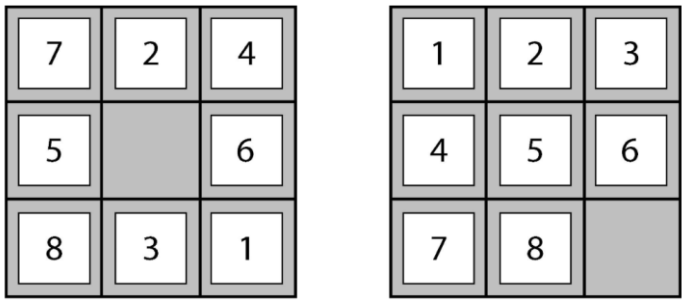
FORMULATING THE PROBLEM

1. Formulate your goal
2. Specify your task
3. Find solution
4. Execute

E.G. for the Romania tasks can be specified as follows:

1. STATE SPACE = set of cities on the map
2. INITIAL STATE = 'at Arad'
3. ACTIONS /OPERATORS = the transition between directly connected cities
4. GOAL STATE = 'at Bucharest'
5. PATH COST = total driving distance along the roads in the path

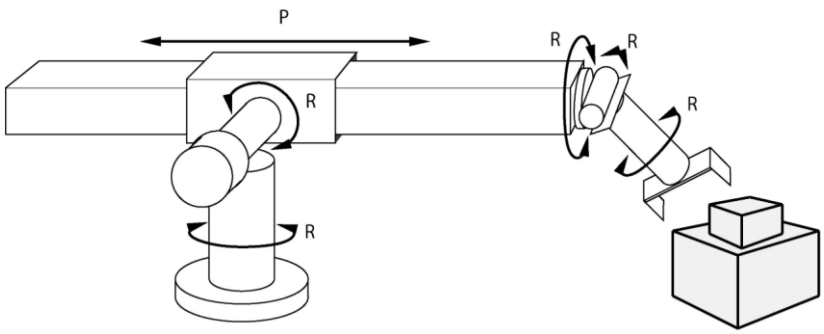
THE 8-PUZZLE



Start State Goal State

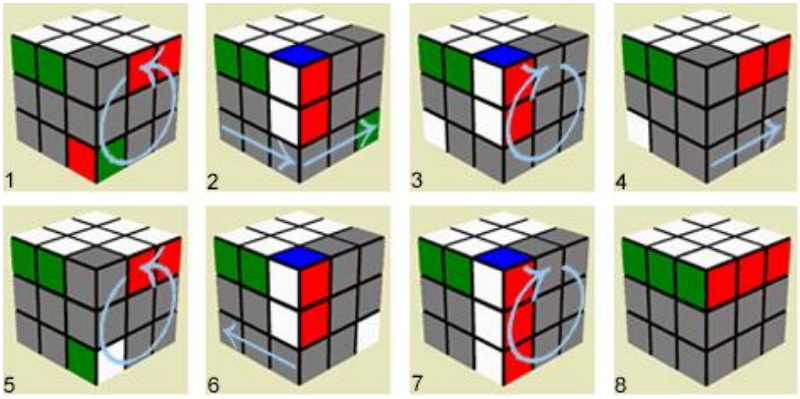
- A 3x3 board with 8 numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The objective if to move the tiles so you can reach the specifies goal state
- STATE = the integer locations of tiles
- OPERATORS = how many moves can we choose from in each state
  = Move the blank to left, right, up, or down. So there are at most 4 possible moves from each state, this number is called the "branching factor"
- GOAL TEST = there is a unique goal state which is given
- PATH COST = one unit of cost for moving one tile. In some formulations, a series of consecutive moves of the blank in the sane direction is considered as a single move.

ROBOTIC ASSEMBLY



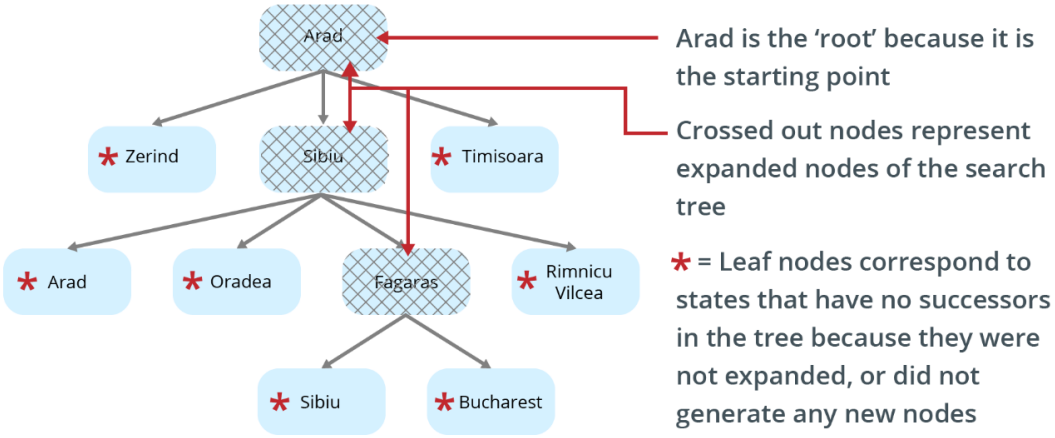
- STATE = the real-valued coordinates of robot joint angles and parts of the object to be assembled.
- OPERATORS = the continuous motions of joint angles
- GOAL TEST = the complete assembly of object
- PATH COST = the time to execute.

RUBIK'S CUBE



- STATE = integer locations of small cubes
- OPERATORS, BRANCHING FACTOR = if we allow only quarter-turns, then the front, back, left right, top or bottom face can be turned either clockwise or anti-clockwise, making a branching factor of 12. Otherwise, if we consider a half-turn as a single move, then the branching factor is 18 (12 quarter-turns plus 6 half turns).
- GOAL TEST = there is a unique goal state (it is given)
- PATH COST = one per move, note that the path cost depends on our formulation. If we consider a half-turn as a single move, it will change the cost of a path (and will change what we consider to be the shortest path to the goal state).

PATH SEARCH ALGORITHMS



- All of the path search algorithms we will discuss follow the same basic pattern of TREE SEARCH
- At each step in the search, one of the leaf nodes in the search tree in EXPANDED, and all possible transitions from that node are EXPLORED, in order to generates new leaves for the tree which are children of the expanded node.

STRUCTURE OF ALGORITHM:

1. Start with a priority queue consisting of just the initial state (the root = starting point)
2. Choose state from the queue of states which have been generated but not yet expanded (在所有已生成路径但还未展开其可能生成路径的 node 中选择一个)
3. Check if the selected state is a GOAL STATE, if it is, STOP (solution has been found).
4. Otherwise, expand the chosen state by applying all possible transitions and generating all its children. (如果其已选择 node 并不是最终结果, 展开并且生成所有这个 node 可能生成的路径).
5. If queue is empty stop (no solution exists)
6. Otherwise, go back to step 2

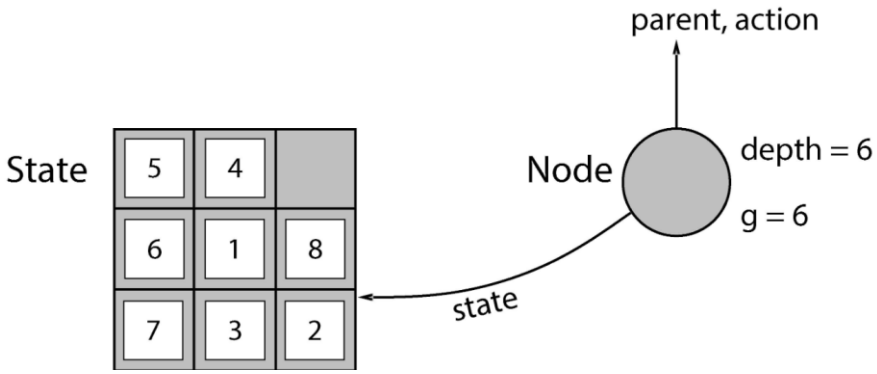
DATA STRUCTURES FOR A NODE

Nodes are not the same as states, some states may appear multiple times in tree, while others night not(yet) occur at all (e.g. Arad exist twice in the tree above)

In some cases we can implement data structures that are specific to a particular search problem (for example, if the state space is a 2-dimensional grid, it makes sense for the nodes to also be stored in a 2-dimensional array), however if the search space has a specific structure, one possibility is to have a NODE data structure with 5 component:

1. Corresponding state
2. Parent node: the node which generated the current node
3. Operator that was applied to generate the current node.
4. Depth: number of nodes from the root to the current node
5. Path cost: (from the root to the current node)

THE DIFFERENCE BETWEEN STATES AND NODES



- Some states may appear multiple times in tree, so you may have two different NODES corresponding to the same STATES, or you can have states that are not in the tree yet because they haven't been generated.
- STATE = a representation of a physical configuration
- NODE = a data structure constituting part of a search tree includes PARENT, CHILDREN, DEPTH, PATH COST g(x).

DATA STRUCTURE FOR SEARCH TREES ADTS

Frontier: a collection of nodes waiting to be expanded. It can be implemented as a priority queue with the following operations:

- **MAKE-QUEUE (ITEMS) creates queue with given items.**
- **Boolean EMPTY (QUEUE) returns TRUE if no items in queue.**
- **REMOVE-FRONT (QUEUE) removes the items at the front of the queue and returns it.**
- **QUEUEING – FUNCTION (ITEMS, QUEUE) inserts new items into the queue.**

SEARCH STRATEGIES

- A strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - Completeness – does it always find a solution if one exists?
  - time complexity – number of nodes generated/expanded (how long does it take to find the solution)
  - space complexity – maximum number of nodes in memory (how much memory does it use)
  - optimality – does it always find a least cost solution? (is it the shortest path to the goal)
- Time and space complexity are measured in terms of
  - **b – maximum branching factor of the search tree**
    - e.g. for the 8-puzzle, b is 4; for the rubik's cube, b is 6 or 12 or 18
  - **d – depth of the least-cost solution**
    - e.g. in the Romania question, the d is tree, Arad -> Sibiu -> Fagaras -> Bucharest (3 steps)

- **m – maximum depth of the state space (may be infinity)**
  - we can say normally its number of cities in graph (for Romania question)

HOW FAST AND HOW MUCH MEMORY?

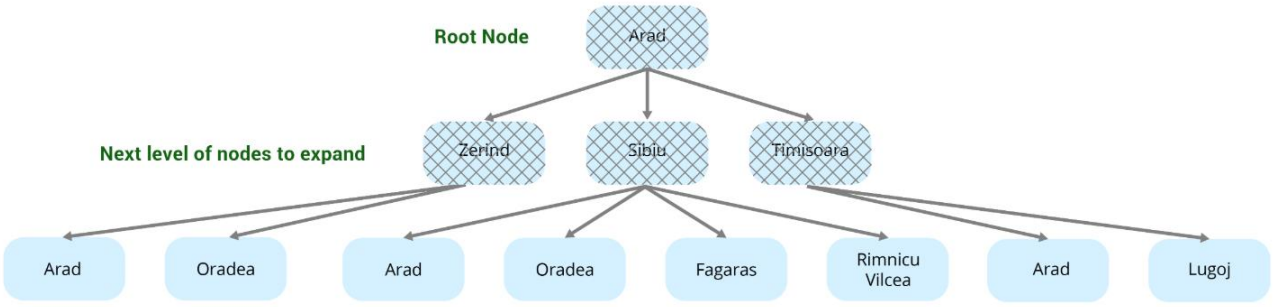
1. **Benchmarking – run both algorithms on a/one/same computer and measure speed (various issues:)**
   a. Run two algorithms on a computer and measure speed
   b. Depends on implementation, compiler, computer, data, network …
   c. Measuring time
   d. Processor cycles
   e. Counting operations
   f. Statistical comparison, confidence intervals
2. **Analysis of algorithms – mathematical analysis of the algorithm**
   a. $T(n)$ is $O(f(n))$ means …$T(n) <= kf(n)$
      i. n = input size
      ii. $T(n)$ = total number of step of the algorithm
   b. Independent of the implementation, compiler, …
   c. Asymptotic analysis: for large n, and $O(n)$ algorithm is better than an $O(n^2)$ algorithm
   d. O () abstracts over constant factors
      i. E.g. $T(100 * n + 100)$ is better than $T(n^2 + 1)$ only for $n > 100$
   e. O () notation is a good compromise between precision and ease of analysis

UNINFORMED SEARCH STRATEGIES

- Uninformed ("blind") search strategies use only the information available in the problem definition (can only distinguish a goal from a non-goal state):
  - Breadth first search
  - Uniform cist search
  - Depth first search
  - Depth limited search iterative deepening search
- For uninformed search, when you reach the goal, you reach the goal, but when you haven't reach the goal yet you don't know if you are close or far away

①BREADTH-FIRST SEARCH

- All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded
  - Sequence: expand the root node first -> expand all nodes generated by the root nodes -> expand all nodes generated by those nodes and so on. (we would normally avoid to repeat to choose the nodes containing the same, which may occur a loop)
  - **RULE: choosing which node to expand, all the nodes in a given depth must be expanded before you move on and expand any nodes in the next depth.**



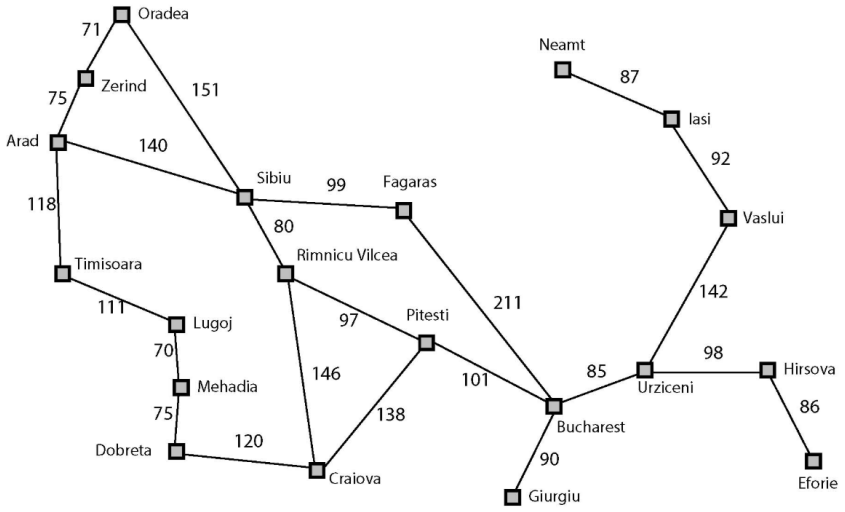
- Expand root first, then all nodes generated by root, then all nodes generated by those nodes…
- Expand shallowest unexpanded node
- Implementation: QUEUEING0FUNCTION = put newly generated successors at end of queue
- Very systematic

- Finds the shallowest goal first

- **COMPLETE**? YES (if b is finite the shallowest goal is at a fixed depth d and will be found before any deeper nodes are generated, e.g. when it reaches the last depth, its guaranteed that it would find the goal)
- **TIME complexity**: $1 + \mathrm{b} + \mathrm{b}^2 + \mathrm{b}^3 + \cdots + \mathrm{b}^{\mathrm{d}} = \frac{b^{d+1}-1}{b-1} = O(b^d)$
  - It can multiply b each time, it's the worst case. **(b is the branching factor, the max number of children that each node can have)**
  - d is the depth (till we go to the last level/depth we would find the goal)
- **SPACE complexity**: $O(b^d)$
  - The problem of breadth-first is that you have to keep all the nodes in memory because of the goal is not there, then we need to go back and generate the new path
  - !!! Space is the big problem for breadth-first search: It grows exponentially with depth
- **OPTIMAL**? YES, but only if all actions have the same cost
- ** When is the best time to use breadth-first search: in the low-dimensional search like a maze, graph, 2-dimensional graph. (if the branching factor is low)

②UNIFORM COST SEARCH

- In the previous breadth first search, we assume that all steps have equal cost. However we are often looking for the path with the shortest total distance rather than the number of steps, the following graph shows Romania question with measured step costs in km.
  Breadth-first search will find the path, Arad -> Sibiu -> Fagaras -> Bucharest, because it is only 3 steps, but it has a total distance 450km. if we want the shortest distance, this is not always the optimal search.



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

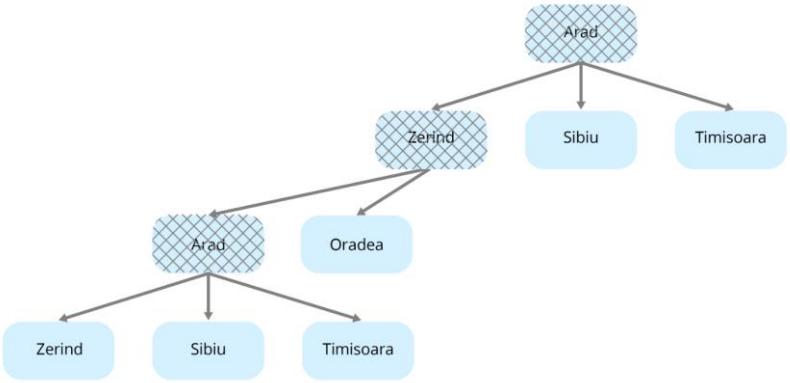
- Expand root first, then expand least-cost unexoanded node
  - For each node, it keeps checking which nodes have the shorest path length
  - As we are going to find the shortest path every time, some situation might happen such as from Arad goes to Sibiu, it only cost 140, but if we chose to go through Zerind and Oradea it will takes 71+75+151 which is much greater than 140.
  - Thus we might use **Dijkstra's algirithm**, keep a record of the shortest path to go each nodes, and we replace them till we find a shorter path.
- Implementation: QUEUEING-FUNCTION = insert nodes in order to increasing path cost
- Reduces to breadth first search when all actions have same cost
- Finds the cheapest goal provided path cost is monotonically increasing along each path (no negative-cost steps)

- **COMPLETE**? YES, if b is finite and step cost $\geq$ ε with $\varepsilon > 0$
- **TIME complexity**: $O(\mathrm{b}^{\lceil C^*/\varepsilon \rceil})$ where c* = cost of optinal solution and assume every action costs at least ε

- **SPACE complexity**: $O(b^{\lceil C^*/\varepsilon \rceil})$ (note: : $b^{\lceil C^*/\varepsilon \rceil} = b^d$ if all step costs are equal)
- **OPTIMAL**? YES, it will always find the shortest-distance path.

③DEPTH-FIRST SEARCH

- For breadth-first search, kept the items in a queue, and the new items go to the back of the queue
- Depth-first is the opposite, it puts items in the front of the STACK, its really important to avoid repeated states along the path.



- This forces us to always expand the deepest unexpanded node, and explore the graph as if we were physically mobbing around in Romania, backtracking when we reach a dead-end
- We keep explore, till it forms a loop/ go back to the states that we've viewed before, and backtracking. We don't need to remember all the nodes that we previously visited, only need to remember the current path that we're exploring, and we have to remember the children of the nodes along the path just in case dead-end happens.
- E.g. to solve maze, depth-first search is faster than breadth-first search

- **COMPLETE**? NO! fails in infinite-depth spaces, spaces with loops; modify to avoid repeated states along path -> complete in finite spaces
- **TIME complexity**: $O(b^m)$ (terrible if m is much larger than d but If solutions are dense, maybe much faster than breadth-first)
- **SPACE complexity**: $O(b * m)$, its linear space!
- **OPTIMAL**? NO, can find suboptimal solutions first.
- Advantage: it uses very little memory

④ITERATIVE DEEPENING SEARCH

- BREADTH-FIRST SEARCH is complete and optimal but uses a lot of memory, and DEPTH-FIRST SEARCH is neither complete or optimal but it has the advantages of using very little memory.
- ITERATIVE DEEPENING SEARCH differs in that it is designed to combine the benefits of BFS and DFS by doing a series of depth-limited searches to depth 1,2,3, etc. until a solution is found
- When we are in depth 0, we just simply test whether the initial state is the goal, then expand the initial state, test any of those are the goal. And then go back to the beginning start from the initial state and do another depth-first search but only search to depth 2, then start again, search to depth 4… till you find the goal.
- So we can say it's a series of depth limited search.
- **COMPLETE**? YES.
- **TIME complexity**: nodes at the bottom level are expanded once, nodes at the next level twice, and so on. (we assume b > 1)
    - Depth limited: $1 + b^1 + b^2 + \cdots + b^{d-1} + b^d = O(b^d)$
    - Iterative deepening: $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + 2 * b^{d-1} + 1 * b^d = O(b^d)$
    - Example: branching factor b = 10, depth d = 5:
        - Depth-limited = 1 + 10 + 100 + 1000 + 10000 + 100000 = 111,111

- Iterative-deepening: 6 + 50 + 400 + 3000 + 20000 + 100000 = 123456
- Only about 11% more nodes (for b = 10)

- **SPACE complexity**: $O(b * d)$