today’s topics:

- behavior-based AI (finish up from last time)
- problem solving agents
production systems, continued from last class

- Another kind of production system will have an overall goal.
- Imagine that we want the robot to follow the boundary until it finds a north-east corner (like the top-left corner in the example) and then stop there.
- We can define another item in the feature vector:

\[ x_5 = s_1s_2s_3s_4s_5s_6s_7s_8 \]

and then write the production system:

\[ x_5 \rightarrow \text{nil} \]
\[ 1 \rightarrow \text{boundaryfollowing} \]

where nil is an action which does nothing, and boundary_following is a call to the previous production system.
• There are three points to make about this.
• First, in goal-achieving production systems, the topmost rule identifies the situation we are aiming for.
• Once this is achieved, we need do nothing more.
• Second, conditions and actions further down in the production system lead towards the achievement of the topmost condition.
• Indeed, action $a_i$ is intended to bring about $c_j$ where $j < i$.
• Third, we can build up a hierarchy of production systems, where systems lower in the hierarchy move the robot towards meeting the conditions of productions in systems higher up.
• This gives us a means of procedural abstraction.
• Systems of rules like this are called *teleo-reactive* (T-R) programs.
• Every action in a T-R program works towards the achievement of a condition higher in the program.
• It is typically easy to write such programs.
• T-R programs are also very robust.
• Even in the face of faulty sensor readings, carefully constructed T-R programs will get back on track.
Subsumption Architecture

- Another approach to combining simple sensory-driven behavior:

```
[Diagram showing subsumption architecture with nodes for Perception and Action Computation, and processes for Corridor traveling, Obstacle avoidance, and Wandering.]
```

© 1998 Morgan Kaufman Publishers
• Each module receives sensory information directly from the world.
• If the sensory inputs match the preconditions of a module, it executes.
• Modules can subsume each other (in the picture upper modules can subsume lower ones).
• When module \( i \) subsumes \( j \), then if \( i \)'s precondition is met, the program of \( i \) replaces that of \( j \).
• So in the example:
  – The robot wanders until it has to avoid an obstacle;
  – Avoids an obstacle until it is travelling in a corridor.
• Subsumption architecture started with Brooks.

• Idea is that:
  – Build basic behavior;
  – When that is refined, add a subsuming behavior;
  – When that is refined, add another;
  – …

• So far as I know, the maximum “stack height” is not *that* high.

• However, there are other ways of making the approach more sophisticated.
• We can make the approach more flexible:
  – Rather than having a fixed set of behaviors, construct a task specific set.
  – (Plan, but in terms of behaviors not actions.)

• We can improve on subsumption.
  – Rather than having one behavior replace another, merge behaviors.
  – (Imagine being able to do a weighted sum of actions.)

• Both these features are available in Saffiotti’s THINKING CAP.
• How could we program this?

• As follows:

  if <some condition>
  then <some action>
  else if <another condition>
  then <another action>
  else ...

• Here actions higher up in the compound if statement take precedence.
Problem Solving Agents

- earlier, we introduced *rational agents*.
- Now consider agents as *problem solvers*:
  Systems which set themselves *goals* and find *sequences of actions* that achieve these goals.
- What is a problem?
  A *goal* and a *means* for achieving the goal.
- The goal specifies the state of affairs we want to bring about.
- The means specifies the operations we can perform in an attempt to bring about the goal.
- The difficulty is deciding what *order* to carry out the operations.
• Operation of problem solving agent:

/* s is sequence of actions */
repeat {
    percept = observeWorld();
    state = updateState(state, p);
    if s is empty then {
        goal = formulateGoal(state);
        prob = formulateProblem(state,p);
        s = search(prob);
    }
    action = recommendation(s);
    s = remainder(s, state);
}
until false; /* i.e., forever */
• Key difficulties:
  – formulateGoal(…)
  – formulateProblem(…)
  – search(…)

• It isn’t easy to see how to tackle any of these.

• Here we will concentrate mainly on search.
Goal Formulation

• Where do an agent’s goals come from?
  – Agent is a *program* with a *specification*.
  – Specification is to maximise performance measure.
  – Should *adopt goal* if achievement of that goal will maximise this measure.

• Goals provide a *focus* and *filter* for decision-making:
  – *focus*: need to consider how to achieve them;
  – *filter*: need not consider actions that are incompatible with goals.
Problem Formulation

- Once goal is determined, formulate the problem to be solved.
- First determine set of possible states $S$ of the problem.
- Then problem has:
  - *initial state* — the starting point, $s_0$;
  - *operations* — the actions that can be performed, $\{a_1, \ldots, a_n\}$.
  - *goal* — what you are aiming at — subset of $S$. 

• The initial state together with operations determines *state space* of problem.
• Operations cause *changes* in state.
• Solution is a sequence of actions such that when applied to initial state $s_0$, we have goal state.
• Pictorially:
Examples of Toy Problems

- *Example 1*: The 8 puzzle.
  
  Do the following transformation, moving tile from occupied space to filled space.

```
<table>
<thead>
<tr>
<th>2</th>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
```
• Initial state as shown above.
• Goal state as shown below.
• Operations:
  – $a_1$: move any tile to left of empty square to right;
  – $a_2$:
  – $a_3$:
  – $a_4$:
• This defines the following state space:
• Example 2: The $n$ queens problem from chess.
• Place $n$ queens on chess board so that no queen can be taken by another.
• Initial state: empty chess board.
• Goal state: $n$ queens on chess board, one occupying each space, so that none can take others.
• Operations: place queen in empty square.
Solution Cost

• For most problems, some solutions are better than others:
  – in 8 puzzle, number of moves to get to solution;
  – number of moves to checkmate;
  – length of distance to travel.
• Mechanism for determining cost of solution is path cost function.
• This is the length of the path through the state-space from the initial state to the goal state.
• As an example, consider the following state in the 8-puzzle:

```
    2  8  3
   --- ---
    1  6  4
   --- ---
    7  --- 5
```

• How many moves are there to the solution?
• There are four moves:
  1.
  2.
  3.
  4.

• And the path through the solution space looks like:
Problem Solving as Search

- In the state space view of the world, finding a solution is finding a path through the state space.
- When we solve a problem like the 8-puzzle, we have some idea of what constitutes the next best move.
- It is hard to program this kind of approach.
- Instead we start by programming the kind of repetitive task that computers are good at.
- A brute force approach to problem solving involves exhaustively searching through the space of all possible action sequences to find one that achieves goal.
- Systematically generate a *search tree*
- For the 8-puzzle setup as:

```
  2  8  3
  1  6  4
  7  5
```
• The search tree is:
• The tree is built by taking the initial state and identifying some states that can be obtained by applying a single operator.

• These new states become the *children* of the initial state in the tree.

• These new states are then examined to see if they are the goal state.

• If not, the process is repeated on the new states.

• We can formalise this description by giving an algorithm for it.
• General algorithm for search:

```plaintext
agenda = initial state;
while agenda not empty do{
    pick node from agenda;
    new nodes = apply operations to state;
    if goal state in new nodes
        then {
            return solution;
        }
    add new nodes to agenda;
}
```

• Question: How to pick states for expansion?

• Two obvious solutions:
  – depth first search;
  – breadth first search.
Breadth First Search

- Start by *expanding* initial state — gives tree of depth 1.
- Then expand all nodes that resulted from previous step — gives tree of depth 2.
- Then expand all nodes that resulted from previous step, and so on.
- Expand nodes at depth $n$ before level $n + 1$. 
/* Breadth first search */

agenda = initial state;

while agenda not empty do
{
    pick node from front of agenda;
    new nodes = apply operations to state;
    if goal state in new nodes then
    {
        return solution;
    }

    APPEND new nodes to END of agenda;
}
• Advantage: *guaranteed* to reach a solution if one exists.
• If all solutions occur at depth $n$, then this is good approach.
• Disadvantage: time taken to reach solution!
• Let $b$ be *branching factor* — average number of operations that may be performed from any level.
• If solution occurs at depth $d$, then we will look at

$$1 + b + b^2 + \cdots + b^d$$

nodes before reaching solution — *exponential*.
• Time for breadth first search (circa 1995 hardware):

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 msec</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>.01 sec</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>.1 sec</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 secs</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 mins</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>2500 years</td>
</tr>
<tr>
<td>20</td>
<td>$10^{20}$</td>
<td>$3^{15}$ years</td>
</tr>
</tbody>
</table>

• *Combinatorial explosion!*
Importance of ABSTRACTION

• When formulating a problem, it is crucial to pick the right level of abstraction.
• Example: Given the task of driving from New York to Boston.
• Some possible actions...
  – depress clutch;
  – turn steering wheel right 10 degrees;

... inappropriate level of abstraction.
Too much irrelevant detail.
- Better level of abstraction:
  - Take the Henry Hudson Parkway north
  - Take the Cross County turnoff
  . . . and so on.
- Getting abstraction level right lets you focus on the specifics of problem and is one way to combat the combinatorial explosion.
- (Tell that to Mapquest).
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from 1st step, and expand it.
- Pick one of nodes resulting from 1nd step, and expand it, and so on.
- Always expand *deepest* node.
- Follow one “branch” of search tree.
/* Depth first search */

agenda = initial state;

while agenda not empty do
{
    pick node from front of agenda;
    new nodes = apply operations to state;
    if goal state in new nodes then
    {
        return solution;
    }

    put new nodes on FRONT of agenda;
}
• Depth first search is \textit{not} guaranteed to find a solution if one exists.
• However, if it \textit{does} find one, amount of time taken is much less than breadth first search.
• \textit{Memory requirement} is much less than breadth first search.
• Solution found is \textit{not} guaranteed to be the best.
Performance Measures for Search

- **Completeness:**
  Is the search technique guaranteed to find a solution if one exists?

- **Time complexity:**
  How many computations are required to find solution?

- **Space complexity:**
  How much memory space is required?

- **Optimality:**
  How good is a solution going to be w.r.t. the path cost function.
Summary

• This lecture finished simple behavior-based systems from last time.
  – subsumption architecture
• This lecture also introduced the basics of problem solving.
  – problem solving
  – goal formulation
  – state space search
  – abstraction
  – undirected search
    * breadth 1st search
    * depth 1st search
  – performance measures for search
• state space models
  – search for the goal through the state space
  – solution is a/the (best, shortest, cheapest, ...) path through the state space