Solving Problems by Searching

- Chapter 3 of the textbook

Problem Solving Agents

- A problem solving agent is one which decides what actions and states to consider in completing a goal
- Examples:
  - Finding the shortest path from Arad to Bucharest
  - 8-puzzle

Example: Romania

Problem Statement

- How do I get from initial state to goal?
  - Use a search algorithm
  - From initial state, apply operators (actions) to the current state to generate new states: expand
    - find all adjacent cities
  - Choose one of the adjacent states, and expand the state from there
  - Continue until you find solution
  - Search strategy: Which states do you expand first?

Example: The 8-puzzle

- Finding shortest path
  - Action: Move from city X to city Y
  - State: Which city you’re on
  - Goal Test: Am I in Bucharest?
  - Cost: 1 for each city I visit

- 8-Puzzle
  - Action: Move blank square up, down, left, or right
  - State: arrangement of squares including blank
  - Goal Test: Is it in correct order?
  - Cost: 1 for each move it took
For 8-puzzle, there is an implicit graph: each node is a configuration of tiles diagram, a move links two nodes.

Distinction between state and node
- State: configuration of the world
  - What city am I in?
  - What is the arrangement of tiles in the puzzle?
- Node: data structure within the search tree.
  - Contains:
    - state
    - parent node
    - operator used to generate node
    - depth of this node
    - path cost from root node to this node

Uninformed Search Strategies
- Uninformed: We have no knowledge of the world to help guide our search
  - How could knowledge help? (Rubik's cube?)
  - Informed search uses heuristics
- Interested in following:
  - Completeness: Solution guaranteed if it exists
  - Time complexity: How long?
  - Space complexity: How much memory?
  - Optimality: Does it find the best solution? Does it find it first?

Uninformed search strategies
- Uninformed (blind) search strategies use only the information available in the problem definition:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

Breadth-first search
- Overview
  - Expand root node
  - Expand all children of root node
  - Expand all grandchildren, etc.
- In general
  - Expand all nodes at depth \( d \) before expanding nodes at depth \( d+1 \)
Breadth-First Analysis
- Completeness: Solution is guaranteed
- Optimality:
  - Finds shallowest state first
  - Best state first if:
    - depth of node = path cost; OR
    - cost = g(depth), where g is nondecreasing
- Time and space?
  - Let $b =$ branching factor: maximum number of branches a given node can yield
  - What is branching factor for map?
  - What is branching factor for 8-puzzle?

Breadth-First Search
- Time complexity: Worst case is
  \[ 1 + b + b^2 + b^3 + b^4 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} \]
  - where $d$ is the depth of the tree
  - $O(b^d)$
- Space complexity: $O(b^d)$
  - Need to maintain all nodes on most recent level
  - Usually implemented with a queue

Uniform Cost Search
- Completeness: Solution is guaranteed
- Same complexity in worst case as for Breadth-First
- Optimality
  - If path cost never decreases, will stop at optimal solution
  - Does not necessarily find best solution first
  - Let $g(n) =$ path cost at node $n$: need
    \[ g(\text{child}(n)) \geq g(n) \]
**Depth-First Search**

- Expand root node
- Expand node at deepest level of tree
- Repeat

**Warning:** There are two implementations which produce different results
  - Recursive one
  - Using a stack

**Recursive DFS**

```java
Boolean recDFS(node X) {
    markAsFound(X);
    if isGoal(X) return true
    for each node Y in myNeighbors(X)
        if (not found(Y))
            if (recDFS(Y)) return true ;
    // markAsFinished(X);
    return false;
}
```

**Depth-First Search**

- **Space complexity:**
  - Must store all nodes on current path
  - Must store all unexplored sibling nodes on path
  - At depth \( m \), required to store \( bm \) nodes
  - \( O(b^d) \): Much better than \( O(b^d) \)
  - **Time complexity:**
    - Still need to explore all nodes: \( O(b^d) \)
    - Depth-first can get lucky and find long path quickly
    - Depth-first can get “lost” down a really long path

- **Complete**
  - No – if tree is infinite, could spend forever exploring one branch

- **Optimality**
  - Might never find any solutions
Depth-Limited Search

- Depth-First search, but limit maximum depth allowed
  - Map example: limit maximum depth to number of cities
- Complete:
  - only if depth limit (L) is large enough
- Optimality:
  - if L large enough to include best soln, will find it
  - will not necessarily find it first
- Time: $O(b^L)$
- Space: $O(bL)$

Iterative Deepening Search

- Depth-limited search, with
  - depth = 0
  - then again with depth = 1
  - then again with depth = 2
  - ... until you find a solution
Iterative Deepening Search

- Why iterative deepening search?
  - Complete: eventually will find solution
- Why not use BFS?
  - BFS traverses in same order, apart from repeats.
  - Aren’t repeats inefficient?

Memory requirements are same as those as DFS: $O(bd)$ instead of $O(b^d)$

Can think of it as BFS where store less info, and rediscover it when you need it

Completeness and optimality the same as for BFS

How much time do you lose due to repeats?

It’s not so bad, since you don’t repeat the bottom levels as much (the big ones)

Iterative Deepening Search: Time Analysis

- Number of depth-first search expansions:
  $$1 + b + b^2 + b^3 + b^4 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$
- Number of iterative deepening expansions:
  Analysis on blackboard

Bidirectional Search

- Start searching forward from initial state and backwards from goal, and try to meet in the middle
- Should reduce from $O(b^d)$ to $O(2b^{d/2}) = O(b^{d/2})$
- All sorts of questions and issues, see text

Avoiding Repeated States

- How do you make sure you don’t cycle?
- Need to store all the states you have already been to: lots of memory! $O(b^d)$
- Checking would only pay off if space has lots of cycles
- Hash table usually used to make lookup efficient

Simultaneously:

- Search forward from start
- Search backward from the goal
- Stop when the two searches meet

If branching factor = b in each direction

- with solution at depth d
- $\Rightarrow$ only $O(2b^{d/2}) = O(2b^{d/2})$

Checking a node for membership in the other search tree can be done in constant time (hash table)

Key limitations:

- Space $O(b^{d/2})$
- Also, how to search backwards can be an issue (e.g., in Chess)?
- What’s tricky?
- Problem: lots of states satisfy the goal; don’t know which one is relevant.
Repeated states

- Failure to detect repeated states can turn linear problem into an exponential one!
- Problems in which actions are reversible (e.g., routing problems or sliding-blocks puzzle). Also, in eg Chess; uses hash tables to check for repeated states. Huge tables 100M+ size but very useful.

Example: The 8-puzzle "sliding tile puzzle"

- states? the boards, i.e., locations of tiles
- actions? move blank left, right, up, down
- goal test? goal state (given; tiles in order)
- path cost? 1 per move

Note: finding optimal solution of n-puzzle family is NP-hard!
- Also, from certain states you can't reach the goal.
- Total number of states 9! = 362,880 (more interesting space; not all connected... only half can reach goal state)

Aside: variations on goal state. eg empty square bottom right or in middle.

Example: The 15-puzzle Goal state

- Search space: 16!/2 = 1.0461395 e+13, about 10 trillion.
- Too large to store in RAM (≈ 100 TB). A challenge to search for a path from a given board to goal.
- Longest minimum path: 80 moves. Just 17 boards, e.g.

- Average minimum soln. length: 53.
- People can find solns. But not necessarily minimum length.

See Fifteen Puzzle Optimal Solver.
http://kociemba.org/fifteen/fifteensolver.html
With effective search: optimal solutions in seconds!
Average: milliseconds.
Longest minimum path: 80 moves. Just 17 boards, e.g.

17 boards farthest away from goal state (80 moves)

What is it about these 17 boards out of over 10 trillion?

- Each require 80 moves to reach:
  - Intriguing similarities. Each number has its own few locations.
- Interesting machine learning task:
  - Learn to recognize the hardest boards!

17 boards farthest away from goal state (80 moves)

Most regular extreme case:

- Each quadrant reflected along diagonal. "move tiles furthest away"

Thanks to Jonathan GS

Summary: General, uninformed search

- Original search ideas in AI where inspired by studies of human problem solving, eg, puzzles, math, and games, but a great many AI tasks now require some form of search (e.g., find optimal agent strategy; active learning).
- 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 double time than other uninformed algorithms.
- Avoid repeating states / cycles.
Play the fifteen puzzle on-line
http://migo.sixbit.org/puzzles/fifteen/

Let’s consider the search for a solution.

Searching for a solution
to the 8-puzzle.

Aside: in this tree, immediate duplicates are removed.

A breadth-first search tree. (More detail soon.)

Branching factor 1, 2, or 3 (max). So, approx. 2 --- # nodes roughly doubles at each level.
Number states of explored nodes grows exponentially with depth.

For 15-puzzle, hard initial states: 80 levels deep, requires exploring approx. $2^{80} \approx 10^{24}$ states.

If we block all duplicates, we get closer to 10 trillion ($10^{13}$, the number of distinct states: still a lot!).
Really only barely feasible on compute cluster with lots of memory and compute time. (Raw numbers for 24 puzzle, truly infeasible.)
Can we avoid generating all these boards? Do with much less search?
(Key: bring average branching factor down.)

For 15-puzzle, hard initial states: 80 levels deep, requires exploring approx. $2^{80} \approx 10^{24}$ states.

If we block all duplicates, we get closer to 10 trillion ($10^{13}$, the number of distinct states: still a lot!).
Really only barely feasible on compute cluster with lots of memory and compute time. (Raw numbers for 24 puzzle, truly infeasible.)
Can we avoid generating all these boards? Do with much less search?
(Key: bring average branching factor down.)

Gedanken experiment

Assume that you knew for each state, the minimum number of moves to the final goal state. (Table too big, but assume there is some formula/algorithim based on the board pattern that gives this number for each board and quickly.)

Using the minimum distance information, is there a clever way to find a minimum length sequence of moves leading from the start state to the goal state? How?

Hmm. How do I know?

Note: at least one neighbor with $d = 4$.

$d = \text{min dist. to goal}$

A breadth-first search tree. (More detail soon.)

$d = \text{min dist. to goal}$

Branching factor approx. 2. So, with “distance oracle” we only need to explore approx. $2^4$ (min. solution length).

For 15-puzzle, hard initial states: 80 levels deep, requires exploring approx. $2^{80} \approx 10^{24}$ states.
But, with distance oracle, we only need to explore roughly 80 --- $2 \times 160$ states (only linear in size of solution length).

We may not have the exact distance function (“perfect heuristics”), but we can still “guide” the search using an approximate distance function.

This is the key idea behind “heuristic search” or “knowledge-based search.”

We use knowledge / heuristic information about the distance to the goal to guide our search process. We can go from exponential to polynomial or even linear complexity. More common: brings exponent down significantly.

E.g. from $2^L$ to $2^{L/100}$.

The measure we considered would be the “perfect” heuristic. Eliminates tree search! Find the right “path” to goal state immediately.
Basic idea: State evaluation function can effectively guide search.
Also in multi-agent settings. (Chess: board eval.)
Reinforcement learning: Learn the state eval function.

A breadth-first search tree. (Chess board)

Perfect "heuristics," eliminates search.
Approximate heuristics, significantly reduces search.
Best (provably) use of search heuristic info: A* search (soon).