### Project 1

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. DFS</td>
<td>(2)</td>
</tr>
<tr>
<td>2. BFS</td>
<td>(1)</td>
</tr>
<tr>
<td>3. UCS</td>
<td>(2)</td>
</tr>
<tr>
<td>4. A*</td>
<td>(3)</td>
</tr>
<tr>
<td>5. Corners</td>
<td>(2)</td>
</tr>
<tr>
<td>6. Corners Heuristic</td>
<td>(3)</td>
</tr>
<tr>
<td>7. foodHeuristic</td>
<td>(5)</td>
</tr>
<tr>
<td>8. Suboptimal Search</td>
<td>(2)</td>
</tr>
<tr>
<td>Extra day: 10%</td>
<td></td>
</tr>
</tbody>
</table>

Out of 20 points

Only 30% of final grade

5-6 projects in total

Extra day: 10%
Minimax Properties

- Optimal against a perfect player. Otherwise?

- Time complexity?
  - \( O(b^m) \)

- Space complexity?
  - \( O(bm) \)

- For chess, \( b \approx 35, m \approx 100 \)
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
Resource Limits

- Cannot search to leaves
- Depth-limited search
  - Instead, search a limited depth of tree
  - Replace terminal utilities with an eval function for non-terminal positions
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - $\alpha-\beta$ reaches about depth 8 – decent chess program
Evaluation Functions

- Function which scores non-terminals

- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

- e.g. \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Evaluation for Pac-Man?

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]
Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on 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.

Why do we want to do this for multiplayer games?

Note: wrongness of eval functions matters less and less the deeper the search goes!
Minimax Example

```
3  12  8  2  4  6  14  5  2
```

Pruning in Minimax Search
Alpha-Beta Pruning

- **General configuration**
  - We’re computing the MIN-VALUE at \( n \)
  - We’re looping over \( n \)’s children
  - \( n \)’s value estimate is dropping
  - \( a \) is the best value that MAX can get at any choice point along the current path
  - If \( n \) becomes worse than \( a \), MAX will avoid it, so can stop considering \( n \)’s other children
  - Define \( b \) similarly for MIN
Alpha-Beta Pruning Example

a is MAX’s best alternative here or above
b is MIN’s best alternative here or above
Alpha-Beta Pruning Example

Starting a/b

Raising a

Lowering b

Raising a

a is MAX’s best alternative here or above
b is MIN’s best alternative here or above
Minimax with alpha-beta pruning on a two-person game tree of 4 plies

What move will Max take, and what is its utility? Which nodes will Alpha/Beta pruning leave unexpanded?
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    $v \leftarrow -\infty$
    for $a, s$ in Successors(state) do $v \leftarrow \max(v, \text{Min-Value}(s))$
    return $v$

function Max-Value(state, $\alpha$, $\beta$) returns a utility value
    inputs: state, current state in game
    $\alpha$, the value of the best alternative for MAX along the path to state
    $\beta$, the value of the best alternative for MIN along the path to state
    if Terminal-Test(state) then return Utility(state)
    $v \leftarrow -\infty$
    for $a, s$ in Successors(state) do
        $v \leftarrow \max(v, \text{Min-Value}(s, \alpha, \beta))$
        if $v \geq \beta$ then return $v$
    $\alpha \leftarrow \max(\alpha, v)$
    return $v$
Alpha-Beta Pruning Properties

- This pruning has no effect on final result at the root
- Values of intermediate nodes might be wrong!
  - Important: children of the root may have the wrong value
- Good child ordering improves effectiveness of pruning
- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)
Expectimax Search Trees

- What if we don’t know what the result of an action will be? E.g.,
  - In solitaire, next card is unknown
  - In minesweeper, mine locations
  - In pacman, the ghosts act randomly

- Can do **expectimax search**
  - Chance nodes, like min nodes, except the outcome is uncertain
  - Calculate **expected utilities**
  - Max nodes as in minimax search
  - Chance nodes take average (expectation) of value of children

- Later, we’ll learn how to formalize the underlying problem as a **Markov Decision Process**
Maximum Expected Utility

- Why should we average utilities? Why not minimax?

- Principle of maximum expected utility: an agent should choose the action which maximizes its expected utility, given its knowledge.

- General principle for decision making
- Often taken as the definition of rationality
- We’ll see this idea over and over in this course!

- Let’s decompress this definition…
Reminder: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

Example: traffic on freeway?
- Random variable: \( T = \) whether there’s traffic
- Outcomes: \( T \) in \{none, light, heavy\}
- Distribution: \( P(T=\text{none}) = 0.25, P(T=\text{light}) = 0.55, P(T=\text{heavy}) = 0.20 \)

- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one

- As we get more evidence, probabilities may change:
  - \( P(T=\text{heavy}) = 0.20, P(T=\text{heavy} \mid \text{Hour=8am}) = 0.60 \)
  - We’ll talk about methods for reasoning and updating probabilities later