Recap of Search (1)

Search involves exploring a tree of possible operator sequences. When we formulate a problem as search, we define:

- the state space
  - What states will we consider?
  - What actions will we consider?
  - What is the initial state?
- the goal test
  - How does the agent know when it is done?

All search algorithms can be thought of exploring states in a list of states yet to be visited. "Exploring a state" means running the goal test on the state and, if it's not a goal, adding its successors to the list.

What distinguishes one search technique from another is how it orders the list.

Recap of Search (2)

Uninformed techniques use *problem-independent* information to determine the order in which the agent considers states.

- BFS and DFS
- iterative deepening
- generate-and-test

Informed techniques use *problem-dependent* information to determine the order in which the agent considers states. They use domain knowledge in two ways:

- to order the list of states yet to consider, so that we consider sooner states that are likely to be better

- to evaluate states, so that we don't have to go to the leaves of the state space before knowing if the path is a good one

We considered three informed search methods:

• UCS	`g(n)`
• greedy search	`h(n)`
• A*	`f(n) = g(n) + h(n)`

Recap of Search (3)

With the right sort of `h(n)`, A* can be quite efficient.

If we use an **admissible** heuristic function,
`h′(n),` then A* is *optimal*.

If we take A* as our search algorithm of choice, then the key to doing good search is to find good `h'(n)` functions. This is where the real domain knowledge comes in:

> What do we know about the problem domain that will allow us to evaluate the closeness of a state to a goal?

We would like `h'(n)` to be as accurate as possible: ”Better is better”. But `h'(n)` must **never** overestimate future cost.

If you have to err, then err on the side of (badly) underestimating...

An Example with Heuristic Functions

We could create any number of different h'(n)'s for the 8-puzzle game that use knowledge of the game to make their estimates.

1	2	3
7	8	4
6		5

Two possible h'(n) are:

- h1 = number of tiles out of place.
- h2 = sum of Manhattan distances for each.

Both are admissible, because they underestimate the number of moves required in a given position.

Actually, h2 <= h1 at all depths, so h2 must be always better. In such a case, we say that **h2 dominates h1**.

But how do we invent candidate heuristics in the first place? One useful technique is to *relax the constraints* on operators and then imagine a solution.

Problems with Search in the Real World

How can these search techniques “break”?

1. Even as fast as it is, A* is not fast enough. The search tree for chess contains approximately 10^{160} nodes, and chess is simple compared to most tasks that an intelligent faces.

2. An agent cannot evaluate every node in a search tree from its own perspective. Sometimes, other agents affect the state of the world.

Handling Such Problems in the Real World

How do humans address these problems?

1. *We don’t consider the whole state space.*

   When Emanuel Lasker (world chess champion from 1894 to 1920) was asked how many moves he considered when analyzing a chess position, he replied, “Only one. But it's always the best move.”

   Humans filter the search space, eliminating most of the bad moves. (And, occasionally, some of the good ones.) Last week, we called this *pruning*.

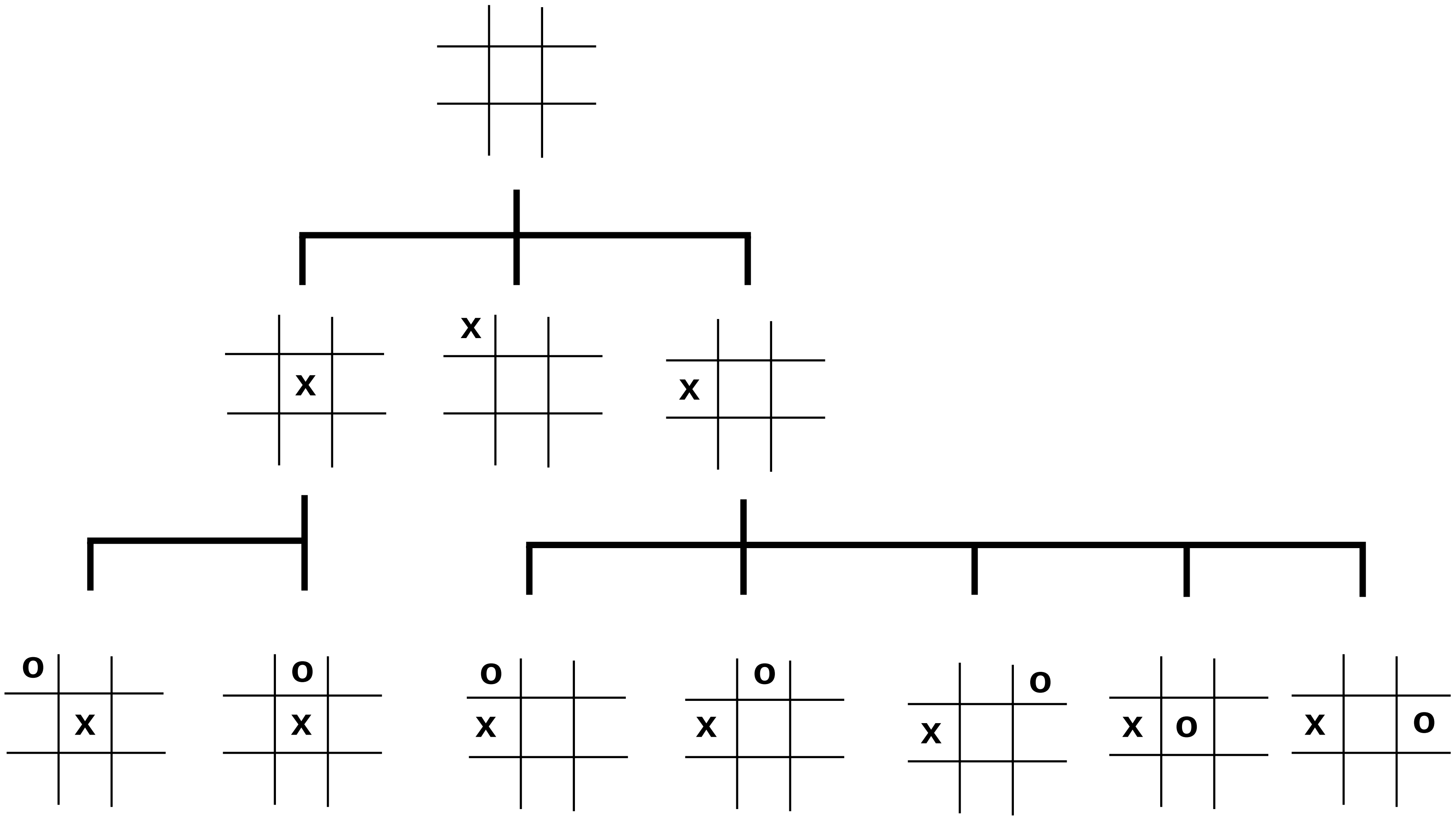
2. *We take into account our opponent’s different perspective.*

   “If I go here, then she’ll go there. But then I can do this, but she’ll do that. ...”

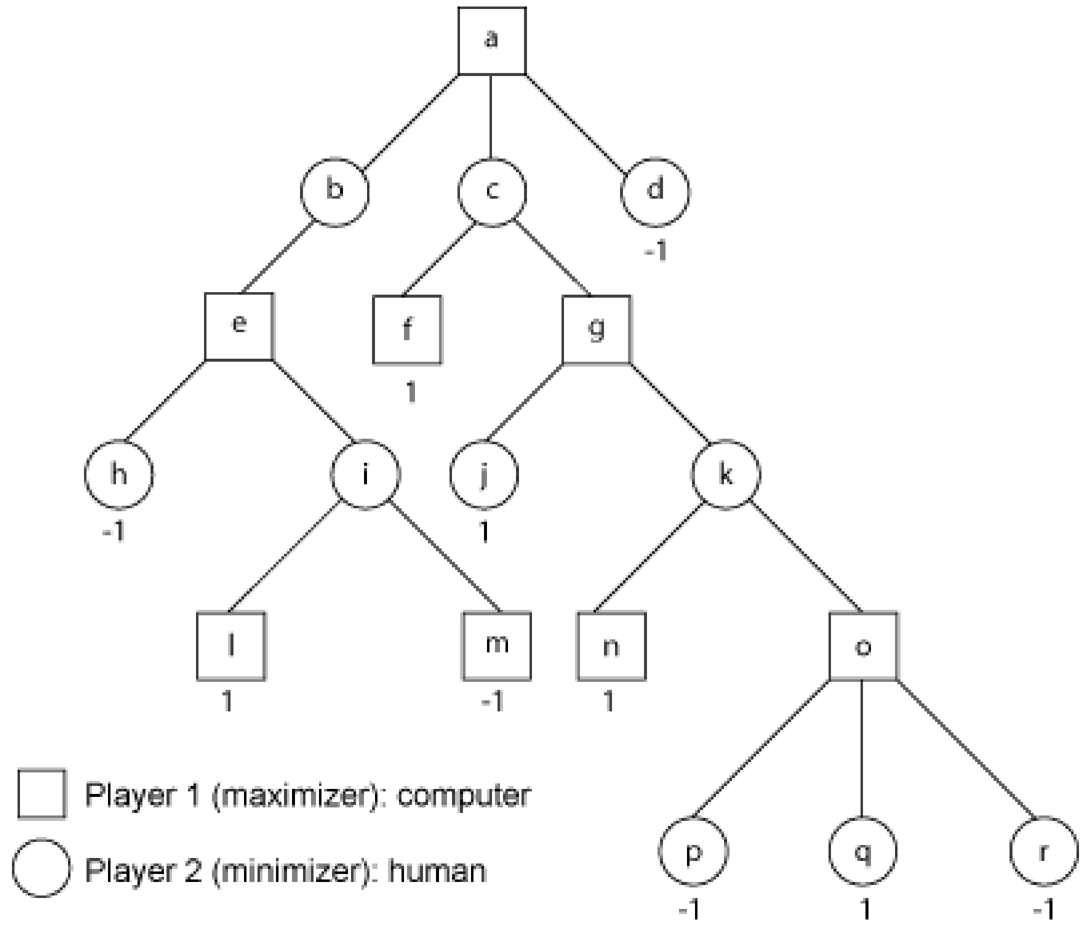
   Humans evaluate positions from the perspective of the player who is to move.

Let’s look at a way to solve the second problem first, and then at a way to solve the first problem second.

Search in Tic-Tac-Toe



Search in an Adversarial World



A simple minimax algorithm:

1. Label the leaves of the tree as “wins” or “losses”.

2. Pick an unlabeled node whose children are all labeled.

3. If the node is on the max level, label it with the max of its children; otherwise, label it with the min of its children.

4. Go to 2.

Problems with Minimax 1.0

In this simplest form, our algorithm has some problems:

- Minimax stores the entire search tree while it works. This requires an amount of space that is exponential in the size of the problem.

- Minimax looks at every state in the search tree. This requires an amount of time that is exponential in the size of the problem.

We know that depth-first search generally reduces time and space costs, so...

Improving Minimax 1.0

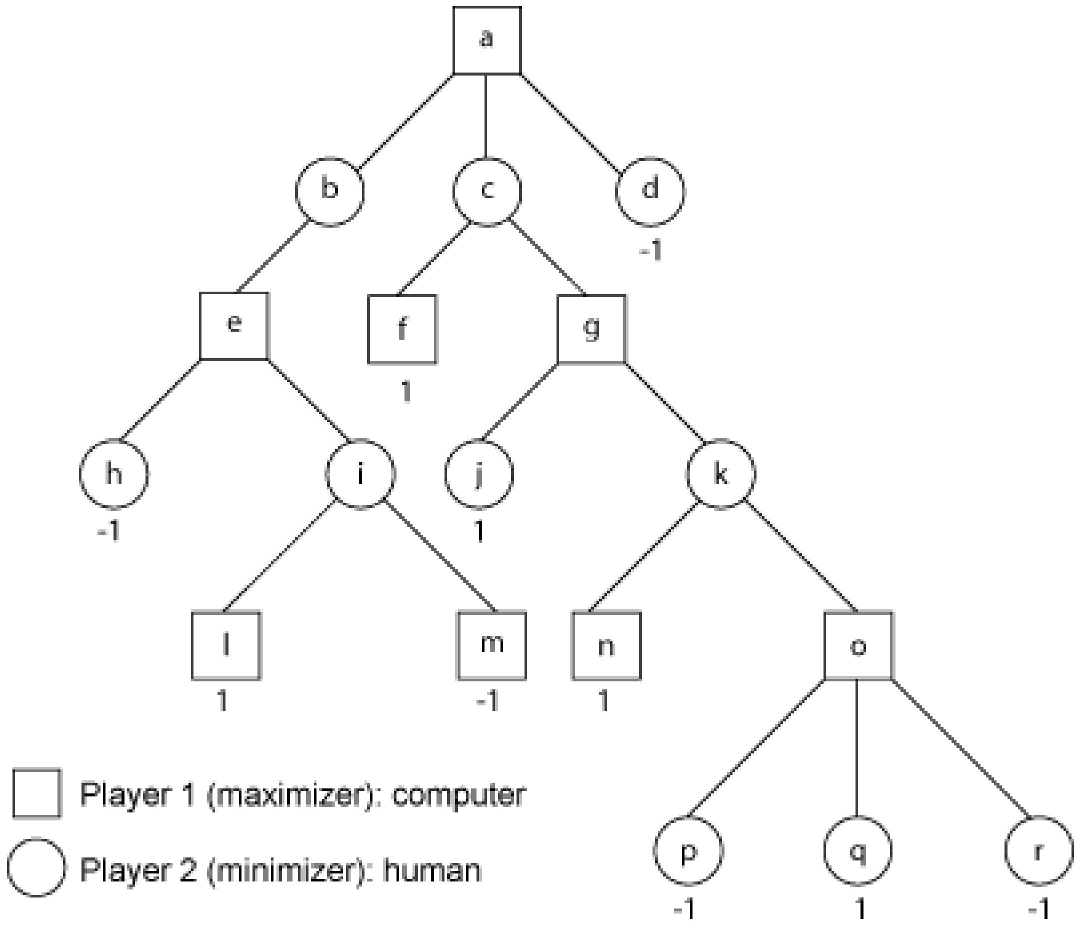
Here is a more efficient Minimax algorithm:

```
1. Set L to be a list containing the initial state.

2. Forever:

   a. Let x = first item on L.  If it is the initial state and has a value, stop.

   b. If x has a value, then
          Retrieve x's parent, p.
          If p is a max node, set p's value to max(p's value, x's value).
          If p is a min node, set p's value to min(p's value, x's value).
          Remove x from L.

      Else if x is a leaf, then
          Set x's value to the value of the position.

      Else if x is an interior node,
          Set x's value to either $-\infty$ if it's a max node or $+\infty$ if it's a min.
          Add x's children to the front of the list.
```

This does a depth-first search and drastically cuts the amount of space required by Minimax.

What are some of the costs of the new algorithm?

Applying Minimax 2.0



Improving Minimax 2.0

Our new algorithm uses much less time, by doing a depth-first search of the state space.

But it still has a major problem:

- Minimax still uses an amount of time exponential in the size of the problem, because it goes all the way to leaf states.

We simply can't afford to consider every state in a large search space...

Let's borrow a solution from A*:

- Devise a *static evaluation function* `e(n)` that assigns a value between -1 (loss) and 1 (win) to each state as a prediction of its value.
- Cut search at some pre-detemined depth `d`.
- Use `e(n)` to evaluate non-leaf states at level `d`.

Could we use a heuristic function from A* as a static evaluation function?

Minimax 3.0

```
1. Set L to be a list containing the initial state.

2. Forever:

   a. Let x = first item on L.  If it is the initial state and
      has a value, stop.

   b. If x has a value, then
        Retrieve x's parent, p.
        If p is a max node, set p's value
          to max(p's value, x's value).
        If p is a min node, set p's value
          to min(p's value, x's value).
        Remove x from L.

      Else if x is a leaf or is at level d, then
        Set x's value to the value of the position,
        now e(x).

      Else if x is an interior node,
        Set x's value to either $-\infty$ if it's a max node
          or $+\infty$ if it's a min.

        Add x's children to the front of the list.
```

Evaluating Minimax 3.0

Depth-first search means: the amount of space used is a linear function of the depth of the tree.

Stopping search at a level `d` means that the amount of time used is an exponential function of `d`—which can be made arbitrarily small. It also further reduces the time requirements.

Setting `d` arbitrarily has disadvantages, too, though:

- In some positions, the values of states and their children may be fluctuating wildly, that is,

  ```
  abs( e(x) - e(x’s parent) ) >> 0
  ```

- For other reasons, one move at the end of a path may be vastly better than its competitors.

- If I know that my opponent is searching only to level `d`, then I can try to make the critical move on a search path fall at level `d + 1`!!