Searching: Deterministic single-agent
Actually, this is optimization over time with discrete variables
Andrew W. Moore
Professor
School of Computer Science
Carnegie Mellon University
www.cs.cmu.edu/~awm
awm@cs.cmu.edu
412-268-7599

Overview
• Deterministic, discrete, single-agent, search problems
• Uninformed search
• Breadth First Search
• Optimality, Completeness, Time and Space complexity
• Search Trees
• Depth First Search
• Iterative Deepening
• Best First “Greedy” Search

A search problem
How do we get from S to G? And what’s the smallest possible number of transitions?

Formalizing a search problem
A search problem has five components:
Q, S, G, succs, cost
• Q is a finite set of states.
• S ⊆ Q is a non-empty set of start states.
• G ⊆ Q is a non-empty set of goal states.
• succs : Q → P(Q) is a function which takes a state as input and returns a set of states as output. succs(s) means “the set of states you can reach from s in one step”.
• cost : Q x Q → Positive Number is a function which takes two states, s and s’, as input. It returns the one-step cost of traveling from s to s’. The cost function is only defined when s’ is a successor state of s.

Our Search Problem
Q = {START, a, b, c, d, e, f, h, p, q, r, GOAL}
S = {START}
G = {GOAL}
succs(b) = {a}
succs(e) = {h, r}
succs(a) = NULL ... etc.
cost(s, s’) = 1 for all transitions

Why do we care? What problems are like this?
A search problem has five components:

- **Q**: a finite set of states.
- **S**: a non-empty set of start states.
- **G**: a non-empty set of goal states.
- **succs**: a function that takes a state as input and returns a set of states as output. \( \text{succs}(s) \) means "the set of states you can reach from \( s \) in one step".
- **cost**: a function that takes two states, \( s \) and \( s' \), as input. It returns the one-step cost of traveling from \( s \) to \( s' \). The cost function is only defined when \( s' \) is a successor state of \( s \).

But there are plenty of things which we’d normally call search problems that don’t fit our rigid definition…

- Game against adversary
- Chance
- Hidden State
- Continuum (infinite number) of states
- All of the above, plus distributed team control

Label all states that are reachable from \( S \) in 1 step but aren’t reachable in less than 1 step. Then label all states that are reachable from \( S \) in 2 steps but aren’t reachable in less than 2 steps. Then label all states that are reachable from \( S \) in 3 steps but aren’t reachable in less than 3 steps. Etc... until Goal state reached.
Remember the path!

Also, when you label a state, record the predecessor state. This record is called a backpointer. The history of predecessors is used to generate the solution path, once you've found the goal:

"I've got to the goal. I see I was at f before this. And I was at r before I was at f. And I was…
…. so solution path is S → e → r → f → G"
Starting Breadth First Search

For any state \( s \) that we've labeled, we'll remember:
- previous\( (s) \) as the previous state on a shortest path from \( \text{START} \) state to \( s \).

On the \( k \)th iteration of the algorithm we'll begin with \( V_k \) defined as the set of those states for which the shortest path from the start costs exactly \( k \) steps.

Then, during that iteration, we'll compute \( V_{k+1} \), defined as the set of those states for which the shortest path from the start costs exactly \( k+1 \) steps.

We begin with \( k = 0, V_0 = \{ \text{START} \} \) and we'll define, previous\( (\text{START}) \) = NULL.

Then we'll add in things one step from the \( \text{START} \) into \( V_1 \). And we'll keep going.
**Breadth First Search**

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

while (no goal state is in \( V_k \) and \( V_k \) is not empty) do
  \[ V_{k+1} := \text{empty set} \]
  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 signal FAILURE

Else build the solution path thus: Let \( S_i \) be the \( i \)th state in the shortest path.

\[ \text{Define } S_k = \text{GOAL, and for all } i < k \text{, define } S_{i-1} = \text{previous}(S_i). \]

**Another way: Work back**

Label all states that can reach \( G \) in 1 step but can’t reach it in less than 1 step.
Label all states that can reach \( G \) in 2 steps but can’t reach it in less than 2 steps.
Etc. … until start is reached.

“number of steps to goal” labels determine the shortest path. Don’t need extra bookkeeping info.

**Suppose your search space conveniently allowed you to obtain predecessors(state).**
- Can you think of a different way to do BFS?
  - And would you be able to avoid storing something that we’d previously had to store?

**Breadth First Details**

- It is fine for there to be more than one goal state.
- It is fine for there to be more than one start state.
- This algorithm works forwards from the start. Any algorithm which works forwards from the start is said to be **forward chaining**.
- You can also work backwards from the goal. This algorithm is very similar to Dijkstra’s algorithm.
- Any algorithm which works backwards from the goal is said to be **backward chaining**.
- Backward versus forward. Which is better? Checkmate example
Costs on transitions

Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

We will quickly review an algorithm which does find the least-cost path.

On the $k$th iteration, for any state $S$, write $g(s)$ as the least-cost path to $S$ in $k$ or fewer steps.

Least Cost Breadth First

$V_k$ is the set of states which can be reached in exactly $k$ steps, and for which the least-cost $k$-step path is less cost than any path of length less than $k$. In other words, $V_k$ is the set of states whose values changed on the previous iteration.

previous(start) := NIL

$g(\text{start}) := 0$

$k := 0$

while ($V_k$ is not empty)

$V_{k+1}$ := empty set

For each state $s$ in $V_k$

For each state $s'$ in succs($s$)

If $s'$ has not already been labeled

OR if $g(s) + \text{Cost}(s,s') < g(s')$

Set previous($s'$) := $s$

Set $g(s') := g(s) + \text{Cost}(s,s')$

Add $s'$ into $V_{k+1}$

$k := k + 1$

If GOAL not labeled, exit signaling FAILURE

Else build the solution path thus: Let $S_k$ be the $k$th state in the shortest path. Define $S_0 := \text{GOAL}$, and forall $i <= k$, define $S_{i-1} := \text{previous}(S_i)$.

Uniform-Cost Search

• A conceptually simple BFS approach when there are costs on transitions

• It uses priority queues

Priority Queue Refresher

A priority queue is a data structure in which you can insert and retrieve (thing, value) pairs with the following operations:

Init-PriQueue(PQ): initializes the PQ to be empty.

Insert-PriQueue(PQ, thing, value): inserts (thing, value) into the queue.

Pop-least(PQ): returns the (thing, value) pair with the lowest value, and removes it from the queue.

Priority Queues can be implemented in such a way that the cost of the insert and pop operations are $O(\log(\text{number of things in priority queue}))$. Very cheap (though not absolutely, incredibly cheap!)

Uniform-Cost Search

• A conceptually simple BFS approach when there are costs on transitions

• It uses a priority queue

$PQ = \text{Set of states that have been expanded or are awaiting expansion}$

Priority of state $s = g(s) = \text{cost of getting to s using path implied by backpointers}$. 

For more details, see Knuth or Sedgwick or basically any book with the word "algorithms" prominently appearing in the title.
Starting UCS

PQ = \{ (S,0) \}

UCS Iterations

Iteration:
1. Pop least-cost state from PQ
2. Add successors

PQ = \{ (S,0) \}

UCS Iterations

Iteration:
1. Pop least-cost state from PQ
2. Add successors

PQ = \{ (p,1), (d,3), (e,9) \}

UCS Iterations

Iteration:
1. Pop least-cost state from PQ
2. Add successors

PQ = \{ (d,3), (e,9), (q,16) \}

UCS Iterations

Iteration:
1. Pop least-cost state from PQ
2. Add successors

PQ = \{ (b,4), (e,5), (c,11), (q,16) \}

UCS Iterations

Iteration:
1. Pop least-cost state from PQ
2. Add successors

PQ = \{ (b,4), (e,5) \}

Note what happened here:
• d realized that getting to e via d was better than the previously best-known way to get to e
• and so e’s priority was changed
Note what happened here:

- h found a new way to get to p
- but it was more costly than the best known way
- and so p's priority was unchanged
UCS Iterations

PQ = \{ (r,13) \}

Iteration:
1. Pop least-cost state from PQ
2. Add successors

UCS Iterations

PQ = \{ (f,18) \}

Iteration:
1. Pop least-cost state from PQ
2. Add successors

UCS Iterations

PQ = \{ (G,23) \}

Iteration:
1. Pop least-cost state from PQ
2. Add successors

UCS terminates

PQ = \{ \}

Question: Is "terminate as soon as you discover the goal" the right stopping criterion?

UCS terminates

Terminate only once the goal is popped from the priority queue. Else we may miss a shorter path.

Judging a search algorithm

- Completeness: is the algorithm guaranteed to find a solution if a solution exists?
- Guaranteed to find optimal? (will it find the least cost path?)
- Algorithmic time complexity
- Space complexity (memory use)

Variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>number of states in the problem</td>
</tr>
<tr>
<td>B</td>
<td>the average branching factor (the average number of successors) (B&gt;1)</td>
</tr>
<tr>
<td>L</td>
<td>the length of the path from start to goal with the shortest number of steps</td>
</tr>
</tbody>
</table>

How would we judge our algorithms?
### Judging a search algorithm

<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</td>
<td>O(min(N, B^L))</td>
<td>O(min(N, B^L))</td>
</tr>
<tr>
<td>LCBS</td>
<td>Y</td>
<td>Y</td>
<td>O(min(N, B^L))</td>
<td>O(min(N, B^L))</td>
</tr>
<tr>
<td>UCS</td>
<td>Y</td>
<td>Y</td>
<td>O(log(Q) * min(N, B^L))</td>
<td>O(min(N, B^L))</td>
</tr>
</tbody>
</table>

### Depth First Search

An alternative to BFS. Always expand from the most-recently-expanded node, if it has any untried successors. Else backup to the previous node on the current path.

### Search Tree Representation

What order do we go through the search tree with BFS?

### DFS in action

START
START d
START db
START d, b, a
START dc
START d, c
START d, e
START d, e, r
START d, e, r, f
START d, e, r, f, c
START d, e, r, f, c, a
START d, e, r, f

### DFS Search tree traversal

Can you draw in the order in which the search-tree nodes are visited?
### DFS Algorithm

We use a data structure we’ll call a Path to represent the path from the START to the current state.

E.G. Path \( P = <\text{START}, d, e, r> \)

Along with each node on the path, we must remember which successors we still have available to expand. E.G. at the following point, we’ll have

\[ P = <\text{START (expand=e , p)}, \text{d (expand = NULL)}, \text{e (expand = h)}, \text{r (expand = f)}> \]

---

### DFS Algorithm

Let \( P = <\text{START (expand = succs(START))}> \)

While (\( P \) not empty and top(\( P \)) not a goal)

- if expand of top(\( P \)) is empty
  - remove top(\( P \)) ("pop the stack")
- else
  - let \( s \) be a member of expand of top(\( P \))
  - remove \( s \) from expand of top(\( P \))
  - make a new item on the top of path \( P \):
    \[ s \ (\text{expand = succs}(s)) \]

If \( P \) is empty

- return FAILURE
Else

- return the path consisting of states in \( P \)

---

### Judging a search algorithm

<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</td>
<td>( O(\min(N,B^L)) )</td>
<td>( O(\min(N,B^L)) )</td>
</tr>
<tr>
<td>LCBFS</td>
<td>Y</td>
<td>Y</td>
<td>( O(\min(N,B^L)) )</td>
<td>( O(\min(N,B^L)) )</td>
</tr>
<tr>
<td>UCS</td>
<td>Y</td>
<td>Y</td>
<td>( O(\log(Q) \times \min(N,B^L)) )</td>
<td>( O(\min(N,B^L)) )</td>
</tr>
<tr>
<td>DFS</td>
<td>N</td>
<td>N/A</td>
<td>( O(\min(N,B^L)) )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Judging a search algorithm

<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</td>
<td>( O(\min(N,B^L)) )</td>
<td>( O(\min(N,B^L)) )</td>
</tr>
<tr>
<td>LCBFS</td>
<td>Y</td>
<td>Y</td>
<td>( O(\min(N,B^L)) )</td>
<td>( O(\min(N,B^L)) )</td>
</tr>
<tr>
<td>UCS</td>
<td>Y</td>
<td>Y</td>
<td>( O(\log(Q) \times \min(N,B^L)) )</td>
<td>( O(\min(N,B^L)) )</td>
</tr>
<tr>
<td>DFS</td>
<td>N</td>
<td>N/A</td>
<td>( O(\min(N,B^L)) )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Assumptions

- **Assuming Acyclic Search Space**

---

### Assumptions

- **Assuming Acyclic Search Space**
Questions to ponder

• How would you prevent DFS from looping?

• How could you force it to give an optimal solution?

Answer 1: PC-DFS (Path Checking DFS):
Don’t recurse on a state if that state is already in the current path

Answer 2: MEMDFS (Memoizing DFS):
Remember all states expanded so far. Never expand anything twice.

Judging a search algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Y</td>
<td>O(min(N,B))</td>
<td>O(min(N,B))</td>
</tr>
<tr>
<td>LCBFS</td>
<td>Y</td>
<td>O(min(N,B))</td>
<td>O(min(N,B))</td>
</tr>
<tr>
<td>UCS</td>
<td>Y</td>
<td>O(log(Q) * min(N,B))</td>
<td>O(min(N,B))</td>
</tr>
<tr>
<td>PCDFS Path Check DFS</td>
<td>Y</td>
<td>O(min(N,B))</td>
<td>O(min(N,B))</td>
</tr>
<tr>
<td>MEMDFS</td>
<td>Y</td>
<td>O(min(N,B))</td>
<td>O(min(N,B))</td>
</tr>
</tbody>
</table>
Judging a search algorithm

<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</td>
<td>$O(\min(N,B))$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>LCBFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(\min(N,B))$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>UCS</td>
<td>Y</td>
<td>Y</td>
<td>$O(\log(Q) \cdot \min(N,B))$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>PCDFS</td>
<td>Y</td>
<td>N</td>
<td>$O(B\cdot \text{MAX})$</td>
<td>$O(\text{LMAX})$</td>
</tr>
<tr>
<td>MEMDFS</td>
<td>Y</td>
<td>N</td>
<td>$O(\min(N,B)^{\text{MAX}})$</td>
<td>$O(\min(N,B)^{\text{MAX}})$</td>
</tr>
</tbody>
</table>

Maze example

Imagine states are cells in a maze, you can move N, E, S, W. What would plain DFS do, assuming it always expanded the E successor first, then N, then W, then S?

Other questions:
- What would BFS do?
- What would PCDFS do?
- What would MEMDFS do?

Two other DFS examples

Order: N, E, S, W

Order: N, E, S, W with loops prevented

Forward DFSearch or Backward DFSearch

If you have a predecessors() function as well as a successors() function you can begin at the goal and depth-first-search backwards until you hit a start.

Why/When might this be a good idea?

Invent An Algorithm Time!

Here's a way to dramatically decrease costs sometimes. Bidirectional Search. Can you guess what this algorithm is, and why it can be a huge cost-saver?
Iterative Deepening

Iterative deepening is a simple algorithm which uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up any path of length 2)
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.
   …and so on until success

Cost is $O(b^1 + b^2 + b^3 + b^4 \ldots + b^L) = O(b^L)$
Heuristic Search
Suppose in addition to the standard search specification we also have a heuristic.

A heuristic function maps a state onto an estimate of the cost to the goal from that state.

Can you think of examples of heuristics?
- E.G. for the 8-puzzle?
- E.G. for planning a path through a maze?

Denote the heuristic by a function \( h(s) \) from states to a cost value.

Euclidian Heuristic

Another priority queue algorithm.
- But this time, priority is the heuristic value.

Best First “Greedy” Search

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BestFS</td>
<td>Y</td>
<td>N</td>
<td>O(min(N,B_t))</td>
<td>O(min(N,B_t))</td>
</tr>
</tbody>
</table>

A few improvements to this algorithm can make things much better. It’s a little thing we like to call: A*... ...to be continued!

What you should know
- Thorough understanding of BFS, LCBFS, UCS, PCDFS, MEMDFS
- Understand the concepts of whether a search is complete, optimal, its time and space complexity
- Understand the ideas behind iterative deepening and bidirectional search
- Be able to discuss at cocktail parties the pros and cons of the above searches