1 Local and Adversarial Search

(Un-)Informed Search shows some limitations. Typically these algorithms can only handle search spaces with up to 10^{100} states due to memory constraints. They also only consider ”paths” as a solution.

1.1 Optimization Problems

An optimization problem is a problem where every state can be a solution (to different degrees) but the target is to find the state that optimizes (min, max,...) the solution according to an objective function.
This means that there is no explicit goal state and also no path to reach it (no cost).
For example: Darwinian evolution could be seen as an optimization problem.

Objective (Evaluation) Function

An objective function shows how good a state is, also in comparision to other states. Its value is minimized or maximized depending on the problem.

1.1.1 Terminology

Convergence

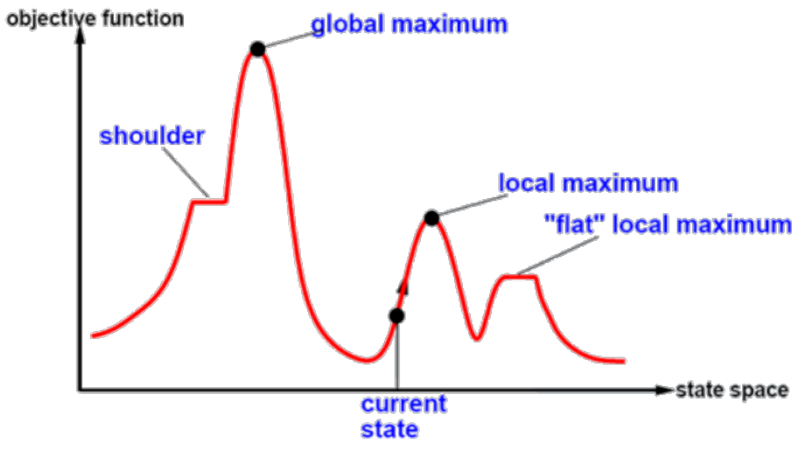
Describes a sequence of (function) values that approach a limit / value more and more.

Global Optimum

The extremum of an objective function over the **entire** input search space

Local Optimum

The extremum of an objective function over **a subset** of the input search space



1.2 Local Search

Local Search algorithms traver only a single state rather than saving multiple paths. It modifies its state iterative, trying to improve a specific criteria.

Optimization problems often times do not care about the path taken, but only to fulfill the goal constraint.

Advantages

- Uses little and constant memory
- Finds reasonable solution is large state spaces

Disadvantages

- No guarantee for completeness or optimality

Basic Idea (Travelling Salesman Problem):

- Start with a complete but likely suboptimal tour
- Modify the tour (e.g. pairwise swap) to improve the objective function
- Repeat until the tour is sufficiently good
- This approach often gets very good results quickly

1.2.1 Hill Climbing

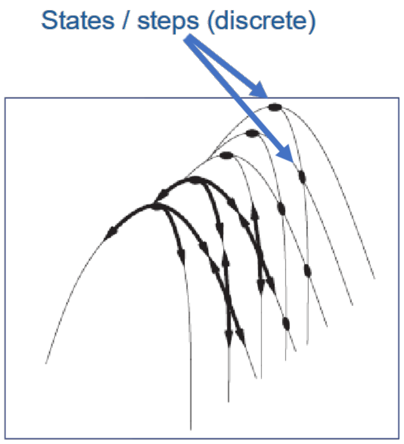
Basic Idea:

- Expand the current state
- Move to the one with the highest value
- Repeat until a maximum is reached

Hill Climbing

```
Function hill_climbing(problem):
    current = make_node(initial_state(problem));
    neighbor;
    While true do
        neighbor = highest_valued_successor(current);
        If value(neighbor) $\leq$ value(current) then
            return state(current);
        current = neighbor;
```

Ridge Problem:
Most Local Search algorithms are implemented by expanding their neighbors and then selecting the one that increases the objective function the most. Often times the problem space is not as simple as that theres only two neighbors, resulting in a more ”3D” or higher search space. This can cause an issue, when all neighbors are worse than the current state, but the seach space has an uphill.

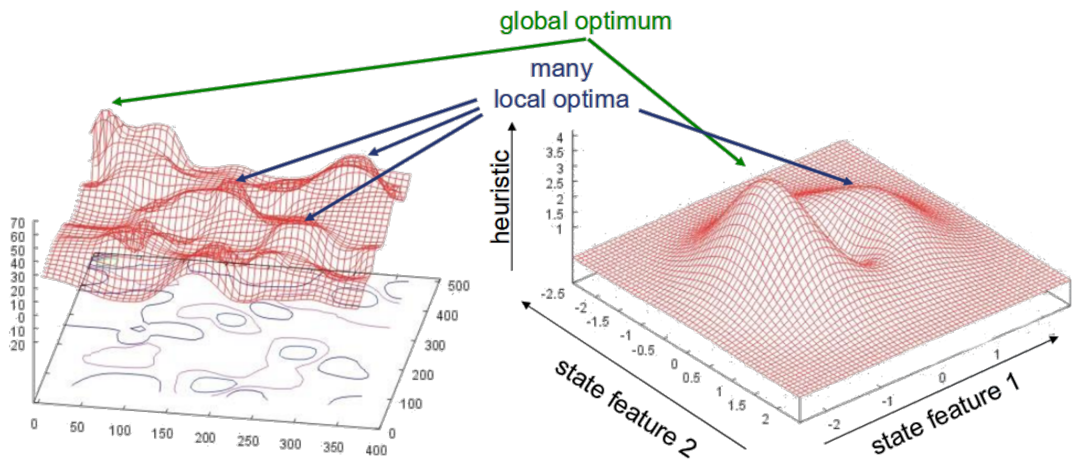


Problem - Local Optima:

- Algorithm will stop at the nearest local optima
- This might be very far from the global optimum (plateus, ridges, shoulders...)
- Solution: Random Restart
  - Random initial positions result in different local optima
  - $\rightarrow$ Iterate over multiple local optima and select the best one
- Alternative Solution: Stochastic Hill Climbing
  - Select successor randomly with a higher chance for better successors

1.2.2 Gradient Descent

As mentioned before, search spaces often are not only 2-dimensional, as we might need to consider multiple features. This, of course, makes it much harder to find an optimal solution with higher dimensional search spaces.



Before our objective function only had a single input feature, now we also have to consider more.

Gradient

The **derivative** of a function that has more than one input variable. In mathematics it would be known as the **slope** of a function, which measures the change in all weights with regard to the change in error.

Gradient Descent

The **gradient descent** is an optimization algorithm. It can be considered as Hill-Climbing in continous state space.

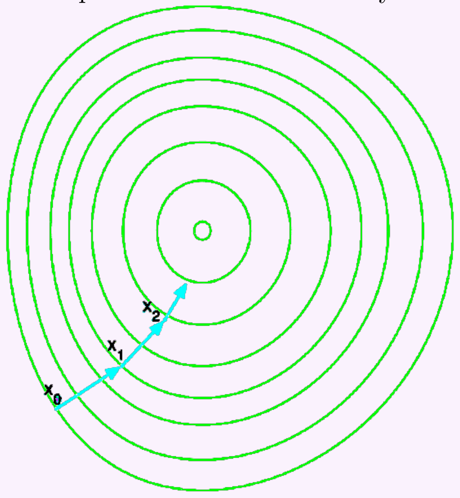
Gradient Descent: Working Principle

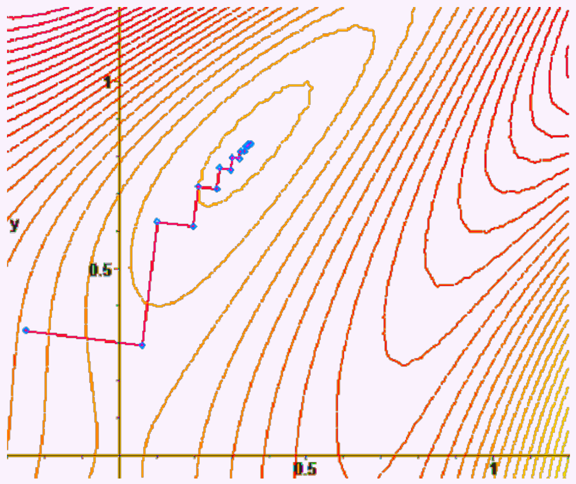
Gradient Vector **Cost Function**

$$\boxed{\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0}, \ldots, \frac{\partial J(\underline{\theta})}{\partial \theta_n}\right]} \quad \boxed{J(x_1, x_2, \ldots, x_n)}$$

We now want to minimize over the continous variables $\boxed{x_1, x_2, \ldots, x_n}$.

1. Compute gradient: $\boxed{\frac{\partial}{\partial x_i} J(x_1, x_2, \ldots, x_n) \quad \forall i \in n}$
2. Take a step downhill in the direction of the gradient: $\boxed{x_i\prime = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, x_2, \ldots, x_n)}$
3. If $\boxed{J(x_1\prime, x_2\prime, \ldots, x_n\prime) < J(x_1, x_2, \ldots, x_n)}$, accept move, else reject.
4. Repeat until desired accuracy is reached





Learning Rate

"The size of the step taken in the gradient descent". The **learning rate** is a hyperparameter, controlling how quickly the model adapts to the problem.

Finding the right learning rate is a very important task. It can be done by trial and error, or by using a learning rate scheduler.

In general:

- **Smaller learning rate:**
  - Smaller changes → requires more training epochs
- **Larger learning rate:**
  - Larger changes → requires fewer training epochs
  - Can converge to local optima or not at all

Determining the gradient can be difficult.

- Derive formula using multivariate calculus
- Ask mathematician or domain expert
- Literature search

- Automatic differentiation

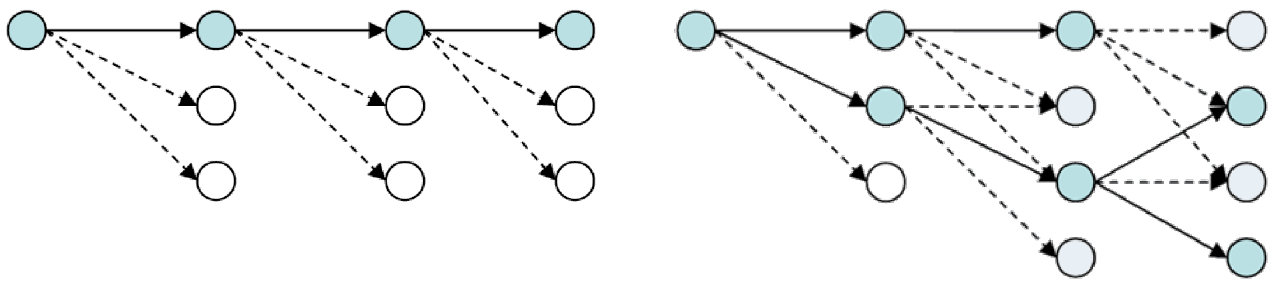
Gradient Descent in general works well for ”smooth” spaces; poorly for ”rough” spaces.

1.2.3 Beam Search

Basic Idea:

- Keep track of k states (beam size) rather than just one

1. Start with k randomly generated states
2. At each iteration, all successors of all k states are generated
3. Select the k best successors from complete list
4. Repeat



1.2.4 Simulated Annealing

Basic Idea:

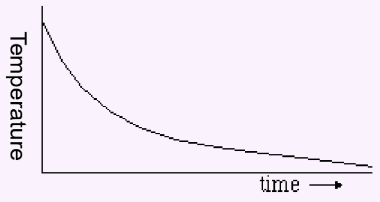
- Use hill-climbing, but occasionally take a step into a direction that does not show improvement
- Reduce the probability of a down-hill step and decrease the size of the step as the number of iterations grows
- Allows some ”bad moves” to escape local optima

Simulated Annealing Algorithm

```
// schedule maps time t to temperature T
Function simulated_annealing(problem, schedule):
    current = make_node(initial_state(problem));
    temperature;
    next;
    For $t = 1$ to $\infty$ do
        temperature = schedule[t];
        next = select_random_successor(current);
        $\Delta E$ = value(next) - value(current);
        If $\Delta E > 0$ then
            current = next;
        // Still accepts worse solution with a probability of $e^{\Delta E/\text{temperature}}$
        Else If random_number(0,1) < $e^{\Delta E/temperature}$ then
            current = next
```

Temperature

Temperature is a hyperparameter, controlling how frequently we accept worse solutions to escape local optima. Temperature usually decays exponentially.



Simulated Annealing converges to a global optimum **if** the temperature is lowered slowly enough. This is not a strong claim as even random guessing would eventually yield the global optimum.

As such, simulated annealing can take a very long time.

1.3 Adversarial Search

Adversarial Search assumes an ”adversary”, who acts against the agent. The goal of adversarial search is to plan ahead, while taking the adversarys actions into account.

Adversarial search is often used to model games as search problems. Each player has to consider other players actions and their effect on the game state.

Adversarial search is often time constrained, so it is unlikely to find an optimal solution.

1.3.1 Games

Zero-Sum Game

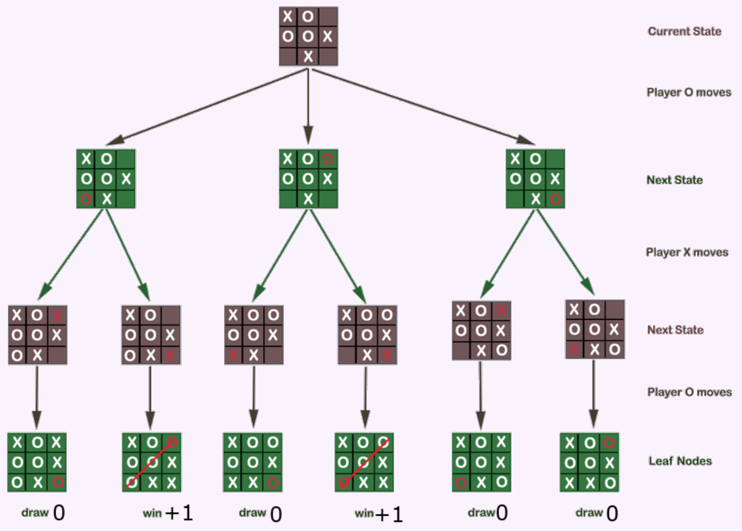
Describes a game whereby, if one party loses, the other must win, therefore the net change in ”points” is zero.

A game can be defined as a search problem:

1. `Initial State:` Game set-up
2. `Player(s):` Specifies which players turn it is
3. `Action(s):` Returns all possible moves in state s
4. `Result(s, a):` Specifies the state s′ after action a in state s is taken
5. `Terminal(s):` Tests if state s fulfills the goal/terminal constraints
6. `Utility(s, p):` Returns a numeric value for a terminal state s from the perspective of player p

Game Trees

Game Trees are used to represent the possible states of a game. Hereby each level corresponds to a player. The **root node** is always the initial (empty) state with current player. **Leaf Nodes** are always terminal states. Every **terminal node** has a utility value corrsponding to the outcome of the game (e.g. $+1$ for a win, 0 for a draw, -1 for a loss).



1.3.2 Games vs. Search Problems

- ”Unpredictable” opponent
  - Specify a move for every possible reply
  - Different goals for every agent
- Time limits
  - Likely not enough time to find a goal state
  - Needs approximation
- Most games are
  - Deterministic, turn-based, two-player, zero-sum
- Real problems are
  - Stochastic, parallel, multi-agent, utility based

1.3.3 Minimax Algorithm

Basic Idea:

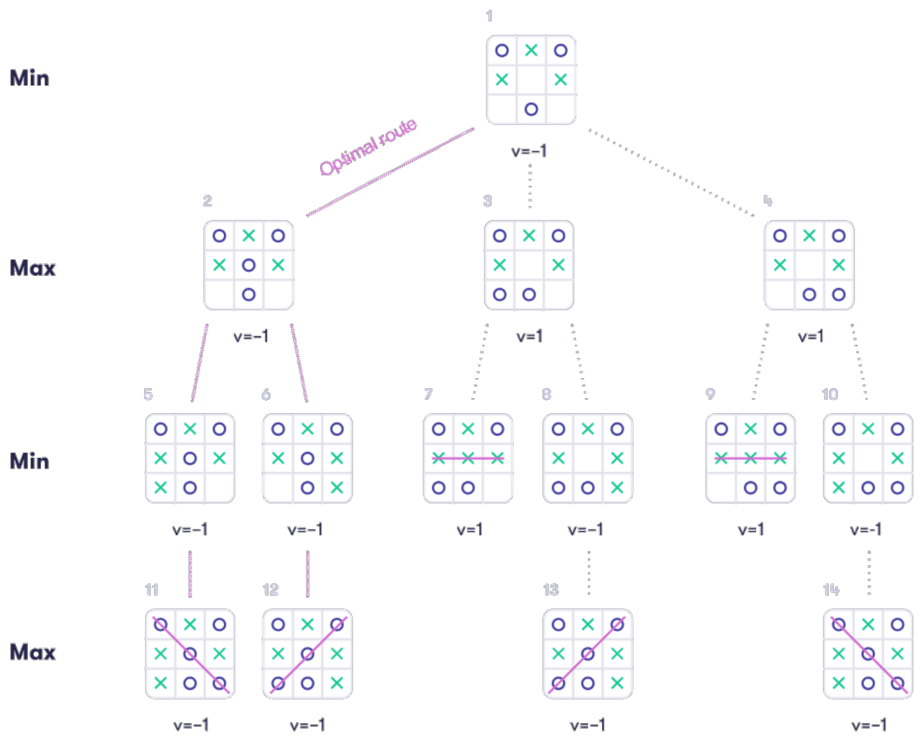
- Build a game tree where nodes represent the states of the game and edges represent the possible moves.
- The players:
  - **MIN:** Decreases the chances of **MAX** winning (Opponent)
  - **MAX:** Increases his own chances of winning (Agent)
- Players take alternating turns following their respective strategies.
- Choose move to position with the highest **minimax value**
- Assume opponent to play the best response to their own action

Minimax Algorithm

```
Function minimax_decision(state):
    v = maximizer(state);
    // Returns the action that leads to state with utility value v
    return get_action(v, get_successors(state));
Function maximizer(state):
    If is_terminal(state) then
        return utility(state);
    v = -∞;
    For s in get_successors(state) do
        v = max(v, minimizer(s));
    return v;
Function minimizer(state):
    If is_terminal(state) then
        return utility(state);
    v = ∞;
    For s in get_successors(state) do
        v = min(v, maximizer(s));
    return v;
```

Characteristics

- **Optimality**: Yes, assuming an optimal opponent
- **Completeness**: Yes, if the tree is finite (e.g. no infinite game loops)
- **Time Complexity**: $O(b^m)$
- **Space Complexity**: $O(b \cdot m)$
- (b is the branching factor, m is the maximum depth)



Problem:
Many games have too many possible moves and go on for too long which causes the time complexity to grow exponentially. Go for example has a branching factor of about 250 and can go on for 150+ turns, which would result in approximately $5 \times 10^{359}+$ iterations in the worst case.

1.3.4 Alpha-Beta Pruning

A modified, optimized version of **Minimax Algorithm**. It uses **pruning** to reduce the amount of exploration without compromising the correctness of minimax.

Alpha-Beta Pruning is based on two parameters

- **Alpha:** The best (highest-valued) choice found so far at any point along the path of the Maximizer to the root. Initial Value: $-\infty$
- **Beta:** The best (lowest-valued) choice found so far at any point along the path of the Minimizer to the root. Initial Value: $+\infty$

Basic Idea:
Remove all nodes which are not affecting the final decision, but slow down the algorithm.

Alpha-Beta Pruning Algorithm

```
Function alpha_beta(state):
    alpha = -∞;
    beta = +∞;
    v = maximizer(state, alpha, beta);
    return v;
Function maximizer(state, alpha, beta):
    If is_terminal(state) then
        return utility(state);
    v = -∞;
    For s in get_successors(state) do
        eval = minimizer(s, alpha, beta);
        v = max(v, eval);
        alpha = max(alpha, v);
        If beta ≤ alpha then
            break;
    return v;
Function minimizer(state, alpha, beta):
    If is_terminal(state) then
        return utility(state);
    v = +∞;
    For s in get_successors(state) do
        eval = maximizer(s, alpha, beta);
        v = min(v, eval);
        beta = min(beta, v);
        If beta ≤ alpha then
            break;
    return v;
```

Differences to Minimax:

- Max Player will only update alpha
- Min Player will only update beta
- While backtracking, the node values will be passed to upper nodes instead of alpha and beta
- Alpha and beta will only be passed to child nodes

Problems:

- Needs a fast evaluation function
- Games with large branching factors (e.g. Go) exploration with alpha-beta pruning is very slow