CS 461
Introduction to AI

Basis and Concepts

What is AI?

- Does it involve thought? Behavior? To what standard?

- Acting *humanly*
  - Turing test: Can an AI system communicating via text interface mimic a human well enough to convince someone that it's human?
  - **General** knowledge, natural language processing, social *cues*

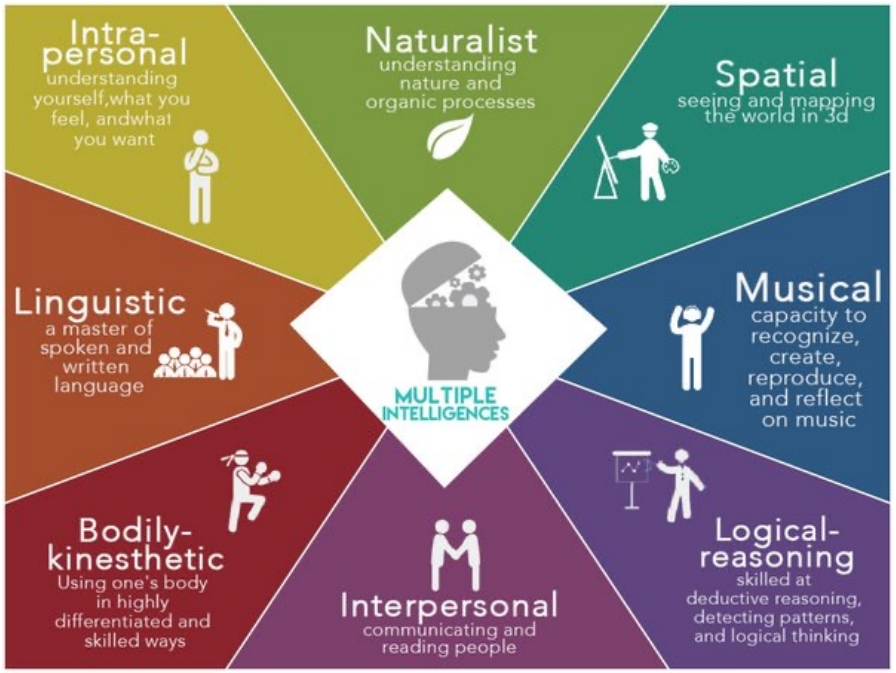
- Acting **Rationally**
  - Can a system **integrate** information about its **environment** and formulate a plan of action?
  - Sensing, reasoning in the presence of uncertain or incomplete information, identifying costs & tradeoffs

- Thinking *humanly*
  - Identifying "laws of thought" and human reasoning processes
  - Logic, sensing, learning via same **mechanisms** as humans use

- Thinking **rationally**
  - Logic, *perception*, drawing conclusions & new facts from known ones (learning)

What is AI?

- How do we determine if someone (or something) is intelligent?



Smart	VS	Intelligent
Definition		
Is a person who uses his intelligence practically and efficiently daily		Is something which a person is born with
Measurement		
Non-measurable		IQ Test (Intelligence Quotient)
Refers To		
Refers to intellect and the appearance		The intellect of a person
Nature		
Practical and has a good judgment		Not always practical

What is AI?

- Is it important that it use the same methods humans use?
  - What if humans can't really describe in depth how they reach a conclusion?
  - What if animal intelligence only emerges in groups?
    - Individual ants don't show signs of intelligence, but ant colonies demonstrate coordinated actions

- Working definition: *Artificial* Intelligence is the science of making machines **do** things that would require *intelligence* if done by humans

- Implications of more complex design?
  - "A year spent in Artificial Intelligence is enough to make one believe in God." –Alan Perlis, *Epigrams on Programming*

Identity

Leibniz's Law

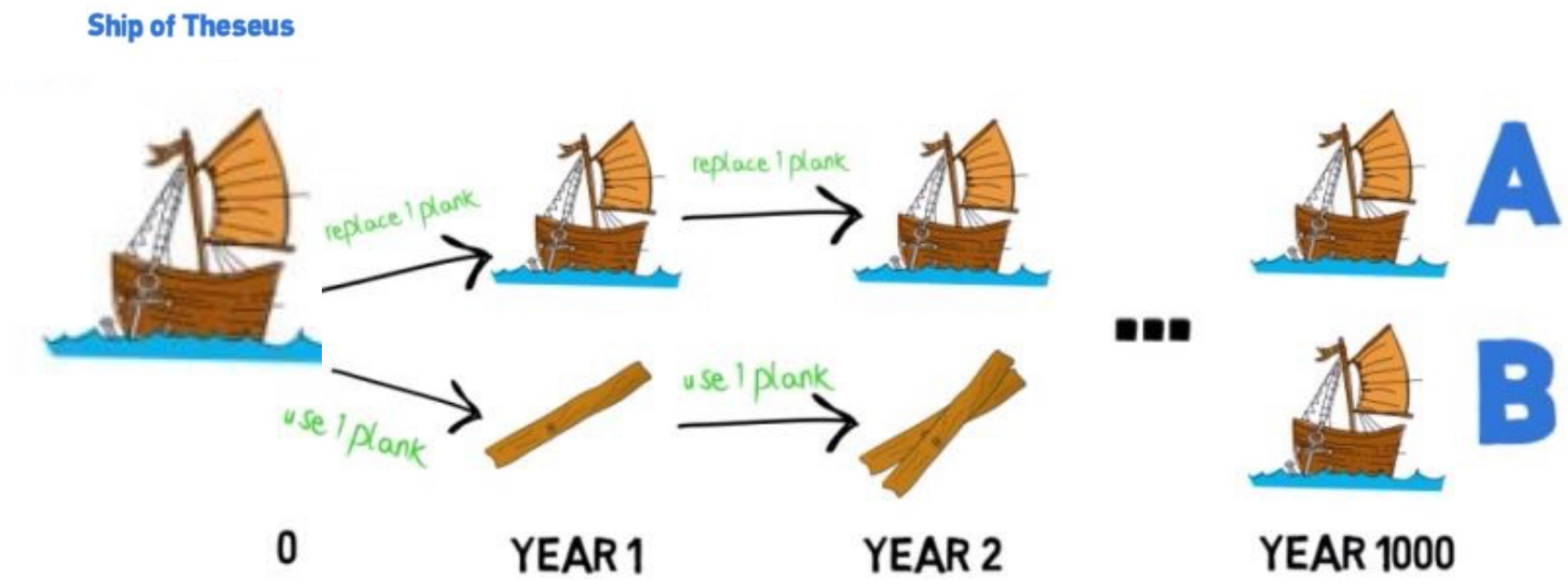
For any two objects X and Y: If X = Y, then X and Y share all the same properties.

As we've noted, this is logically equivalent to the following principle:

For any two objects X and Y: If X and Y do not share all the same properties, then X ≠ Y.

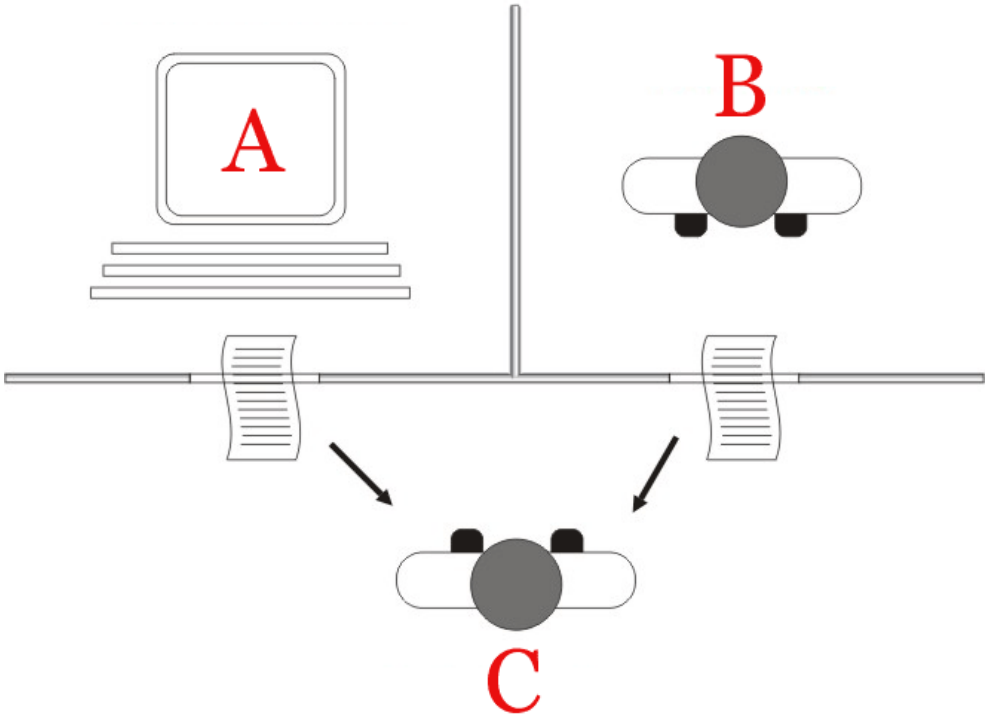
$$(a = b) \rightarrow \forall F\,(F(a) \leftrightarrow F(b))$$

Identity


Ship of Theseus
replace 1 plank
replace 1 plank
use 1 plank
use 1 plank
A
B
0
YEAR 1
YEAR 2
YEAR 1000

The Turing Test

- Turing proposed an imitation game.
- An interrogator interacts with 2 entities—1 human, 1 machine—via text interface
  - The human is assumed truthful; the machine can lie
- The **interrogator** must determine which is ***human*** and which is the ***machine***
  - So, for example, if asked for the square root of 23,921, the machine would not reply with the answer correct to 12 decimal places within microseconds, as humans don't do that; it might delay its answer, give an approximation, or make small errors
  - If asked to discuss today's weather (because the computer can't look out the window), the computer can perhaps look up the weather on the Web before responding
  - If asked about something emotional ("Tell me about your first romantic crush"), the machine would not have direct experience, but would know how humans have described these things; also, emotional depth isn't synonymous with intelligence



Objections to the Turing test

- The head in the sand objection: Intelligent machines would depose us, and thus are too horrible to think about
  - Not really a serious objection…
- Machines don't have **souls** and thus can't be "**really**" intelligent
  - We don't know what the limits are to what can and can't have souls; also, if the soul is by definition immaterial and immeasurable, how can we know for sure if it's there or not, i.e. how do we know that *we* have souls?
- A machine can't do something truly, surprisingly original
  - Machines have already surprised us with their capabilities.
  - This assumes that humans can immediately deduce all consequences of a given fact or action; this simply isn't true
- A machine will never, for example, make a human fall in love with it.
  - But we know how to make objects people come to love. Ever own a teddy bear?
  - Also, people have fallen in love with (and 'married') video game characters
- Or appreciate *beauty*…
  - That's your definition of intelligence? Aesthetic appreciation?

More basic objections to the Turing test

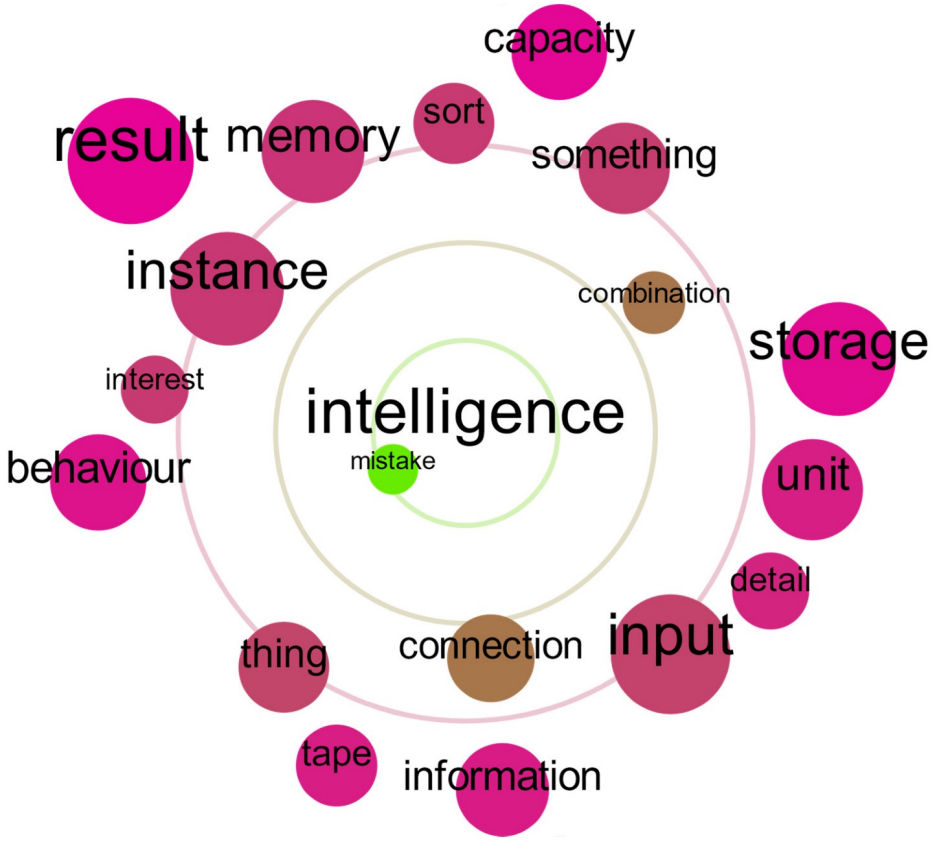
- Searle's Chinese Room: Users pass notes written in Chinese under a door. Inside, a human, who does not know Chinese, looks them up in a series of complex rulebooks and writes out some other symbols (which he also does not understand) and passes them back out under the door. People reading those responses on the other side believe he is answering in Chinese; yet neither the person nor the rulebooks can be said to understand Chinese.
  - Same setup, with 1000 users, each with their own rulebook, either what output to produce, or who to pass their note to. Again, where does the so-called "knowledge of Chinese" reside?
- These rely on the idea that we cannot deduce internal state purely from observing external behavior of a black box; but we know from some situations that we can
  - Consider a person who does know Chinese. The knowledge of Chinese can't be localized to a particular part of the brain or set of neurons.
    - There are parts of the brain that process language, but English and Chinese use the same parts of the brain

More basic objections to the Turing test

- **Block's** objection:
  Text is stored in binary.

Given a **large enough** database, it's possible to store a library of queries & plausible answers to mimic intelligence via table lookup.

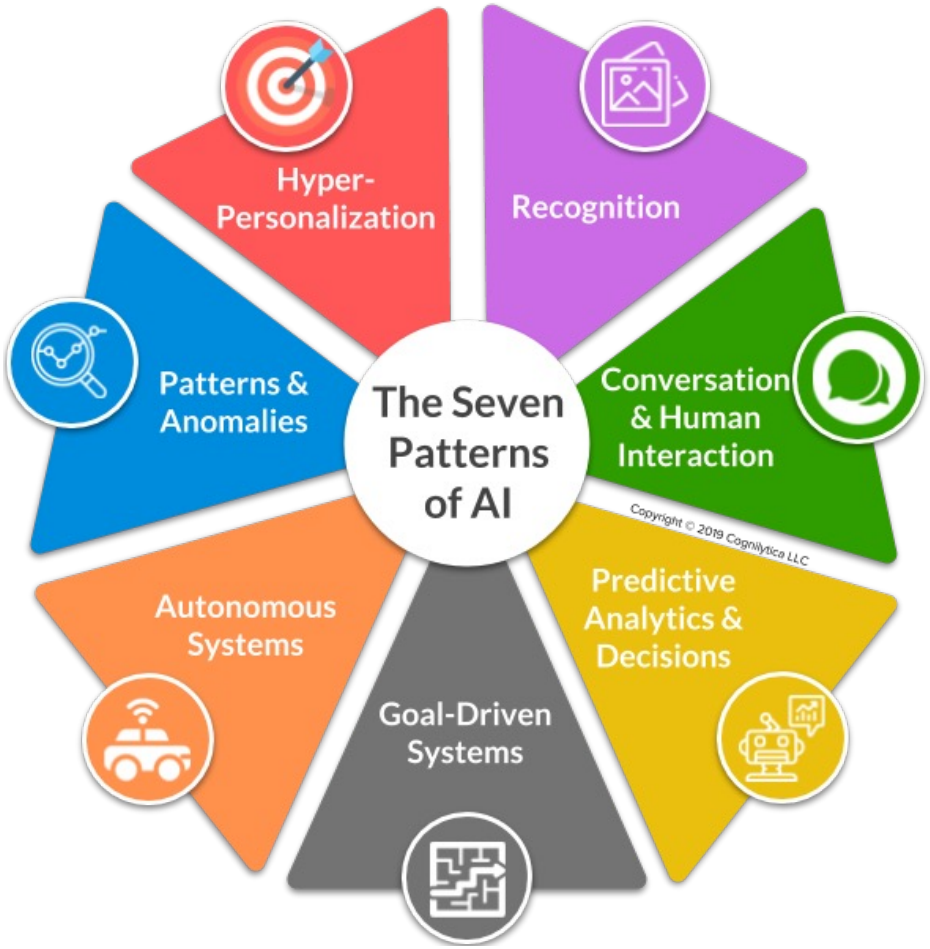
Do we ascribe intelligence to such a system?



Strong v. Weak AI

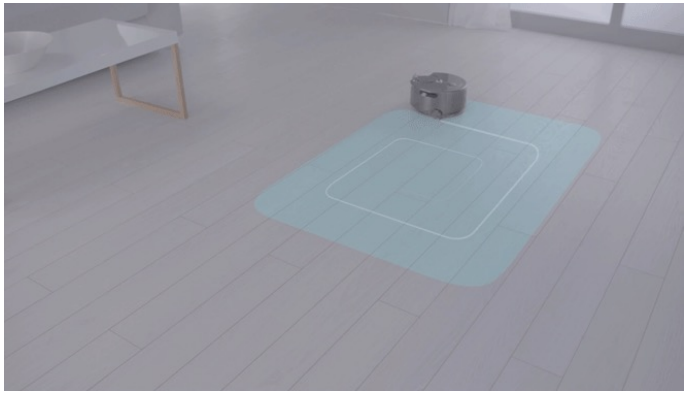
- One school of thought (dominated by MIT) argues that any system demonstrating intelligent behavior is AI, regardless of how it achieves its end. This is ***weak AI***.
- Another (dominated by Carnegie-Mellon) holds that a system should be based on the same methods of learning & cognition used by humans. This is ***strong AI***.
- Weak AI proponents argue that the purpose of AI is to solve difficult problems
  - if we obtain a solution, we are more concerned with the correctness & generalizability of that solution
  - If the process used to obtain it is different than a human would use, who cares?
- Strong AI proponents argue that the end goal is "true" intelligence, which would require self-awareness (consciousness)
- To date, there is **no** general-purpose strong AI in existence. All systems in existence today are weak AI, and often very limited
  - However, a system that can read MRI scans and diagnose cancers as well as an experienced radiologist, *and nothing else*, is still **very useful**

Common Implementations



Agents, Environments, and Perception

- **Agent**: Anything that can be viewed as perceiving its **environment** through **sensors** and acting on that environment through **actuators**.
- Percept: The agent's perceptual input at a given instant
- Percept sequence: Complete history of percepts
  - Choice of action can depend on percept or percept sequence, but not on anything it has not observed yet.
- Agent's behavior is described by an **agent function** that maps percepts or percept sequence to actions
  - Note that **agent function** does not base its actions on the environmental state, only on what has been perceived about that state
  - An agent function is implemented by an **agent program.**
  - Conceptually can be viewed as a table lookup or set of if-then rules; most implementations are more complex
- Vacuum cleaner world: 2 squares, A & B, each can be clean or dirty. Actions are GO LEFT, GO RIGHT, CLEAN
  - Can build up table: given this percept (or sequence), do this

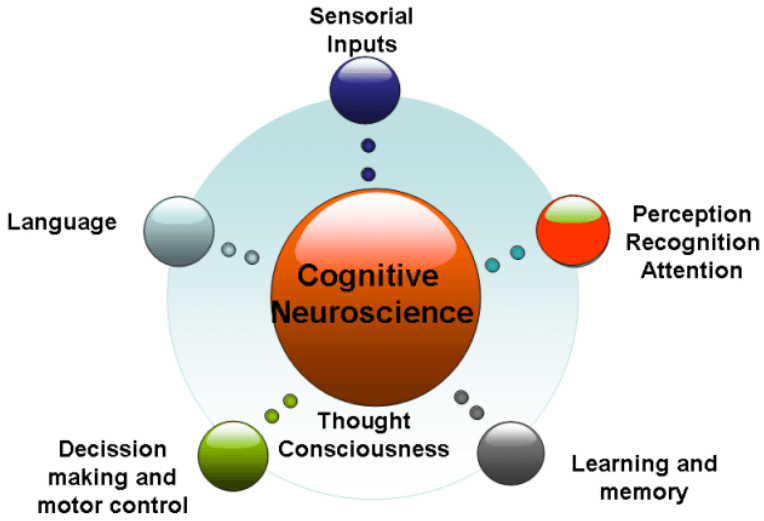


What makes an agent ‘rational’?

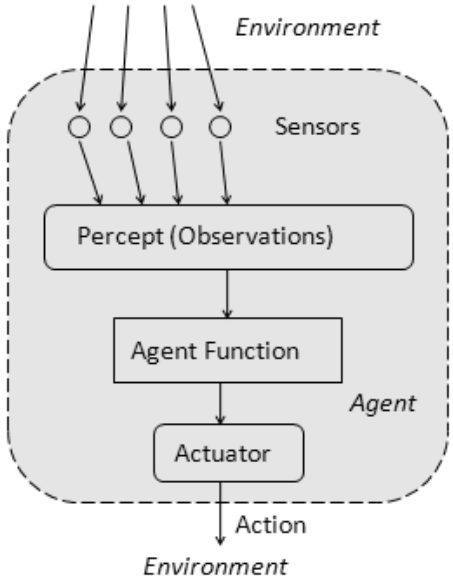
- One definition is that a rational agent does the right thing—with the proper “then” action for every “if” state
  - But what does “doing the right thing” mean?
- We consider consequences.
- The right action is the one that leads to the desired outcome based on the series of environmental states produced by interacting with the environment
  - Not agent states; otherwise our agent can delude itself by simply defining whatever it does (even if random) as the ‘correct’ thing to do
  - Compare the tendency among humans to tell themselves they don’t really want something that’s out of reach anyway (sour grapes)
- Agent function/program should be designed according to circumstances
  - For example, our vacuum cleaner should get credit for cleaning up dirt
  - But be careful! A vacuum could clean up some dirt, get credit, dump the dirt onto the floor, clean it up, get credit, dump it, clean it up, get credit…
  - Probably better to design it based on state of square (clean or dirty), perhaps with penalty for electricity use or noise, to discourage unnecessary work
- *In general, it’s better to define the goal state in terms of what is actually wanted, not on how the agent should behave.*
- Still issues to work out, based on how we reward cleanliness
  - Mediocre, kinda-sorta clean all the time?
  - Or immaculate as soon as it’s done, at the cost of long breaks (and therefore dirt buildup) between cleanings?

Foundations of AI

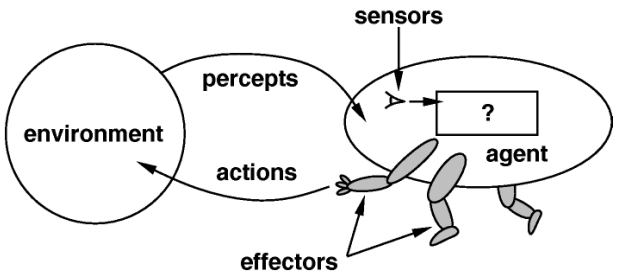
- Neuroscience
  - How do brains process information?
    - Brains are MUCH more energy efficient than any CPU
- Psychology
  - How do humans and animals think and act?
- Computer Engineering
  - How can we build an efficient computer?
- Control Theory & Cybernetics
  - How can artifacts operate under their own control?
    - Homeostasis, objective function
- Linguistics
  - How does language relate to thought?



Define “rational.”

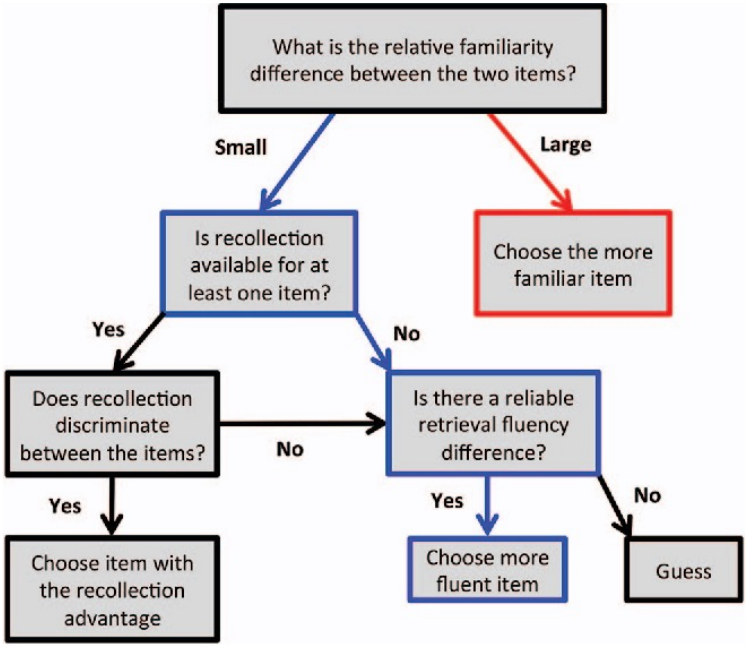


- Rationality depends on:
  - The performance **measure** that defines **success**
  - The agent’s **knowledge** of the **environment**
  - The **actions** the agent can perform
  - The agent’s **percept sequence**
- For each possible percept sequence, a rational agent **should** *select* an action that is expected to **maximize its performance measure**, given the evidence provided by the **percept sequence** and whatever built-in **knowledge** the agent has



Heuristics

- AI often relies on applying **heuristics** (which usually get a correct answer, at least approximately, and do it **quickly**) rather than algorithms (which always get a correct answer, exactly, **eventually**)
  - Solving a **simpler** but related problem
  - **Working backward** from a solution toward the starting state
  - Identifying similar solved problems & **testing** those solutions
  - Successive approximation (iteration)—if we can't find a solution, can we find a state that is in some way "closer"?
    - And if not, what do we do?



Problems suitable for AI

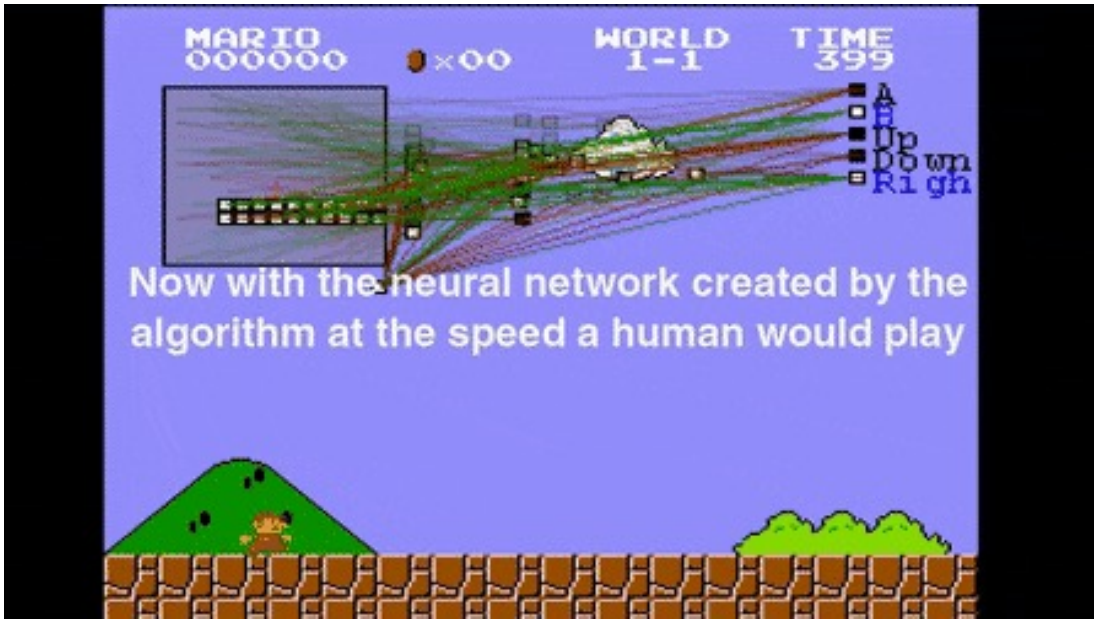
- Most AI problems are **large**
- Cannot be solved by straightforward algorithms
- Embody (encapsulate / integrate) a large amount of ***human expertise*** [representation]
- Examples:
  - Medical diagnosis (one of the first applications of ***expert systems***)
    - A lot of expertise, much of it in the form of if-then rules in a roughly hierarchical structure
    - Very complex, with complex interactions between rules & possible causes
  - Given a user's shopping history, what specials are likely to lure them into the store in the next week?
    - Given all of our shoppers' histories, what specials would bring the most total business in to the store?
    - Given this user's travel history, how much are they likely willing to pay for this airline ticket right now?
  - Given a user's spending history, what changes to their budget would we recommend? ("Do you really need to eat out quite so often?")
  - Given that chess has about 10^{42} "reasonable" games and about 10^{120} possible states, can we make a program that plays chess "well"?
    - How well?
    - With perfect play by both sides, who wins?
    - What is the best move in a particular position?
    - Note that this is still weak AI—a strong AI system could not only beat the very best humans, it could also explain the reasoning behind its moves and identify how to teach others to play better

Properties of Task Environments

- Fully v. Partially Observable
  - Can sensors detect all aspects of environment relevant to the choice of action?
    - May only be partially observable due to noise, distance, or inaccessibility (robot taxi can't tell what other drivers are thinking)

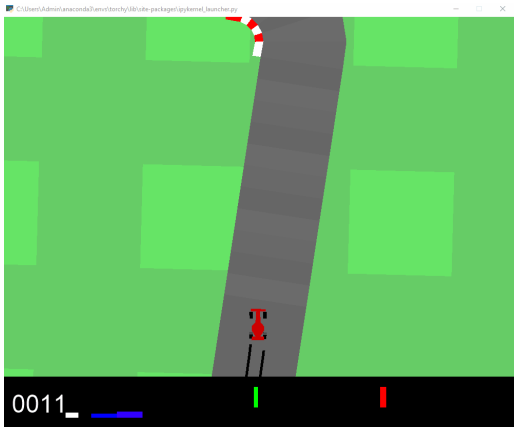
If there are no sensors at all the environment is **unobservable**

All is not lost, we can still behave rationally, and make certain deductions about our environment, without sensors



Properties of Task Environments

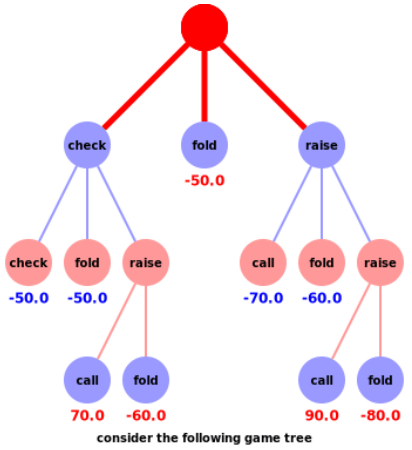
- Single Agent v. Multiagent
  - Is there more than one agent active in the environment?
    - Do we classify other drivers as agents, or semi-random features of the environment?
      - Is the other agent best described by assuming it's trying to maximize its own performance measure?
    - Are other agents **cooperative or competitive**?
      - Chess is competitive
      - Taxi-driving is partially both; vehicles cooperate to avoid collision, compete for limited parking spaces
    - Communication & cooperation emerge as rational strategies in cooperative environments; randomized behavior avoids predictability in competitive environments



Properties of Task Environments

- Deterministic v Stochastic
  - Is the environment's next (or immediate future) state completely determined by the current state (or state history) and the action executed by the agent?
    - Ignore the uncertainty from the actions of other agents; we deal with that elsewhere
  - Note that a fully deterministic but only partially observable environment might *seem* stochastic
    - Tires blow out; traffic can't be predicted exactly
  - Environments not fully observable or not deterministic are **uncertain**.
  - "Deterministic" implies we know something about the probabilities of possible outcomes. If we only know *possible* outcomes but not their probabilities, the situation is **nondeterministic.**
    - These agents usually have performance measures requiring success for *all possible* outcomes of its actions

Properties of Task Environments

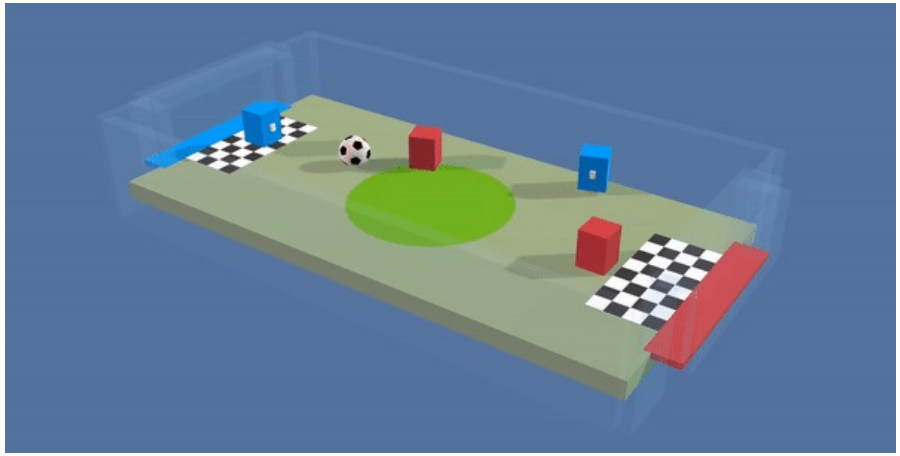


- Episodic v. Sequential
  - Does the agent need to remember the history of previous
  - states, or just attend to the current one?
  - Can this decision affect future ones?

a. Winning or losing this hand of poker doesn't affect the next hand

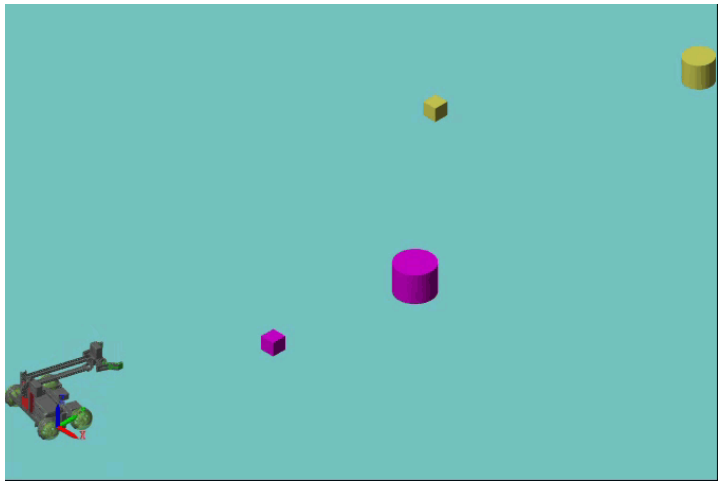
b. If the same position occurs 3 times in a game of chess, the game is a draw; thus our agent must remember the sequence of prior positions

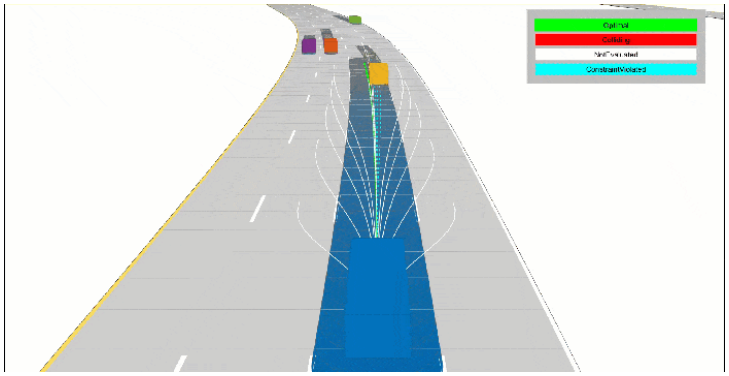
Episodic environments usually much easier to deal with



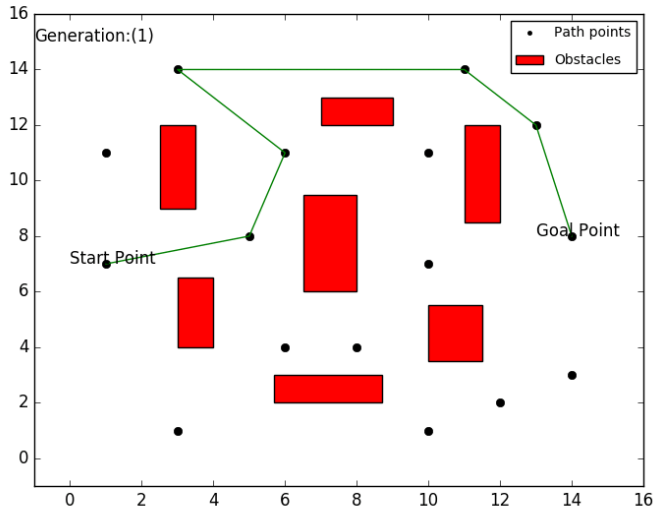
Properties of Task Environments

- Static v. Dynamic
  - Does the environment change while the agent is deliberating?
    - If yes, and an agent takes too long to decide, it counts as a decision to do nothing
  - If the environment doesn't change with time but the performance score does, the environment is **semidynamic**
  - Taxi driving is dynamic
  - Chess (played under tournament time-limit rules) is semidynamic
  - Sudoku is static
- Discrete v. Continuous
  - How are **state** and **time** handled? ***Discrete*** steps, or a **continuous** flow?
    - Some sensors discretize a continuous property; some input is technically discrete but treated as continuous (e.g. digital video)
- Known v. Unknown
  - The agent's (or designers') knowledge about the '**laws of physics**' related to a task.
    - In a known environment, the outcomes (or their probabilities) of all actions are known.
    - An unknown environment must be explored, or actions tried, to see what effect they have
  - A known environment can still be partially observable
    - A card-playing agent can't see other players' hands, but knows what game it's playing





Environments

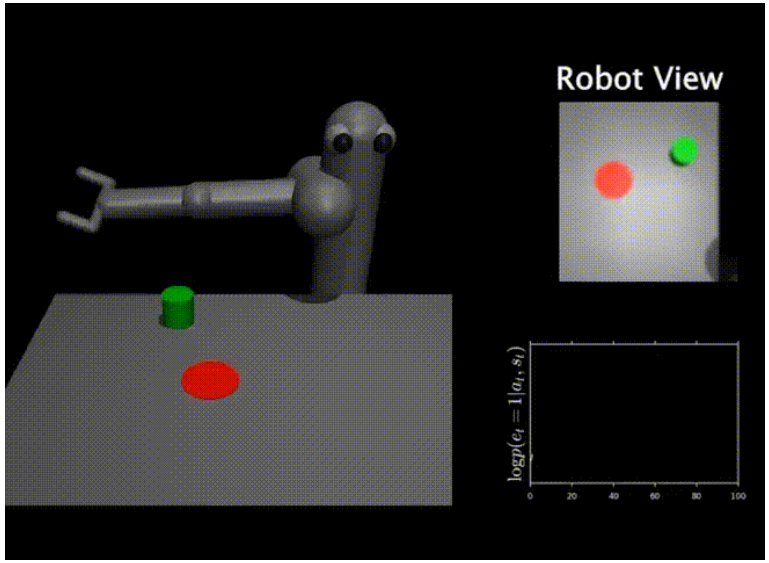


I. Most challenging cases:
   I. partially observable
   II. multiagent
   III. stochastic
   IV. sequential
   V. dynamic
   VI. Continuous
   VII. unknown
   - Driving a rental car in a new country with unfamiliar geography and traffic laws can be… interesting.

- Some environments can be classified multiple ways depending on how we define the boundaries of the problem
  - Is medical diagnosis single agent, or does the agent also have to deal with medical staff?
  - Is it single episode or sequential? (Can it propose a series of tests to refine the diagnosis? Review prior medical history from previous episodes?)

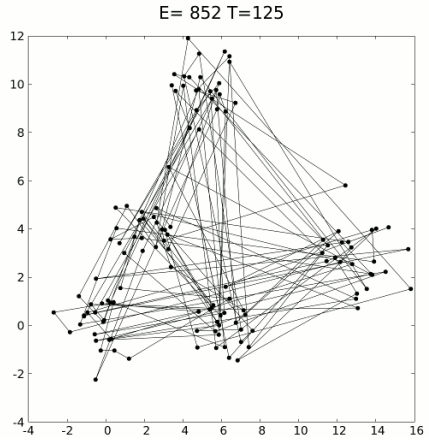
Foundations of AI

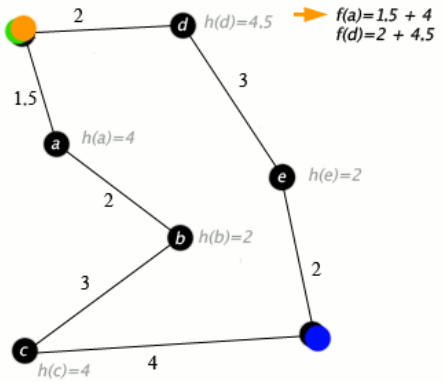
- Philosophy
  - Can formal rules be used to draw valid conclusions?
    - Rationalism
  - How does a mind arise from a physical brain?
    - Dualism (mind/body problem) v. Materialism (the body *is* the mind)
  - Where does knowledge come from?
  - How do we know what we know (epistemology)?
  - How does knowledge lead to action?
- Mathematics
  - What are the formal rules used to draw valid conclusions?
  - What can be computed?
    - Godel's Incompleteness Theorem
    - Halting problem
    - Tractability
  - How do we reason with uncertain information?
- Economics
  - How should we make decisions to maximize payoff?
    - Decision theory
  - How should we do this when others may not go along?
    - Game theory
  - How should we do this when the payoff may be far in the future?
    - Operations research, Markov processes



(Not covered in class) Looking Forward: Methods/techniques of AI

- Search algorithms
  - Unguided (blind)
    - Depth-first
    - Breadth-first
  - Guided (heuristic)
    - Hill-climbing
    - Beam search
    - Best first
    - Branch and bound
- Two-person games
  - Adversarial search
  - Iterated prisoner's dilemma
- Automated reasoning
  - Requires knowledge representation system, and inference engine
- Production rules & expert systems
- Cellular automata (complex behavior from simple rules, e.g. Conway's Game of Life)
- Neural Computation
- Genetic & Evolutionary computation
- Probabilistic & Fuzzy Reasoning





(Optional)Additional Readings

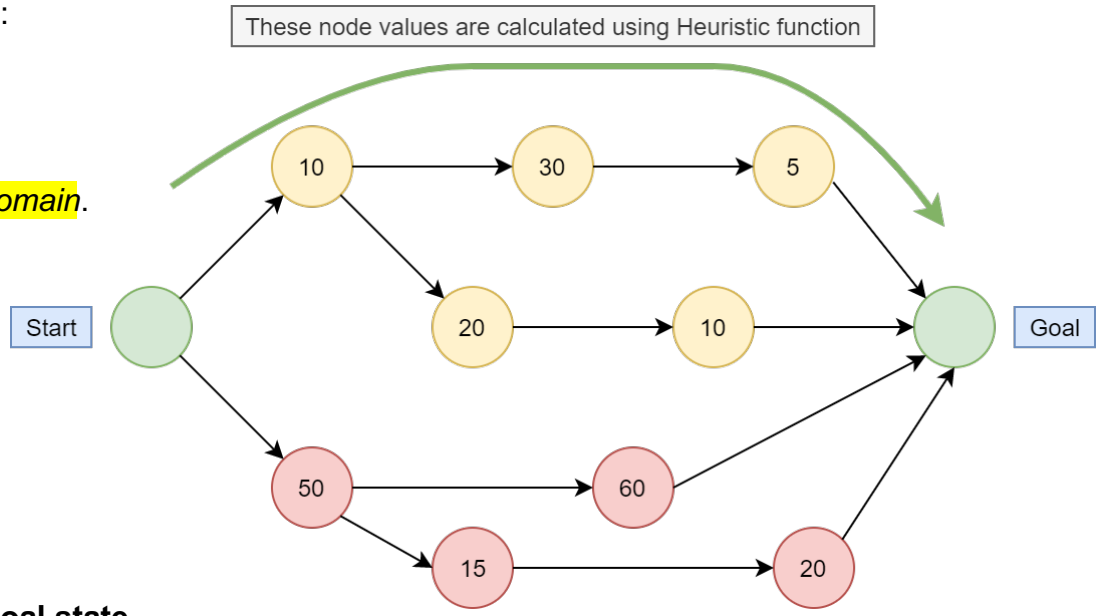
- Turing, *Computing Machinery and Intelligence*
  - http://phil415.pbworks.com/f/TuringComputing.pdf
    - Esp. Turing's treatment of various objections to AI, p. 443 & following
- Video - *Crash Course Philosophy: AI and Personhood*
  - https://www.youtube.com/watch?v=39EdqUbj92U&index=23&list=PL8dPuuaLjXtNgK6MZucdYldNkMybYIHKR

Search

•A lot of AI boils down to search—searching through:

- **states**
- **<u>assignments</u> of <u>values</u>**
- orderings of a **route**

•**Uninformed** or **unsupervised** search
does not require of *any knowledge of the problem domain*.
It is a **brute-force** approach.



•Terminology:
•We begin in some **starting state**, searching for a **goal state**.
•We have some method of **transitioning** from the ***current*** state to 1 or more ***successor*** states.
•Unsupervised search offers no method of selecting one transition over another for a given state.
There's no ***fixed*** metric to tell us if we're *getting closer* to the goal until we're there.
•One partial solution:
Use some measurement (derived from purely **local** information) to <u>make a selection</u> from offered successor states.
Search methods which use this solution are said to employ a **greedy algorithm**

Evaluating Search Methods

•We obviously need some way of determining whether we're at the **goal** state

•The **path length** between **two states** is the **number of transitions necessary** to move between them.

•**If** each transition has its own **independent** cost that may differ from 1, the path cost is the **sum of all transitions** on the path

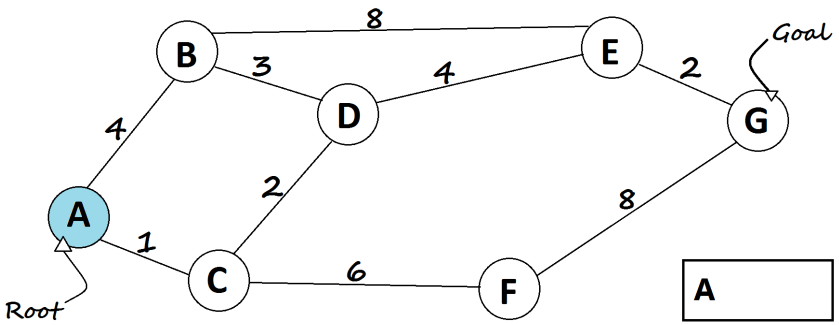
•Note that in this case the **lowest** cost path to the goal ***may not be the shortest*** (lowest number of transitions)

•Depending on the problem, state transitions **may** –or– **may not** be **reversible**

•We can produce a state-space graph showing the possible transitions.

•The maximum or average **number of transitions** existing out of a particular state is the ***branching factor***

•If we select only transitions that avoid cycles, a search tree results



False Coin Problem

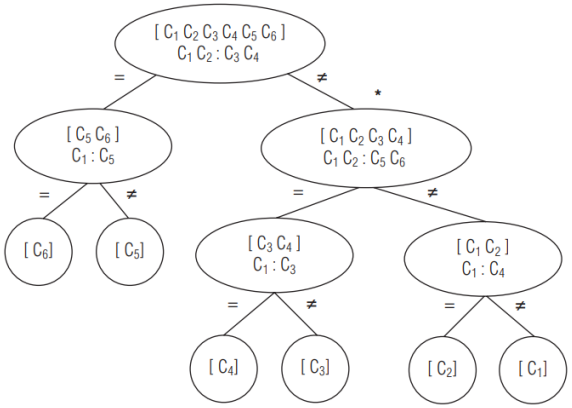
•Each state is a node (the list of which coin might be false, and what should be done at each), and the path chosen on the result of each comparison.

•As every possible outcome is accounted for, and each path terminates with exactly 1 solution, this is a map of the search space that must be dealt with.

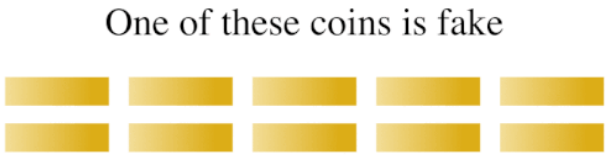
•The total state-space map of a problem contains every state the problem might be in, and all transitions between states; obviously it can become quite complex

•Sometimes we want to find out if any path from a start state exists to any solution; other times we want to find a shortest or lowest-cost or in some way optimal path

•Note that it may be convenient to represent a state space graph as having more than 1 node for a specific state; see fig. 2.2, p. 48



•A sample search tree showing a solution to the 6-coin False Coin problem is on p. 47.



•One method of finding a solution (in cases where we just want to know if any solutions exist) is to just generate all possible states, testing each to see if it's a goal.

Move Solver

•One example is the N-queens problem

•In chess, the queen can move horizontally along rows, vertically along files, or along either diagonal, and attacks every square it can move to.
•The N-queens problem is: On an NxN board, place N queens such that **no queen is attacking any other**.

•We could just generate every permutation of N queens on an NxN board and pick out the solution(s)
•For N = 10 (so a 10x10 board), there are 100 ways to place the first queen, 99 ways to place the second, etc., so total placements , or in general (n^2/n)
•We can reduce the size of this by observing, for example, that each row can contain only 1 queen, so begin by placing 1 queen in each
row and only move queens along rows
•Terminology:
•An algorithm is correct if it can find a valid solution.
•It is complete if it can find every solution; either every solution that exists, or every solution reachable from a given start state.
•It is optimal if it finds the best (lowest-cost, nearest, whatever) solution.
•It is optimally efficient if it finds the solution at least as fast as any other algorithm (in big-O form).
•It is nonredundant if any state rejected as a possible solution is not proposed again.
•It is informed if it is able to limit its proposals in some way rather than blindly generating every possible state (for example, every
possible placement of N queens on the board)

Strategies

•Exhaustive enumeration consists of generating all possibilities

•But this can lead to wasted effort

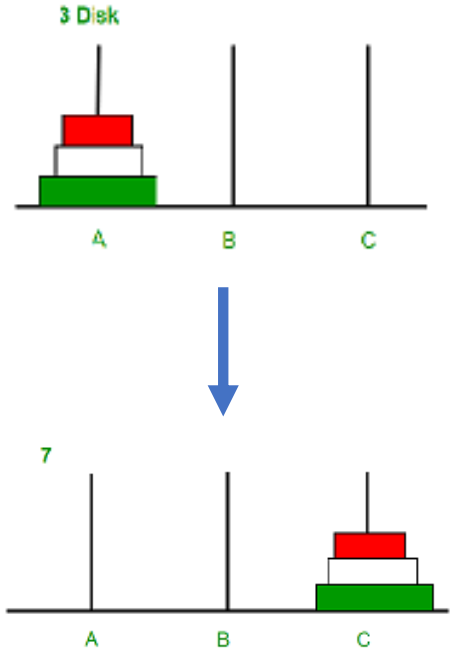
•If the first 2 queens we place are attacking each other, there's no point in placing the other N-2.

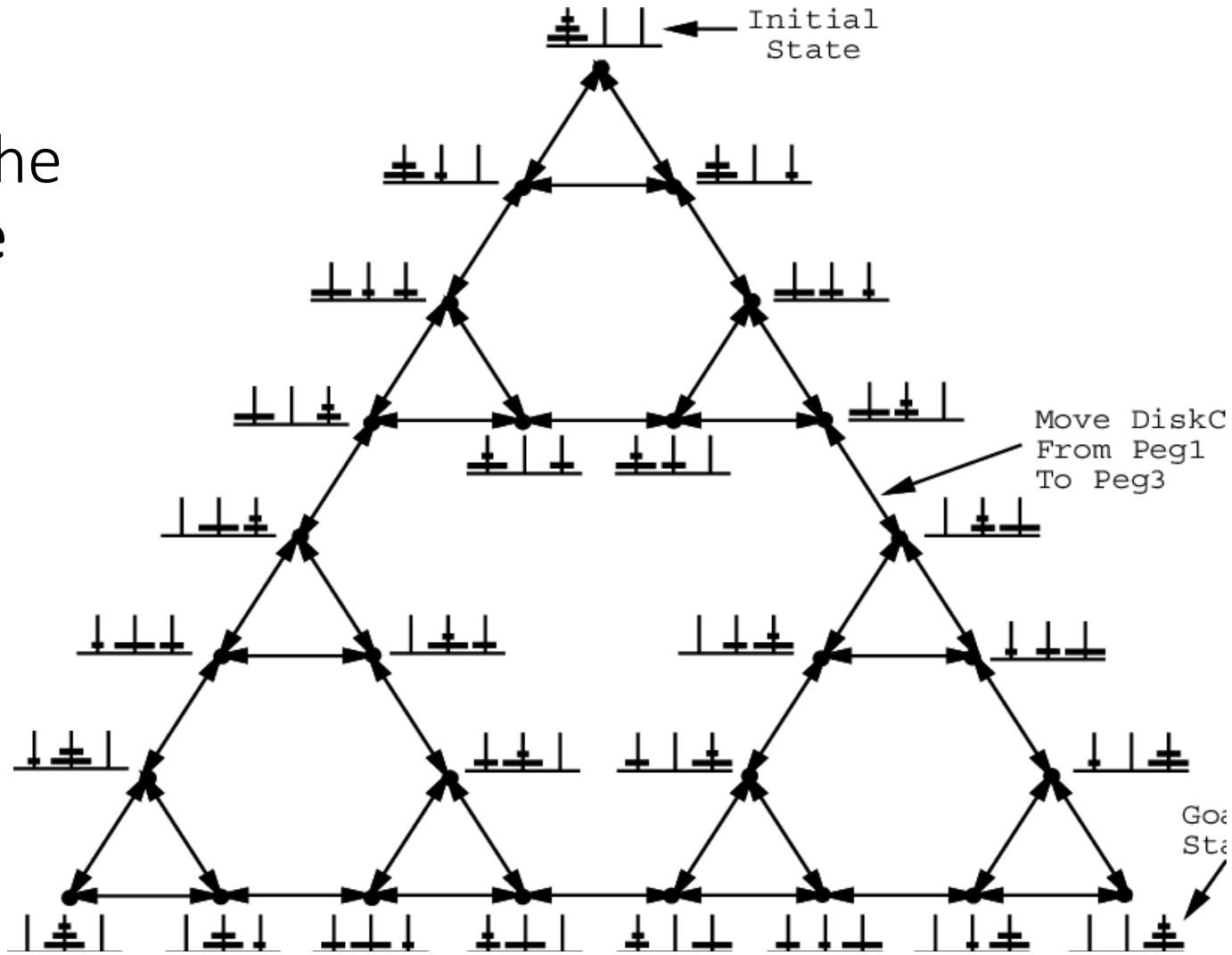
•While generating a state, we should verify that the partially-constructed solution satisfies all constraints

•If not, we try another; if no further progress can be made, we must backtrack, undoing part of what we have done so far, and making another attempt; if none can be found from there, we backtrack another step, and so on.

•Backtracking to solve 4-Queens, p 50-52

Unknown localization of the **Objective State**



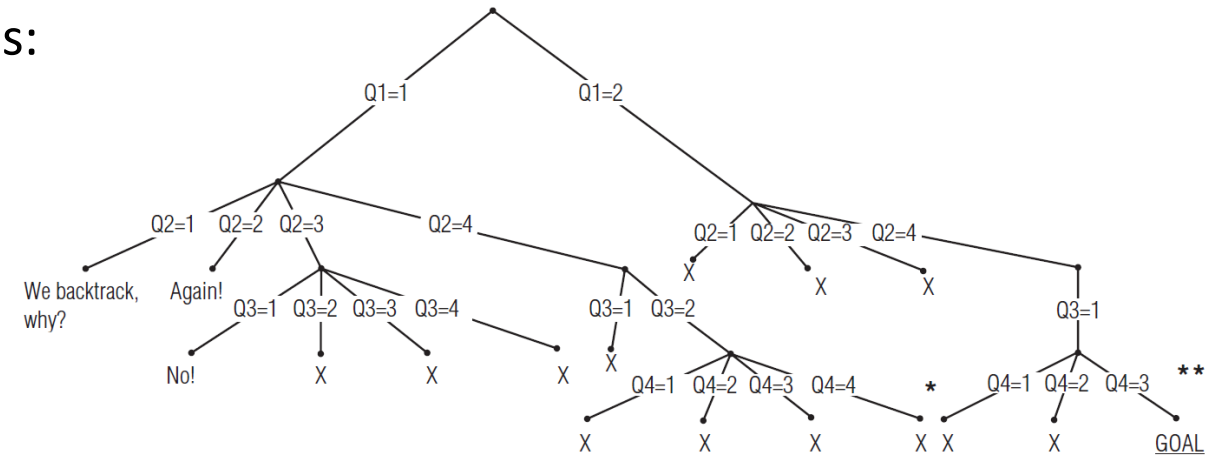


Brute Force algorithms

- Blind, or **uninformed**, algorithms, use *no domain knowledge* to select a path towards finding a solution
  - They are called "**brute force**" methods because they use:
    an exhaustive **undirected** ("blind") state search.

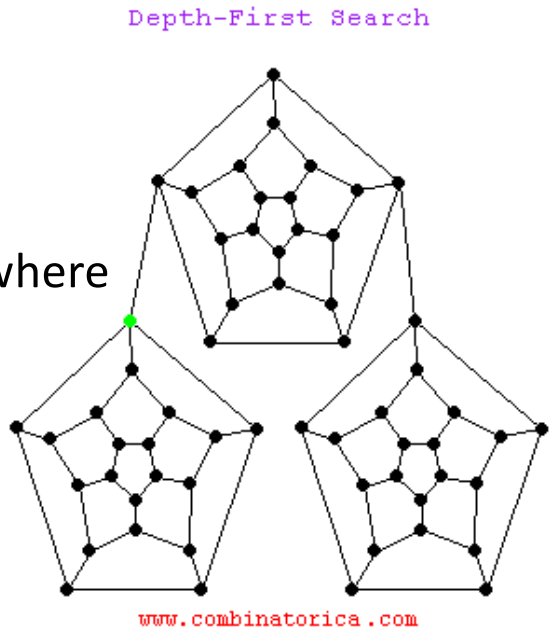
https://algorithm-visualizer.org/brute-force/binary-tree-traversal

- Generally, the **goal** is finding ***some*** solution
- The main 3 blind methods:
  - Depth first search (DFS)
  - Breadth first search (BFS)
  - Iterative deepening DFS
- Sometimes used:
  - Bidirectional search



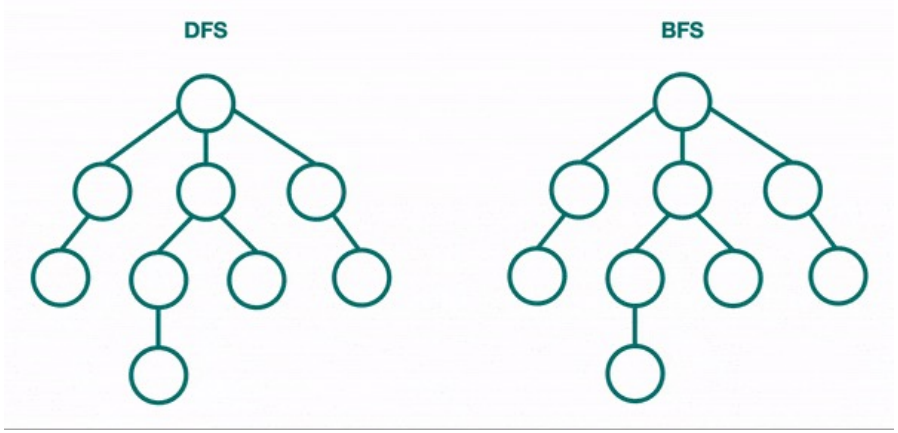
https://graphonline.ru/en/?graph=bigTree

Depth First Search



- DFS selects a branch and attempts to go as deep as possible.
- Given a choice, the leftmost branch is usually selected
- If a dead end is reached, we **back up** until we reach a point where another choice is possible, and try that
  - Thus, we will eventually try **every possibility**
  - In general, we can only determine that **no solution** exists by **exhaustively trying every state**
  - This can matter because, for example, for the 15-puzzle, the goal state is only reachable from half the possible states.
    Thus if **we begin in an arbitrary start state**, we may have to check:
    16!/2
    states before concluding the goal is unreachable
- If each state has a branching factor of **b** and we search to depth **d**, then total storage needs are **O**(**b*d**).
- If we generate states only when we need them (that is, take the first-generated branch; don't generate any others unless search down that branch fails) then **storage** needs are O(**d**)

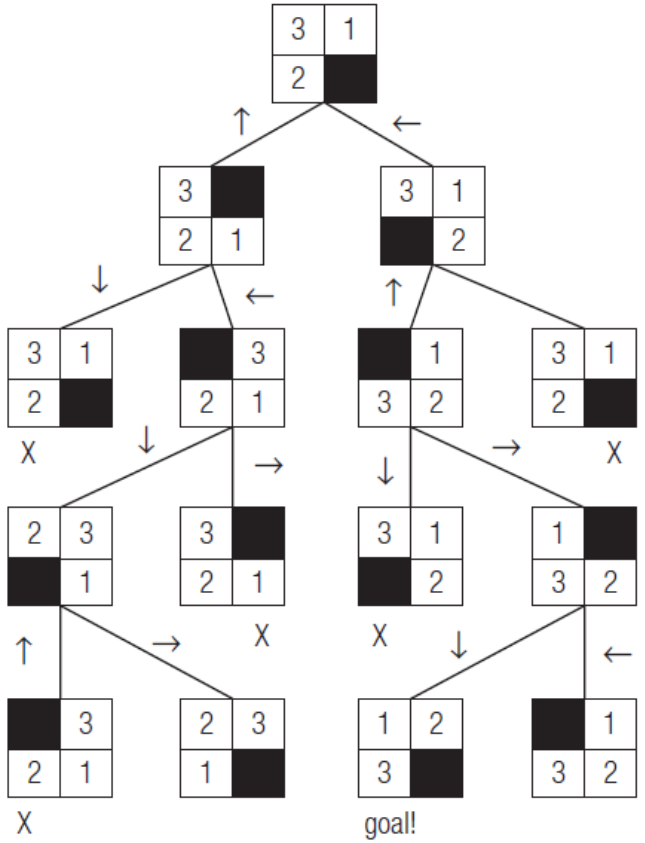
Breadth First Search



- In BFS, nodes are visited in level order; all nodes at level J are visited before any nodes at level J+1
- Since the successors of each node are usually generated & enqueued when the node is explored, storage for branching factor b to depth d is O($\mathbf{b}^{\mathbf{d[+1]}}$).
  - This is the primary drawback of BFS
- Since nodes are explored in level order, if a solution is found at depth J, we know there is no solution at a shallower depth.
  - May be **other solutions** at this depth that we **haven't explored yet**, of course

Breadth First Search (BFS)

- Walkthrough of solving 3-puzzle via DFS, p. 58
  - Note that we may want to have some way of recording which states we have checked to avoid redundant checks. This is an issue if we can reach the same state via more than one path.



Example of a solver for the 15-puzzle, (TB p. 57-58)

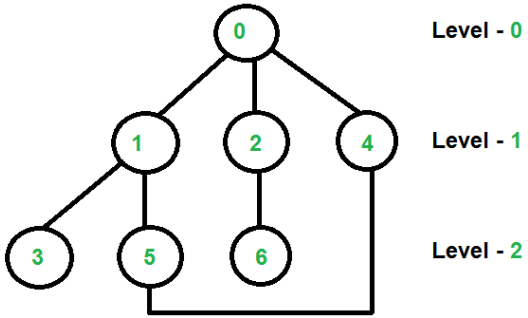
Implementation Issues

- Pseudocode, p. 61
  - DFS uses a stack to store states
  - BFS uses a queue
- Completeness: Since both exhaustively search all options, they are both complete in finite search space with finite branching factor b
  - If the search space is finite and any solution exists, it will be found
  - If the algorithm continues to search and the search space is finite, it will eventually find all solutions
- If the search space is infinite, BFS is complete (assuming its storage needs can be met); DFS may proceed down some path infinitely far and never find the solution if the solution is on a different branch
- Optimality
  - BFS is optimal; it will find the shortest-path solution. Since it searches in order of length, we know that no shorter path to a solution exists.
  - If costs are variable, the *uniform-cost search* generates and searches nodes in order of lowest cost, and will find the lowest cost solution
  - DFS is NOT optimal; it may find a solution very deep on the left branch before finding a solution that is shallow but on the right branch
- We implicitly assumed each node has the same number of successors—the branching factor. If the number of successors (children) is variable, the *effective branching factor* is the number of branches necessary to produce a tree of the same average depth

DFS with Iterative Deepening (ID-DFS)

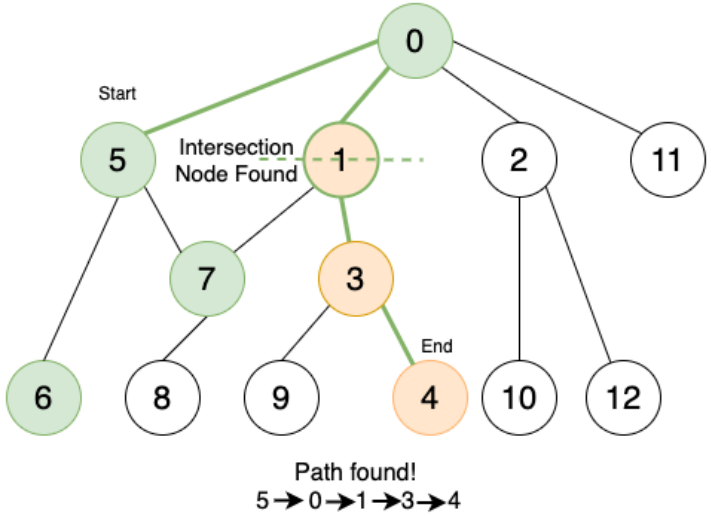
- The idea is to get a similar effect to BFS without paying BFS's exponential storage space.
- Do an exhaustive DFS to depth
- 1. Then do exhaustive DFS to depth
- 2. Then do exhaustive DFS to depth
- 3. And so on. Continue increasing search depth by 1 each round that a solution is not found.
- Note that we generate child nodes at upper levels of search tree many times
- Space complexity is O(bd), time complexity is still exponential in worst case
- DFS-ID is complete if the branching factor is finite and optimal when path costs are a nondecreasing function of node depth

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Bidirectional search



- Often we know what the goal state looks like, we just don't know if we can get to it from a given start state
- We can sometimes make "un-moves" to work backwards from the goal
  - "What positions/states/whatever at time N-1 can possibly lead to this state at time N?"
- So we do two searches at once, working forward from the start state and backwards from the goal state; we hope to find some state that both searches share in common
- This is sometimes used to do random exploration of very large search spaces
- It is not complete or optimal; there is no guarantee the 2 searches will ever overlap or that any solution found will be low-cost
- But if they do, it will be at an average depth **d/2** for each search separately, and **2*b$^{d/2}$** is much less than **b^{d}**.

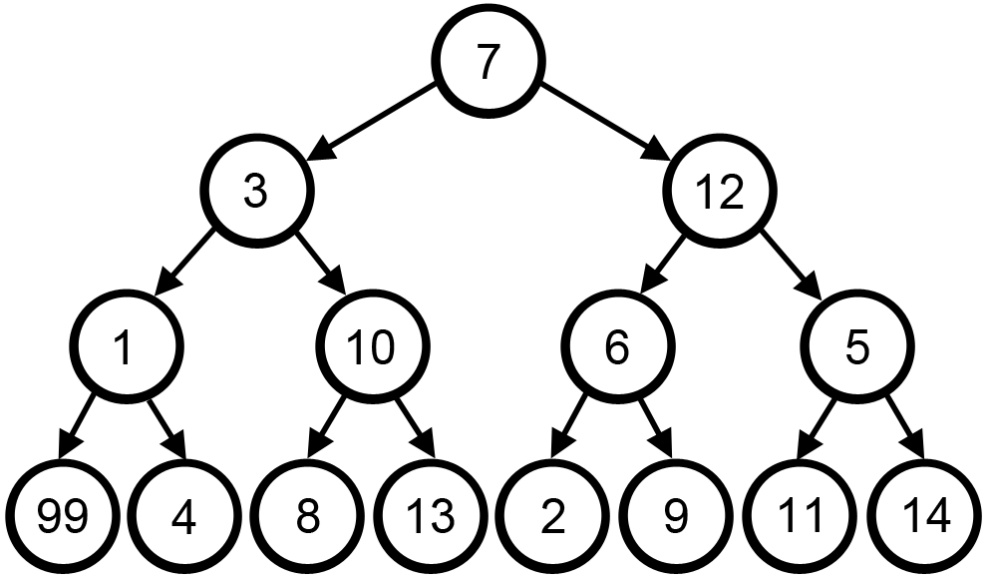
(back to) Greedy Algorithms

•A greedy algorithm always has some objective function that is to be optimized—usually minimized (cost, distance, time, etc)

•A greedy algorithm selects the path that looks 'best' based on **local information**.

•Suppose we are planning a trip involving visiting 4 different cities

•Start with a list of all possible destinations from the home city. Select the nearest one.



•To the list of **possible** destinations, add any city accessible from the first destination not directly accessible from home.

•Select the shortest available from that list.
•Etc... We always select the **lowest-cost** choice we have that is valid

•This is **Dijkstra's Algorithm**

https://graphonline.ru/en/?graph=weightedGraph

Greedy Algorithms continued

Depending on the problem, greedy algorithms can sometimes fail

•For example, consider the traveling salesman problem: Given a list of cities, find the shortest route taking the salesman to each city exactly once and returning home (minimum-length Hamilton cycle) https://tspvis.com/

•Simply selecting the nearest city to a city currently on the route can fail; example, p.55-6.

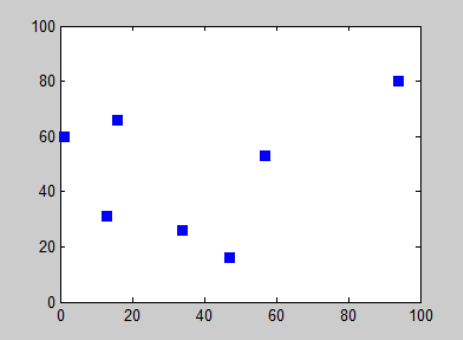
•An **exact** solution requires *exponential time* (the problem is **NP-Complete**) and is thus intractable for large problems

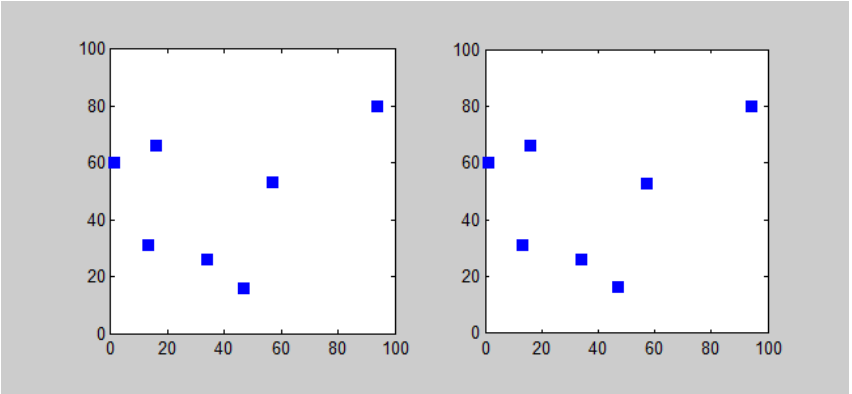
•Unfortunately, **several real-world larger-scale problems** map to (can be directly related) this problem.

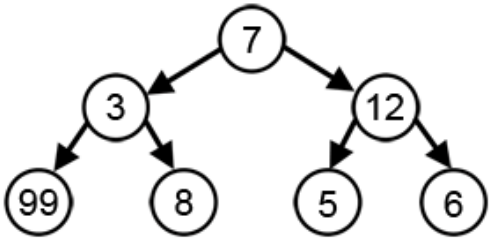
•For example, given 30,000 packages to be delivered in KC (what UPS does in a day) and 200 trucks, assign packages to trucks and find routes so that fuel costs are minimized

•And this is still a simplified problem, since it ignores package sizes and how many packages we can fit on each truck (the knapsack problem, also NP-complete for optimal solution)

•While there is no **polynomial-time** algorithm for finding the absolute shortest route, the technique of **simulated annealing** (covered later) can find a "pretty good" solution in practical time







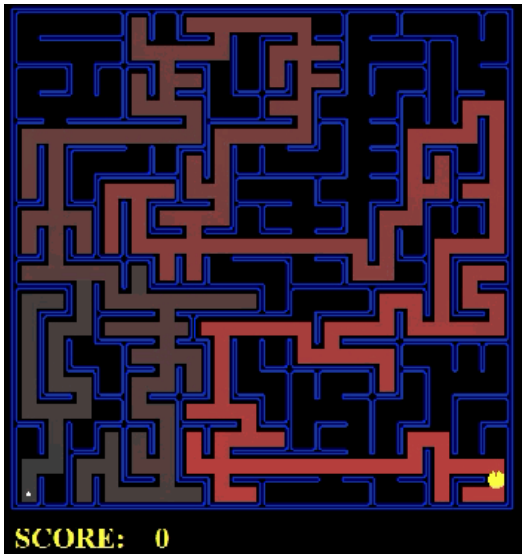
Uninformed Search Won't Do

- DFS can follow very *long searches* down one path and **overlook** much closer goal states down other branches
- BFS has **exponential** time and **space** requirements
- ID-DFS has more *modest* storage needs but still has **exponential** time requirements.
  - Search for route from KC to Los Angeles
    - DFS happens to pick I-70 Eastbound as first highway to try and takes weeks…
    - BFS runs out of memory quickly… (all 1-mile journeys, then all 2-mile journeys, etc…)
    - DFS-ID still has to check the same states as DFS so still takes weeks (solution is deep in the search tree)
  - Huge search space makes blind search unfeasible for most games, where time constraints are an issue

Uninformed Search Won't Do

- Guided search uses ***heuristics*** to guide the search
  - **Hill climbing** as one possibility
  - **Beam search** and best-first searches are also applied
  - These algorithms ***typically*** don't back up; they go forward until either a **goal** is found or a **dead end** is hit
  - Example heuristic:
    - List cities '**adjacent**' to KC on our map.
    - Select the one with the **smallest straight-line distance** to Los Angeles as first intermediate step.
    - From there, select '**adjacent**' cities and repeat this process.
  - An ***algorithm*** always finds the optimum, 'best,' etc. solution, in finite time.
  - A heuristic finds a good solution (or at least a candidate solution) and does it quickly. Heuristics may be incorrect.
    - One heuristic used in chess programs is to look at **capturing** moves first. But sometimes a capture is the **worst** move on the board.
    - The '**closest**' town may be a dead-end in the mountains; we should have cut south 50 miles to take a pass thru the mountains

Properties of Heuristics

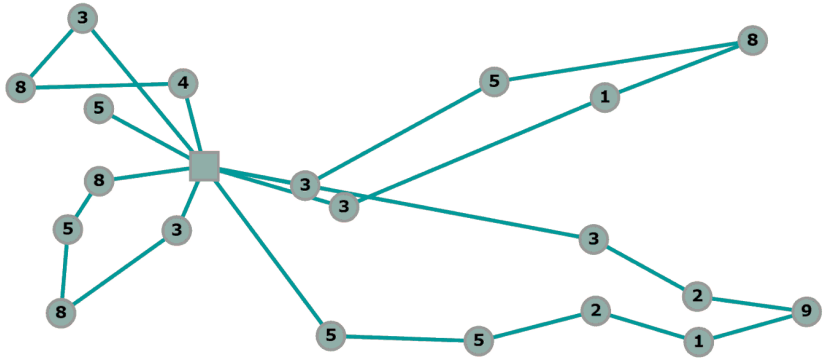


- We want heuristics to have certain properties:
  - They should **underestimate** the actual cost to the goal. (This ensures that we don't overshoot the actual goal. We **don't** want to **reject** an option as "too expensive" when the problem is that our heuristic is wrong.)
  - They should be ***monotonic***; that is, as we progress, the estimate should decrease. (This may not be true of our route-finder; for example, road construction may require a detour)
  - A search algorithm that lets us search a **smaller portion** of the tree before finding a goal is said to be more informed.
  - Obviously, the **less** we have to search, the **faster** we can complete that search.
  - Some search **algorithms** only examine a single path; they can get stuck on a **local** optimum. Other search methods are tentative in that they allow exploring alternate paths.

Properties of Heuristics

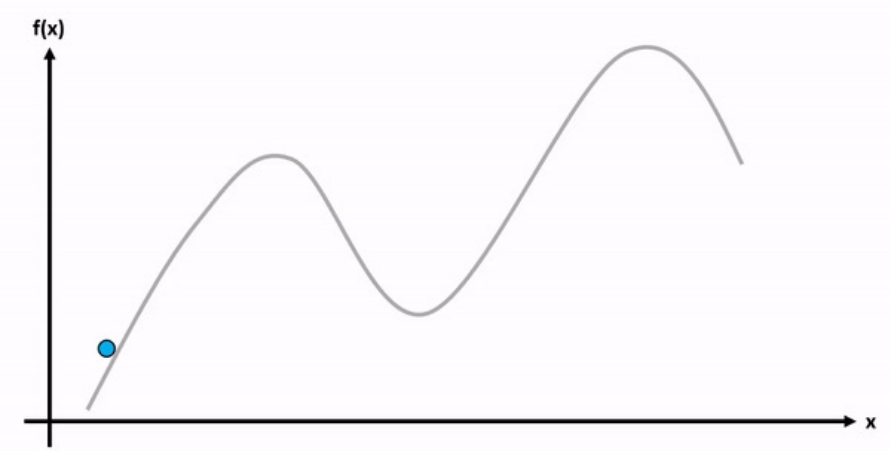
- Hill climbing, beam search, and best-first typically don't look back.
- Other methods include the distance/cost from the **root** as part of their calculations.
- B*ranch and bound* methods always look backwards to monitor progress so far.
  - Adding an estimated cost to reach the goal gives the well-known A* algorithm.
- Other methods include **constraint satisfaction** (e.g. applying no-conflicts rule of N-queens to reduce space that must actually be searched) and bidirectional search
- Solving a problem generally involves solving **subproblems**
  - In some cases, we must solve **ALL** *subproblems*; in others, only a **subset**
  - Laundry requires **washing** AND **drying**. But drying can involve using a **dryer**, OR a **clothesline**.
  - The data structure used to deal with these kind of problems is called, oddly enough an AND/OR tree.

Heuristics



- The goal of any heuristic is to reduce as much as possible the **amount** of the search tree we have to actually search
- Thus these are well-suited to search problems showing **high combinatorial complexity**.
- As an adjective, heuristic describes learning without necessarily having a guiding theory; or by applying rules based on general knowledge learned from experience
- As a noun, a heuristic is a specific technique to simplify or reduce a problem, speeding up solutions by identifying the most probable path and eliminating less probable ones, hopefully avoiding dead ends
- In search, heuristics can be used to select:
  - Which node should be examined next, rather than blind BFS or DFS
  - Which nodes (children) should be generated next rather than all successors at once
  - Whether certain nodes should be discarded (pruned) from the search tree
- Because they are not completely reliable, they increase the uncertainty of a solution. But in situations where algorithms give unsatisfactory results or don't guarantee any result, they can be quite helpful, particularly in very complex problems (image or speech recognition, robotics, game strategy, etc).
  - UPS would like to know what packages go on which trucks to minimize fuel costs, but can't wait 10,000 years for the exact solution to be computed; they need a 'good enough' solution by 4 AM tomorrow, when it's time to start loading the trucks.
- Sample heuristics:
  - In chess, look at capturing moves first
  - In route finding, choose the next step that comes closest to a straight line towards the goal
  - Avoid driving across town between 7:00 and 9:00 am and between 4:00 and 6:00 pm if possible
    - For the Grandview Triangle, between 4:00 and 7:00 pm…

Hill Climbing



- The idea is that if we are trying to reach the **top** of the mountain, always select the path going **up**; if more than one choice, select the **steepest ascent**.
- Note that *we do not add information* to our search or look backwards; we only look at which direction appears to lead up from where we are right now. Past nodes/steps are forgotten.
- This can lead to dead ends.

Hill Climbing

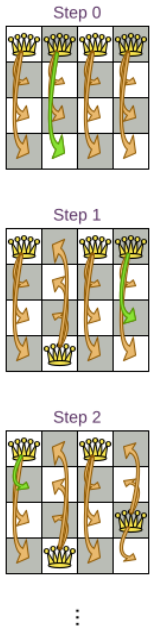
- A hill-climbing approach to N-queens would be:
  1. Begin by placing 1 queen in each row, selecting columns with uniform probability
  2. If the number of conflicts (queens attacking each other) == 0, we have a solution and we're done.
  3. Otherwise, check for each row:
     1. Check for each column:
        1. If the queen in this row were in this column and all others stayed where they are, how many conflicts would result?
  4. If the move with the minimum number of conflicts has fewer conflicts than the current position, make that move and return to step 2.
     1. If more than 1 move is tied with the same minimum number, choose between them with uniform probability
  - Note that there is no guarantee we'll find a solution; we may reach a state where no move leads to a reduction in conflicts

Local Search: Hill Climbing

N queens (n = 4)

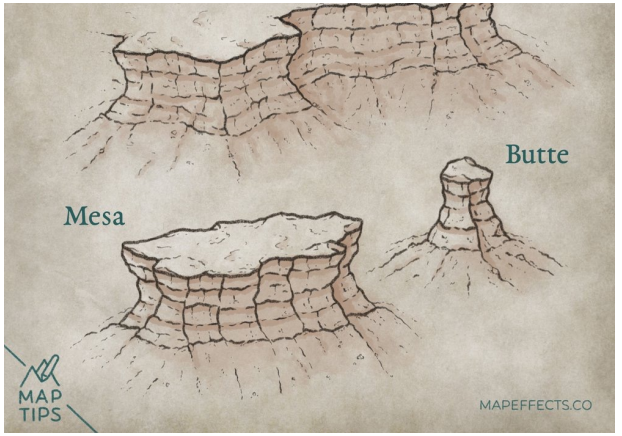
n: <= s * n^2 iterations

Selected moves for each step



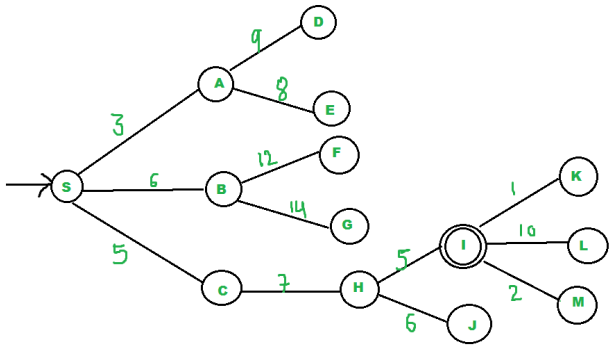
Uses a search path, not a search tree
⇒ highly scalable

Problems with hill climbing



- The **foothills** problem: We reach a **local maximum**.
  We are not at the overall maximum, but from the node we're at, every successor is '**downhill**'
  - Likewise, a hill-climbing approach to route-finding would be defeated if a detour took us even a block in the 'wrong' direction
- The **plateau** problem: There are many similar **plateaux**, about equally good, but to reach the real solution we need to be on a different plateau (you're on the 23rd floor, and the apartment you're looking for is on the 23rd floor—of the building next door)
  - Or, there are additional steps up but we have to go through several steps at the same level (no progress) to get there—can we even find the way up?
- The ridge problem: Heuristic values indicate we're approaching the goal, but we're really not (right end of the department store, but on the wrong floor)
- Remedying problems:
  - Backtracking from local minima to a previous point where a decision was made, trying a different path
  - Try to get to a new region of the search space: random restart is one way of doing this. (Add step 5 to our N-queens method: If no progress can be made, return to step 1 and place queens on board from scratch)

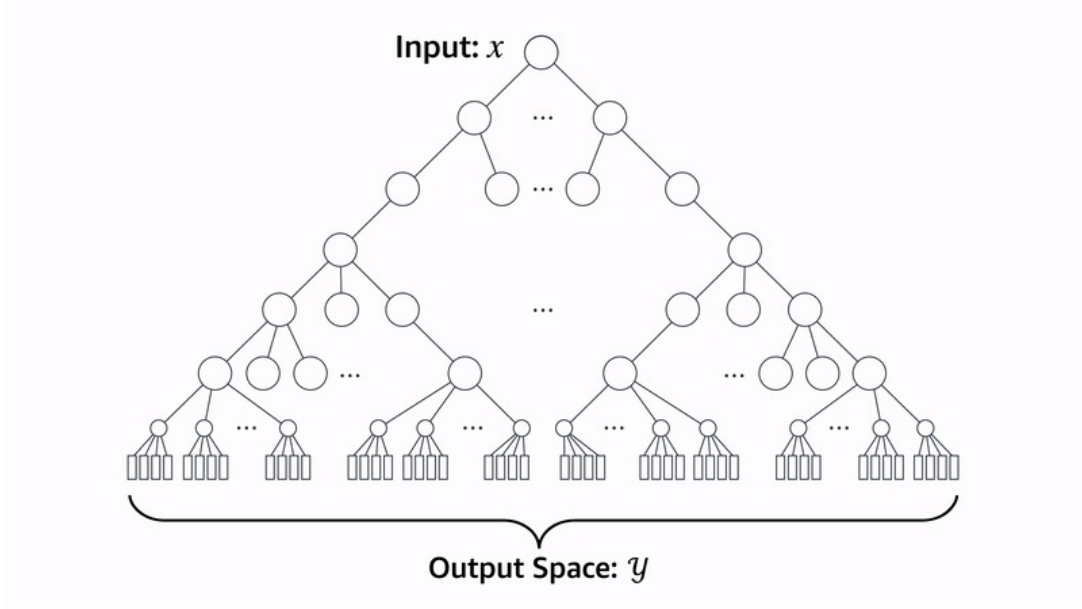
Best-first search



- The basic problem with hill climbing is that it's short sighted
- Steepest Ascent tries to choose intelligently but is still susceptible to dead ends and all the other problems of hill climbing (ridge, plateau, etc)
- Like hill climbing, *best-first* search maintains an open list of nodes that are on the frontier and may be explored, and a closed list of nodes that have been examined or excluded from further search
- The open list is ordered by estimated (heuristic) proximity to the goal and the node estimated closest to the goal is tried first
- Thus the search is ordered by the most promising nodes first
- Duplicate nodes with more than one path of different costs are not maintained separately; only the lowest-cost heuristically shortest node is maintained
- Pseudocode, p. 87
- Unlike hill climbing, best-first can recover from dead ends, false starts, etc
- If we filter out false starts, dead ends, etc., the closed list gives the best solution found
- Note that the quality of the solution found depends heavily on the quality of the heuristic

Beam Search

- The beam search focuses a search by only going down part of the search tree at each level.
- From the start state, all successors are generated as for **BFS**.
- The best W are then selected.
- For that **W**, all **successors** are generated, and the **best W of those successors** are generated.
- The number of items in the 'beam' remains W as we move down the tree.
  - Fig. 3.11, p. 89
- In general a wider beam (**larger** value of **W**) leads to better solutions
- Adding backtracking can increase efficiency more
- Useful in problems with high branching factor and deep searches



Choosing heuristics

- A search begins at start node S and is searching for goal node G, the path to which is unknown.
- Heuristics are used to search most efficiently for the goal
- A search algorithm is ***admissible*** if it always results in an optimal solution if such a solution exists
  - Note that an admissible algorithm does NOT guarantee a shortest path to intermediate nodes, only to the goal
- A search algorithm is ***monotonic*** or ***consistent*** if it also guarantees optimal paths to intermediate nodes
- A **monotonic** algorithm is always **admissible**; in practice, devising a heuristic that's admissible but not monotonic takes some doing
- Suppose we have two admissible heuristics, h1 and h2, and that for some node N, h1(N) < h2(N). Then h2 is said to be *more informed* than h1.
- If this relationship holds for all nodes N, then h2 *dominates* h1.
  - For the 15-puzzle, the sum of Manhattan distances each tile needs to be moved dominates just counting the number of tiles out of place. Either will work, but h2 will work better.
- Suppose we have multiple admissible heuristics and none dominates the others?
  - If all are admissible, then all heuristic values are <= the actual value.
  - Therefore, the maximum of these comes closest to the actual value and gives the best estimate
  - Thus, if we have multiple admissible heuristics, we can compute each and take the maximum as our final value. If that's different heuristics for different nodes, so what?

Generating heuristics

- If the problem definition is written in a formal language, we can develop **relaxed** problems automatically
- A tile can move from square A to square B **if**:
  - A is horizontally or vertically adjacent to B AND
  - B is blank
- We can generate 3 heuristics by removing constraints, or ***relaxing*** the problem:
  - A tile can move from square A to square B if A is adjacent to B
  - A tile can move from square A to square B if B is blank
  - A tile can move from square A to square B
- The first is the Manhattan distance (H2 above); the second says we can pick up any tile and drop it into the blank square (thus swapping locations with the blank); the last is H1.
- Note that the relaxed problems can be solved easily, with little if any search

Generating admissible heuristics from subproblems

- We can also generate heuristics from subproblems.
- Suppose we marked tiles 5,6,7,8 with stars to indicate “don’t care”
- Then a solution of this problem would have 1-4 in place and 5-8 in some arbitrary arrangement
- Clearly this would be a lower bound on a solution for the full puzzle
- The idea behind a **pattern database** is to store exact solution costs for these subproblems—5-8 don’t care, 1-4 don’t care, odd numbers don’t care, even numbers don’t care, etc—and take the largest of them
  - Build up a database of every possible configuration of each, and exact solution time for each
  - Each database yields an admissible heuristic
- Can they be added (i.e. h(1-4 don’t care) + h(5-8 don’t care))?
- NO, they may have overlapping moves.
- BUT we can omit overlapping moves. In other words, the 1-4 don’t care database only records the number of moves of tiles 1-4. The 5-8 don’t care database only counts moves of tiles 5-8
- The sum of these is still a lower bound on solving the entire problem. These are **disjoint pattern databases**
- With these, we can solve 15-puzzle in a few milliseconds, reducing the number of nodes generated by a factor of 10,000 compared to Manhattan distance. For 24-puzzles, a speedup of a million can be obtained
- This only works because each move only affects 1 tile. Each move of a Rubik’s cube affects 8 or 9 of the 26 cubies

Combining actual and heuristic values

- Suppose we are search for a **goal** and are at some node N.
- We may be **interested** in the **shortest** path from **S** to **G** that goes through that **node**.
- That would be H(N) + H(G),the **estimated** cost **from Start** to that node, and the cost from that node to Goal.
- But if we're using a **monotonic** heuristic, the **actual cost from start** to N is the **minimum** cost. Thus we can **estimate** the total cost as:
  - **F(N) = g(N) + h(G)**
  - Actual cost from start to N, and **heuristic** cost from N to Goal
  - We'll see this again shortly, after laying some groundwork…

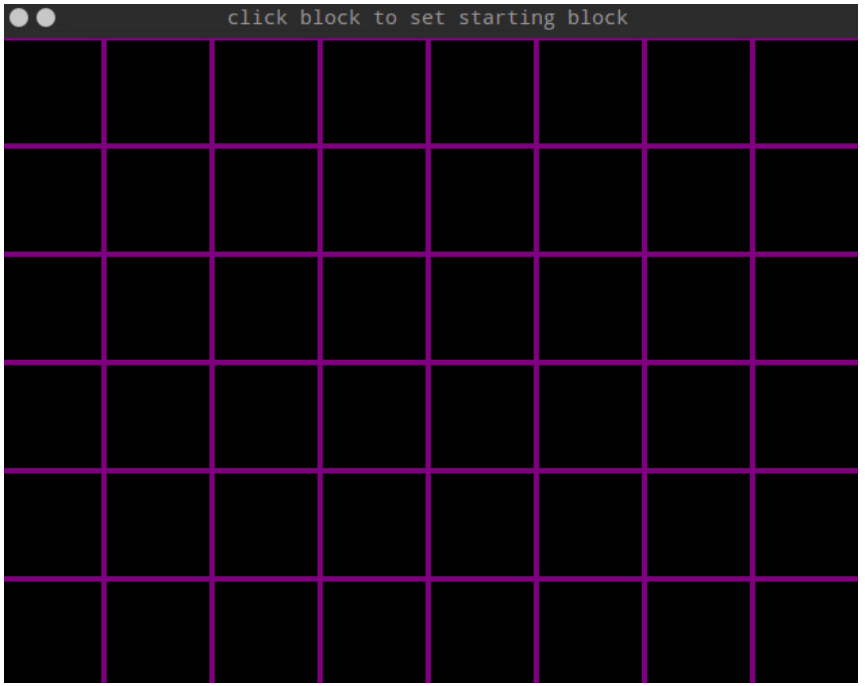
Branch & Bound Methods

- "Plain vanilla" branch-and-bound is also known as ***uniform*** *cost* search
- It orders nodes by **distance from the start node**; there is no estimate of remaining distance
- This is **complete**, since every node is eventually checked
- It is **optimal**, since when the **goal state is reached**, there must **not** be **any** *shorter* path (else we would have found it earlier)
- We can **ensure** this by continuing to develop other partial paths we've located until all have **costs greater than the one we found**
- Example, p. 92-93
- Still doesn't work well for Traveling Salesman, which is NP-Complete

Branch and Bound with underestimates

- Suppose we only have estimated costs to nodes, and won't know the actual cost until we expand that node?
- If our heuristic is admissible, then the actual cost to a node may be higher, perhaps much higher, than its heuristic cost.
- But we can also use the heuristics to at least roughly decide which to expand next.
  - We are at a node with an actual cost of 20 (from the start), and two successors, one with a heuristic cost of 22 and one with a heuristic cost of 24.
  - We select the node with heuristic cost 22 and expand it, finding out that its actual cost is 40.
  - We note that actual cost… and return to the node with a heuristic cost of 24 to try next
- Note that we still don't use any estimate of the distance to the goal.
- This version of branch and bound is more informed than uniform-cost; either is admissible
- Example, p. 97-98
- Note that we may find alternate paths to some nodes—in the above example, we may find a path to the 40-cost node that only costs 30.
  - In that case, we should delete the higher-cost path and retain the lower cost path
- This relates to the principle of optimality: Optimal paths are constructed from optimal sub-paths
  - If the optimal path goes from S through N to G, then the path from S to N must be the shortest such path

The A* Search



- If we have heuristic estimates available for the distance to the goal, we can combine them by selecting nodes based on the sum of their actual cost from the start node plus the estimated (heuristic) cost to the goal:
  $$f(n) = g(n) + h(n)$$
- This is A*, the workhorse of informed search
- It is complete and optimal
- It is also *optimally efficient*: No other optimal algorithm examines fewer nodes in finding the goal path
- If heuristics are available, A* is usually method of choice
- Its storage needs can still become large, depending on the search graph

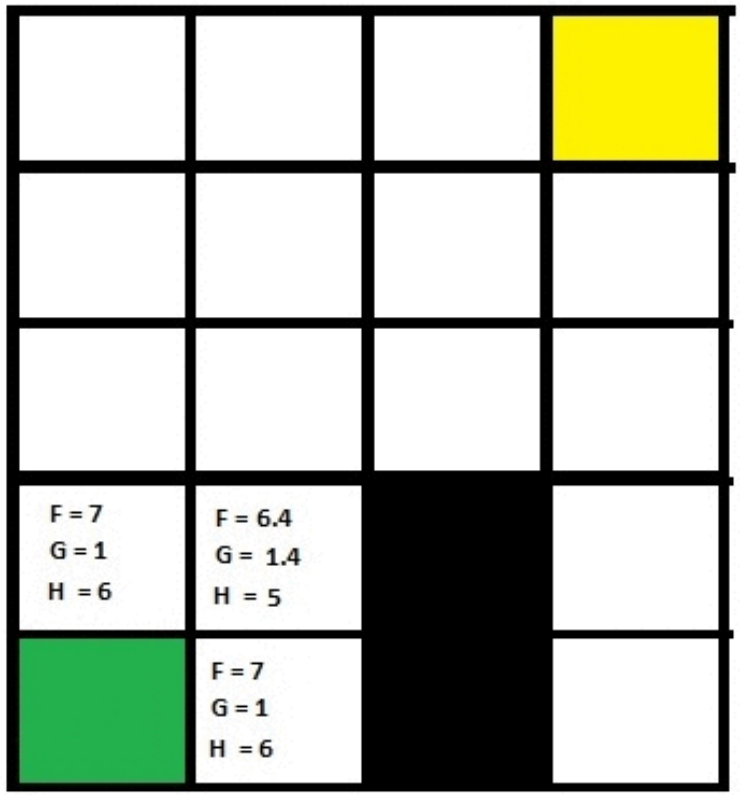
The A* Search

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

where

- $f(n)$ = total estimated cost of path through node n
- $g(n)$ = cost so far to reach node n
- $h(n)$ = estimated cost from n to goal.

(This is the heuristic part of the cost function)



The A* Search

https://qiao.github.io/PathFinding.js/visual/

Five men who are originally from different countries live in consecutive houses on a street. **Given** are each man's **job**, the colors their houses are **painted**, the favorite **drink** of each household, and the **pets** they keep.
We want to know: Who owns a zebra? And whose favorite drink is mineral water?
The following rules (constraints) are provided:

- The English man lives in a red house
- The Spaniard owns a dog
- The Japanese man is a painter
- The Italian drinks tea
- The Norwegian lives in the first house on the left
- The green house immediately to the right of the white one
- The photographer breeds snails
- The diplomat lives in the yellow house
- Milk is drunk in the middle house
- The owner of the green house drinks coffee
- The Norwegian's house is next to the blue one
- The violinist drinks orange juice
- The fox is in a house that is next to that of the physician
- The horse is in a house next to that of the diplomat

House	**FL**: Far Left	**L**: Left	**M**: Middle	**R**: Right	**FR**: Far Right
Nation (N)	**N1**: English	**N2**: Spanish	**N3**: Japanese	**N4**: Italian	**N5**: Norwegian
Color (C)	**C1**: Red	**C2**: Green	**C3**: White	**C4**: Yellow	**C5**: Blue
Drink (D)	**D1**: Tea	**D2**: Orange Juice	**D3**: Milk	**D4**: Coffee	**D5**: Mineral Water
Pet (P)	**P1**: Dog	**P2**: Fox	**P3**: Horse	**P4**: Snails	**P5**: Zebra
Job (J)	**J1**: Painter	**J2**: Diplomat	**J3**: Physician	**J4**: Violinist	**J5**: Photographer

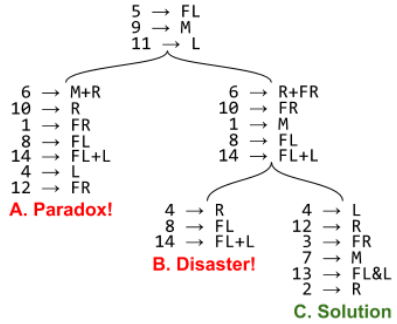
House	FL	L	M	R	FR
Nation	N5	N4	N1	N2	N3
Color	C4	C3	C1	C2	C5
Drink	D5	D1	D3	D2	D4
Pet	P2	P5	P4	P1	P3
Job	J2	J5	J5	J3	J1

The English man lives in a red house	N1 = C1
The Spaniard owns a dog	N2 = P1
The Japanese man is a painter	N3 = J1
The Italian drinks tea	N4 = D1
The Norwegian lives in the first house on the left	H1 = N5
The green house immediately to the right of the white one	C2 = Hx & C3 = Hx+1
The photographer breeds snails	J5 = P4
The diplomat lives in the yellow house	C4 = J2
Milk is drunk in the middle house	H3 = D3
The owner of the green house drinks coffee	C2 = D4
The Norwegian's house is next to the blue one	N5 = Hx & C5 = Hx±1
The violinist drinks orange juice	J4 = D2
The fox is in a house that is next to that of the physician	P2 = Hx & J3 = Hx±1
The horse is in a house next to that of the diplomat	P3 = Hx & J2 = Hx±1

Rule	1	2	3	4	5	6		7	8	9	10	11		12	13		14	
House					FL					M								
Nation	N1	N2	N3	N4	N5							N5						
Color	C1					C3	C2		C4		C2		C5					
Drink				D1						D3	D4			D2				
Pet		P1						P4							P2		P3	
Job			J1					J5	J2					J4		J3		J2

Constraint Satisfaction

Rule	1	2	3	4	5	6		7	8	9	10	11		12	13		14	
House					FL					M								
Nation	N1	N2	N3	N4	N5							N5						
Color	C1					C3	C2		C4		C2		C5					
Drink				D1						D3	D4			D2				
Pet		P1						P4							P2		P3	
Job			J1					J5	J2					J4		J3		J2



Zebra Puzzle

- Another application of search is **constraint satisfaction**.
- Rather than searching through states, we search through possible assignments of variables
- More formally:
  - Given a collection of **n variables**, with each variable Vn (V1, V2, etc) having a domain of **possible values** D1, D2, D3, etc., it can take on, and a set of **k** constraints on those possible values C1, C2, … Ck:
  - A *solution* is an **assignment** of values to **variables**, with each value **assigned** being from the correct domain for that variable
  - A solution that assigns a value to all variables is *complete (*otherwise it is *partial)*.
  - A solution is *consistent* **iff** all constraints are satisfied by it
  - A solution is *correct* **iff** it is complete and consistent.
- An advantage of constraint satisfaction is that it's **general**; we can write a general constraint-satisfaction solver and turn it loose on ANY problem that can be formulated appropriately
  - We construct a **constraint graph**, with each node a variable, *connected if they share a constraint*.
  - We can then apply search techniques to that graph
  - We can also go beyond "this isn't a solution, go on to the next state" by **identifying which constraints are violated** & focusing on **assignments** that **satisfy** them
- Many real-world problems fall into this category; for example, observation time on the Hubble telescope involves starting & ending times & telescope orientations, subject to a variety of priority & power constraints

Constraints

- There are several types of constraints:
  - **Unary** constraints apply to 1 variable
  - **Binary** constraints apply to 2 variables
  - **N-ary**, etc., …(follow from here)
  - **Global** constraints apply universally
    - ALLDIFF: each variable must be assigned a *unique* value.
  - There are also softer *preference constraints* in which we have a **preferred** state and apply a **penalty (cost)** to other options
    - I'd prefer to fly in the **afternoon**, but will fly in the **morning** if it'll save me $100 or more.
    - Prof. Smith prefers to teach in the **morning**, Prof. Jones in the **afternoon**. A schedule with both of them teaching in the afternoon is valid but ***not optimal***.

Constraint Propagation & Inference

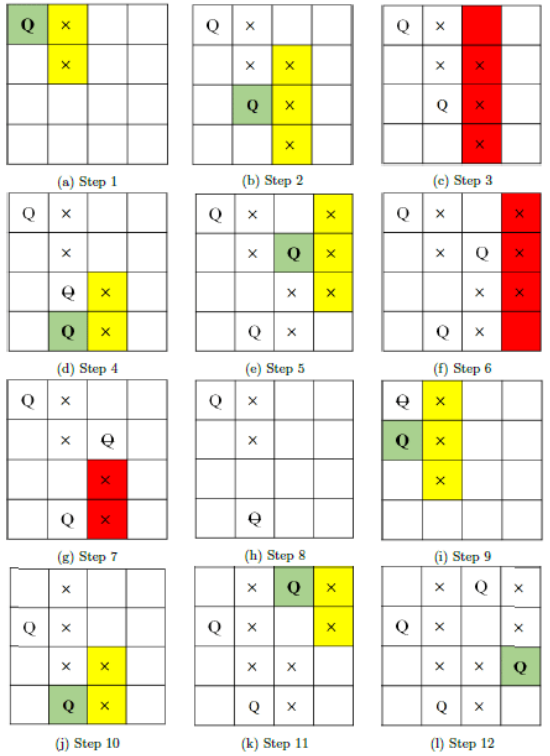
- In *Constraint Satisfaction Problems* (CSPs), an algorithm can search (*try* another assignment of values to variables) or do a type of **inference** called **constraint propagation**
  - Using the constraints ***already known*** to reduce the number of legal values for a variable, which in turn might limit possible values for other variables.
- Key idea is **local consistency**
  - Each variable is a node on a constraint graph.
  - Enforcing local consistency throughout the constraint graph causes inconsistent values to be eliminated
- A variable is **node-consistent** if all values in that variable's **domain** satisfy the variable's unary constraints
  - Node consistency can be enforced **one variable at a time**.
  - N-ary constraints can be made into (decomposed)**binary** constraints
  - Therefore most CSP solvers only work with binary constraints
  - But we can generalize this concept to **hyperarc consistency** or **generalized arc consistency** to deal with multiple values
- A variable is **arc-consistent** if every value in that variable's domain satisfies the variable's binary constraints
  - Arc consistency **eliminates** values of each variable domain that can ***never satisfy a particular constraint*** (an **arc**).
  - More formally, X_i is **arc**-consistent with respect to X_j if **for every value in D_i** there **exists some value** D_j that **satisfies** the binary constraint
- A network is arc-consistent if every variable is arc-consistent with respect to every other variable
  - For example, suppose X and Y are integers in {1,2,3…9,10} and we add the constraint $Y = X^2$.
  - We can make this arc-consistent if we reduce the range of X to {1, 2, 3} (to make X arc-consistent with Y) and Y to {1, 4, 9} (to make Y arc-consistent with X).

AC-3

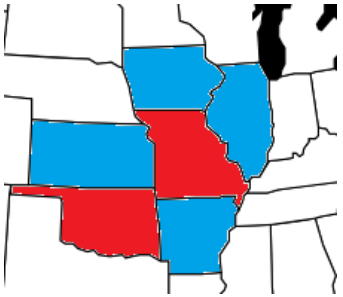
- The most popular algorithm for arc consistency is AC 3
  - Maintain a **queue** (or set) of **arcs** to consider;
    (initially this is all arcs in the CSP)
    - Binary constraints become *two* arcs, one in each direction

Take an arbitrary arc (X_i, X_j) from the queue
  make X_i arc-consistent with X_j
  If this has no effect on D_i
    go on to the next arc
  But if this reduces D_i,
    add to the queue all arcs (X_k, X_i) [where X_k is a neighbor of X_i]
    (The reduction in D_i might affect D_k even if we've already processed D_k)
  If D_k is reduced down to nothing
    CSP has no consistent solution
      return **failure**
Otherwise continue (until no more arcs are in queue)

- At this point we have a CSP equivalent to the original—it has the same solutions—but it has smaller domains and so can be searched faster
- If variables all have domains of size at most *d* and there are *c* binary constraints, checking consistency of an arc takes at most time $O(d^2)$ and so we get total worst-case time of $O(cd^3)$



Path consistency



- Arc consistency can go a long way (sometimes all the way) to a solution
  - Solution, by reducing every **domain** size to **1**
  - OR -- Verifying that there is **no solution**, by reducing some **domain** size to **0**
- But sometimes not enough
- Consider the map coloring with only R, B
- Every variable can be arc-consistent (R at one end, B at the other) but still no solution
  - Since MO, IL, KY all touch each other, we need 3 colors just for them
- Path consistency **tightens** binary constraints by looking at **triples**
  - A 2-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m
    - if for every assignment $\{X_i = a, X_j = b\}$ consistent with binary constraints on X_i and X_j
      - there is assignment to X_m that satisfies binary constraints on (X_i, X_m) and (X_m, X_j)
  - That is, there is an assignment for all variables that makes a path from X_i to X_j passing through X_m consistent with all binary constraints on all 3 variables
- So, for example, we try the 2-color map problem for MO, IL, IN, KY.
- Possible solutions are {MO = R, IL = B} and {IN = R, IL = B} So far so good.
- But with both of those assignments, KY can be neither R nor B because it would conflict with one or the other
- Therefore there is no valid option for KY. The domain is of size 0, which tells us this problem has no solution.
- The PC-2 algorithm applies the same basic idea as the AC-3 algorithm.

K-consistency

- Taking this further, a CSP is **K-consistent** if for any set of K-1 variables and for any consistent assignment to those variables, a consistent value can be assigned to the **k**th variable.
  - 1-consistency says given the empty set we can make any one variable consistent (node consistency)
  - 2-consistency is the same as **arc** consistency
  - **3**-consistency is the same as **path** consistency
- A CSP is **strongly K-consistent** if it is k-consistent and also (k-1) consistent, (k-2) consistent, etc all the way down to node consistent.
- If we have a graph of N nodes and make it strongly N consistent, we have a solution
  - Choose any consistent value for X1.
  - We are guaranteed to be able to choose a consistent value for X2 because it is 2-consistent
  - We can also choose a consistent value for X3 because it is 3-consistent
  - And so on to Xn. Worst case time is $O(n^2d)$
- No free lunch! Establishing strong N-consistency takes exponential time and space in worst case
- In practice, we commonly compute 2-consistency & sometimes 3-consistency, then fall back on search

Global constraints

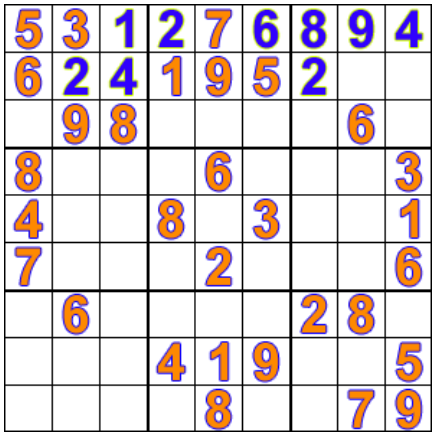
- Recall that a global constraint is one involving an arbitrary number of variables (not necessarily all)
- Special algorithms can be used to reduce the search space or detect inconsistency
  - Consider the Alldiff constraint. If we have **M** values involved in the constraint, and there are a total of **N** possible values among all of them, and **M** > **N**, the constraint cannot be satisfied
  - This leads to a simple algorithm:
    - **If any variable has a singleton domain** (1 possible value), assign **that** *value* to **that** *variable*, and *remove that value from the domain of the remaining variables*
    - Repeat as long as there are singletons.
    - If an empty domain ever occurs or there are more variables than domain values, an inconsistency has been detected

Other constraints

- We also have the **resource constraint**, also known as the **atmost** constraint.
  - We have **4** tasks {P1,P2,P3,P4} and 10 employees.
  - We can specify this as **atmost**(10, P1, P2, P3, P4)
  - Check the sum of the minimum values for each job
    - If each variable has the domain {3, 4, 5, 6} (need **at least** 3 employees for each job), **the constraint cannot be satisfied**
  - We can also enforce **consistency** by deleting **maximum values** if they are inconsistent with **minima**
    - If each variable has range {2, 3, 4, 5, 6}, we can delete values 5 & 6 (after assigning minimum numbers, there aren't enough workers left to put 5 or more on 1 job)

- A CSP is **bounds consistent** if for every variable X and both the lower and upper bounds of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y.

Backtracking search for CSPs



- Some problems (e.g. Sudoku) can be solved entirely via constraint satisfaction
- But sometimes after satisfying all constraints we do not have a unique solution
- Could apply depth-limited search
  - A state is a partial assignment
  - A step is adding var = value
- But this is problematic
  - The branching factor at top level is nd (assign any of d values to any of n variables)
  - At the next level it's (n-1)d, and so on for n levels
  - Thus we generate n! * d^n leaves even though there are only d^n possible assignments
- But this ignores commutativity
  - A problem is **commutative** if the order of application of any given set of actions has no effect on the outcome
- CSP problems are commutative because when assigning values to variables we reach the same partial assignments regardless of application order
- So we only need to consider *one* variable at each node in the search tree
  - We might choose between KY = red, KY = green, and KY = blue, but not between KY = red and MO = blue
- With this restriction, the number of leaves is d^n.

Backtracking search

- A **backtracking search** is a *depth-first search* that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign
  - Choose a variable, then try all values in the domain of that variable in turn
  - If an inconsistency is found, then it returns a failure, causing the previous call to try another value
- With uninformed depth-first search we could speed up search by applying problem-specific heuristics. CSPs can be sped up without problem-specific heuristics, by addressing these questions:
  - Which variable should be assigned next?
  - In what order should values be tried?
  - What inferences should be performed at each step in the search?
  - When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

Variable Ordering

- Simplest method is to choose next unassigned variable
  - But this seldom leads to the most efficient search
  - For example, in our map coloring, after MO = red and IL = green, there is only 1 possible value for KY, so it makes sense to assign KY = blue immediately
    - And after that, assignment for IN is also forced, with only 1 possible value
- This is the **minimum-remaining-values** heuristic: Choosing the variable with the fewest legal values
  - Also known as “most constrained variable” or “fail-first” heuristic
  - If some variable has 0 values left, it will be picked next and the failure detected immediately
- Performs better than a random or static ordering, sometimes by a factor of 1000 or more, but it depends quite a bit on the problem
- But MRV is no help choosing the *first* variable. In this case we use the **degree heuristic**, choosing the variable involved in the most binary constraints
  - Again, the idea is to reduce possible values in other variables as quickly as possible
- MRV is more powerful; degree heuristic is a useful tie-breaker

Value ordering

- Once we've selected a variable, what order do we try to assign its values?
- The **least-constraining-value** heuristic prefers to use the value that rules out the fewest values in surrounding variables in the constraint graph
  - In general, we want the maximum flexibility for future assignments
  - Of course, if we're looking for ALL solutions, not just the first, this doesn't matter since we have to try them all eventually anyway
  - Likewise if there are no solutions
- *Choosing variables is fail-first (most constrained variable), choosing values is fail-last (least constraining value)*
  - Choosing variables fail-first prunes off entire branches of the search tree earlier
  - For values, we only need to find one solution, so we look for the most likely

Local search for CSP

- Local search techniques can be very effective for CSPs
- Use a complete-state formulation, and start in a random state
- Select a variable and search for a value that reduces the number of conflicts and constraint violations
  - The obvious heuristic is **min-conflicts**, the value that results in the minimum number of conflicts. If more than one value gives same minimum number of conflicts, choose arbitrarily
- For n-queens, the solution is approximately independent of problem size, solving the 1,000,000 queens problem in 50 steps after initial assignment.
  - For this problem, solutions are dense throughout the search space
- This has also been used in practical problems.
  - Scheduling observation time for the Hubble telescope used to take 3 weeks to schedule 1 week's observations; now takes 10 minutes

Local search for CSP

- Most of the local search methods we looked at can be used for CSP
- CSP problems often have a series of plateaus, states 1 conflict away from a solution
  - Allowing sideways movement to other states with the same score allows exploring this space
  - Maintaining a **taboo list** of recently-visited states & not allowing return to them can prevent aimless wandering
  - Simulated annealing can also get a search off a plateau
- Another approach is **constraint weighting**
  - Start with all constraints having a weight of 1
  - At each step, choose variable/value that minimizes total weights of violated constraints
  - Increment the weight of all constraints still violated
  - This adds topography, ensuring progress is possible, and adds weight to constraints that are proving difficult to satisfy
- Another advantage of local search is that it minimizes the number of changes, useful in online or dynamic situations
  - Airline scheduling involves elaborate constraints of time, personnel, equipment, etc.
  - If something changes (bad weather at an airport, crew member ill, etc) we must find another solution
  - Local search will find something requiring only a few changes; re-starting the assignments from scratch might find a valid solution requiring dozens (or hundreds!) of changes

AND-OR trees

- For some problems, we have choices which might be equally valid, or some things that must all be done, and other choices of which any one would work
  - We want dinner AND a movie; for dinner we can have pizza OR middle eastern OR Thai; what's within walking distance?
    - Any of the restaurant options can be chosen as part of a solution
    - If a restaurant AND movie theater aren't within the specified range, there is no solution to the overall problem no matter how good the restaurant is, even if there's a pizza place, a middle eastern place, and a Thai place all right across the street
- It can be represented with an AND-OR tree
- A node in such a tree is solvable if
  - It is a terminal node
  - It is nonterminal whose successors are AND nodes that are all solvable or
  - It is nonterminal whose successors are OR nodes and at least one of them is solvable
- Sample AND-OR tree, p. 101

Heuristic Bidirectional Search

- A bidirectional search can be more time effective, if the two parallel searches actually intersect
- A naïve bidirectional search can suffer from the missile metaphor problem: A missile and anti-missile can be targeted toward each other, and both miss
- However, this is not a general problem; as it turns out, wave-shaping algorithms can be used to direct the two searches toward each other (diagram, p. 102)
- Rather than front-to-end heuristic estimates, this method uses front to front estimates—heuristic cost of a path from some node in the "source" front to some node in the "goal" front
- This helps address the frontiers problem—as each search front grows, storage space becomes a problem as the two fronts try to find each other
- Or a frontier search is carried out
  - A node is selected and a BFS out a certain distance from it is carried out, and all nodes stored (usually in a hash table)
  - A forward search then starts from source S, targeting the perimeter nodes, using A* or ID-DFS
  - Likewise, a backward search from goal G, targeting the perimeter, can also be carried out
  - This results in opening about ¼ the nodes of a unidirectional search
  - While there is no connection between the S and G searches, their intersection is the empty set.
  - Once they collide, a search is carried out to find the optimal path within that (smaller) intersection set