Problem Solving

- Problem solving agent: Select a sequence of actions to achieve a goal
  - Moves to solve a Rubik’s cube
  - Find a route from USF to SFO
  - Arrange components on a chip
05-1: Problem Solving
The process of sequentially considering actions in order to find a sequence of actions that lead from start to goal is called *search*.

A search algorithm returns an action sequence that is then executed by the agent.

- Search typically happens “offline.”

Note: this assumes the environment is static.

Also, environment is assumed to be discrete.

Environment is (usually) considered to be deterministic.
Some classic search problems

- Toy problems: useful to study as examples or to compare algorithms
  - 8-puzzle
  - Vacuum world
  - Rubik’s cube
  - N-queens
- Real-world problems: typically more messy, but the answer is actually interesting
  - Route finding
  - Traveling salesman
  - VLSI layout
  - Searching the Internet
We’ll often talk about the *state* an agent is in. This refers to the values of relevant variables describing the environment and agent.

- Vacuum World: (A, ’clean’, ’dirty’)
- Romania: t = 0, in(Bucharest)
- Rubik’s cube: current arrangement of the cube.

This is an *abstraction* of our problem.

Focus only on the details relevant to the problem.
05-5: Formulating a Search Problem

- Initial State
- Goal Test
- Actions
- Successor Function
- Path cost
- Goal / Goal Test
05-6: Initial State

- Initial State: The state that the agent starts in.
  - Vacuum cleaner world: (A, 'dirty, 'dirty')
  - Romania: In(Arad)
05-7: **Actions**

- **Actions**: What actions is the agent able to take?
  - Vacuum: Left, Right, Suck, Noop
  - Romania: Go(adj. city)
Successor Function

• Successor function: for a given state, returns a set of action/new-state pairs.
  • This tells us, for a given state, what actions we’re allowed to take and where they’ll lead.
• In a deterministic world, each action will be paired with a single state.
  • Vacuum-cleaner world: \((A, \text{dirty}, \text{clean}) \rightarrow ('\text{Left}', (A, \text{dirty}, \text{clean})), ('\text{Right}', (B, \text{dirty}, \text{clean})), ('\text{Suck}', (A, \text{clean}, \text{dirty})), ('\text{NoOp}, (A, \text{dirty}, \text{clean}))\)
  • Romania: \(\text{In}(\text{Arad}) \rightarrow ((\text{Go}(\text{Timisoara}), \text{In}(\text{Timisoara})), (\text{Go}(\text{Sibiu}), \text{In}(\text{Sibiu})), (\text{Go}(\text{Zerind}), \text{In}(\text{Zerind})))\)
Successor Function

- Successor function: for a given state, returns a set of action/new-state pairs.
  - This tells us, for a given state, what actions we’re allowed to take and where they’ll lead.
- In a deterministic world, each action will be paired with a single state.
- In stochastic worlds an action may be paired with many states (potentially with probabilities)
Goal Test

- Goal test: This determines if a given state is a goal state.
  - There may be a unique goal state, or many.
  - Vacuum World: every room clean.
  - Chess - checkmate
  - Romania: in(Bucharest)
The combination of problem states (arrangements of variables of interest) and successor functions (ways to reach states) leads to the notion of a state space.

This is a graph representing all the possible world states, and the transitions between them.

Finding a solution to a search problem is reduced to finding a path from the start state to the goal state.
State space for simple vacuum cleaner world
05-13: Types of Solutions

- Depending on the problem, we might want different sorts of solutions
  - Any path to solution
  - Optimal path to solution
  - Goal state itself (n-queens)

- We’ll often talk about the size of these spaces as a measure of problem difficulty.
  - 8-puzzle: \( \frac{9!}{2} = 181,000 \) states (easy)
  - 15-puzzle: \( \sim 1.3 \) trillion states (pretty easy)
  - 24-puzzle: \( \sim 10^{25} \) states (hard)
  - TSP, 20 cities: \( 20! = 2.43 \times 10^{18} \) states (hard)
05-14: **Path cost**

- The *path cost* is the cost an agent must incur to go from the initial state to the currently-examined state.
- Often, this is the sum of the cost for each action:
  - This is called the *step cost*
- We’ll assume that step costs are nonnegative.
  - What if they could be negative?
05-15: Examples

• What are the states/operators/path cost for the following:
  • Sliding tile puzzle
  • Rubic’s cube
  • 8-Queens puzzle
8-Queens puzzle
  - Incremental: Place queens one by one
    - States: Arrangement of 0-8 Queens
    - Operators: Add a queen to the board somewhere
  - States: Arrangement of 0-8 Queens, no attacks
    - Operators: Place a queen in leftmost empty column, no attacks
  - What if you get stuck?
05-17: Examples

- 8-Queens puzzle
  - Complete: Place all queens, move
    - States: Arrangement of 8 Queens on board
    - Operators: Move any attacked queen to another square
  - States: Arrangement of 8 Queens on board, one in each column
  - Operators: Move any queen to another square in the same column

- Can’t get stuck
You’ve probably seen other algorithms for solving path-finding problems on a graph:
- Djikstra’s algorithm, Prim’s algorithm, Max-flow, All-pairs shortest-path
- These algorithms are quadratic or cubic in the number of vertices.
- We’ll talk about search being exponential in the number of state variables.
- Is this a contradiction?
Most search problems are too large to hold in memory
  - We need to dynamically instantiate portions of the search space
We construct a search tree by starting at the initial state and repeatedly applying the successor function.
Basic idea: from a state, consider what can be done. Then consider what can be done from each of those states.
Some questions we’ll be interested in:

- Are we guaranteed to find a solution?
- Are we guaranteed to find the optimal solution?
- How long will the search take?
- How much space will it require?
05-21: Example Search Tree

- The beginnings of a Romania search tree:
  
  (a) The initial state

  (b) After expanding Arad

  (c) After expanding Sibiu
The basic search algorithm is surprisingly simple:

\[
\text{fringe} \leftarrow \text{initialState}
\]
\[
do
\quad \text{select node from fringe}
\quad \text{if node is not goal}
\quad \text{generated successors of node}
\quad \text{add successors to fringe}
\]

- We call this list of nodes generated but not yet expanded the \textit{fringe}.
- Question: How do we select a node from the fringe?
  - Differentiates search algorithms
The simplest sort of search algorithms are those that use no additional information beyond what is in the problem description.

We call this *uninformed search*. Sometimes these are called weak methods.

If we have additional information about how promising a nongoal state is, we can perform *heuristic search*. 
Breadth-first search

- Breadth-first search works by expanding a node, then expanding each of its children, then each of their children, etc.
- All nodes at depth $n$ are visited before a node at depth $n + 1$ is visited.
- We can implement BFS using a queue.
05-25: Breadth-first search

- BFS Python-ish code

```python
queue.enqueue(initialState)
while not done :
    node = queue.dequeue()
    if goalTest(node) :
        return node
    else :
        children = successor-fn(node)
        for child in children
            queue.enqueue(child)
```
BFS example: Arad to Bucharest

- dequeue Arad
- enqueue Sibiu, Timisoara, Zerind
- dequeue and test Sibiu
- enqueue Oradea, Fagaras, Rimniciu Viclea
- dequeue and test Timisoara
- enqueue Lugoj
- ...

...
Some subtle points

• How do we avoid revisiting Arad?
  • Closed-list: keep a list of expanded states.
• How do we avoid inserting Oradea twice?
  • Open-list (our queue, actually): a list of generated but unexpanded states.
• Why don’t we apply the goal test when we generate children?
  • Not really any different. Nodes are visited and tested in the same order either way. Same number of goal tests are performed.
05-28: Analyzing BFS

- Completeness: Is BFS guaranteed to find a solution?
- Optimality: If there are multiple solutions, will BFS find the best one?
- Time complexity: How long does BFS take to run, as a function of solution length?
- Space Complexity: How much memory does BFS require, as a function of solution length?
05-29: Analyzing BFS

• Completeness: Is BFS guaranteed to find a solution?
  • Yes. Assume the solution is at depth $n$. Since all nodes at or above $n$ are visited before anything at $n + 1$, a solution will be found.

• Optimality: If there are multiple solutions, will BFS find the best one?
  • BFS will find the shallowest solution in the search tree. If step costs are uniform, this will be optimal. Otherwise, not necessarily.
  • Arad -> Sibiu -> Fagaras -> Bucharest will be found first. (dist = 450)
  • Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest is shorter. (dist = 418)
05-30: Analyzing BFS

- Time complexity: BFS will require $O(b^{d+1})$ running time.
  - $b$ is the branching factor: average number of children
  - $d$ is the depth of the solution.
  - BFS will visit
    $$b + b^2 + b^3 + \ldots + b^d + b^{d+1} - (b - 1) = O(b^{d+1})$$
    nodes
- Space complexity: BFS must keep the whole search tree in memory (since we want to know the sequence of actions to get to the goal).
- This is also $O(b^{d+1})$. 
Analyzing BFS

- Assume $b = 10$, 1kb/node, 10000 nodes/sec
- depth 2: 1100 nodes, 0.11 seconds, 1 megabyte
- depth 4: 111,000 nodes, 11 seconds, 106 megabytes
- depth 6: $10^7$ nodes, 19 minutes, 10 gigabytes
- depth 8: $10^9$ nodes, 31 hours, 1 terabyte
- depth 10: $10^{11}$ nodes, 129 days, 101 terabytes
- depth 12: $10^{13}$ nodes, 35 years, 10 petabytes
- depth 14: $10^{15}$ nodes, 3523 years, 1 exabyte
- In general, the space requirements of BFS are a bigger problem than the time requirements.
Recall that BFS is nonoptimal when step costs are nonuniform. We can correct this by expanding the shortest paths first. Add a path cost to expanded nodes. Use a priority queue to order them in order of increasing path cost. Guaranteed to find the shortest path. If step costs are uniform, this is identical to BFS. This is how Djikstra’s algorithm works.
Depth-first Search

- Depth-first search takes the opposite approach to search from BFS.
  - Always expand the deepest node.
- Expand a child, then expand its left-most child, and so on.
- We can implement DFS using a stack.
Depth-first Search

- DFS python-ish code:

```python
stack.push(initialState)
while not done :
    node = pop()
    if goalTest(node) :
        return node
    else :
        children = successor-fn(node)
        for child in children :
            stack.push(child)
```
DFS example: Arad to Bucharest

- pop Arad
- push Sibiu, Timisoara, Zerind
- pop and test Sibiu
- push Oradea, Fagaras, Rimniciu Viclea
- pop and test Oradea
- pop and test Fagaras
- push Bucharest
- ...

05-35:
Analyzing DFS

- Completeness
- Optimality
- Time requirement
- Space requirement
05-37: Analyzing DFS

- Completeness: no. We can potentially wander down an infinitely long path that does not lead to a solution.
- Optimality: no. We might find a solution at depth $n$ under one child without ever seeing a shorter solution under another child. (what if we popped Rimnciu Viclea first?)
- Time requirements: $O(b^m)$, where $m$ is the maximum depth of the tree.
  - $m$ may be much larger than $d$ (the solution depth)
  - In some cases, $m$ may be infinite.
Analyzing DFS

- Space requirements: $O(bm)$
  - We only need to store the currently-searched branch.
  - This is DFS’ strong point.
  - In our previous figure, searching to depth 12 would require 118 KB, rather than 10 petabytes for BFS.
A Search problem consists of:

- A description of the states
- An initial state
- A goal test
- Actions to be taken
- A successor function
- A path cost
First, a question

- Why are we looking at algorithms that perform an exhaustive search? Isn’t there something faster?
- Many of the problems we’re interested in are NP-complete.
  - No known polynomial-time algorithm
  - Worse, many are also inapproximable.
- In the worst case, the best one can hope for is to enumerate all solutions.
Avoiding Infinite Search

- There are several approaches to avoiding DFS’ infinite search.
- Closed-list
  - May not always help.
  - Now we have to keep exponentially many nodes in memory.
- Depth-limited search
- Iterative deepening DFS
Depth-limited Search

- Depth-limited search works by giving DFS an upper limit $l$.
- Search stops at this depth.
- Solves the problem of infinite search down one branch.
- Adds another potential problem
  - What if the solution is deeper than $l$?
  - How do we pick a reasonable $l$?
- In the Romania problem, we know there are 20 cities, so $l = 19$ is a reasonable choice.
- What about 8-puzzle?
05-43: Depth-limited Search

- DLS pseudocode

```python
stack.push(initialState)
while not done :
    node = pop()
    if goalTest(node) :
        return node
    else :
        if depth(node) < limit :
            children = successor-fn(node)
            for child in children
                push(child)
        else :
            return None
```
Iterative Deepening DFS (IDS)

- Expand on the idea of depth-limited search.
- Do DLS with $l = 1$, then $l = 2$, then $l = 3$, etc.
- Eventually, $l = d$, the depth of the goal.
  - This means that IDS is complete.
- Drawback: Some nodes are generated and expanded multiple times.
Iterative Deepening DFS (IDS)

- Due to the exponential growth of the tree, this is not as much of a problem as we might think.
  - Level 1: \(b\) nodes generated \(d\) times
  - Level 2: \(b^2\) nodes generated \(d - 1\) times
  - ...
  - Level \(d\): \(b^d\) nodes generated once.
  - Total running time: \(O(b^d)\). Slightly more nodes generated than BFS.
  - Still has linear memory requirements.
IDS pseudocode:

d = 0
while True:
    result = depth-limited-search(d)
    if result == goal
        return result
    else
        d = d + 1
IDS is actually similar to BFS in that all nodes at depth $n$ are examined before any node at depth $n + 1$ is examined.

As with BFS, we can get optimality in non-uniform step cost worlds by expanding according to path cost, rather than depth.

This is called *iterative lengthening search*.

Search all paths with cost less than $p$. Increase $p$ by $\delta$.

In continuous worlds, what should $\delta$ be?
05-48: Constraint Satisfaction

- Set of variables & constraints
  - 8-Queens
  - Map Coloring
  - Crossword Puzzles
- Assign values to variables to satisfy all constraints
- How can we define this as a search problem?
05-49: **Constraint Satisfaction**

- Pick an ordering of the variables
- While not all values have been chosen
  - Assign a value to the next variable, consistent with all previous values
- If no value is consistent, back up
- Variant of DFS, backtracking
05-50: Backtracking

• What happens when DFS and its cousins reach a failure state?
• They go up to the parent and try the next sibling.
• Assumption: The most recently-chosen action is the one that caused the failure.
  • This is called chronological backtracking - undo the most recent thing you did.
• This can be a problem - failure may be a result of a previous decision.
  • Example: 4-queens, map coloring
Constraints can help you limit the size of the search space.

Intelligent backtracking tries to analyze the reason for the failure and unwind the search to that point.

- Can unwind to the most recent conflicting variable (backjumping)
- Can also do forward checking - is there a possible assignment of values to variables at this point?
Backtracking is not just in CSPs

Bridge problem
- 5 people to cross a bridge
- Takes time 1, 2, 5, 10 minutes
- Time bound: 17 minutes
Bidirectional Search

- Search forward from initial state, and backwards from goal
- Find solution when fringes meet
  - Advantages?
  - Disadvantages?
Summary

• Formalizing a search problem
  • Initial State
  • Goal Test
  • Actions to be taken
  • Successor function
  • Path cost
• Leads to search through a state space using a search tree.
Summary

• Algorithms
  • Breadth First Search
  • Depth First Search
  • Uniform Cost Search
  • Depth-limited Search
  • Iterative Deepening Search