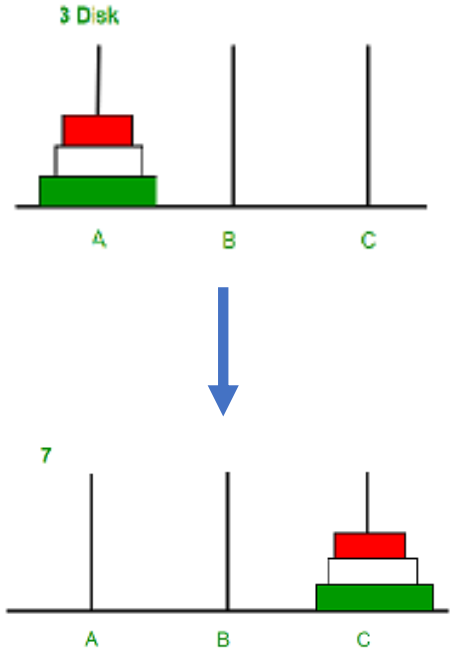
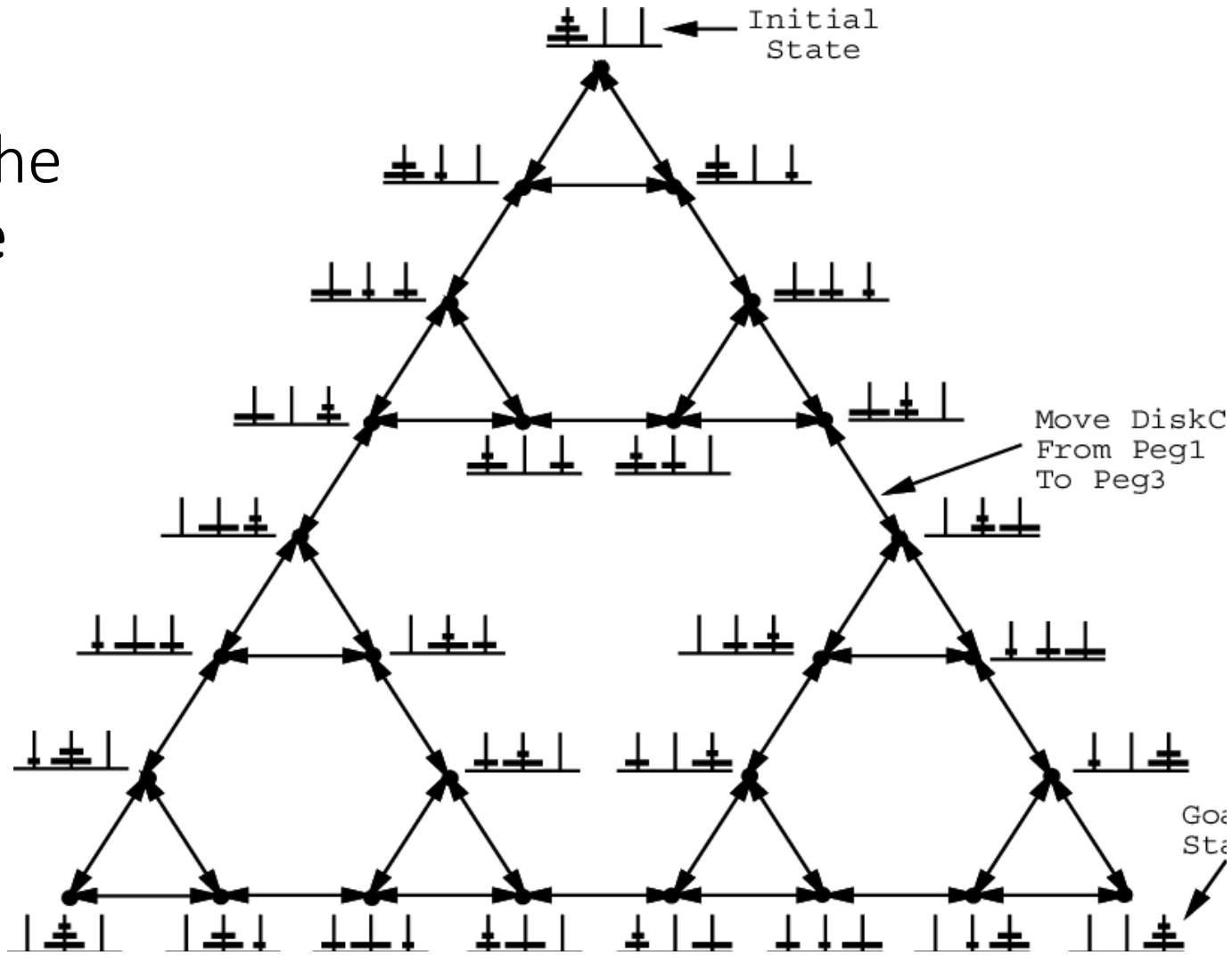
Unknown localization of the **Objective State**



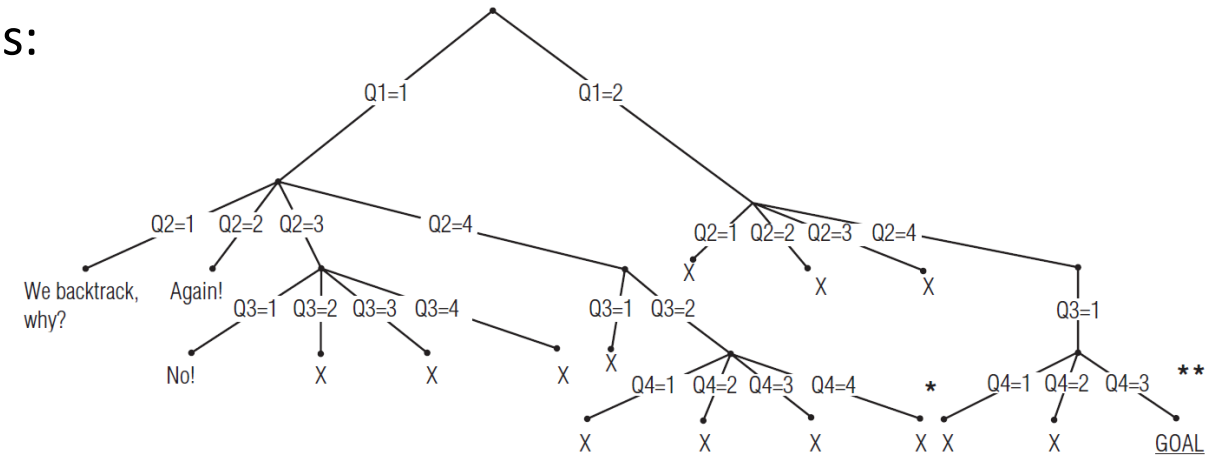


Brute Force algorithms

- 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.

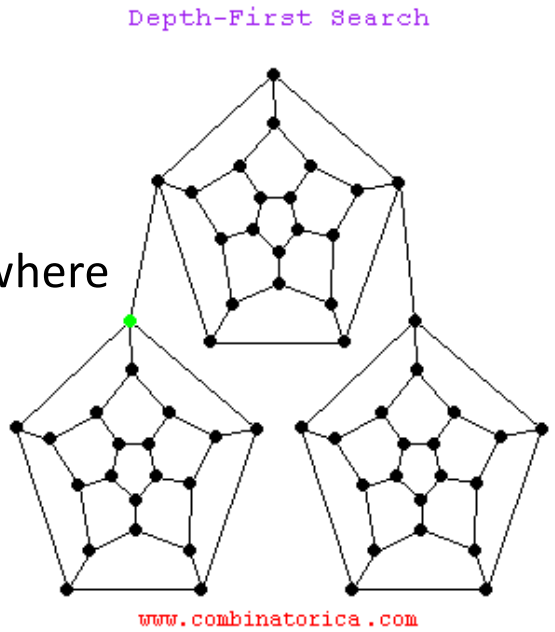
https://algorithm-visualizer.org/brute-force/binary-tree-traversal

- 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



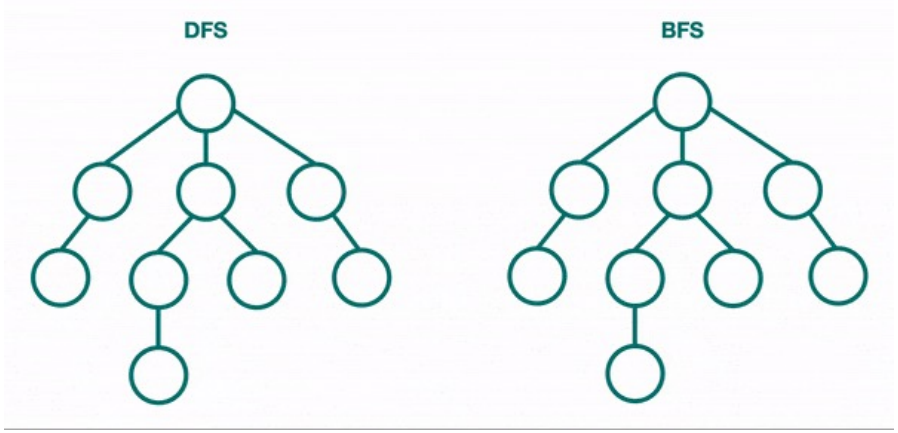
https://graphonline.ru/en/?graph=bigTree

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**)

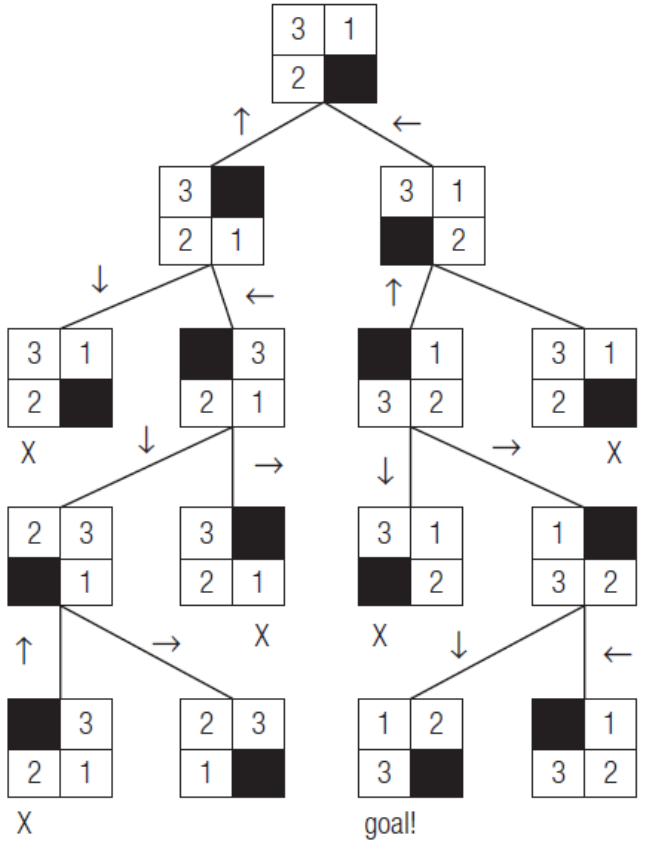
Breadth First Search



- 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)

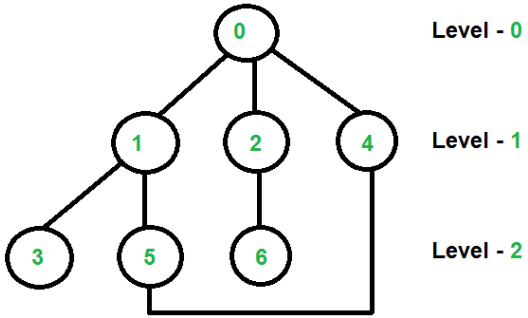
Implementation Issues

- 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)

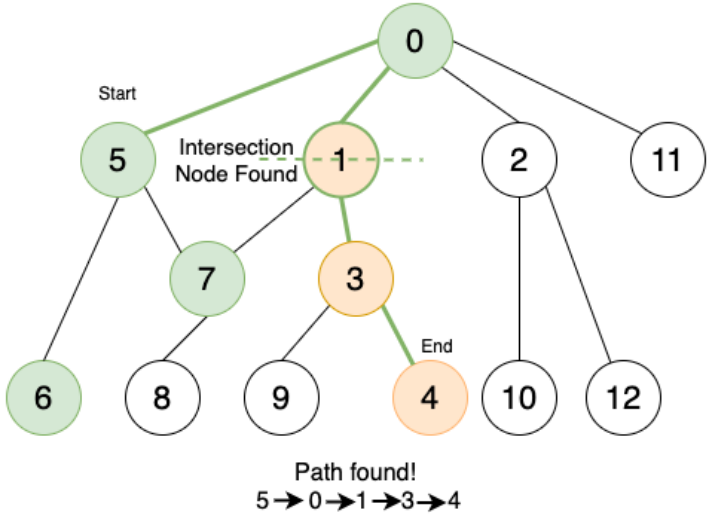
- The idea is to get a similar effect to BFS without paying BFS's exponential storage space.
- Do an exhaustive DFS to depth
- 1. Then do exhaustive DFS to depth
- 2. Then do exhaustive DFS to depth
- 3. And so on. Continue increasing search depth by 1 each round that a solution is not found.
- Note 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}**.

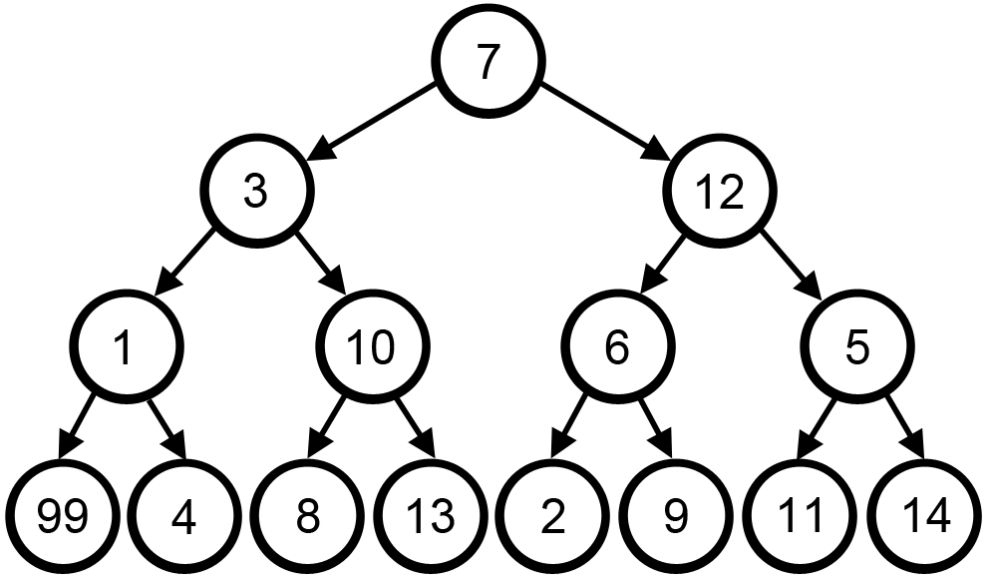
(back to) Greedy Algorithms

•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.

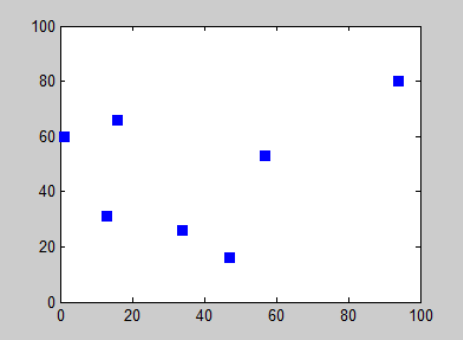
•An **exact** solution requires *exponential time* (the problem is **NP-Complete**) and is thus intractable for large problems

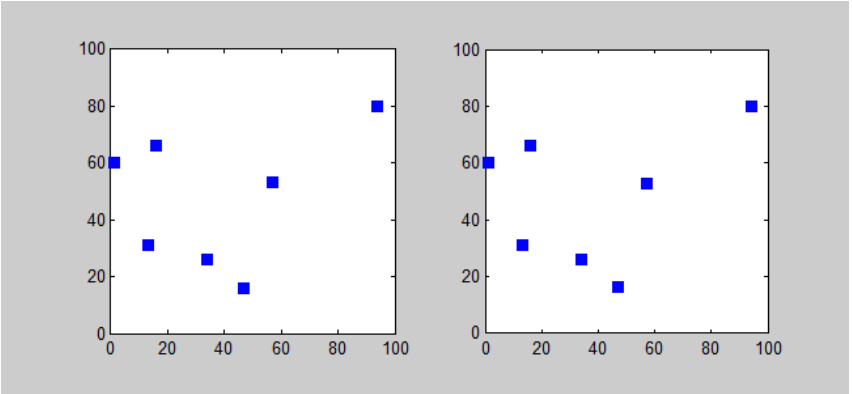
•Unfortunately, **several real-world larger-scale problems** map to (can be directly related) this problem.

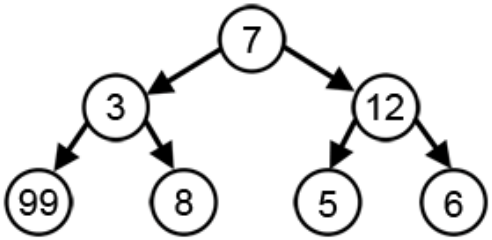
•For example, given 30,000 packages to be delivered in KC (what UPS does in a day) and 200 trucks, assign packages to trucks and find routes so that fuel costs are minimized

•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







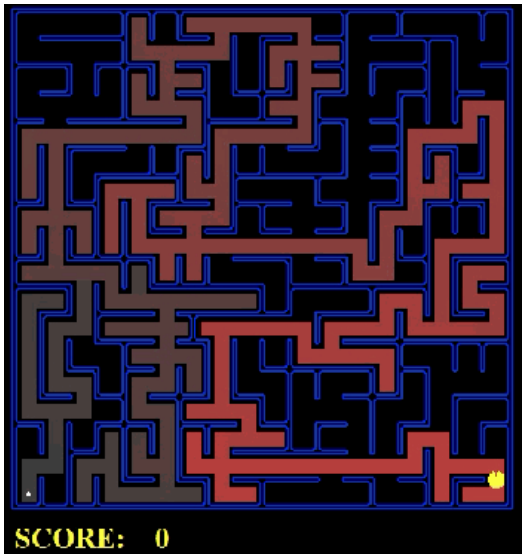
Uninformed Search Won't Do

- 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

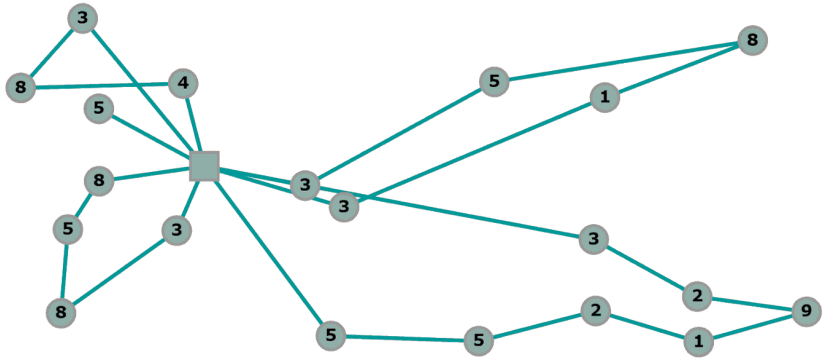


- We want heuristics to have certain properties:
  - They should **underestimate** the actual cost to the goal. (This ensures that we don't overshoot the actual goal. We **don't** want to **reject** an option as "too expensive" when the problem is that our heuristic is wrong.)
  - They should be ***monotonic***; 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

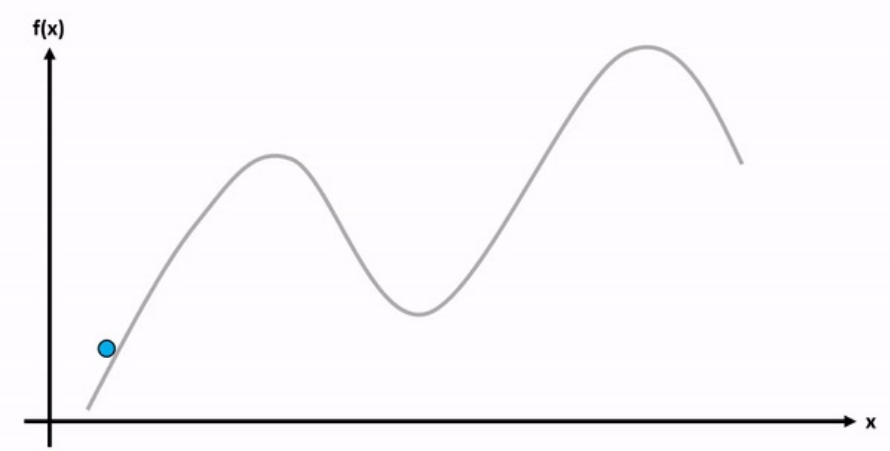
- Hill climbing, beam search, and best-first typically don't look back.
- Other methods include the distance/cost from the **root** as part of their calculations.
- B*ranch and bound* methods always look backwards to monitor progress so far.
  - Adding an estimated cost to reach the goal gives the well-known 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

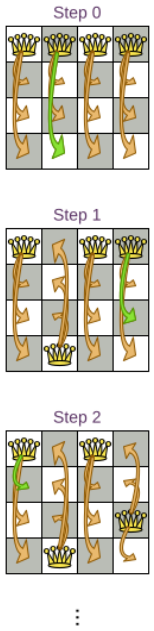
- A hill-climbing approach to N-queens would be:
  1. Begin by placing 1 queen in each row, selecting columns with uniform probability
  2. If the number of conflicts (queens attacking each other) == 0, we have a solution and we're done.
  3. Otherwise, check for each row:
     1. Check for each column:
        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)

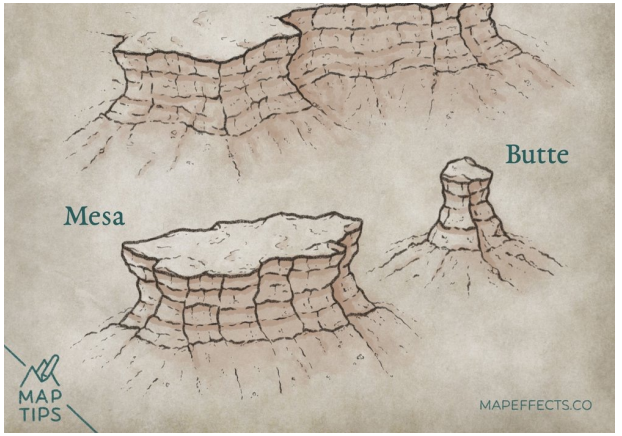
n: <= s * n^2 iterations

Selected moves for each step



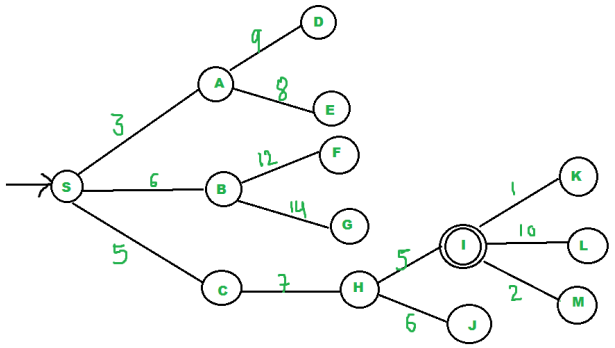
Uses a search path, not a search tree
⇒ highly scalable

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)

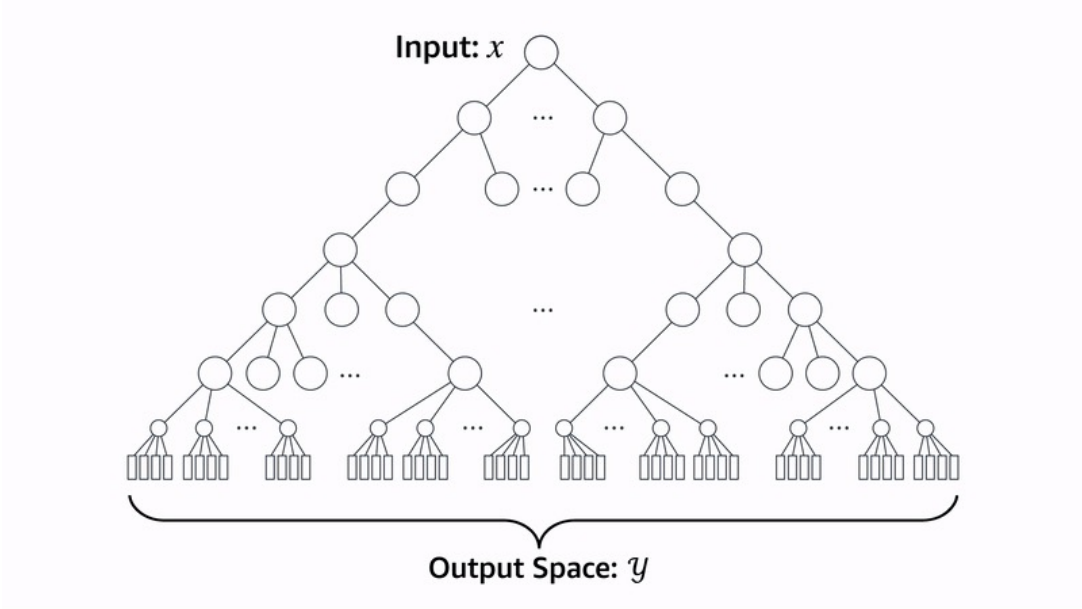
Best-first search



- 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…

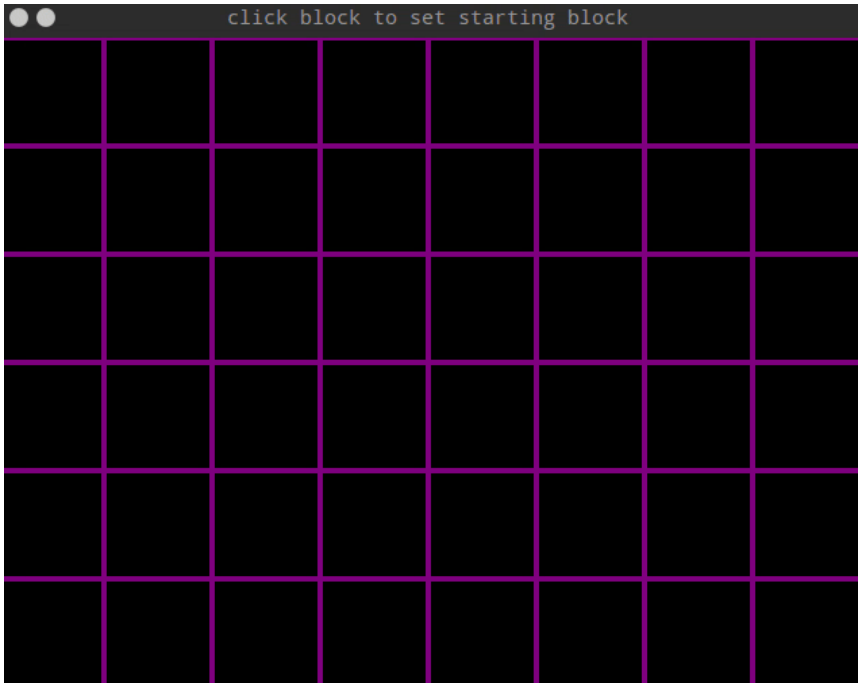
Branch & Bound Methods

- "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

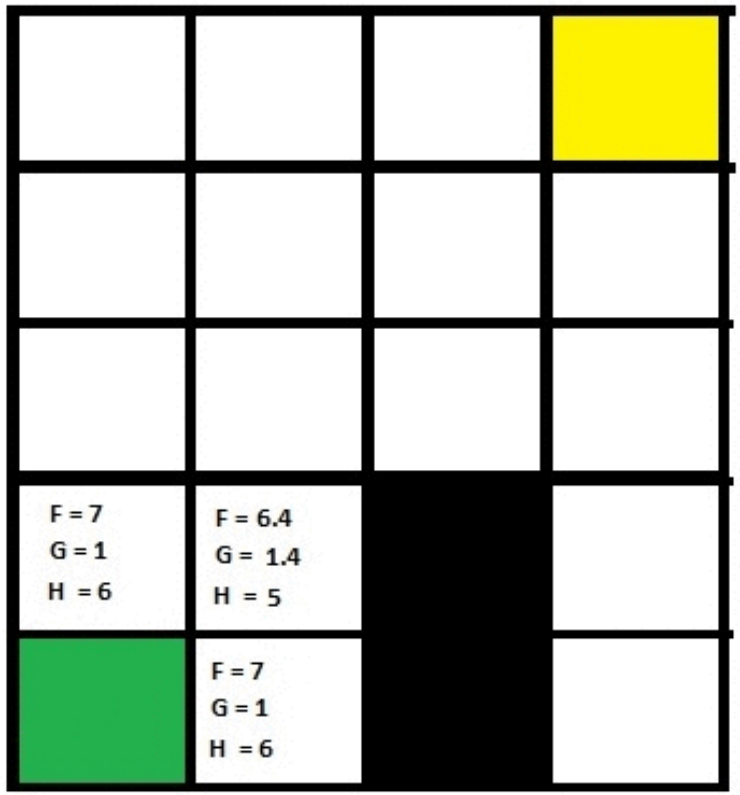
The A* Search

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

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/