COMP9414/9814/3411: Artificial Intelligence

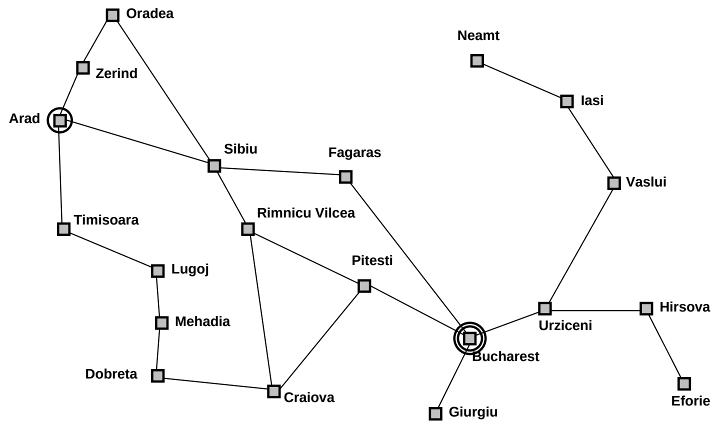
Week 3: Path Search

Russell & Norvig, Chapter 3.

Motivation

- Reactive and Model-Based Agents choose their actions based only on what they currently perceive, or have perceived in the past.
- a Planning Agent can use Search techniques to plan several steps ahead in order to achieve its goal(s).
- two classes of search strategies:
  - Uninformed search strategies can only distinguish goal states from non-goal states
  - Informed search strategies use heuristics to try to get "closer" to the goal

Romania Street Map



Example: Romania

On touring holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest; non-refundable ticket.

- Step 1 Formulate goal: be in Bucharest on time
- Step 2 Specify task:
  - states: various cities
  - operators or actions (= transitions between states): drive between cities
- Step 3 Find solution (= action sequences): sequence of cities, e.g. Arad, Sibiu, Fagaras, Bucharest
- Step 4 Execute: drive through all the cities given by the solution.

Single-State Task Specification

A task is specified by states and actions:

- state space e.g. other cities
- initial state e.g. "at Arad"
- actions or operators (or successor function $S(x)$)
  e.g. Arad → Zerind Arad → Sibiu etc.
- goal test, check if a state is goal state
  In this case, there is only one goal specified ("at Bucharest")
- path cost e.g. sum of distances, number of actions etc.

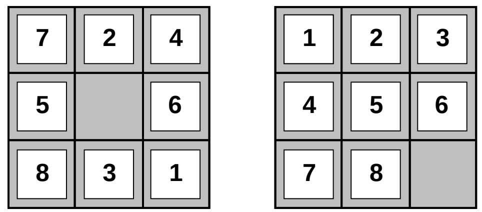
Choosing States and Actions

- Real world is absurdly complex
  ⇒ state space must be abstracted for problem solving
- (abstract) state = set of real states
- (abstract) action = complex combination of real actions
  - e.g. "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - for guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (abstract) solution = set of real paths that are solutions in the real world

Example Problems

- Toy problems: concise exact description
- Real world problems: don't have a single agreed desription

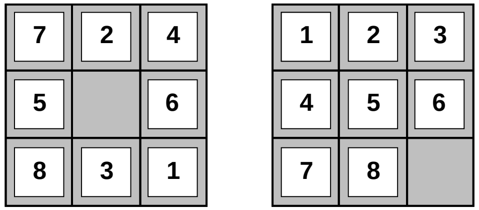
The 8-Puzzle



Start State Goal State

- states: ?
- operators: ?
- goal test: ?
- path cost: ?

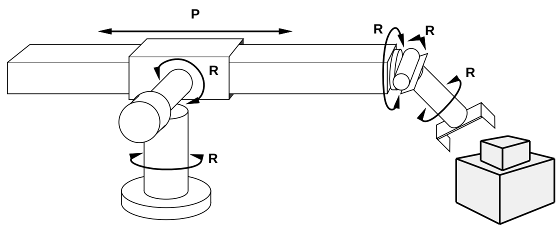
The 8-Puzzle



Start State **Goal State**

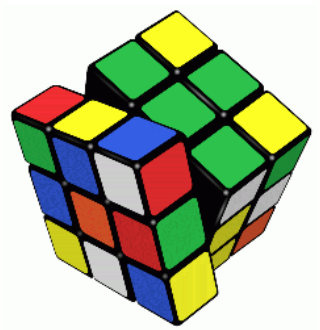
- states: integer locations of tiles (ignore intermediate positions)
- operators: move blank left, right, up, down (ignore unjamming etc.)
- goal test: = goal state (given)
- path cost: 1 per move

Robotic Assembly



- states: ?
- operators: ?
- goal test: ?
- path cost: ?

Rubik's Cube



- states: ?
- operators: ?
- goal test: ?
- path cost: ?

Path Search Algorithms

Search: Finding state-action sequences that lead to desirable states. Search is a function

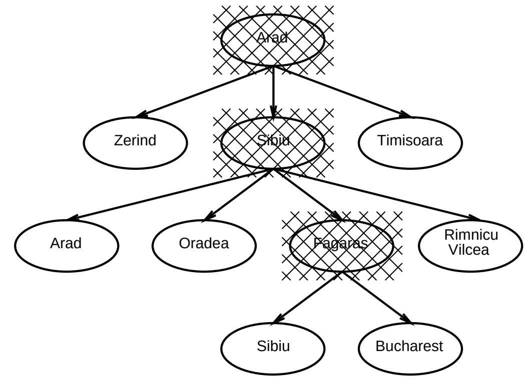
solution search(*task*)

Basic idea:
Offline, simulated exploration of state space by generating successors of already-explored states (i.e. "expanding" them)

Generating Action Sequences

1. Start with a priority queue consisting of just the initial state.
2. Choose a state from the queue of states which have been generated but not yet expanded.
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.
5. If the queue is empty, Stop (no solution exists).
6. Otherwise, go back to Step 2.

General Search Example



Search Tree

- Search tree: superimposed over the state space.
- Root: search node corresponding to the initial state.
- Leaf nodes: correspond to states that have no successors in the tree because they were not expanded or generated no new nodes.
- state space is not the same as search tree
  - there are 20 states = 20 cities in the route finding example
  - but there are infinitely many paths!

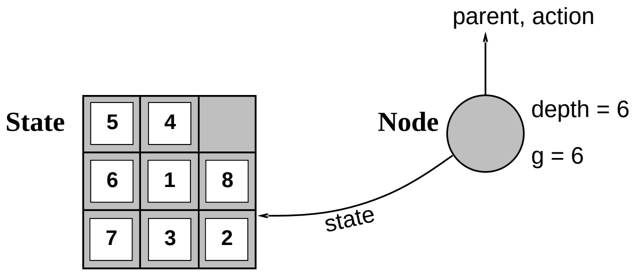
Data Structures for a Node

One possibility is to have a node data structure with five components:

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.

States vs. Nodes

a state is (a representation of) a physical configuration
a node is a data structure constituting part of a search tree
includes parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!



Note: two different nodes can contain the same state.

Data Structures for Search Trees

Frontier: 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 item 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
  - space complexity – maximum number of nodes in memory
  - optimality – does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b – maximum branching factor of the search tree
  - d – depth of the least-cost solution
  - m – maximum depth of the state space (may be ∞)

How Fast and How Much Memory ?

How to compare algorithms ? Two approaches:

1. Benchmarking: run both algorithms on a computer and measure speed
2. Analysis of algorithms: mathematical analysis of the algorithm

Benchmarking

- Run two algorithms on a computer and measure speed.
- Depends on implementation, compiler, computer, data, network ...
- Measuring time
- Processor cycles
- Counting operations
- Statistical comparison, confidence intervals

Analysis of Algorithms

- T(n) is $O(f(n))$ means $\exists n_0, k : \forall n > n_0$ T$(n) \leq k f(n)$
  - n = input size
  - T(n) = total number of step of the algorithm
- Independent of the implementation, compiler, ...
- Asymptotic analysis: For large n, an $O(n)$ algorithm is better than an $O(n^2)$ algorithm.
- $O()$ abstracts over constant factors
  - e.g. T$(100 \cdot n + 1000)$ is better than T$(n^2 + 1)$ only for $n > 110$.
- $O()$ notation is a good compromise between precision and ease of analysis.

Uninformed search strategies

Uninformed (or "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 Cost Search
- Depth First Search
- Depth Limited Search
- Iterative Deepening Search

Strategies are distinguished by the order in which the nodes are expanded.

Informed search strategies

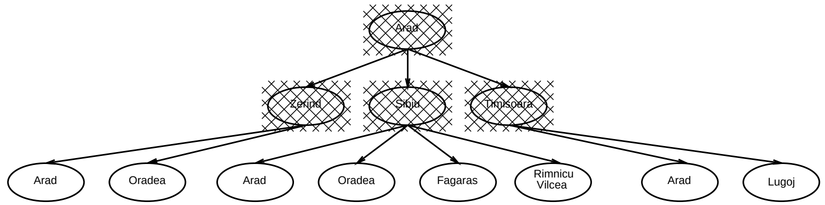
Informed (or "heuristic") search strategies use task-specific knowledge.

- Example of task-specific knowledge: distance between cities on the map.
- Informed search is more efficient than Uninformed search.
- Uninformed search systematically generates new states and tests them against the goal.

Breadth-First Search

- All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded
- Expand root first, then all nodes generated by root, then All nodes generated by those nodes, etc.
- Expand shallowest unexpanded node
- implementation: QUEUEING-FUNCTION = put newly generated successors at end of queue
- Very systematic
- Finds the shallowest goal first

Breadth-First Search



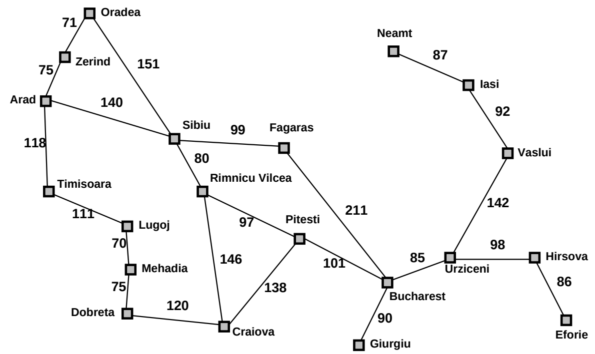
Properties of Breadth-First Search

- 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)
- Time: $1+b+b^2+b^3+\ldots+b^d = \frac{b^{d+1}-1}{b-1} = O(b^d)$
- Space: $O(b^d)$ (keeps every node in memory; generate all nodes up to level d)
- Optimal? Yes, but only if all actions have the same cost

Space is the big problem for Breadth-First Search; it grows exponentially with depth!

Romania with step costs in km

Breadth First Search assumes that all steps have equal cost.

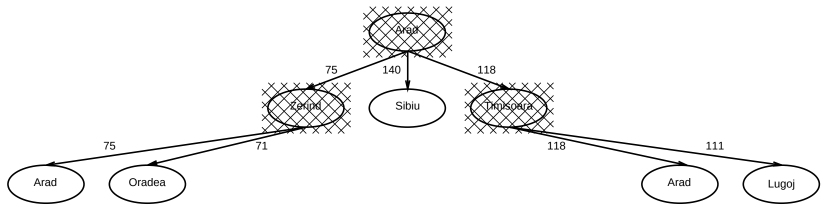


However, we are often looking for the path with the shortest total distance rather than the number of steps.

Uniform-Cost Search

- Expand root first, then expand least-cost unexpanded node
- Implementation: QUEUEINGFUNCTION = insert nodes in order of 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 (i.e. no negative-cost steps)

Uniform-Cost Search



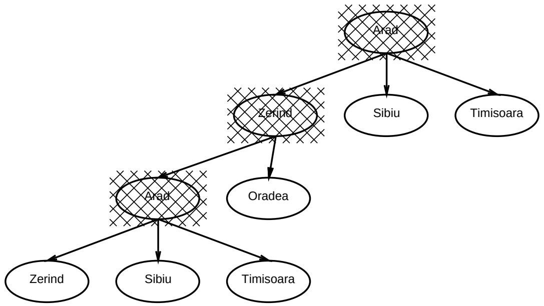
Properties of Uniform-Cost Search

- Complete? Yes, if b is finite and step cost $\geq \varepsilon$ with $\varepsilon > 0$
- Time: $O(b^{\lceil C^*/\varepsilon \rceil})$ where C^* = cost of optimal solution, and assume every action costs at least ε
- Space: $O(b^{\lceil C^*/\varepsilon \rceil})$ $(b^{\lceil C^*/\varepsilon \rceil} = b^d$ if all step costs are equal)
- Optimal? Yes.

Depth First Search

- Expands one of the nodes at the deepest level of the tree
- Implementation:
  - QUEUEINGFUNCTION = insert newly generated states at the front of the queue (thus making it a stack)
  - can alternatively be implemented by recursive function calls

Depth First Search



©Alan Blair, 2013-18

Properties of Depth First Search

- Complete? No! fails in infinite-depth spaces, spaces with loops; modify to avoid repeated states along path ⇒ complete in finite spaces
- Time: $O(b^m)$ (terrible if m is much larger than d but if solutions are dense, may be much faster than breadth-first)
- Space: $O(bm)$, i.e. linear space!
- Optimal? No, can find suboptimal solutions first.

©Alan Blair, 2013-18

Depth Limited Search

Expands nodes like Depth First Search but imposes a cutoff on the maximum depth of path.

- Complete? Yes (no infinite loops anymore)
- Time: $O(b^k)$, where k is the depth limit
- Space: $O(bk)$, i.e. linear space similar to DFS
- Optimal? No, can find suboptimal solutions first

Problem: How to pick a good limit ?

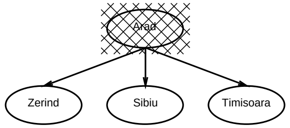
©Alan Blair, 2013-18

Iterative Deepening Search

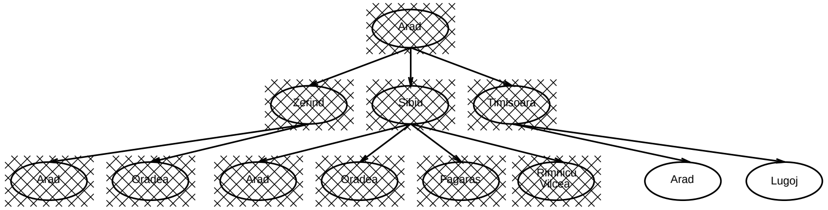
- Tries to combine the benefits of depth-first (low memory) and breadth-first (optimal and complete) by doing a series of depth-limited searches to depth 1, 2, 3, etc.
- Early states will be expanded multiple times, but that might not matter too much because most of the nodes are near the leaves.

©Alan Blair, 2013-18

Iterative Deepening Search



Iterative Deepening Search



Properties of Iterative Deepening Search

- Complete? Yes.
- Time: nodes at the bottom level are expanded once, nodes at the next level twice, and so on:
  - depth-limited: $1+b^1+b^2+\ldots+b^{d-1}+b^d=O(b^d)$
  - iterative deepening:

$$(d+1)b^0+db^1+(d-1)b^2+\ldots+2\cdot b^{d-1}+1\cdot b^d=O(b^d)$$

(We assume $b>1$)

Properties of Iterative Deepening Search

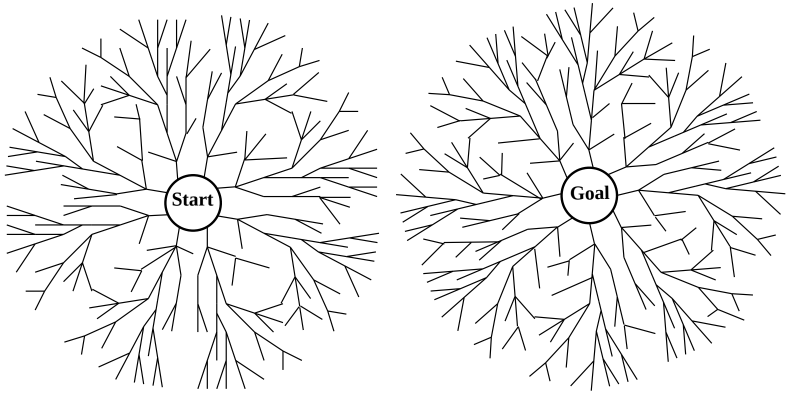
- Complete? Yes.
- Time: nodes at the bottom level are expanded once, nodes at the next level twice, and so on:
  - depth-limited: $1+b^1+b^2+\ldots+b^{d-1}+b^d=O(b^d)$
  - iterative deepening: $(d+1)b^0+db^1+(d-1)b^2+\ldots+2\cdot b^{d-1}+1\cdot b^d=O(b^d)$
  - example b = 10, d = 5 :
    - depth-limited: 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
    - iterative-deepening: 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
    - only about 11% more nodes (for b = 10).

Properties of Iterative Deepening Search

- Complete? Yes
- Time: $O(b^d)$
- Space: $O(bd)$
- Optimal? Yes, if step costs are identical.

 ©Alan Blair, 2013-18

Bidirectional Search



 ©Alan Blair, 2013-18

Bidirectional Search

- Idea: Search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle.
- We need an efficient way to check if a new node already appears in the other half of the search. The complexity analysis assumes this can be done in constant time, using a Hash Table.
- Assume branching factor = b in both directions and that there is a solution at depth = d. Then bidirectional search finds a solution in $O(2b^{d/2}) = O(b^{d/2})$ time steps.

 ©Alan Blair, 2013-18

Bidirectional Search – Issues

- searching backwards means generating predecessors starting from the goal, which may be difficult
- there can be several goals – e.g. chekmate positions in chess
- space complexity: $O(b^{d/2})$ because the nodes of at least one half must be kept in memory.

 ©Alan Blair, 2013-18

Summary

- problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- variety of Uninformed search strategies
- Iterative Deepening Search uses only linear space and not much more time than other Uninformed algorithms.

©Alan Blair, 2013-18

Complexity Results for Uninformed Search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Time	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^k)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bk)$	$O(bd)$
Complete?	Yes[1]	Yes[2]	No	No	Yes[1]
Optimal ?	Yes[3]	Yes	No	No	Yes[3]

b = branching factor, d = depth of the shallowest solution,
m = maximum depth of the search tree, k = depth limit.

1 = complete if b is finite.

2 = complete if b is finite and step costs $\geq \varepsilon$ with $\varepsilon > 0$.

3 = optimal if actions all have the same cost.

©Alan Blair, 2013-18