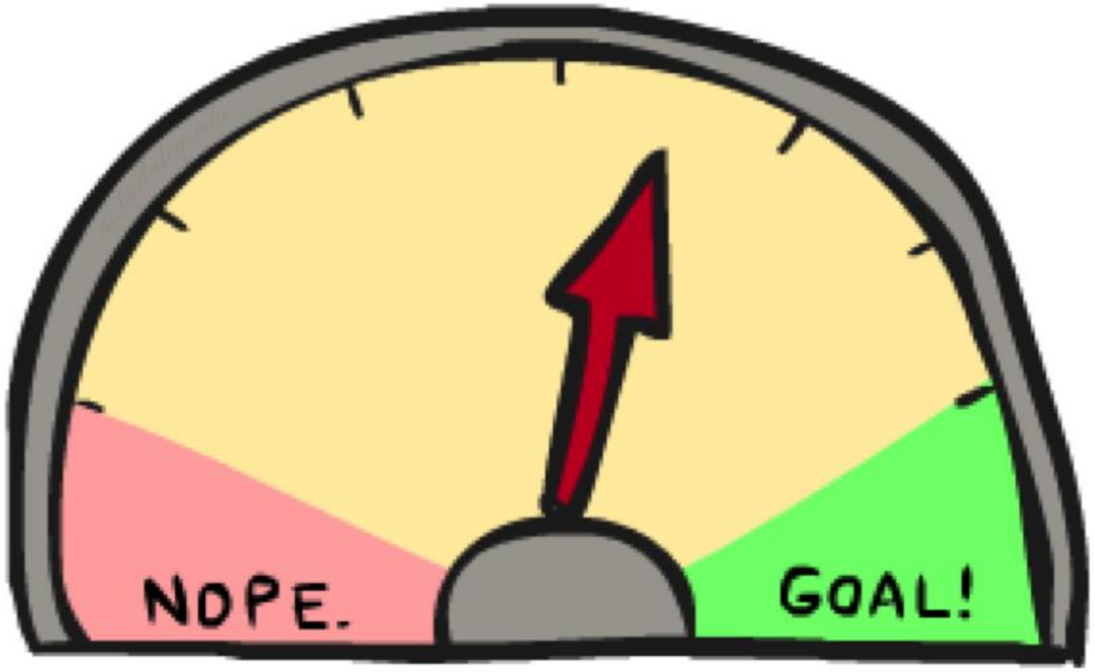


- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every "direction"
  - No information about goal location



Informed Search



Heuristic

- Origin: from Greek Word ‘heuriskein’, means, “to discover”.
- Webster’s New World Dictionary defines Heuristic as “helping to discover or learn”
- As an adjective, means, serving to discover
- As noun, a heuristic is an aid to discovery.

A **heuristic** is a method to help solve a problem, commonly informal. It is particularly used for a method that often rapidly leads to a solution that is usually reasonably close to the best possible answer.

Heuristics are "rules of thumb", educated guesses, intuitive judgments or simply common sense.

A heuristic contributes to the reduction of search in a problem-solving activity.

Heuristic Search

- Uses domain-dependent (heuristic) information in order to search the space more efficiently.

<u>Ways of using heuristic information:</u>

- Deciding which node to expand next, instead of doing the expansion in a strictly single direction.
- In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
- Deciding that certain nodes should be discarded, or *pruned*, from the search space.

Heuristic Search

Use Of heuristics in our everyday lives:

- If the sky is grey we conclude that it would be better to put on a coat before going out.
- We book our holidays in August because that is when the weather is best.

Heuristic Search

- For many real world problems, it is possible to find specific information to guide the search process and thus reduce the amount of computation.

- This specific information is known as Heuristic information, and the search procedures that use it are called Heuristic Search Methods.

- <u>Heuristics are decision rules regarding how a problem should be solved.</u> Heuristics are developed on the basis of solid, rigorous analysis of the problem, and sometimes involved designed experimentation.

Heuristic Function

- It is a function that maps from problem state description to measure of desirability, usually represented as number.

- Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path.

Major Benefits of Heuristics

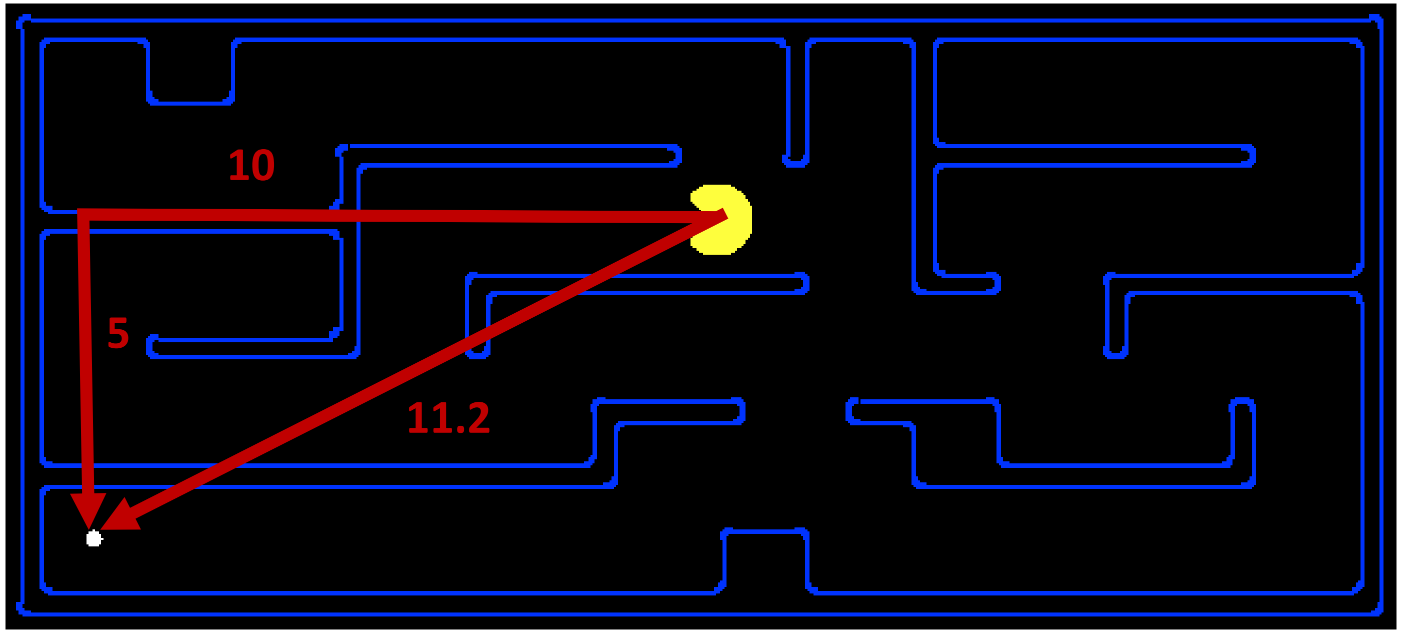
1. Heuristic approaches have an inherent flexibility, which allows them to be used on ill-structured and complex problems.

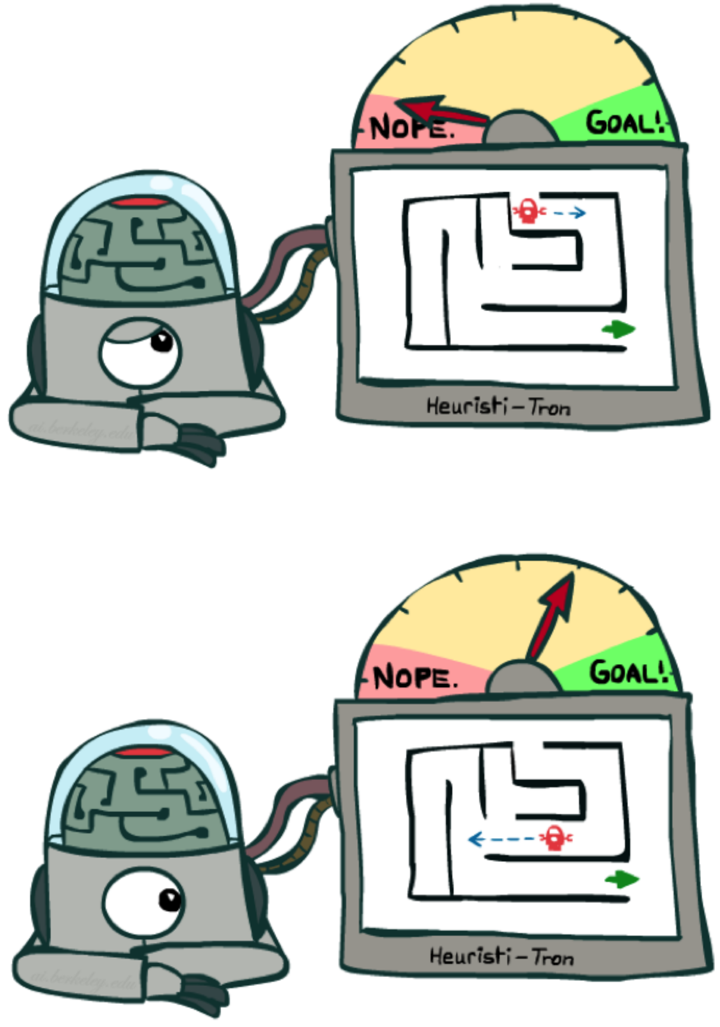
2. These methods by design may be simpler for the decision maker to understand, especially when it is composed of qualitative analysis. Thus chances are much higher for implementing proposed solution.

3. These methods may be used as part of an iterative procedure that guarantees the finding of an optimal solution.

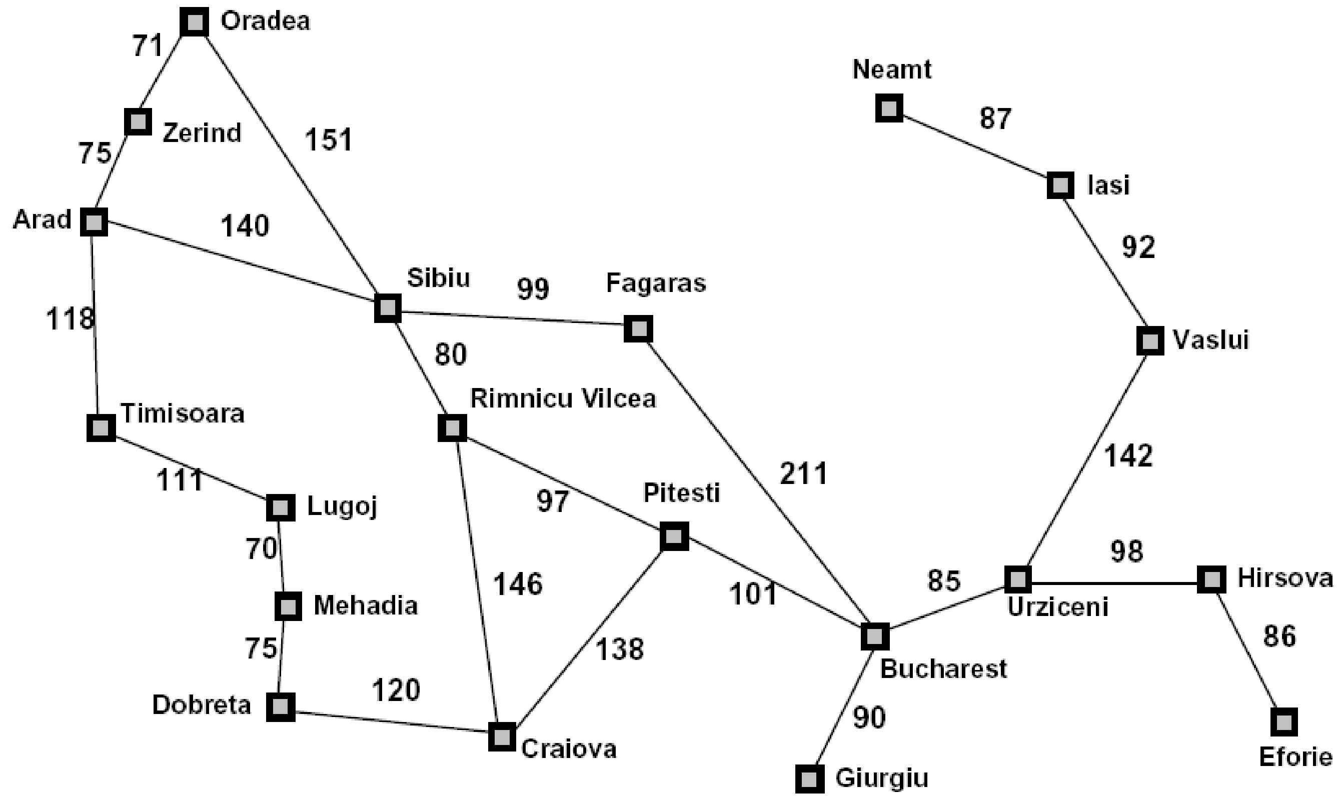
Search Heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing





Example: Heuristic Function



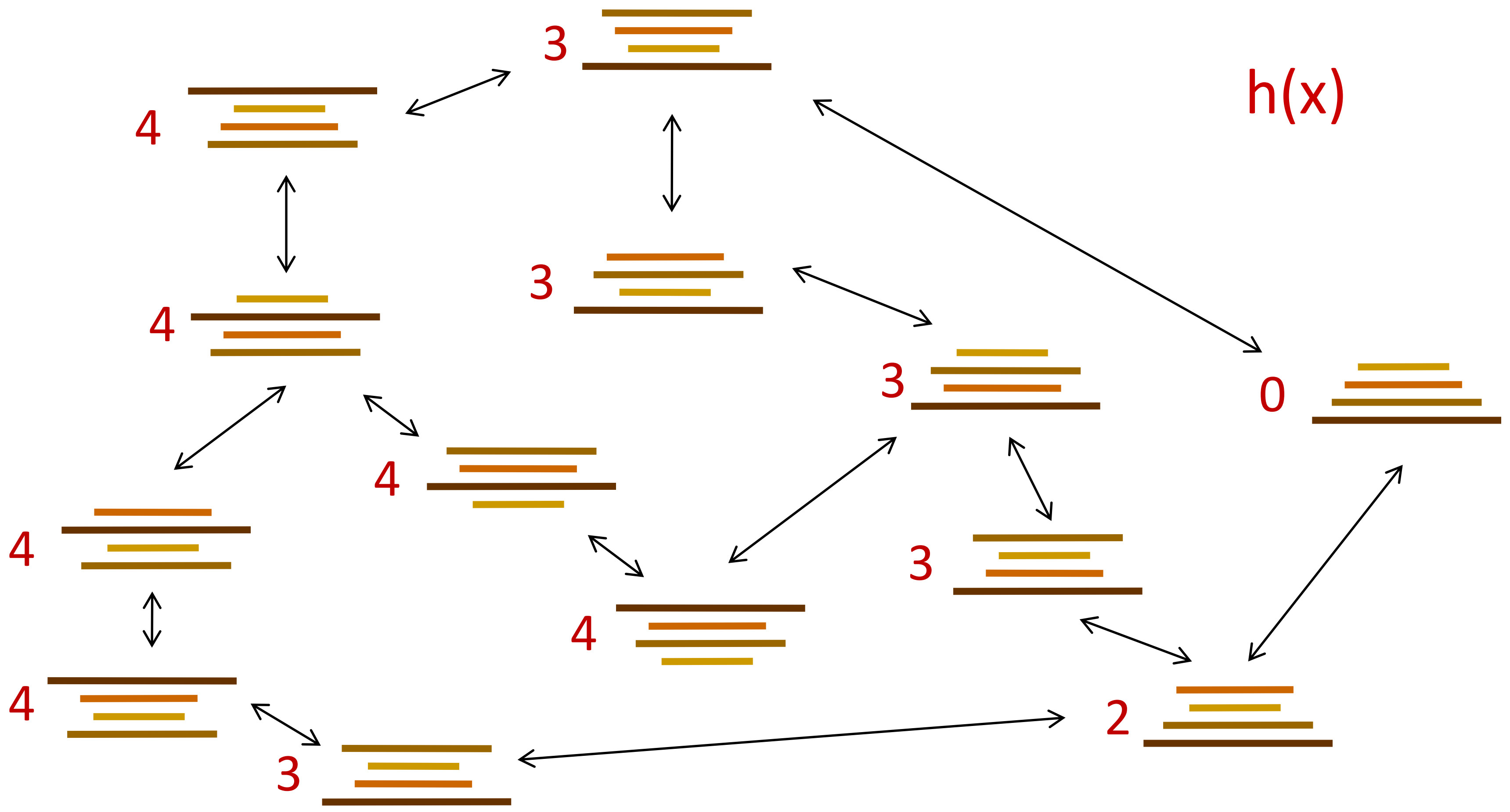
Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

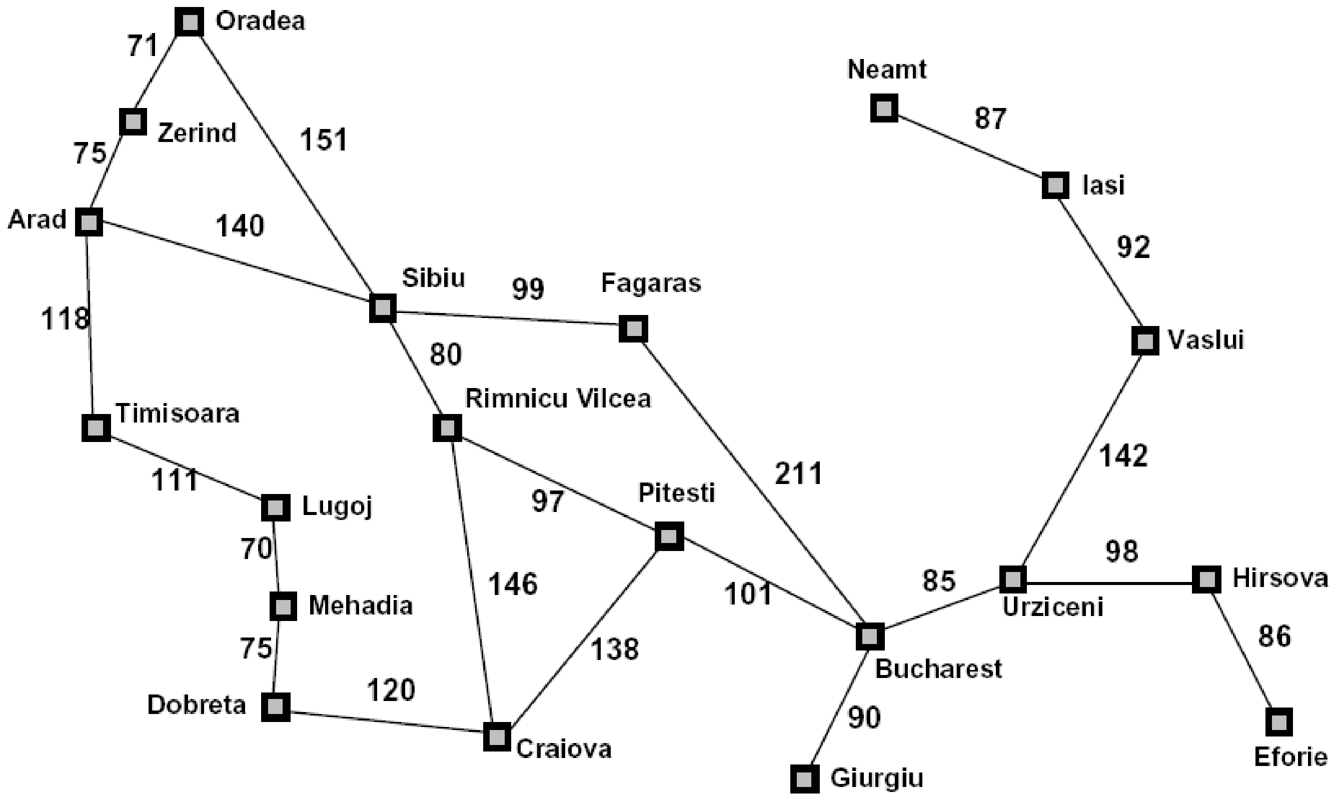
Example: Heuristic Function

Heuristic: the number of the largest pancake that is still out of place



Greedy Search

Example: Heuristic Function



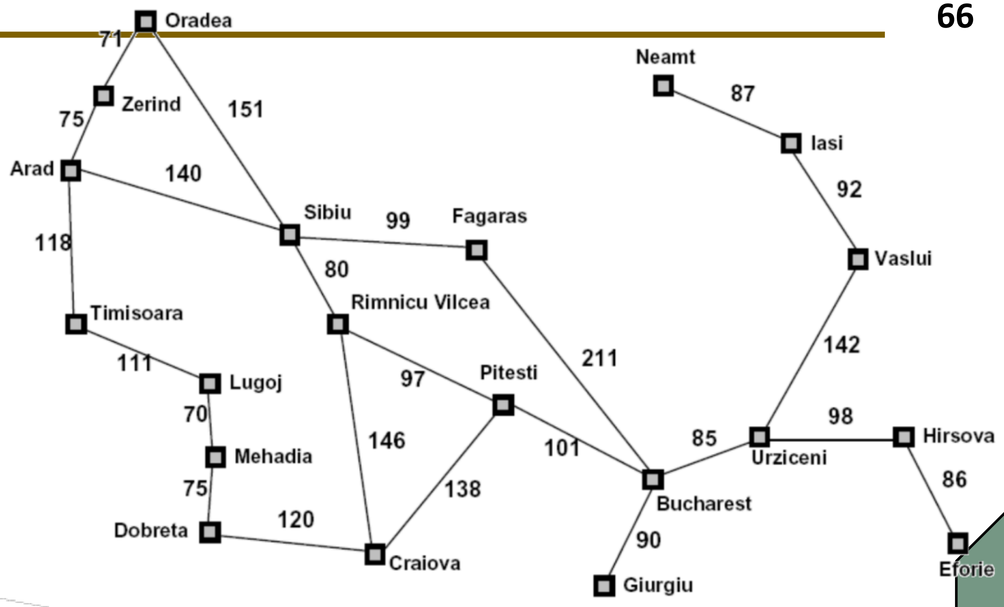
Straight−line distance to Bucharest

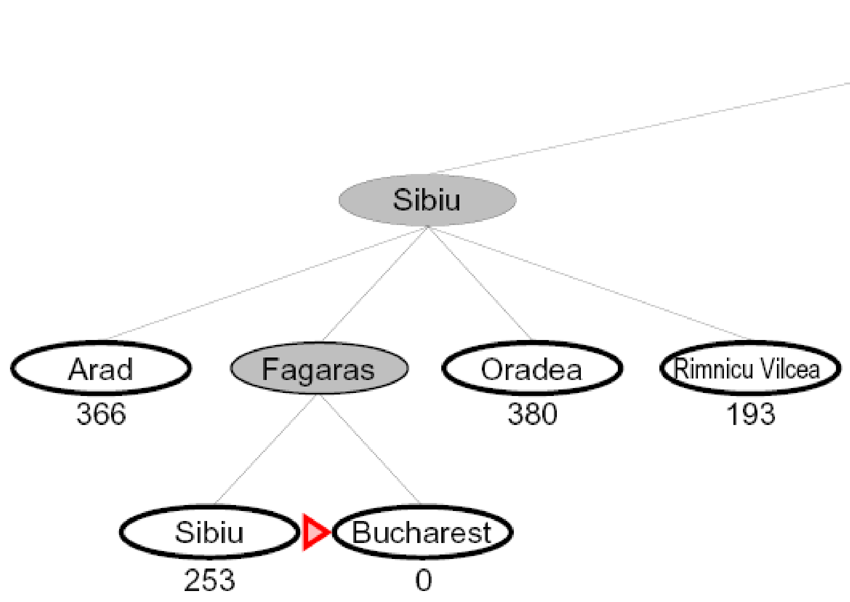
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

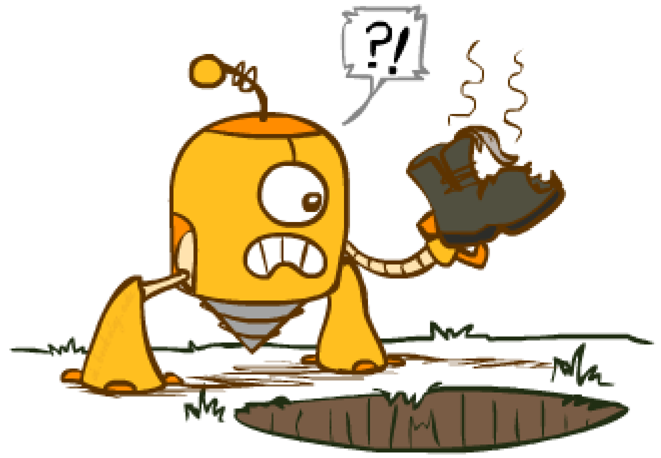
Greedy Search

- Expand the node that seems closest…



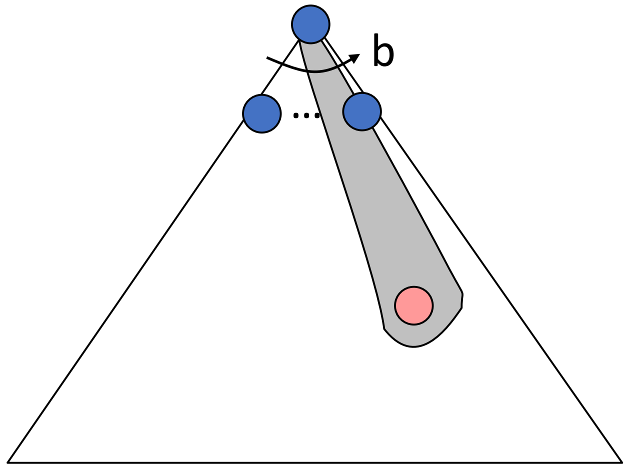


- What can go wrong?

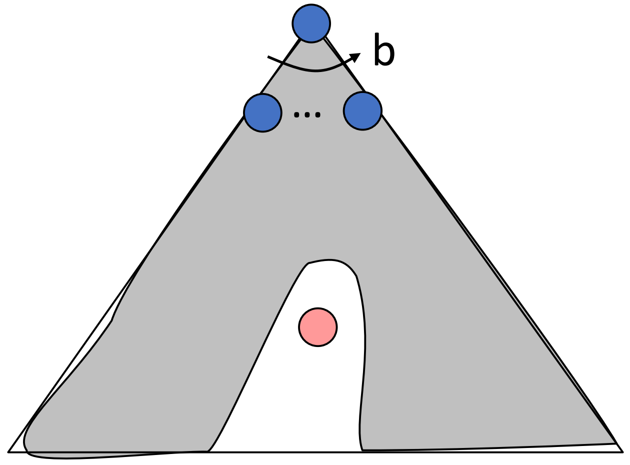


Greedy Search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



- A common case:
  - Best-first takes you straight to the (wrong) goal



- Worst-case: like a badly-guided DFS

A* Search

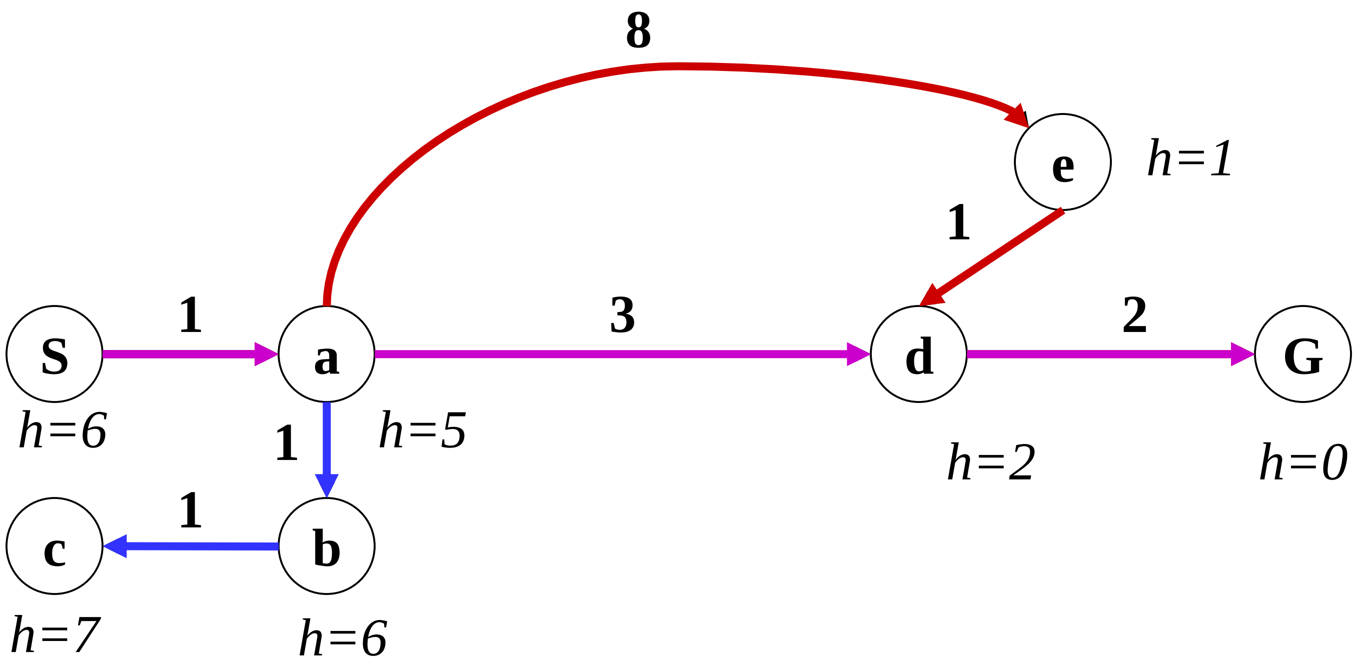
A* Search Heuristic

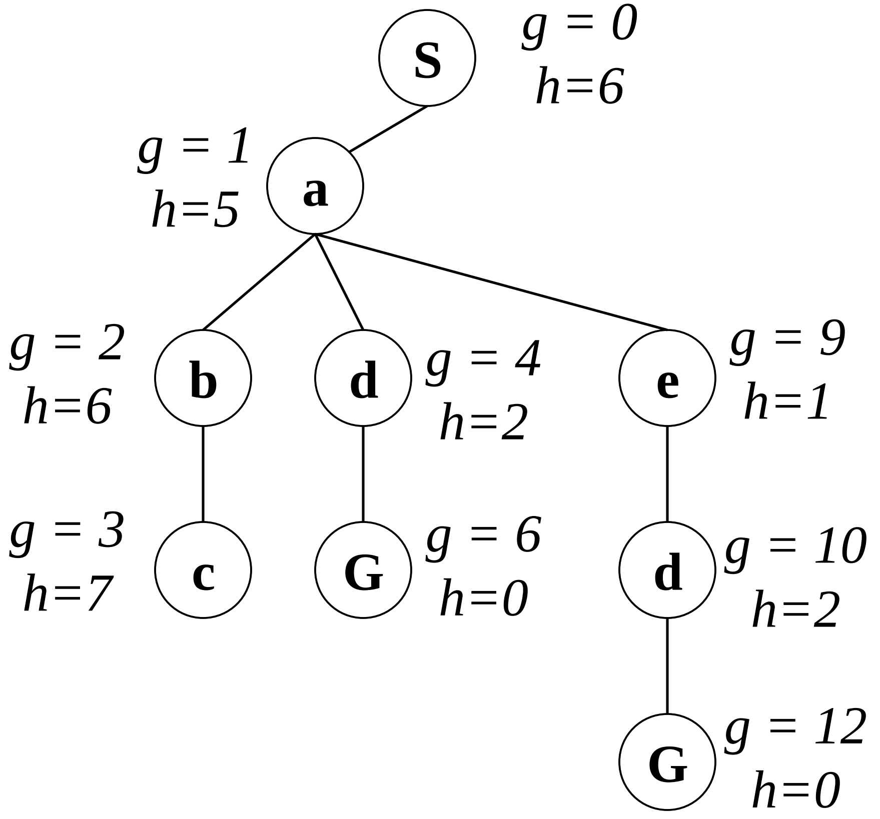
- A search algorithm to find the shortest path through a search space to a goal state using a heuristic.
- f = g + h
- g - the cost of getting from the initial state to the current state
- h - the cost of getting from the current state to a goal state

A* Search

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)

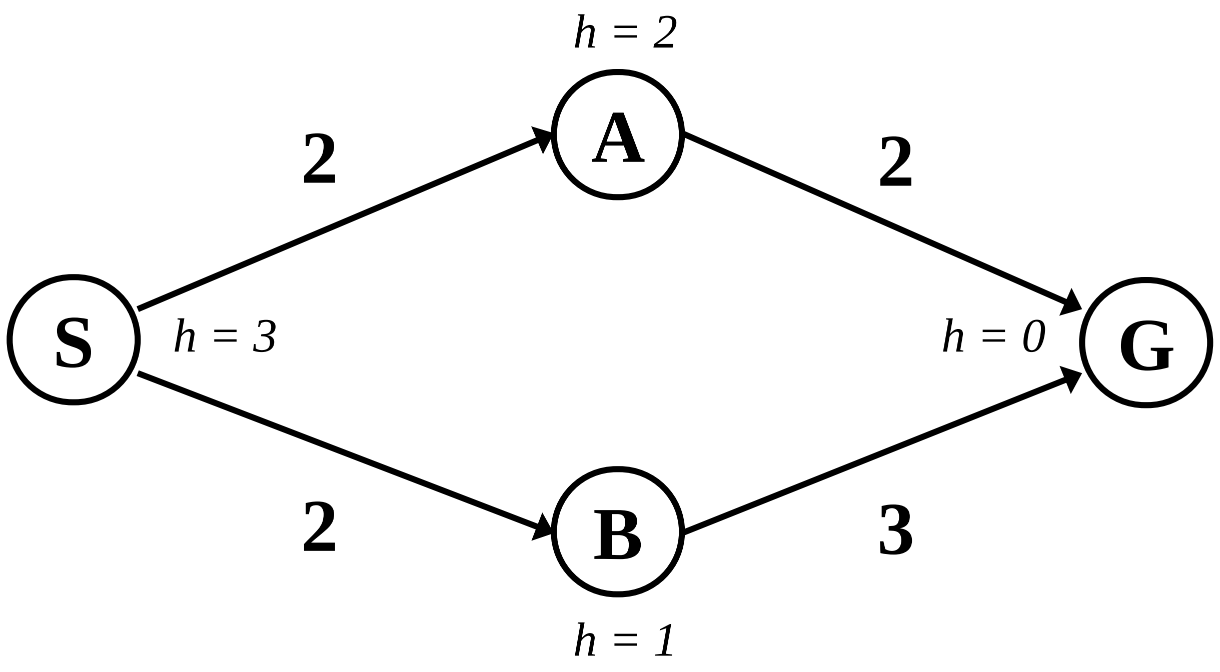




- A* Search orders by the sum: f(n) = g(n) + h(n)

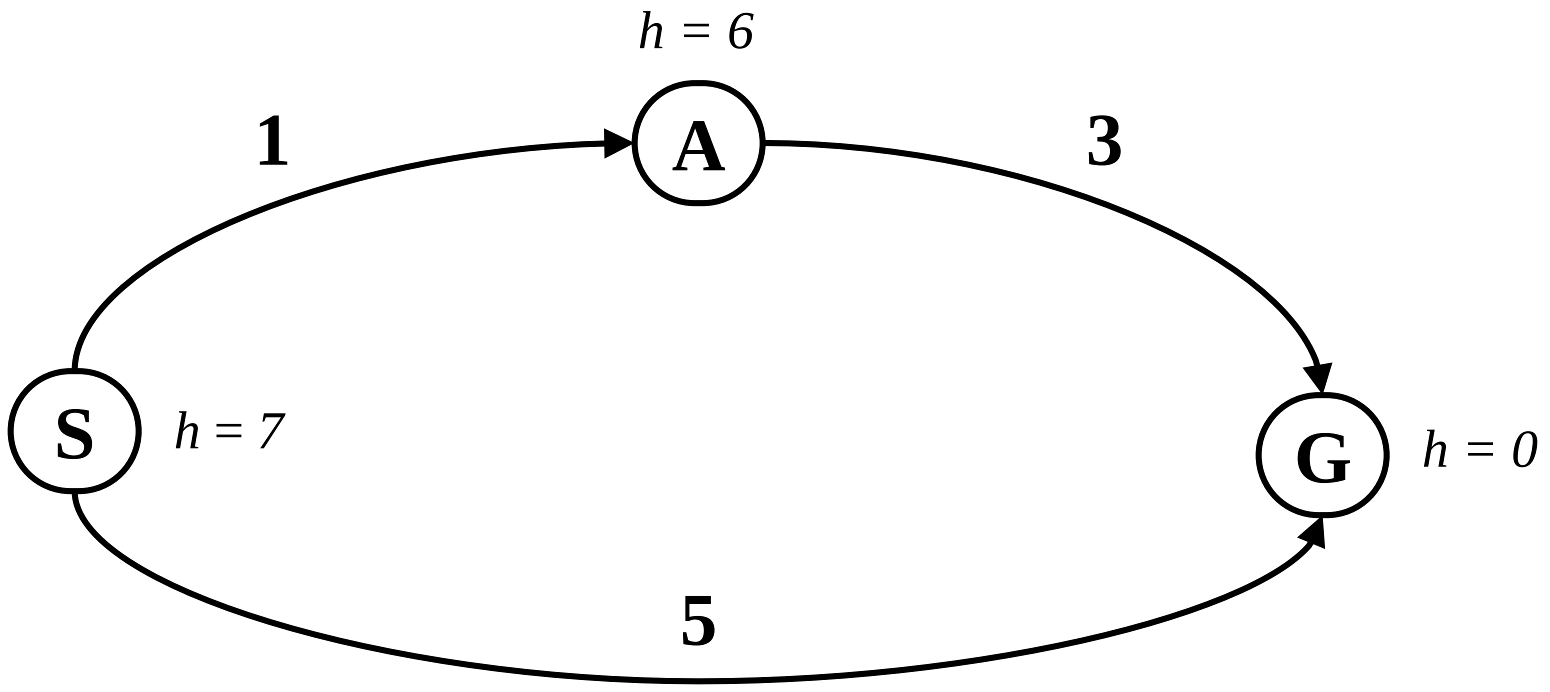
When should A* terminate?

- Should we stop when we enqueue a goal?



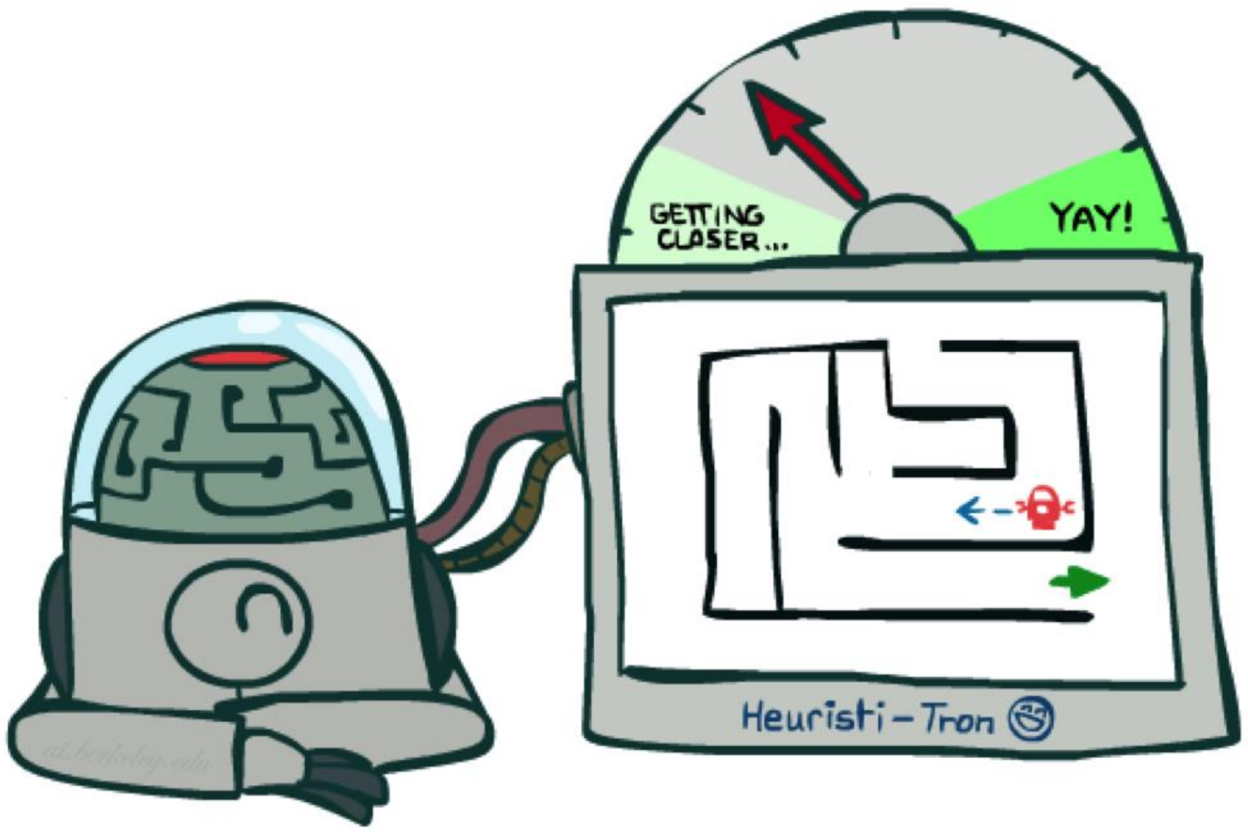
- No: only stop when we dequeue a goal

Is A* Optimal?

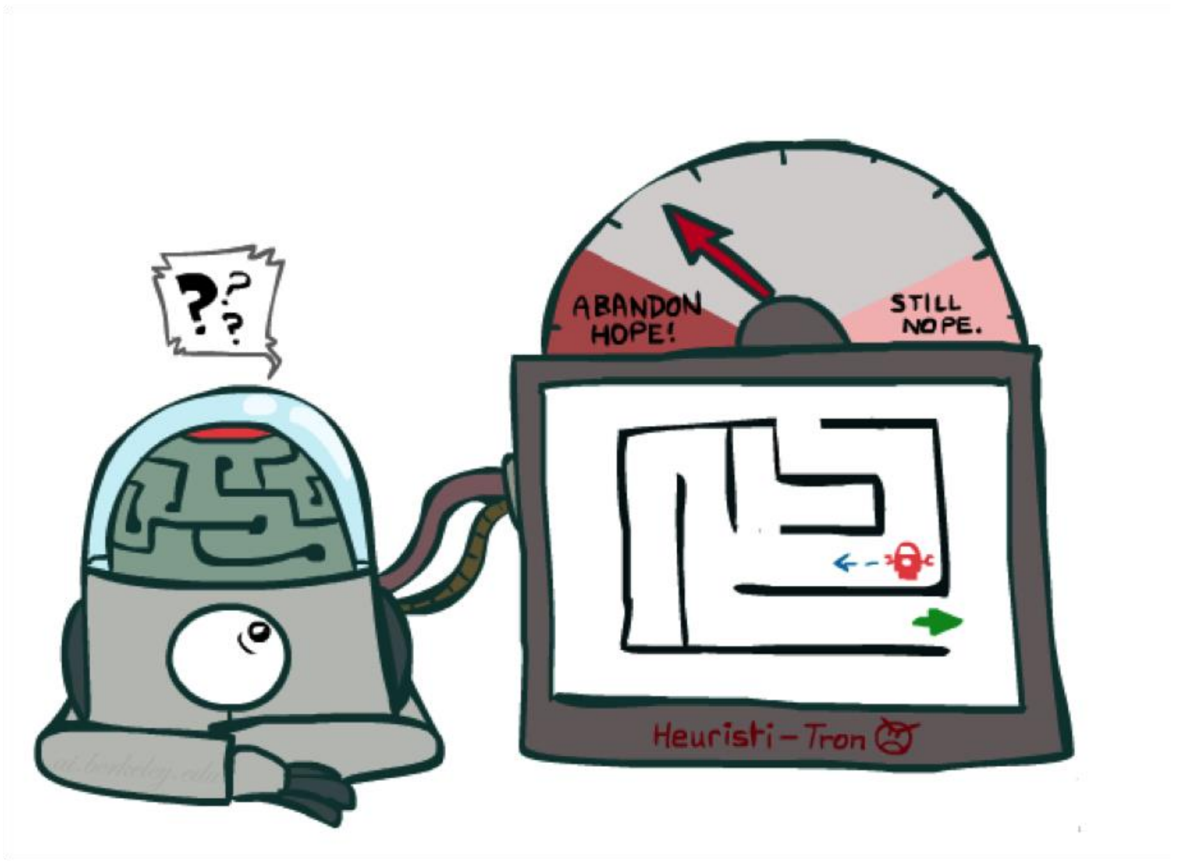


- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

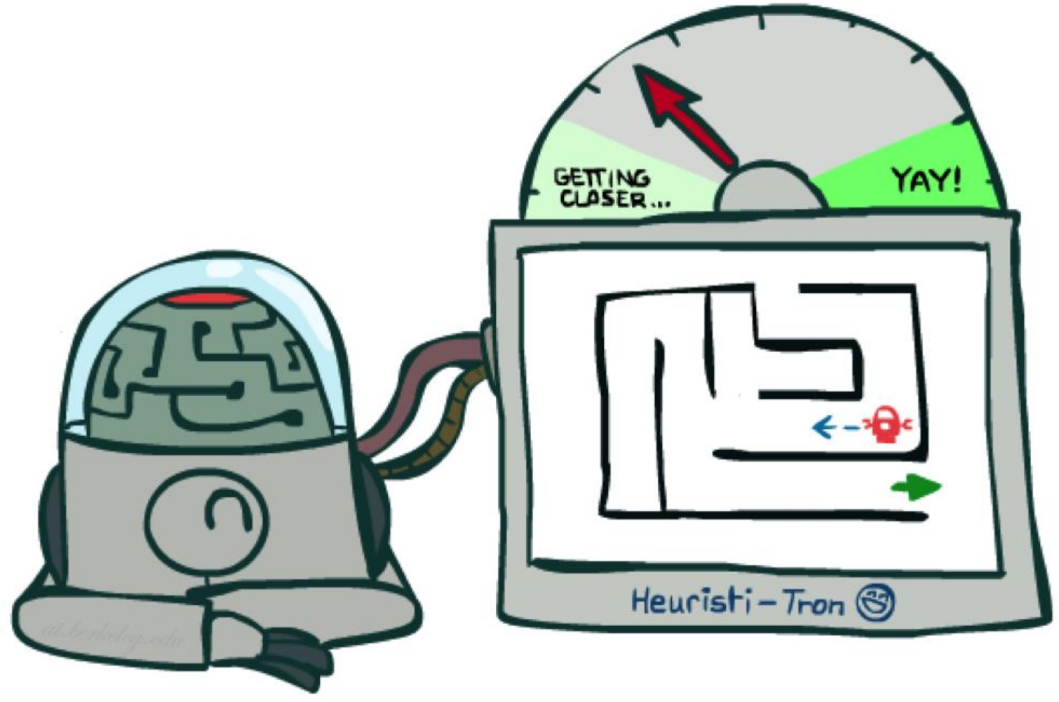
Admissible Heuristics



Idea: Admissibility



Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe



Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs

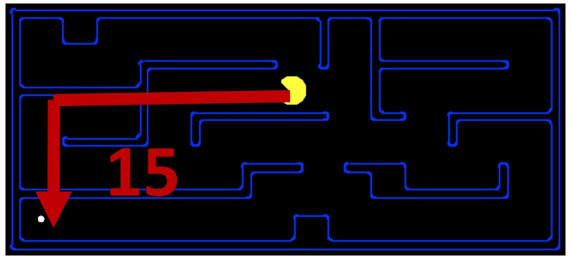
Admissible Heuristics

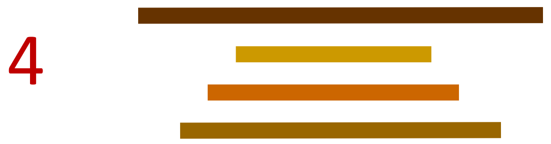
- A heuristic *h* is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal

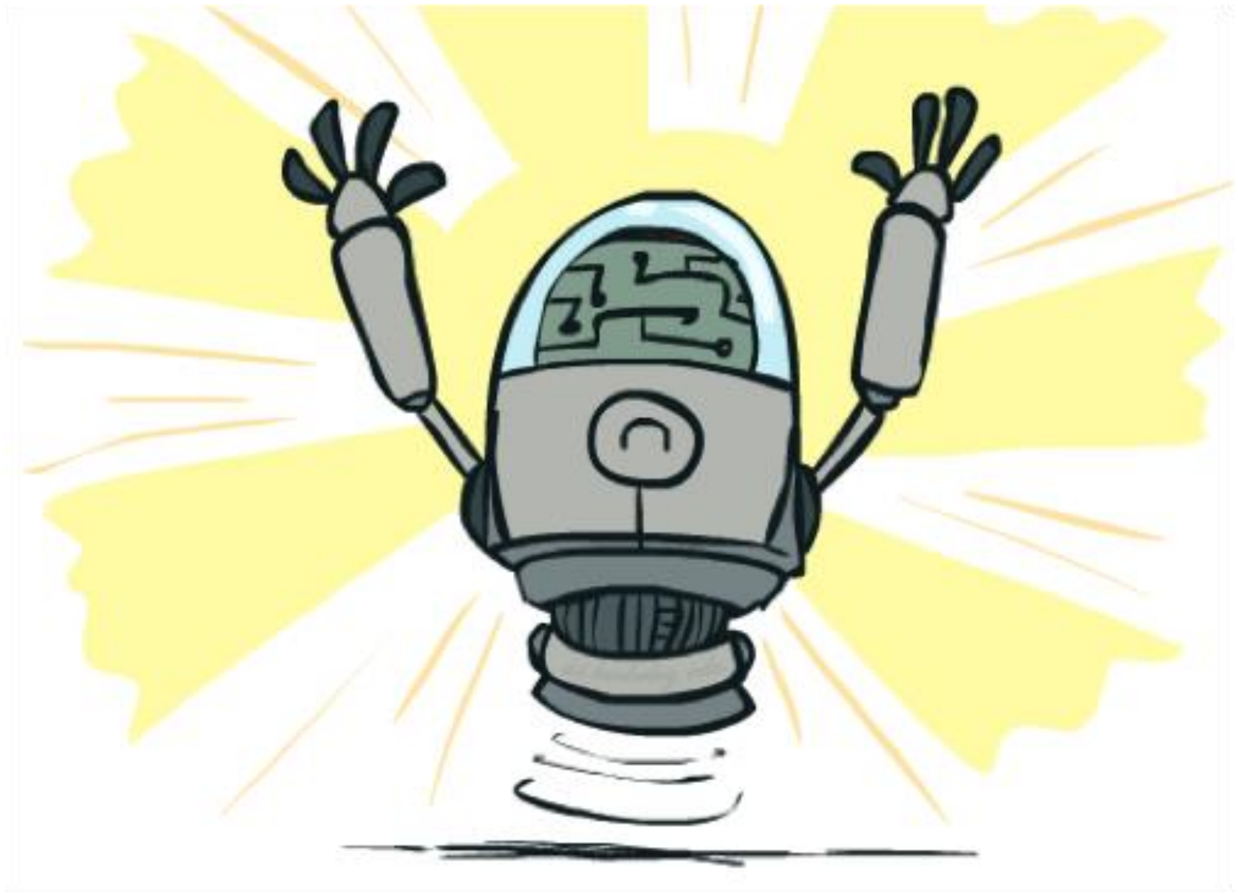
- Examples:





- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A* Tree Search



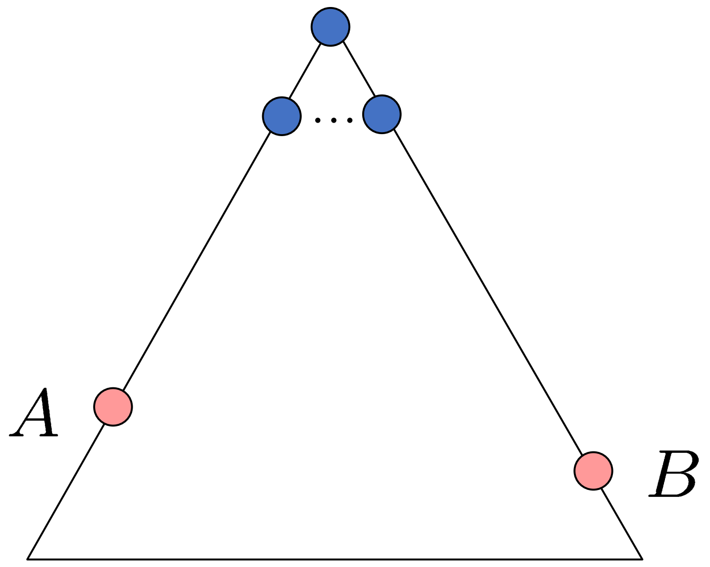
Optimality of A* Tree Search

Assume:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:

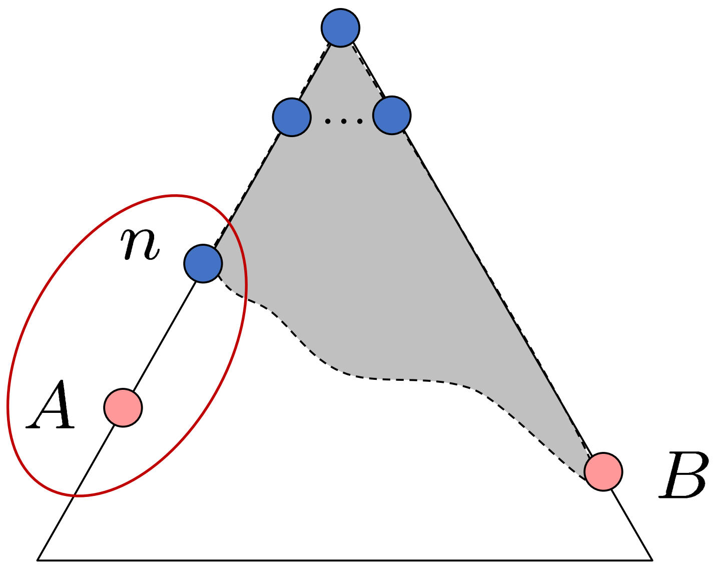
- A will exit the fringe before B



Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
    1. f(n) is less or equal to f(A)

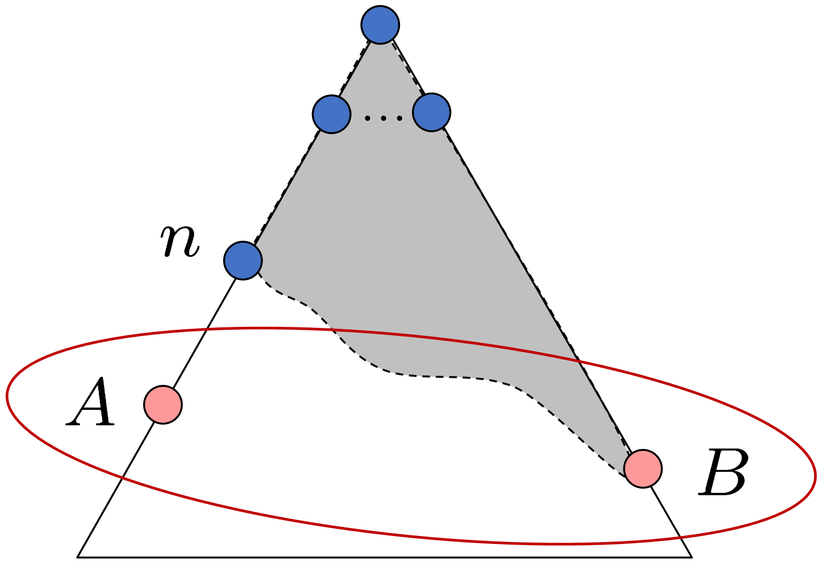


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
   1. f(n) is less or equal to f(A)
   2. f(A) is less than f(B)



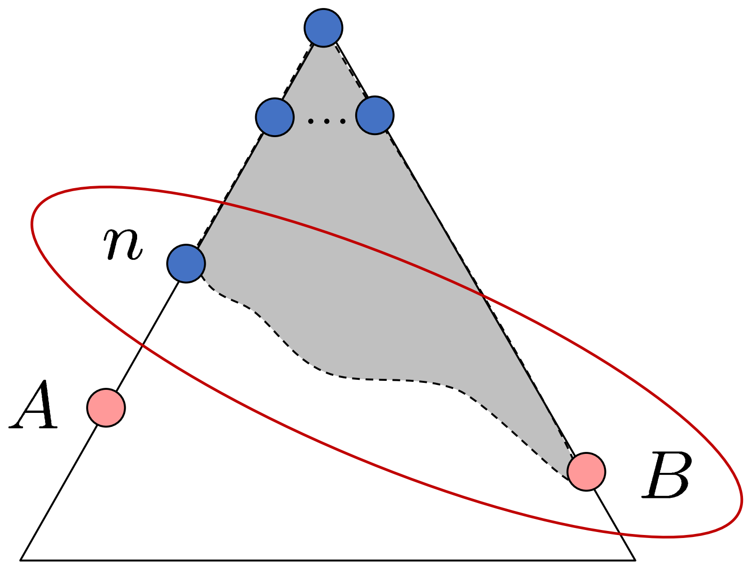
$g(A) < g(B)$ B is suboptimal

$f(A) < f(B)$ h = 0 at a goal

Optimality of A* Tree Search: Blocking

Proof:

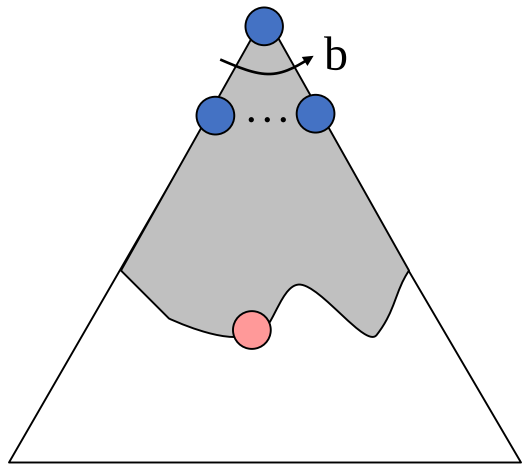
- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)
  3. *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal



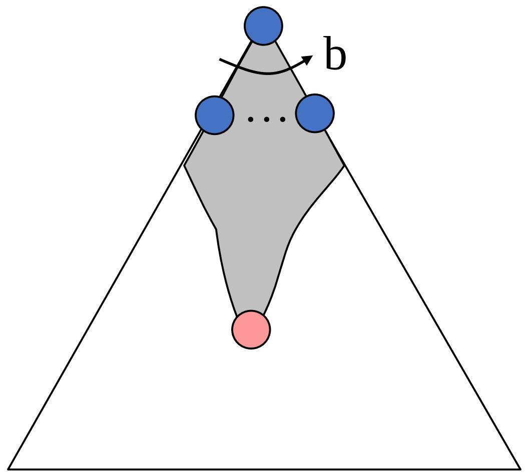
$$f(n) \leq f(A) < f(B)$$

Properties of A*

Uniform-Cost

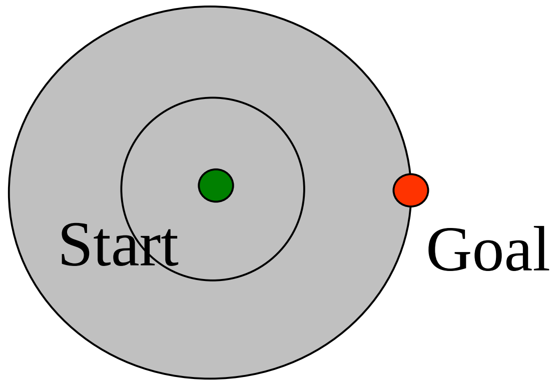


A
*

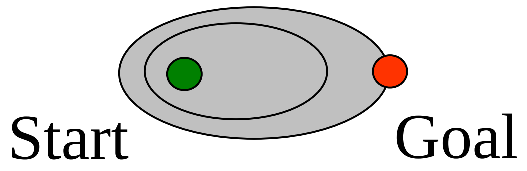


UCS vs A* Contours

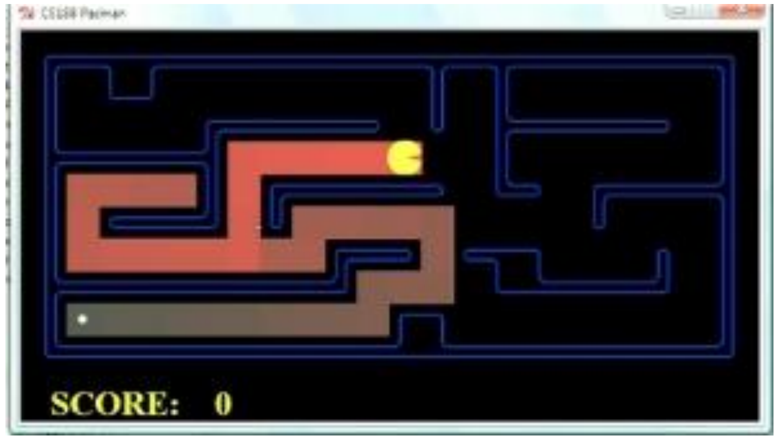
- Uniform-cost expands equally in all "directions"

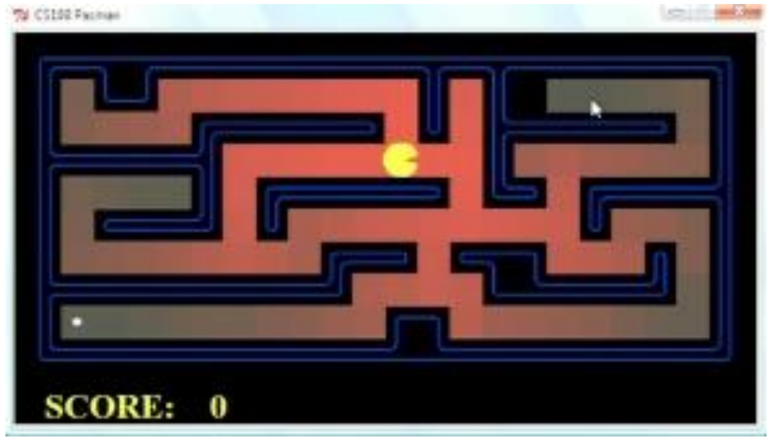


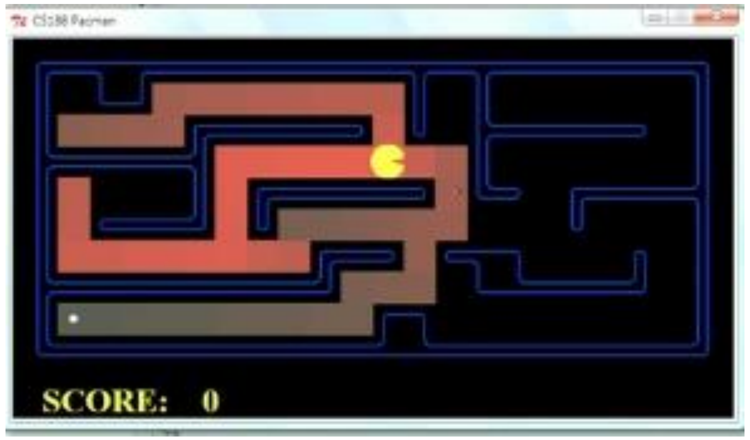
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



Comparison





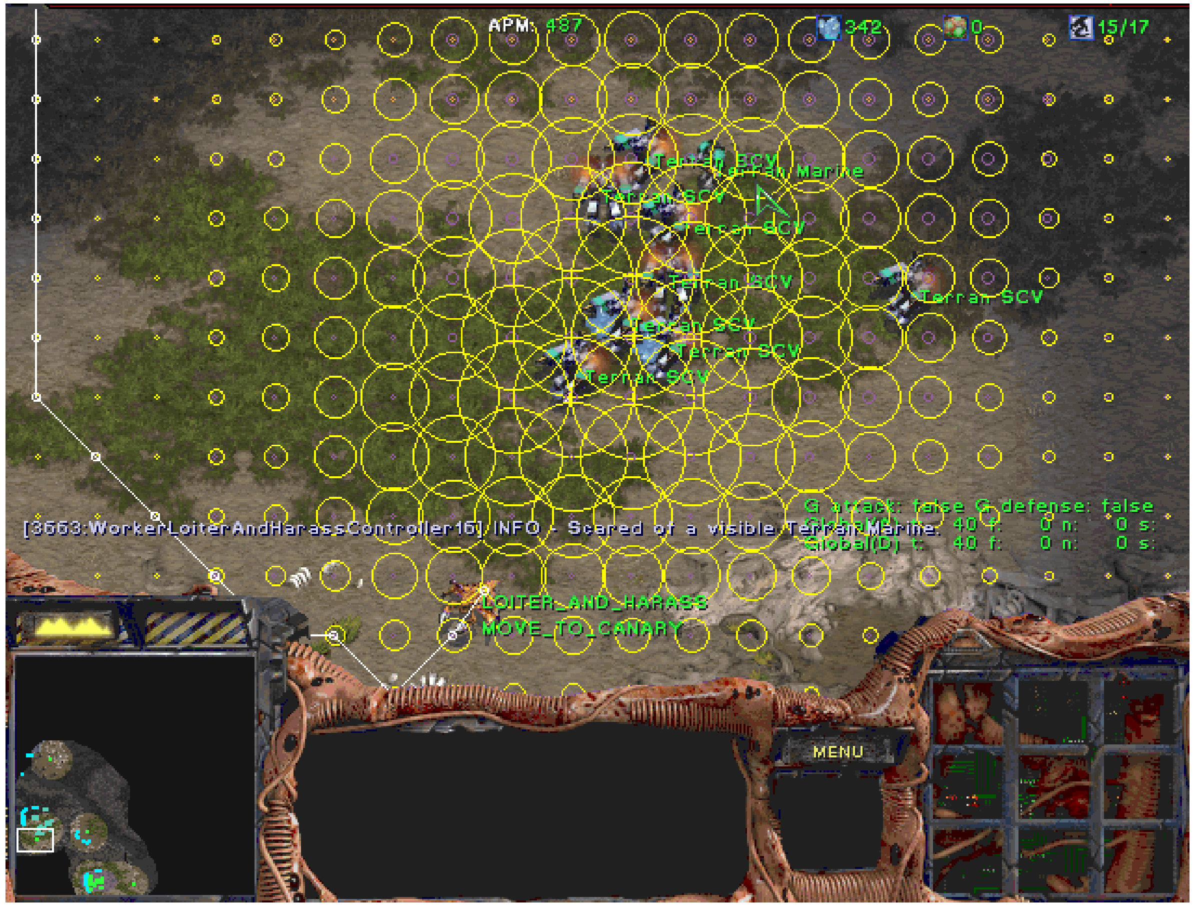


Greedy

Uniform Cost

A*

A* Applications

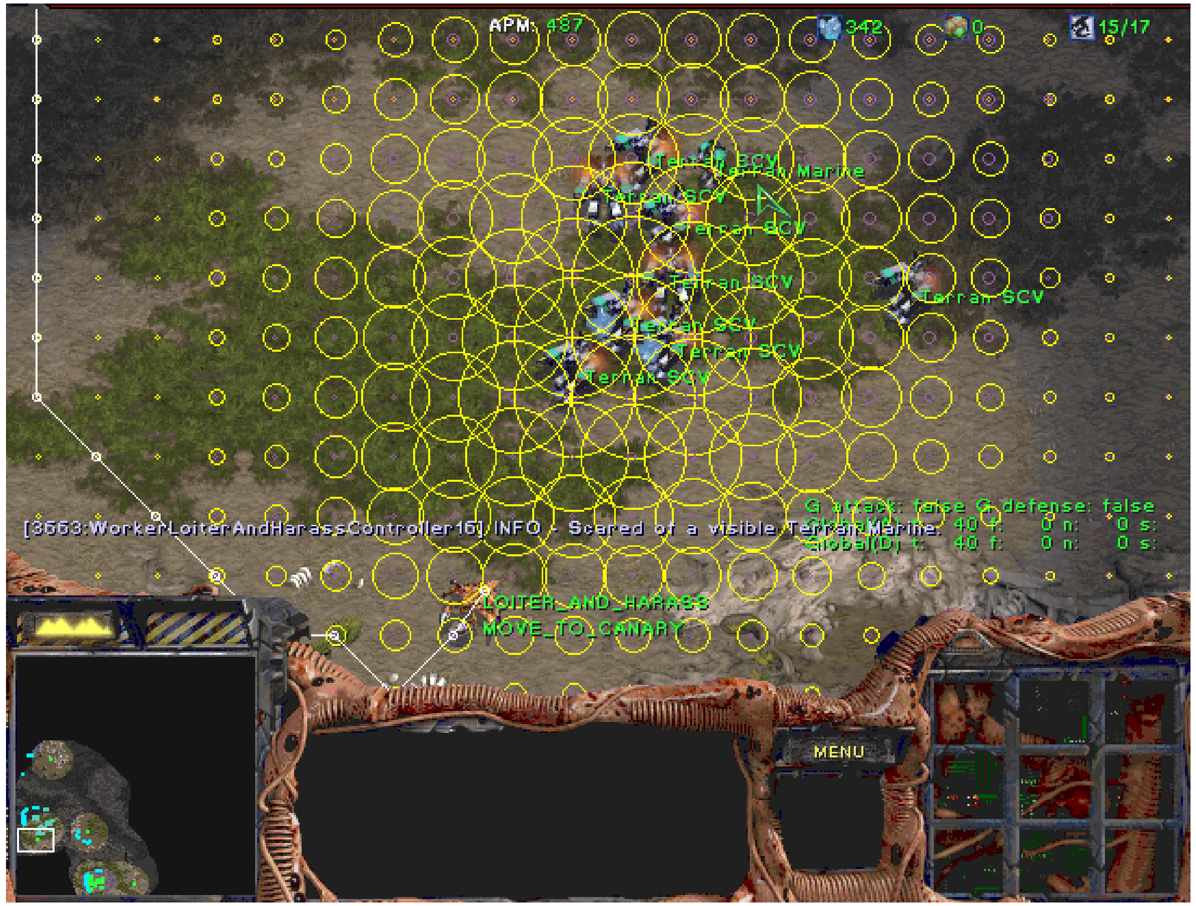


```
[3663:WorkerLoiterAndHarassController16] INFO - Scared of a visible TerranMarine.
```

```
G attack: false G defense: false
Global(A) t:   40 f:    0 n:    0 s:
Global(D) t:   40 f:    0 n:    0 s:
```

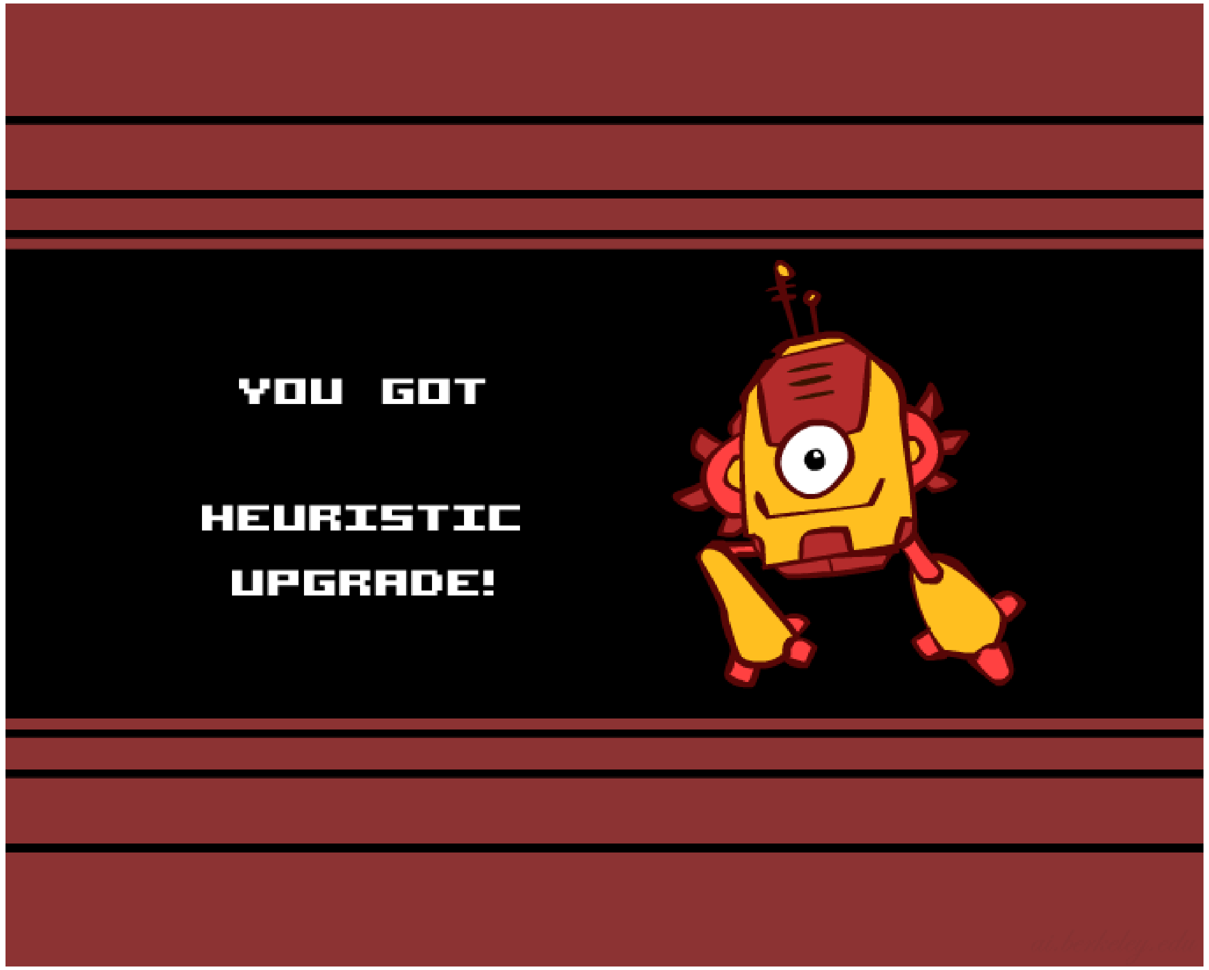
A* Applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …



```
[3663:WorkerLoiterAndHarassController16] INFO - Scared of a visible Terran Marine
G attack: false G defense: false
Global(A) t:  40 f:   0 n:   0 s:
Global(D) t:  40 f:   0 n:   0 s:
```

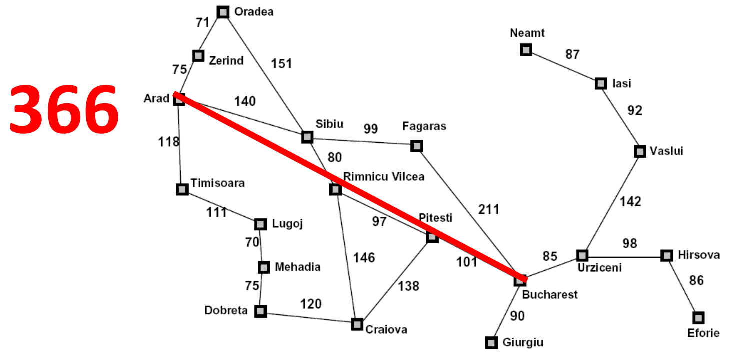
Creating Heuristics

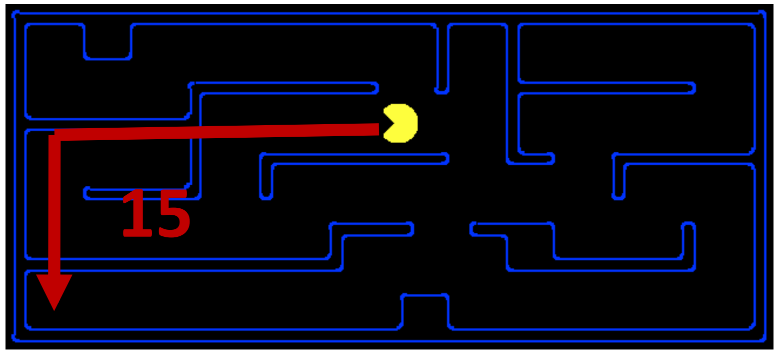


Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics

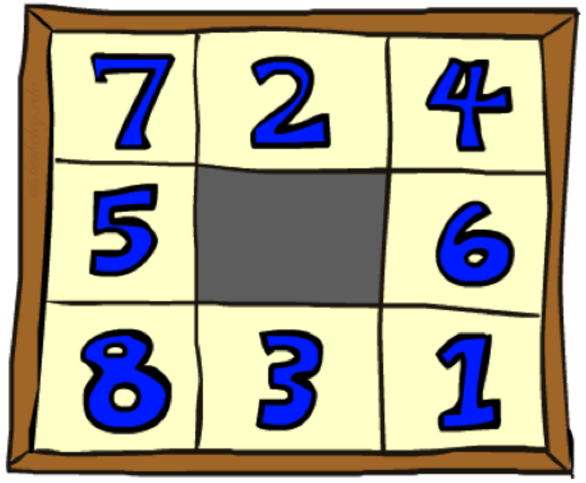
- Often, admissible heuristics are solutions to *relaxed problems*, where new actions are available



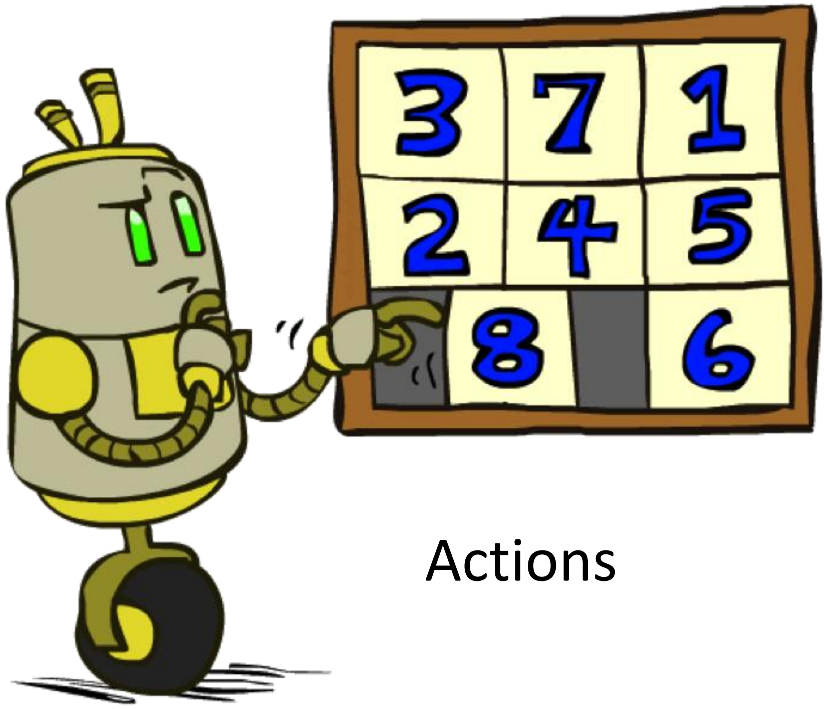


- Inadmissible heuristics are often useful too

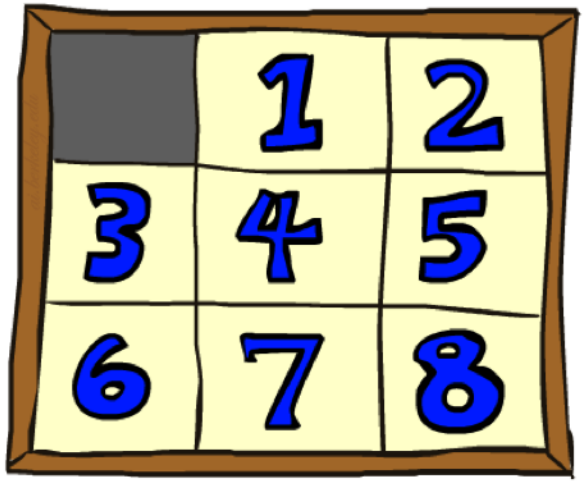
Example: 8 Puzzle



Start State



Actions

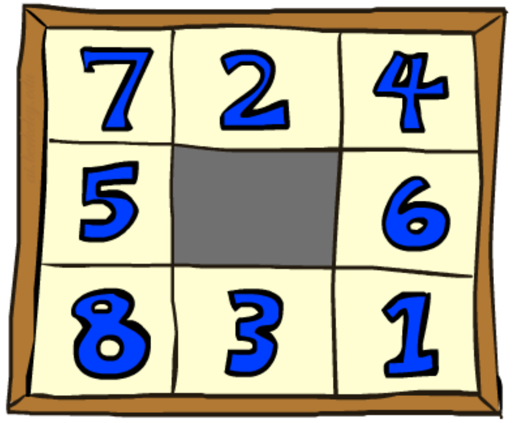


Goal State

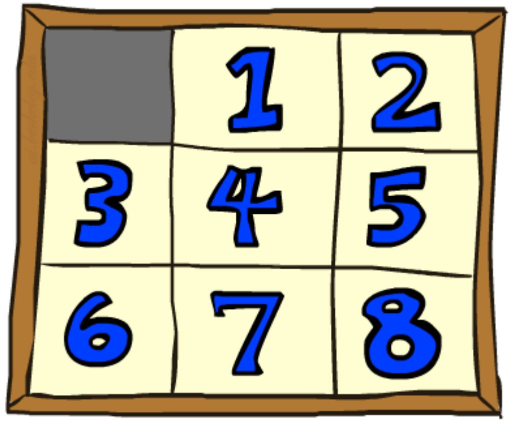
- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

8 Puzzle I

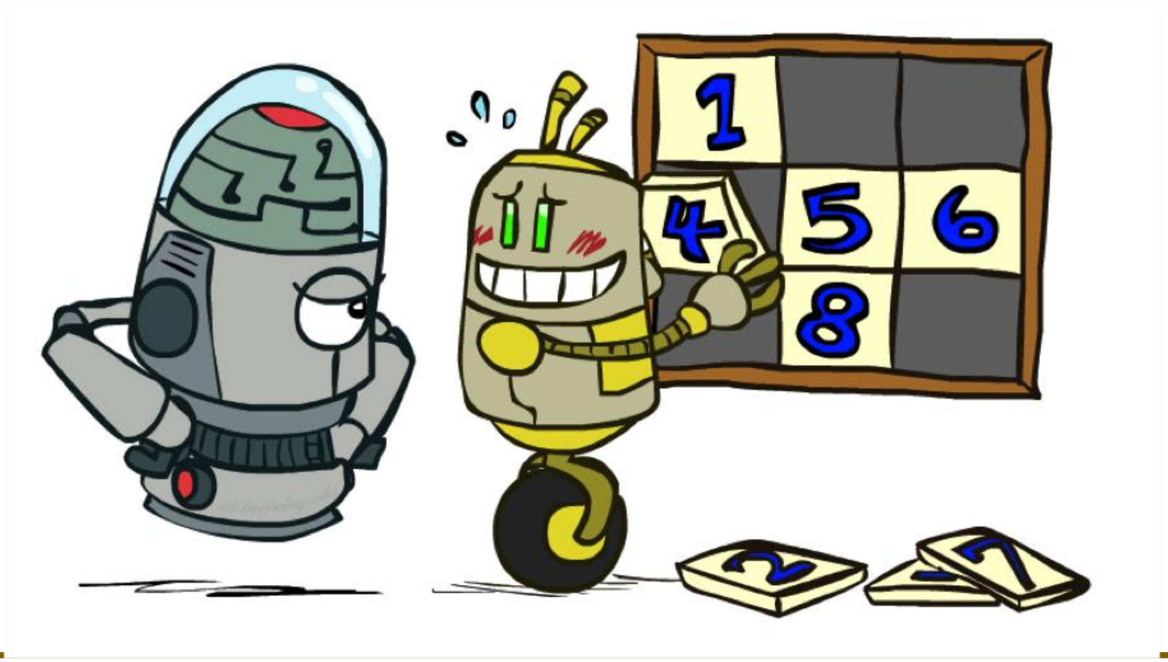
- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a *relaxed-problem* heuristic



Start State



Goal State



	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

Statistics from Andrew Moore

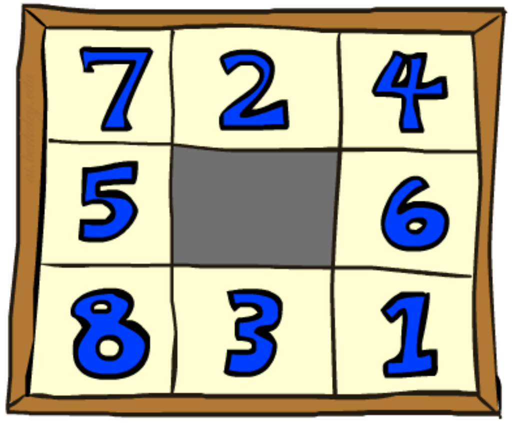
8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?

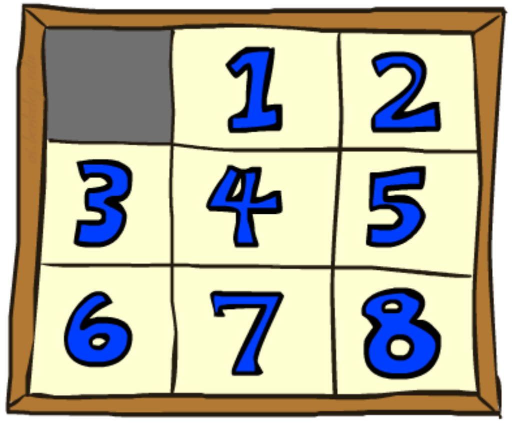
- Total *Manhattan* distance

- Why is it admissible?

- h(start) = 3 + 1 + 2 + … = 18



Start State

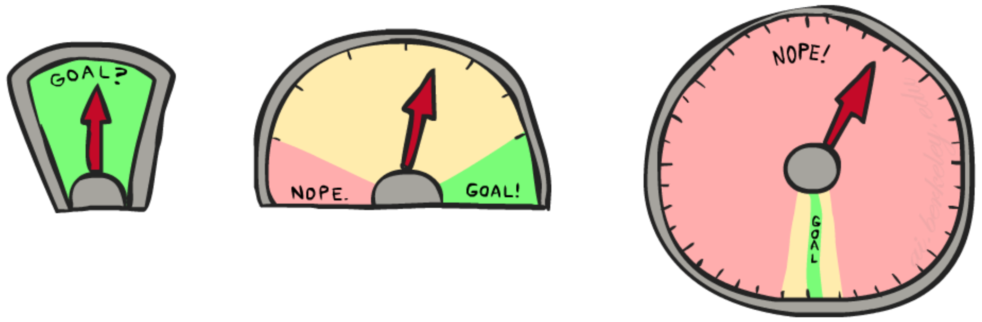


Goal State

	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MANHATTAN	12	25	73

8 Puzzle III

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?



- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

Hill Climbing:-

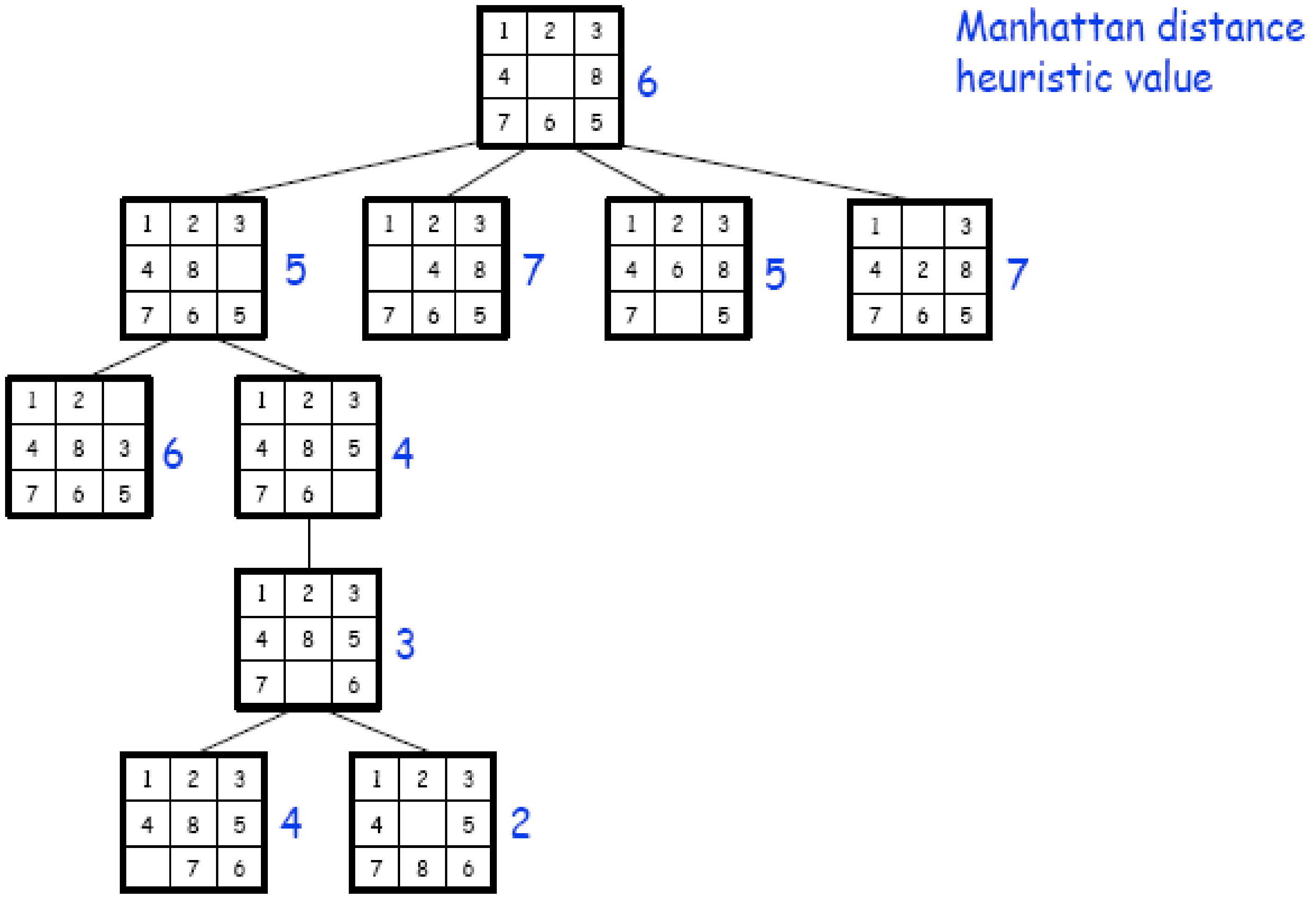
- In this technique we begin by testing nodes in order just as we would in a blind search, each time we discover that the node we are testing does not satisfy the goal state, **we calculate the difference between the node and the goal state**, **by comparing the difference we can determine that whether we are moving closer to the goal state or further away from it**. If we are moving away from the goal then we can backtrack and select a new path.

Hill Climbing:-

- In hill climbing the basic idea is to always head towards a state which is better than the current one.

So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

- Hill climbing terminates when there are no successors of the current state which are better than the current state itself.



Algorithm:

- Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
- Loop until a solution is found or until there are no new operators left to be applied in the current state:
  - Select an operator that has not yet been applied to the current state and apply it to produce a new state.
  - Evaluate the new state
    - If it is a goal state, then return it and quit.
    - If it is not a goal state but it is better than the current state, then make it the current state.
    - If it is not the better than the current state, then continue in the loop.

Constraint Satisfaction Problems

- Many problems in AI can be viewed as problems of constraint satisfaction in which the goal is to discover some problem states that satisfies a given set of constraints, such problems are known as *Constraint Satisfaction Problems (CSPs)*.

Examples:

- Example 1: The n-Queen problem: The local condition is that no two queens attack each other, i.e. are on the same row, or column, or diagonal.
- Example 2: A crossword puzzle: We are to complete the puzzle

```
    1   2   3   4   5
  +---+---+---+---+---+ Given the list of words:
1 | 1 |   | 2 |   | 3 |              AFT        LASER
  +---+---+---+---+---+ ALE          LEE
2 | # | # |   | # |   |              EEL        LINE
  +---+---+---+---+---+ HEEL         SAILS
3 | # | 4 |   | 5 |   |              HIKE       SHEET
  +---+---+---+---+---+ HOSES        STEER
4 | 6 | # | 7 |   |   |              KEEL       TIE
  +---+---+---+---+---+ KNOT
5 | 8 |   |   |   |   |
  +---+---+---+---+---+
6 |   | # | # |   | # |
  +---+---+---+---+---+
```

- The numbers 1,2,3,4,5,6,7,8 in the crossword puzzle correspond to the words that will start at those locations.

- **Example 3: A cryptography problem:** In the following pattern

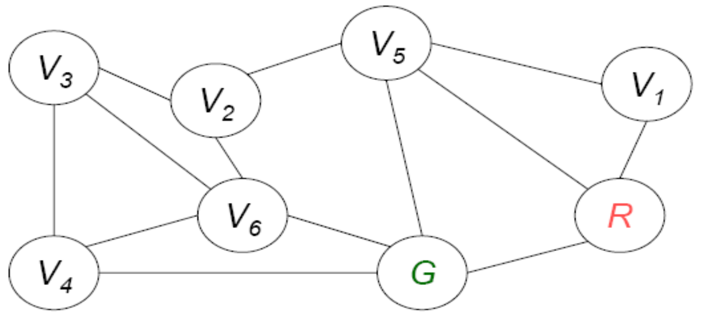
```
    S E N D
    M O R E
=========
M O N E Y
```

- We have to replace each letter by a distinct digit so that the resulting sum is correct.
- **Example 4: A map coloring problem:** We are given a map, i.e. a planar graph, and we are told to color it using three colors, green, red, and blue, so that no two neighboring countries have the same color.

A **Constraint Satisfaction Problem** is characterized by:

- a *set of variables* {x1, x2, .., xn},
- for each variable xi a *domain* Di with the possible values for that variable, and
- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognizing function.] We will only consider constraints involving one or two variables.
- The constraint satisfaction problem is to find, for each *i* from 1 to n, a value in Di for xi so that all constraints are satisfied.

Coloring Example:



- Inside each circle marked *V1* .. *V6* we must assign: *R*, *G* or *B*.
- No two connected circles may be assigned the same symbol.
- Notice that two circles have already been given an assignment.
- A CSP is a triplet { *V* , *D* , *C* }.
- A CSP has a finite set of variables *V* = { *V1* , *V2*... *VN* }.
- Each variable may be assigned a value from a domain *D* of values.
- Each member of *C* is a pair. The first member of each pair is a set of variables.
- The second element is a set of legal values which that set may take.

Example:

V = { *V1* , *V2* , *V3* , *V4* , *V5* , *V6* }

D = { *R* , *G* , *B* }

C = { (*V1*,*V2*) : { (*R*,*G*), (*R*,*B*), (*G*,*R*), (*G*,*B*), (*B*,*R*) (*B*,*G*)},

{ (*V1*,*V3*) : { (*R*,*G*), (*R*,*B*), (*G*,*R*), (*G*,*B*), (*B*,*R*) (*B*,*G*)},

:

: }

Problem Reduction Searching

- Planning to solve a problem that can be recursively decomposed into sub-problems into multiple ways, and then to select the best way in order to minimize or maximize any given problem criteria.
  - AND-OR Graphs
  - Means-End Analysis
  - Game Trees

Problem Reduction ----- AND-OR Graphs

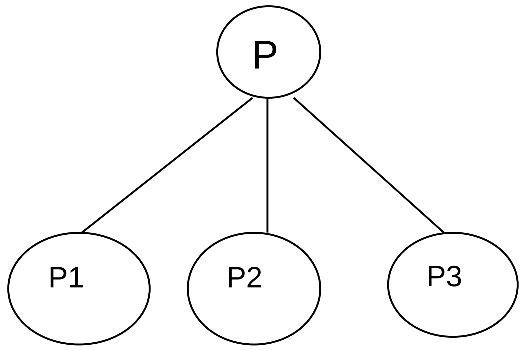
- AND-OR Graph is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved.
- This decomposition or reduction generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution.
- AND arcs are represented by line connecting all the components or states.
- In order to find solutions in an AND-OR graph, we need an algorithm similar to best first search but with the ability to handle the AND arcs appropriately.

Areas of Usage:

- Route finding
- Design
- Symbolic integration
- Game playing
- Theorem proving

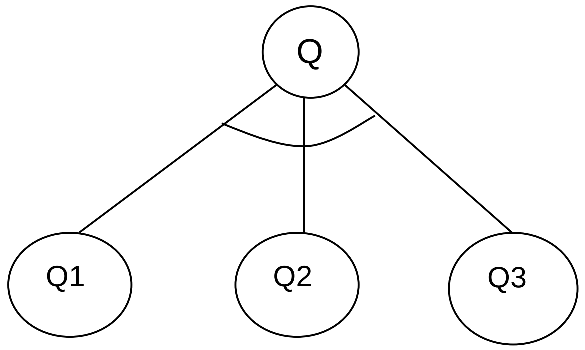
OR and AND Relations

OR Graph



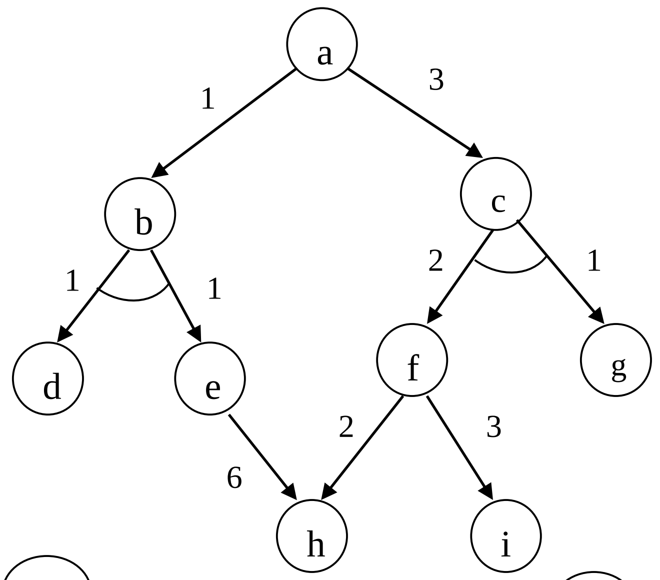
a)To solve P, solve any one of P1 or P2 or P3

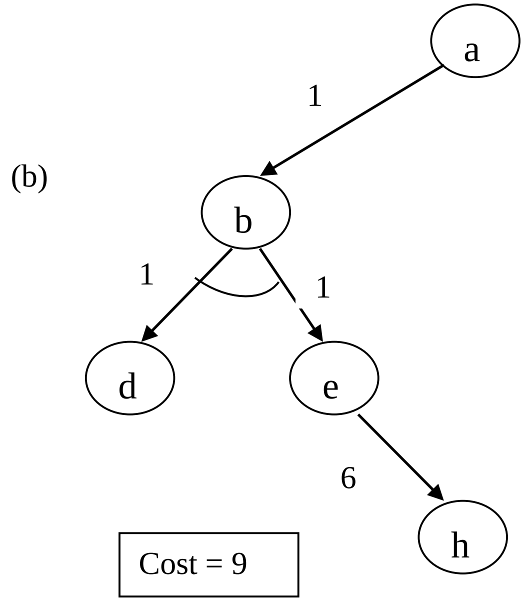
AND Graph

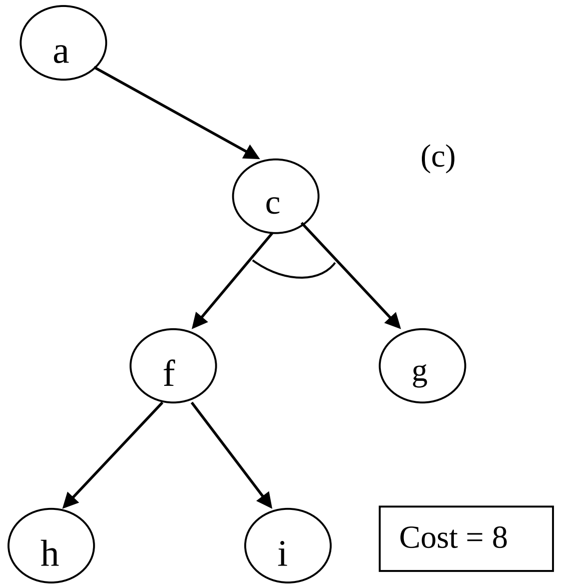


a)To solve Q, solve all Q1, Q2 and Q3

AND-OR Graph







Solution of AND/OR Graph

- In the state-space representation, a solution to the problem is a path in the state space.
- In the AND/OR graph representation, a solution is a tree that includes all the subproblems of an AND node.
- To describe an algorithm for searching an AND-OR graph, we need to exploit a value that we call *FUTILITY*. If the estimated cost of a solution becomes greater than the value of *FUTILITY*, then we abandon the search. *FUTILITY* should be chosen to correspond to a threshold such that any solution with a cost above it is too expensive to be practical, even if it could ever be found.

Algorithm:

- Initialize the graph to the starting node.
- Loop until the starting node is labeled *SOLVED* or until its cost goes above *FUTILITY.*
  - Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
  - Pick one of these unexpanded nodes and expand it. If there are no successors, assign *FUTILITY* as the value of this node. Otherwise, add its successors to the graph and for each of them compute cost. If cost of any node is 0, mark that node as *SOLVED*.

Algorithm (Cont…)

- Change the cost estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as *SOLVED*.
- At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propagation of revised cost estimates back up the tree was not necessary in the best first search algorithm. Here expanded nodes are re-examined so that the best current path can be selected. Thus it is important that their cost estimate values be the best available.

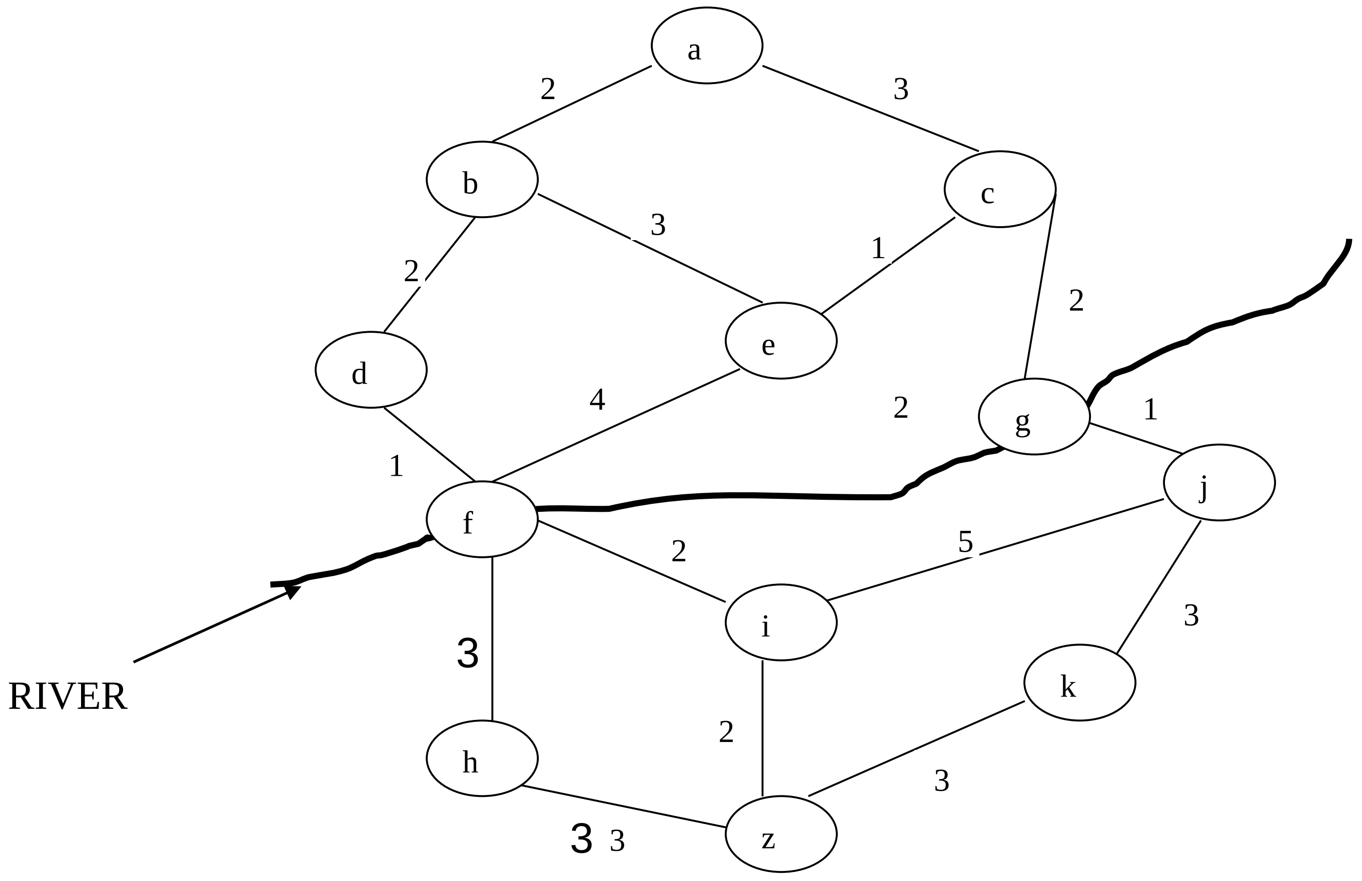
Route Finding Example:

Finding a route from a to z in a road map

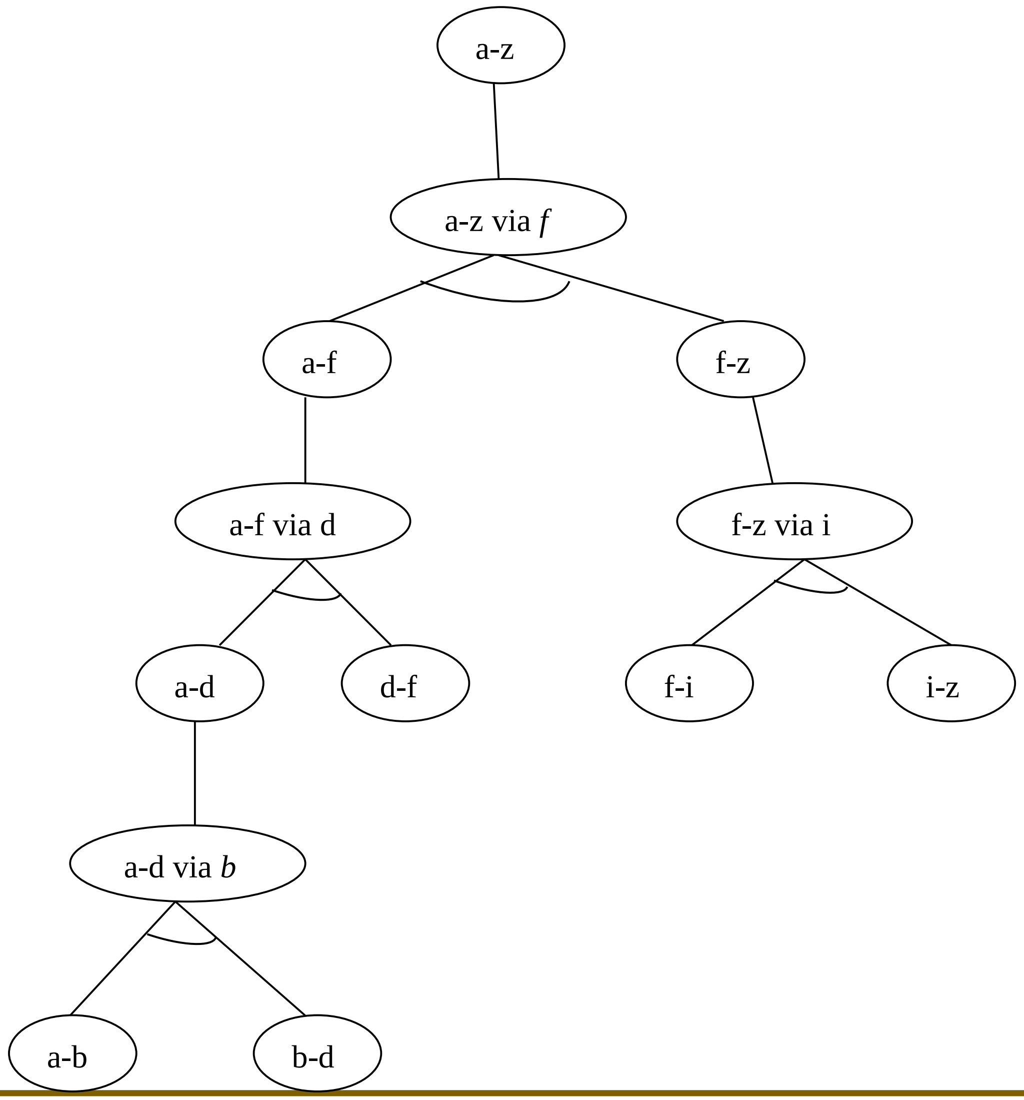
To find a path between a and z, find *either*

(1) a path from a to z via f, *or*

(2) a path from a to z via g.



AND – OR representation of Route finding



Means – Ends Analysis

- Means-Ends Analysis (MEA) is a technique used in Artificial Intelligence for controlling search in problem solving computer programs.

  'Means' = the different 'means' you can possibly use to solve that problem

  'Ends' = the 'end' goals of that problem

- Means-ends analysis is combination of forward and backward strategies.

- The MEA technique is a strategy to control search in problem-solving. Given a current state and a goal state, an action is chosen which will reduce the *difference* between the two states. The action is performed on the current state to produce a new state, and the process is recursively applied to this new state and the goal state.

Effective MEA

- In order for MEA to be effective, the goal-seeking system must have a means of associating to any kind of detectable difference those actions that are relevant to reducing that difference.
- It must also have means for detecting the progress it is making (the changes in the differences between the actual and the desired state), as some attempted sequences of actions may fail and, hence, some alternate sequences may be tried.

Effective MEA (Cont…)

- When knowledge is available concerning the importance of differences, the most important difference is selected first to further improve the average performance of MEA over other search strategies.
- However, even without the ordering of differences according to importance, MEA improves over other search heuristics by focusing the problem solving on the actual differences between the current state and that of the goal.

Means-ends analysis - Steps:

1. Note the difference between the current state and the goal state
2. Create an achievable intermediate state (subgoal) that will reduce this difference
3. Select and apply a suitable operator to move to the intermediate state

In means-ends analysis, the problem solver compares the present situation with the goal, detects a difference between them, and then searches memory for actions that are likely to reduce the difference.

Compare your current state with the goal and choose an action to bring you closer to the goal. Generally break a problem down into smaller sub goals.

Continuously set new subgoals to shorten the distance between the current state and the final goal.

- **Example 1:**If the difference is a fifty-mile distance from the goal, the problem solver will retrieve from memory knowledge about autos, carts, bicycles, and other means of transport; walking and flying will probably be discarded as inappropriate for that distance.

- **Example 2:**Problem: How do you get to computer shop on the 3th floor, in 10th street in sector G-10 Islamabad, if you do not know your way around the area?

  Answer: You start by asking directions to G-10, then to 10th street, then you go to the 3rd floor, then you ask where the computer shop is.

Game Playing

- Games hold an inexplicable fascination for many peoples and the perception that computer might play games has existed at least as long as computers. It is one of the oldest areas of endeavor in Artificial Intelligence.
- There were two reasons that games appeared to be a good domain in which to explore machine intelligence:
  - They provide a structures task in which it is very easy to measure success or failure.
  - They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.

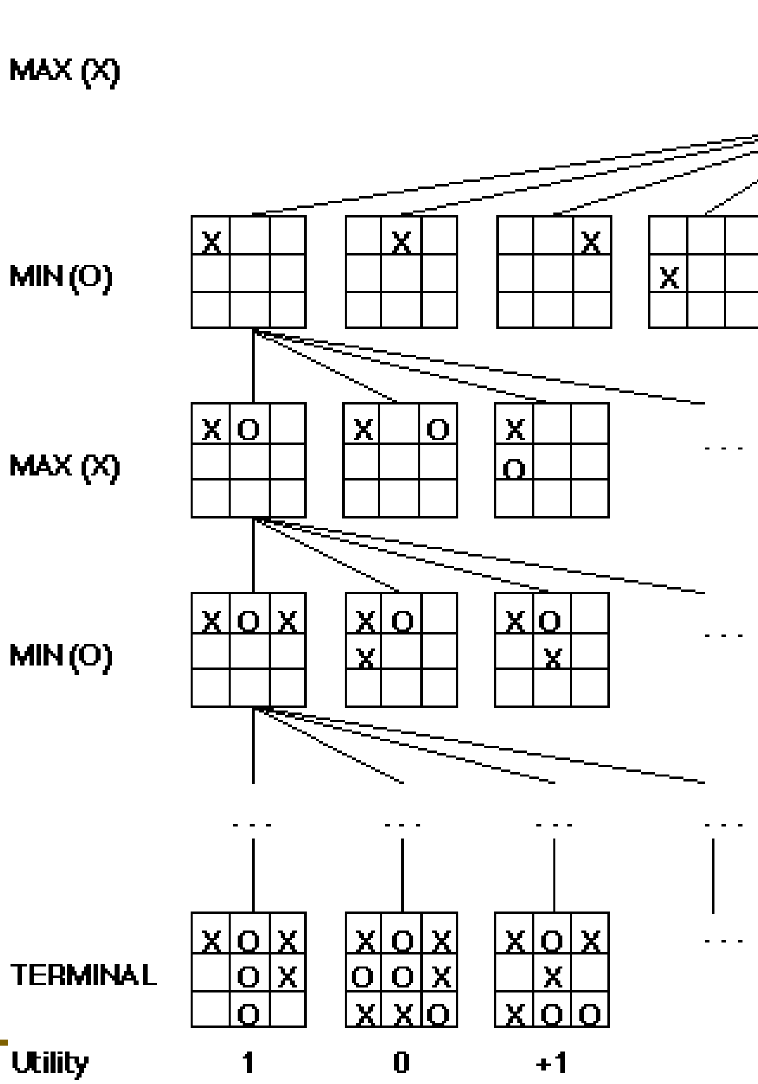
Game Trees (Or Tree of Max/Min Nodes)

- It is Problem spaces for typical games represented as OR-tree structure.
- Root node represents the "board" configuration.
- Arcs represent the possible legal moves for a player (no costs associates to arcs
- If it is MAX turn to move, then the root is labeled a "MAX" node; otherwise it is labeled a "MIN" node indicating my opponent's turn.
  - Max Node will Select maximum cost from successors
  - Min Node will select minimum cost from successors
- Terminal nodes represent end-game configurations (the result must be one of "win", "lose", and "draw", possibly with numerical payoff).
- Evaluator function rates a board position. f(board) (a real number).
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1
- Complete game tree: includes all configurations that can be generated from the root by legal moves (all leaves are terminal nodes)

Evaluation Function

- **Evaluation function** or **static evaluator** is used to evaluate the "goodness" of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node.
- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to both players.
  - **f(n) > 0**: position n good for max and bad for min.
  - **f(n) < 0**: position n bad for max and good for min
  - **f(n) near 0**: position n is a neutral position.
  - **f(n) >> 0**: win for max.
  - **f(n) << 0**: win for min..
- Evaluation function is a heuristic function, and it is where the domain experts' knowledge resides.

An Example (partial) game tree for Tic-Tac-Toe



f(n) = +1 if the position is a win for X.

f(n) = -1 if the position is a win for O.

f(n) = 0 if the position is a draw.

MINIMAX Search Procedure

- Goal of game tree search: to determine **one move** for Max player that **maximizes** the **guaranteed payoff** for a given game tree for MAX

  (Regardless of the moves the MIN will take)

- The value of each node (Max and MIN) is determined by (back up from) the values of its child nodes.
- MAX plays the worst case scenario:
  - Always assume MIN to take moves to maximize his pay-off (i.e., to minimize the pay-off of MAX)
- For a MAX node, the backed up value is the **maximum** of the values associated with its children
- For a MIN node, the backed up value is the **minimum** of the values associated with its children.

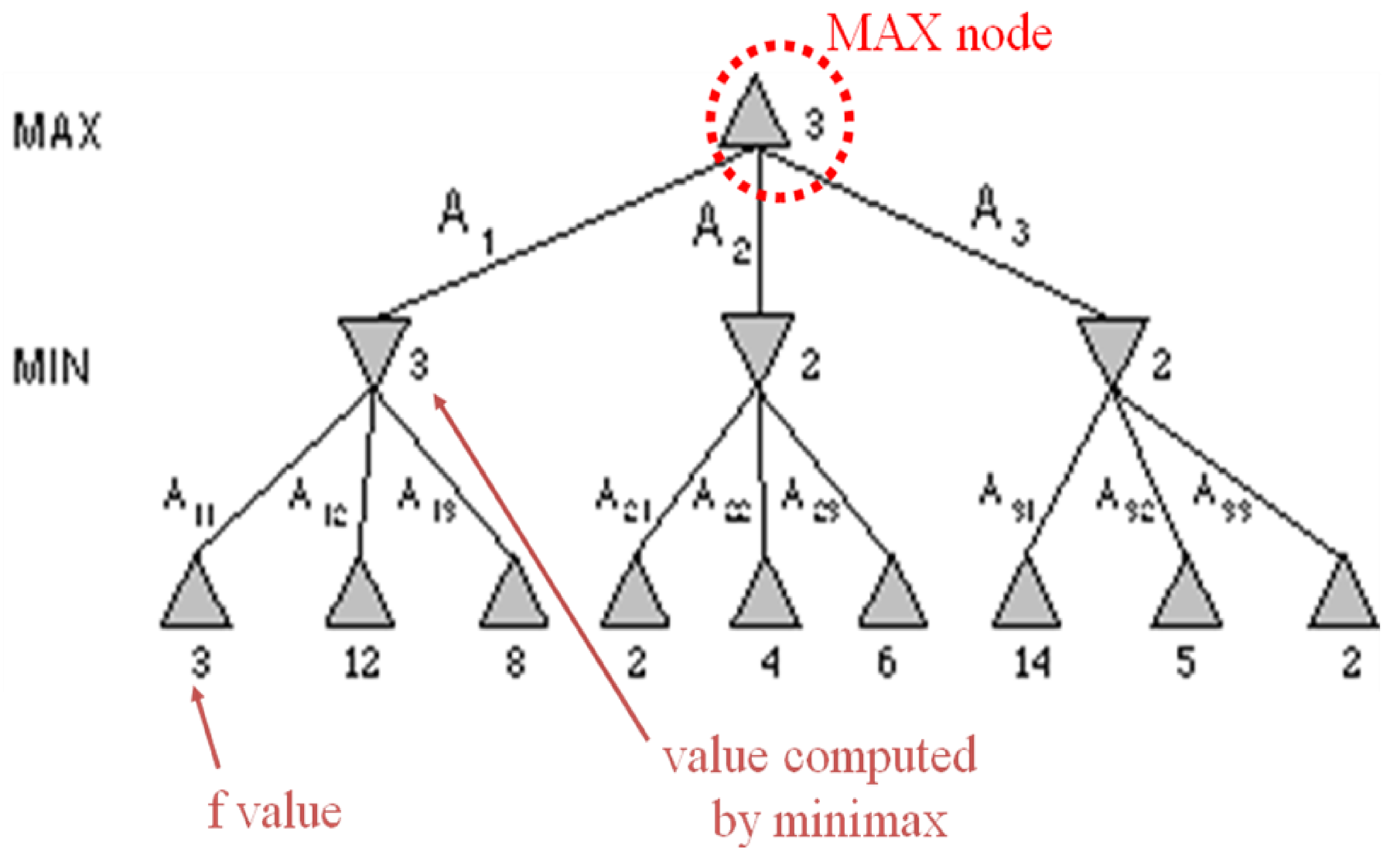
MINIMAX Procedure

- Create start node as a MAX node with current board configuration
- Expand nodes down to some depth (i.e., ply) of look ahead in the game.
- Apply the evaluation function at each of the leaf nodes
- Obtain the “back up" values for each of the non-leaf nodes from its children by Minimax rule until a value is computed for the root node.
- Pick the operator associated with the child node whose backed up value determined the value at the root as the move for MAX

Comments on MINIMAX search

- The search is **depth-first** with the given depth as the limit.
  - Time complexity: O(b^d)
  - Linear space complexity
- Performance depends on
  - Quality of evaluation functions (domain knowledge)
  - Depth of the search (computer power and search algorithm)
- Different from ordinary state space search
  - Not to search for a solution but for one move only
  - No cost is associated with each arc
  - MAX does not know how MIN is going to counter each of his moves
- MINIMAX rule is a basis for other game tree search algorithms

MINIMAX Tree
(Zero sum Game)



Alpha Beta Pruning

- The process of eliminating a branch of the search tree from consideration without examining is called pruning the search tree
- To improve the efficiency of MINIMAX procedure, the ‘branch and bound’ strategy is applied.
- It includes two bounds, one for each of the players.
- It requires the maintenance of two threshold values,
  - one representing a lower bound on the value that a maximizing node may ultimately be assigned

    (known as *Alpha*, it **can never decrease**; it can only go up)
  - Another representing an upper bound on the value that minimizing node may be assigned

    (known as *Beta*, it **can never increase**; it can only go down).
- It is now possible to compute the correct MINIMAX decision without looking at every node in the search tree.

- *Alpha-Beta pruning* returns the same move as MINIMAX would, but prunes away branches that cannot possibly influence the final decision.

Working of Alpha Beta Pruning

- Consider a node *n* somewhere in the tree such that player has a choice of moving to that node.
- If player has a better choice *m* either at the parent node of *n*, or at any choice point further up, then *n* will never be reached in actual play.
- Once we have found out enough about *n* (by examining some of its descendants) to reach this conclusion, we can prune it.

Alpha-Beta Pruning Working:

- Traverse the search tree in depth-first order
- At each **Max** node n, **alpha(n)** = maximum value found so far
  - Start with -infinity and only increase
  - Increases if a child of n returns a value greater than the current alpha
  - Serve as a tentative lower bound of the final pay-off
- At each **Min** node n, **beta(n)** = minimum value found so far
  - Start with infinity and only decrease
  - Decreases if a child of n returns a value less than the current beta
  - Serve as a tentative upper bound of the final pay-off
- **Alpha Pruning**: Given a Max node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(n)

  (alpha increases and passes beta from below)
- **Beta Pruning:** Given a Min node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if beta(n) <= alpha(n)

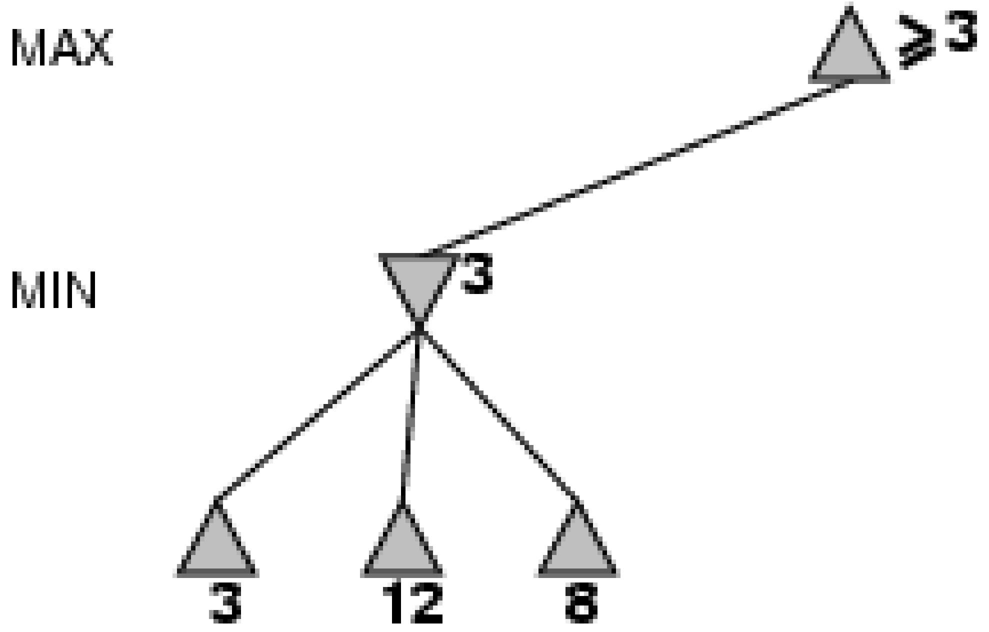
  (beta decreases and passes alpha from above)
- Carry alpha and beta values down during search

Pruning occurs whenever alpha >= beta

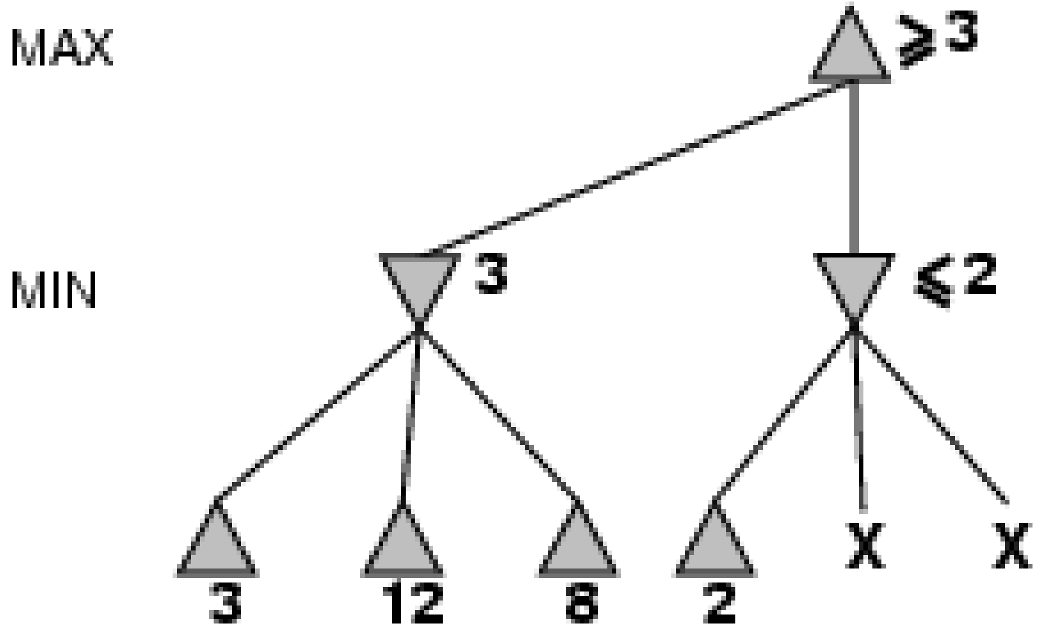
Alpha-Beta Pruning Algorithm: V(J; α, β)

- If J is terminal, return V(J)=h(J)
- If J is a MAX node:
  - For each successor J_k of J in succession:
    - Set α = max { α, V(Jk; α, β) }
    - If α >= β then return β, else continue.
  - Return α
- If J is a MIN node:
  - For each successor J_k of J in succession:
    - Set β = max { β, V(Jk; α, β) }
    - If α >= β then return α, else continue.
  - Return β.

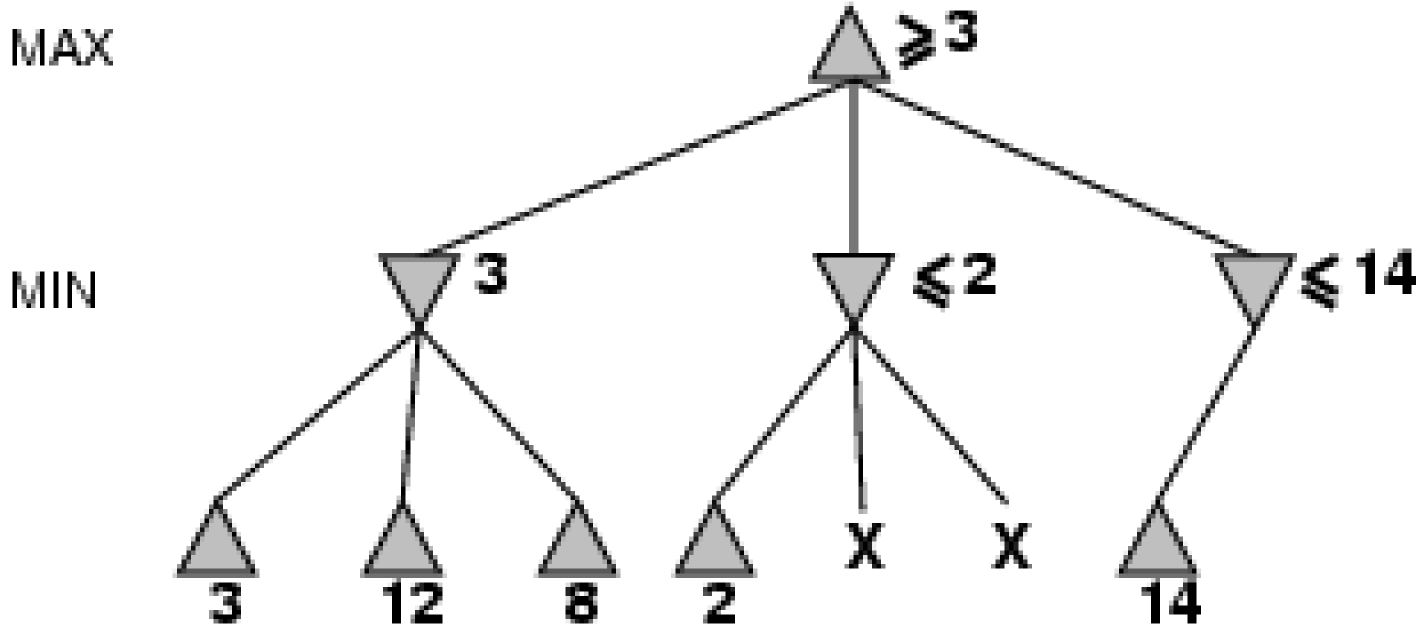
α-β pruning example



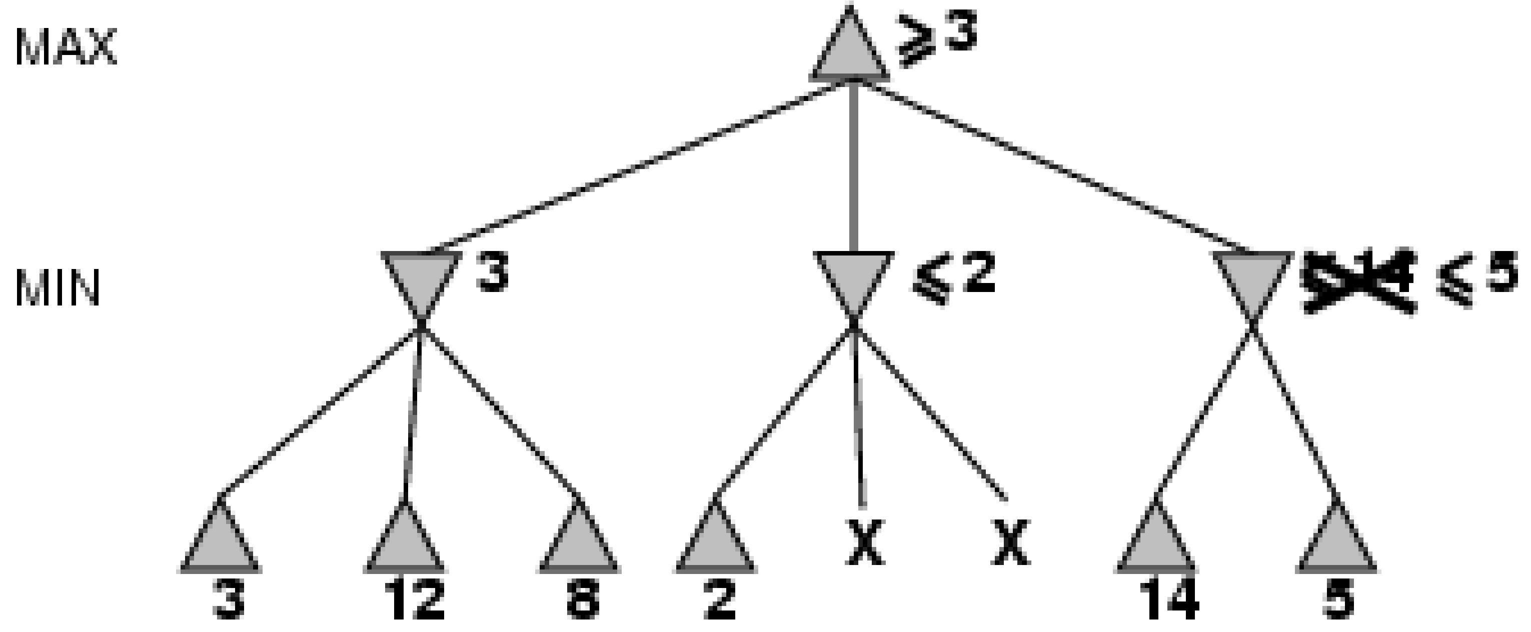
α-β pruning example



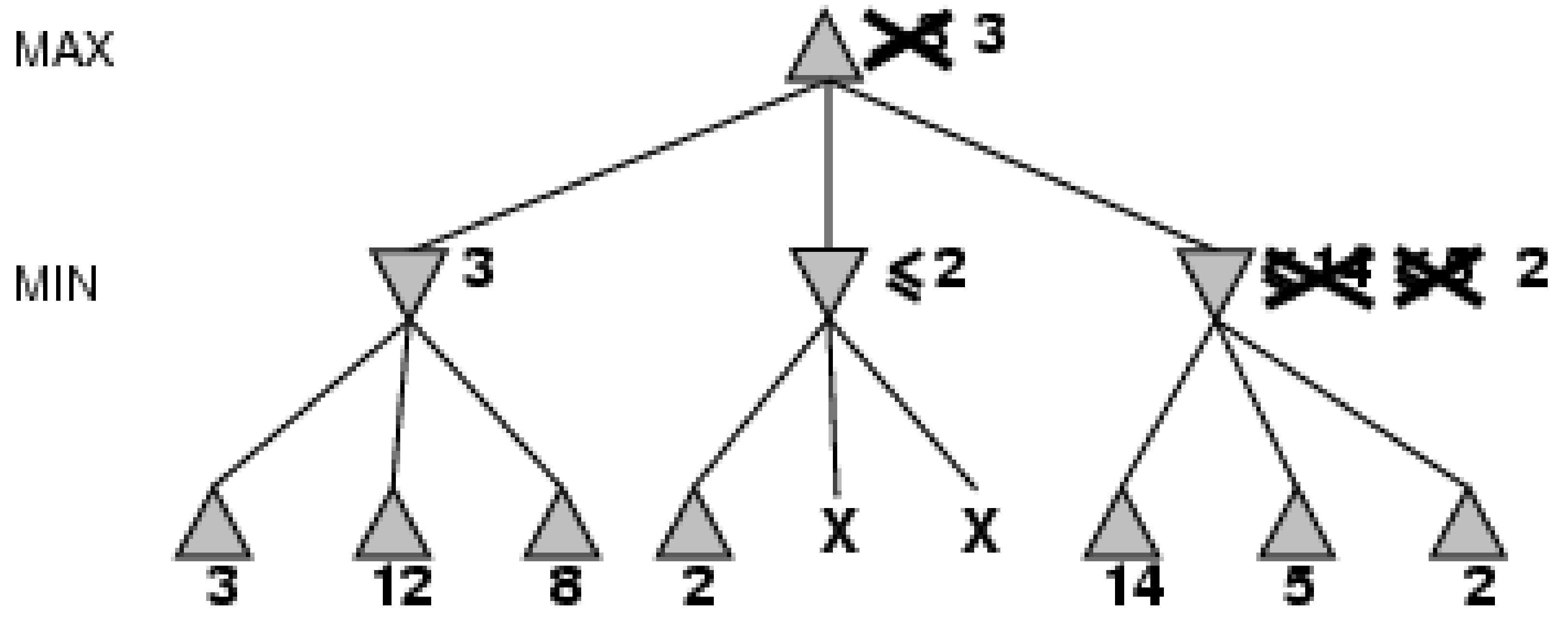
α-β pruning example



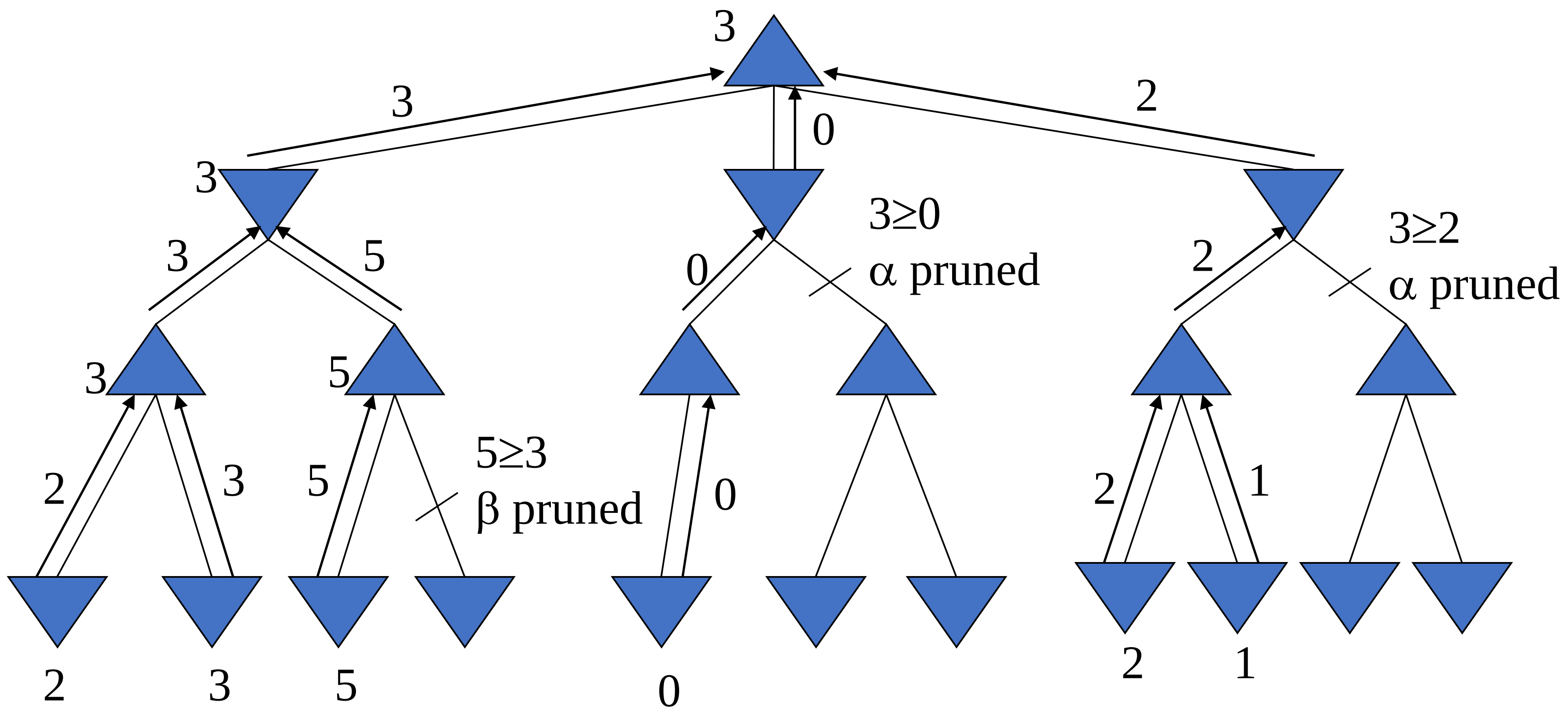
α-β pruning example



α-β pruning example



Alpha-Beta Pruning Example


3
3
0
2
3
3
5
3≥0
α pruned
3≥2
α pruned
0
2
3
5
2
3
5
5≥3
β pruned
0
2
1
2
3
5
0
2
1