Search Controls

CPSC 433: Artificial Intelligence
Fall 2024

Jonathan Hudson, Ph.D.
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

August 8, 2024

UNIVERSITY OF CALGARY

Search Controls

Control Yourself!

Search Controls

General tasks:

- Determining all possible transitions, i.e.
  $\{(s_1,s_2) \in T \mid s_1 \text{ is actual state}\}$
- By selecting the next state

Transitions are usually based on applying general rules to parts of the actual state

Examples:
- extension rules in set-based search
- processing a leaf in tree- or graph-based search

Determining all possible transitions

Many general rules that were applicable in the last state usually are applicable in the current

Therefore

1. Have list of potential transitions from last state
2. Delete from list potential transitions not possible any more
3. Update remaining transitions if necessary (we are in new state)
4. Add newly possible transitions (that are not already in the list)

☞List of all candidates for next transition and let control K select one

Selecting the next state

Have to find best transition

☞ evaluation necessary

- Store evaluation with transition so that evaluation can be reused (but not always reusable, remember min-max search)
- Organize list of transitions as heap (priority queue!), since always the transition with best evaluation is looked for
  - Finding best transition takes constant time
  - Inserting new transitions much faster than in ordered list

Evaluating transitions

Candidates for measuring

- Result state
- Parts of actual state enabling general rule for transition
- Parts new in the result state vs actual state

What to use?

Depends on how difficult it is to compute needed data

(i.e. resulting state resp. parts)

General Ideas for What to Measure

- Distance to a goal state or parts of it
- Best that can be achieved from a state (using an approximation, used for optimization problems)
- Difficulty of new problems in state (needs knowledge about problems)
- Number of transitions that become possible
- Size of state
- History of search
- Use of similar search experiences

General Problems (and solution approaches)

- States get too big
  ☞ local search, backtracking, forget history

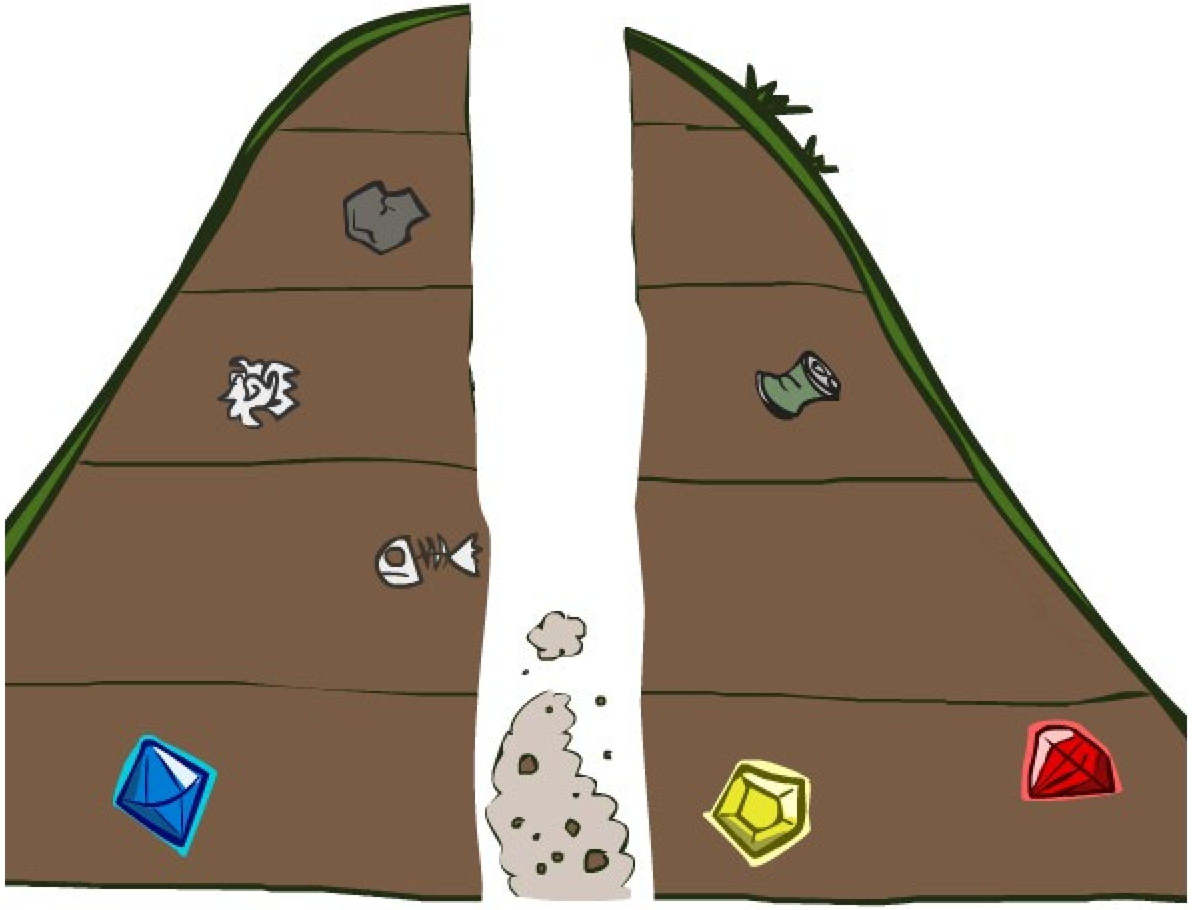
- Measuring states too time consuming
  ☞ abstract to significant parts, use less complex measures

- Combining pieces of knowledge
  ☞ normalizing weights + weighted sums

- Contradicting control knowledge
  ☞ distributed search approaches, competition

Simple Tree Search Controls

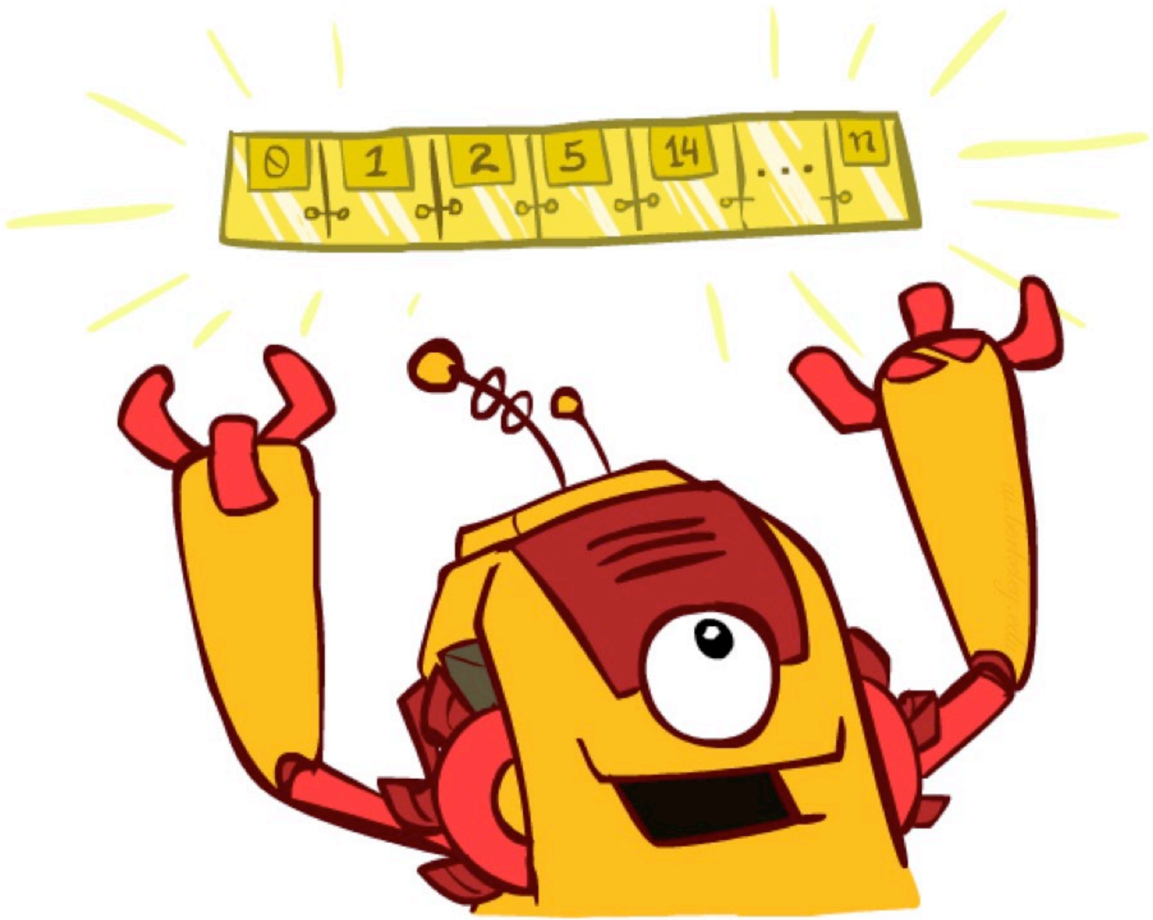
Search Algorithm Properties



Queueing

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 actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object



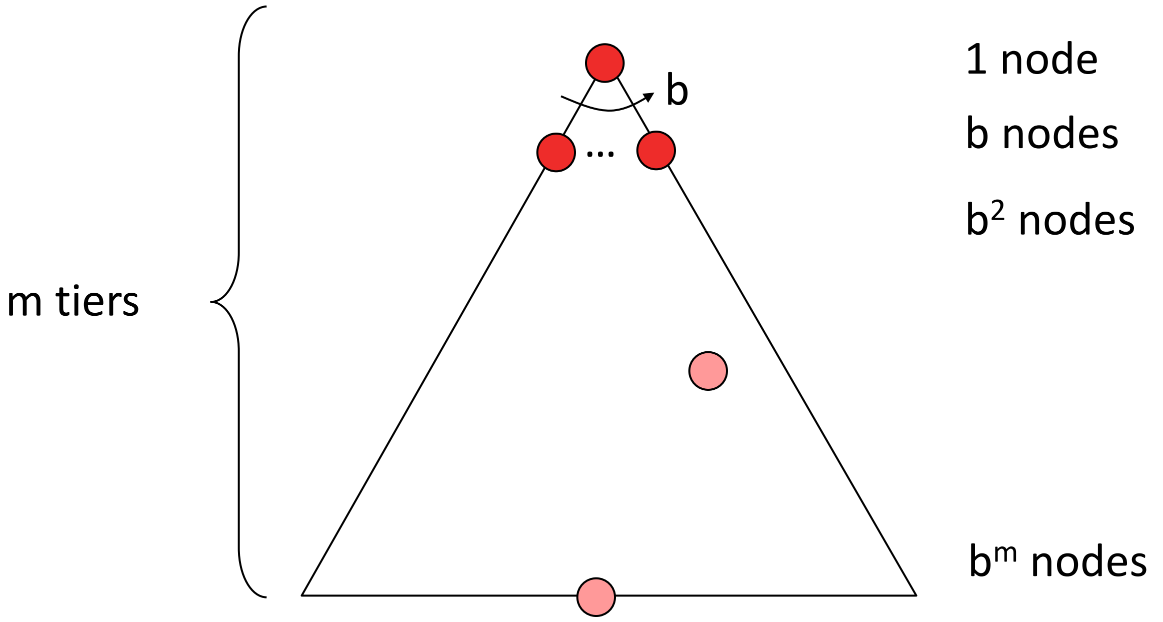
UNIVERSITY OF CALGARY

Properties

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

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



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

Search Algorithm Properties

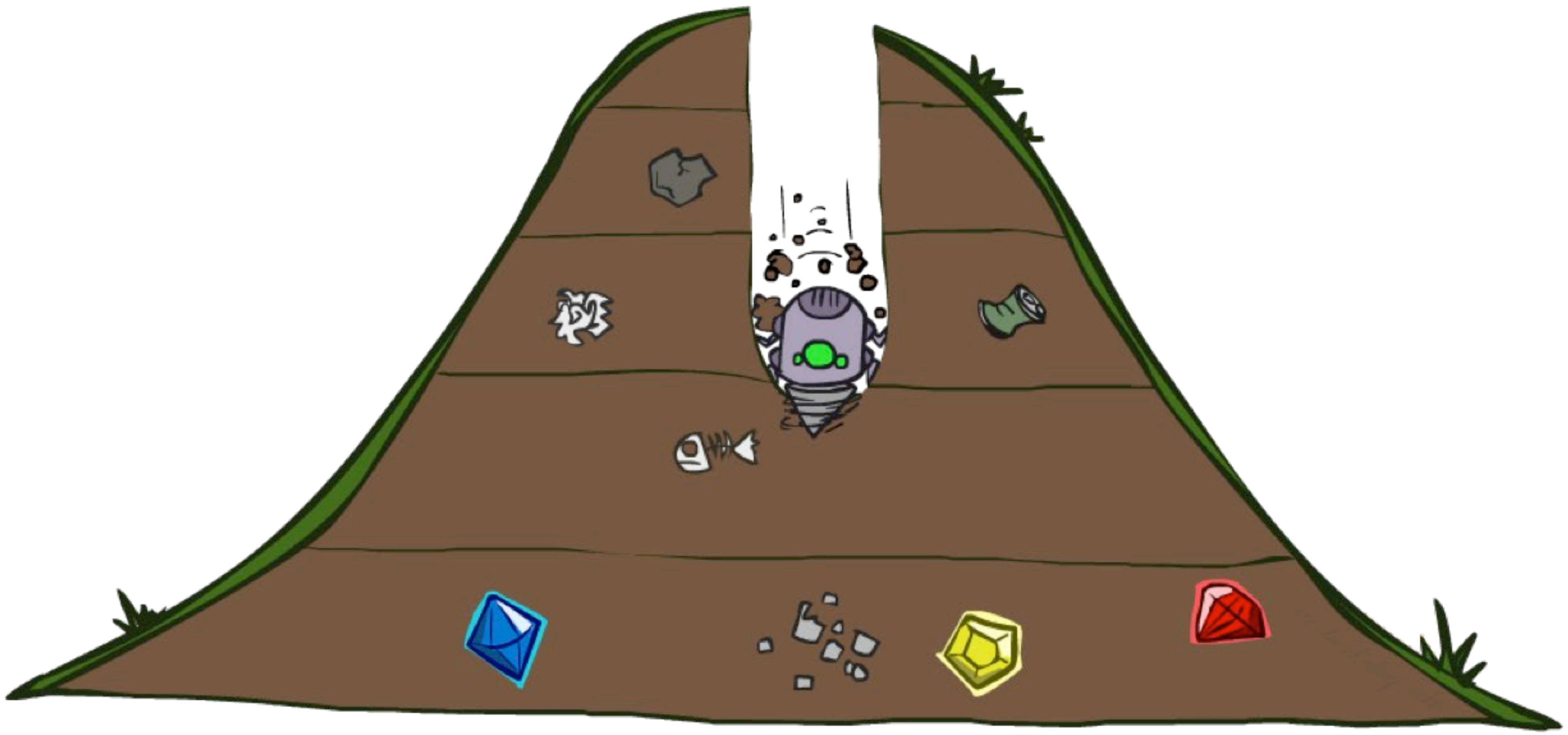
- Reminder we have two types of trees
  - And-trees
    - Need all leafs yes
  - Or-trees
    - Need one leaf as yes

- The following performance of tree search controls will be examining **OR-TREE** where we can end without exploring the whole tree.
  - And-trees gain more from pruning (bounding by f_{bound}), usually the best pairing is a some variant of depth preferring search (to find good bounds, followed by deep exploration that can now be pruned well), so one of the variants of DFS that will follow

UNIVERSITY OF CALGARY

DFS

Depth-First Search

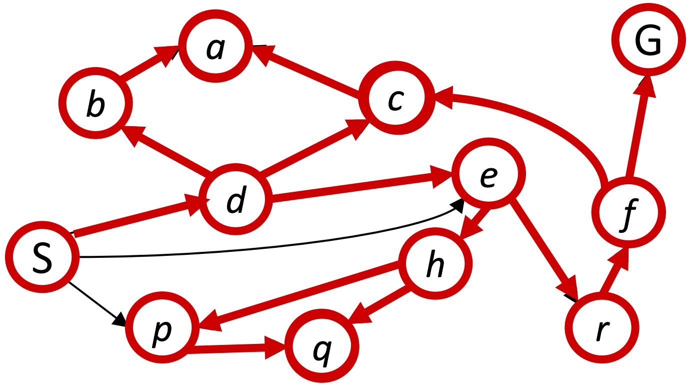


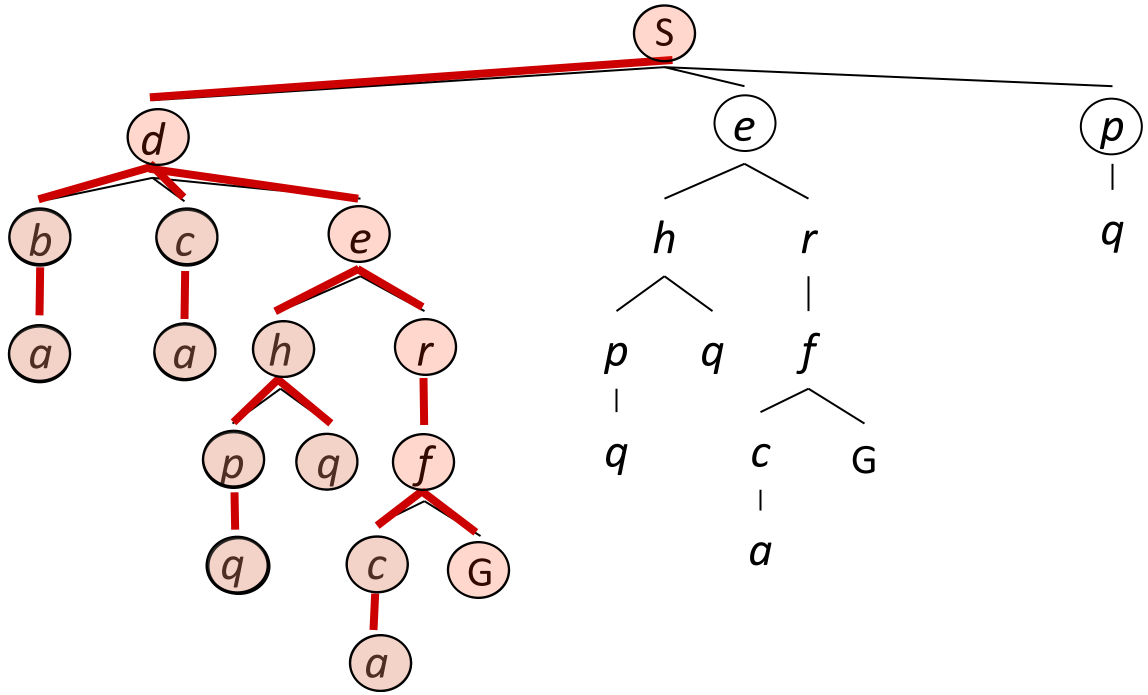
UNIVERSITY OF CALGARY

Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

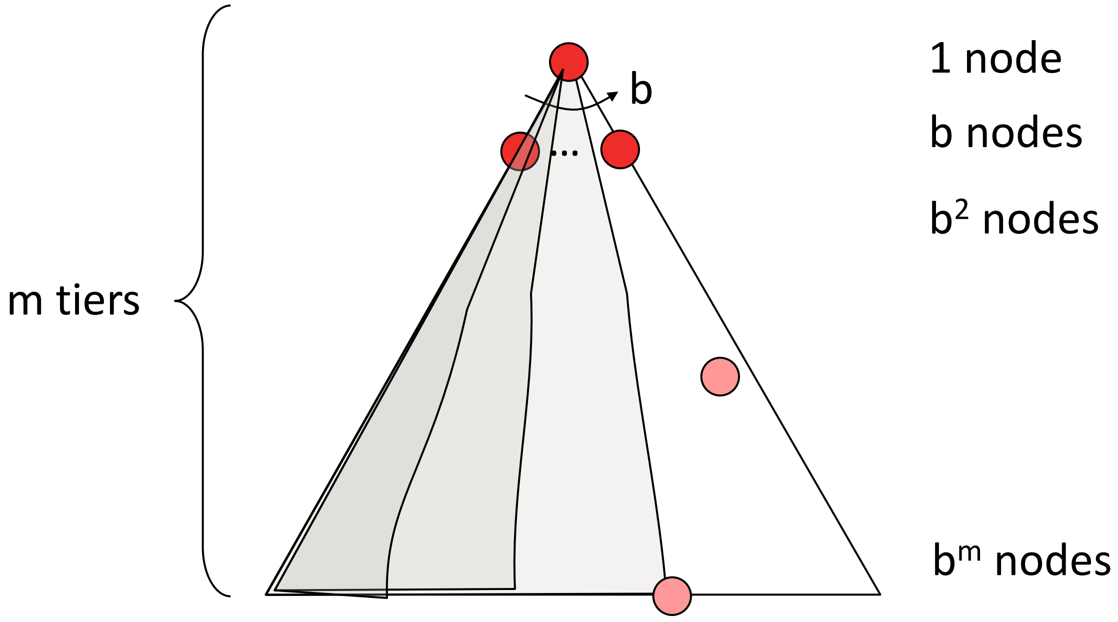




UNIVERSITY OF CALGARY

Depth-First Search (DFS) Properties (OR-tree)

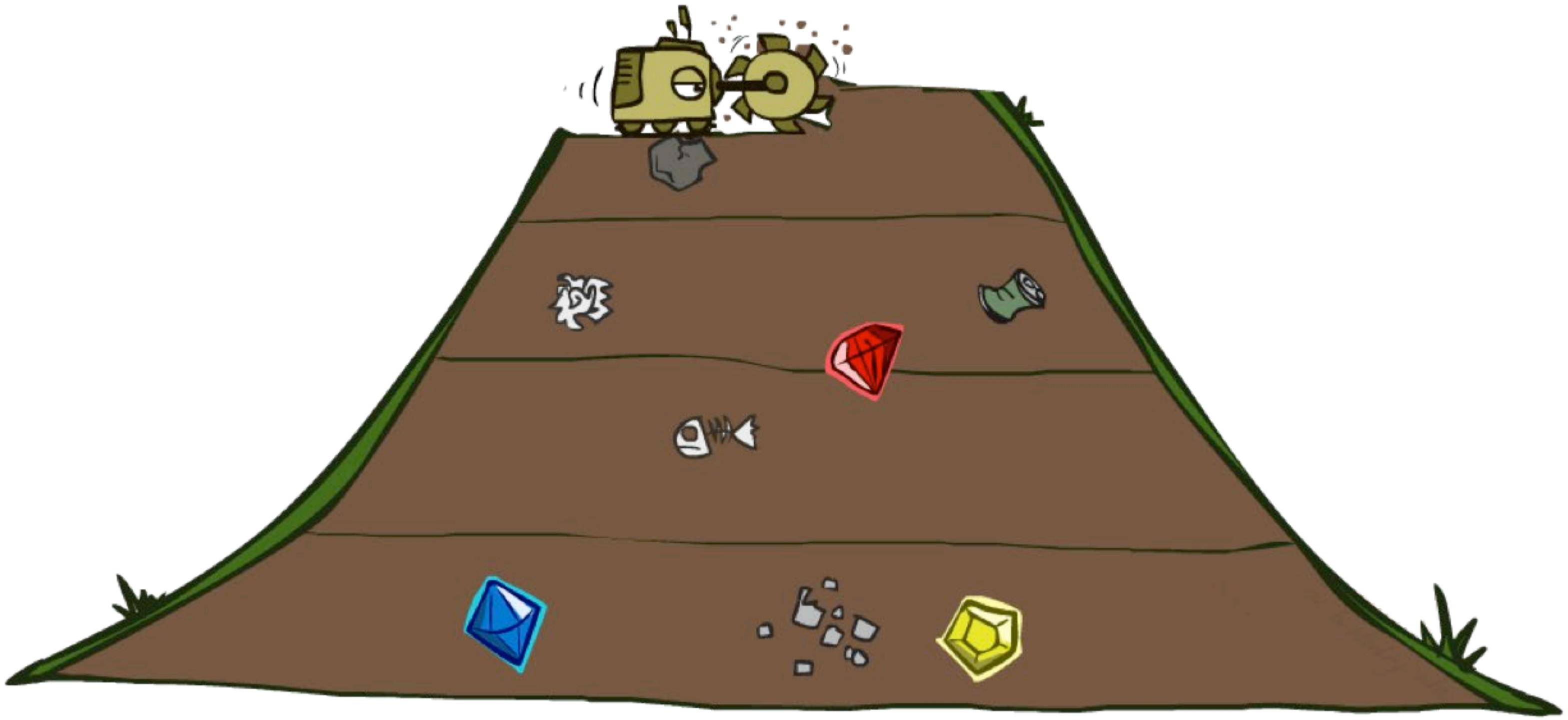
- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space for fringe (OR-tree)?
  - Only has siblings on path to root, so O(bm)
    - Full expansion down left to farthest leaf is length m, and for each tier there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



UNIVERSITY OF CALGARY

BFS

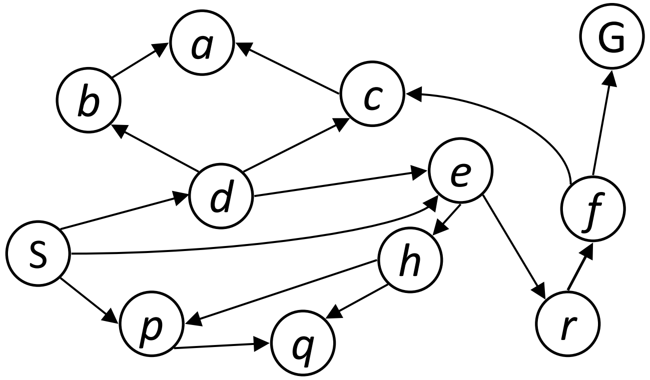
Breadth-First Search

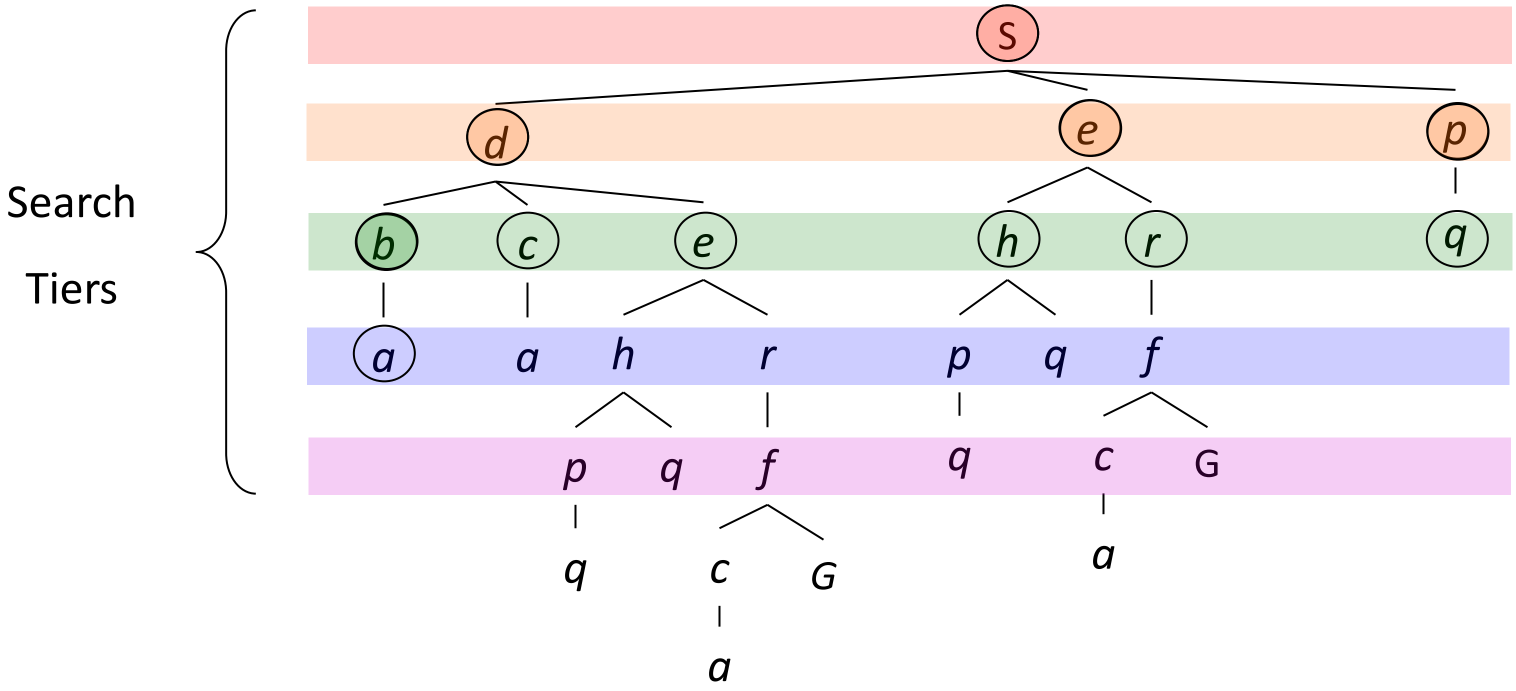


Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

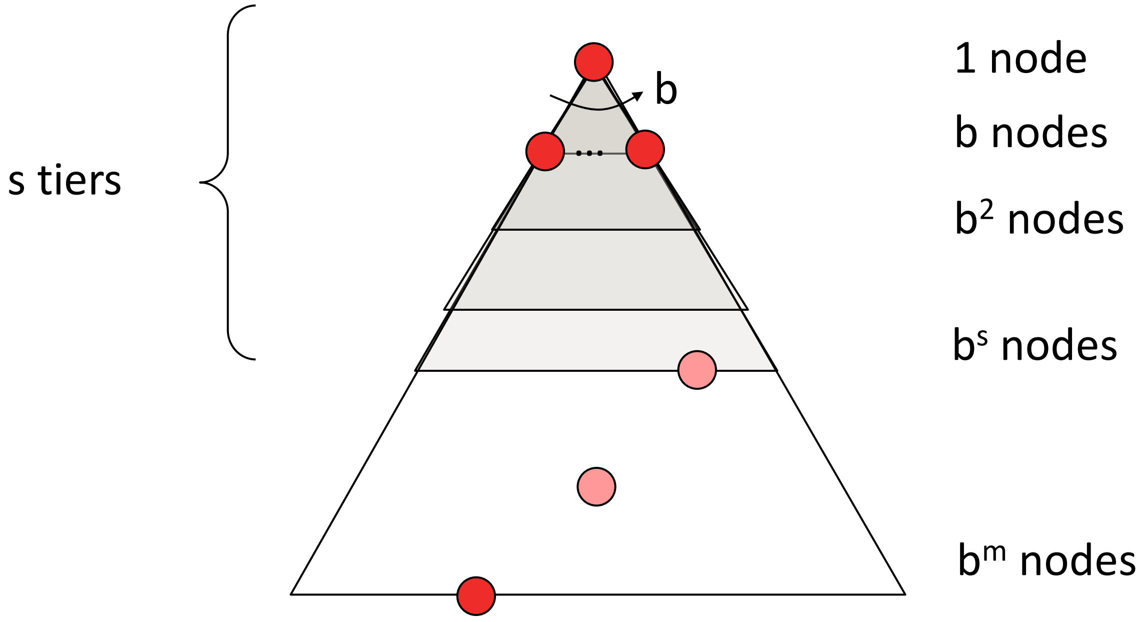




UNIVERSITY OF CALGARY

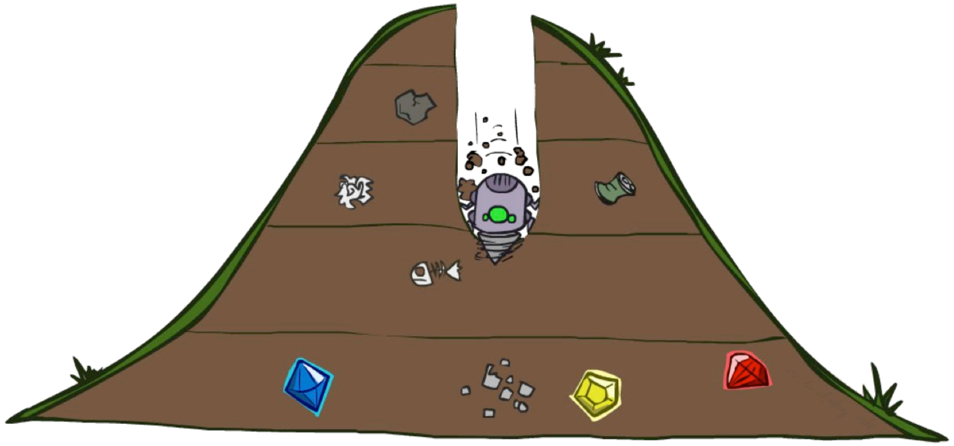
Breadth-First Search (BFS) Properties (OR-tree)

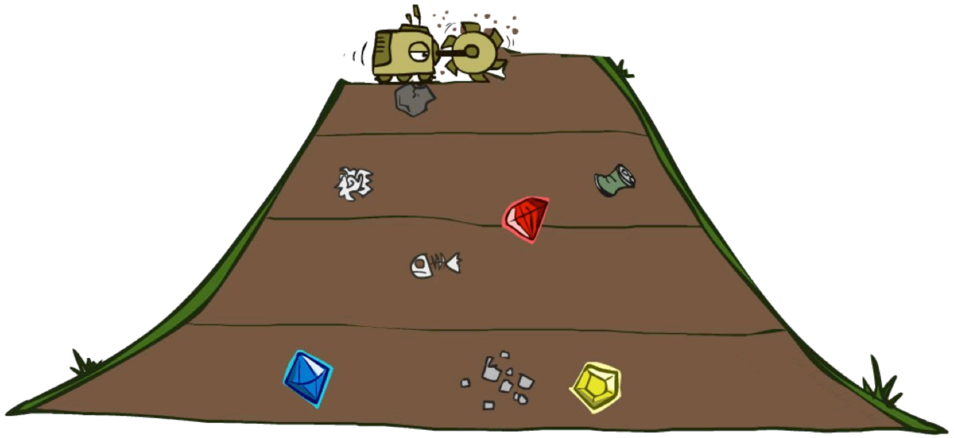
- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space for fringe (OR-tree)?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete (OR-tree)?
  - s must be finite if a solution exists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



UNIVERSITY OF CALGARY

DFS vs BFS

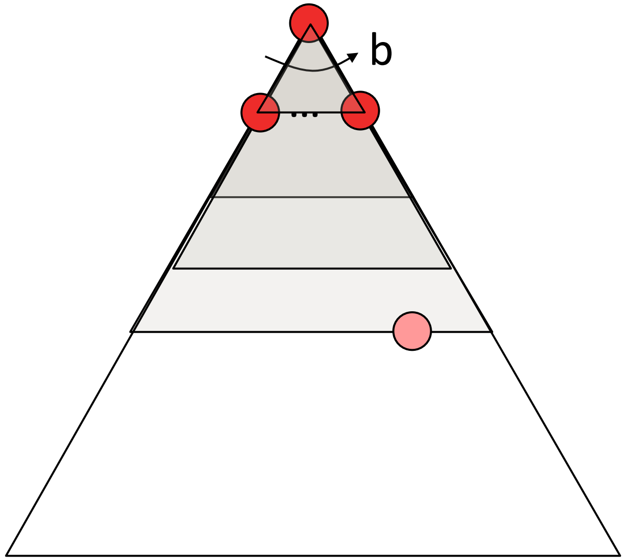




Iterative Deepening

Iterative Deepening

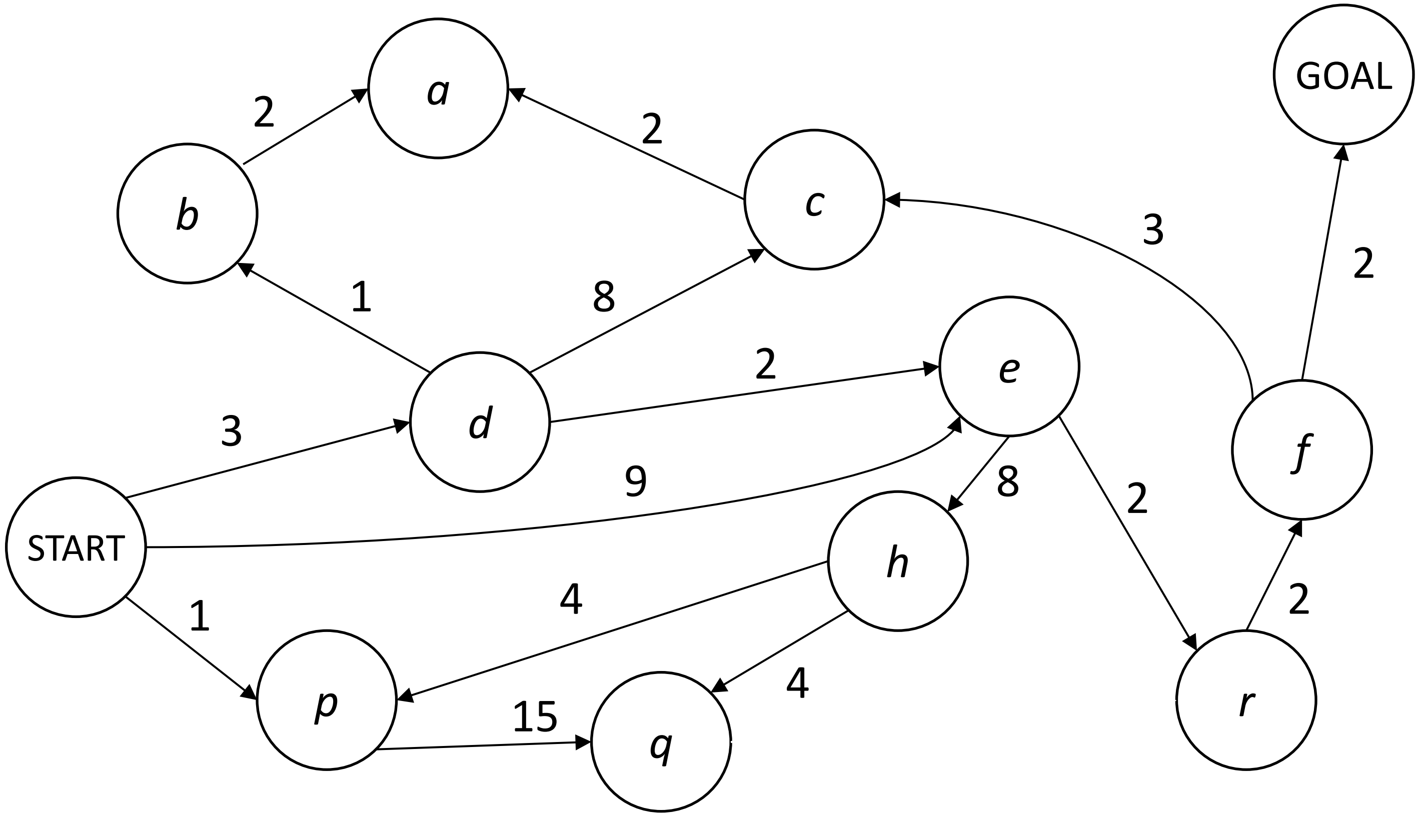
- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution dispose each DFS
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..



- Isn't that wastefully redundant?
  - If depth is reasonably shallow, not too bad
  - Generally, most work happens in the lowest level searched, so not so bad!

Costs!

Cost-Sensitive Search

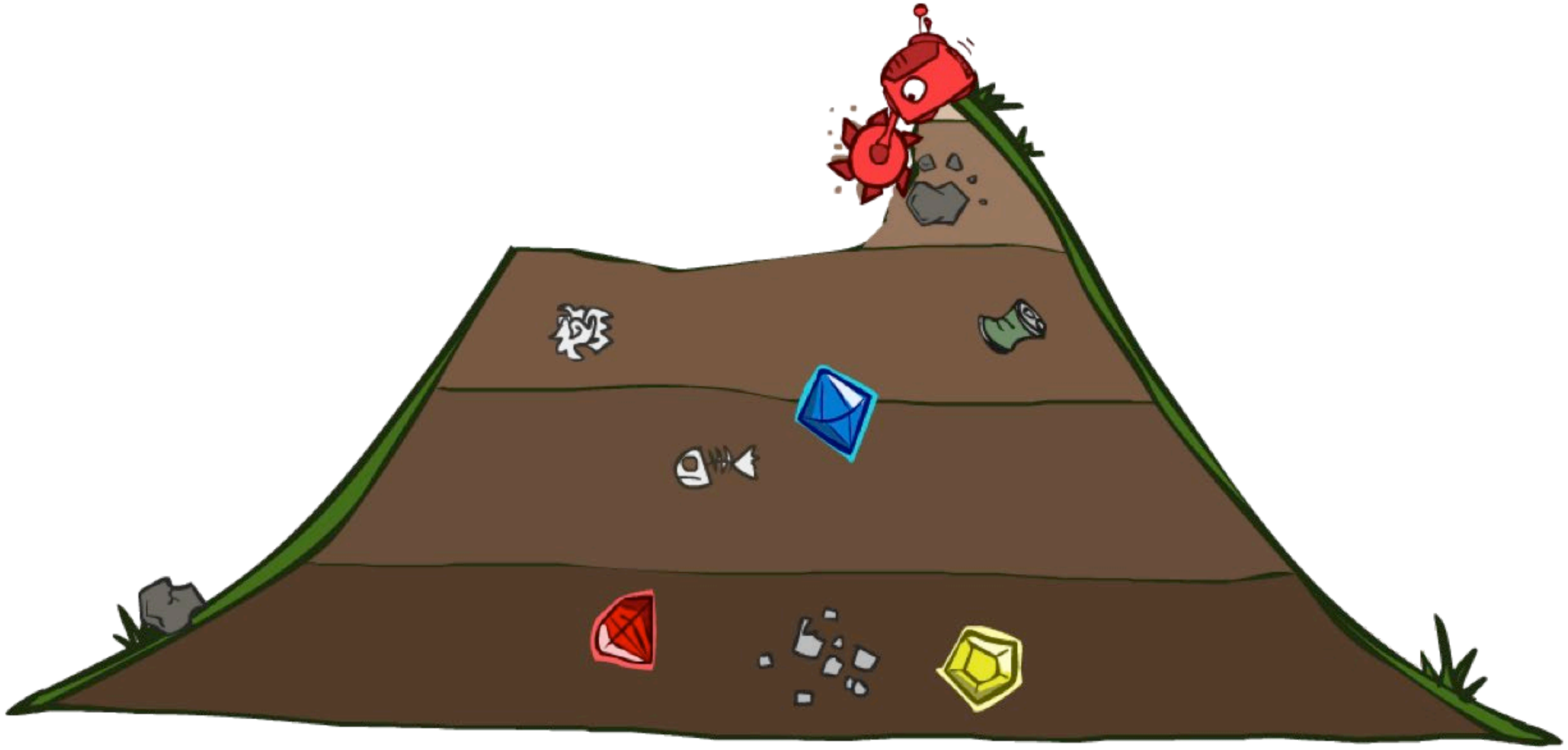


How?

UNIVERSITY OF CALGARY

Uniform Cost

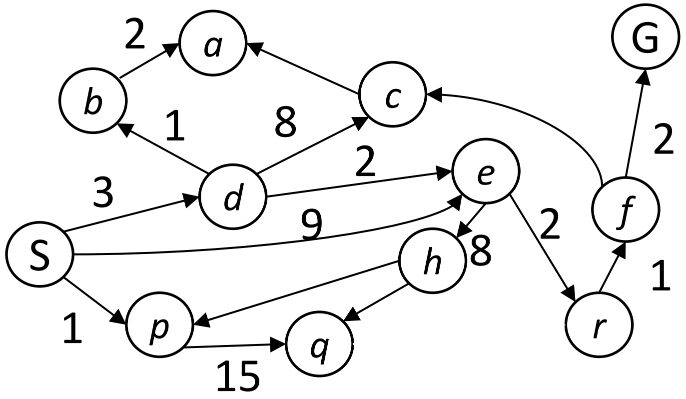
Uniform Cost Search



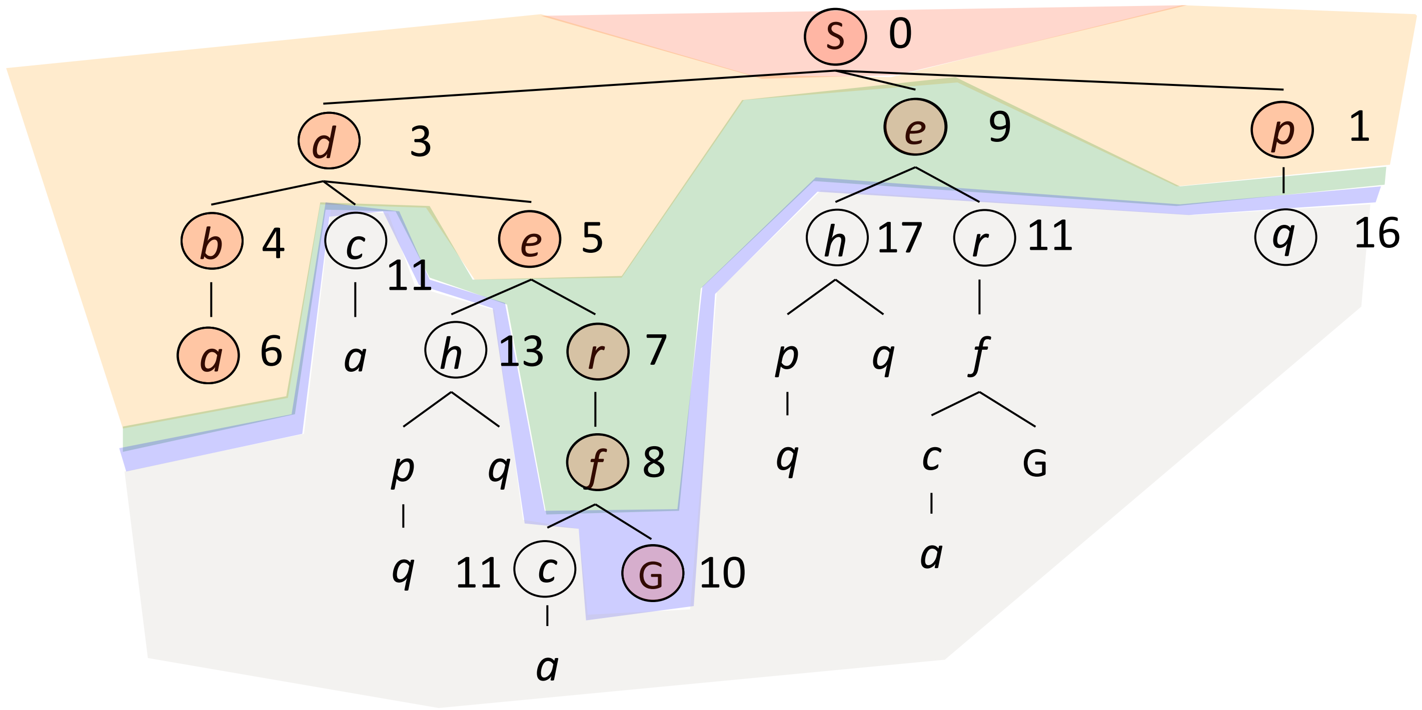
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



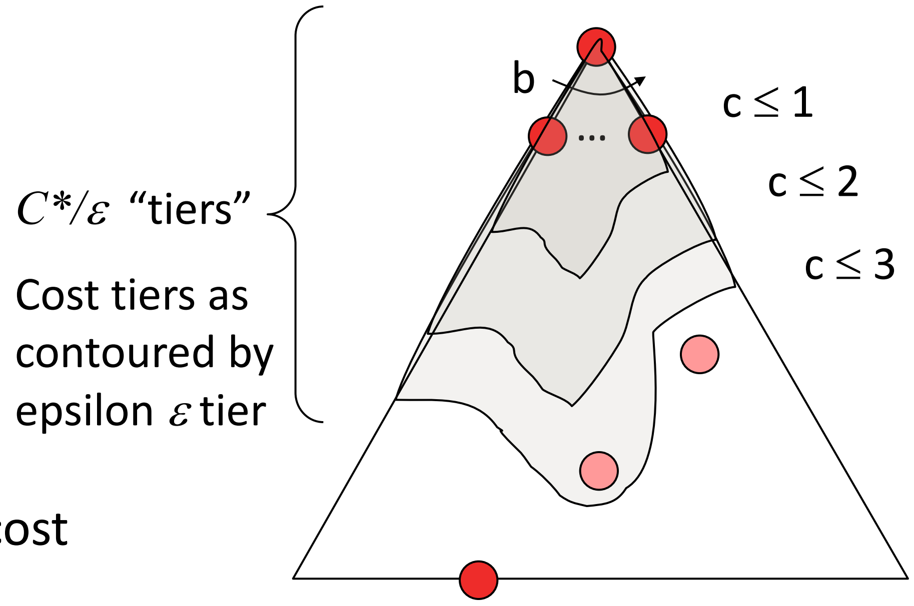
Cost contours



UNIVERSITY OF CALGARY

Uniform Cost Search (UCS) Properties (OR-tree)

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
- How much space for fringe (OR-tree)?
  - Has roughly the last tier, so O($b^{C^*/\varepsilon}$)
- Is it complete (OR-tree)?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal (OR-tree)?
  - Yes! (A*)

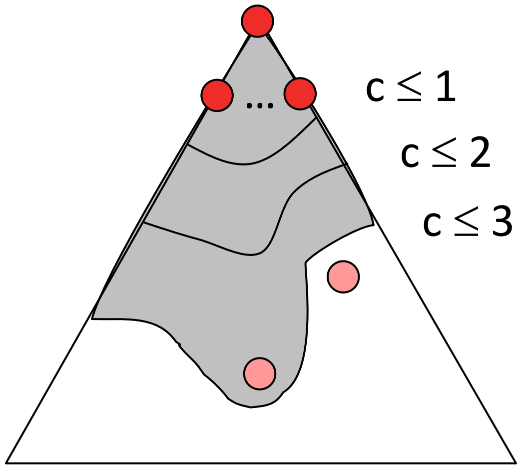


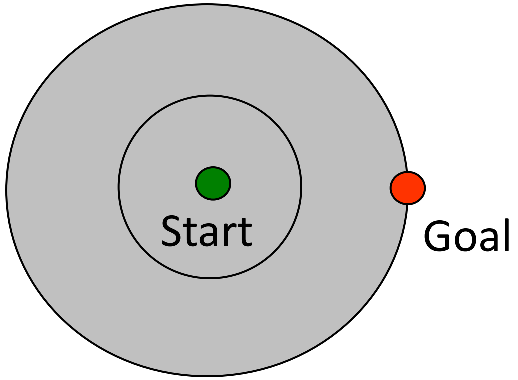
Uniform Cost Issues

- 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



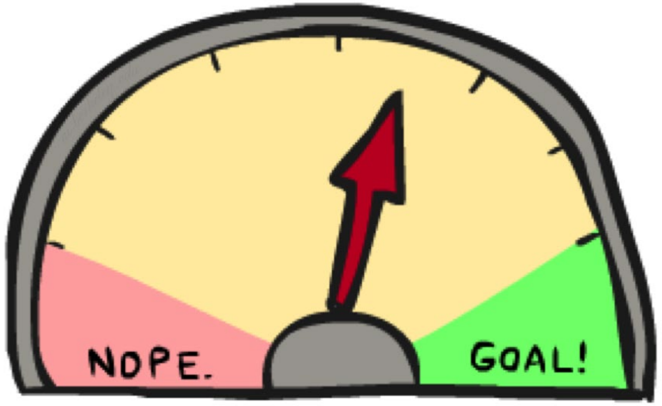


Informed Search

Informed Search

- Uninformed Search
  - DFS
  - BFS
  - UCS

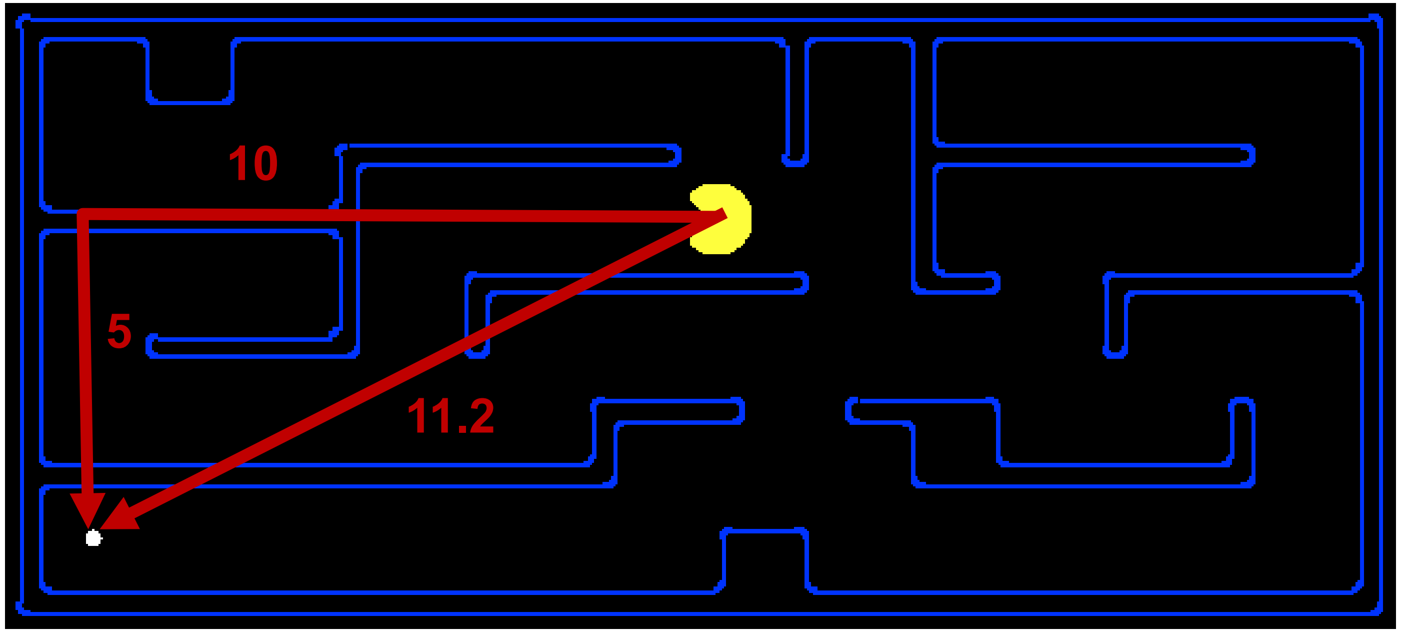
- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search

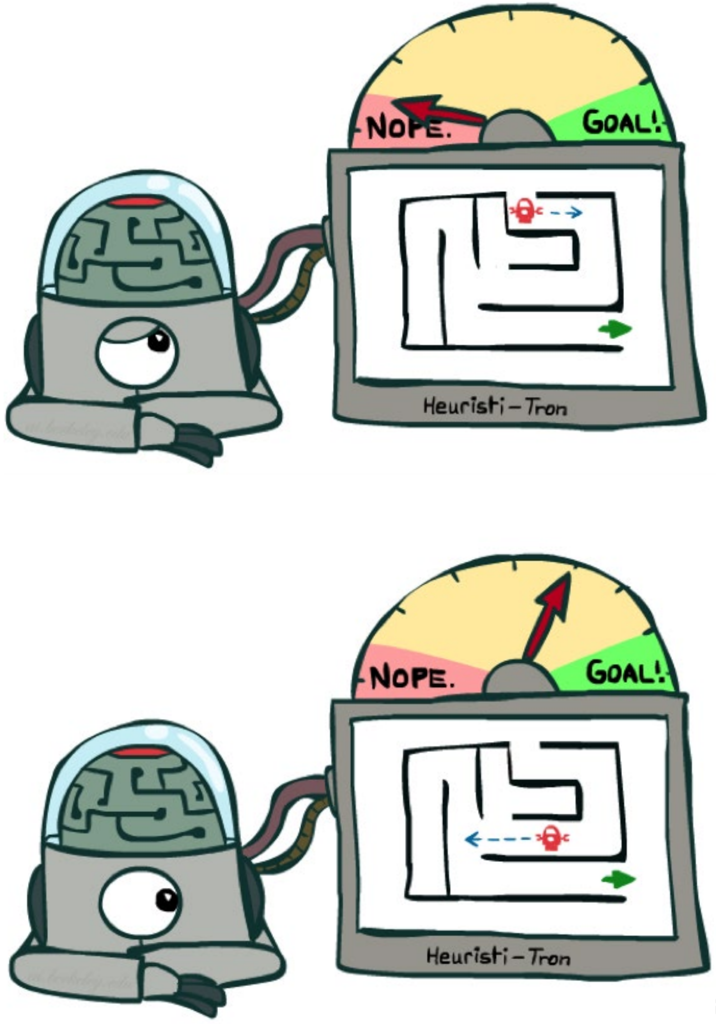


UNIVERSITY OF CALGARY

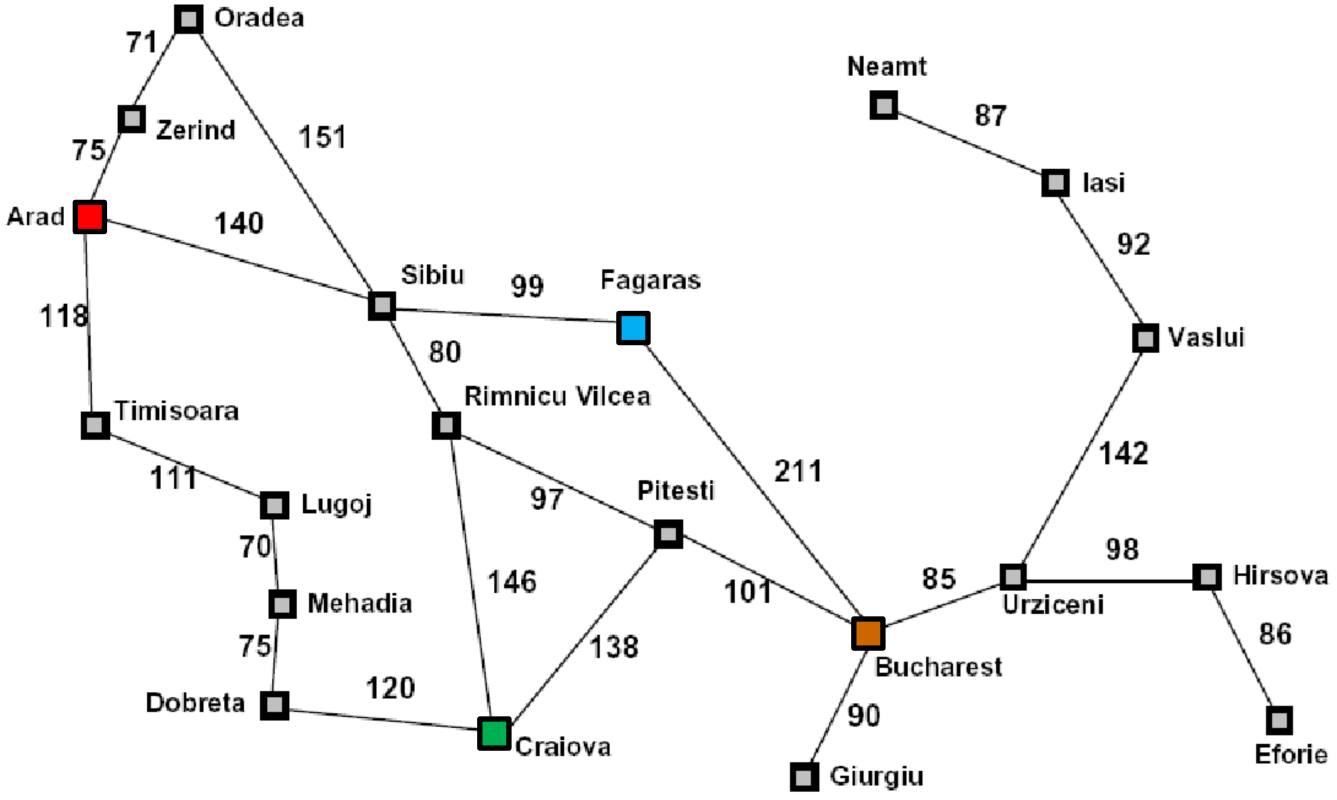
Search Heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Pathing?
  - 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

$h(x)$

Greedy Search

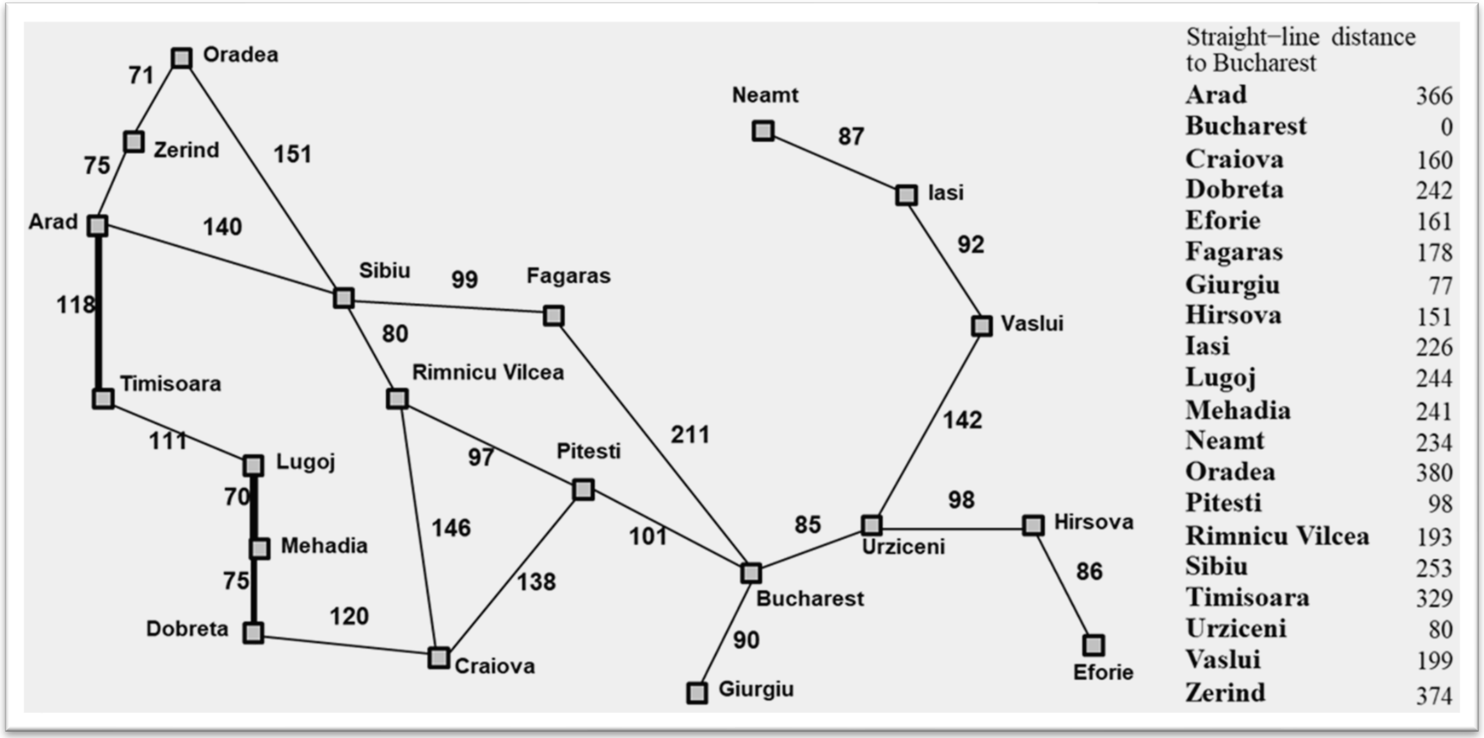
Greedy Search

- Evaluation function **h** (heuristic)
- Estimate value of node expansion to solution and perform it next
- Variant of uniform but costing is not heuristic and based on specific problem instance being explored
- Greedy search expands the node that appears to be closest to goal

Example: Romania

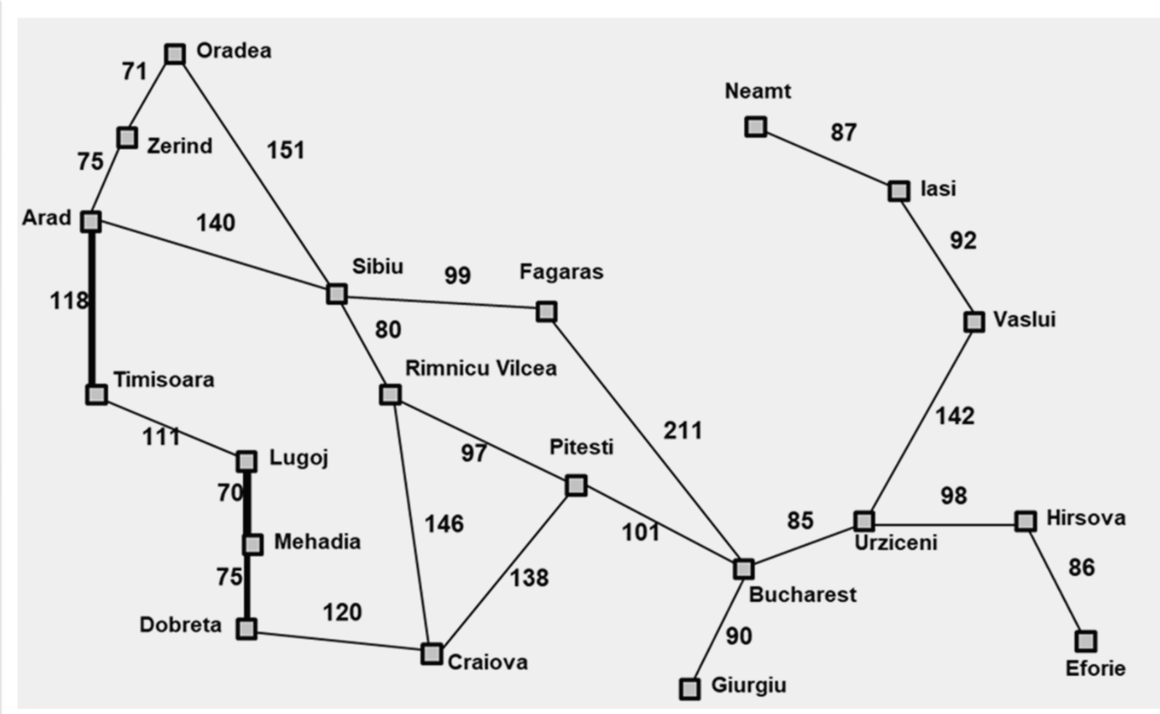
- Currently in Arad.
- Need to get to Bucharest

- Formulate goal:
  - be in Bucharest
- Formulate problem
  - states: various cities
  - actions: drive between cities
- Find solution
  - sequence of cities



Greedy search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

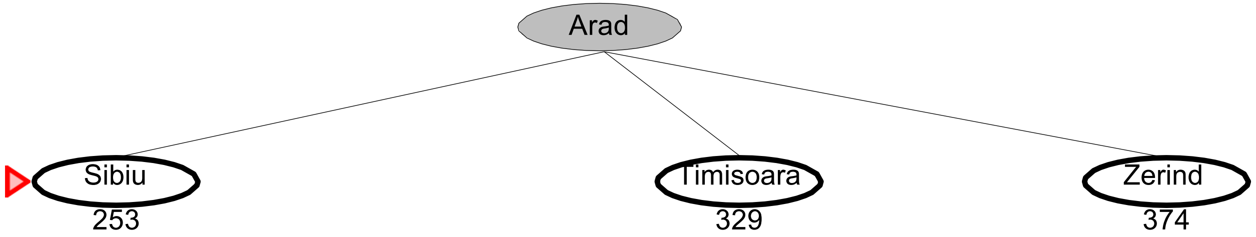


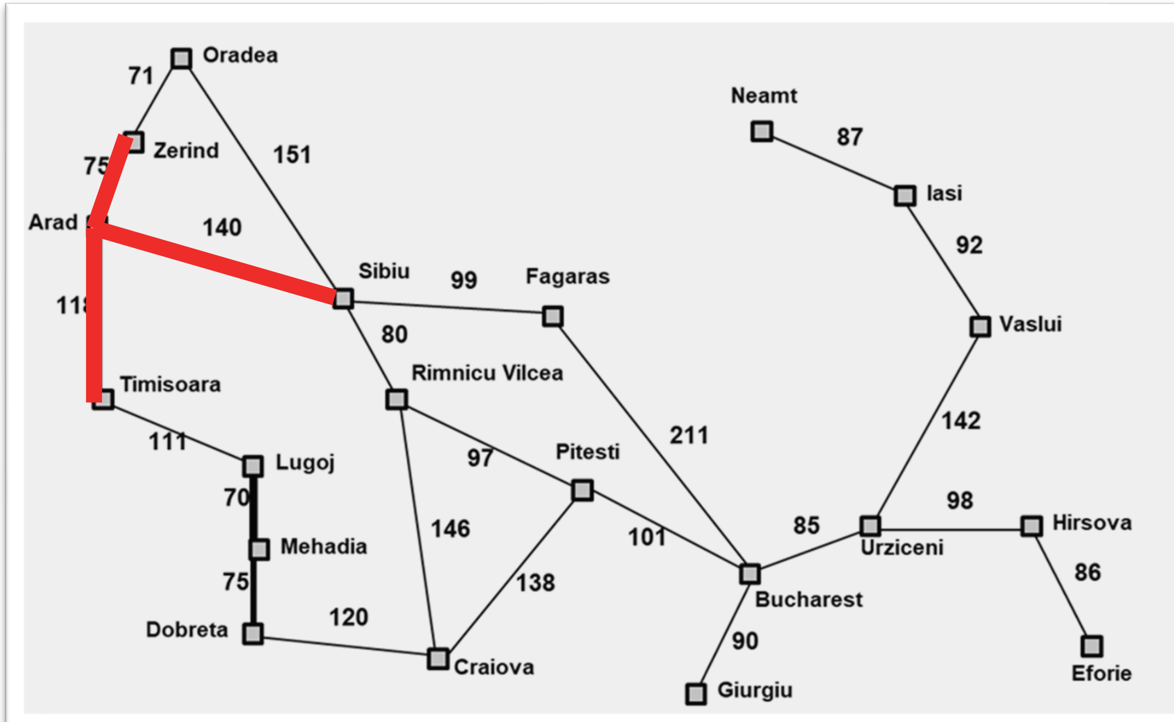
Straight−line distance to Bucharest

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

Greedy search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

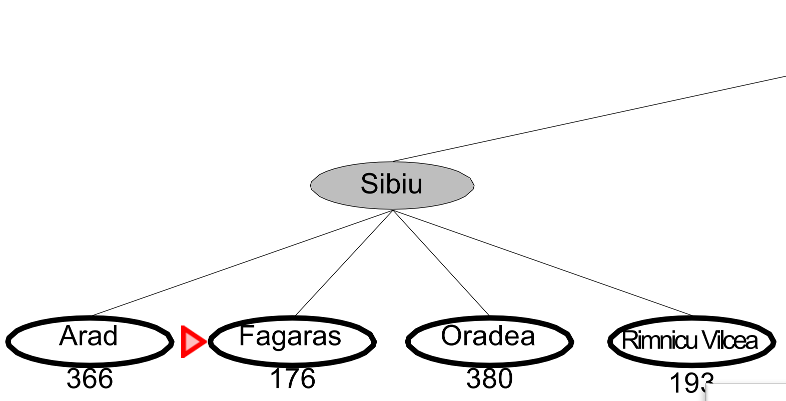


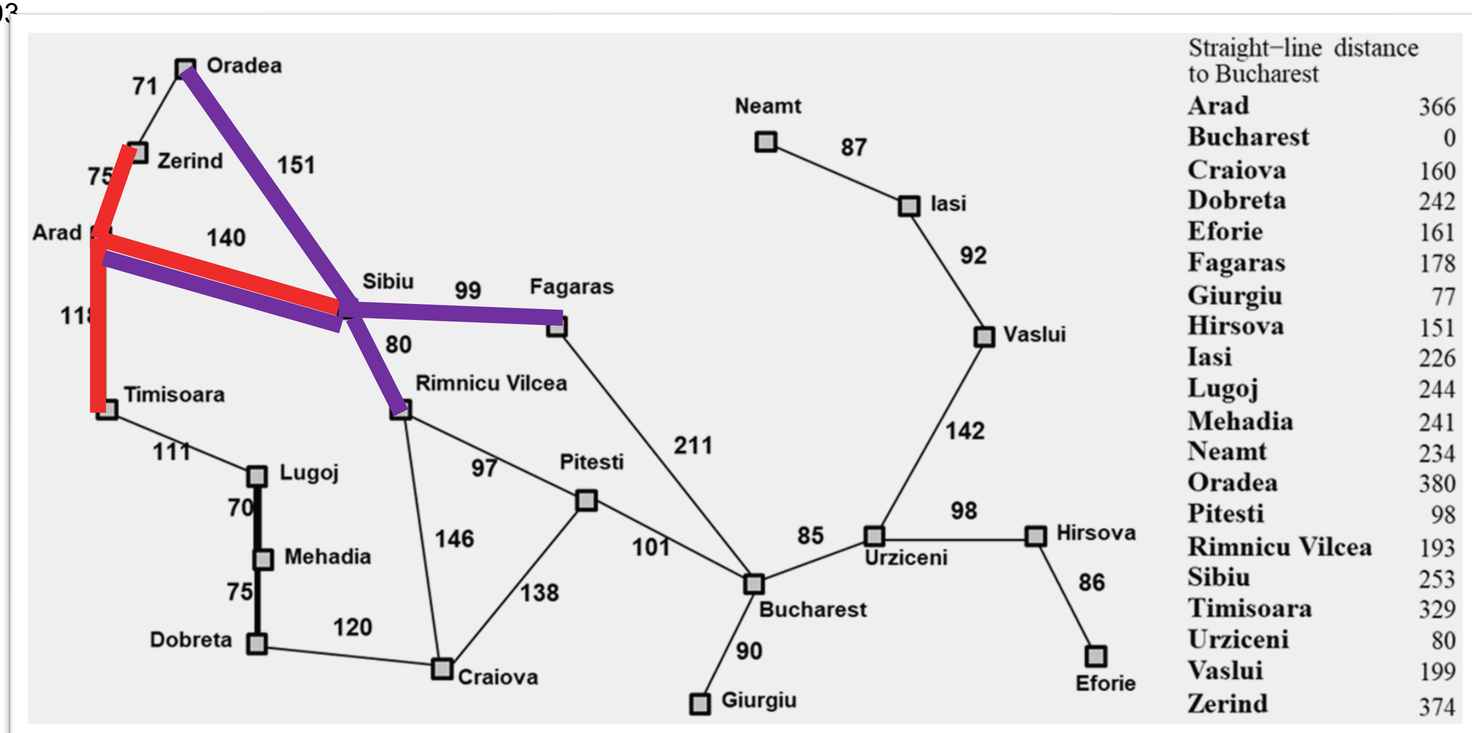


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example

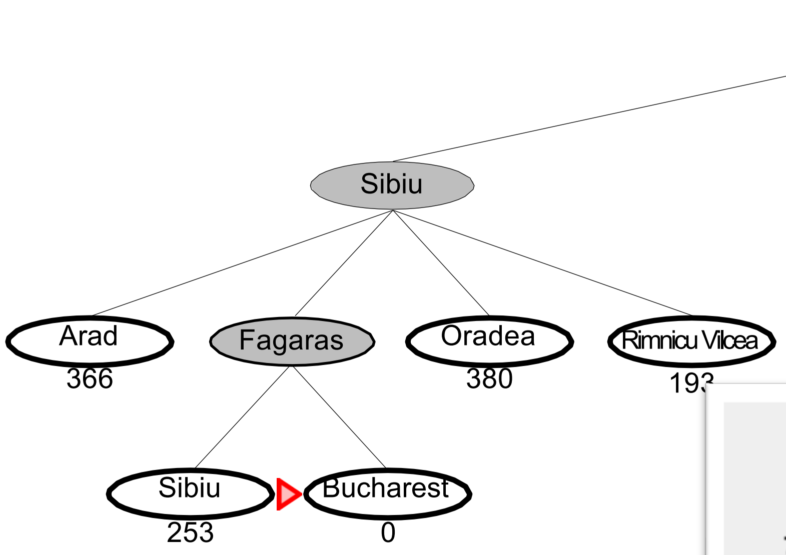
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

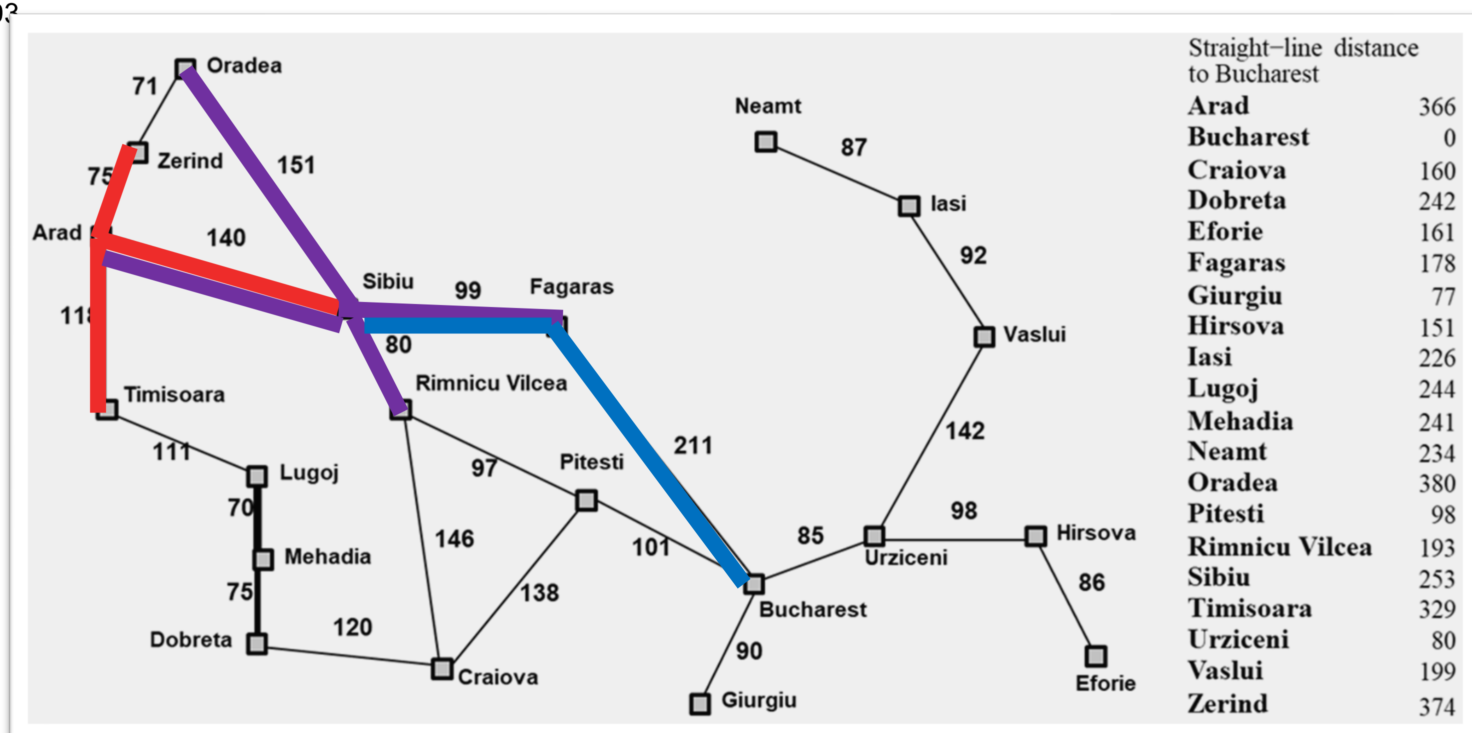




Greedy search example

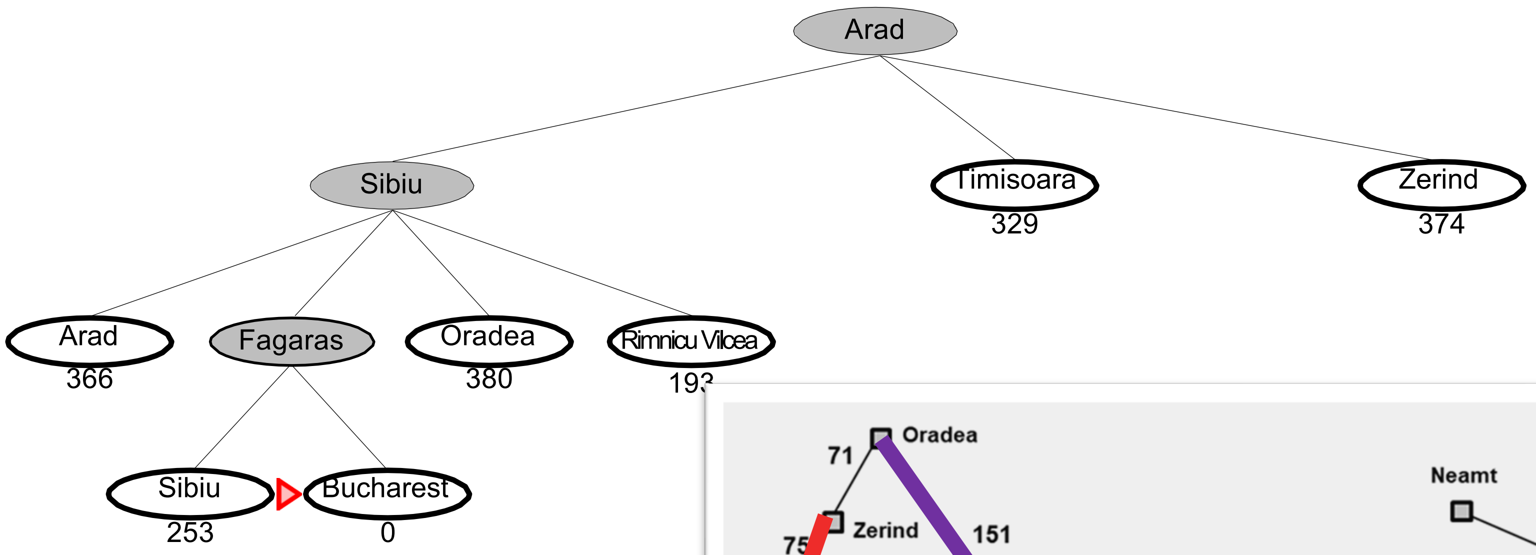
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

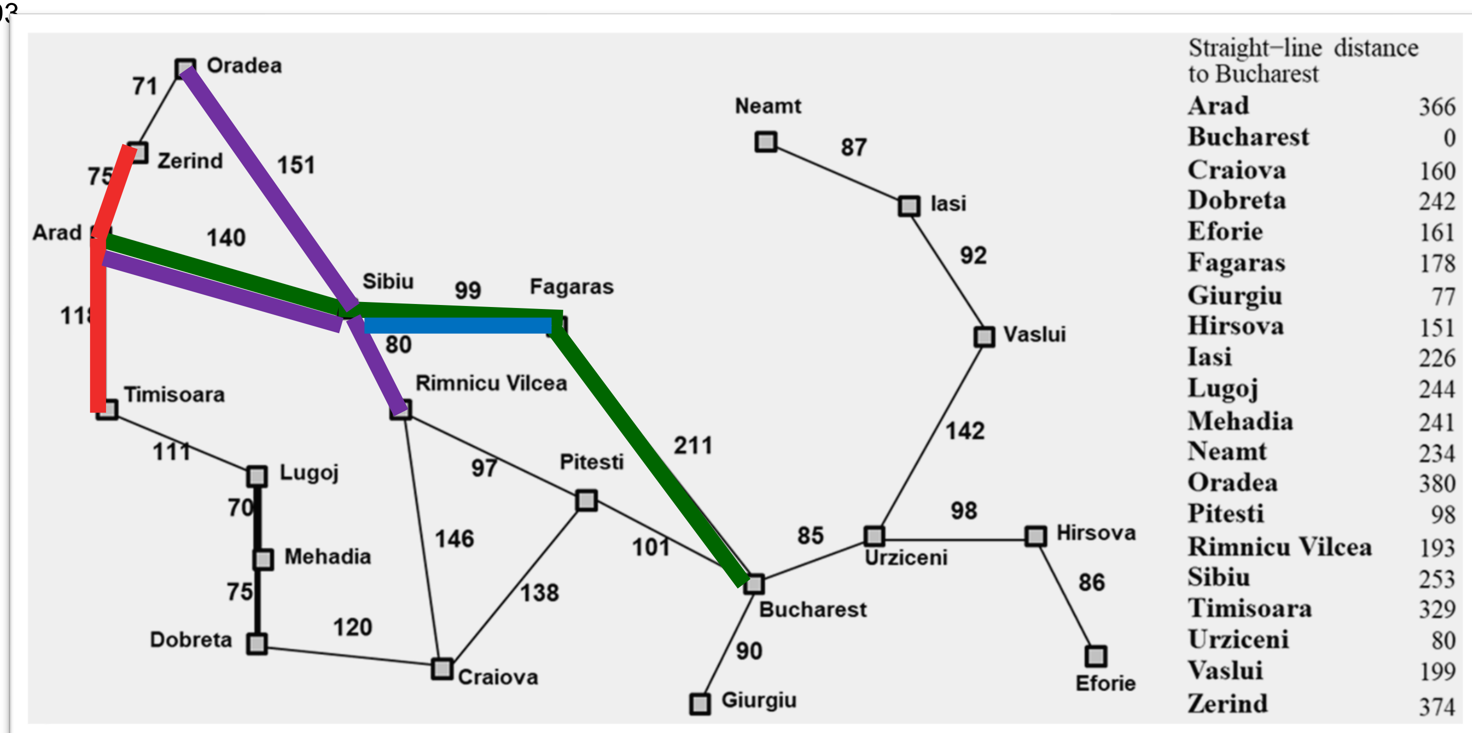




Greedy search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

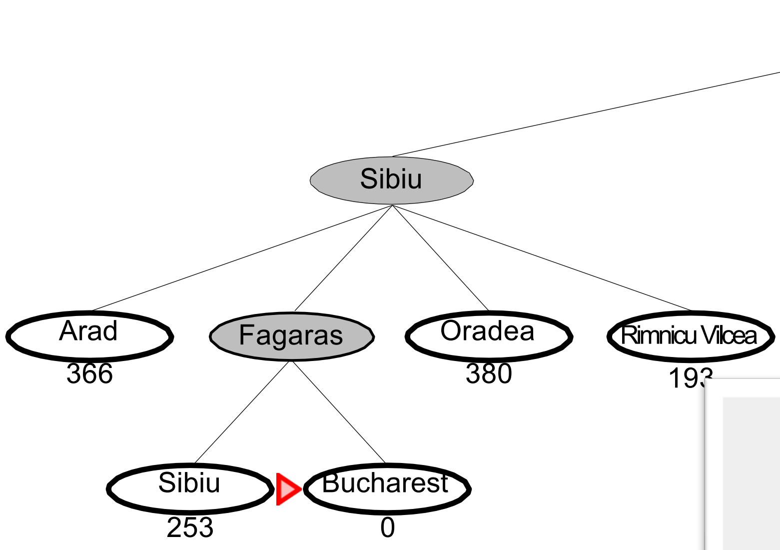


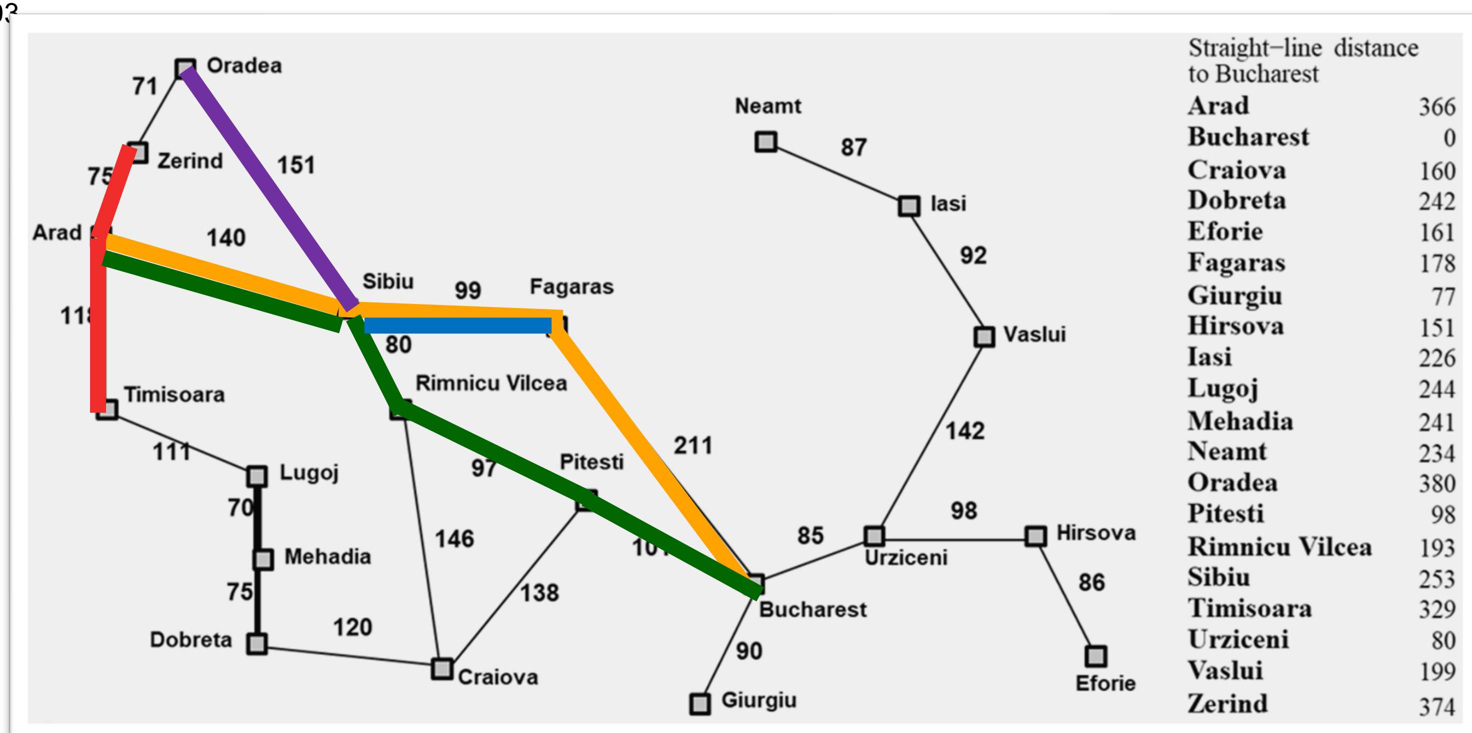


140+99+211

Greedy search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest



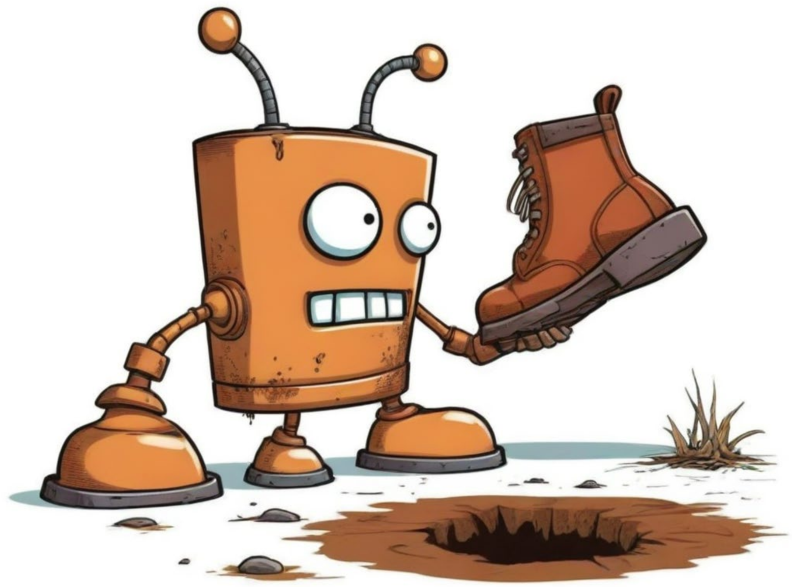


But 140+99+211 is more than
140+80+97+101

By following a local optima via heuristic we missed the global optima

Greedy Search Properties (OR-tree)

- What nodes does Greedy expand?
  - Processes node closest to solution (forward looking)!
- Time (Or-tree):
  - exponential b^m (if bad heuristic could take whole tree)
  - But good heuristic can give dramatic improvement
- Space (Or-tree):
  - Keeps all nodes in memory until found destination
- Is it complete (OR-tree)?
  - Can get stuck in loops
  - But complete in finite space with repeated-state checking
- Is it optimal (OR-tree)?
  - No (ex. we reached Bucharest and didn't explore other paths)



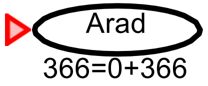
A* Search

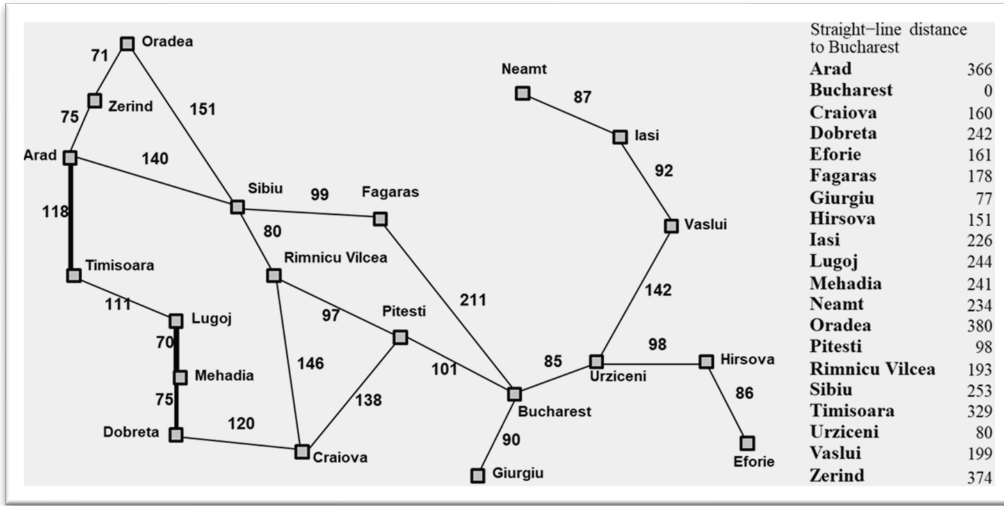
A* search

- Idea: Start greedy (only forward looking was an issue)
  - Add backwards looking, confirm one property about new heuristic
- Evaluation function f (n) = g(n) + h(n)
  - g(n) = cost so far to reach n (backwards looking)
  - h(n) = estimated cost to goal from n (greedy forward-looking part)
  - f (n) = estimated total cost of path (**A* heuristic**)
- A∗ search requires an **admissible heuristic** (fully defined later)
  - Short defn: **never overestimates the cost**
- **Theorem: A∗ search is optimal**

A* search example

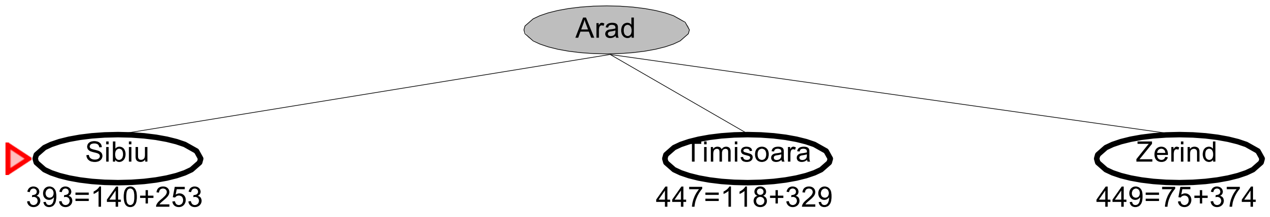
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

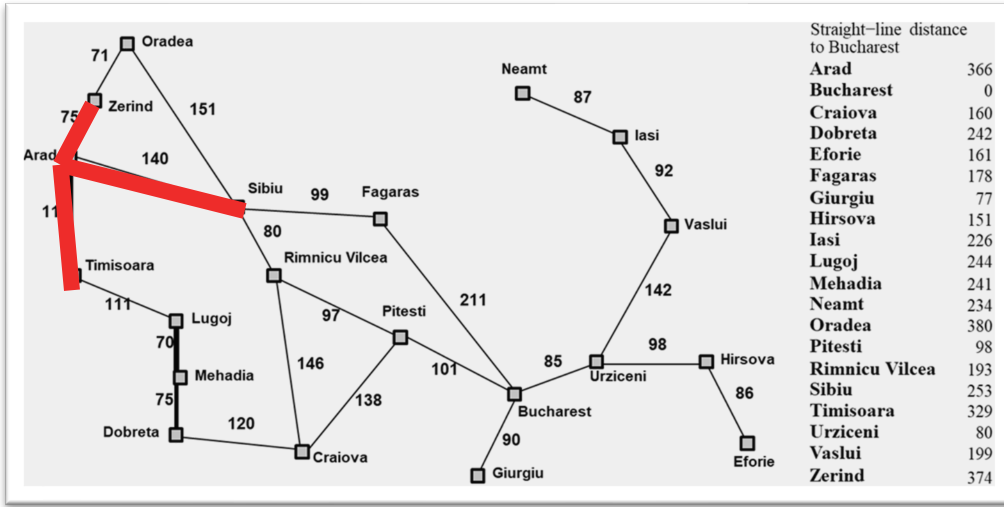




A* search example

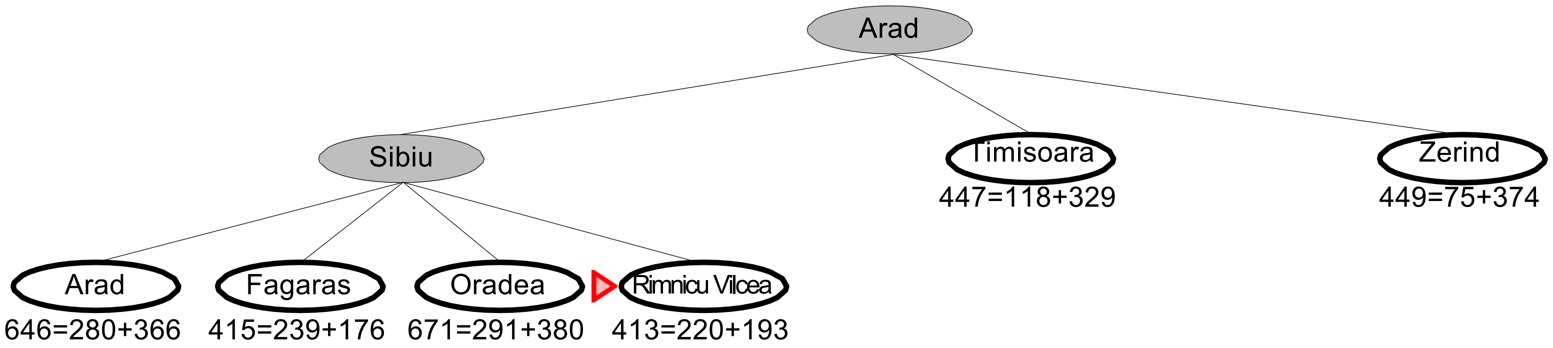
E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest



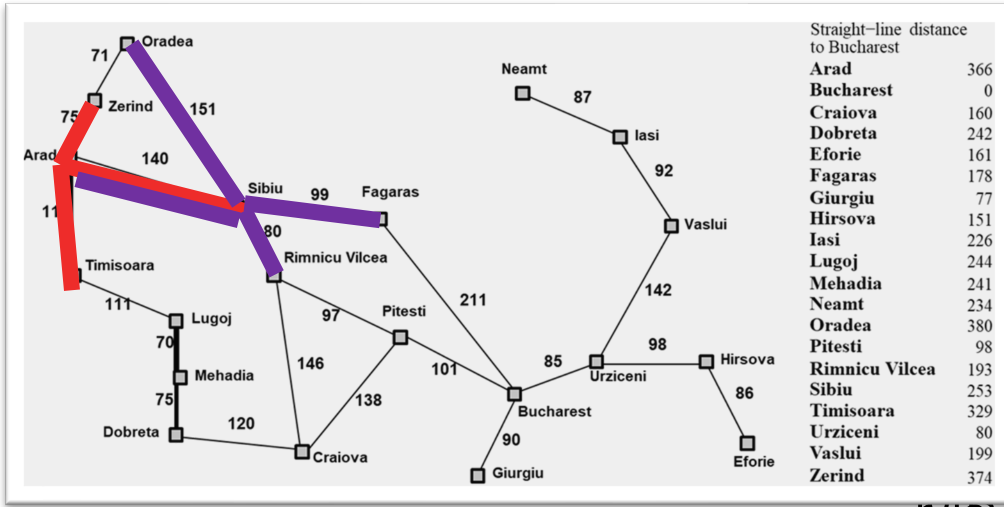


A* search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

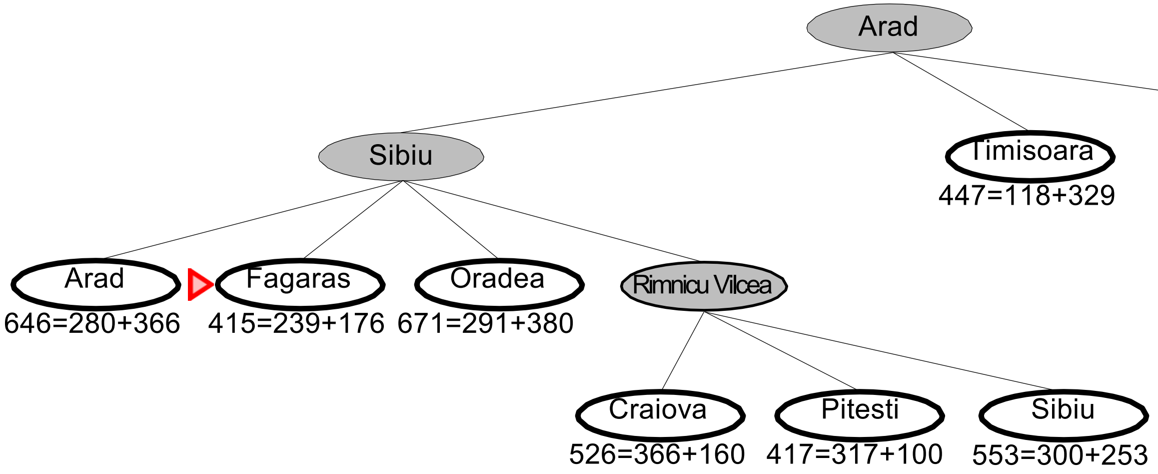


Here we are different than Greedy as we explore Rimnicu Vilcea instead of Faragas next due to heuristic

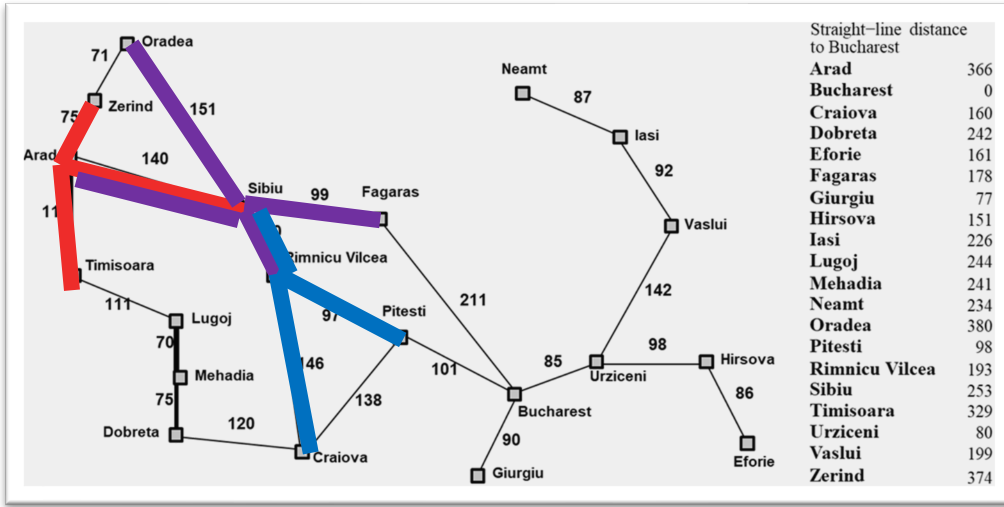


A* search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

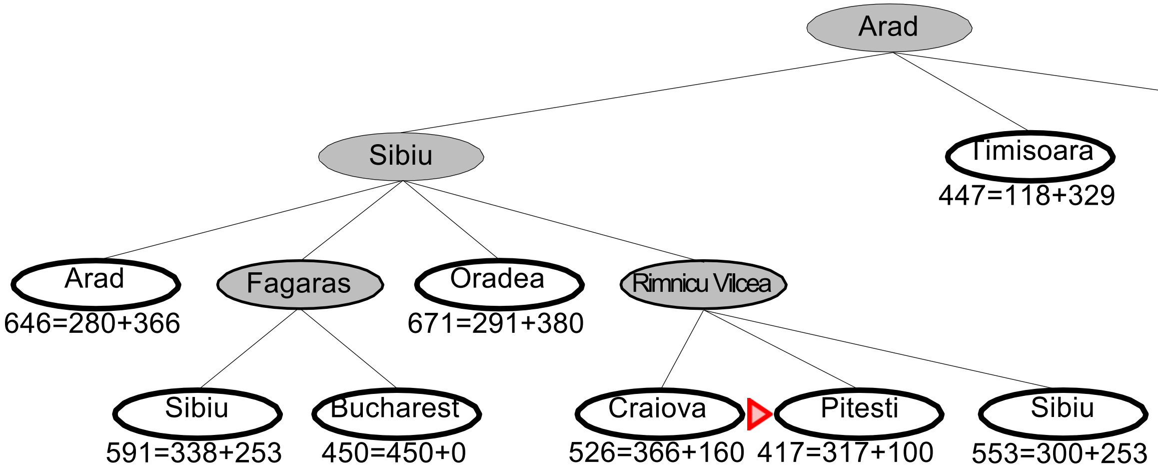


We return to look at Faragas because paths out of Rimnicu Vilcea aren't clearly better

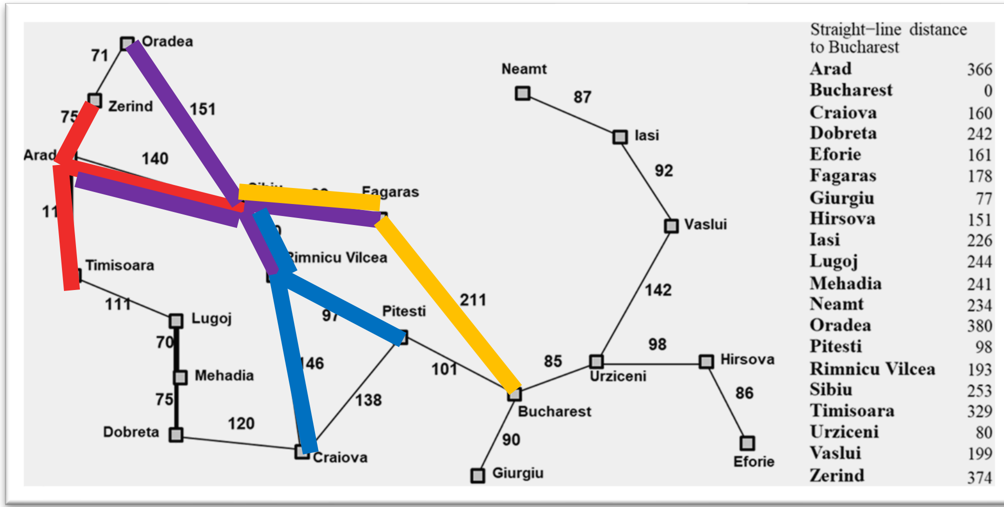


A* search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

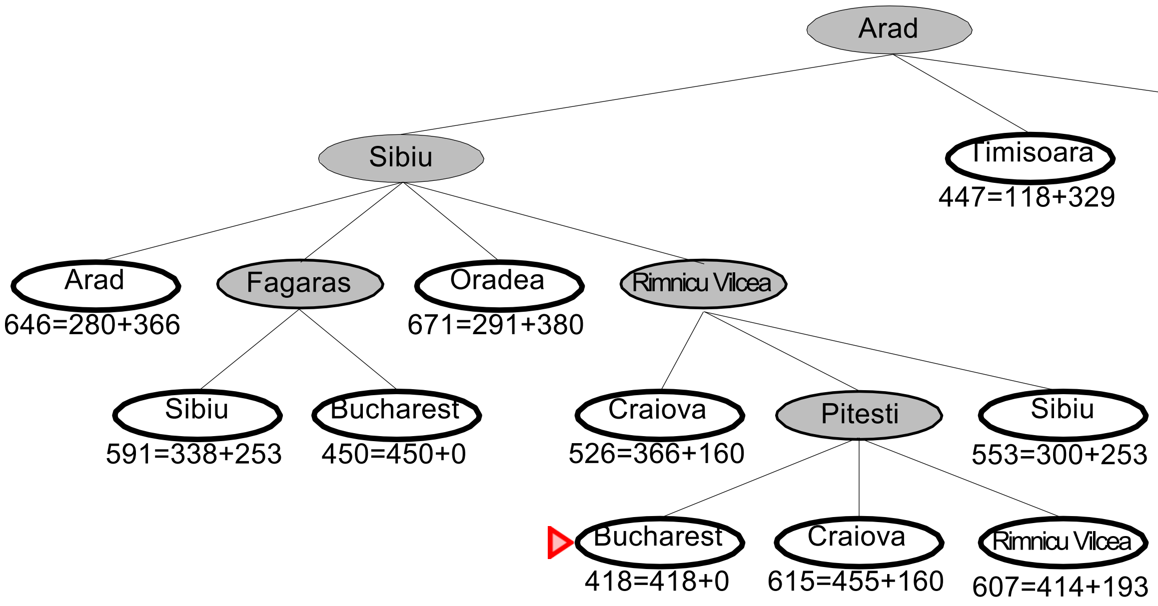


We go back to Rimnicu Vilcea to explore as at path there is more intriguing than through Faragas (at the moment)

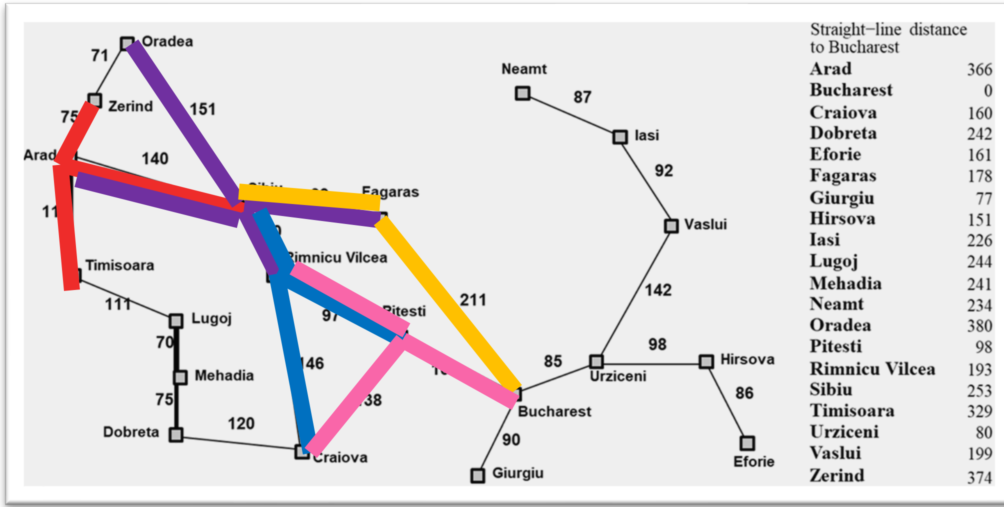


A* search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

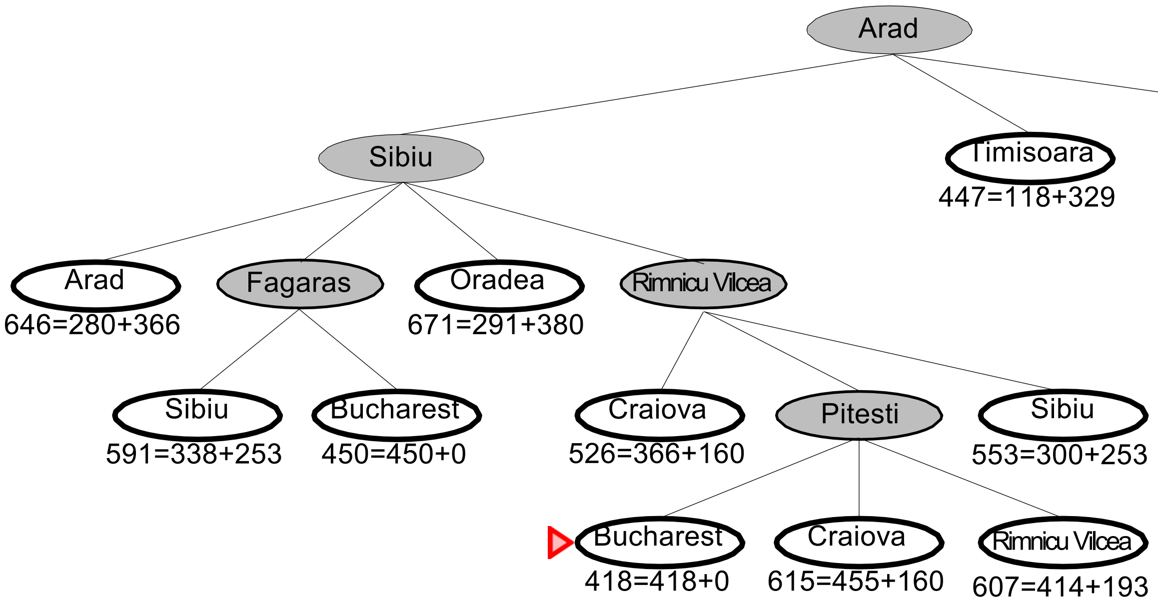


Expand Pitesti

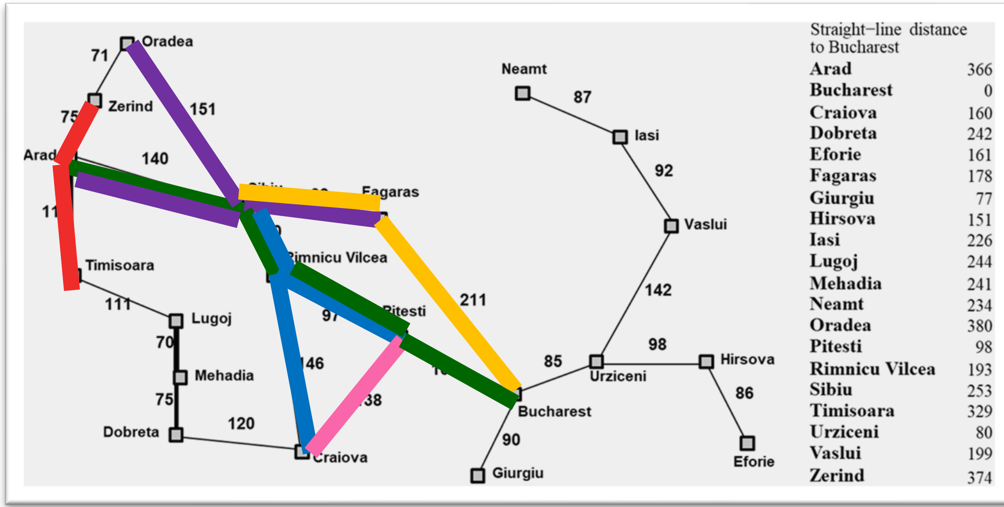


A* search example

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

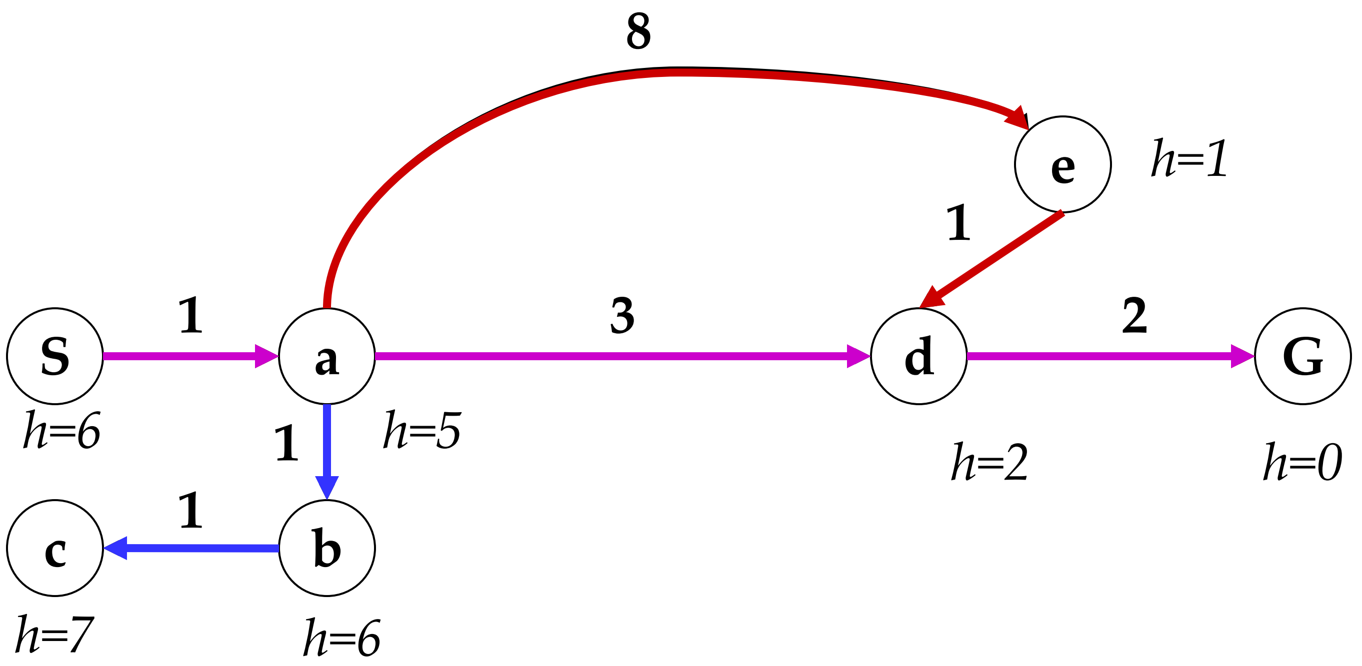


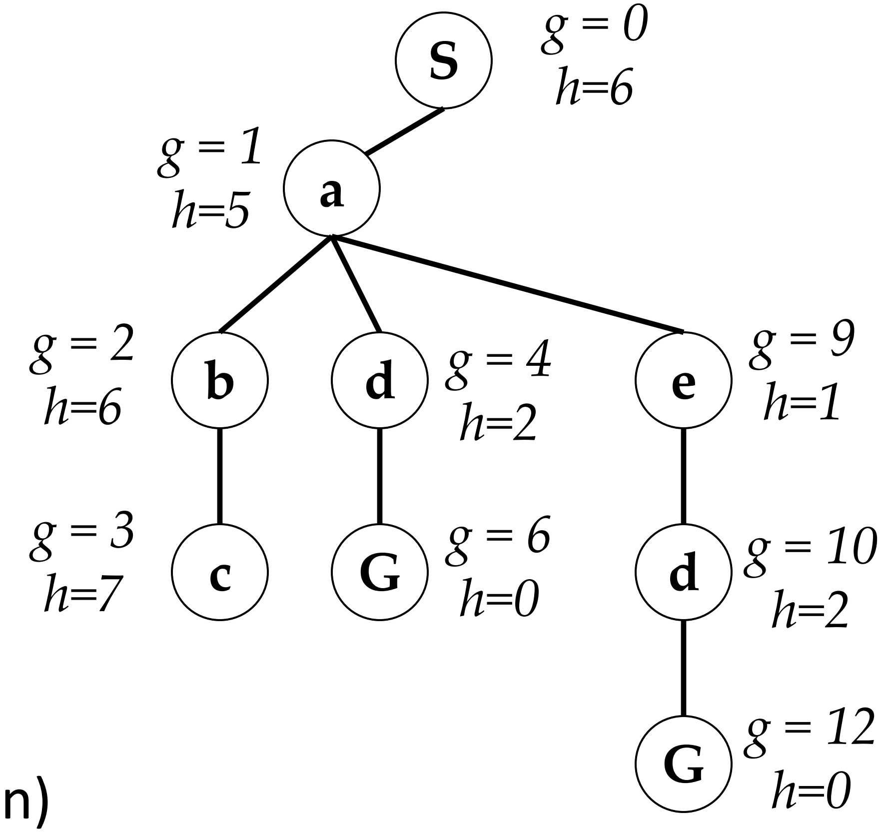
We go to Bucharest as minimal next transition (but out of Pitesti instead of Faragas!) and find the shortest path!



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)





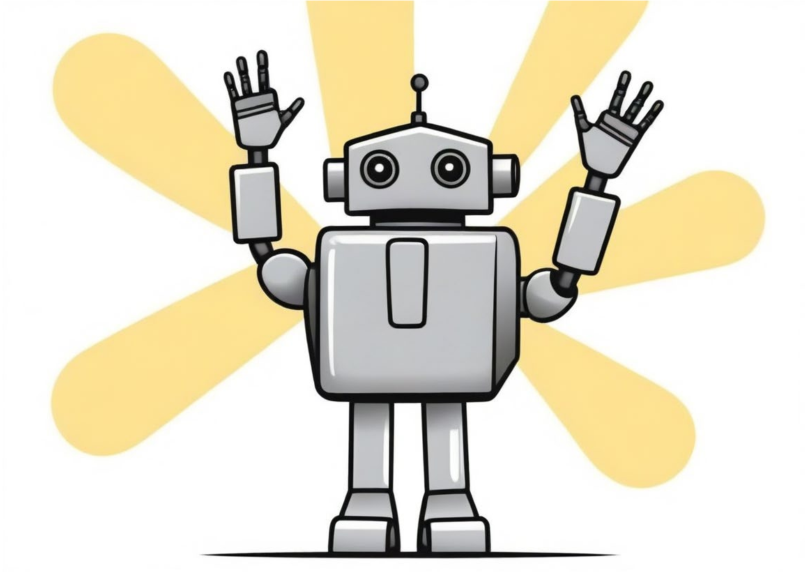
- A* Search orders by the sum: $f(n) = g(n) + h(n)$

UNIVERSITY OF CALGARY

Example: Teg Grenager

A* Search Properties (OR-tree)

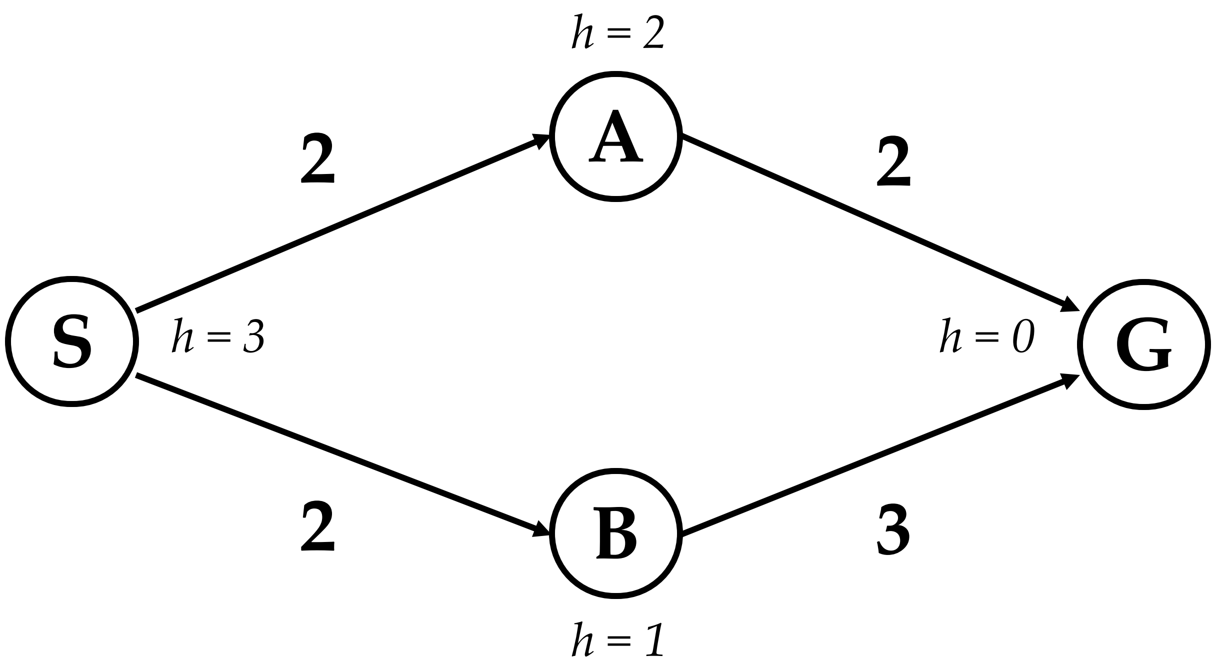
- What nodes does Greedy expand?
  - Processes all nodes with heuristic cost less than optimal solution! (does it in cost tiers)
- Time (Or-tree):
  - exponential b^m
  - but only in regard to heuristic error relative to solution
- Space (Or-tree):
  - Keeps all nodes in memory until found destination
- Is it complete (OR-tree)?
  - Yes, unless infinite expansion
- Is it optimal (OR-tree)?
  - Yes (Cannot move to a greater cost contour until smaller one is checked, i.e. will always find smallest first)



UNIVERSITY OF CALGARY

When should A* terminate?

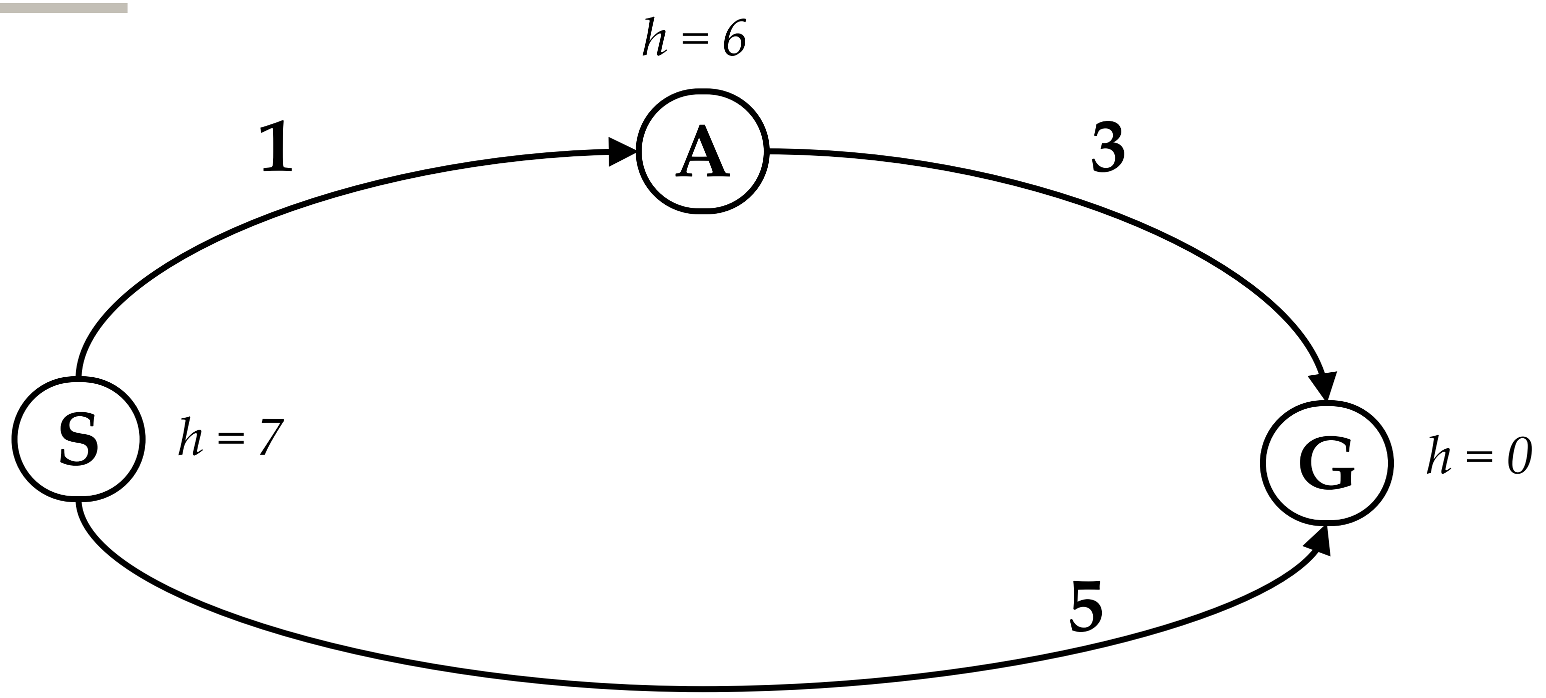
- Should we stop when we enqueue a goal?



	g	h	+
~~S~~	~~0~~	~~3~~	~~3~~
~~S->A~~	~~2~~	~~2~~	~~4~~
~~S->B~~	~~2~~	~~1~~	~~3~~
S->B->G	5	0	5
S->A->G	4	0	4

- No: only stop when we dequeue a goal

Is A* Optimal?



	g	h	+
~~S~~	~~0~~	~~7~~	~~7~~
S->A	1	6	7
S->G	5	0	5

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

Admissable Heuristics

Admissable Heuristic

- An optimistic cost guess
- Evaluation function **f** = g + h
  - g = cost so far to reach n
  - h = estimated cost to goal
  - **f = estimated total cost goal**

- **Never overestimates** (thinks things that turn out bad are better than they are)
  - This means it doesn't eliminate them from exploration too early
- But some estimate of cost allows rational limiting of what to explore first

- **A good admissible heuristic will be more accurate, a useless one would estimate 0 and have no benefit to search**

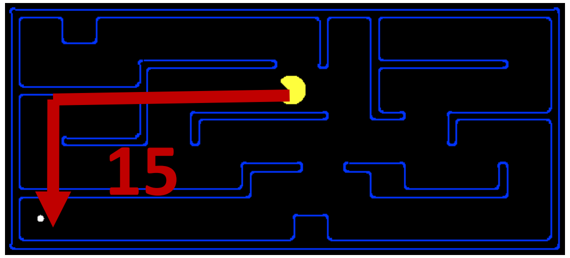
Admissible Heuristics

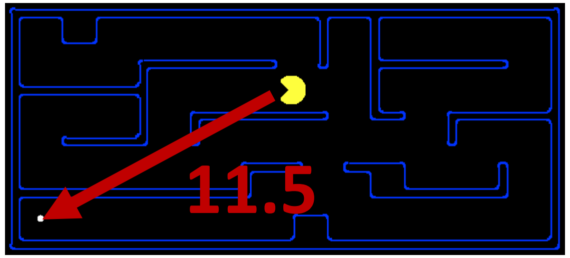
- A heuristic h is *admissible* (optimistic) if:

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

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.

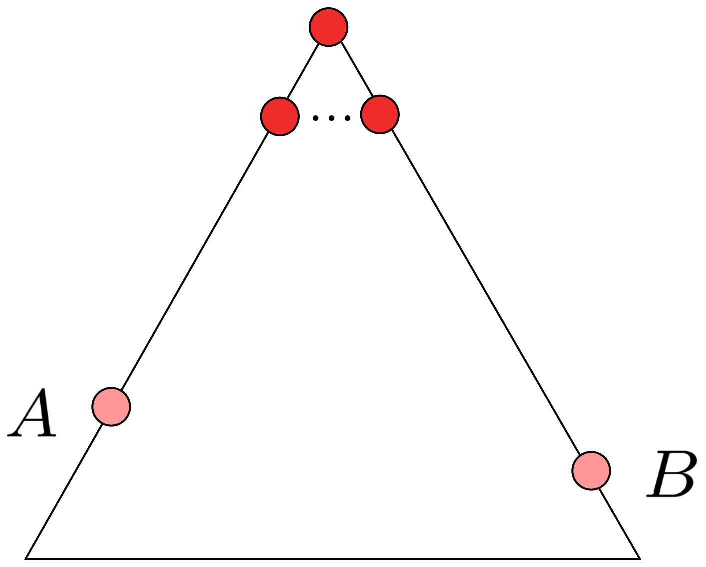
Optimality of A* Tree Search

Assume:

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

Claim:

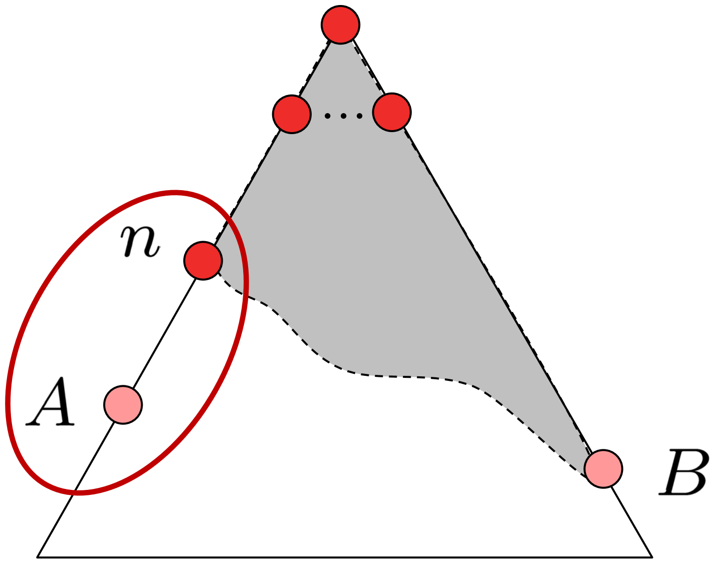
- A will exit the fringe before B



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)

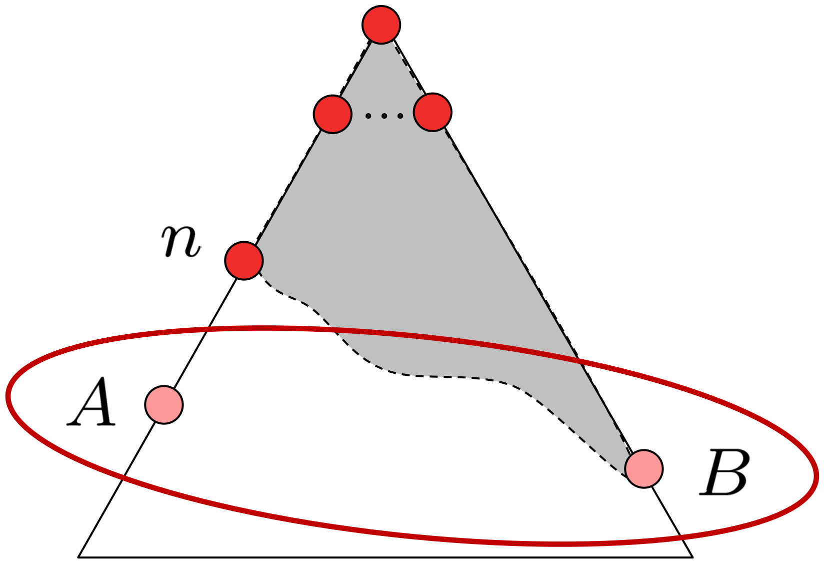


$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

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)

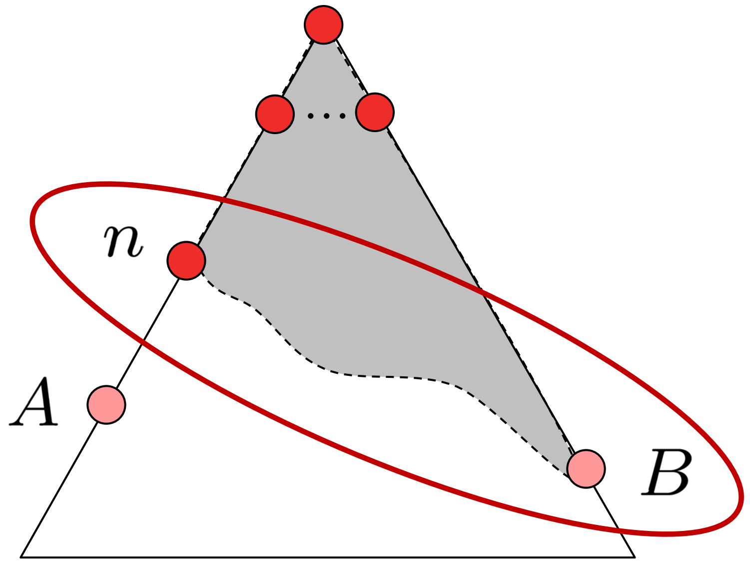


$g(A) < g(B)$	B is suboptimal
$f(A) < f(B)$	h = 0 at a goal

Optimality of A* Tree Search: Blocking

Proof:

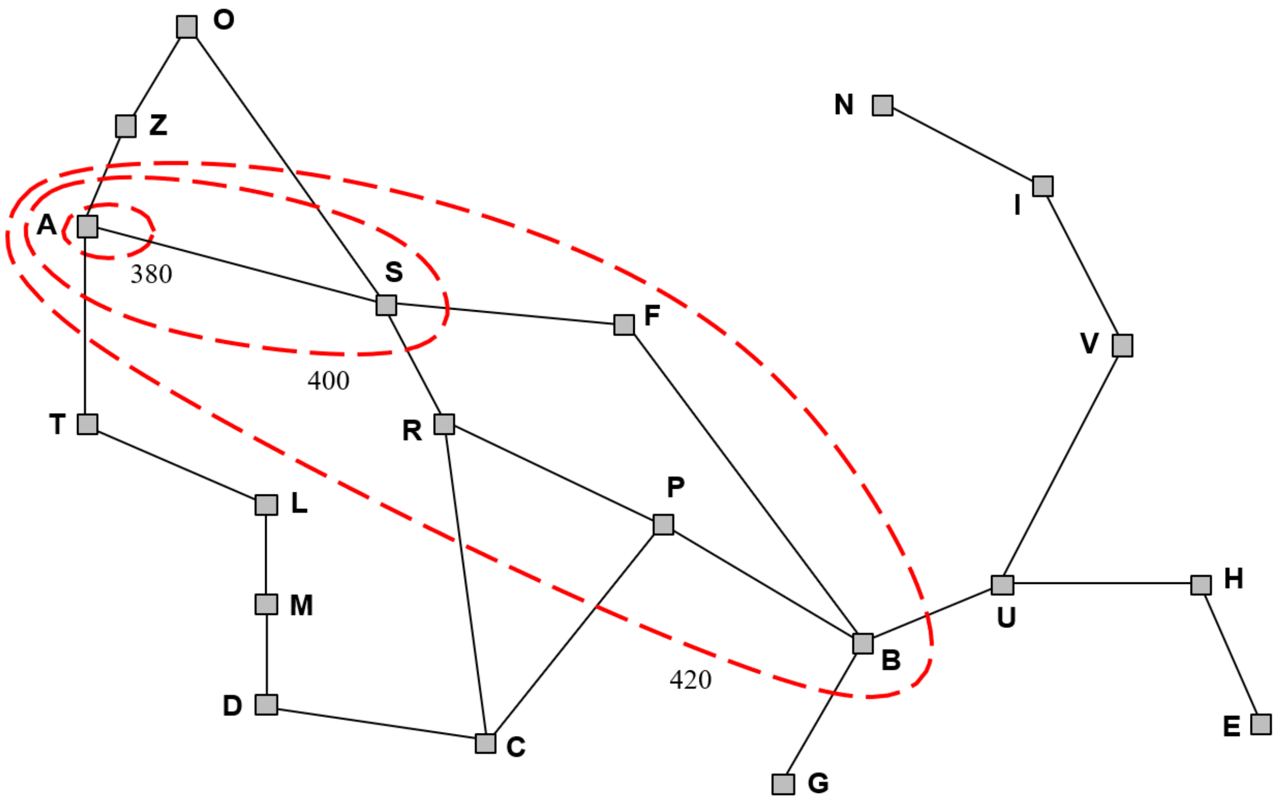
- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)
  3. *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal



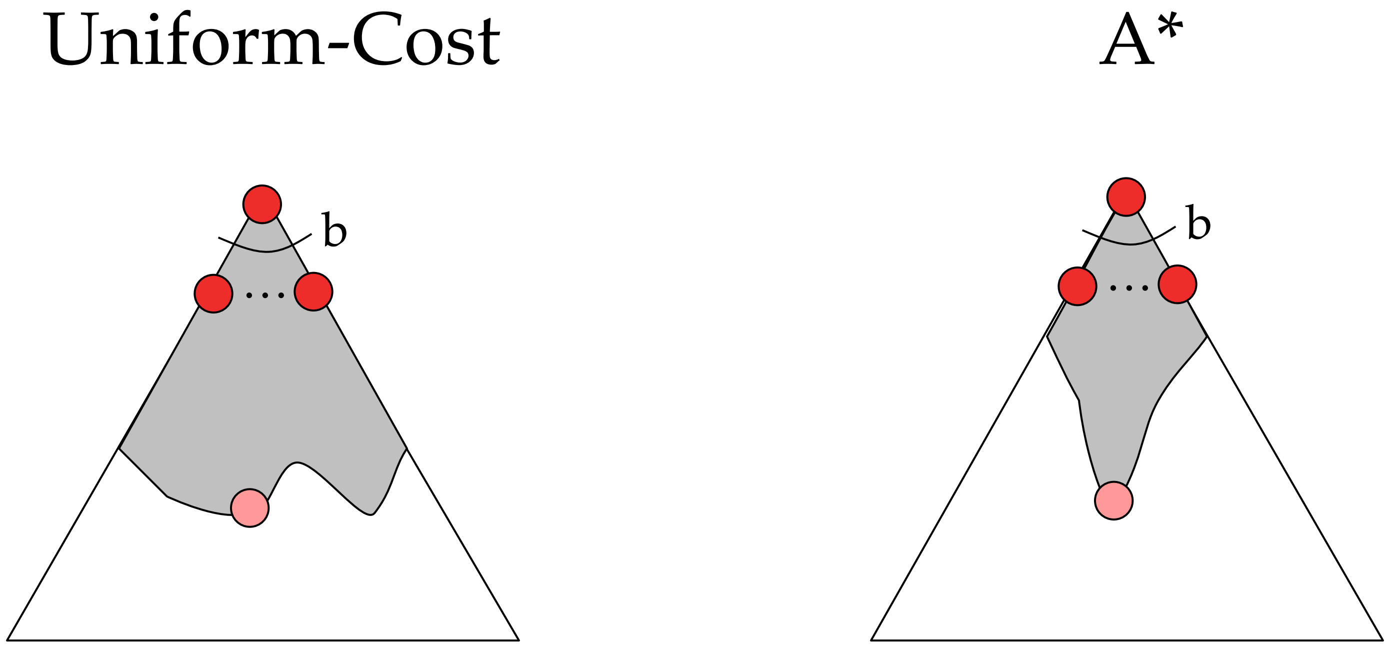
$$f(n) \leq f(A) < f(B)$$

Optimality of A∗ (more useful)

- Lemma: A∗ expands nodes in order of increasing f value
- Gradually adds "f -contours" of nodes (lowest cost breadth like expansion)

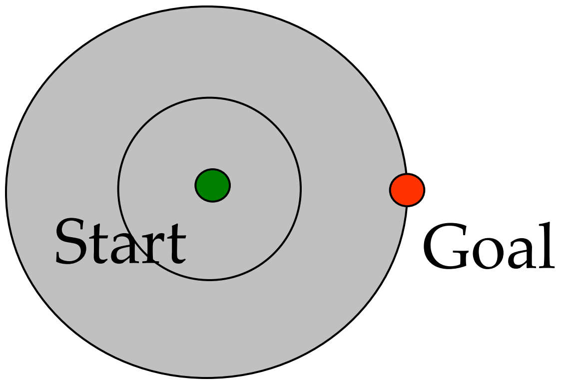


Properties of A*

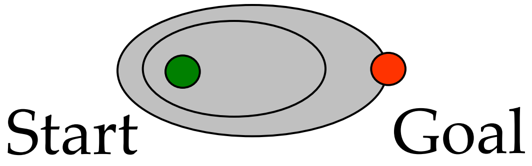


UCS vs A* Contours

- Uniform-cost expands equally in all "directions"



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



Comparison

Greedy

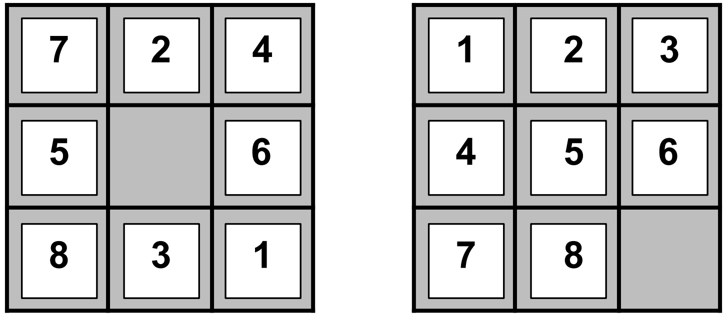
Uniform Cost

A*

Generating Admissable Heuristic

Admissible heuristics

- E.g., for the 8-puzzle:



Start State Goal State

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance
  - (i.e., no. of squares from desired location of each tile)

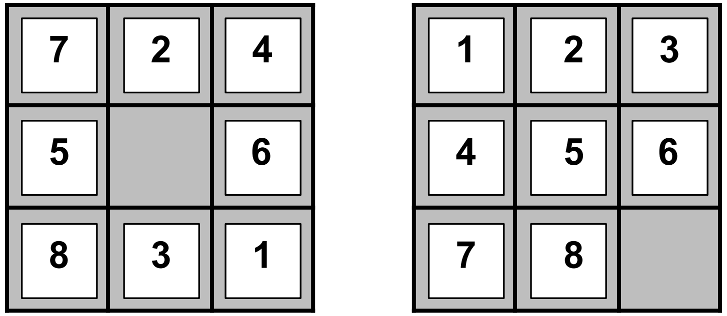
- $h_1(S)$ =
- $h_2(S)$ =

https://murhafsousli.github.io/8puzzle/#/

UNIVERSITY OF CALGARY

Admissible heuristics

- E.g., for the 8-puzzle:



Start State Goal State

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance
  - (i.e., no. of squares from desired location of each tile)

- $h_1(S)$ =?? 6
- $h_2(S)$ =?? 4+0+3+3+1+0+2+1 = 14

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 dominates h_1 and is better for search
- Typical search costs:
- $d = 14$
  - IDS = 3,473,941 nodes
  - A$*(h_1)$ = 539 nodes A$*(h_2)$ = 113 nodes
- $d = 24$
  - IDS $\approx$ 54,000,000,000 nodes
  - A$*(h_1)$ = 39,135 nodes A$*(h_2)$ = 1,641 nodes

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 **dominates** h_1 and is better for search

- Given any admissible heuristics h_a, h_b, $h(n) = \max(h_a(n), h_b(n))$
- is also admissible and dominates h_a, h_b

Relaxed problems

- Admissible heuristics can be derived from the exact
- solution cost of a relaxed version of the problem

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution

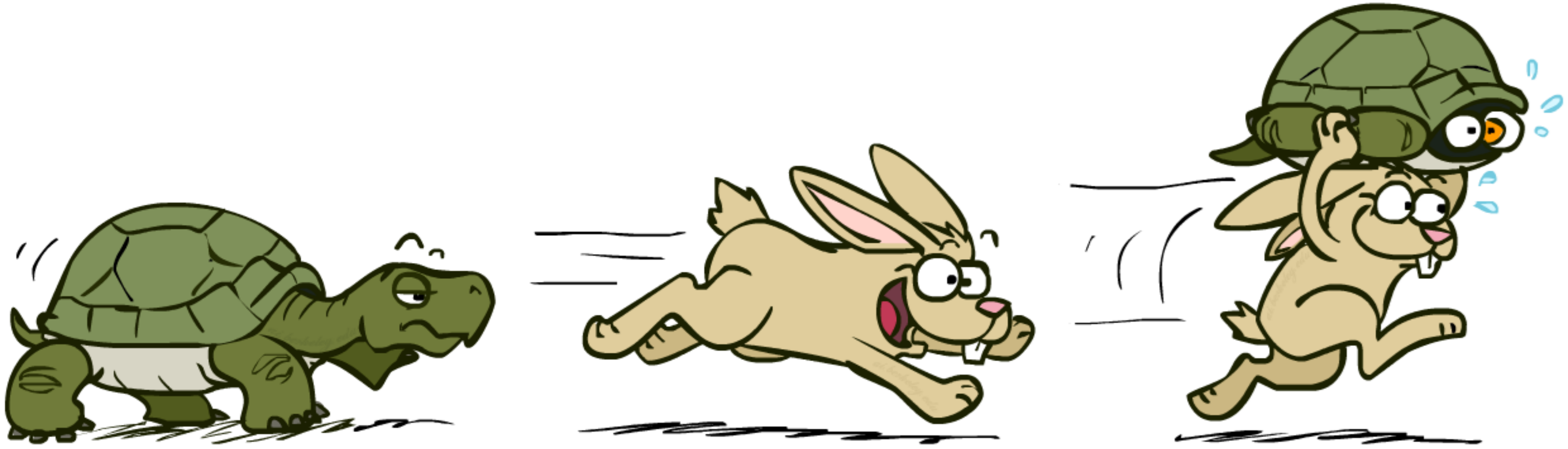
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

- Key point: the optimal solution cost of a relaxed problem
- is no greater than the optimal solution cost of the real problem

A* Summary

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



UNIVERSITY OF CALGARY

Local Search

Local Search

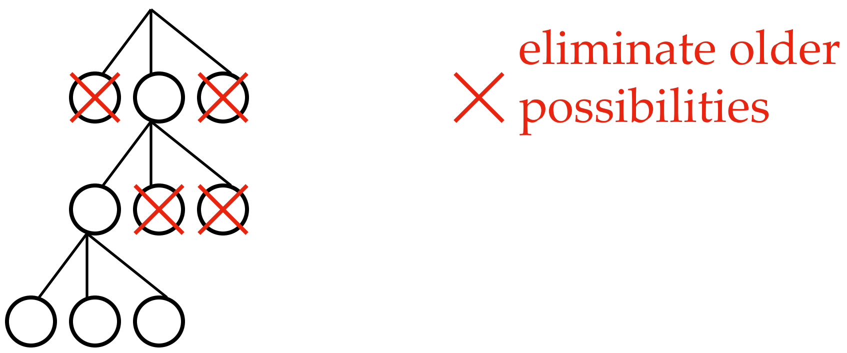
Local Search (I)

General Idea:

After selecting a transition, do not consider any transitions that were possible in previous states

☞ "Never-look-back-Heuristic"

Example: trees (works for sets also ☞ one-element sets)



UNIVERSITY OF CALGARY

Local Search (II)

Advantages:

- Less decisions
- Complexity can be bound by depth of tree (number of solution steps)
- Each transition contributes to found solution
- Predictable behavior with regard to run time

Disadvantages

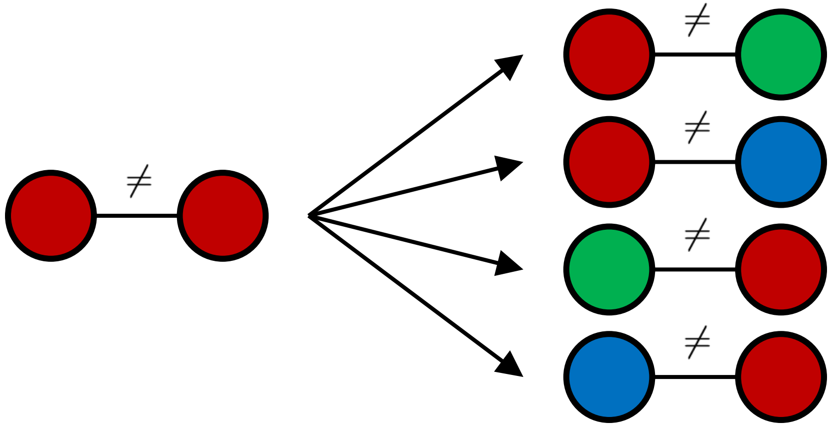
- No guarantee for optimality of solution
- No guarantee for optimality of number of necessary transitions

Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)

- Local search: improve a single option until you can't make it better (no fringe!)

- New successor function: local changes



- Generally much faster and more memory efficient (but incomplete and suboptimal)

Simple Local Search

Hill Climbing

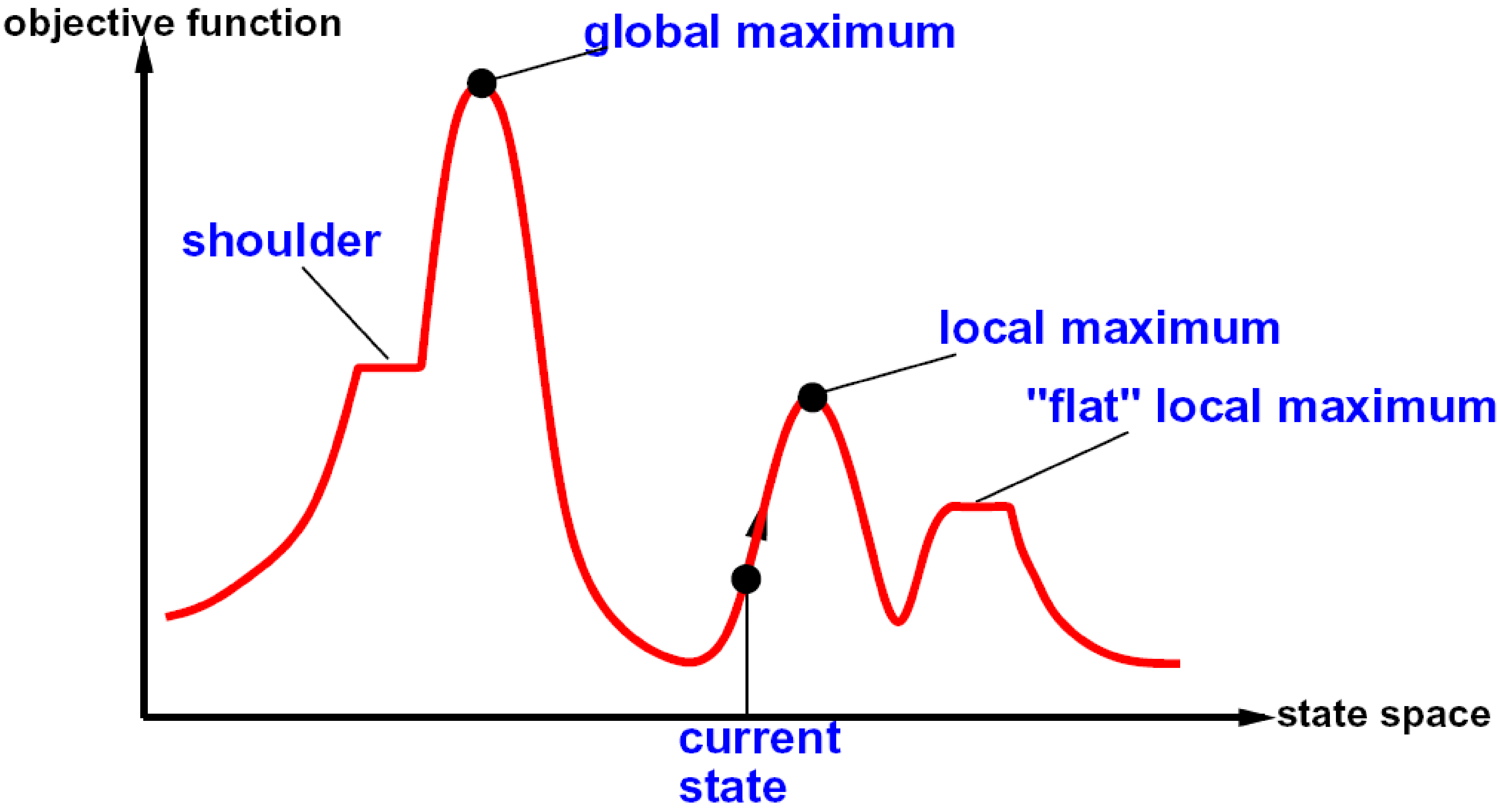
- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- What's bad about this approach?

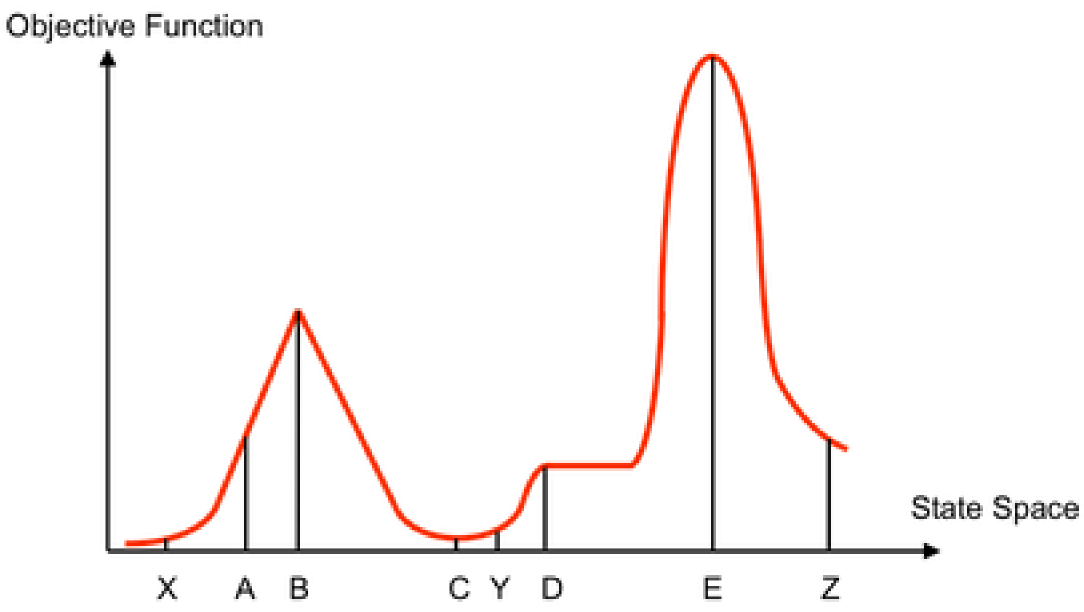
- What's good about it?

CALGARY

Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

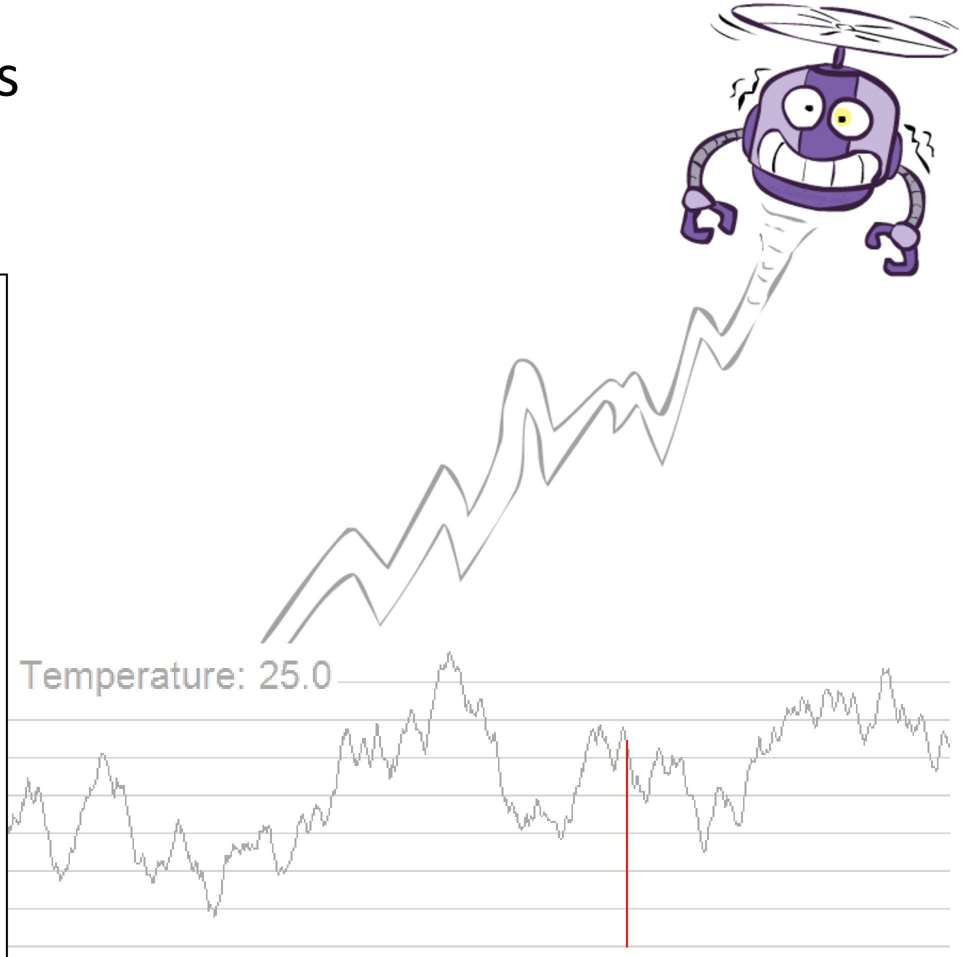
Advanced Local Search

Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

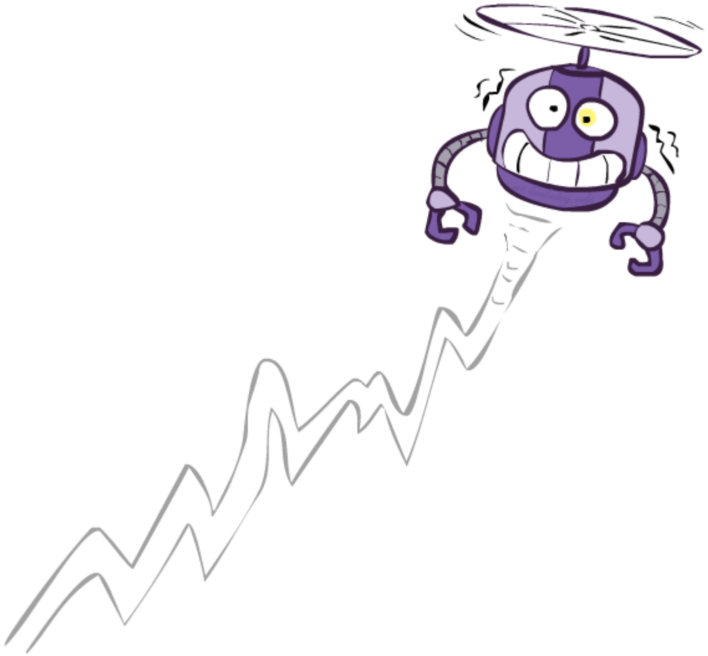


Simulated Annealing

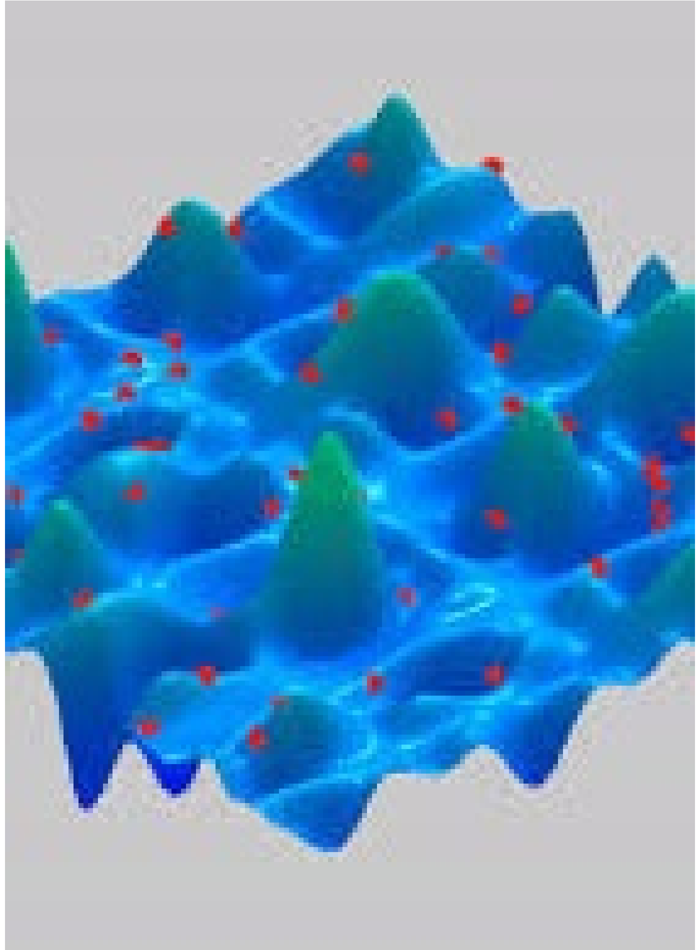
- Theoretical guarantee:
  - If 'Temperature' decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?

- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

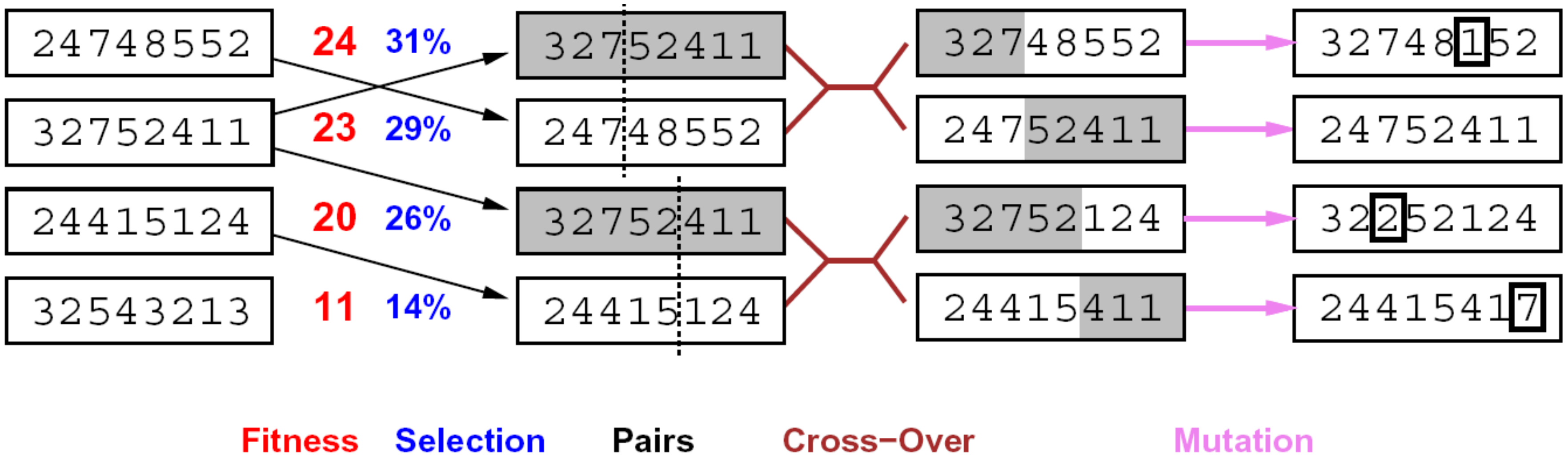


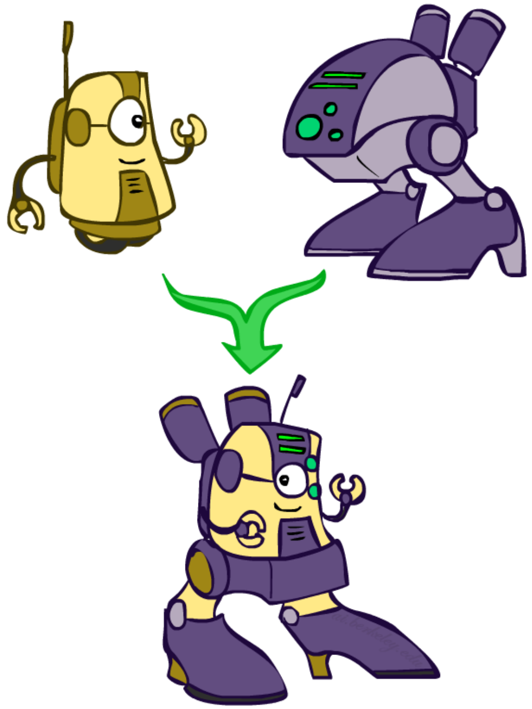
Particle Swarm Optimization



- Design complexity grows.
- Think of particles as having ‘gravity’. The better the solution the more ‘gravity’.
- Particles also have momentum.
- Have many particles.
- Each step, particles follow their current direction of change with influence of the nearby local optima and global optima.
- Less touchy to parameters and good at exploration. Often cooling principle included to help find best at end.
- Challenges with discrete problems.

Genetic Algorithms



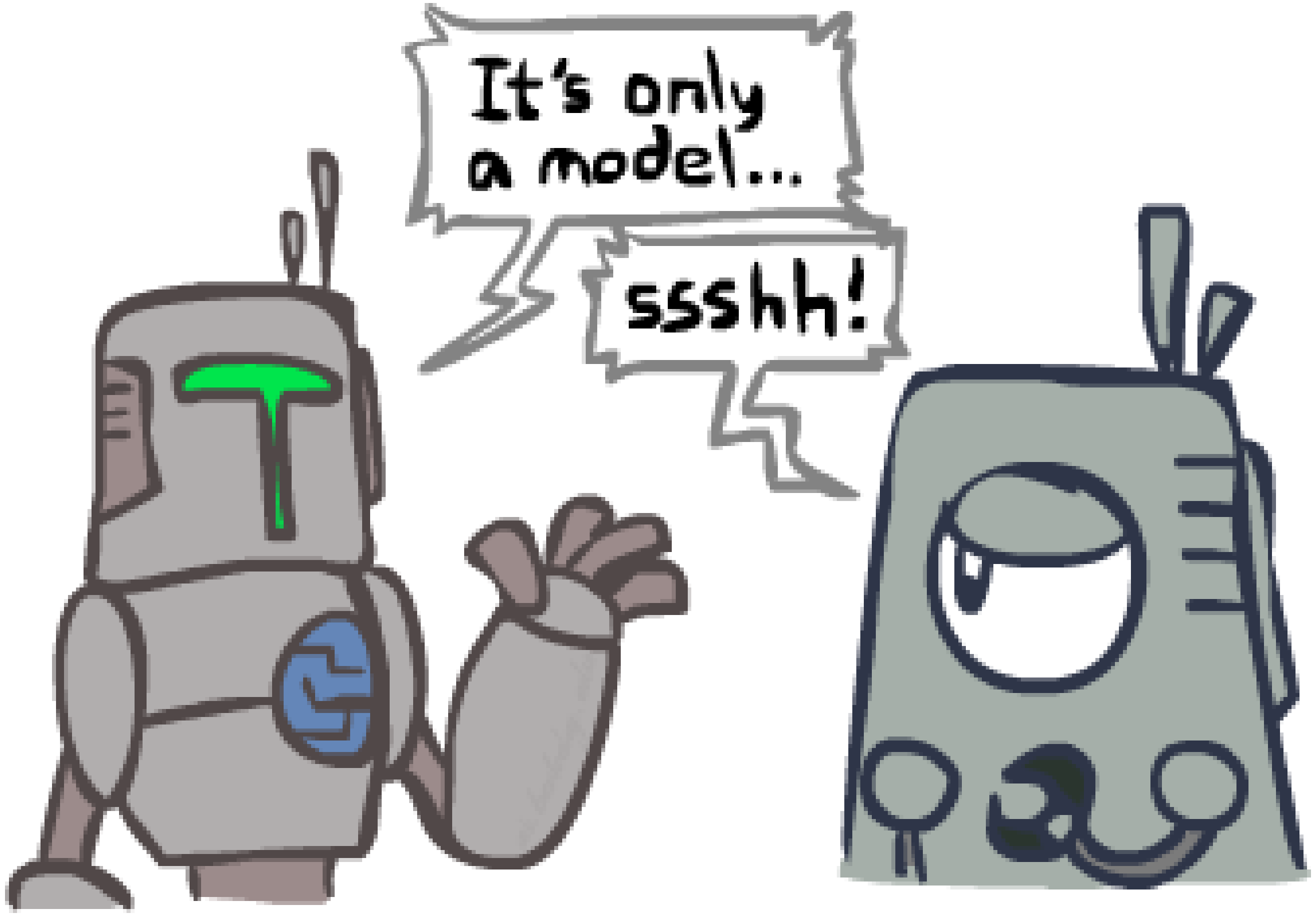


- Genetic algorithms use a natural selection metaphor
  - Survival of the fittest (fit being best solution value)
    - Keep best N hypotheses at each step (selection) based on a fitness function
  - Create next generation by combining ‘DNA’ of the previous
    - Crossover operators (two parents) and mutation operators
- Possibly the most misunderstood, misapplied (and even maligned) technique around

Search Summary

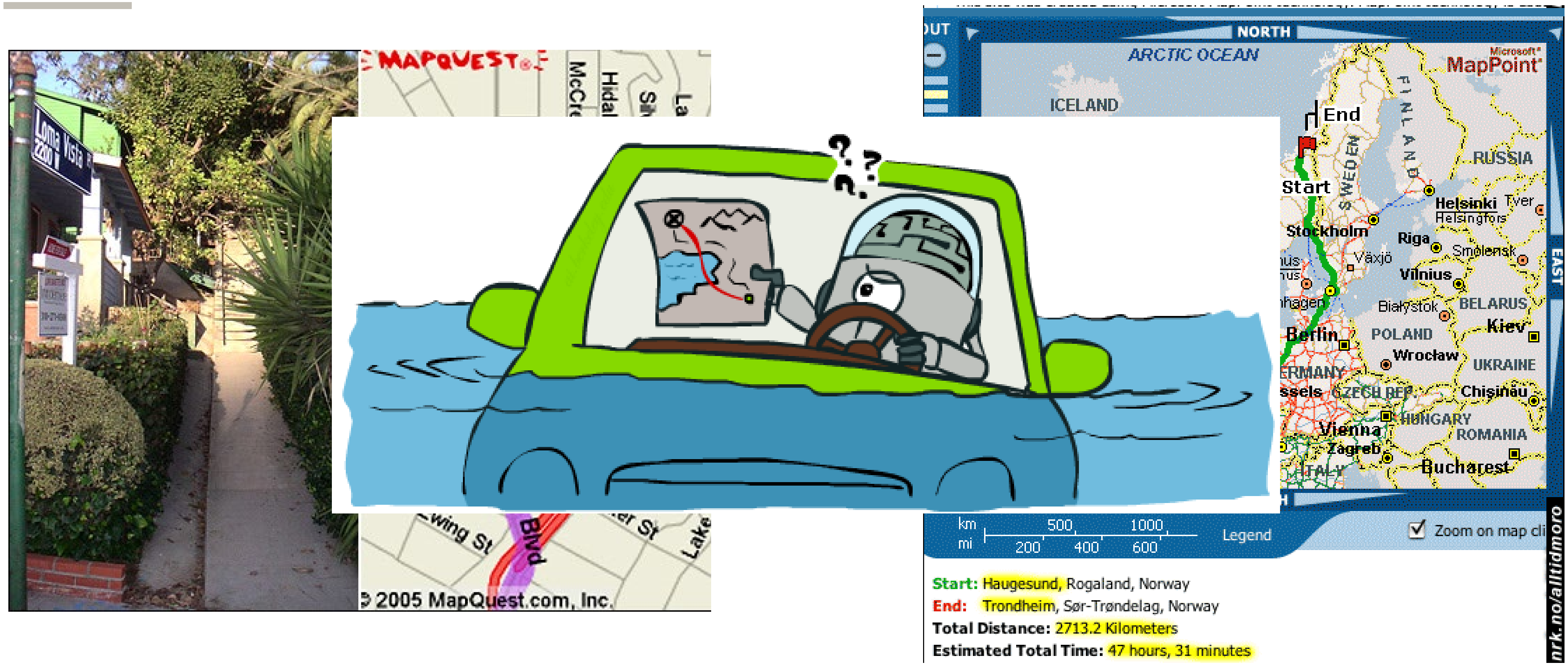
Search and Models

- 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…



UNIVERSITY OF CALGARY

Search Gone Wrong?



UNIVERSITY OF CALGARY

Onward to … neural networks

Jonathan Hudson, Ph.D.
jwhudson@ucalgary.ca
https://cspages.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY