Uninformed Search

9 September 2020
Reading schedule

Reading journal due 11:59 p.m. before most classes

Assignment 0: Python basics

Out this week; will be posted to class website and announced on Campuswire

Assignment 1: Search

Out next week
Where are we?
An agent perceives its environment through sensors and acts upon it through actuators.
The environment type largely determines the agent design

*Partially observable* → agent requires memory (internal state)

*Continuous* → agent may not be able to enumerate all states

*Stochastic* → agent may need to prepare for contingencies

*Multi-agent* → agent may need to behave randomly
A *rational agent* chooses actions that maximize the expected utility.

*Today:* Agents that have a goal and a cost.
- Reach goal with lowest cost

*Later:* Agents that have numerical utilities, rewards, etc.
- Take actions that maximize total reward over time.
- E.g., largest profit in dollars
Reflex agents
Re
flex agents

Agents choose the next action based on the current percept (and maybe memory).

May have memory or a model of the world’s current state.

Agents don't consider the future consequences of their actions.
Planning agents
Planning agents

Ask “what if…?”

Make decisions based on hypothesized consequences of actions.

Agent must have a model of how the world evolves in response to actions.

Must formulate a goal.
**Complete** planning: Find a solution if one exists

**Optimal** planning: Find the best solution
Today

Search problems

Uninformed search methods

Breadth-first search
Depth-first search
Iterative-deepening
Uniform-cost search
Search problems
A search problem consists of:

A state space:

A successor function
   (with actions, costs)

A start state

A goal test

A solution is a sequence of actions (i.e., a plan) that transforms the start state to a goal state.
A **search problem** consists of:

- **A state space:**

![State space diagram](image)

- **A successor function**
  (with actions, costs)

- **A start state**

- **A goal test**

A **solution** is a sequence of actions (i.e., a plan) that transforms the start state to a goal state.
A search problem consists of:

A state space:

A successor function
(with actions, costs)

A start state

A goal test

A solution is a sequence of actions (i.e., a plan) that transforms the start state to a goal state.
Search problems are models
Example: Traveling in Romania

State space: Cities

Successor function: Roads: Go to adjacent city with cost = distance

Start state: Arad

Goal test: Is the state Bucharest?
What’s in a state space?

The *world state* includes all details of the environment.

E.g., everything you need to know to display the screen of Pacman.

A *search state* keeps only the details needed for planning.

E.g., don’t care what color each ghost is, don’t care if the Pacman glyph is currently open-mouth or closed, etc.
Problem: *Pathing*

**States**: $(x, y)$ locations

**Actions**: north, south, east, west

**Successor**: update location

**Goal test**: Is $(x, y) = \text{END}$?
Problem: *Eat-all-dots*
Problem: *Eat-all-dots*

*States*: \{(x, y) \text{ locations, dot Booleans}\}
Problem: *Eat-all-dots*

*States*: \{(x, y) \text{ locations, dot Booleans}\}

*Actions*: north, south, east, west
Problem: *Eat-all-dots*

**States**: \((x, y)\) locations, dot Booleans

**Actions**: north, south, east, west

**Successor**: update location and possibly a dot Boolean
Problem: *Eat-all-dots*

*States*: \{(x, y) \text{ locations, dot Booleans}\}

*Actions*: north, south, east, west

*Successor*: update location and possibly a dot Boolean

*Goal test*: Are all dots false?
State space sizes

World state:

Agent positions: 120
Dot count: 30
Ghost positions: 12
Agent facing: north, south, east, west

How many

World states: $120 \cdot 2^{30} \cdot 12^2 \cdot 4$
Search states for “pathing”: 120
Search states for “eat all dots”: $120 \cdot 2^{30}$
Problem: *Eat all dots while keeping the ghosts scared*

What does the state space need to specify?
State-space graphs and search trees
A *state-space graph* is a mathematical representation of a search problem.

Nodes are (abstracted) world configurations.

Arcs represent successors (action results).

The goal test is a set of goal nodes.

This is a useful idea even though it’s rare that we can build this full graph in memory.
A state-space graph is a mathematical representation of a search problem.

Nodes are (abstracted) world configurations.
Arcs represent successors (action results).
The goal test is a set of goal nodes.

This is a useful idea even though it’s rare that we can build this full graph in memory.
A **search tree** is a “what if” tree of plans and their outcomes.

- The start state is the root node.
- Children correspond to successors
- Nodes show states but correspond to plans that achieve those states

For most problems, we can never actually build the whole tree
A search tree is a “what if” tree of plans and their outcomes.

The start state is the root node.
Children correspond to successors
Nodes show states but correspond to plans that achieve those states

For most problems, we can never actually build the whole tree
A **search tree** is a “what if” tree of plans and their outcomes.

The start state is the root node.

Children correspond to successors

Nodes show states but correspond to plans that achieve those states

For most problems, we can never actually build the whole tree
A **search tree** is a “what if” tree of plans and their outcomes.

The start state is the root node.

Children correspond to successors

Nodes show states but correspond to plans that achieve those states

For most problems, we can never actually build the whole tree
Each node in the search tree is an entire path in the state-space graph.

We construct both on demand – and we construct as little as possible.
Consider this 4-state graph: How big is its search tree?
Consider this 4-state graph:

How big is its search tree?

∞
Consider this 4-state graph:

```
Consider this 4-state graph: How big is its search tree?
```

```
\[
\begin{array}{c}
s \\
\downarrow \\
b \\
\uparrow \\
a \\
\downarrow \\
G \\
\end{array}
\]
```

```
\[
\begin{array}{c}
s \\
\downarrow \\
b \\
\uparrow \\
a \\
\downarrow \\
G \\
\end{array}
\]
```

\[
\begin{array}{c}
\infty \\
\end{array}
\]

```
\[
\begin{array}{c}
s \\
\downarrow \\
b \\
G \\
\downarrow \\
G \\
\downarrow \\
a \\
\downarrow \\
G \\
\downarrow \\
b \\
\downarrow \\
G \\
\end{array}
\]
```

\[
\begin{array}{c}
\infty \\
\end{array}
\]
Uninformed search
Solving a search problem

First, consider the start state.

Next, explore the state space using the successor function:

Compute the successors of various states until you arrive at a goal state, at which point you have a **plan** – a path for getting from the start state to the goal state.
Maintain a *frontier* of partial plans under consideration

Try to expand as few tree nodes as possible

The order in which states are considered is based on a predetermined *strategy*. 
Breadth-first search
**Strategy:** Expand a shallowest node first

**Implementation:** Frontier is a FIFO queue
**Strategy**: Expand a shallowest node first

**Implementation**: Frontier is a FIFO queue
**Strategy**: Expand a shallowest node first

**Implementation**: Frontier is a FIFO queue
**Strategy:** Expand a shallowest node first

**Implementation:** Frontier is a FIFO queue
**Strategy:** Expand a shallowest node first

**Implementation:** Frontier is a FIFO queue
**Strategy**: Expand a shallowest node first

**Implementation**: Frontier is a FIFO queue
**Strategy**: Expand a shallowest node first

**Implementation**: Frontier is a FIFO queue
BFS processes all nodes above shallowest solution.

It’s *complete*; if a solution exists, we’ll find it!

It’s only *optimal* if the shallowest solution is best (i.e., all costs are 1).
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$. 

$b$ is branching factor
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$. 

$b$ is branching factor

Diagram:
- 1 node
- $b$ nodes
- $b^2$ nodes
- $b^m$ nodes
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$. 

$b$ is branching factor
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$. $b$ is branching factor.
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$. 
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$.

The frontier is roughly the last tier, so space is also $O(b^s)$.
Let depth of shallowest solution be $s$. BFS takes time $O(b^s)$.

The frontier is roughly the last tier, so space is also $O(b^s)$.
Depth-first search
**Strategy:** Expand a deepest node first

**Implementation:** Frontier is a LIFO stack
**Strategy**: Expand a deepest node first

**Implementation**: Frontier is a LIFO stack
**State graph**

**Strategy:** Expand a deepest node first

**State-search tree**

**Implementation:** Frontier is a LIFO stack
Strategy: Expand a deepest node first

Implementation: Frontier is a LIFO stack
**Strategy:** Expand a deepest node first

**Implementation:** Frontier is a LIFO stack
Strategy: Expand a deepest node first

Implementation: Frontier is a LIFO stack
**Strategy:** Expand a deepest node first

**Implementation:** Frontier is a LIFO stack
**Strategy:** Expand a deepest node first

**Implementation:** Frontier is a LIFO stack
**Strategy:** Expand a deepest node first

**Implementation:** Frontier is a LIFO stack
**Strategy**: Expand a deepest node first

**Implementation**: Frontier is a LIFO stack
**Strategy**: Expand a deepest node first

**Implementation**: Frontier is a LIFO stack
**Strategy**: Expand a deepest node first

**Implementation**: Frontier is a LIFO stack
**Strategy**: Expand a deepest node first

**Implementation**: Frontier is a LIFO stack
Strategy: Expand a deepest node first

Implementation: Frontier is a LIFO stack
**Strategy**: Expand a deepest node first

**State graph**

**State-search tree**

**Implementation**: Frontier is a LIFO stack
State graph

Strategy: Expand a deepest node first

State-search tree

Implementation: Frontier is a LIFO stack
Strategy: Expand a deepest node first

Implementation: Frontier is a LIFO stack
DFS expands some *left prefix* of the tree (maybe the whole thing).

DFS is *complete* only if we can prevent cycles. Otherwise the depth could be infinite!

Not *optimal*! Finds the “leftmost” solution, regardless of depth or cost.
If $m$ is finite, DFS takes time $O(b^m)$. 
If $m$ is finite, DFS takes time $O(b^m)$. 

$b$ is branching factor
If $m$ is finite, DFS takes time $O(b^m)$. 

$b$ is branching factor
If $m$ is finite, DFS takes time $O(b^m)$. 

$b$ is branching factor

$m$ tiers

1 node
$b$ nodes
$b^2$ nodes

$b^m$ nodes
If $m$ is finite, DFS takes time $O(b^m)$.

Frontier only has siblings on path to root, so the space for DFS to store the frontier is $O(bm)$. 
BFS vs DFS

When will BFS outperform DFS?

When will DFS outperform BFS?
Iterative deepening
Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages
Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages

Run a DFS with depth limit 1. If no solution…
Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages

Run a DFS with depth limit 1. If no solution...
Run a DFS with depth limit 2. If no solution…
**Idea:** get DFS's space advantage with BFS's time / shallow-solution advantages

- Run a DFS with depth limit 1. If no solution...
- Run a DFS with depth limit 2. If no solution...
- Run a DFS with depth limit 3. ...
Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages

Run a DFS with depth limit 1. If no solution…
Run a DFS with depth limit 2. If no solution…
Run a DFS with depth limit 3. …

Isn’t that wastefully redundant?

Generally most work happens in the lowest level searched, so not so bad!
Uniform-cost search
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
**Strategy:** Expand a cheapest node first

**State graph**

**State-search tree**

**Implementation:** Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue
(priority: cumulative cost)
**Strategy:** Expand a cheapest node first

**Implementation:** Frontier is a priority queue (priority: cumulative cost)
**Strategy:** Expand a cheapest node first

**Implementation:** Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**State graph**

**State-search tree**

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
**Strategy:** Expand a cheapest node first

**State graph**

**State-search tree**

**Implementation:** Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
**Strategy**: Expand a cheapest node first

**Implementation**: Frontier is a priority queue (priority: cumulative cost)
Uniform-cost search issues

Remember: UCS explores increasing cost contours

The good:

UCS is complete and optimal!

The bad:

Explores options in every “direction”
Uniform-cost search issues

Remember: UCS explores increasing cost contours

The good:

UCS is complete and optimal!

The bad:

Explores options in every “direction”
No information about goal location

We’ll fix that soon!
Uniform-cost search issues

Remember: UCS explores increasing cost contours

The good:

UCS is complete and optimal!

The bad:

Explores options in every “direction”

No information about goal location

We’ll fix that soon!
Recognizing search strategies
All of these search algorithms are the same except for the frontier strategies

Conceptually, all frontiers are priority queues (i.e., collections of nodes with attached priorities)

Practically, for DFS and BFS, you can avoid the overhead from an actual priority queue by using stacks and queues
Search and models

Search operates over models of the world

The agent doesn’t actually try all the plans out in the real world
Planning is all “in simulation”
Your search is only as good as your models
Search and models

Search operates over models of the world

The agent doesn’t actually try all the plans out in the real world

Planning is all “in simulation”

Your search is only as good as your models
Next: Informed search
Acknowledgments

The lecture incorporates material from:

Pieter Abbeel, Dan Klein, Stuart Russell, et al., University of California, Berkeley; ai.berkeley.edu
Ketrina Yim (illustrations)