IN SEARCH OF INTELLIGENCE I: HILL CLIMBING

Covered so far:
- AI Overview
- Production Systems
- Agents
- AI Programming (LISP + Python)
PRODUCTION SYSTEMS AND SEARCH

- **Starting w/ an Initial State**
- **Rules** that transform states
  - **Precondition:** determines if rule applicable
  - **Action:** apply rule to state
- **Find path to a state** that satisfies **goal (state)**
  - not necessarily a single state, but a predicate that tells us whether or not a given state satisfies the **goal condition**

<table>
<thead>
<tr>
<th>Knowledge Base</th>
<th>Rules</th>
<th>Search Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defines the problem domain</td>
<td>a.k.a. “Actions”, “Moves” Transform state of system.</td>
<td>An algorithm that describes how we will find a path from Initial State to state satisfying goal condition (if such a path exists)</td>
</tr>
<tr>
<td>State Representation</td>
<td><strong>If(precondition) can apply action:</strong></td>
<td></td>
</tr>
<tr>
<td>Initial State</td>
<td>state ← action(state)</td>
<td></td>
</tr>
<tr>
<td>Goal condition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global knowledge about problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instead of writing an algorithm that will solve the problem directly,

- use an algorithm that:
  - decides which of (potentially many) applicable rules should be applied next
  - knows how to recover if no rules are applicable
“FLAIL WILDLY” SEARCH STRATEGY

• Not intelligent
• But may work sometimes
• ..and may not work other times

“Fox, Goose, Corn” example:
• Many states revisited

======
state=[[ farmer fox corn ]][ goose ]
Choosing rule[2]=Move farmer and corn from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose from Right to Left
======
state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and goose from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose from Right to Left
======
state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and corn from Left to Right
======
state=[[ fox ][ farmer goose corn ]]
Choosing rule[1]=Move farmer and corn from Right to Left
======
"FLAIL WILDLY" SEARCH STRATEGY

- Not intelligent
- But may work sometimes
- ..and may not work other times

"Fox, Goose, Corn" example:
- Many states revisited
- **Step 1: Quest for Intelligence**
  - Reduce stupidity
  - Don't revisit previous states!

<table>
<thead>
<tr>
<th>State</th>
<th>Choice</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>farmer</td>
<td>fox</td>
<td>goose</td>
</tr>
<tr>
<td>farmer</td>
<td>fox</td>
<td>goose</td>
</tr>
<tr>
<td>farmer</td>
<td>fox</td>
<td>goose</td>
</tr>
<tr>
<td>farmer</td>
<td>fox</td>
<td>goose</td>
</tr>
<tr>
<td>farmer</td>
<td>fox</td>
<td>goose</td>
</tr>
<tr>
<td>farmer</td>
<td>fox</td>
<td>goose</td>
</tr>
</tbody>
</table>

---

```python
state=[[ farmer fox corn ][ goose ]]
Choosing rule[2]=Move farmer and corn from Left to Right

state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose from Right to Left

state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and goose from Left to Right

state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and goose from Right to Left

state=[[ farmer fox goose ][ corn ]]
Choosing rule[1]=Move farmer and corn from Left to Right

state=[[ fox ][ farmer goose corn ]]
Choosing rule[0]=Move farmer and corn from Right to Left
```
EXPLORING THE PROBLEM GRAPH

- Building a solution tree
  - Implicit Graph
  - Explicit Graph
Explicit Graph:

- a set of nodes
- a set of edges connecting nodes
- **Directed graph:**
  - edges have direction
  - $a \rightarrow b$ means you can go from $a$ to $b$, but says nothing about whether you can go from $b$ to $a$
- **Undirected graph:**
  - edges do not have direction
  - $x$ connected to $y$ means you can go from $x$ to $y$, or from $y$ to $x$
EXPLICIT GRAPH

Find simplest route from $a \rightarrow e$:

- $a \rightarrow b \rightarrow d$  $\times$
- $a \rightarrow b \rightarrow c \rightarrow e$  $\checkmark$
- $a \rightarrow c \rightarrow e$  $\checkmark$
- $a \rightarrow c \rightarrow a \rightarrow c \rightarrow a \rightarrow c \rightarrow e$  $\checkmark$

All nodes, edges known explicitly
Implicit Graph:
- Generated from initial node(s) and rules for generating successors.
- e.g., *Fox-Goose-Corn* problem:

```
[ farmer fox goose corn ] []
```
Implicit Graph:
- Generated from initial node(s) and rules for generating successors.
- e.g., Fox-Goose-Corn problem:

```plaintext
[ farmer fox goose corn ] [ ]
[ fox goose corn ] [ farmer ]
```
Implicit Graph:
- Generated from initial node(s) and rules for generating successors.
- e.g., *Fox-Goose-Corn* problem:
Implicit Graph:

- Generated from initial node(s) and rules for generating successors.
- e.g., Fox-Goose-Corn problem:

```
[ farmer fox goose corn ] []

[ fox goose corn ] [ farmer ]
[ goose corn ] [ farmer fox ]
[ fox corn ] [ farmer goose ]
```

FEAST

FEAST
Implicit Graph:
- Generated from initial node(s) and rules for generating successors.
- e.g., *Fox-Goose-Corn* problem:
Implicit Graph:
- Generated from initial node(s) and rules for generating successors.
- e.g., Fox-Goose-Corn problem:

FEAST

[ farmer fox goose corn ] []

[ fox goose corn ] [ farmer ]

[ fox goose corn ] [ farmer corn ]

[ fox corn ] [ farmer goose ]

[ goose corn ] [ farmer fox ]

FEAST
Implicit Graph:
• Generated from initial node(s) and rules for generating successors.
• e.g., Fox-Goose-Corn problem:
Implicit Graph:
• Generated from initial node(s) and rules for generating successors.
• e.g., Fox-Goose-Corn problem:

Note: solution space is a subset of a graph
• That subset will be a tree (no node is its own ancestor)
SEARCH STRATEGIES

**Tentative strategy**
- Keep track of all moves from start.
- Don't "actually" apply a move - just "tentatively" apply it - and if it doesn't work, try something else.

When applying a rule in a tentative strategy, the "state" includes your partially constructed solution tree. (Knowledge Base includes this)

**Irrevocable strategy**
- Only keep track of current state - past states are irrelevant
- e.g. solving crossword puzzle in ink: can't "undo" a move:
  - it remains part of solution path, once tried
Tentative Strategy:

- Continue building sub-graph until finding a node satisfying goal condition
- **Note**: when expanding a node, can tell whether a new node \( n \) has
  - already been visited; or
  - already generated but not yet visited
- **Note**: can be costly in terms of space & time
  - can “go back” to earlier states and try new directions
  - need storage space for nodes and links
  - may take too long to examine all previous nodes
IDEA: “IRREVOCABLE STRATEGY”

Irrevocable Strategy:

- Choose a move & make the move.
- Keep current state only.
- Not costly in space and time like tentative strategies.
- **Note:** Irrevocable strategies are generally less space-intensive and time-intensive at each step.
  - if a problem can be solved with an irrevocable strategy, it's probably good to do so.
Production Systems:
- Initial State
- Goal condition
- Rules:
  - precondition
  - action
- Knowledge Base
  - Find a sequence of rules which can be applied to the Initial State, to find a state that satisfies the goal condition.
Production Systems:

- Initial State
- Goal condition
- Rules:
  - precondition
  - action
- Knowledge Base

Find a sequence of rules which can be applied to the Initial State, to find a state that satisfies the goal condition.

“Flailing Wildly”

\[
\text{state } \leftarrow \text{ Initial State} \\
\text{while not } \text{goal} \text{ (state):} \\
\text{M } \leftarrow \text{ applicable Rules} \text{ (state)} \\
\text{choose some } m \in M \\
\text{state } \leftarrow \text{ applyRule} \text{ (m, state)}
\]
**Production Systems:**
- Initial State
- Goal condition
- Rules:
  - precondition
  - action
- Knowledge Base
- **Find a sequence of rules which can be applied to the Initial State, to find a state that satisfies the goal condition.**

**“Flailing Wildly”**

\[
\text{state} \leftarrow \text{Initial State}
\]

while not \( \text{goal} \) (state):

\[
\text{M} \leftarrow \text{applicable Rules} \text{(state)}
\]

choose some \( m \in \text{M} \)

\[
\text{state} \leftarrow \text{applyRule} \text{(m, state)}
\]

How do we choose?
PRODUCTION SYSTEMS, REVISITED

• “Flailing Wildly” can find a solution if you’re lucky/persistent (and if “undo” rules exists).

• **Commutative Production System:**
  • Every state is reachable from every other state
    • Sufficient condition: every rule has an “undo”, and the system cannot be decomposed into separate subsystems
    • “Undo” is not a necessary condition, though – if it is possible to cycle back to another state, for instance, it is not necessary to have an “undo” for each rule.
HILL-CLIMBING

Is there a way of preventing re-visiting a state?

**Hill-Climbing:**
- Create a function $f()$ that "measures" a state and returns a single value in $\mathbb{R}$.
- High value of $f()$: good state.
- Low value of $f()$: bad state.
- Only move in direction that improves value of $f()$.
- Can't revisit earlier state!
- May not always work 😞.
HILL-CLIMBING

Hill-Climbing Strategy:
• Use a function $f(x)$ that increases as solution is approached.
• Choose between moves by increasing $f(\text{state})$
HILL-CLIMBING

**Hill-Climb:**

\[
\text{state} \leftarrow \text{Initial State} \\
\text{value} \leftarrow f(\text{state}) \\
\text{while not goal} (\text{state}): \\
\quad \mathbf{M} \leftarrow \text{applicable Rules} (\text{state}) \\
\quad \text{for each } m \in \mathbf{M} \\
\quad \quad \text{nextState} \leftarrow \text{applyRule} (m, \text{state}) \\
\quad \quad \text{if} \ f(\text{nextState}) > \text{value} \\
\quad \quad \quad \text{value} \leftarrow f(\text{nextState}) \\
\quad \quad \quad r \leftarrow m \\
\quad \text{state} \leftarrow \text{applyRule} (r, \text{state})
\]
HILL-CLIMBING

Hill-Climb:

\[\text{state} \leftarrow \text{Initial State}\]
\[\text{value} \leftarrow f(\text{state})\]
while not \text{goal} (\text{state}):

\[\text{M} \leftarrow \text{applicable Rules} (\text{state})\]
for each \(m \in \text{M}\)

\[\text{nextState} \leftarrow \text{applyRule} (m, \text{state})\]
if \(f(\text{nextState}) > \text{value}\)

\[\text{value} \leftarrow f(\text{nextState})\]
\[r \leftarrow m\]

\[r = \arg \max_{m \in \text{M}} \{ f(\text{applyRule}(m, \text{state})) \}\]

\[\text{state} \leftarrow \text{applyRule}(r, \text{state})\]
HILL-CLIMBING

Hill-Climb:

\[
\text{state } \leftarrow \text{Initial State} \\
\text{value } \leftarrow f(\text{state}) \\
\text{while not goal (state):} \\
\text{M } \leftarrow \text{applicable Rules (state)} \\
\text{for each } m \in M \\
\text{nextState } \leftarrow \text{applyRule (m, state)} \\
\text{if } f(\text{nextState}) > \text{value} \\
\text{value } \leftarrow f(\text{nextState}) \\
\text{r } \leftarrow m \\
\text{state } \leftarrow \text{applyRule (r, state)}
\]

Can't go backwards, revisit earlier state

Pick a rule that improves state (according to \( f(\text{state}) \)) and gives best improvement

Note: if no rule satisfies this, we're stuck.

\[
r = \arg \max_{m \in M} \{ f(\text{applyRule(m, state))} \}
\]
HILL-CLIMBING

**Hill-Climb:**

\[
\text{state} \leftarrow \text{Initial State} \\
\text{value} \leftarrow f(\text{state}) \\
\text{while not } \text{goal} (\text{state}): \\

\text{M} \leftarrow \text{applicable Rules} (\text{state}) \\
\text{for each } m \in \text{M} \\

\text{nextState} \leftarrow \text{applyRule} (m, \text{state}) \text{ if } f(\text{nextState}) > \text{value} \\
\text{value} \leftarrow f(\text{nextState}) \\
\text{r} \leftarrow m \\
\text{state} \leftarrow \text{applyRule} (\text{r}, \text{state})
\]

Note: could modify so that we always choose the alternative providing highest value of \( f \) - even if it's lower than current point.

We won't get stuck at a local optimum, but we also won't stop at a global optimum, either.
EXAMPLES

Mountain Climbing in Fog

- stuck at local optimum
- summit (global optimum)
EXAMPLES

Mountain Climbing in Fog, II

“plateau”, “ridge”
Mountain Climbing in Fog, II

IDEA ... run algorithm several times with different starting points

“plateau”, “ridge”
EXAMPLES

• Adjusting multiple audio controls for sound quality
• Finding optimal set of weights for links in a neural net
• Evolving a population of candidate solutions in an evolutionary computing setting
• Swarm intelligence
EXAMPLES

• Adjusting multiple audio controls for sound quality
• Finding optimal set of weights for links in a neural net
• Evolving a population of candidate solutions in an evolutionary computing setting
• Swarm intelligence

• Some other alternatives:
  • Simulated Annealing
  • Tabu search

  Idea - It’s ok to step backwards occasionally, as long as overall value of $f$ improves over the long run.
PARTICLE SWARM OPTIMIZATION

- generate \( n \) initial guesses \( x_1, x_2, \ldots, x_n \)
- compute \( f(x_1), f(x_2), \ldots, f(x_n) \)
- move in a random direction
  \[ x'_1 = x_1 + \theta_1, \ldots, x'_n = x_n + \theta_n \]
- compute \( \{f(x'_i)\} \)
- for each \( i \), keep track of \( \text{pbest}_i \) (that particle's best value)
- keep track of \( \text{gbest} \) (global best)
- move each \( x'_i \) towards \( \text{pbest}_i \) and \( \text{gbest} \)
- particles tend to “swarm” to a best solution
Sample demos with
\[\alpha = \frac{1}{3}\]
\[\beta = 0.15\]
Sample demos with
\[ \alpha = \frac{1}{3} \]
\[ \beta = 0.15 \]
Sample demos with
\( \alpha = \frac{1}{3} \)
\( \beta = 0.15 \)
Sample demos with
\( \alpha = \frac{1}{3} \)
\( \beta = .15 \)
Sample demos with

\[ \alpha = \frac{1}{3} \]

\[ \beta = 0.15 \]