Lecture 2: Uninformed Search
(Russell and Norvig Chapter 3)

Hey guess what you’re learning pseudoscience.
Correct URL of Class Website:
www.andrew.cmu.edu/course/15-381-f09/index.html
What happens when computers become as intelligent as humans?

Are they going to kill us? Will they be slaves? That’s what we’re working towards.
Lecture 2: Uninformed Search
A Search Problem

Find a path from START to GOAL
OR
Find the minimum number of transitions

Many problems can be encoded this way
How to encode as search problem?
Make a graph of possible states.
State: config of puzzle
Transition: up to 4 poss moves from each state
Solvable in 22 steps on average
But state space is really big. 2*1^5.
Example

State: Configuration of the puzzle

Transitions: Up to 4 possible moves from each state (up, down, left, right)

Solvable in 22 steps on average

But: $1.8 \times 10^5$ states ($1.3 \times 10^{12}$ states for the 15-puzzle)

Cannot represent a set of states explicitly
Allowed motions: You can move up, down, left, or right. You can’t move through walls.

Example: Robot Navigation

States = positions in the map

Transitions = allowed motions

Navigation: Going from point START to point GOAL given a (deterministic) map
In the real world the graph is really really big. V large # states
10cm resolution
$4 \text{km}^2 = 4 \times 10^8 \text{ states}$
What We are Not Addressing (yet)

• Uncertainty/Chance: State and transitions are known and deterministic

• Game against an adversary

• Multiple agents and cooperation

• Continuous state space: For now, the set of states is discrete
A Search Problem

START

b

a

c

d

e

f

h

GOAL

p

q

r
Formulation

Q: Finite set of states
S ⊆ Q: Non-empty set of start states
G ⊆ Q: Non-empty set of goal states
succs: Function Q → ℱ(Q)
succs(s) = Set of states that can be reached from s in one step
cost: function Q × Q → Positive Numbers
cost(s, s') = Cost of taking a one-step transition from s to s'

Problem: Find a sequence \{s_1, \ldots, s_K\} such that:

s_1 ∈ S
s_K ∈ G
s_{i+1} ∈ succs(s_i)
Σ cost(s_i, s_{i+1}) is the smallest among all possible sequences (desirable but optional)
Example

\[ Q = \{\text{START}, \text{GOAL}, a, b, c, d, e, f, h, p, q, r\} \]
\[ S = \{\text{START}\}, G = \{\text{GOAL}\} \]
\[ \text{succs}(d) = \{b, c\}, \text{succs}(\text{START}) = \{p, e, d\}, \text{succs}(a) = \text{NULL} \]
\[ \text{cost}(s, s') = 1 \text{ for all transitions} \]
Desirable Properties

Completeness: An algorithm is complete if it is guaranteed to find a path if one exists
Optimality: The total cost of the path is the lowest among all possible paths from start to goal
Time and Space Complexity

Keep in mind storing graph in mem is not an option.
Breadth-First Search

Label all states that are 0 steps from S. Call that set $V_o$
Breadth-First Search

0 steps
1 step

Start

Label the successors of the states in $V_o$ that are not yet labeled as the set $V_1$ of states that are 1 step away from the start.
Label the successors of the states in $V_1$ that are not yet labeled as the set $V_2$ of states that are 2 steps away from the start.
Label the successors of the states in $V_2$ that are not yet labeled as the set $V_3$ of states that are 3 steps away from the start.
Breadth-First Search

Stop when goal is reached in the current expansion set. In this case goal can be reached in 4 steps.
Record the predecessor when labeling a state.
When labeling GOAL, I was expanding the neighbors of f, so f is the predecessor of GOAL.
Final solution: {START, e, r, f, GOAL}
A backpointer previous(s) points to the node that stored the state that was expanded to label s
The path is recovered by following the backpointers starting at the goal state

The “right” way to implement
Example: Robot Navigation

States = positions in the map

Transitions = allowed motions

Navigation: Going from point START to point GOAL given a (deterministic) map
Breadth-First Search

\[ V_0 = S \] (the set of start states)
\[ \text{previous}(\text{START}) = \text{NULL} \]
\[ k = 0 \]

**while** (no goal state is in \( V_k \) and \( V_k \) is not empty):
  \[ V_{k+1} = \text{empty set} \]
  For each state \( s \) in \( V_k \)
    For each state \( s' \) in \( \text{succs}(s) \)
      If \( s' \) has not already been labeled
        Set \( \text{previous}(s') = s \); add \( s' \) into \( V_{k+1} \)
  \[ k = k + 1 \]

**if** \( V_k \) is empty output \text{FAILURE}
**else** build the solution path thus:
  Define \( S_k = \text{GOAL} \), and for all \( i \leq k \), define \( S_{i-1} = \text{previous}(S_i) \)
  Return path = \( \{S_1, ..., S_k\} \)
Properties

- BFS can handle multiple start and goal states
- Can work either by searching forward from the start or backward for the goal (forward/backward chaining)
- (Which way is better?)
- Guaranteed to find the lowest-cost path in terms of number of transitions?

Which is better? Depends on graph. For example, for a dense tree with start at root and goal at leaf then backwards is better. Others may vary.
Guaranteed lowest cost IF all edges have uniform, nonnegative cost.
### Complexity

$B$ = Average number of successors (branching factor)

$L$ = Length from start to goal on shortest path

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Complexity

\( B = \text{Average number of successors (branching factor)} \)

\( L = \text{Length from start to goal on shortest path} \)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td>Y</td>
<td>Y, If all trans. have same cost</td>
<td>( O(B^L) )</td>
</tr>
</tbody>
</table>
Bidirectional Search

- BFS search simultaneously forward from START and backward from GOAL
- What’s the stopping criterion?
- Under what condition is it optimal?

Stopping: when intersect($V', V$) nonempty
Optimal if all costs uniform
# Complexity

\( B = \text{Average number of successors (branching factor)} \)
\( L = \text{Length from start to goal on shortest path} \)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>( O(B^L) )</td>
<td>( O(B^L) )</td>
</tr>
<tr>
<td>BIBFS</td>
<td>Bi-directional BFS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complexity

\[ B = \text{Average number of successors (branching factor)} \]
\[ L = \text{Length from start to goal on shortest path} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>( O(B^L) )</td>
</tr>
<tr>
<td>BIBFS</td>
<td>Bi-directional BFS</td>
<td>Y</td>
<td>Y, if all tran. have same cost</td>
<td>( O(2B^{L/2}) )</td>
</tr>
</tbody>
</table>

**Major savings when bidirectional search is possible because**  
\[ 2B^{L/2} \ll B^L \]

**If** \( B = 10 \), \( L = 6 \), **then 22,200 states generated vs \( \sim 10^7 \)**

Note: Going backwards isn’t always possible
Counting Transition Costs

BFS finds the shortest path in number of steps but does not take into account transition costs

New field to find least cost path: At iteration $k$, $g(s) =$ least cost of path to $s$ in $k$ or fewer steps
Uniform Cost Search

- Strategy to select state to expand next: use the state with the smallest value of $g(\ )$ so far
- Use priority queue for efficient access to minimum $g$ at every iteration
Complexity depends on the implementation of the PQ. This complexity is for a min heap implementation. For a survey of different PQ implementations see http://www.theturingmachine.com/algorithms/heaps.html
Uniform Cost Search

\( PQ = \text{Current set of evaluated states} \)

Value (priority) of state = \( g(s) = \text{current cost of path to } s \)

Basic iteration:

1. Pop the state \( s \) with the cost from \( PQ \)
2. Evaluate the path costs of the successors of \( s \)
3. Add the successors if not visited

We add the successors of \( s \) that have not yet been visited and we update the cost of those currently in the queue
PQ = {(START,0)}

1. Pop the state $s$ with the lowest path cost from $PQ$
2. Evaluate the path cost to all the successors of $s$
3. Add the successors of $s$ to $PQ$
PQ = {(p,1) (d,3) (e,9)}

1. Pop the state s with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of s
3. Add the successors of s to PQ
Add cost from \(p\) to \(q\) to the value of \(p\)

\[\text{PQ} = \{(d,3) \ (e,9) \ (q,16)\}\]
PQ = {(b,4) (e,5) (c,11) (q,16)}

1. Pop the state $s$ with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of $s$
3. Add the successors of $s$ to PQ

The value for item e has been updated from 9 to 5. This is generally not supported by a PQ, but there are ways to fake it.
PQ = \{(b,4) \ (e,5) \ (c,11) \ (q,16)\}

Important: We realized that going to e through d is cheaper than going to e directly, so the value of e is updated from 9 to 5 and it moves up in PQ

1. Pop the state s with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of s
3. Add the successors of s to PQ
PQ = \{(b,4) (e,5) (c,11) (q,16)\}

1. Pop the state $s$ with the lowest path cost from $PQ$
2. Evaluate the path cost to all the successors of $s$
3. Add the successors of $s$ to $PQ$
PQ = \{(e, 5) \quad (a, 6) \quad (c, 11) \quad (q, 16)\}

1. Pop the state \( s \) with the lowest path cost from \( PQ \)
2. Evaluate the path cost to all the successors of \( s \)
3. Add the successors of \( s \) to \( PQ \)
PQ = {(a,6) (h,6) (c,11) (r,14) (q,16)}

1. Pop the state s with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of s
3. Add the successors of s to PQ
PQ = \{ (h, 6) (c, 11) (r, 14) (q, 16) \}

1. Pop the state $s$ with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of $s$
3. Add the successors of $s$ to PQ
PQ = {(q,10) (c,11) (r,14)}

1. Pop the state s with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of s
3. Add the successors of s to PQ
If p is not in the queue but visited, we already visited it.

(If costs aren't negative or 0, this works. In fact this is why it won't work if costs are negative or 0) Note keep in memory everything we've seen.
PQ = \{(q,10) \ (c,11) \ (r,14)\}

1. Pop the state \( s \) with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of \( s \)
3. Add the successors of \( s \) to PQ
PQ = {(c,11) (r,13)}
PQ = \{(r,13)\}

1. Pop the state s with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of s
3. Add the successors of s to PQ
PQ = \{(f,18)\}

1. Pop the state $s$ with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of $s$
3. Add the successors of $s$ to PQ
PQ = {(GOAL,23)}

1. Pop the state $s$ with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of $s$
3. Add the successors of $s$ to PQ
Final path: {START, d, e, h, q, r, f, GOAL}

- This path is optimal in total cost even though it has more transitions than the one found by BFS.
- What should be the stopping condition?
- Under what conditions is UCS complete/optimal?

Can’t stop until you POP the goal!
# Complexity

\[ B = \text{Average number of successors (branching factor)} \]
\[ L = \text{Length from start to goal on shortest path} \]
\[ C = \text{Cost of optimal path} \]
\[ Q = \text{Average size of the priority queue} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td>Y</td>
<td>Y, If all trans. have same cost</td>
<td>(O(B^L))</td>
</tr>
<tr>
<td>BIBFS</td>
<td>Bi-directional Breadth First Search</td>
<td>Y</td>
<td>Y, If all trans. have same cost</td>
<td>(O(2B^{L/2}))</td>
</tr>
<tr>
<td>UCS</td>
<td>Uniform Cost Search</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why is this correct?

Log(Q) is due to priority queue. Each edge cost \(\geq\) epsilon, therefore \(B^{(C/\epsilon)}\) is an upper bound on the number of steps.

Why no log in space? Log has to do with popping priority queue, but space is just storing the thing (so multiply space by 2).
Limitations of BFS
Limitations of BFS

- Memory usage is $O(B^L)$ in general
- Limitation in many problems in which the states cannot be enumerated or stored explicitly, e.g., large branching factor
- Alternative: Find a search strategy that requires little storage for use in large problems

You could visit a lot of nodes (ex driving)
Depth First Search

![Graph Diagram](image)
Depth First Search

General idea:
• Expand the most recently expanded node if it has successors
• Otherwise backup to the previous node on the current path
Depth First Search

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
Depth First Search

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
Depth First Search

General idea:
• Expand the most recently expanded node if it has successors
• Otherwise backup to the previous node on the current path
Depth First Search

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
**Depth First Search**

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
Depth First Search

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
Depth First Search

General idea:
• Expand the most recently expanded node if it has successors
• Otherwise backup to the previous node on the current path
Depth First Search

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
Depth First Search

General idea:
• Expand the most recently expanded node if it has successors
• Otherwise backup to the previous node on the current path
Depth First Search

General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path
DFS Implementation

DFS (s)
if $s = \text{GOAL}$
    return $\text{SUCCESS}$
else
    For all $s'$ in $\text{succs}(s)$
        DFS ($s'$)
    return $\text{FAILURE}$

`s` is current state being expanded, starting with $\text{START}$

In a recursive implementation, the program stack keeps track of the states in the current path.
What’s the problem? Cycles.
Didn’t just remember path (in the stack), also remembered which successor each node has already tried.
BFS:

- Root: START state
- Children of node s: All states in succ(s)
- In the worst case the entire tree is explored $\Rightarrow O(B^{L_{\text{max}}})$
- Infinite branches if there are loops in the graph!
**Complexity**

\[ N = \text{Total number of states} \]
\[ B = \text{Average number of successors (branching factor)} \]
\[ L = \text{Length from start to goal on shortest path} \]
\[ C = \text{Cost of optimal path} \]
\[ Q = \text{Average size of the priority queue} \]
\[ L_{max} = \text{Length of longest path from START to any state} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>(O(B^2))</td>
</tr>
<tr>
<td>BIBFS</td>
<td>Bi-directional Breadth First Search</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>(O(2B^{L/2}))</td>
</tr>
<tr>
<td>UCS</td>
<td>Uniform Cost Search</td>
<td>Y, if cost &gt; 0</td>
<td>Y, if cost &gt; 0</td>
<td>(O(Q \times B^{C/2}))</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Complexity

\( n = \text{Total number of states} \)

\( B = \text{Average number of successors (branching factor)} \)

\( L = \text{Length from start to goal on shortest path} \)

\( C = \text{Cost of optimal path} \)

\( Q = \text{Average size of the priority queue} \)

\( L_{max} = \text{Length of longest path from START to any state} \)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS Breadth First Search</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>( O(B^2) )</td>
<td>( O(B^i) )</td>
</tr>
<tr>
<td>BIBFS Bi-directional Breadth First Search</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>( O(2B^{L/2}) )</td>
<td>( O(2B^{L/2}) )</td>
</tr>
<tr>
<td>UCS Uniform Cost Search</td>
<td>Y, if cost 0</td>
<td></td>
<td>( \log(Q) B^{C/2} )</td>
<td>( O(B^{C/2}) )</td>
</tr>
<tr>
<td>DFS Depth First Search</td>
<td>Y</td>
<td>N</td>
<td>( O(B^{L_{max}}) )</td>
<td>( O(BL_{max}) )</td>
</tr>
</tbody>
</table>

*For graphs without cycles*
Which should you do?

Current path: at least we won’t loop forever, and better for memory usage.
Memdfs means you’d have to keep track of whole graph, and that’s a lot.
**Complexity**

\[ N = \text{Total number of states} \]
\[ B = \text{Average number of successors (branching factor)} \]
\[ L = \text{Length from start to goal on shortest path} \]
\[ C = \text{Cost of optimal path} \]
\[ Q = \text{Average size of the priority queue} \]
\[ L_{max} = \text{Length of longest path from START to any state} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>(O(B^L))</td>
</tr>
<tr>
<td>BIBFS</td>
<td>Bi-Direction BFS</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>(O(2B^{L/2}))</td>
</tr>
<tr>
<td>UCS</td>
<td>Uniform Cost Search</td>
<td>Y, if cost &gt; 0</td>
<td>Y, if cost &gt; 0</td>
<td>(O(\log(Q)B^{Q/2}))</td>
</tr>
<tr>
<td>PCDFS</td>
<td>Path Check DFS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMDF</td>
<td>Memorizing DFS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In PCDFS you can visit the same node twice (though not in the same path). So no luxury of \( N \).

PCDFS is same storage as DFS (we’re already storing the path in the stack).

MEMDFS you would have to store whole graph. However, the time complexity could be as low as \( N \).
DFS Limitation 2

- Need to make DFS optimal
DFS Limitation 2

- Need to make DFS optimal
- IDS (Iterative Deepening Search):
  - Run DFS by searching only paths of length 1 (DFS stops if length of path is greater than 1)
  - If that doesn’t find a solution, try again by running DFS on paths of length 2 or less
  - If that doesn’t find a solution, try again by running DFS on paths of length 3 or less
  - ...
  - Continue until a solution is found
Iterative Deepening Search
Iterative Deepening Search

- Sounds horrible: We need to run DFS many times
- Actually not a problem:
  \[ O(LB^1 + (L-1)B^2 + \ldots + B^L) = O(B^L) \]

- Compare \( B^L \) and \( B^{L_{\text{max}}} \)
- Optimal if transition costs are equal

Why does it not matter to do the same stuff over and over? Exponential growth means the first levels aren’t all that bad.
Iterative Deepening Search

- Memory usage same as DFS
- Computation cost comparable to BFS even with repeated searches, especially for large \( B \).
- Example:
  - \( B = 10, L = 5 \)
  - BFS: 111,111 expansions
  - IDS: 123,456 expansions
**N** = Total number of states

**B** = Average number of successors (branching factor)

**L** = Length from start to goal on shortest path

**C** = Cost of optimal path

**Q** = Average size of the priority queue

**L_{max}** = Length of longest path from **START** to any state

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>O(B^L)</td>
<td>O(B^L)</td>
</tr>
<tr>
<td>BIBFS</td>
<td>Y</td>
<td>Y, if all trans. have same cost</td>
<td>O(2B^{L/2})</td>
<td>O(2B^{L/2})</td>
</tr>
<tr>
<td>UCS</td>
<td>Y, if cost &gt; 0</td>
<td>Y, If cost &gt; 0</td>
<td>O(log(Q)*B^{C/2})</td>
<td>O(B^{C/2})</td>
</tr>
<tr>
<td>PCDFS</td>
<td>Y</td>
<td>N</td>
<td>O(B_{max})</td>
<td>O(B_{max})</td>
</tr>
<tr>
<td>MEMDFS</td>
<td>Y</td>
<td>N</td>
<td>O(B_{max})</td>
<td>O(B_{max})</td>
</tr>
<tr>
<td>IDS</td>
<td>Y</td>
<td>Y, If all trans. have same cost</td>
<td>O(B^L)</td>
<td>O(BL)</td>
</tr>
</tbody>
</table>
Summary

- Basic search techniques: BFS, UCS, PCDFS, MEMDFS, …
- Property of search algorithms: Completeness, optimality, time and space complexity
- Iterative deepening and bidirectional search ideas
- Trade-offs between the different techniques and when they might be used