Recap of Local Search:

- useful when path to goal state does not matter/solving pure optimization problem

Basic Idea:

- Only keep a single "current" state
- Heuristic function to evaluate the "goodness" of the current state
- Try to improve iteratively
- Don't save paths followed

Characteristics:

- Low memory requirements (usually constant)
- Effective - can often find good solutions in extremely large state spaces

**CONTINUOUS STATE SPACE**

## Example

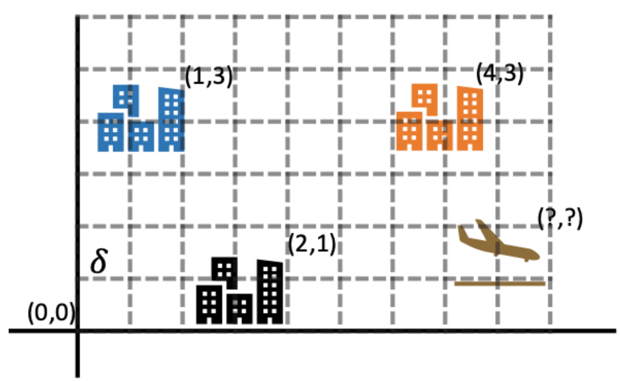
**Problem:**

- Place an airport such that the sum of squared straight-line distances from each city to airport is minimized.

**Formulation:**

- Location of airport $\boldsymbol{x} = [x_1, x_2]$
- Cost function
- $f(\boldsymbol{x}) = \sum_{i=1}^{3} (c_{i,1} - x_1)^2 + (c_{i,2} - x_2)^2$

- **How to handle continuous state?**
  - Discretize into intervals of $\delta$
  - Check "neighbors" and conduct local search
  - E.g., check $f(\boldsymbol{x} + [\delta, 0]) - f(\boldsymbol{x})$



## GRADIENT DESCENT

- $f(\boldsymbol{x}) = \sum_{i=1}^{3}\left(c_{i,1} - x_1\right)^2 + \left(c_{i,2} - x_2\right)^2$
- Without discretization?
  - When discretization approaches 0
- Gradient $\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \lim_{\delta \to 0} \frac{f([x_1+\delta, x_2]) - f([x_1, x_2])}{\delta} \\ \lim_{\delta \to 0} \frac{f([x_1, x_2+\delta]) - f([x_1, x_2])}{\delta} \end{bmatrix}$
- Gradient points to the direction that $f(\boldsymbol{x})$ increases the fastest
- Move in the opposite direction of gradient (steepest slope)

Iterative Algorithm:

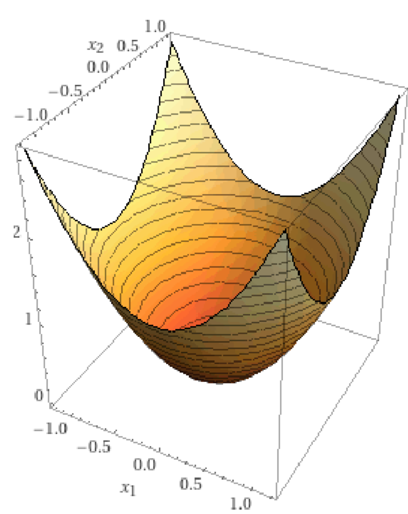
1. Start with some guess $x^0$
2. Iterate $t = 1, 2, 3, \ldots$

- Select direction $d^t$ and stepsize $n^t$
- $x^{t+1} \leftarrow x^t + n^t d^t$
- check stopping condition, $\Delta f(x^t) \approx 0$

Steepest descent: $d^t = -\Delta f(x^t)$

$f(x_1, x_2) = x_1^2 + x_2^2$ $\quad \frac{\partial f}{\partial x_1} = 2x_1 \quad \frac{\partial f}{\partial x_2} = 2x_2$

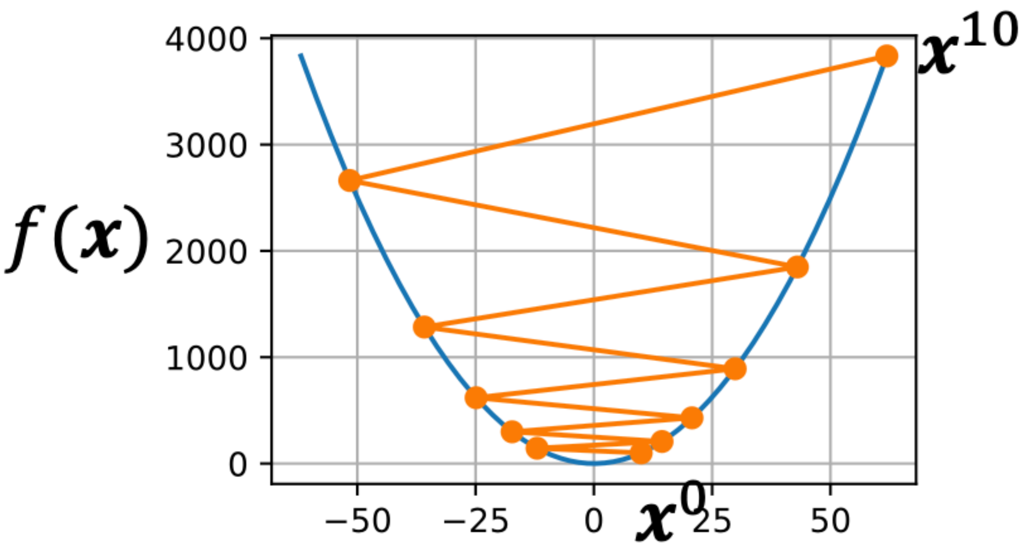
| Initial State | $\frac{\partial f}{\partial \boldsymbol{x}}$ | Step Size |
|---|---|---|
| (2, 3) | (4, 6) | 1 |
| (-2, -3) | (-4, -6) | 0.5 |
| (0, 0) | (0, 0) | - |



**HOW TO SELECT STEPSIZE**?

- Constant: $n^t = 1/L$ for suitable $L$
- Diminishing: $n^t \to 0$ with $\sum_t n^t = \infty$ , ($n^t = 1/t$)

When stepsize is too large:




Types of Games:
Axes:

- Deterministic vs Stochastic
- One, two or more players
- Zero sum vs general sum
- Perfect information (can you see the state) vs Partial information

Algorithms need to calculate a **strategy** (policy) which recommends a **move** (action) from each **position** (state).

**PROBLEM FORMULATION**

- States: $S$ (start at $S_0$)
- Players $P$ = {1, ..., N} ) (take turns)
- ToMove(s): The player whose turn it is to move in state s
- Actions(s): The set of legal moves in state s
- Result(s, a): Transition function, state resulting from taking action a in state s
- IsTerminal(s): A terminal test, true when game is over
- Utility(s,p): $S \times P \to \mathbb{R}$ Final numeric value to player $p$ when the game ends in state $s$

Solution for a player is a policy:

- $S \rightarrow A$

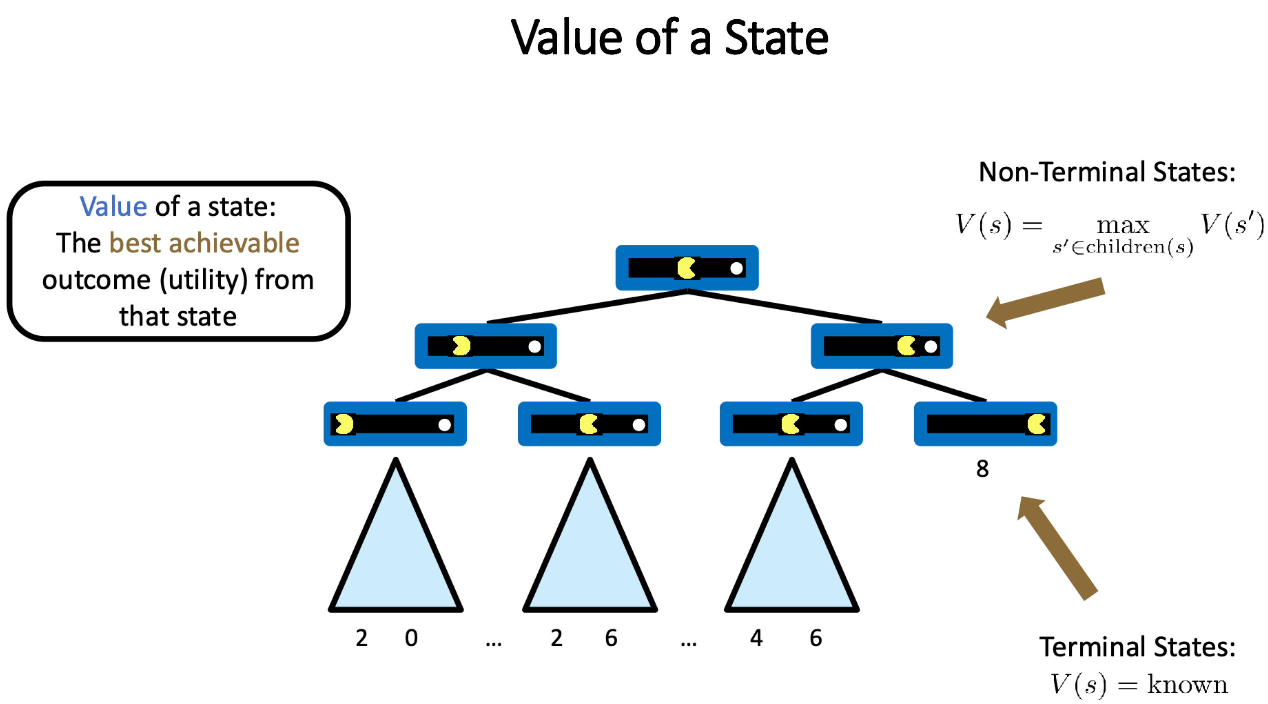
Zero-Sum Games:

- Agents have opposite utilities (values on outcomes)
- Can then think of outcome as a single value that one maximizes, and the other minimizes
- Adversarial, pure competition

General games:

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible

**OPTIMAL DECISION IN GAMES & ADVERSARIAL SEARCH**



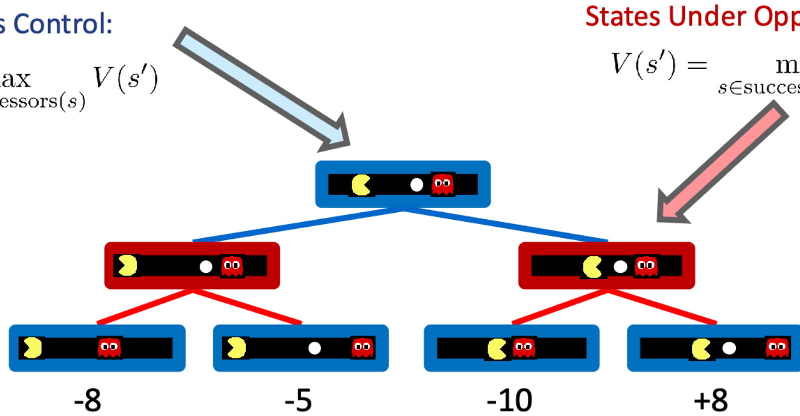
# Minimax Values

States Under Agent’s Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent’s Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$




Terminal States:

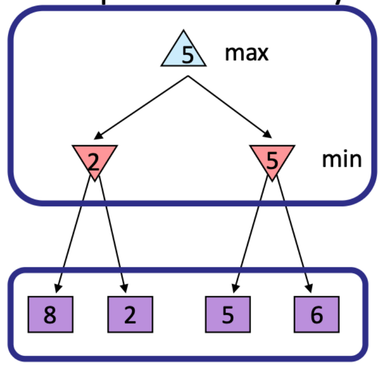
$$V(s) = \text{known}$$

# Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax values:
computed recursively




Terminal values:
part of the game

# Minimax Implementation

```
def max-value(state):
  initialize v = -∞
  for each successor of state:
    v = max(v, min-value(successor))
  return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
  initialize v = +∞
  for each successor of state:
    v = min(v, max-value(successor))
  return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

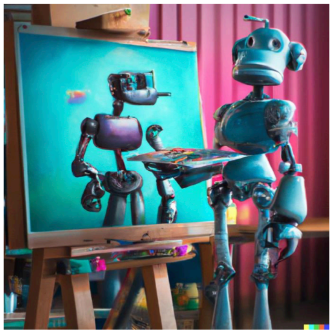
Minimax Efficiency:

- Efficient of minimax search
- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

# Generative Adversarial Network (GAN)

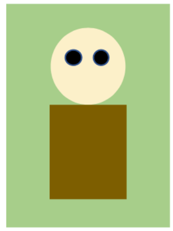
**An adversarial game of image generation**

Generator

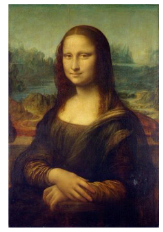


**Objective:** Fool the discriminator

Fake



Real



Generator & Discriminator are neural networks
(Continuous search space!)

Discriminator

**Objective:** Tell the Fake / Real Apart