Search

a.k.a. Long-term Decision Making

Dr. Angelica Lim
Assistant Professor
School of Computing Science
Simon Fraser University, Canada

Oct. 8, 2024

ELSEVIER

Ocean Engineering
Volume 284, 15 September 2023, 115271

Fuel consumption prediction for a passenger ferry using machine learning and in-service data: A comparative study ☆

Pedram Agand [a], Allison Kennedy [b], Trevor Harris [b], Chanwoo Bae [c], Mo Chen [a], Edward J. Park [d]

Show more

+ Add to Mendeley Share Cite

https://doi.org/10.1016/j.oceaneng.2023.115271

Assignment 2 will be released this week

Continue to attend lectures!

Powerpoints from textbooks?

Deep Reinforcement Learning-based Intelligent Traffic Signal Controls with Optimized CO2 emissions

Pedram Agand[1], Alexey Iskrov[2], and Mo Chen[1]

***Abstract*— Nowadays, transportation networks face the challenge of sub-optimal control policies that can have adverse effects on human health, the environment, and contribute to traffic congestion. Increased levels of air pollution and extended commute times caused by traffic bottlenecks make intersection traffic signal controllers a crucial component of modern transportation infrastructure. Despite several adaptive traffic signal controllers in literature, limited research has been conducted on their comparative performance. Furthermore, despite carbon dioxide (CO2) emissions' significance as a global issue, the literature has paid limited attention to this**

achieved by using Reinforcement Learning (RL) algorithms that can adjust according to traffic conditions. In much of past work, reward has been defined as an ad hoc weighted linear combination of numerous traffic measures [7]. In order to take into account more aspects of the traffic conditions, recent RL techniques incorporate sophisticated states such as images from cameras. This added complexity may result in a less efficient learning process without a considerable improvement in performance.

Geoffrey Hinton from University of Toronto awarded Nobel Prize in Physics

Hinton and John Hopfield of Princeton University were honoured for work that enables machine learning

CBC News · Posted: Oct 08, 2024 3:07 AM PDT | Last Updated: 2 hours ago

What is low level stuff in the context of machine learning?

Scientists Geoffrey Hinton, pictured in Toronto in June 2023, from the University of Toronto and John Hopfield of Princeton University won the 2024 Nobel Prize in Physics for discoveries and inventions that enable machine learning within artificial neural networks, the award-giving body said on Tuesday. (Evan Mitsui/CBC)

Preguntar Angelica si quiere que lean una seccion especifica del tercer capitulo

Debrief on machine learning module

Mathematical Foundations

- Multivariate calculus
- Partial derivatives, gradients
- Probability

Machine Learning

- Training sets, test sets
- Cross-validation
- Evaluation metrics

Unsupervised Learning

- K-Means Clustering

Supervised Learning

- Non-parametric methods:
  - K-Nearest Neighbors, Decision Trees
- Parametric methods*:
  - Hypothesis class, loss function, optimization algorithm
  - Linear regression
  - Linear classification
  - Gradient descent
  - Two-layer neural network
  - Backpropagation w/ computation graphs

Search
Markov decision processes
Games

Constraint satisfaction problems
Markov networks
Bayesian networks

Reflex-based models | **State-based models** | **Variable-based models** | **Logic-based models**

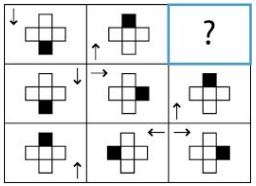
*Credit: Stanford CS 221, used with permission

Thinking, Fast and Slow

Kahneman (2011) proposed that humans have two modes of thought:

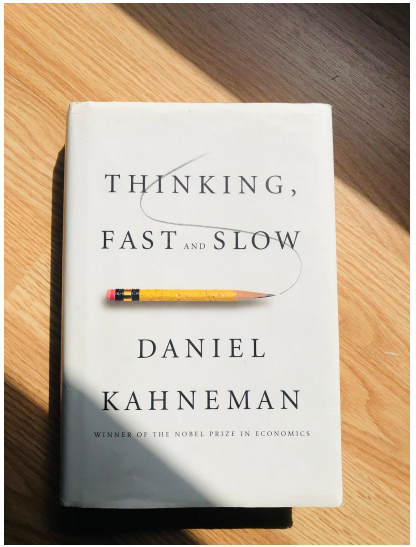
System 1: fast, automatic, intuitive, unconscious → **low-level**

- continuously and involuntarily generates impressions, intuitions, intentions, feelings. How we read facial expressions, react to sounds, colours, images.
- *e.g. complete the phrase "war and ...", solve 2+2=?, drive a car on an empty road*

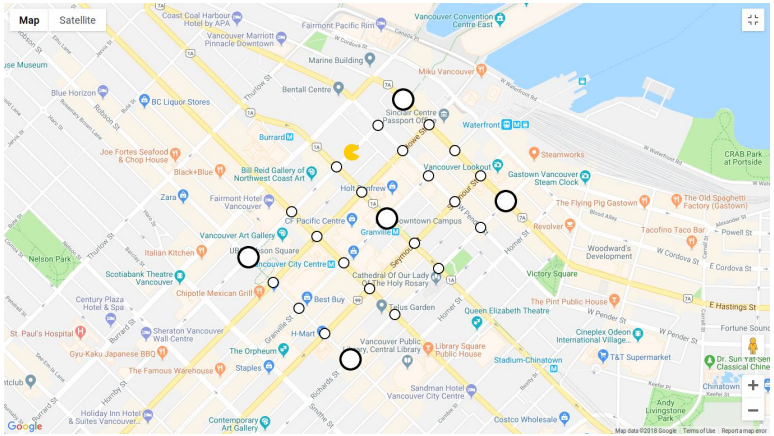


System 2: slow, deliberate, logical, conscious → **high-level**

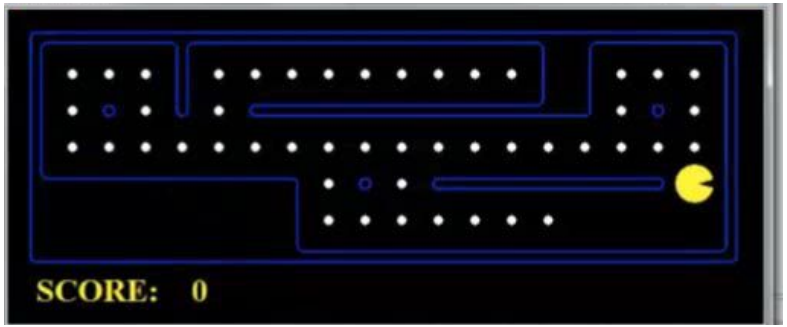
- conscious, reason-based approach that considers evidence
- *e.g. count the number of A's in a certain text, solve 17 × 24, park into a tight parking space*



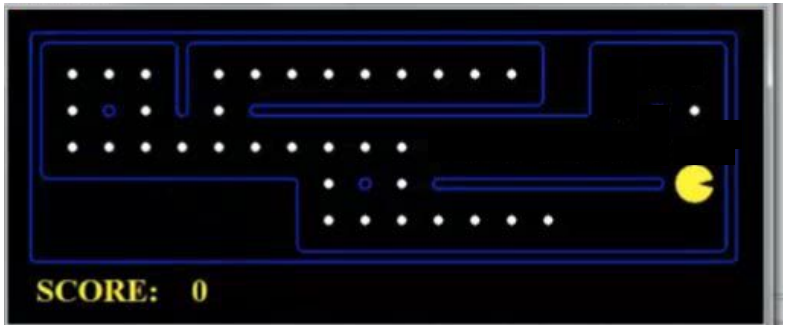
What is the best next move?



https://github.com/pacmacro/pacmacro.github.io



ai.berkeley.edu



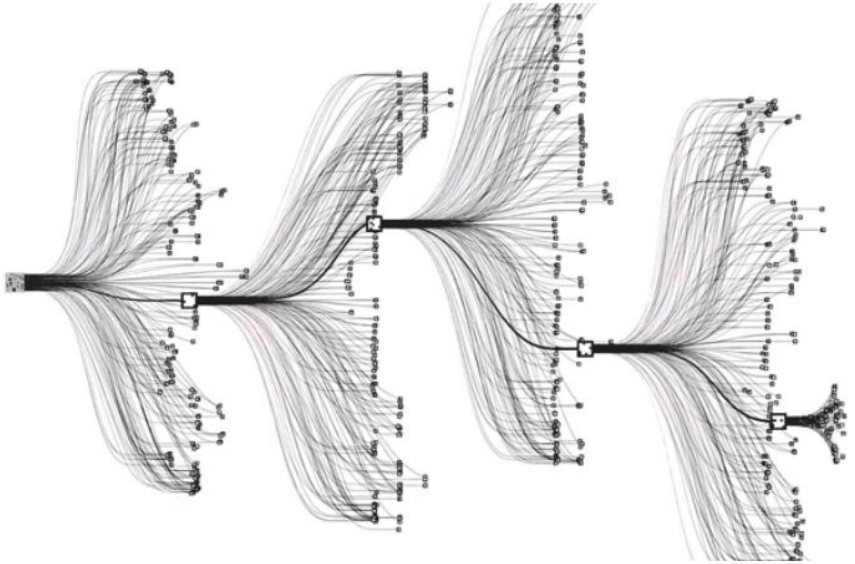
SFU CSSS Frosh circa 2005

State-based models

Key idea: Model the state of the world and transitions between states, triggered by actions

Tasks that require forethought, *e.g. playing chess, planning a big trip, collecting dots on a map*

Solutions are **procedural**, step by step sequences of actions.



Applications:

- Games: Chess, Go, Pac-Man, Starcraft, etc.
- Robotics: motion planning

Beyond reflex

Classifier (reflex-based models):

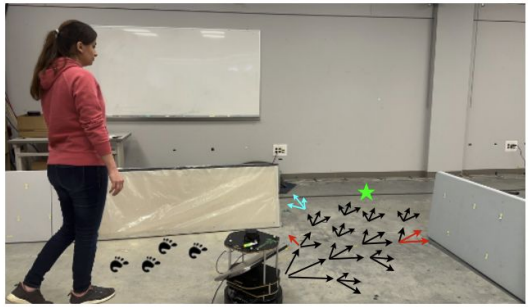
Search problem (state-based models):

Key: need to consider future consequences of an action!

Credit: Stanford CS 221, used with permission

Applications

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



Robot Motion Planning
Actions: straight, turn left 45°, turn right 45°
Goal: robot stays in front of the human

la maison bleue

the blue house

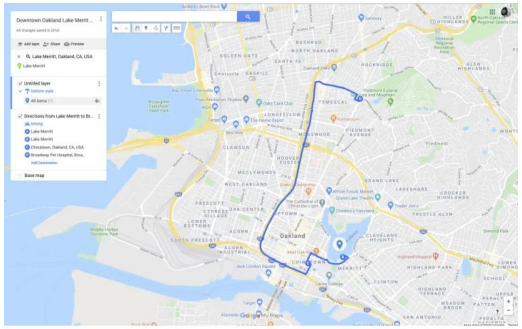
Machine Translation
Actions: append single words (e.g. the)
Goal: a sentence with fluent English and same meaning

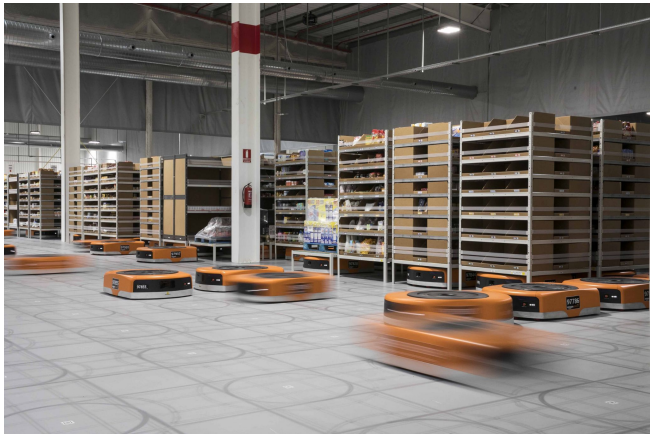
Applications

Mapping

Actions: go straight, turn right, turn left

Goal: destination in minimal time (or energy)





Multi-robot systems

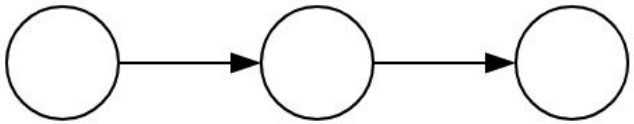
Actions: acceleration and steering of all robots

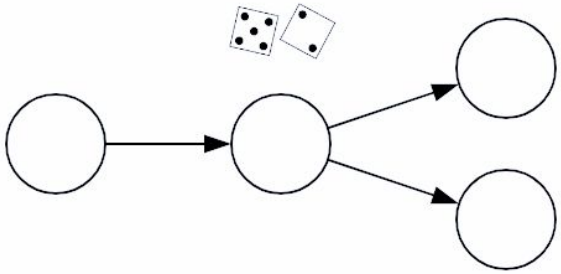
Goal: complete tasks in minimal time (or energy cost)

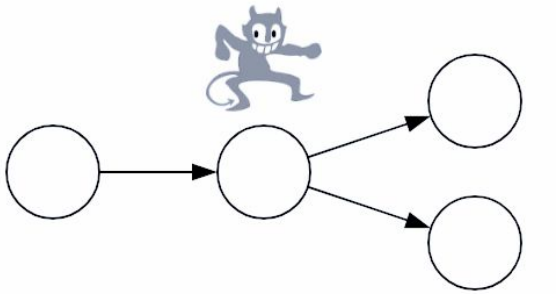
Your state can include the movement of other agents

Would we use a Kalman filter in order to determine the movement of the agent?
How do we account for the movement of other agents?

State-based models







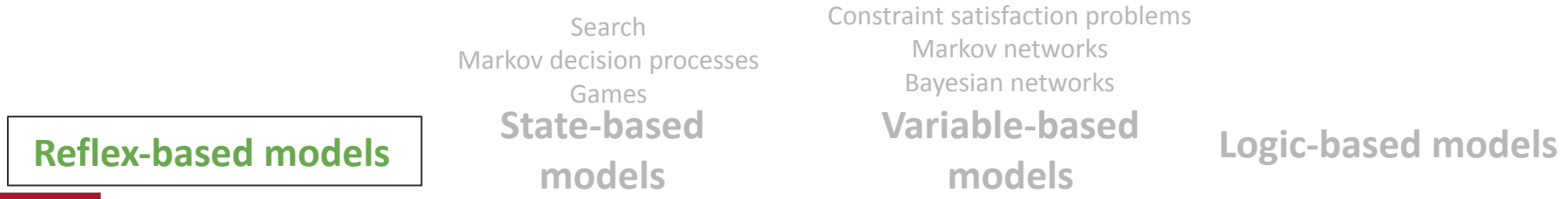
Search problems: when you have an environment with no uncertainty, ie. perfect information. But realistic settings are more complex

Markov decision processes (MDPs) handle situations with randomness, e.g. Blackjack

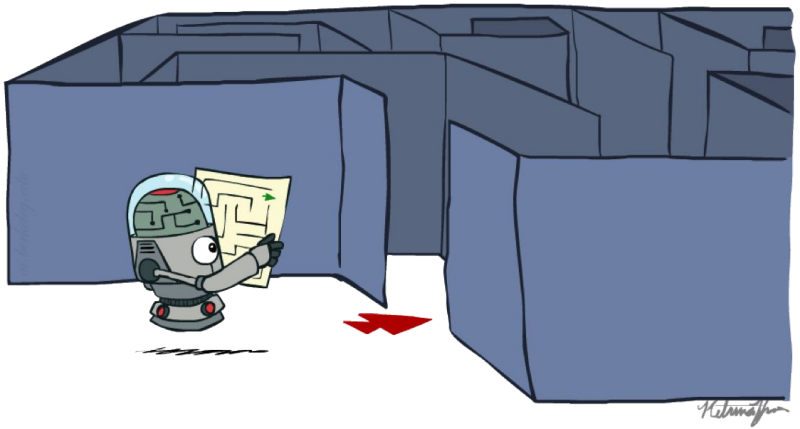
Game playing handles tasks where there is interaction with another agent. Adversarial games assume an opponent, e.g. Chess

Course Overview

Week 1 : Getting to know you
Week 2 : Introduction to Artificial Intelligence
Week 3: Machine Learning I: Basic Supervised Models (Classification)
Week 4: Machine Learning II: Supervised Regression, Classification and Gradient Descent, K-Means
Week 5: Machine Learning III: Neural Networks and Backpropagation
Week 6 : Search
Week 7 : Markov Decision Processes
Week 8 : Midterm
Week 9 : Reinforcement Learning
Week 10 : Games
Week 11 : Hidden Markov Models and Bayesian Networks
Week 12 : Constraint Satisfaction Problems
Week 13 : Ethics and Explainability



Search



Search
Markov decision processes
Games

State-based models

Constraint satisfaction problems
Markov networks
Bayesian networks

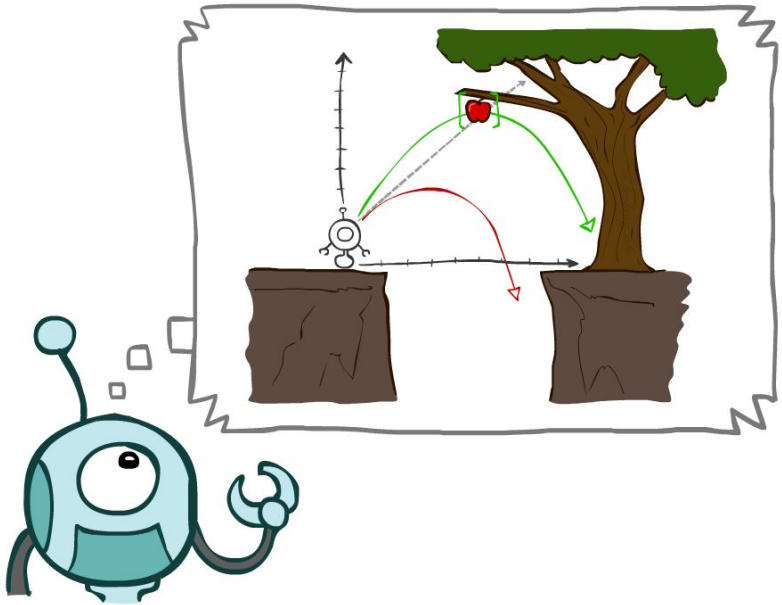
Variable-based models

Reflex-based models

Logic-based models

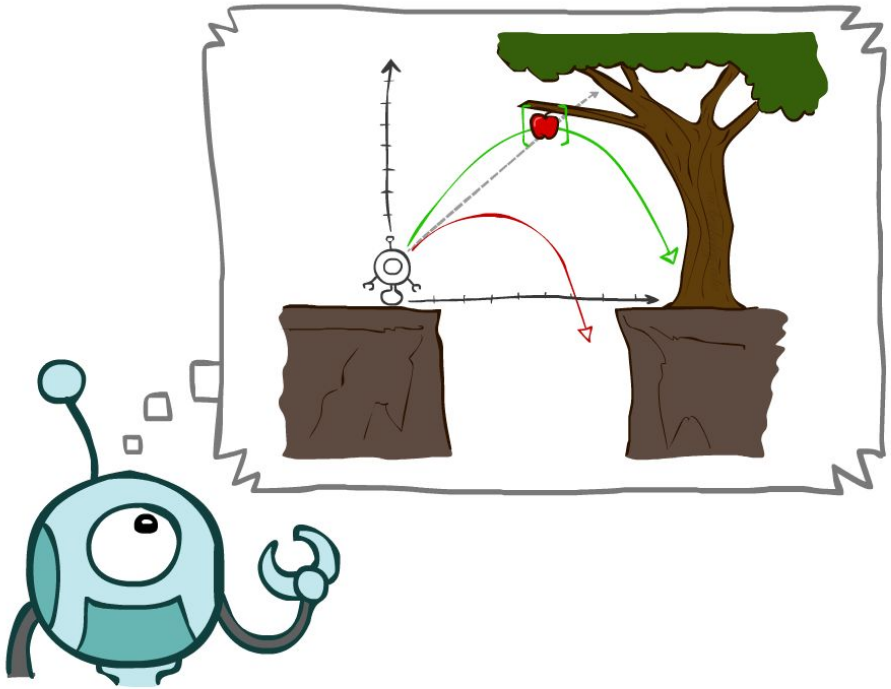
Today's Plan

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods Ch. 3.1 to 3.4
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Informed Search Heuristics Ch. 3.5 to 3.6
  - Greedy Search
  - A* Search



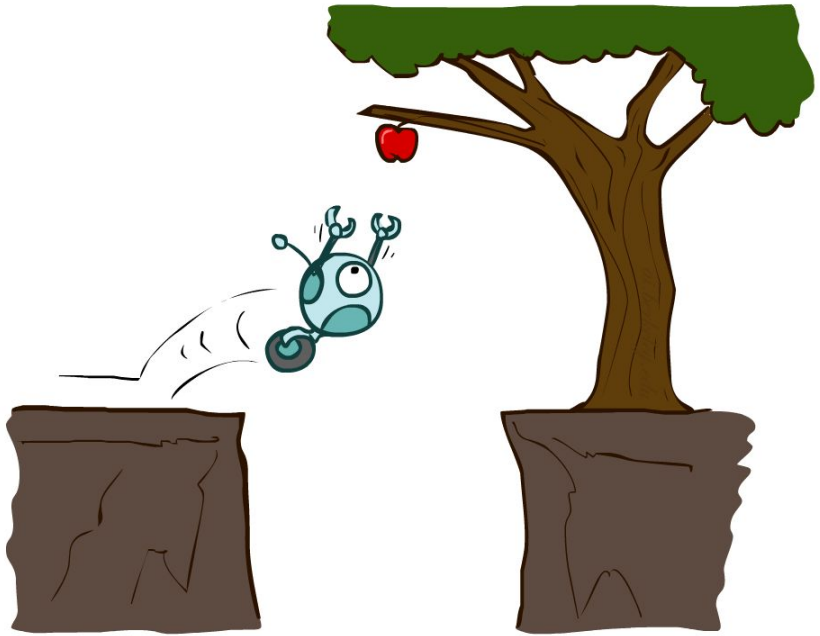
Slide Credits: Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).

Agents that Plan



Reflex Agents

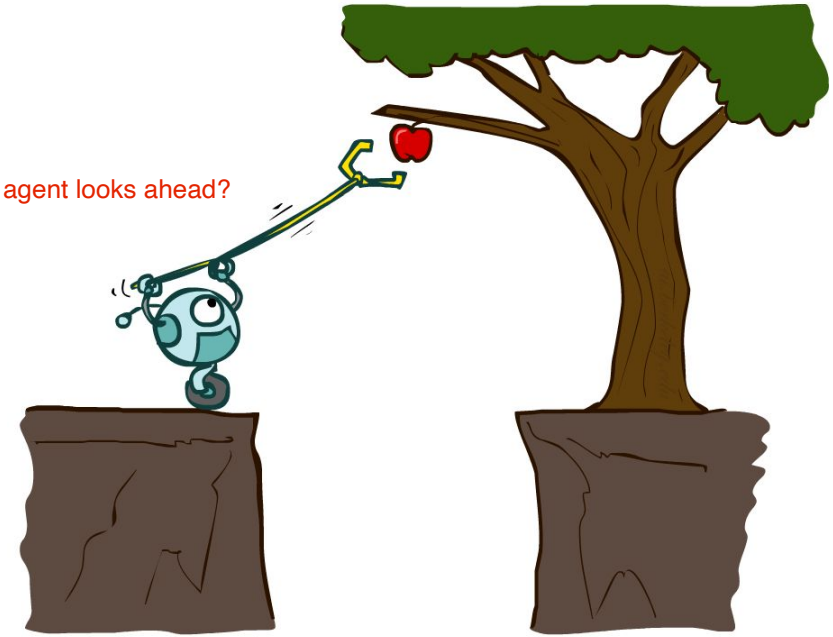
- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS
- Can a reflex agent be rational?



Planning Agents

- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE

Should we prune how far the agent looks ahead?



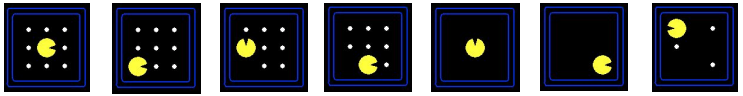
Search Problems

Search Problems

- A search problem consists of:
  - A state space

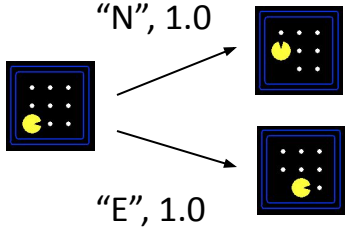
    What happens if the state space if extremely large?

    9 * 2^9 possible states

    A potential starting state

  - A successor function (with actions, costs)

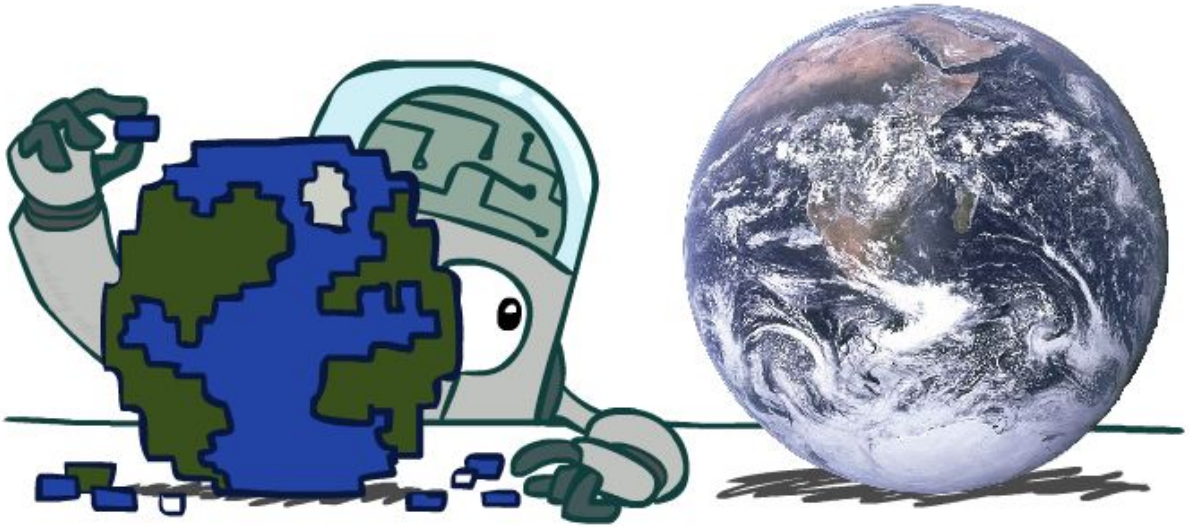
    

    The successor function can have constraints, but they can be relaxed

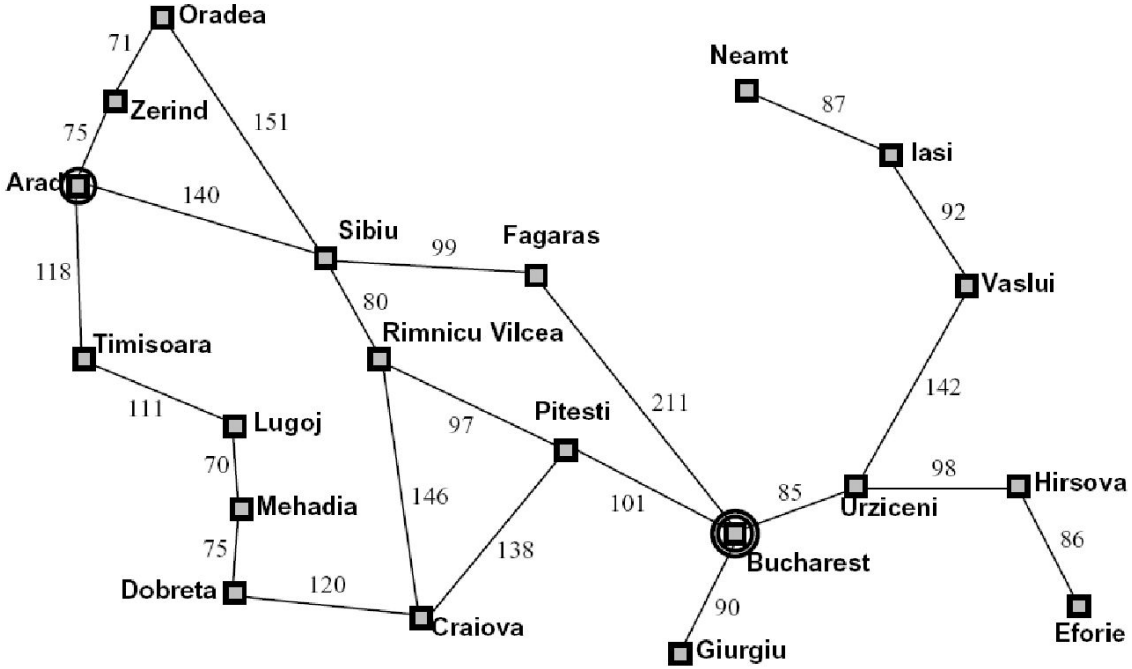
  - A start state and a goal test (am I done?)

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

Search Problems Are Models

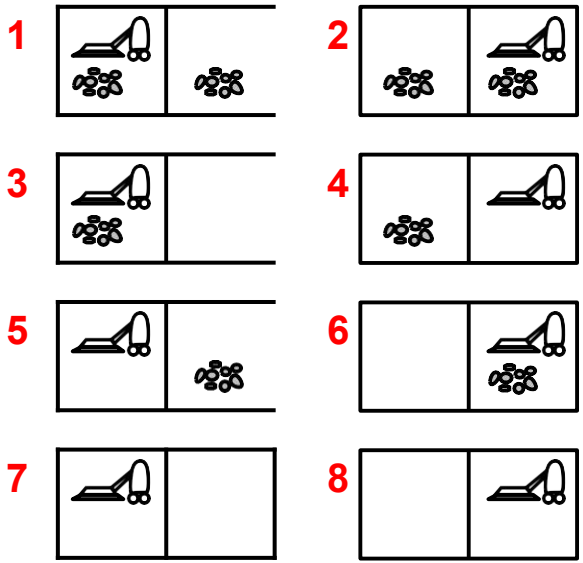


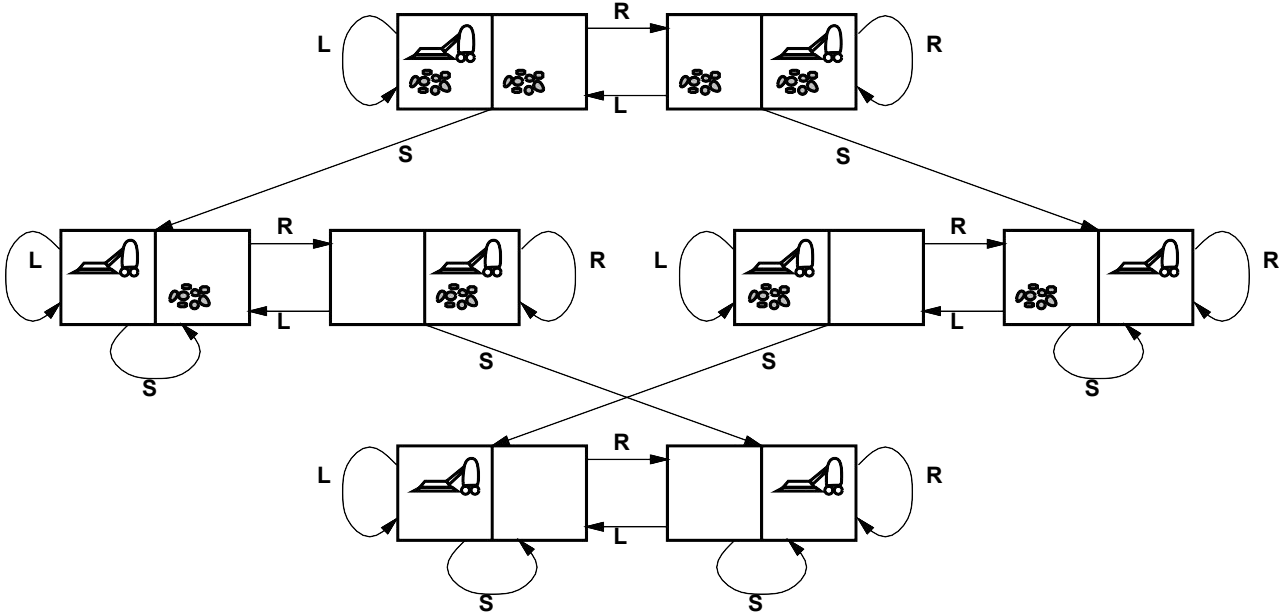
Example: Traveling in Romania



- State space:
  - Cities
- Successor function:
  - Roads: Go to adjacent city with cost = distance
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

Example: Vacuuming Robot





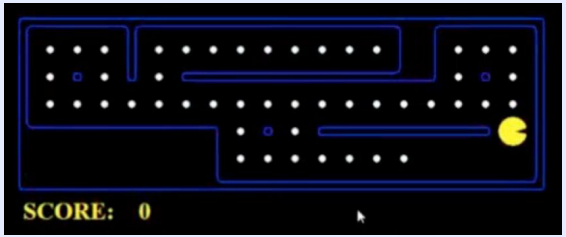
states: {1, 2, 3, 4, 5, 6, 7, 8} // integer dirt and robot locations (ignore dirt amounts etc.)
actions: *Left*, *Right*, *Suction*, *NoOp*
goal test: no dirt
path cost: 1 per action (0 for *NoOp*)

© 2021 Pearson Education Ltd.

What's in a State Space?

The world state includes every last detail of the environment

These examples assume that the agent knows everything about the world state



A search state keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

NSEW = go either North, South, East, or West

- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

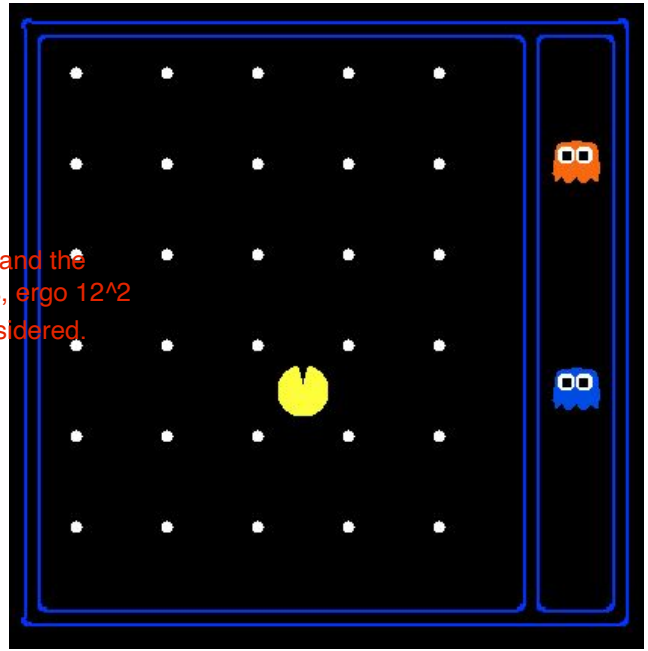
State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

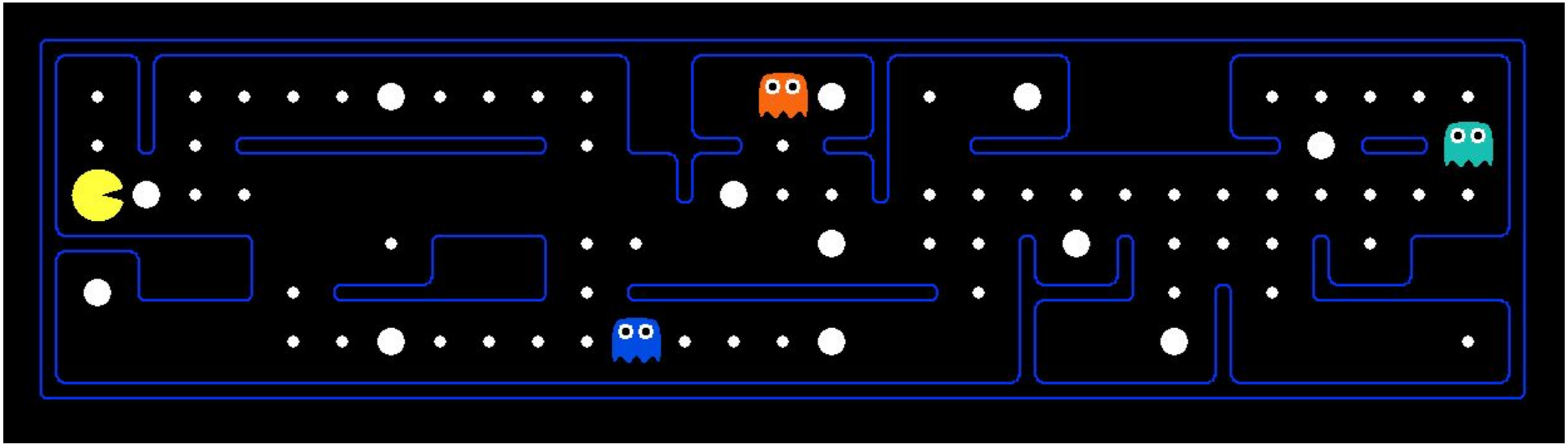
- How many
  - World states?
    $120x(2^{30})x(12^2)x4$
  - States for pathing?
    120
  - States for eat-all-dots?
    $120x(2^{30})$

Each ghost can be in 12 positions, and the ghosts can overlap. There are 2 ghosts, ergo 12^2

No direction for the ghosts is considered.



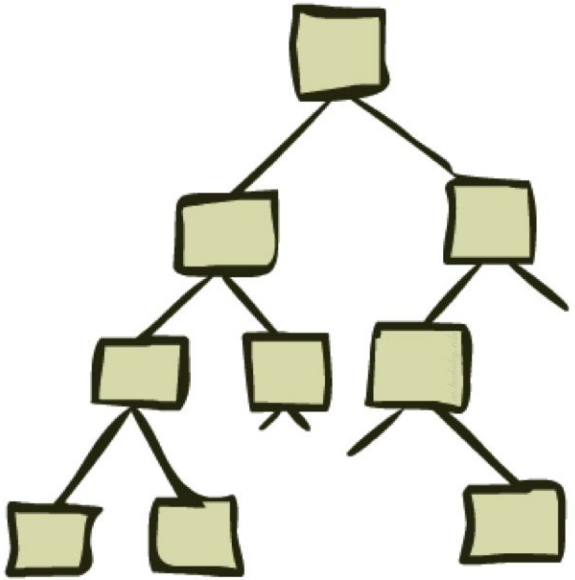
Quiz: Safe Passage



- Problem: eat all dots while keeping the ghosts perma-scared
- What does the state space have to specify?
  - (, , power pellet booleans, remaining scared time)

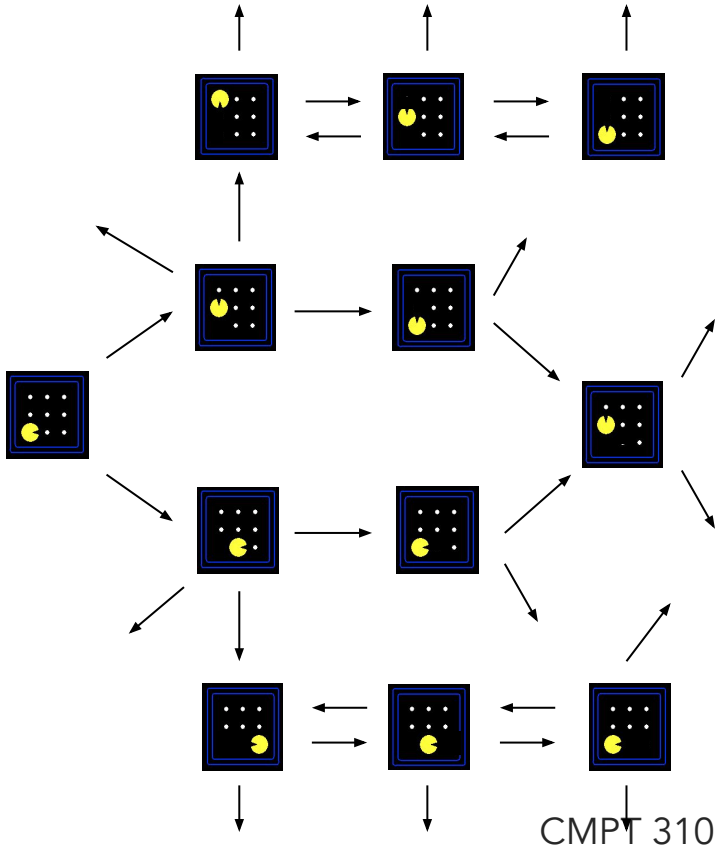
We need the position of pacman and the the state of the dots.

State Space Graphs and Search Trees



State Space Graphs

- State space graph: A mathematical representation of a search problem
  - **Nodes** are (abstracted) world configurations
  - **Arcs** represent successors (action results)
  - The **goal test** is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

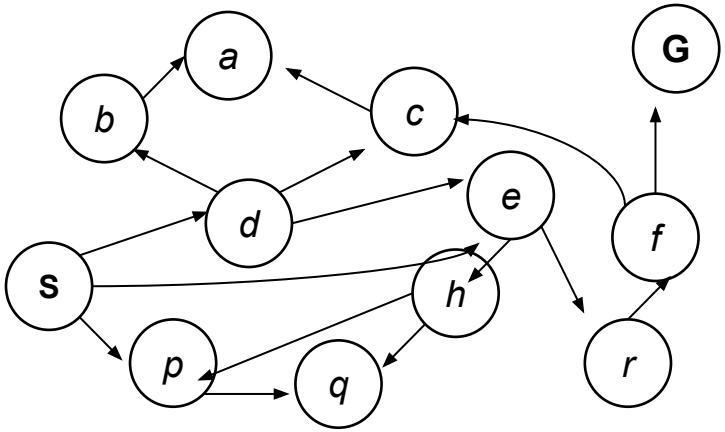


State Space Graphs

- State space graph: A mathematical representation of a search problem
  - **Nodes** are (abstracted) world configurations
  - **Arcs** represent successors (action results)
  - The **goal test** is a set of goal nodes (maybe only one)

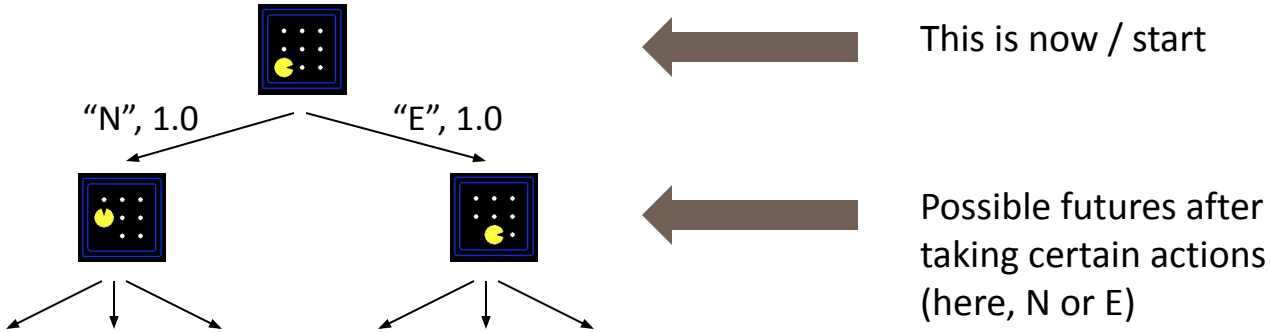
- In a state space graph, each state occurs only once!

- We can rarely build this full graph in memory (it's too big), but it's a useful idea



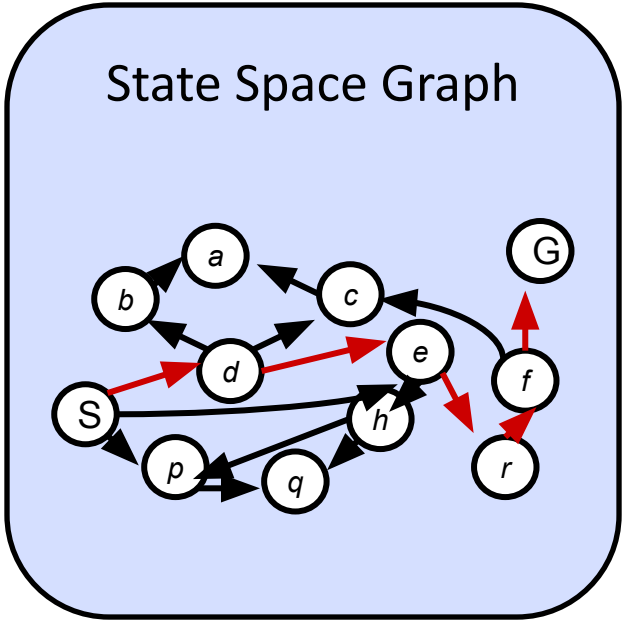
Tiny state space graph for a tiny search problem

Search Trees



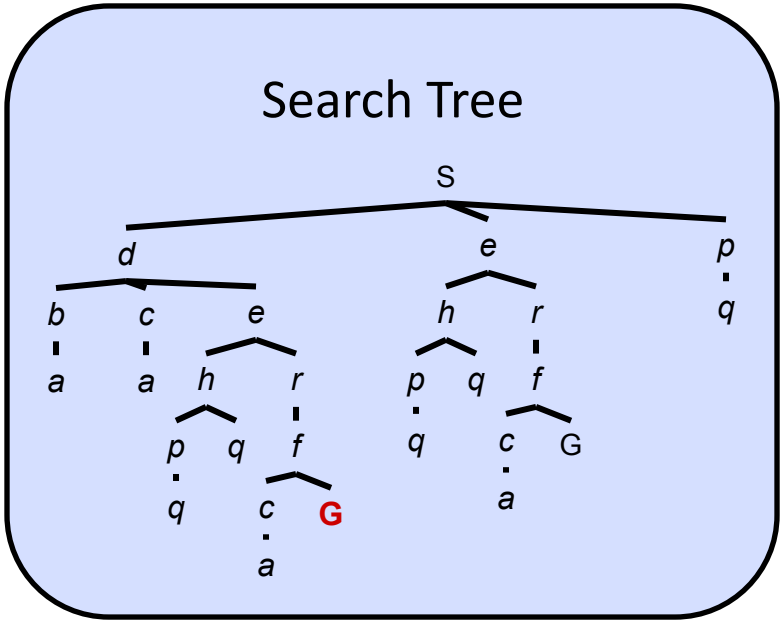
- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

State Space Graphs vs. Search Trees



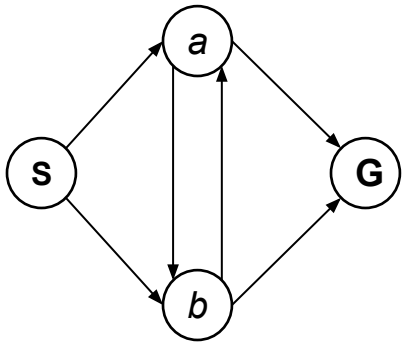
Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.



Quiz: State Space Graphs vs. Search Trees

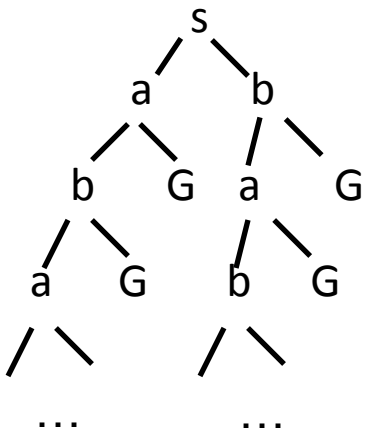
Consider this 4-state graph:



Since you can move from a to b, the tree is infinitely large.

How big is its search tree (from S)?

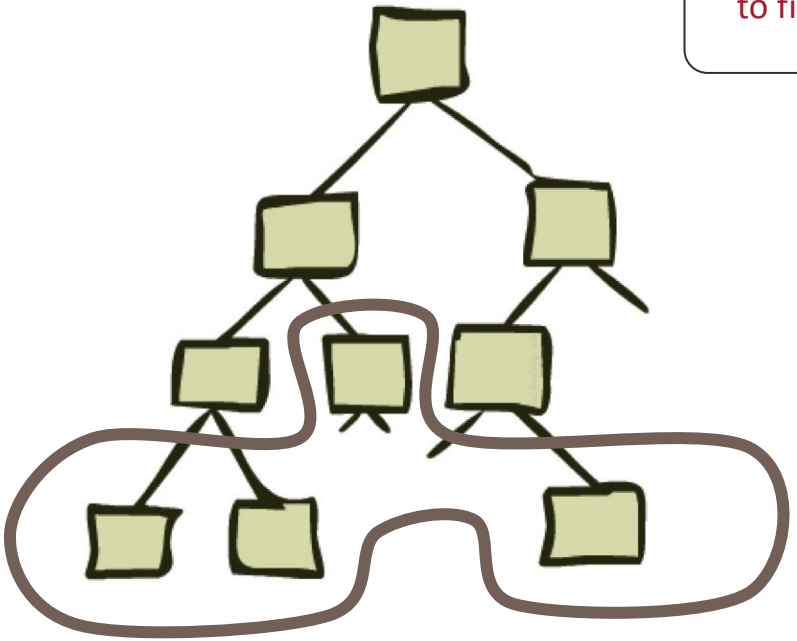
In a search tree, the same state can occur more than once.



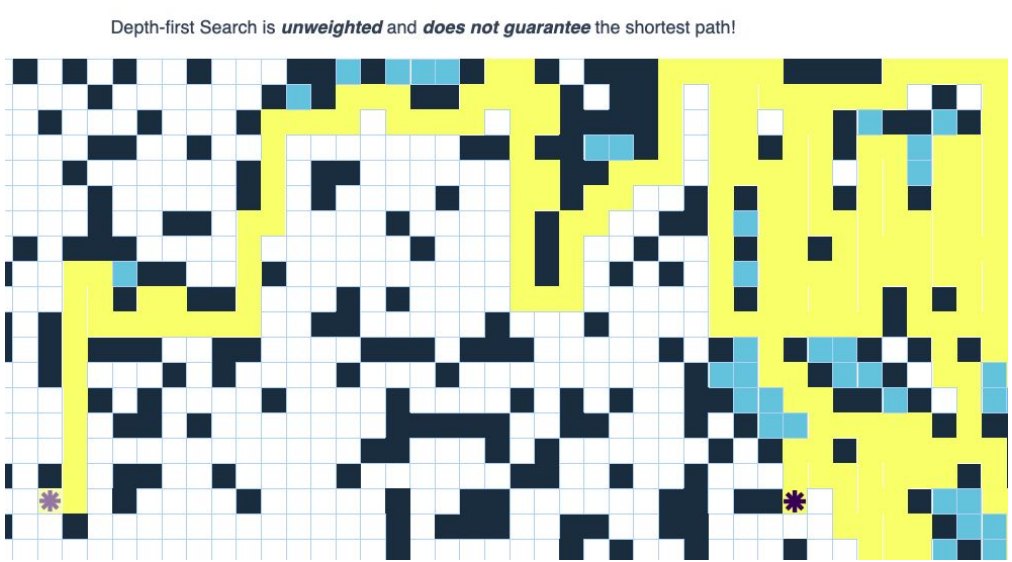
Important: Lots of repeated structure in the search tree!

Tree Search

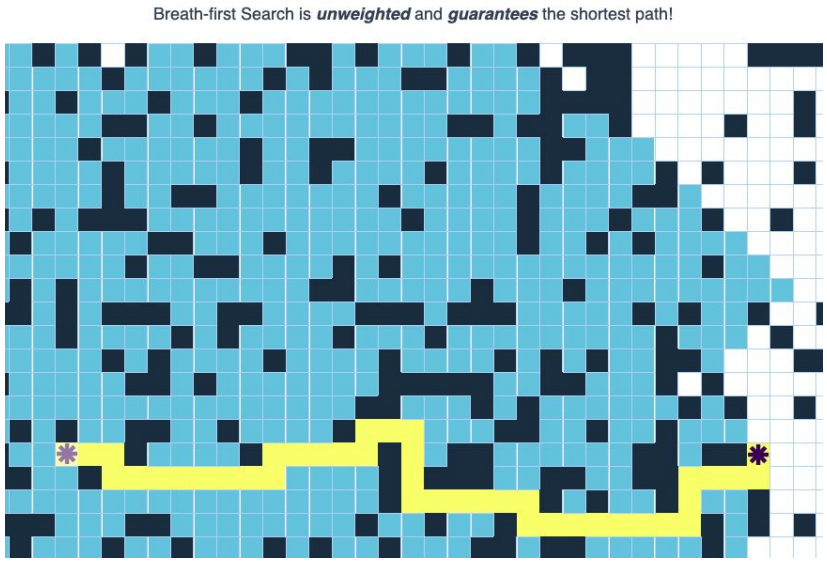
Remember binary search trees to find elements in arrays in CMPT 120?



Teaser



Depth First Search

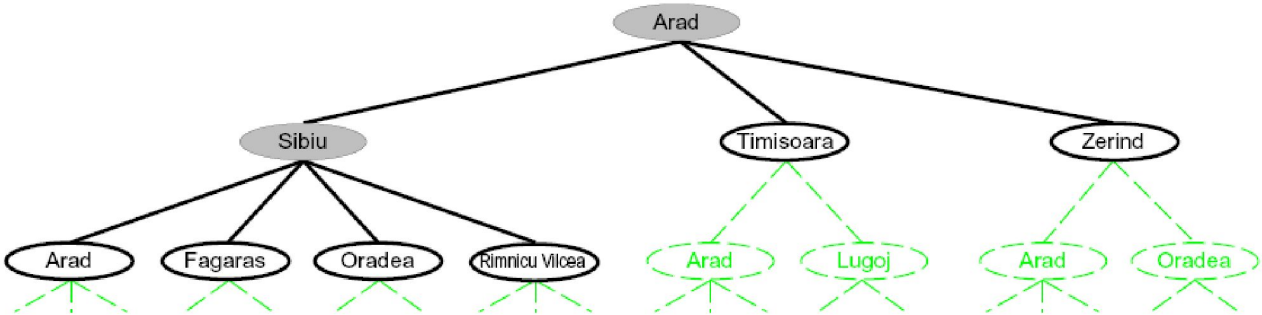


Breadth First Search

https://clementmihailescu.github.io/Pathfinding-Visualizer/

Unweighted Trees

Searching with a Search Tree



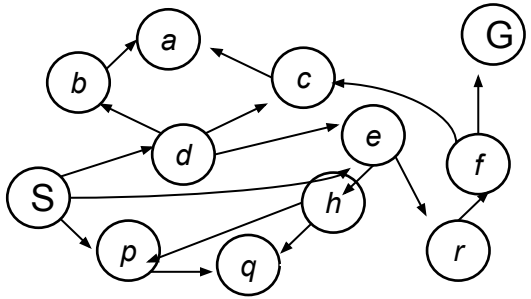
- Search, minimizing number of actions:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible

General Tree Search

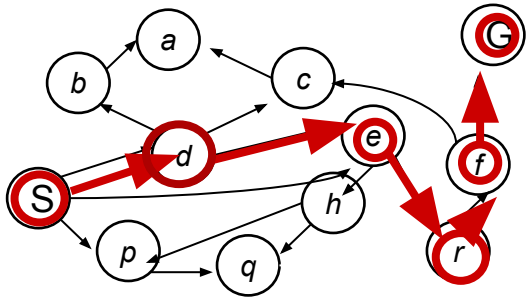
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to
        if the node contains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

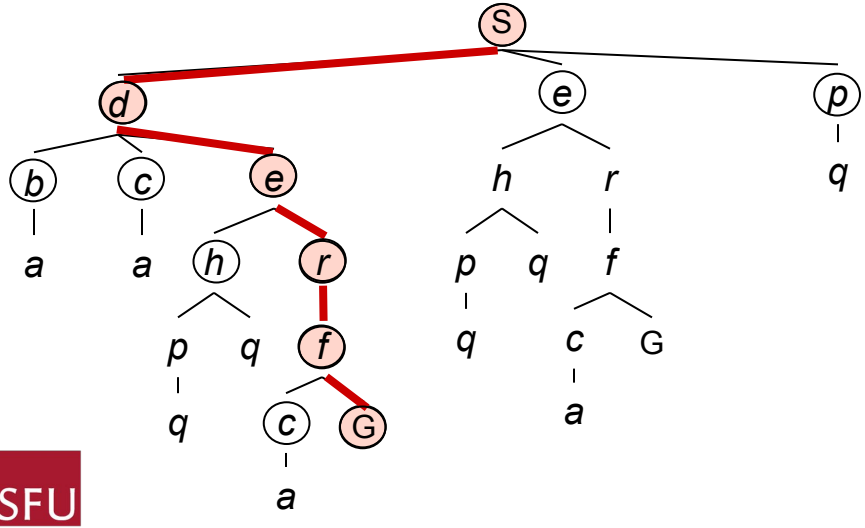
- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

Example: Tree Search



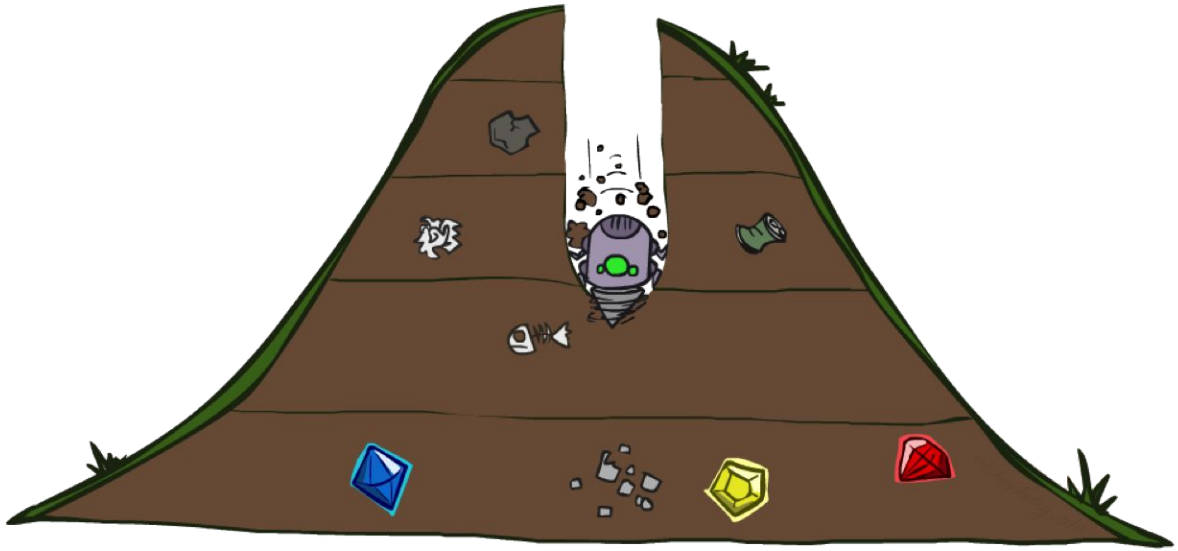
Example: Tree Search





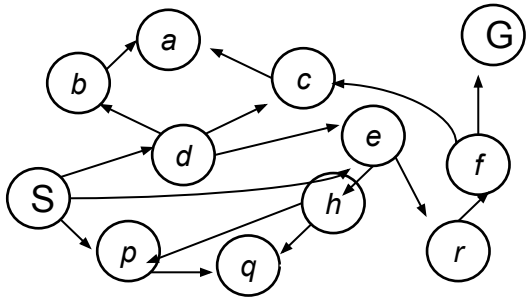
s
s -> d
s -> e
s -> p
s -> d -> b
s -> d -> c
s -> d -> e
s -> d -> e -> h
s -> d -> e -> r
s -> d -> e -> r -> f
s -> d -> e -> r -> f -> c
s -> d -> e -> r -> f -> G

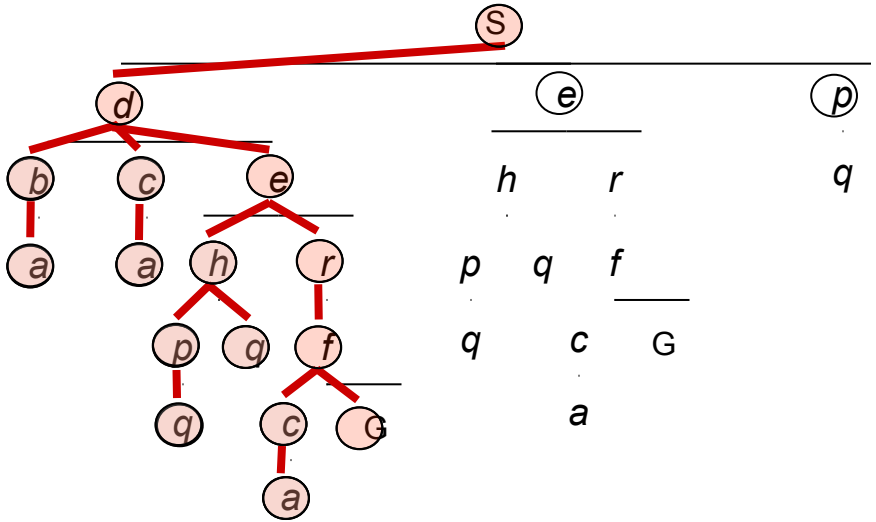
Depth-First Search



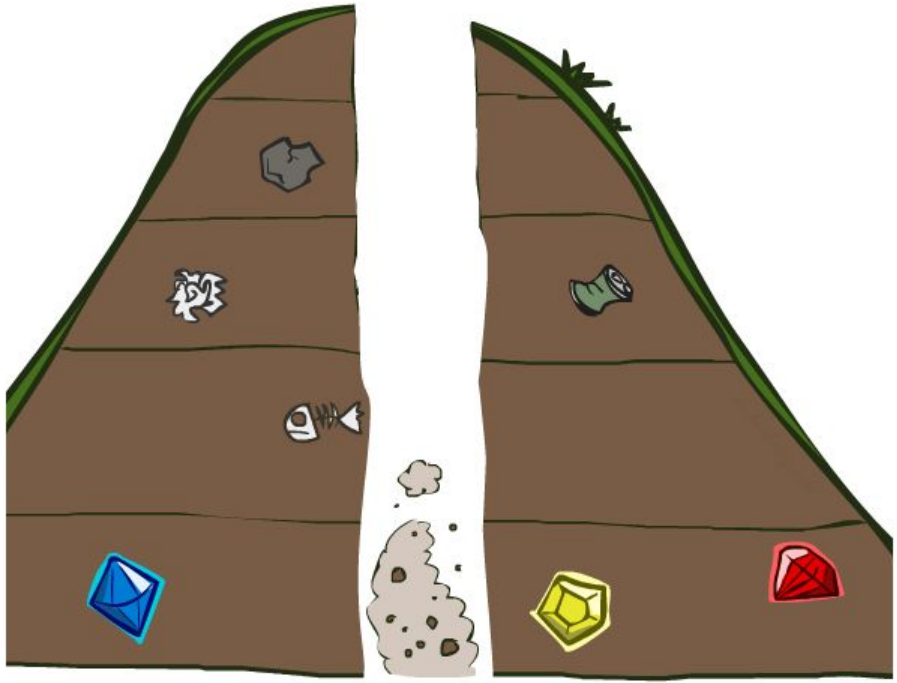
Depth-First Search

Strategy: expand a deepest node first





Search Algorithm Properties



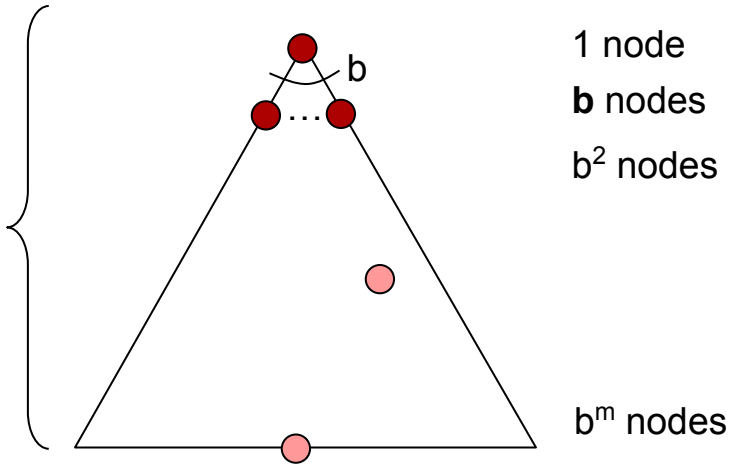
Search Algorithm Properties

- **Complete**: Is it *guaranteed* to **find** a solution if one exists?
- **Optimal**: Is it *guaranteed* to find the **least cost** path?
- **Time complexity?**
- **Space complexity?**

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

Tiers means height

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



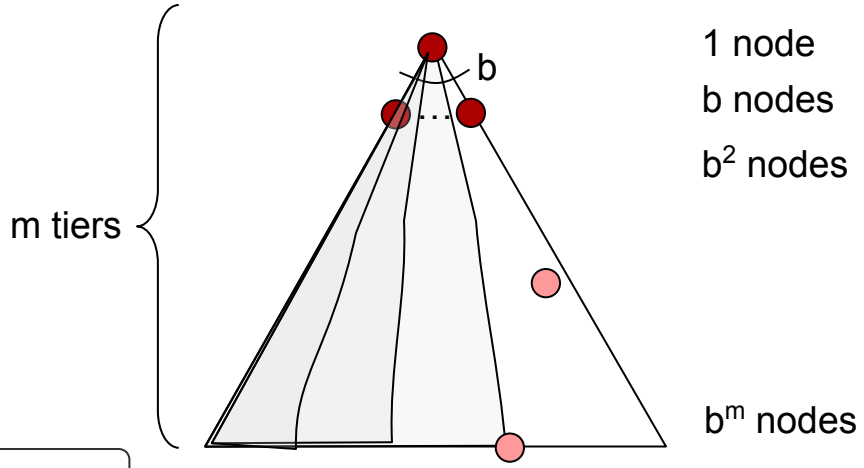
Depth-First Search (DFS) Properties

- What nodes does 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 that

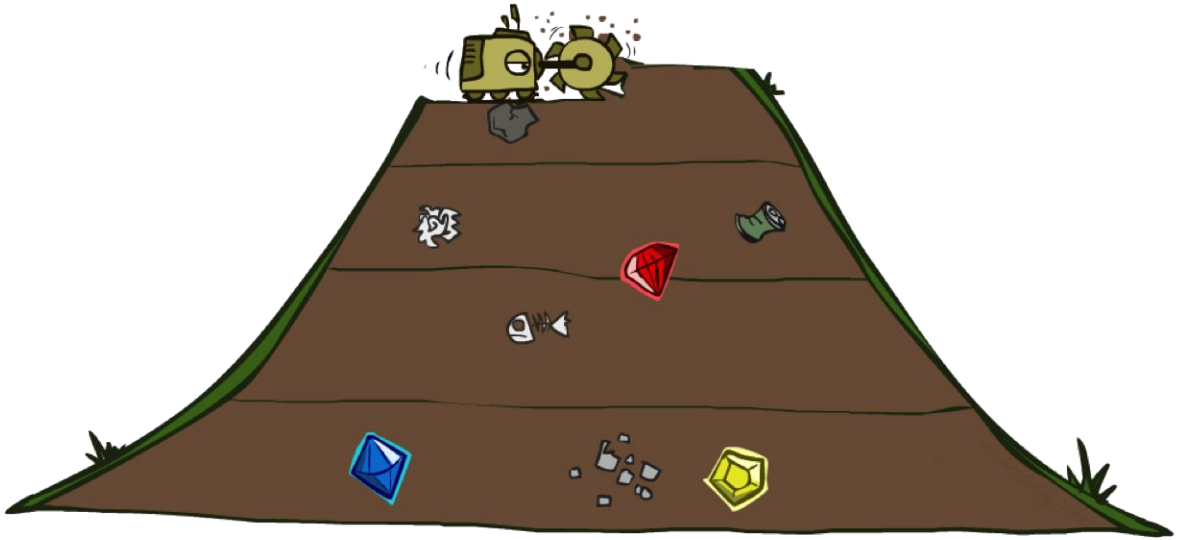
Guaranteed to find a solution if it exists?

- Is it **optimal**?
  - No, it finds the "leftmost" solution, regardless of depth or cost

Guaranteed to find the solution with least cost?



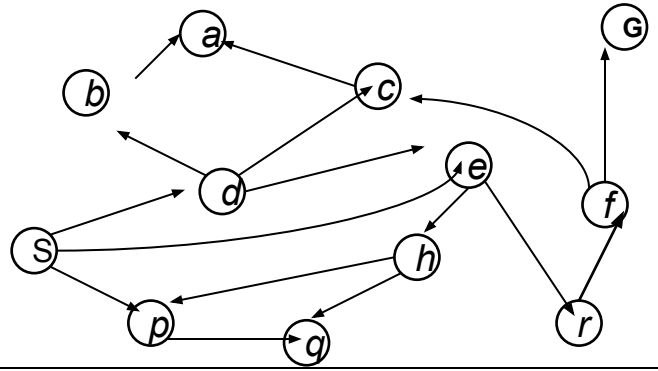
Breadth-First Search

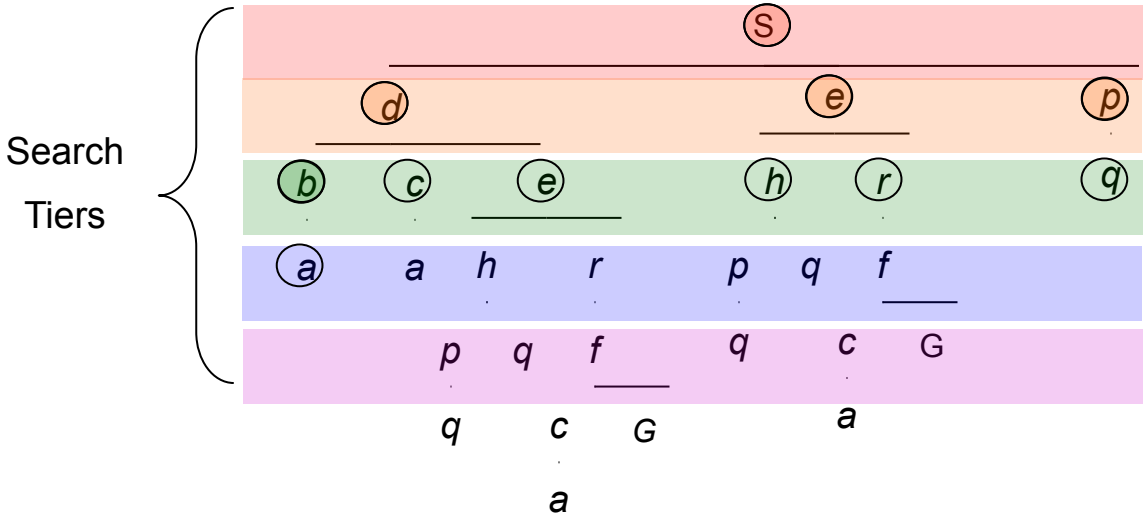


BFS finds the shortest path in terms of number of actions.

Breadth-First Search

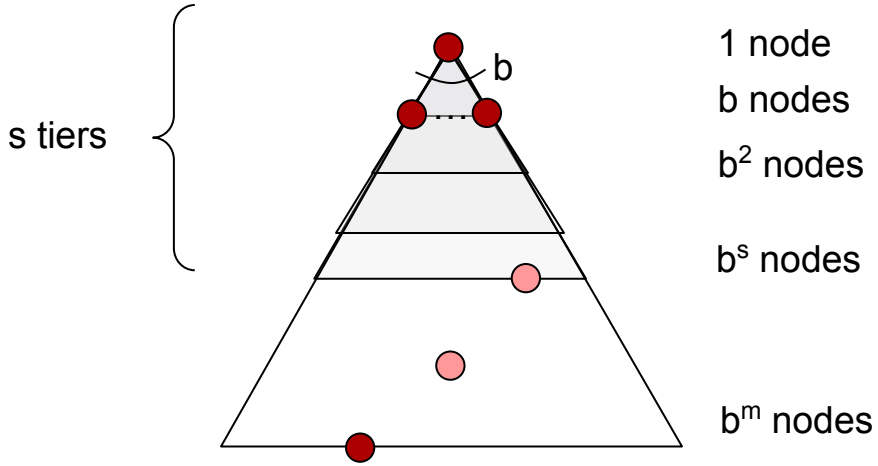
Strategy: expand a shallowest node first





Breadth-First Search (BFS) Properties

- 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)



Quiz: DFS vs BFS

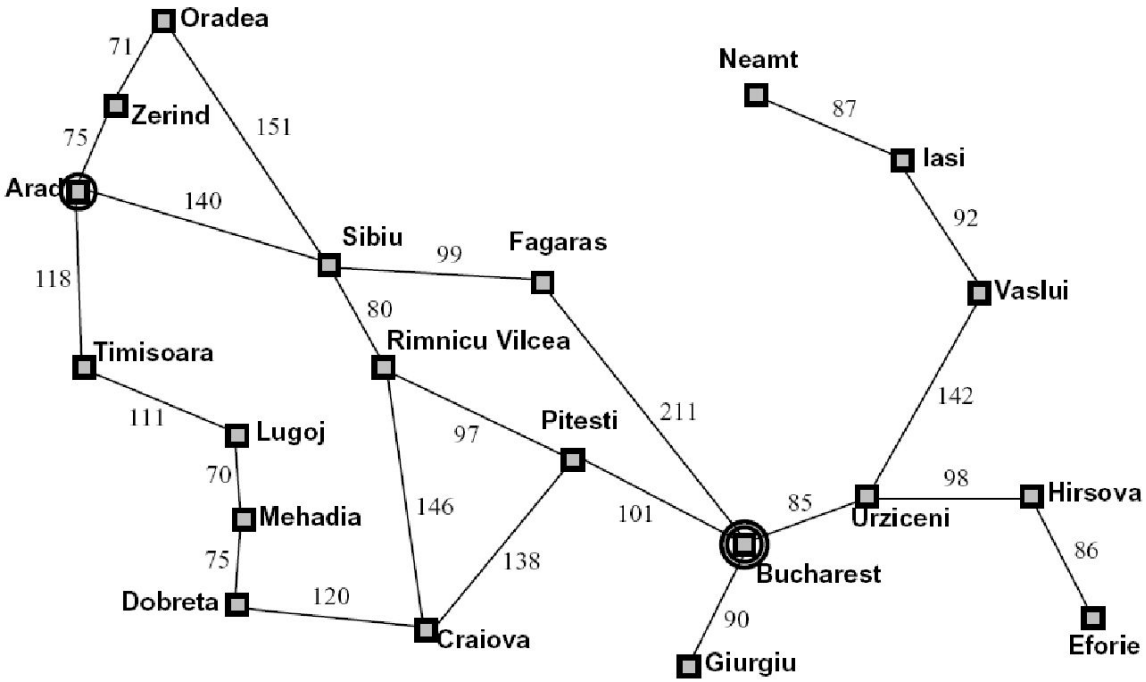
- When will BFS outperform DFS?

- When will DFS outperform BFS?

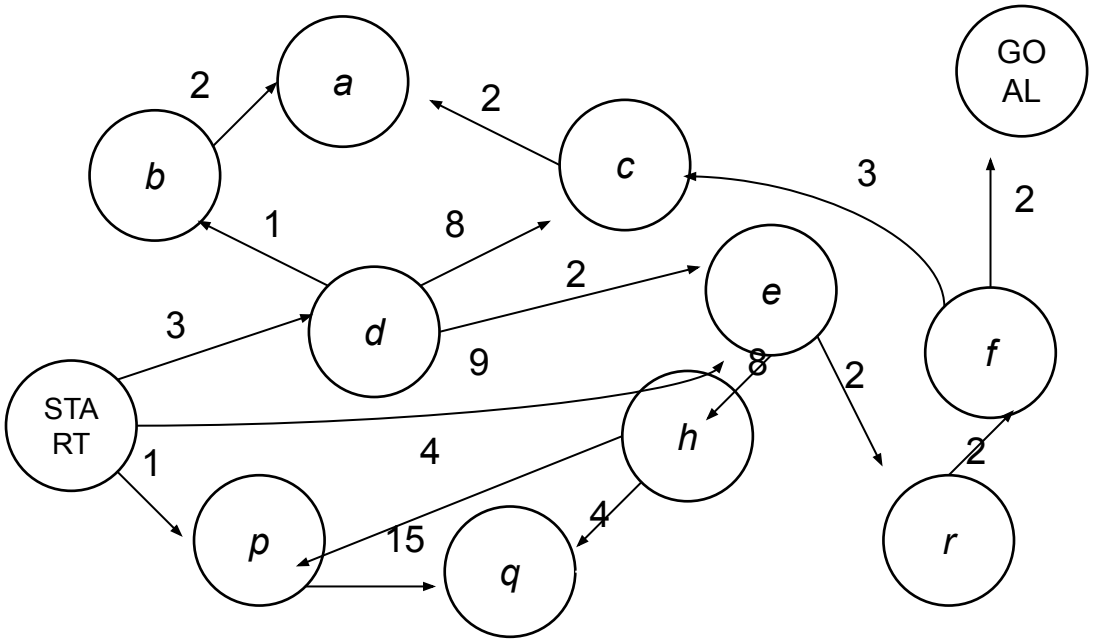
https://clementmihailescu.github.io/Pathfinding-Visualizer/

Weighted Trees

Search Example: Romania

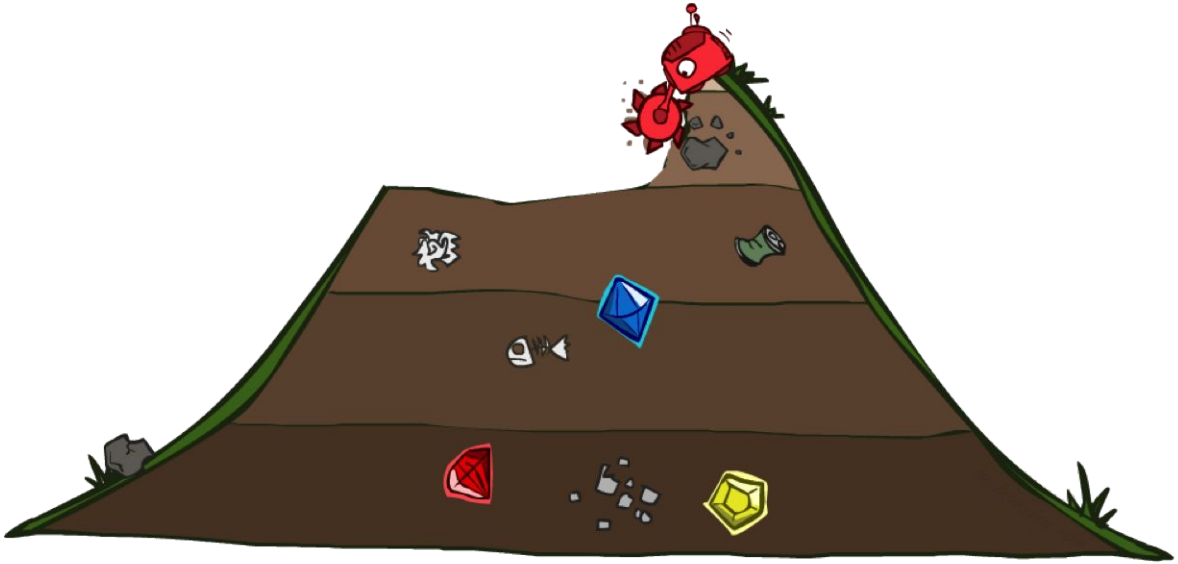


Cost-Sensitive Search



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**.

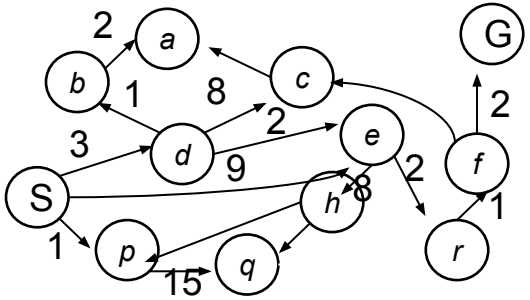
Uniform Cost Search
aka Dijsktra's Algorithm



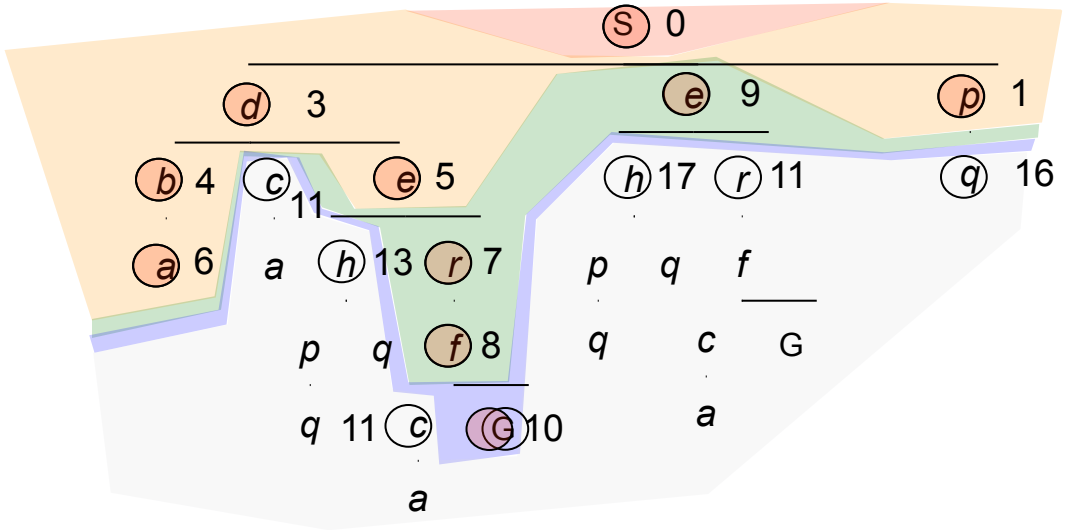
Consider all paths systematically in order of increasing cost

Uniform Cost Search

Strategy: expand a cheapest node first

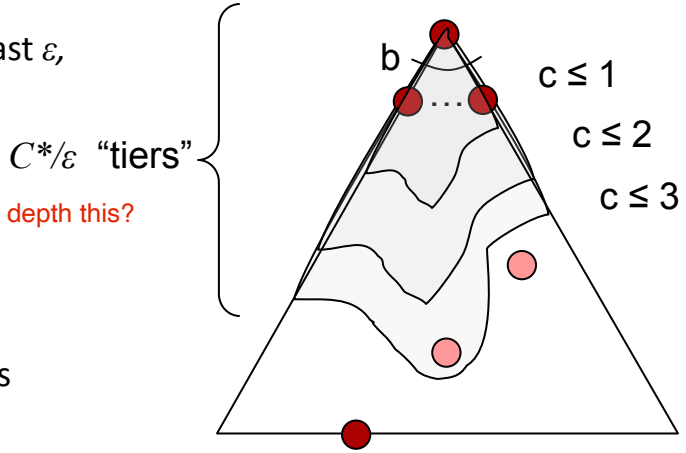


Cost contours



Uniform Cost Search (UCS) Properties

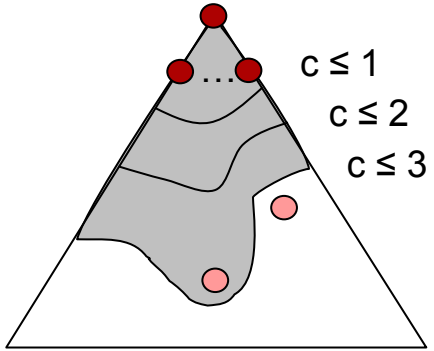
- What nodes does UCS expand?
  - Processes **all nodes with cost less than cheapest solution**!
  - If that optimal solution costs C^* and cost of each action is at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much **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! (Proof next lecture via A*)

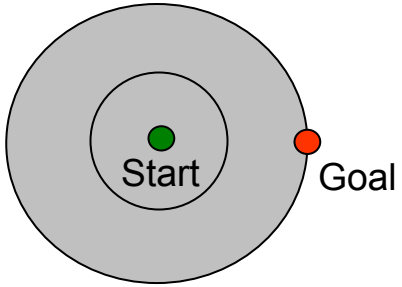


Why is the effective depth this?

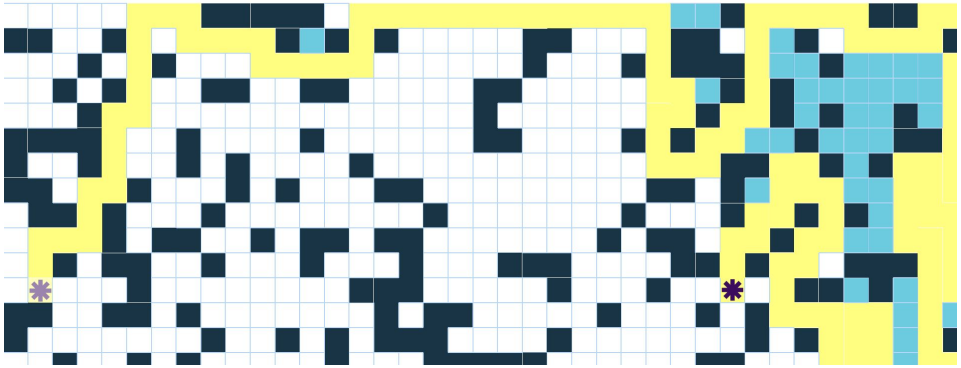
Uniform Cost Issues

- 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
- We’ll fix that soon!

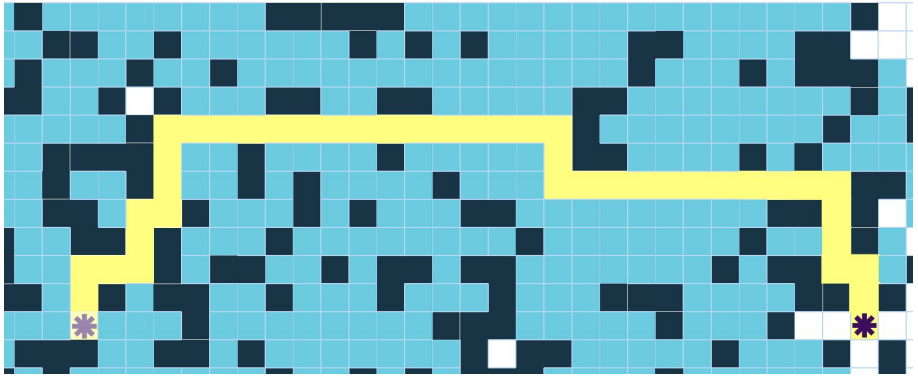




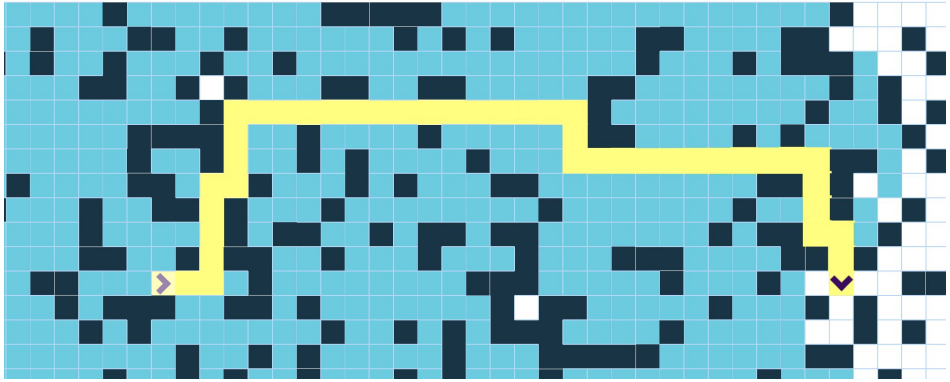
Depth-first Search is ***unweighted*** and ***does not guarantee*** the shortest path!



Breath-first Search is ***unweighted*** and ***guarantees*** the shortest path!



Dijkstra's Algorithm is ***weighted*** and ***guarantees*** the shortest path!

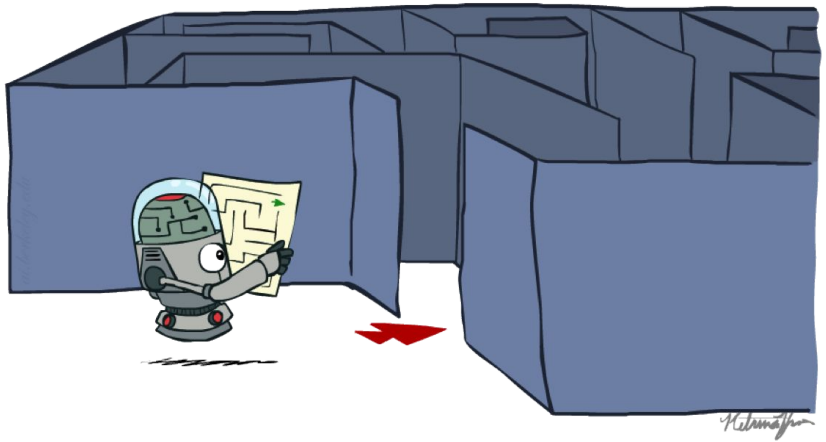


https://clementmihailescu.github.io/Pathfinding-Visualizer/

Informed Search

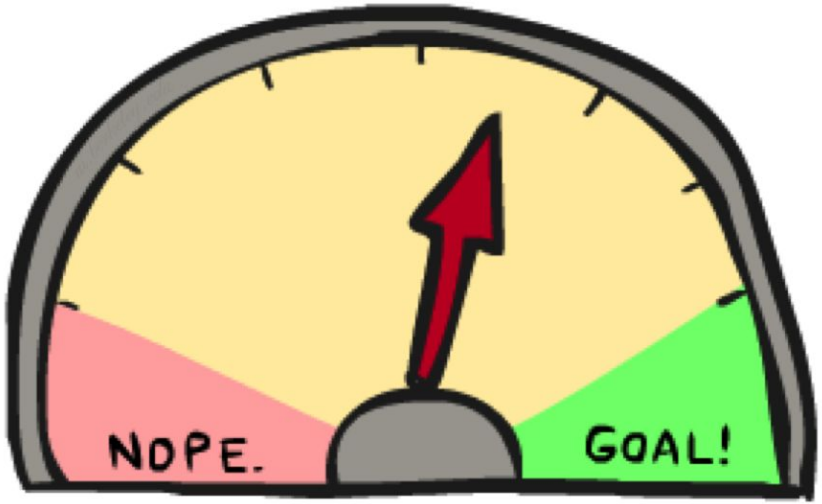
Recap: Search

- Search problem:
  - **States** (configurations of the world)
  - **Actions** and **costs**
  - Successor **function** (world dynamics)
  - **Start** state and **goal** test
- Search tree:
  - **Nodes**: represent plans for reaching states
  - Plans have **costs** (sum of action costs)
- Search algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
  - Optimal: finds least-cost plans



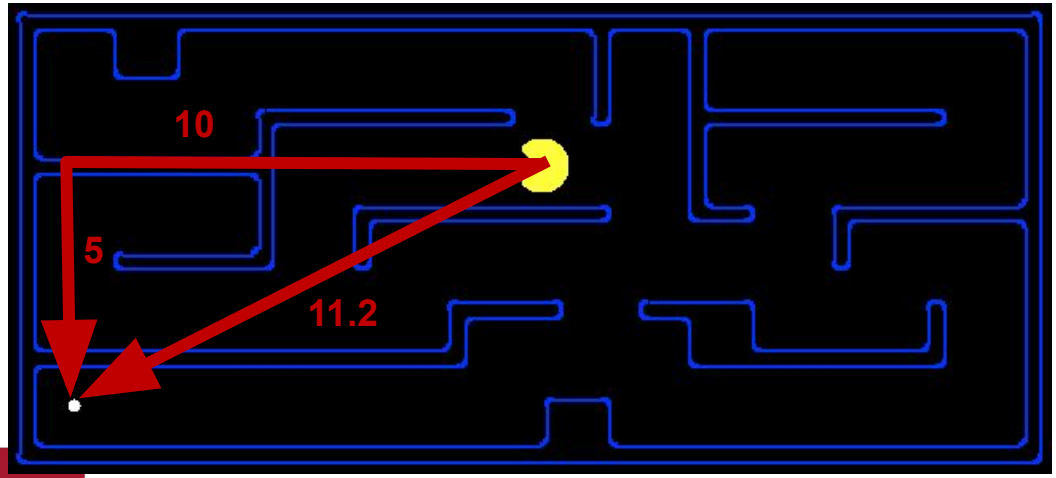
- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search

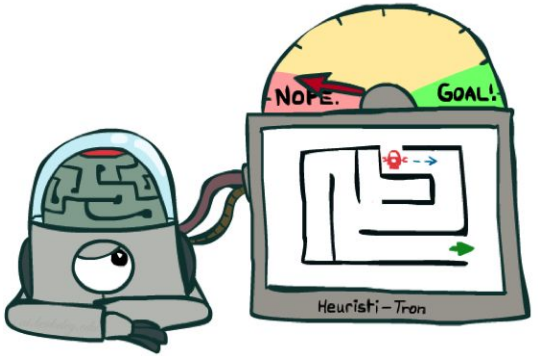
Informed Search

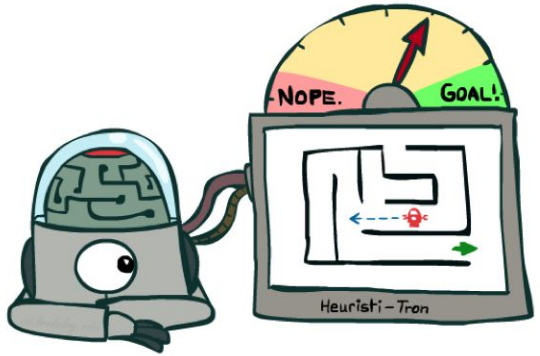


Search Heuristics

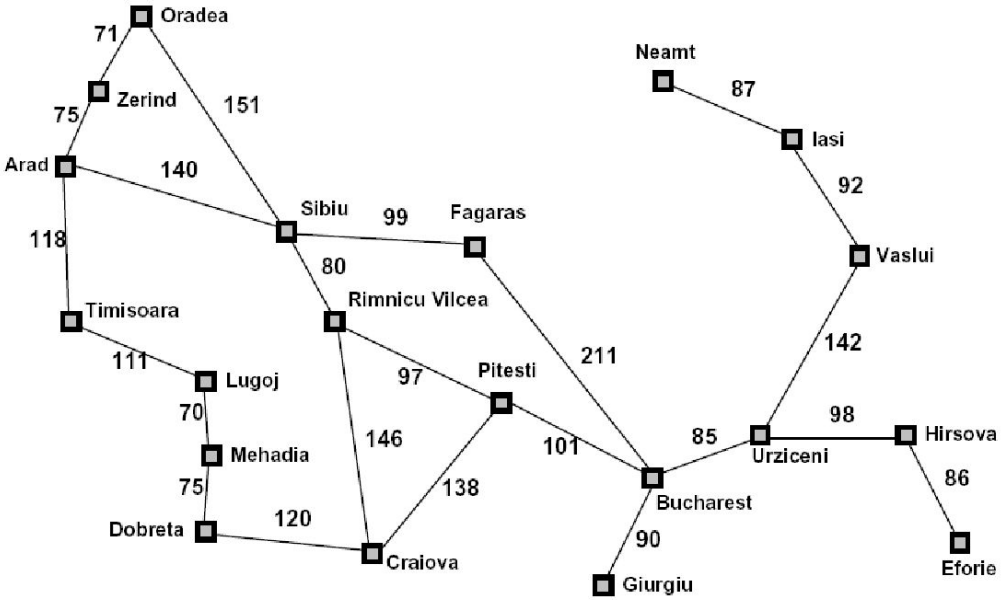
- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing







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

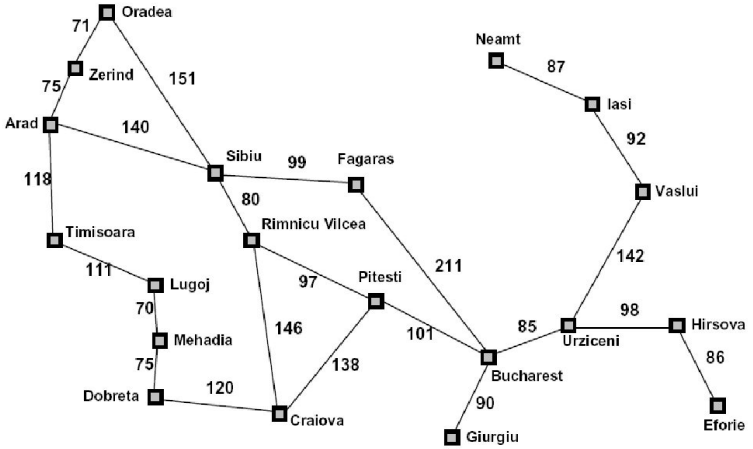
h(x)

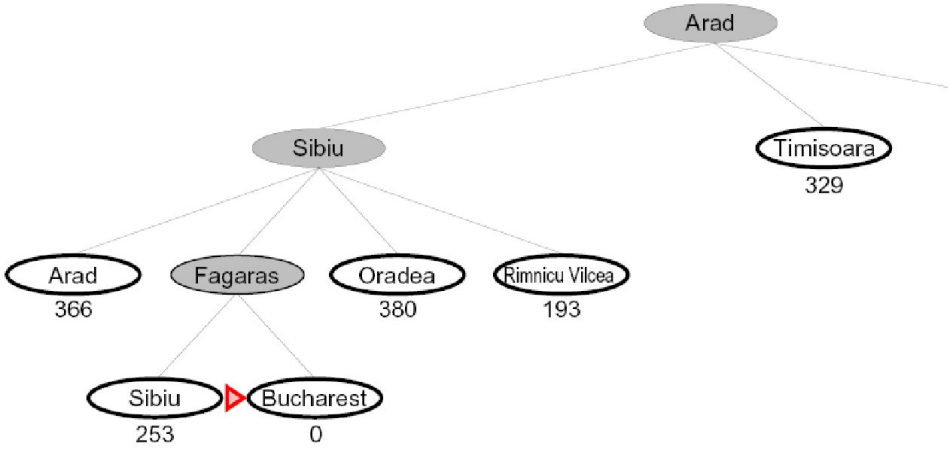
Greedy Search

Greedy Search

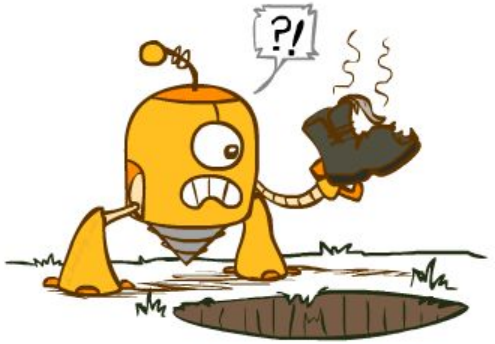
Expand the node that seems closest to the goal state.

- Expand the node that seems closest…



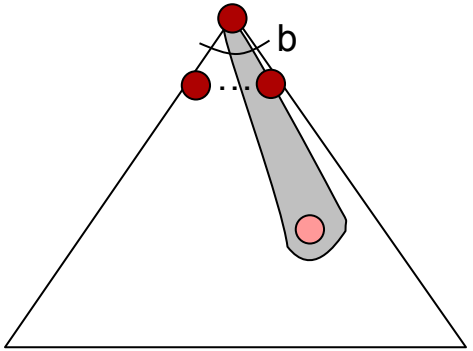


- What can go wrong?

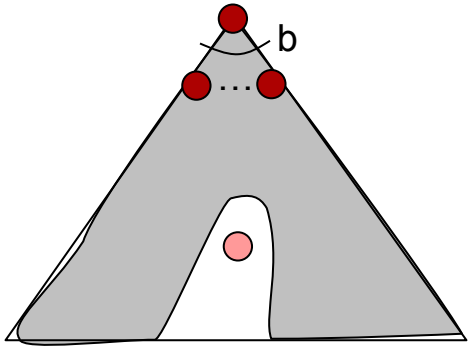


Greedy Search

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

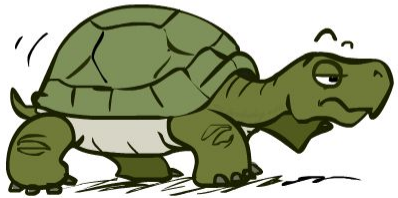


- A common case:
  - Best-first takes you straight to the (wrong) goal

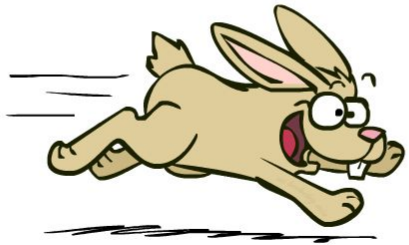


- Worst-case: like a badly-guided DFS

A* Search



UCS

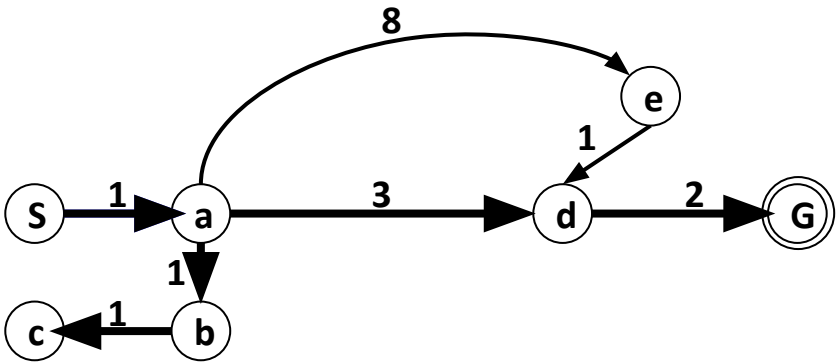


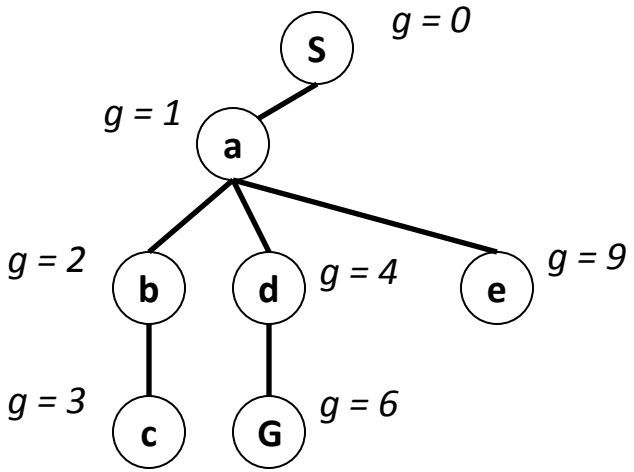
Greedy with a heuristic

A*

Uniform-Cost Search

Uniform-cost orders by path cost, or *backward cost* g(n)

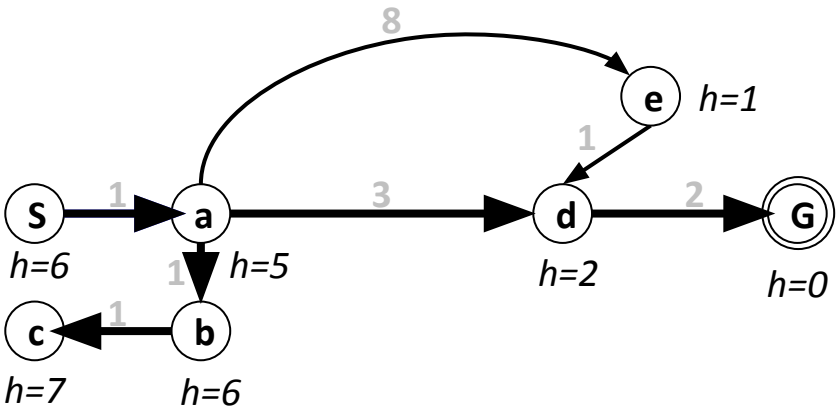


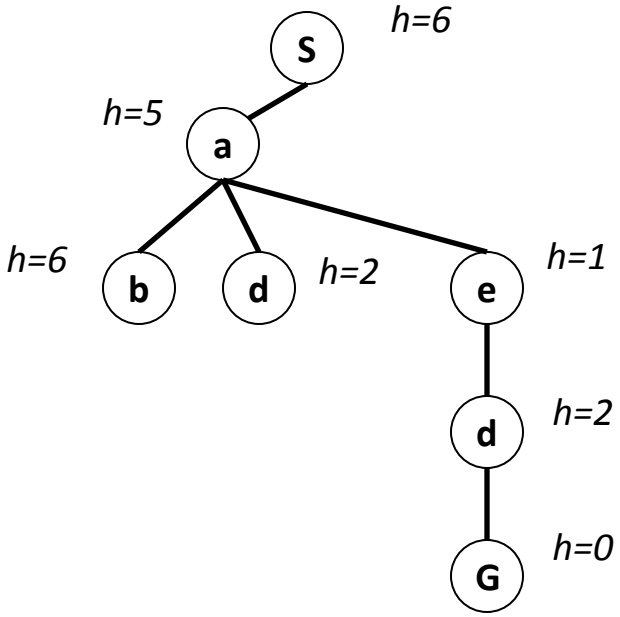


Greedy Search

Greedy orders by goal proximity, or *forward cost* h(n)

Are we always trying to minimize the forward cost?
h represents the distance from a node to the target node.

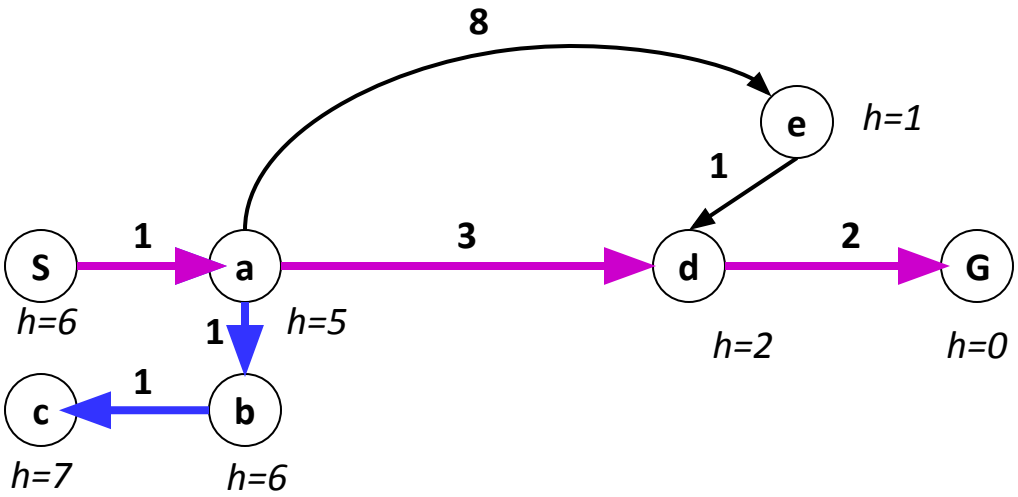




Example: Teg Grenager

Combining UCS and Greedy

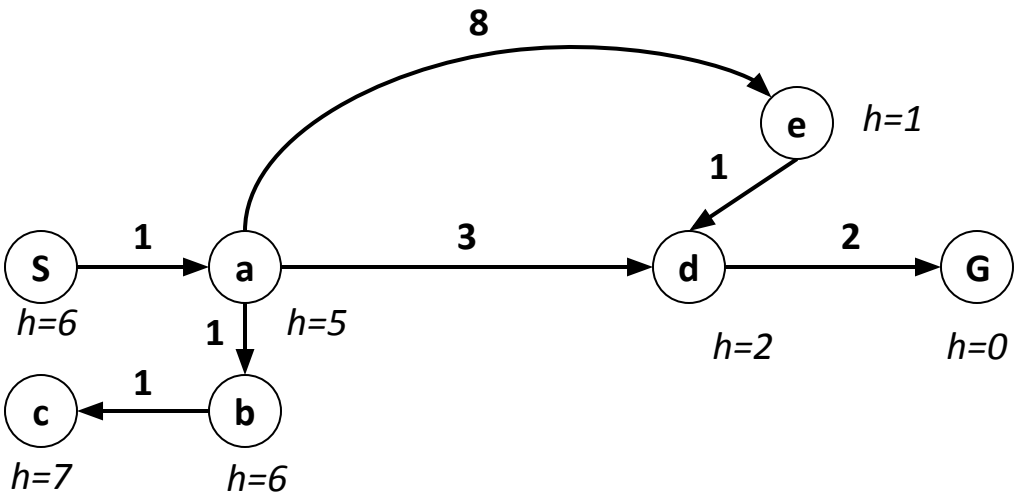
- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)

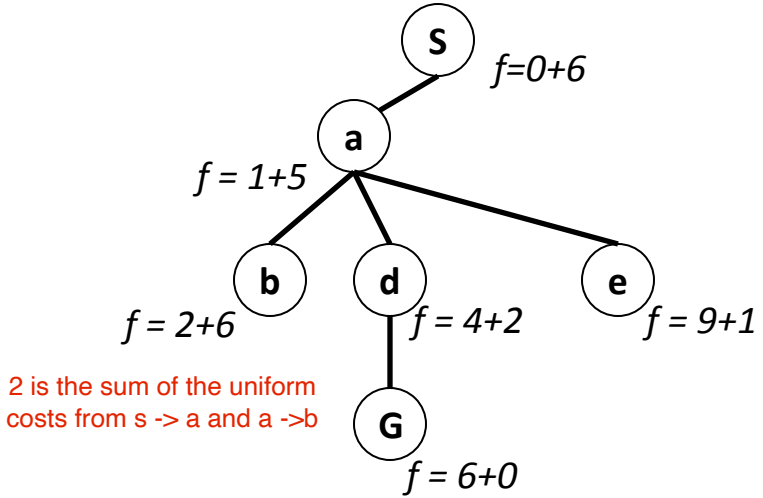


- A* Search orders by the sum: f(n) = g(n) + h(n)

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)

Example: Teg Grenager

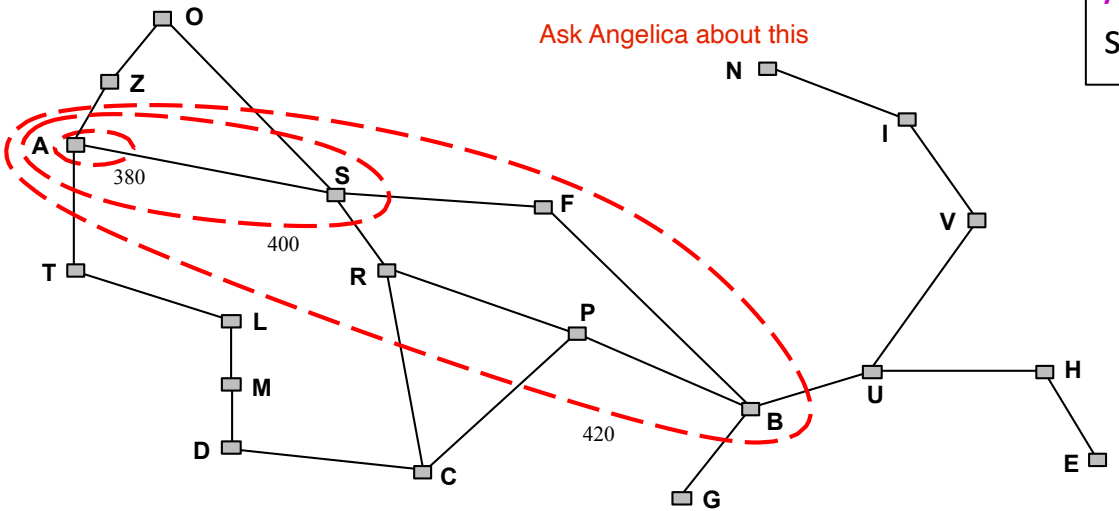
Intuition behind A*

Idea: avoid expanding paths that are already expensive

A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)

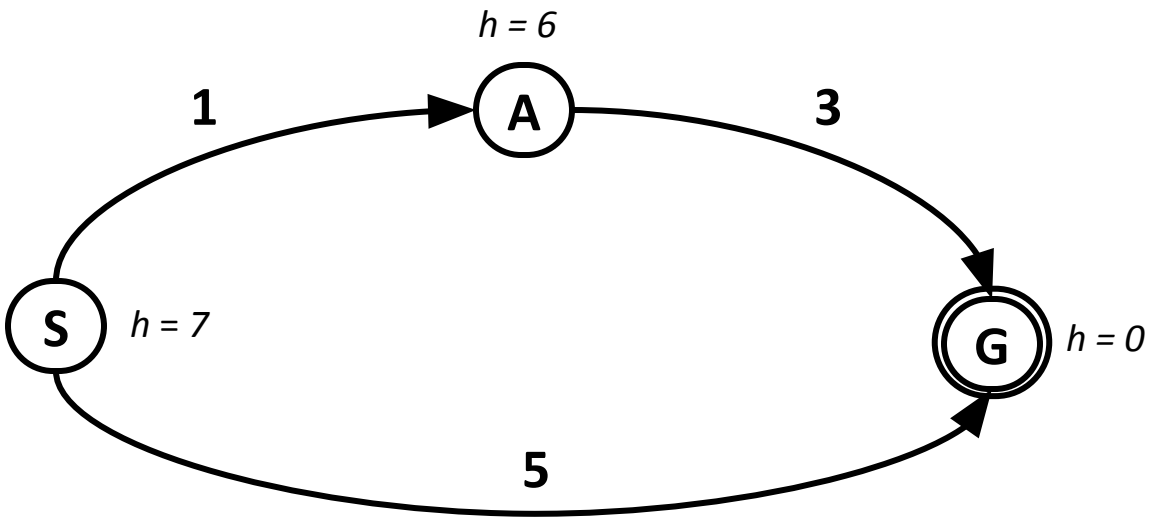
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

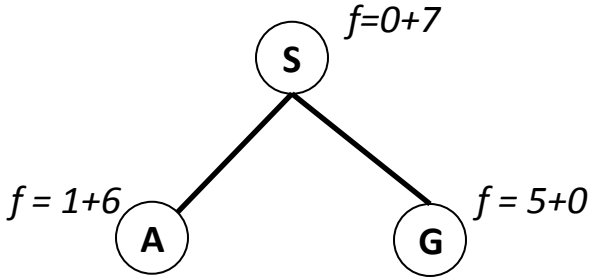


A* Search orders by the sum: f(n) = g(n) + h(n)

71

Is A* Optimal?



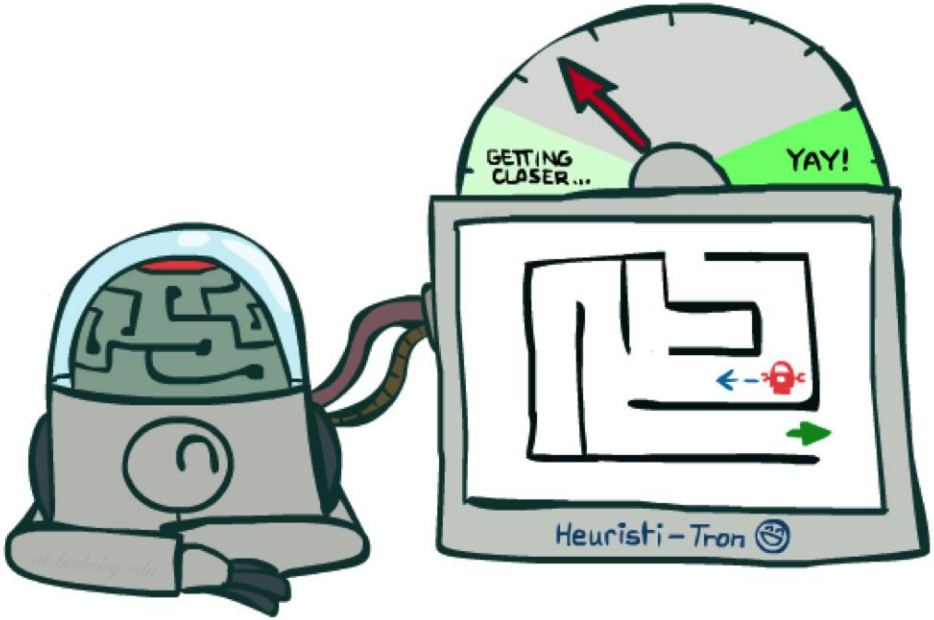


Why does A* fail here?

In the previous examples, A* goes in the direction of the node which minimizes f(n) and g(n)

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

Admissible Heuristics



A heuristic is **admissible** if it never overestimates the cost of reaching the goal.

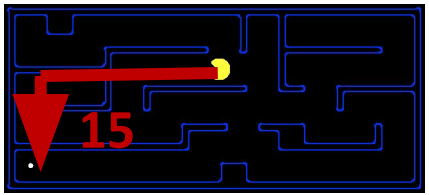
Admissible Heuristics

- 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.

How does A* work?

cost so far to reach n

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

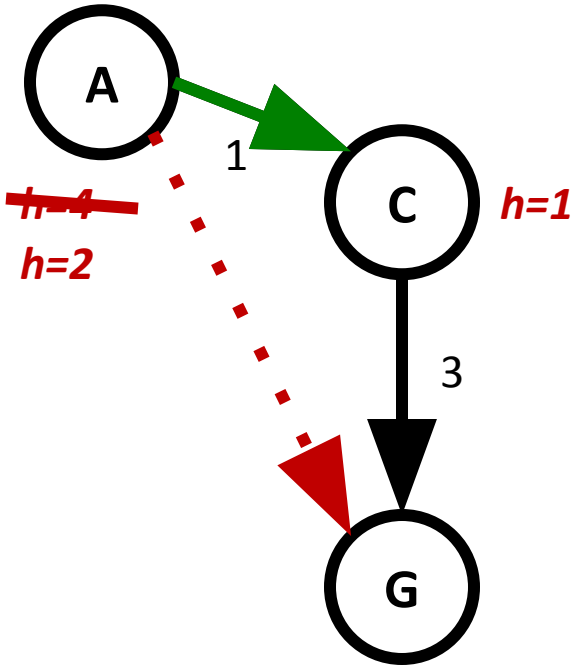
estimated total cost of path through n to goal

estimated cost to goal from n

A* search uses an **admissible heuristic** $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n.

(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

Consistency of Heuristics



- **Main idea:** estimated heuristic costs ≤ actual costs
  - **Admissibility**: heuristic cost ≤ actual cost to **goal**

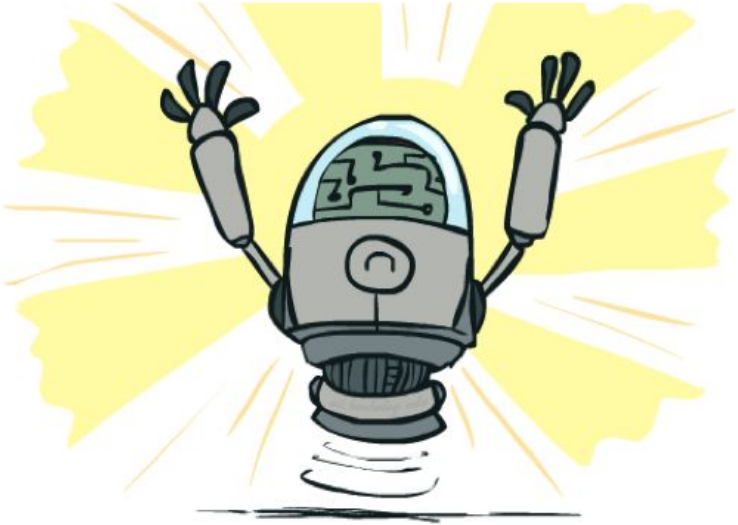
    h(A) ≤ actual cost from A to G
  - **Consistency**: heuristic "arc" cost ≤ actual cost for each **arc**

    h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases

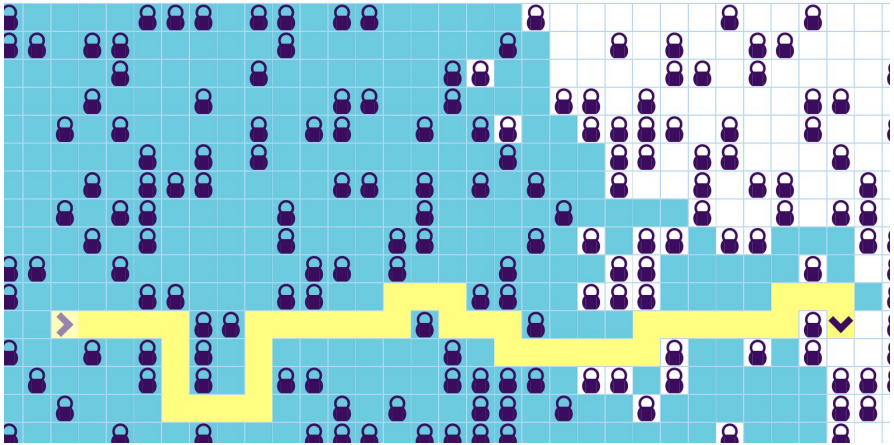
    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

Optimality

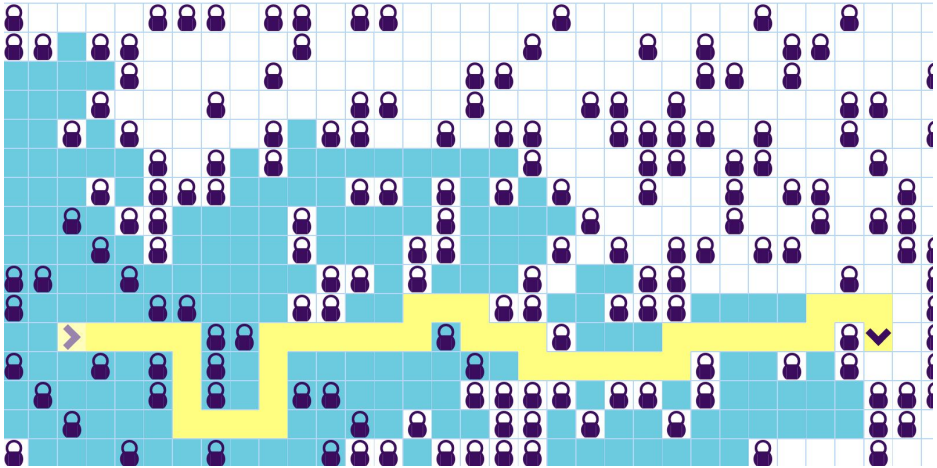
- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



Dijkstra's Algorithm is ***weighted*** and ***guarantees*** the shortest path!



A* Search is ***weighted*** and ***guarantees*** the shortest path!



https://clementmihailescu.github.io/Pathfinding-Visualizer/

Creating Admissible Heuristics

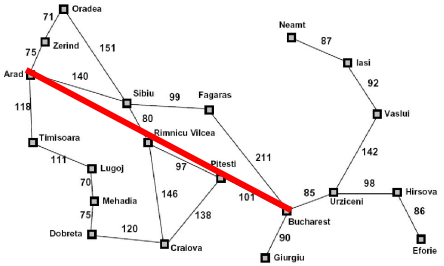
- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available

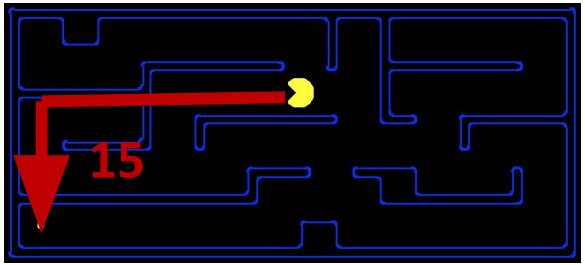
What are relaxed problems?
Are these problems with modified constraints?

- Inadmissible heuristics are often useful too

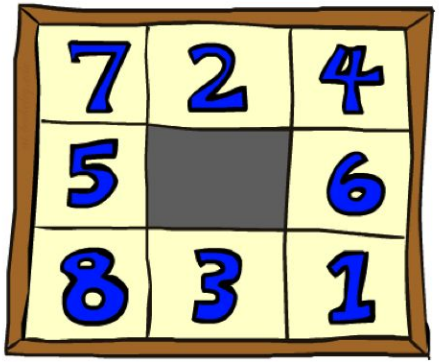
366

Take a plane instead of driving

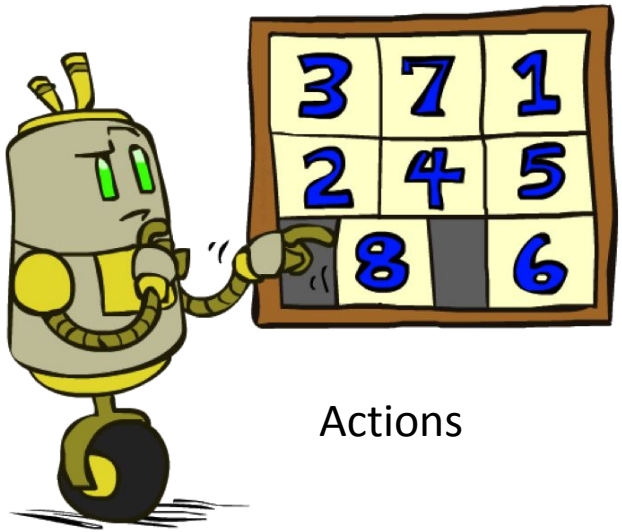




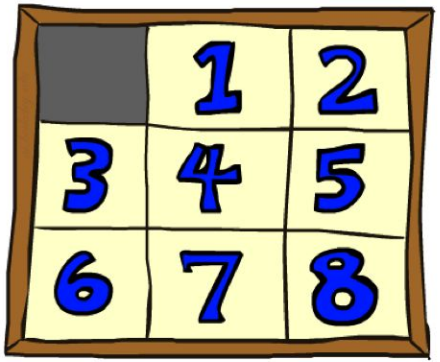
Example: 8 Puzzle



Start State



Actions



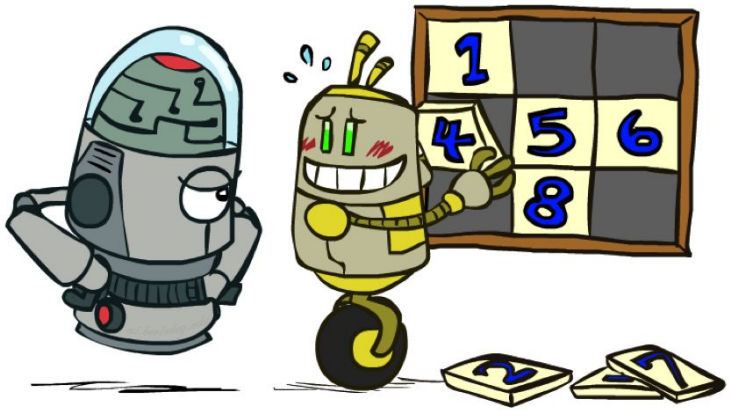
Goal State

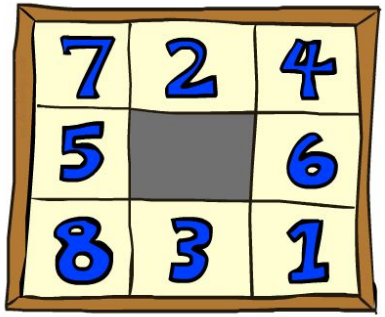
- What are the states?
- How many states?
- What are the actions?
- What should the costs be?

There are 9 * 8! = 9! many possible states

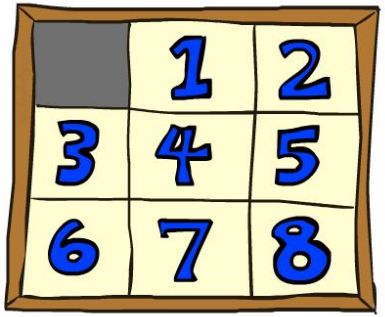
8 Puzzle - Heuristic 1 (TILES)

- Heuristic $h_1(S)$:
  = Number of misplaced tiles
  = 6





Start State



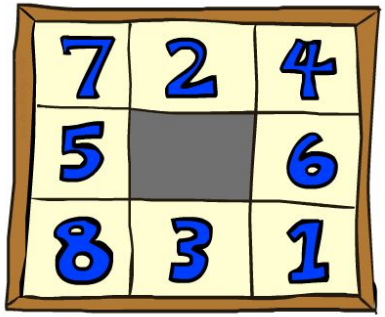
Goal State

	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

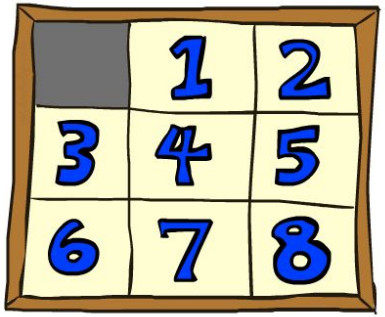
Statistics from Andrew Moore

8 Puzzle - Heuristic 2 (MANHATTAN)

- Heuristic MANHATTAN $h_2(S)$:
  = total *Manhattan* distance (i.e., no. of squares from desired location of each tile)



Start State



Goal State

This heuristic is better since it tells you how close you are to getting the tiles in their proper place. The other heuristic only mentions how many tiles are wrong, which does not say how close the current state is to the solution state.

$h_2(S) = 4+0+3+3+1+0+2+1 = 14$

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

Relaxed versions of the problem

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

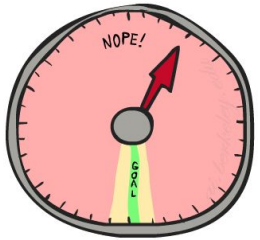
If the rules are relaxed so that a tile can move to any adjacent square,
then $h_2(n)$ gives the shortest solution

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

8 Puzzle - Actual Cost

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

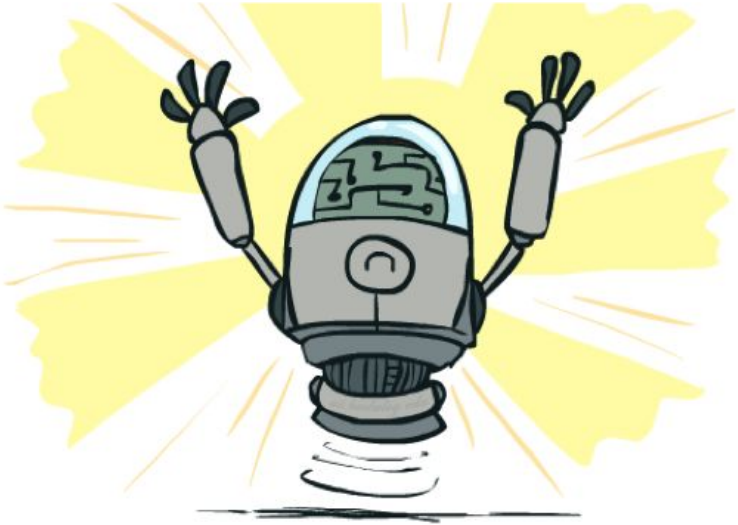
You do not save on the number of nodes expanded since the heuristic does not give you a value regarding the forward cost, not aiding in reaching the target state.



- 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

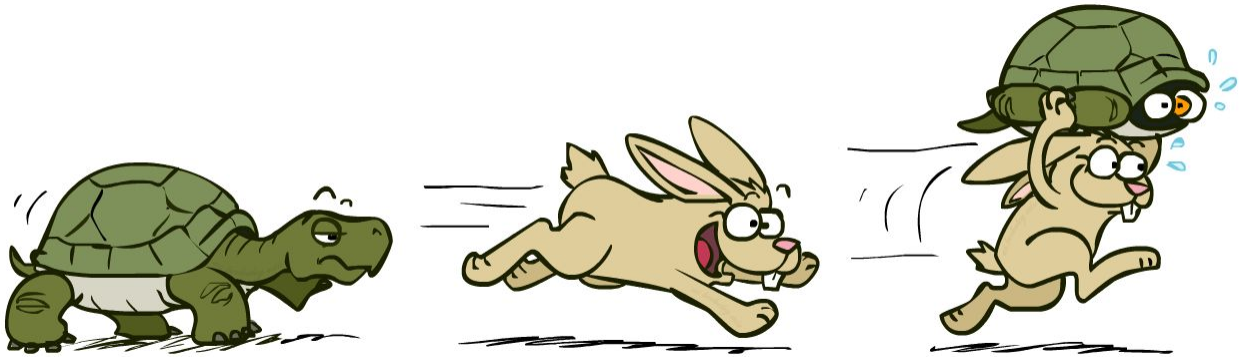
Optimality

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



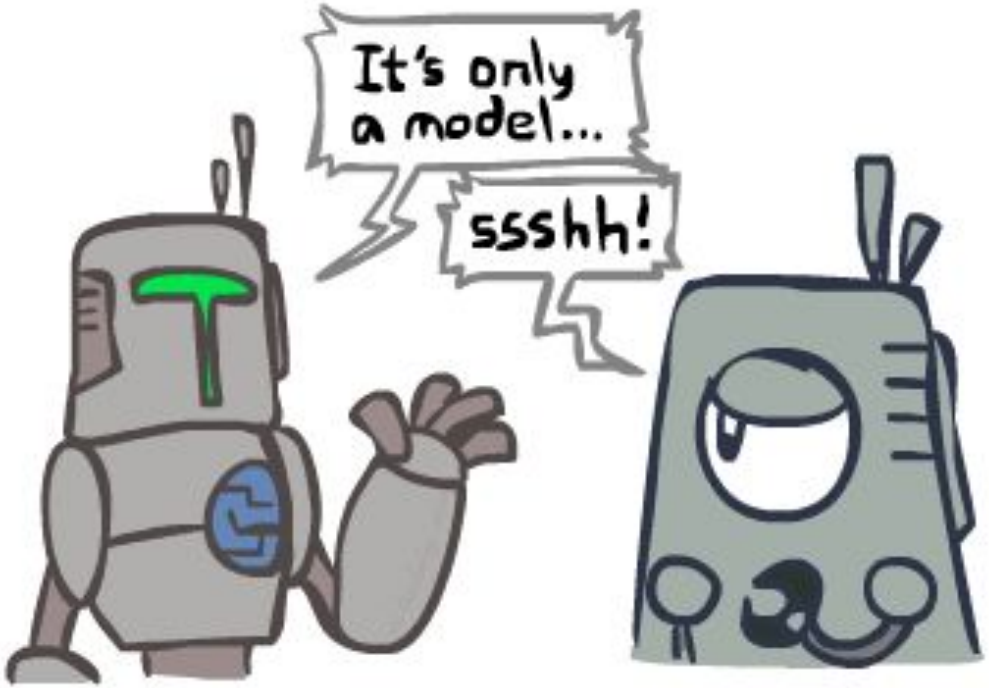
A*: Summary

- 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



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…



Summary

A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a set of **goal states**, and an **action cost function**.

Uninformed search methods have access only to the **problem definition**. Algorithms build a search tree in an attempt to find a solution.

Informed search methods have access to a **heuristic** function h(n) that estimates the cost of a solution from n.

Week 6 Check-In

Please leave your feedback here!
https://forms.gle/PUvvHbrG39p6zqp58

Appendix: Search Pseudo-Code

Tree Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```