EiKI Summary

Moritz Gerhardt

moricetamol

Table of Contents

1 What is AI?

In general, there is no set definition of what artificial intelligence is. The most common definition is:

> A system of machines that can perform tasks that normally require human intelligence.

This is flawed in that intelligence itself also does not have a set definition, and neither does human intelligence.

1.1 What is Intelligence?

1.1.1 Touring Test

Intelligence is often defined by the Touring Test. Hereby it is assumed that an entity is intelligent if it cannot be distinguished from another intelligent entity by observing its behavior.

The Touring Test is set up like this:

1. Human interrogator interacts with two entities, A and B. Hereby one of the two is already assumed to be intelligent (another human).
2. If the interrogator cannot distinguish which entity is the assumed intelligent entity, it is assumed that the other entity is intelligent as well.

This test is also flawed as it does not distinguish between different intelligence levels (knowledge, reasoning, language understanding, learning etc.). It also is very subjective to the interrogator. It is not based on an objective metric and therefore the outcome can differ from person to person.

It also assumes that humans are inherently intelligent, which makes it a circular argument.

1.1.2 Chinese Room Argument

The Chinese Room Argument tries to answer the question whether intelligence is the same as intelligent behavior. It assumes that even if a machine behaves intelligently, that does not mean it is actually intelligent.

The argument goes as follows:

1. A person who doesn't know chinese is put into a room. Outside the room there is a person, who can only interact with this person by slipping them notes written in chinese.
2. The person inside the room has detailed instructions as to how to answer the notes, without any translation or understanding.
3. To the person outside of the room it looks like the person inside is able to understand and answer to these notes, therefore the person outside of the room assumes that the person inside knows chinese.

So the person outside of the room assumes intelligence due to behavior, that does not necessitates the intelligence if the answers are already known.

Is a self-driving car intelligent? Is ChatGPT intelligent?

1.1.3 Does it even matter?

The question of what intelligence is is a long, complex and difficult question. However, as it is a more philosophical question it doesn't actually really affect the scientific field of AI. It's a definition, not a basis.

1.2 Characteristics of AI

AIs are often divided into two categories: general AI and narrow AI.

General & Narrow AI

- General (strong) AI is defined so that it can handle any intellectual task
- Narrow (weak) AI is defined so that it has a specific task or domain it works in.

This mirrors the distinction between intelligence and acting intelligent. Currently, most AIs are considered narrow, general AIs are the goal in most research.

An AI should posses the following characteristics:

Adaptability

The ability to improve performance by learning from experience

Autonomous

The ability to perform tasks without constant guidance from a user / expert

Usually an AI is modelled after one of the two following concepts:

Law of Thoughts

The AI should be able to reason about its own actions, arguments and thought processes.
More akin to human intelligence.

Rational Behavior

The AI should be able to determine what the best action is for a given situation to maximize the achievement.
Based on a mathematical model of rationality that achieves the most desirable outcome given the information available.

Rationality has two advantages

- **More General:** For many situations a provable correct option does not exist. The most likely outcome is a better solution.
- **More ammenable:** Rationality can be defined, refined and optimized.

However, rationality rarely displays a good model of reality.

Systems that think and behave like humans are often talked about in the scientific field of **Cognitive Science**.

1.3 Foundations of AI

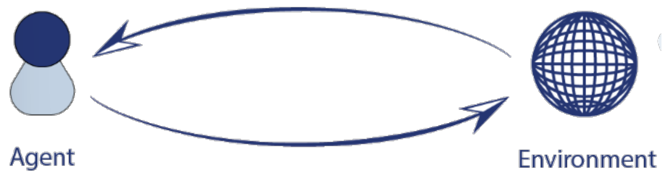
The field of AI is made up by many other fields:

- **Philosophy:** Logic, reasoning, mind as a physical system, foundations of learning, language, rationality
- **Mathematics:** Formal representation and proof algorithms, computation, decidability, tracability, probability
- **Psychology:** Adaptation, phenomena of perception and motor control
- **Economics:** Formal theory of rational decisions, game theory
- **Linguistics:** Knowledge representation, grammar
- **Neuroscience:** Physical substrate for mental processes
- **Control theory:** homeostatic systems, stability, optimal agent design

2 AI Systems

2.1 What is an AI System?

An AI System can be defined as the study of **rational agents** and **their environments**



2.1.1 Environment

- "The surroundings or conditions in which a person, animal, or plant lives or operates" - Oxford Dictionary
- In AI: The surrounding of an (AI) agent, where the agent operates
- Does not have to be real - Can be artificial
- Example:
  - Selfdriving cars: Street, traffic, weather, road signs, ...
  - Chess: Board, pieces,...

Characteristics of Environments

Discrete vs. Continous

Discrete: Environment has countable number of distinct, well defined states
Continous: Not discrete - Uncountable number of states

For Example:

- Discrete: Chess - Every state of the board is mathematically determinable and defined
- Continous: Selfdriving car - Practically infinite positions and conditions

Observable vs. Partially Observable / Unobservable

Observable: State is completely determinable at each point in time
Partially Observable: State is only partially determinable or only determinable at specific points in time
Unobservable: State is never determinable

Static vs. Dynamic

Static: Environment does **not** change while the agent is acting
Dynamic: Environment can change while the agent is acting

For Example:

- Static: Jigsaw puzzle - State does not change without the agents action
- Dynamic: Driving - State still changes even when the agent stops acting

Single Agent vs. Multiple Agents

Single Agent: Environment contains only one agents
Multiple Agents: Environment can contain multiple agents

Accessible vs. Inaccessible

Accessible: Agent can obtain complete and accurate information about the state
Inaccessible: Agent cannot obtain complete or only inaccurate information

Deterministic vs. Stochastic / Probabilistic

Deterministic: Next state is completely determined by the current state and the actions of the agents
Probabilistic: Next state is also influenced by other factors

Episodic vs. Non-episodic / Sequential

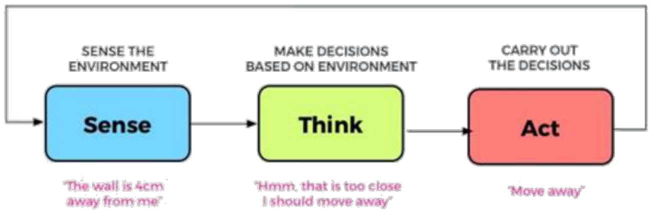
Episodic: Each **episode** consists of the agent perceiving and then acting - every action is dependent only on the episode
Non-episodic: Actions are also dependent on past memory

These characteristics are important, as the environment specifies the specific needs of the agent:

- Different environments require different agent designs
- Not every algorithm works for a specific environment

2.1.2 Agents

- **Sense:** Perceives its environment
- **Think:** Makes decisions autonomously
- **Act:** Acts upon the environments


SENSE THE ENVIRONMENT
MAKE DECISIONS BASED ON ENVIRONMENT
CARRY OUT THE DECISIONS
Sense
Think
Act
"The wall is 4cm away from me"
"Hmm, that is too close I should move away"
"Move away"

Rules of AI Agents

1. Must be able to perceive its environment
2. Observations must be used to make decisions
3. Decisions should be used to act
4. (Action should be rational)

Rational Agents

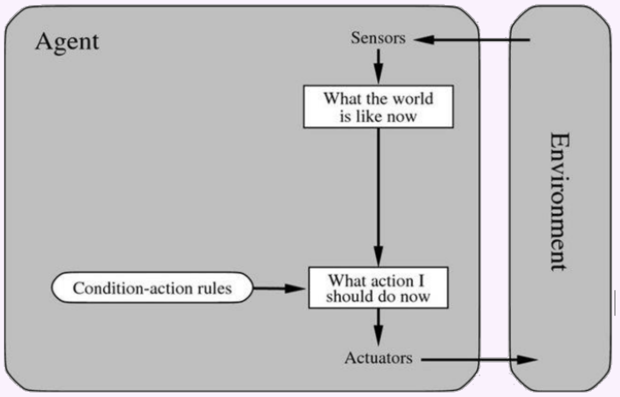
Maximizes the performance and yield the best positive outcome.
Performance is hereby often measured by a function that evaluates a sequence of actions. This function is task-dependent and cannot be generalized.

2.1.3 Types of Agents

Reflex Agent

Act only on the basis of the current percept, ignores past memory

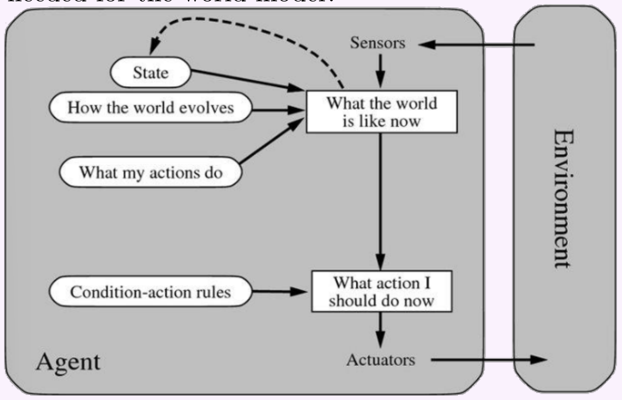
- Implemented through condition-action rules - map state to action
- Problem:
  - Limited decision making
  - No knowledge about anything that cannot be currently perceived
  - Hard to handle in complex environments



Model-Based Agent

Similar decision making to reflex agent, but keep track of the world state

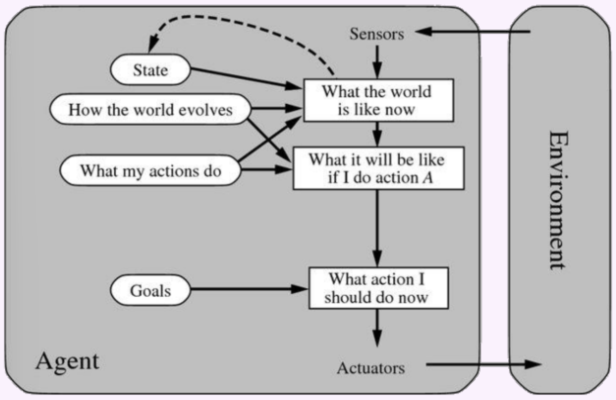
- Input is interpreted and mapped to an internal state representation of the world
- Problem:
  - How do these action affect the internal representation of the world?
  - What details are needed for the world model?



Goal-Based Agent

Essentially a Model-Based Agent with additional functionality that stores desirable states

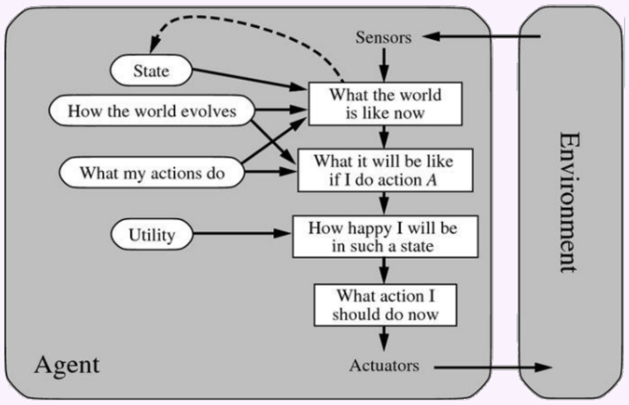
- Knows what states are desirable and acts towards them
- Problem:
  - Difficult to choose actions if a lot of actions are required to achieve a goal


Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Utility-Based Agent

Similar to Goal-Based Agents, but instead of providing goals it provides a utility function for rating actions and scenarios based on the desirability of the result

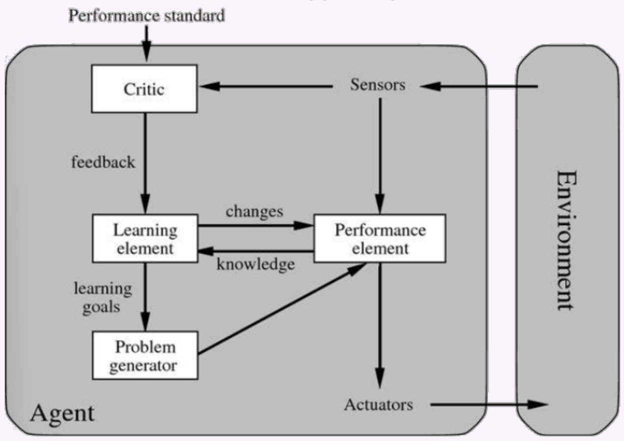
- Goals provide binary distinction, while utility functions provide a continuous measure of desirability
- Can handle selection between two conflicting goals - ”Speed or safety”


Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Learning Agent

Employs additional learning element to gradually improve and become more knowledgable over time about an environment

- Can learn from past experiences
- Is more robust towards unknown environments
- A learning agent has four conceptual components:
  1. **Learning Element:** Makes improvements by learning from the environment
  2. **Critic:** Gives feedback on the performance of the agent according to a fixed metric
  3. **Performance Element:** Selects the best action according to the critic
  4. **Problem Generator:** Responsible for suggesting actions that will lead to new experiences



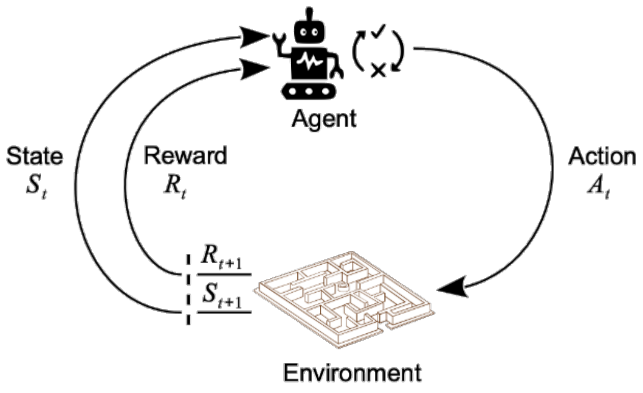
2.2 Creating Intelligent Agents

2.2.1 Search Algorithms

Define ”finding a good action” as a search problem and use search algorithm to solve it. Most of these seach algorithms are tree based, altough Bread-First is used often.

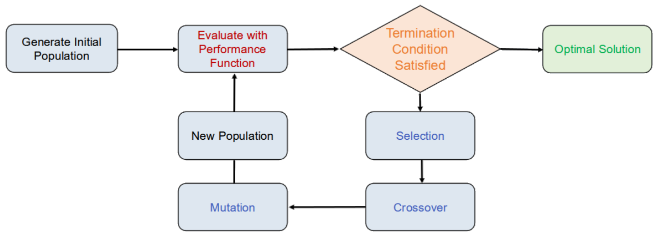
2.2.2 Reinforcement Learning

Developed in psychology, Reinforcement Learning essentially is a process of trial and error - Try something out, have a reaction to it, learn wether it was good or bad. Reactions or actions are based on out observations or experiences.



2.2.3 Genetic Algorithms

Inspired by Darwins ”Theory of natural selection”, this model build multiple different agents and evaluates every one of them. The agent with the highest performance is selected and new models are built based on it. This is done until the performance is good enough.


Generate Initial Population
Evaluate with Performance Function
Termination Condition Satisfied
Optimal Solution
New Population
Selection
Mutation
Crossover

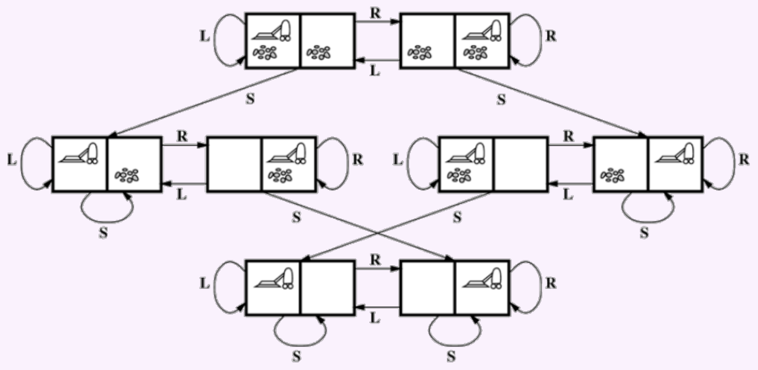
3 Uninformed and Informed Search

3.1 Problem Solving

3.1.1 Key Terminology

State Space / States

A state is a possible situation in our environment - The State Space is a set of all possible states, reachable from the initial state.
Often visualized as a **state-space graph** that displays all reachable states and its transitions.



State-Space Graph of a robot vacuum cleaning

Transition / Action

A Transition describes possible actions to take to get from one state to another. We only count direct transitions between two states (single actions)

Costs

Transitions often differ in different qualities. We add a "cost" to each action, so we can rate an algorithm on how cost effective it is.

Path

A sequence of states connected by a sequence of actions

Solution

A path that leads from the initial state to a goal state

Optimal Solution

The Solution with the minimal path cost

Problem Components:

1. **State Space and Initial State:**

All possible states and the initial environment as a state

2. **Descriptions of Actions:**

   Function that maps a state to a set of possible actions in this state

3. **Goal Test:**

   Typically a function to test if the current state fulfills the goal

4. **Costs:**

   A cost funciton that maps actions to costs

   An easy way it to add costs of all actions taken

Now that we have defined problems, lets go on to solutions. Most problems can be defined as **planning problems**, in which we start from an initial state and transform it into a desired goal considering future actions and outcomes.

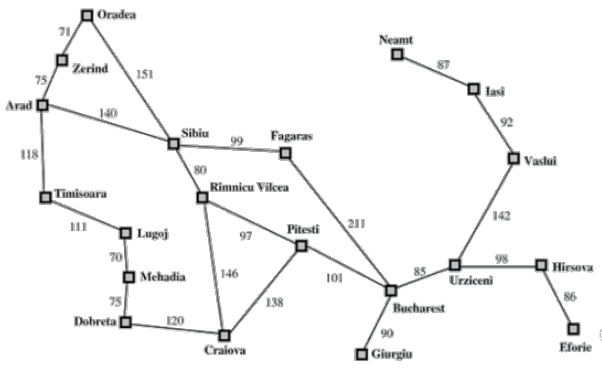
To solve these problems we usually utilize **search** algorithms to find a (optimal) solution in form of a sequence of actions.

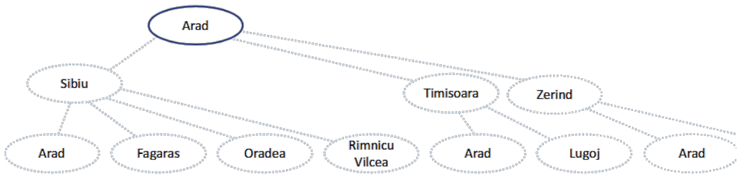
3.1.2 Tree Search Algorithms

Hereby we treat the state-space graph as a tree, starting with the initial state. Using this approach we can define iterative or recursive algorithms to search for suitable paths.

Example Tree Search

```
Function tree_search(problem, strategy):
    initialize search tree with initial state of problem;
    While true do
        If Node contains goal state then
            return solution
        Else If No suitable candidates for expansion then
            return failure
        Else
            Expand node and add resulting nodes to the tree
```





Each Node in the search tree is an entire path in the state-space graph. We construct both on demand - only as much as we need (won't get all solutions)

Node

Describes a part of a tree. Includes **state, parent node, taken action, path cost and depth of the tree**

Fringe

Describes the set of nodes at the end of all visited paths

Depth

Number of levels in the search tree

3.2 Uninformed Search

Definition

The uninformed search strategy only has the problem definition, no additional information.
Some algorithms that utilize uninformed search are:

- Uniform-Cost Search (UCS)
- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Depth-Limited Search (DLS)
- ...

3.2.1 Uniform-Cost Search (UCS)

IN UCS each node is associated with a fixed cost, which accumulate over the path. UCS uses the lowest cumulative cost to find a path.

Only works if each step has a positive cost, as otherwise infinite loops would occur

Complexity: $O(b^{1+\lfloor \text{OptimalCost}/\text{eps} \rfloor})$ with b being the branching factor (average number of children per node) and eps being the minimal cost for a step

3.2.2 Breadth-First Search

BFS is a special case of UCS, when all costs are equal. It starts at the root and goes through each node of a level before progressing to the next level. BFS stops as soon as it finds a solution, while UCS searches for the 'best' solution by lower cost.

Complexity: $O(b^d)$ with b being the branching factor and d being the depth of the search tree

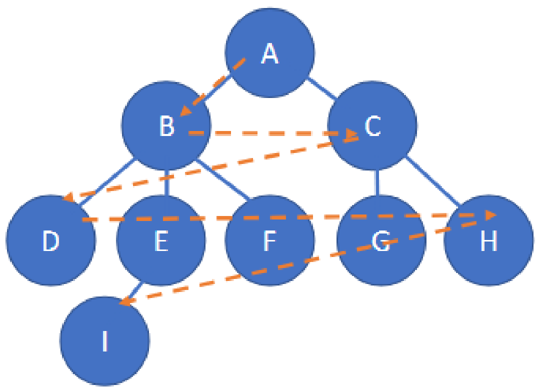
3.2.3 Depth-First Search (DFS)

Depth-First Search starts at the root and continues on one branch until it finds a solution or failure. In case of failure it backtracks to the current parent node and continues on the next branch.

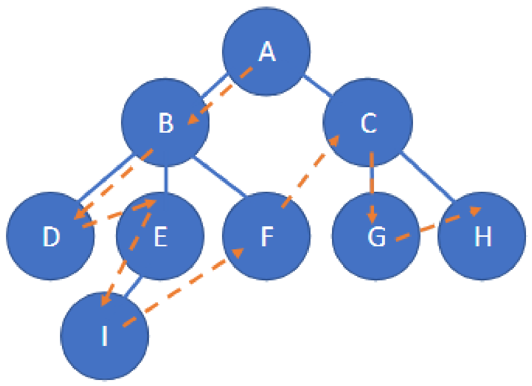
Unlike UCS and BFS DFS is not complete; It fails if the search space is of infinite depth or has loops.

It's also not optimal; More costly solutions may be found before less costly ones.

Complexity: $O(b^m)$ with b being the branching factor and m being the maximum depth of the search tree



BFS: A,B,C,D,E,F,G,H,I



DFS: A,B,D,E,I,F,C,G,H

Criteria	**BFS**	**DFS**
Concept	Traversing tree level by level	Traversing tree sub-tree by sub-tree
Data Structure (Queue)	First In First Out (FIFO)	Last In First Out (LIFO)
Time Complexity	O(Vertices + Edges)	O(Vertices + Edges)
Backtracking	No	Yes
Memory	Requires more memory	Less nodes are stored normally (less memory)
Optimality	Yes	Not without modification
Speed	In most cases slower compared to DFS	In most cases faster compared to BFS
When to use	If the target is relatively close to the root node	If the goal state is relatively deep in the tree

Comparison between BFS and DFS algorithms.

3.2.4 Depth-Limited Search (DLS)

DLS is a variant of DFS. Hereby the search is limited to a depth of d. This means that no infinite search can occur, the trade-off being that it might not find all solutions.

Example Depth-Limited Search

```
Function depth_limited_search(problem, limit):
    recursive_dls(make_node(initial_state(problem)), problem, limit)
Function recursive_dls(node, problem, limit):
    cutoff_occured = false;
    If goal_test(problem, state(node)) then
        return node;
    Else If depth(node) >= limit then
        return cutoff;
    Else
        ForEach successor in expand(node, problem) do
            result = recursive_dls(successor, problem, limit);
            If result == cutoff then
                cutoff_occured = true;
            Else If result != failure then
                return result;
    If cutoff_occured then
        return cutoff;
    Else
        return failure
```

This, of course, is neither complete nor optimal.

Complexity: $O(b^l)$ with b being the branching factor and l being the maximum depth of the search tree

3.2.5 Iterative Deepening Search (IDS)

IDS is a variant of DLS. It works like DLS, but instead of a fixed depth, it iteratively increases the depth until a solution is found.

Example Iterative Deepening Search

```
Function iterative_deepening_search(problem):
    For depth = 0 to ∞ do
        result = depth_limited_search(problem, depth);
        If result != cutoff then
            return result
```

This change makes it complete and optimal.

Complexity: $O(b^m)$ with b being the branching factor and d being the depth of the search tree. This makes its time complexity equal to BFS. However, the space complexity of IDS is $O(bd)$, which is much better than BFS' $O(b^d)$.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{1+\lfloor OptCost/eps \rfloor}$	b^m	b^l	b^d
Space	b^d	$b^{1+\lfloor OptCost/eps \rfloor}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Comparison of search strategies.

3.3 Heuristics h

Denotes a "rule of thumb", a rule that may be helpful in solving a problem.
In Tree-Search, a heuristic is a function that estimates the remaining cost from the current node to the goal.
Can go wrong!

Admissible Heuristic

A heuristic is admissible if it never overestimates the actual cost to reach the goal.

$$\boxed{h(n) \leq h^*(n)}$$

if $h^*(n)$ is the true cost to reach the goal from node n

Consistent Heuristic

A heuristic is consistent if for every node n and successor n' generated by any action a:

$$\boxed{h(n) \leq c(n, a, n') + h(n')}$$

if $c(n, a, n')$ is the cost of the action a from n to n'
Thus, a heuristic is consistent if, when going from neighboring nodes, the heuristic difference / step cost never overestimates the actual cost.

Lemmas

- If a heuristic is consistent, it is also admissible.
- If $h(n)$ is consistent, then the values of $f(n)$ on any path are non-decreasing.

3.3.1 Relaxed Problems

A relaxed problem is a problem that has fewer constraints on the actions than the original problem.

The cost of the optimal solution to a relaxed problem is an admissible heuristic for the original problem.

Example: Relaxed Problem as Admissible Heuristic
Consider a grid-based pathfinding problem where certain movements are restricted due to obstacles. A relaxed version of this problem might allow movement through obstacles by ignoring some constraints. The cost of the optimal solution to this relaxed problem, which is typically lower or equal to the original problem's optimal cost, serves as an admissible heuristic for the original problem.

This means that looking for relaxed problems is a good way to find admissible heuristics.

3.3.2 Dominance

A heuristic h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all nodes n. (Given that h_1 and h_2 are admissible)

This means that a dominant heuristic is always closer to the optimal heuristic h*, which results in less expansion and thus more efficient search.

3.3.3 Combining Heuristics

If h_1, h_2, ...h_m are admissible heuristics, then $h(n) = max(h_1(n), h_2(n), \ldots, h_m(n))$ is also admissible and dominates both h_1 and h_2.

This is useful if we have multiple admissible non-dominated heuristics and want to combine them to get a better heuristic.

3.4 Informed Search

Definition

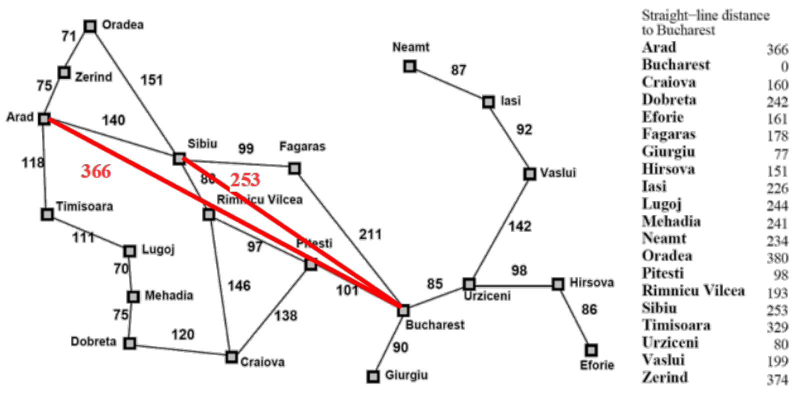
The informed search strategy has additional knowledge about "where" to look for solutions, usually in form of a heuristic function.
Some algorithms that utilize informed search are:

- Greedy Best-First Search
- A* Search
- Memory-Bounded Heuristic Search

3.4.1 Greedy Best-First Search

Greedy Best-First Search (GBFS) is an informed search algorithm that selects the next node to expand based on a heuristic that estimates the cost from the current node to the goal.

One example of a heuristic is the straight-line distance to the goal.



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example of GBFS with a straight-line distance heuristic.

Complexity: The time complexity of Greedy Best-First Search is $O(b^m)$, and its space complexity is also $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree. This is the same as DFS, however due to the heuristic it can be significantly faster.

GBFS is not guaranteed to be optimal or complete, especially if the heuristic function is not admissible or consistent. Its efficiency heavily depends on the quality of the heuristic used.

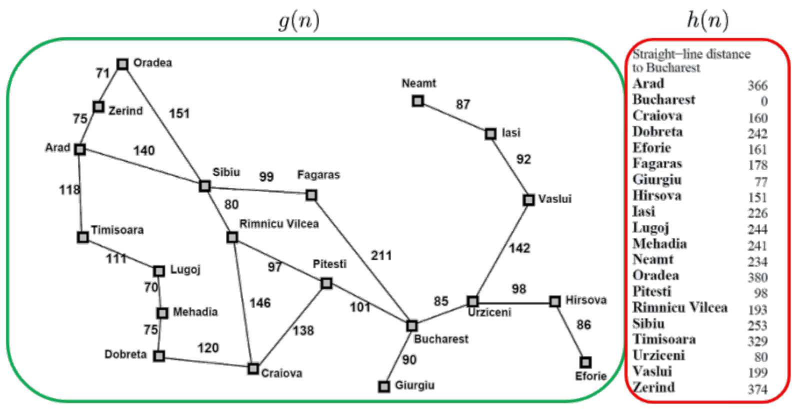
- **Complete?** No, can get stuck in loops.
- **Optimal?** No, does not guarantee an optimal solution as it depends on the heuristic, which may be flawed.

3.4.2 A* Search

Definition

A* Search is an informed search algorithm that combines the strengths of Uniform-Cost Search and Greedy Best-First Search. It searches for the least-cost path to the goal by considering both the cost to reach the current node and an estimated cost to reach the goal.

A* Search tries to avoid paths that are already expensive. It evaluates the complete path cost and the remaining cost to the goal. Its cost function is defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach the current node (root to n) and $h(n)$ (the heuristic) is the estimated cost to reach the goal (n to goal).



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example of A Search with heuristic function.*

Complexity: The time complexity of A* Search is $O(b^m)$, and its space complexity is also $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.

Completeness and Optimality: A* Search is complete, and can be optimal, if the heuristic is admissible.

3.4.3 Alternatives to A*

1. **Iterative-Deepening A* (IDA*)**
    - Like iterative deepening, it explores nodes level by level, but uses A* to evaluate the nodes.
    - Cutoff information is the f-cost (g + h) instead of the depth.
2. **Recursive Best-First Search (RBFS)**
    - Recursive algorithm that mimics best-first search with linear space
    - Keeps track of f-value of best alternative path
    - Path available from any ancestor of the current node and heuristic evaluations are updated with results of successors
3. **(Simple) Memory-Bounded A* ((S)MA*)**
    - Drops the worst leaf node when memory is full
    - Its value will be updated to its parent
    - May be researched later

3.4.4 Graph Search

When traversing a problem, loops can occur. Failure to detect them can turn linear search into exponential search. To avoid this, we can use **graph search**. Hereby we only expand nodes that have not been visited yet.

For example: The graph search version of UCS is **Dijkstra's algorithm**.

4 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.

4.1 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.

4.1.1 Terminology

Convergence

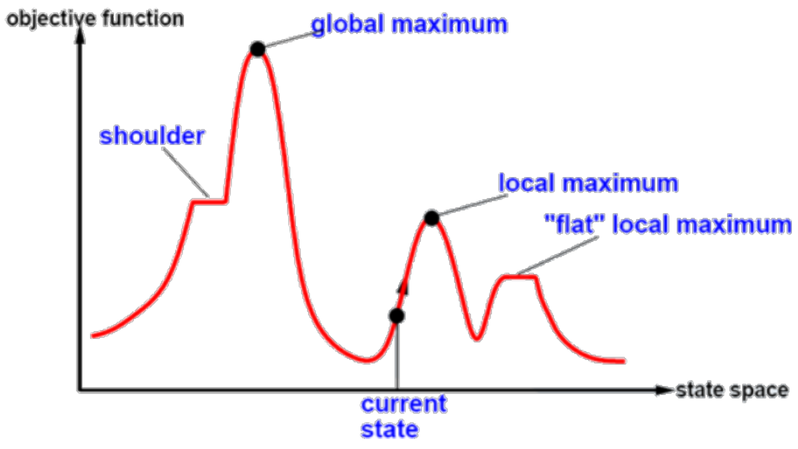
Describes a sequence of (function) values that approach a limit / value more and more.

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



4.2 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

4.2.1 Hill Climbing

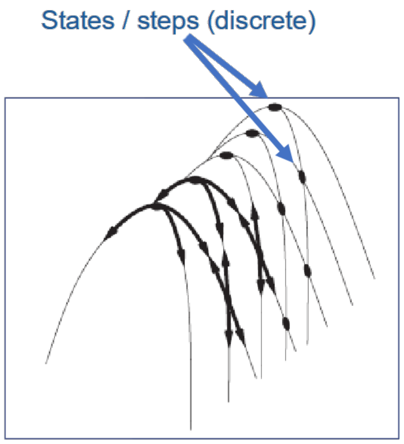
Basic Idea:

- 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) ≤ 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.

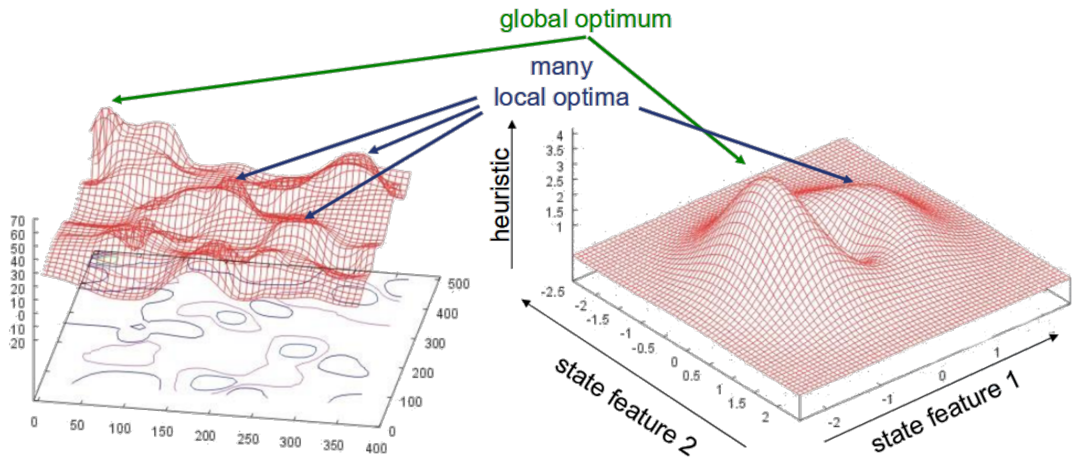


Problem - Local Optima:

- 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
  - → 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

4.2.2 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.

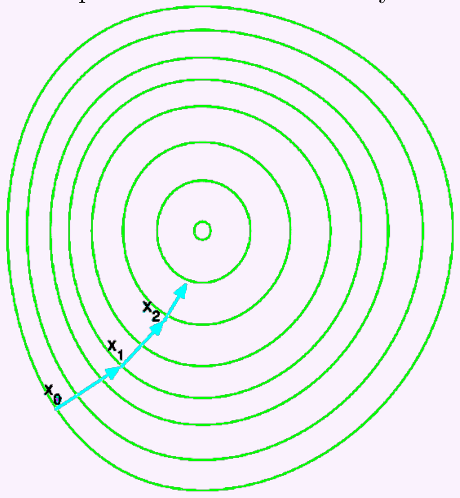
Gradient Descent: Working Principle

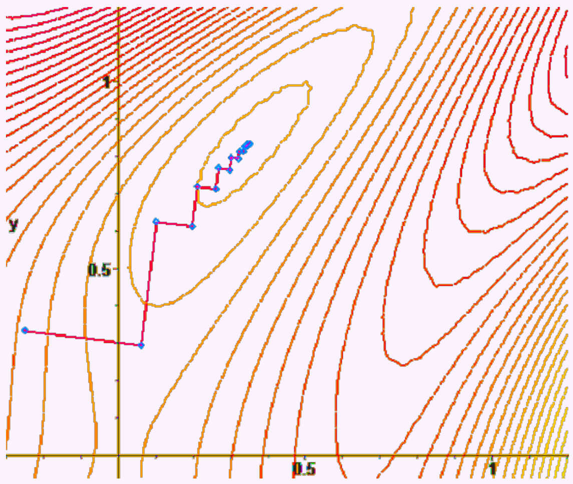
Gradient Vector **Cost Function**

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

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

1. Compute gradient: $\boxed{\frac{\partial}{\partial x_i} J(x_1, x_2, \dots, x_n) \quad \forall i \in n}$
2. Take a step downhill in the direction of the gradient: $\boxed{x_i\prime = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, x_2, \dots, x_n)}$
3. If $\boxed{J(x_1\prime, x_2\prime, \dots, x_n\prime) < J(x_1, x_2, \dots, x_n)}$, 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

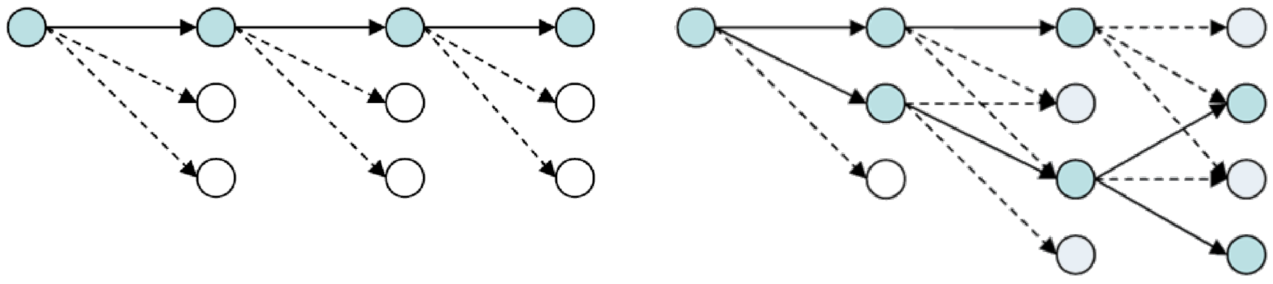
Gradient Descent in general works well for ”smooth” spaces; poorly for ”rough” spaces.

4.2.3 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



4.2.4 Simulated Annealing

Basic Idea:

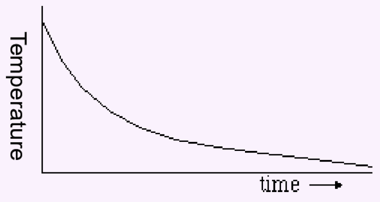
- Use hill-climbing, but occasionally take a step into a direction that does not show improvement
- Reduce the probability of a down-hill step and decrease the size of the step as the number of iterations grows
- Allows some ”bad moves” to escape local optima

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.

4.3 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.

4.3.1 Games

Zero-Sum Game

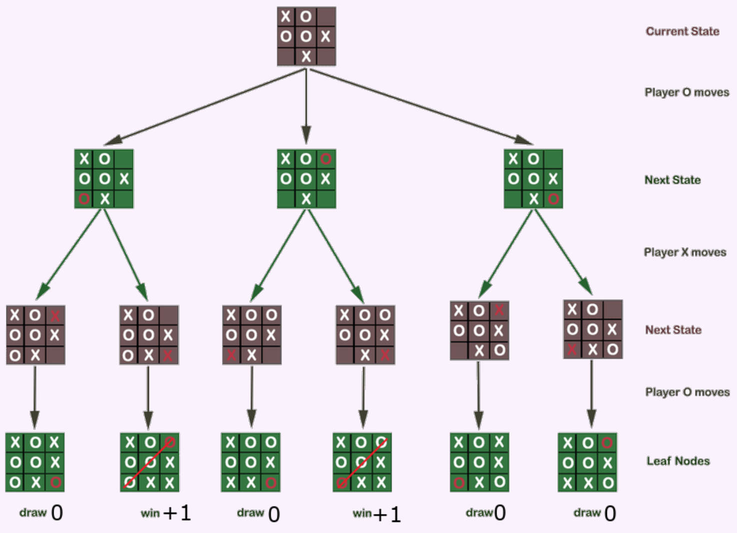
Describes a game whereby, if one party loses, the other must win, therefore the net change in "points" is zero.

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).



4.3.2 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

4.3.3 Minimax Algorithm

Basic Idea:

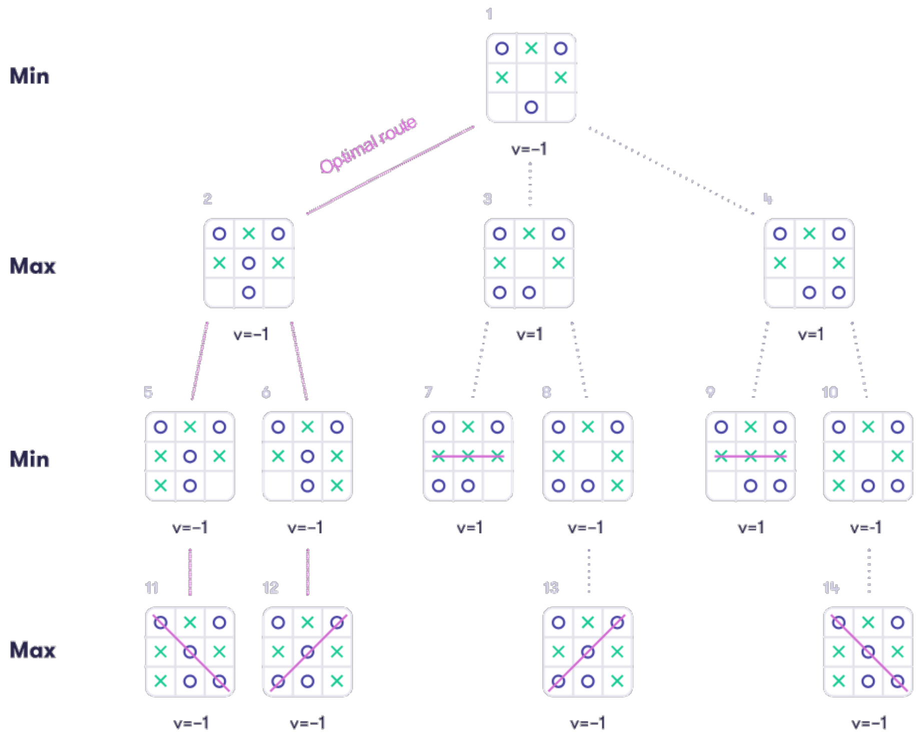
- Build a game tree where nodes represent the states of the game and edges represent the possible moves.
- The players:
  - **MIN:** Decreases the chances of **MAX** winning (Opponent)
  - **MAX:** Increases his own chances of winning (Agent)
- Players take alternating turns following their respective 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.

4.3.4 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

5 Constraint Satisfaction Problems

Constraint Satisfaction Problem

Constraint Satisfaction is a technique where a problem is solved when its solution satisfies certain constraints or rules of the problem.

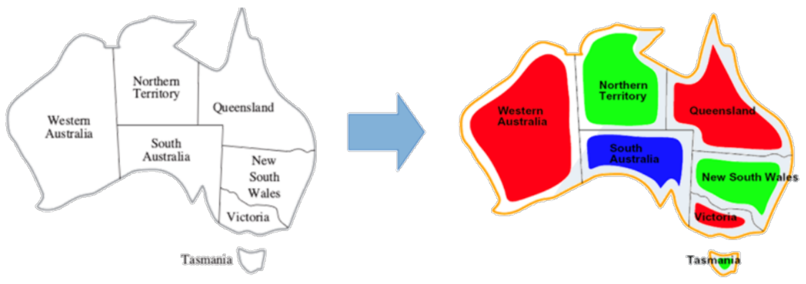
Components:

- A state, defined by **variables** X_i with **d** values from **domain** D_i
- A goal test, defined as a set of **constraints** C specifying allowable combinations of values for subsets of variables.

Solving Constraint Satisfaction Problems:

- A state space
- Notion of the solution

Example of a Constraint Satisfaction Problem



Problem: Assign each territory a colour such that no two adjacent territories have the same colour.

Variables: $X = \{WA, NT, Q, NSW, V, SA, T\}$
Domain of Variables: $D = \{red, green, blue\}$
Constraints: $C = \{SA \neq WA, SA \neq NT, SA \neq Q, \dots\}$

5.1 Assignment of Values to Variables

A state in state-space is not a ”blackbox” as in standard search, but defined by assigning values to some or all variables.

$$X_1 = v_1, X_2 = v_2, \dots, X_d = v_d$$

The assignment of these values can be done in different ways:

1. **Consistent/Legal Assignment**: An assignment is consistent if it satisfies all constraints or rules.
2. **Complete Assignment:** An assignment is complete if every variable is assigned a value, and the solution to the CSP remains consistent.
3. **Partial Assignment:** An assignment is partial if some variables are not assigned values.

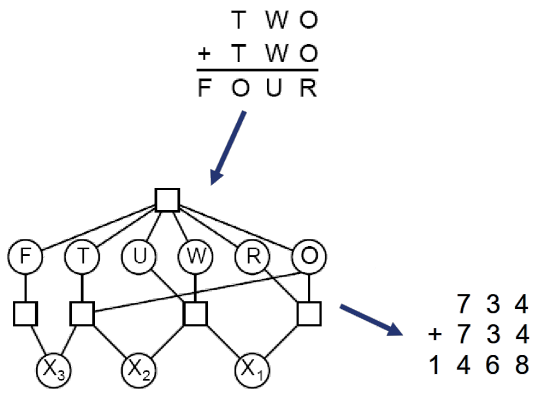
5.2 Constraint Graphs

Constraint Graphs are often constructed because abstraction of the problem makes it easier to solve and understand.

A constraint graph is usually denoted with

- Every variable is represented by a node
- Every edge indicates a constraint between two variables

Example of a Constraint Graph



Problem: Assign uniques value to the variables of each letter, so that resulting equation is true.

Variables: $X = \{T, W, O, F, U, R\}$
Domain of Variables: $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints: $C = \{T \neq W \neq O \neq F \neq U \neq R\}$
$\cup$ $\boxed{\texttt{int}(''T'' +'' W'' +'' O'') + \texttt{int}(''T'' +'' W'' +'' O'') = \texttt{int}(''F'' +'' O'' +'' U'' +'' R'')}$

Here the connected nodes are involved in (in-)equations:

$$\begin{aligned} 2 \cdot O &= 10 \cdot X_1 + R \\ 2 \cdot W + X_1 &= 10 \cdot X_2 + U \\ 2 \cdot T + X_2 &= 10 \cdot X_3 + O \\ F &= X_3 \\ T \neq W \neq O &\neq F \neq U \neq R \end{aligned}$$

5.3 Types of Constraints

- **Unary constraints:** Involve a single variable (e.g. South Australia $\neq$ green)
- **Binary constraints:** Involve two variables (e.g. South Australia $\neq$ Wester Australia)
- **Higher-order constraints:** Involve more than two variables (e.g. $2 \cdot W + X_1 = 10 \cdot X_2 + U$)

Preferences / Soft constraints:

- Not binding, but should be considered during search

  $\rightarrow$ Constrained optimization problems

- e.g. Red is better than green

5.4 Solving CSPs: Search

Basic Idea:

1. Successively assign values to variable
2. Check constraints
3. If constraint is violated $\rightarrow$ backtrack
4. Repeat until all variables have assigned values that satisfy constraints

To do this we map CSPs into search problems:

- Nodes = assignments of values to a subset of the variables
- Neighbors of a node = nodes in which values are assigned to one additional variable
- Start node = empty assignment
- Goal node = a node which assigns a value to each variable and satisfies all constraints

5.4.1 Naive Search

Naive Search is practically a **brute-force** method. It systematically explores all possible assignments of v values to n variables. This, of course, is incredibly innefficient and results in exponential time complexity. The number of leaves in the search tree grows with $n!v^n$.

5.4.2 Backtracking Search

Basic Idea: As assignments are commutative ([WA = red then NT = green] = [Nt = green then WA = red]) we can reduce the number of leaves in the search tree by only considering nodes that have not been visited before.

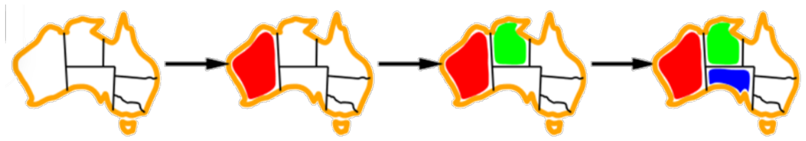
Backtracking Algorithm

```
Function backtrack_search(csp):
    return recursive_backtrack(csp, {});
Function recursive_backtrack(csp, assignment):
    If is_complete(assignment) then
        return assignment;
    var = get_unassigned_variable(get_variables(csp), assignment, csp);
    ForEach value in order_domain_values(var, assignment, csp) do
        If value is consistent with assignment given constraints(csp) then
            assignment.add({var = value});
            result = recursive_backtrack(csp, assignment);
            If result != failure then
                return result;
            assignment.remove({var = value});
    return failure;
```

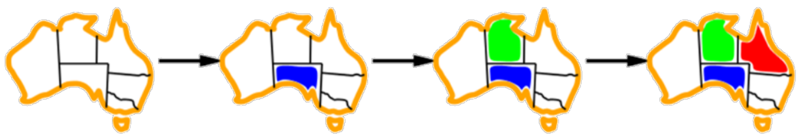
This is still not ideal as in the worst case the complexity is still exponential. This can be improved by including heuristics.

5.5 Heuristics for CSPs

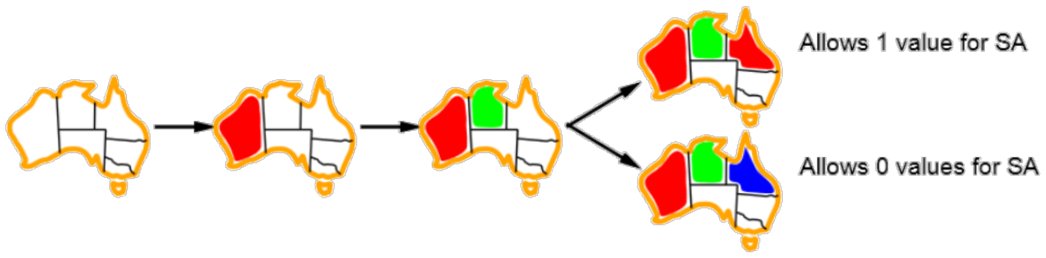
- **Domain-Specific Heuristics:** Depend on the particular characteristics of the problem.
- **General-Purpose Heuristics:** Can work on any CSP.
  - **Minimum Remaining Value:** Choose variable with fewest consistent values



  - **Degree Heuristic:** Choose variable with the most constraints on remaining variables



  - **Least Constraining Value Heuristic:** Given a variable, choose the value that rules out the fewest values in the remaining variables



If utilized in this order, these heuristics will greatly improve search speed.

5.6 Constraint Propagation

Node Consistency

A variable (node) is consistent if the possible values of this variable are conform to all unary constraints.

Local Consistency

Local consistency is defined by a graph where each of its nodes is consistent with its neighbors. This is done by iteratively enforcing the constraint corresponding to the edges.

Arc

A constraint involving two variables is called an **arc** or binary constraint.

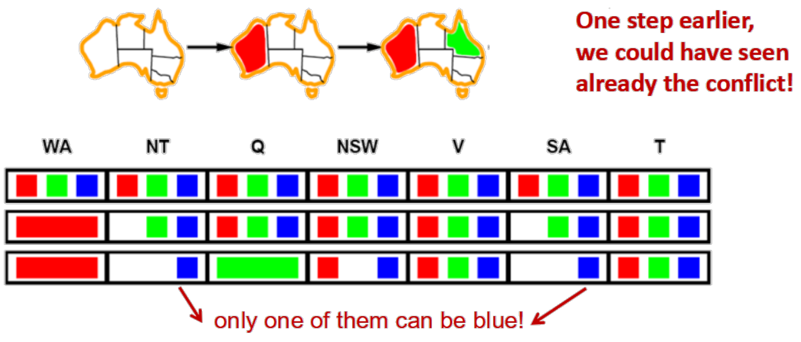
Arc Consistency

An arc is consistent if for each value of X in the domain of X there exists a value Y in the domain of Y such that the constraint $\text{arc}(X, Y)$ is satisfied.

$$\forall X \in \text{dom}(X), \exists Y \in \text{dom}(Y) : \text{arc}(X, Y) \text{ is satisfied}$$

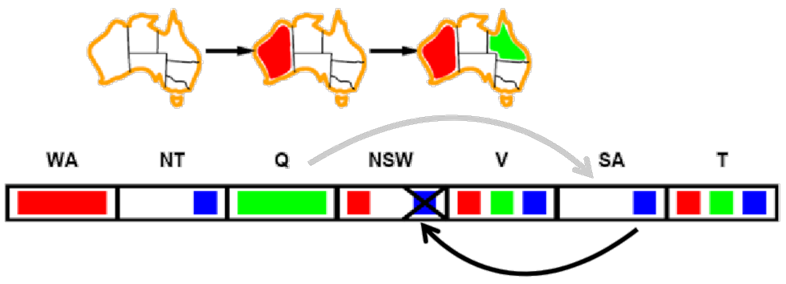
5.6.1 Forward Checking

Basic Idea: Keep track of remaining legal values for unassigned variables and terminate search, when any variable has no legal values left.

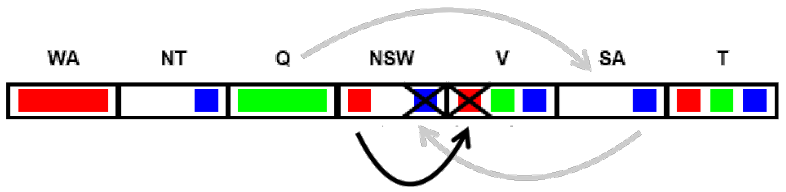


5.6.2 Maintainting Arc Consistency (MAC)

After each assignment of a value to a variable, possible values of the neighbors have to be updated.



If one variable (NSW) losses a value (blue), we need to recheck its neighbors as well because they might have lost a possible value.



AC-3 Algorithm

```
Function AC-3($csp$):
    queue = get_all_arcs($csp$);
    While $queue$ is not empty do
        $(X_i, X_j)$ = remove_first($queue$);
        If remove_inconsistent_values($X_i$, $X_j$) then
            ForEach $X_k$ in get_neighbors($X_i$) do
                queue.add($X_k$);

Function remove_inconsistent_values($X_i$, $X_j$):
    removed = false;
    ForEach $x$ in get_domain($X_i$) do
        If no value $y$ in get_domain($X_j$) satisfies arc($X_i, X_j$) then
            get_domain($X_i$).remove($x$);
            removed = true;
    return removed;
```

5.6.3 Path Consistency

Arc consistency is often sufficient to:

- Solve the problem (all variable domains reduced to one value)
- Show that the problem cannot be solved (some domains empty)

but sometimes may not be enough, for example if theres always a consistent value in the neighboring region.

Path consistency tightens the binary constraint by considering triples of values.

A pair of variables (X_i, X_j) is path-consistent with X_m if

- for every assignment that satisfies the constraint on the arc (X_i, X_j)
- there is an assignment that satisfies the constraints on the arcs (X_i, X_m) and (X_j, X_m)

5.6.4 k-Consistency

k-Consistency is a generalization of path consistency. A set of k values need to be consistent. It may lead to a faster solution but checking for k-consistency is computationally expensive with exponential time in the worst case.

In practice, arc consistency is most frequently used.

5.6.5 Constrain Propagation & Backtracking Search

Basic Idea: Each time a variable is assigned, a constraint propagation algorithm is run in order to reduce the number of choice points in the search. This can improve the speed of backtracking search further. This algorithm can be implemented using Forward Checking or AC-3.

5.7 Local Search for CSPs

Neccessary Modifications for CSPs:
- work with complete states
- allow states with unsatisfied constraints
- operators reassign variable values

Min-Conflicts Heuristic:
- Randomly select a conflicted variable
- Choose the value that violates fewest constraints
- Hill climbing with h(n) = # of violated constraints

Performance:

- Can solve randomly generated CSPs with a high probability
- Except in a narrow range $R = \dfrac{\text{\# of constraints}}{\text{\# of variables}}$

5.8 Problem Decomposition

Assume search space for a constraint satisfaction with **n variables**, each of which can have **d values** $= O(d^n)$

Basic Idea: Decompose the problem into subproblems with **c variables** each

- Each problem has complexity $O(d^c)$
- There are n/c problems → total complexity $O(n/c \cdot d^c)$
- Unconditional independence is rare

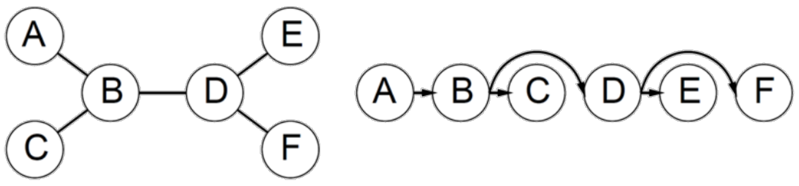
This can reduce the total complexity from exponential to linear, assuming c is constant.

5.9 Tree-Structured CSPs

A CSP is tree-structured if in the constraint graph any two variables are connected by a single path. Any tree structured CSP can be solved in linear time in the number of variables $= O(n \cdot d^2)$

5.9.1 Linear Algorithm

1. Choose variable as root, order nodes so that parent always comes before its children (only one parent per node)
2. For j = n downto 2
   - Make the arc(X_i, X_j) arc-consistent, calling `remove_inconsistent_value(`X_i, X_j`)`
3. For i = 1 to n
   - Assign to X_i any value that is consistent with its parent



5.9.2 Nearly Tree-Structured Problems

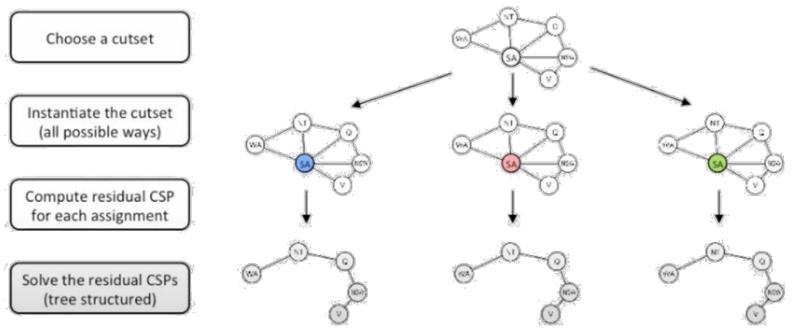
Tree structured problems are rare.

Approaches for making them tree-structured:

1. **Cutset Conditioning:**
   - Removing nodes so that the remaining nodes form a tree
2. **Collapsing nodes together:**
   - Decompose the graph into a set of independent tree-shaped subproblems

Cutset Conditioning

1. Choose a subset S of the variables such that the constraint graph becomes a tree after removal of S (cycle cutset)
2. Choose a consistent assignment of variables for S
3. Remove from the remaining variables all values that are inconsistent with the variables of S
4. Solve the CSP problem with the remainign variables
5. If no solution: Choose different S in **2**


Choose a cutset
Instantiate the cutset (all possible ways)
Compute residual CSP for each assignment
Solve the residual CSPs (tree structured)
WA
NT
Q
SA
NSW
V