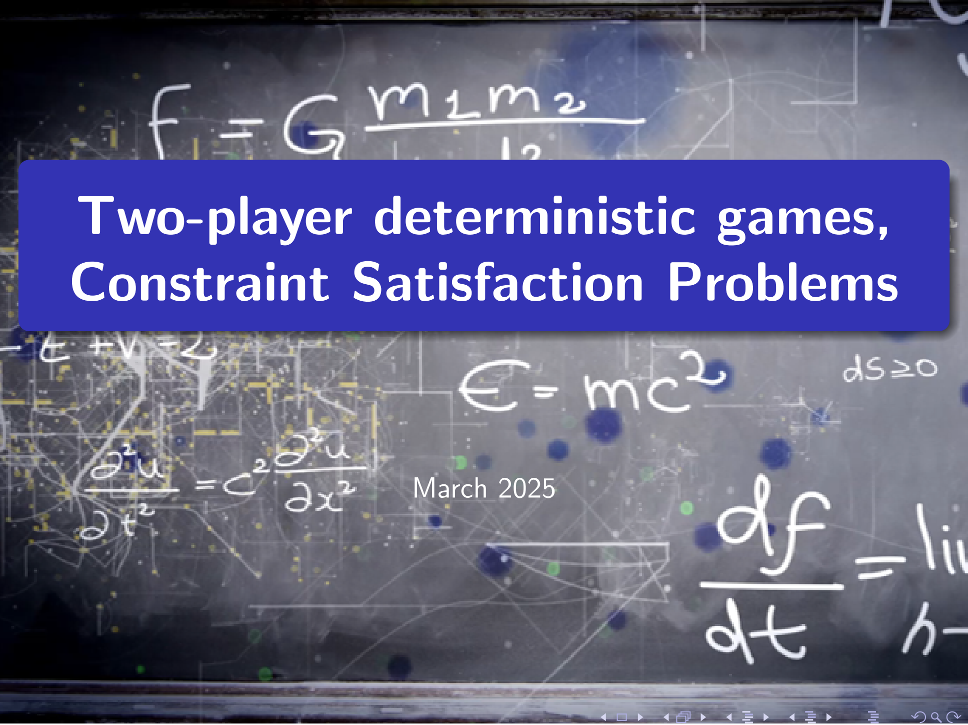
Two-player deterministic games, Constraint Satisfaction Problems

March 2025


$f = G\frac{m_1 m_2}{r^2}$
$E = mc^2$
$ds \geq 0$
$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$
$\frac{df}{dt} = \lim$

Topics to be discussed

Part I. Introduction

1. Introduction to Artificial intelligence

Part II. Search and optimisation.

2. Search - basic approaches
3. Search - optimisation
4. Two-player deterministic games
5. Evolutionary and genetic algorithms

Part III. Machine learning and data analysis.

6. Regression, classification and clustering (Part I & II)
8. Artificial neural networks
9. Bayesian models
10. Reinforcement Learning

Part IV. Logic, Inference, Knowledge Representation

11. Propositional logic and predicate logic
12. Knowledge Representation

Part V. AI in Action: Language, Vision

13. AI in Natural language processing
14. AI in Vision and Graphics

Part VI. Summary

15. AI engineering, Explainable AI, Ethics,

Introduction to Adversarial Search and Constraint Satisfaction Search

- Overview of Previous Lectures
- Adversarial Search (two-person deterministic games)
    - Exhaustive Search Algorithm
    - Minimax Algorithm
    - Alpha-Beta Pruning
    - Other Variants (e.g. Monte Carlo Tree Search)
- Constraint Satisfaction Search
    - CSPs as a Search Problem
    - Constraint Propagation and Local Search
    - Backtracking Search and Improvements
    - Solving CSPs with SAT Solvers
    - Hybrid Approaches
- Applications of Adversarial Search and CSPs in AI

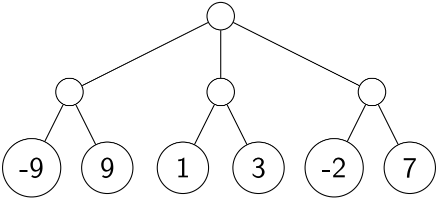
Introduction to Adversarial Search

- Adversarial search is a type of search problem where an agent (or player) must make decisions to reach a goal, while also taking into account the actions of an opponent.
- This is often used in games, where one player's actions affect the other player's options, and the goal is to win the game.
- Adversarial search can be modeled as a tree, with each level representing one player's turn.
- The Minimax algorithm is a popular algorithm for solving adversarial search problems, which works by choosing the move that maximizes the minimum possible gain (or minimizes the maximum possible loss).
- Example games that can be modeled as adversarial search problems include Chess, Checkers, Go, and Tic-Tac-Toe.

Game Tree - introduction

Key idea: game tree

Each node is a decision point for a player.
Each root-to-leaf path is a possible outcome of the game.



Introduction to Adversarial Search Trees

Definition

An adversarial search tree is a conceptual diagram used to represent the decision-making process where multiple players with opposing goals compete.

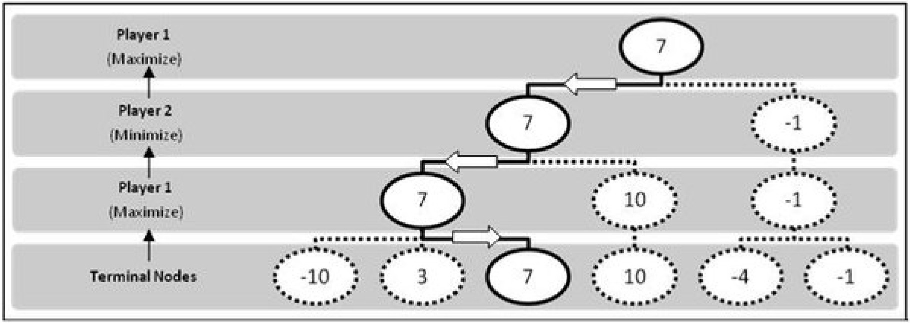
Basic Concepts

- **Nodes:** Decision points where players take actions.
- **Edges:** Connections between nodes that represent player moves.
- **Root Node:** The initial state of the game.
- **Leaf Nodes:** Terminal states with game outcomes (win, lose, draw).

Explanation

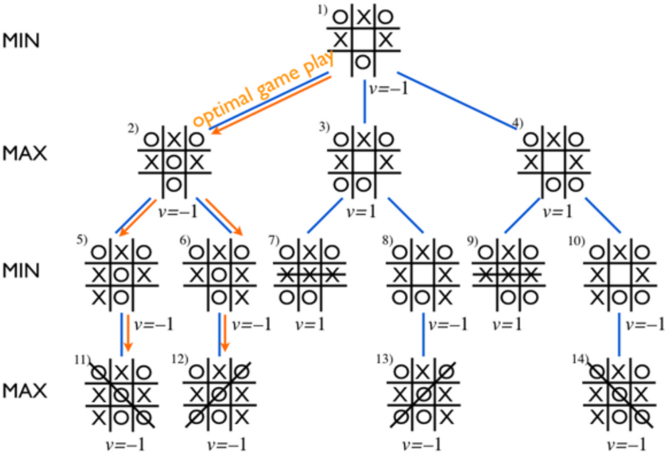
The tree is constructed with each layer representing a turn in the game. Players alternate turns, expanding the tree with possible moves. The goal is to use this tree to determine the optimal strategy by considering all potential responses from the adversary.

Adversarial Search Tree Example



https://materiaalit.github.io/intro-to-ai-17/part2/

Tic Tac Example



https://en.wikibooks.org/wiki/Artificial Intelligence/Search/Adversarial$_s$*earch/Minimax* Search

Definition of a Deterministic, Complete Information Game

Let S be a set of possible game states, P be a set of players, $s_0 \in S$ be the initial state, $T \subseteq S$ be the set of terminal states, and $v : T \rightarrow \mathbb{R}^{|P|}$ be a function that assigns a real-valued payoff vector to each terminal state. The tuple (S, P, s_0, T, v) defines a deterministic, complete information game.

- **State**: $s \in S$
- **Successor Function**: $succ : S \times P \rightarrow 2^S$ such that $succ(s, p) \subseteq S$ is the set of states that can be reached from s by player p making a legal move.
- **Initial Position**: $s_0 \in S$
- **Terminal States**: $T \subseteq S$
- **Utility Function**: $v : T \rightarrow \mathbb{R}^{|P|}$ assigns a real-valued payoff vector to each terminal state, where $|P|$ is the number of players.

Exhaustive Algorithm for Two-Person Deterministic Games

Assumptions

- Two-person deterministic game with a finite number of possible moves and terminal states.
- Perfect information for both players, meaning complete knowledge of the game state and all past moves.

Key Ideas

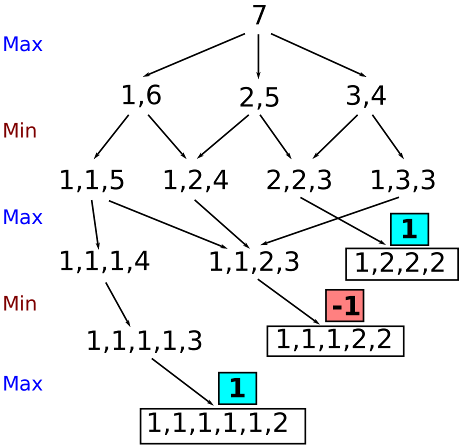
- The algorithm considers all possible moves from the current state and recursively evaluates the resulting game states until a terminal state is reached.
- At each non-terminal game state, the algorithm determines the best move for the current player based on the utility values of the resulting states.
- If the current player is the maximizing player, the algorithm chooses the move that leads to the state with the highest utility value.
- If the current player is the minimizing player, the algorithm chooses the move that leads to the state with the lowest utility value.
- The algorithm repeats this process until a terminal state is reached, and then returns the utility value of that state.
- The exhaustive search algorithm is guaranteed to find the optimal move for the current player, but can be computationally expensive for large game states.

Exhaustive Search Algorithm

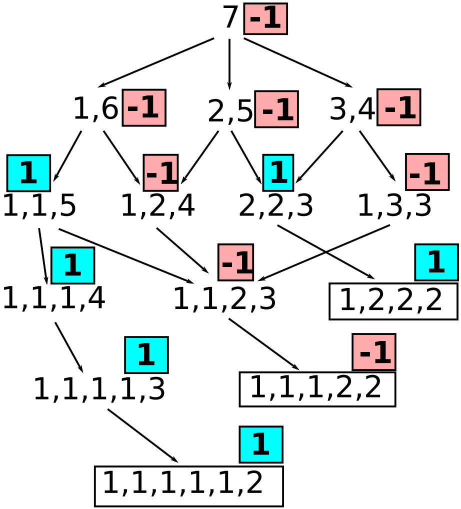
Algorithm Exhaustive Search Algorithm

```
Input: state, player
Output: best-value
Function exhaustive-search(state, player):
    if state is a terminal state then
        return the utility value of state
    end
    if player is the maximizing player then
        best-value ← $-\infty$
        for each possible move m in state do
            value ← exhaustive-search(resulting-state(state, m), opponent)
            best-value ← max(best-value, value)
        end
    end
    else
        best-value ← $+\infty$
        for each possible move m in state do
            value ← exhaustive-search(resulting-state(state, m), maximizing-player)
            best-value ← min(best-value, value)
        end
    end
    return best-value
```

NIM game


7
Max
1,6
2,5
3,4
Min
1,1,5
1,2,4
2,2,3
1,3,3
Max
1
1,1,1,4
1,1,2,3
1,2,2,2
Min
-1
1,1,1,1,3
1,1,1,2,2
Max
1
1,1,1,1,1,2

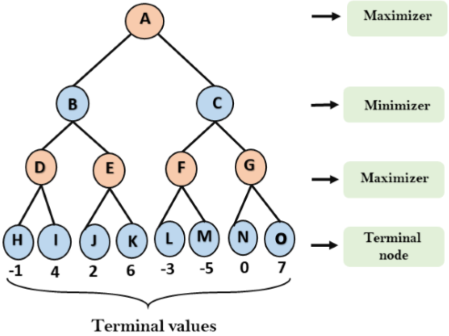
NIM game cont.

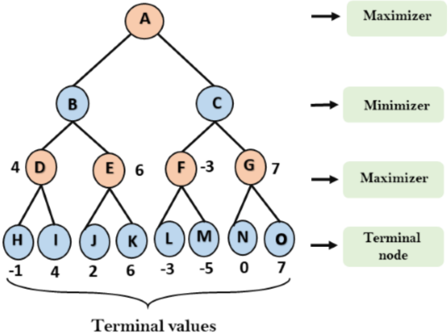


Min-Max Algorithm

- **The Min-Max algorithm** considers all possible moves that each player can make, up to a certain depth in the game tree.
- The algorithm assumes that each player will choose the move that results in the highest or lowest possible payoff, depending on whether they are the maximizing or minimizing player.
- The **evaluation function** used to assign scores to nodes in the game tree is crucial to the success of the algorithm.
- The Min-Max algorithm can be extended to handle games with more than two players, although the complexity of the algorithm increases significantly.
- The algorithm can be optimized using techniques such as alpha-beta pruning, which reduces the number of nodes that need to be evaluated.

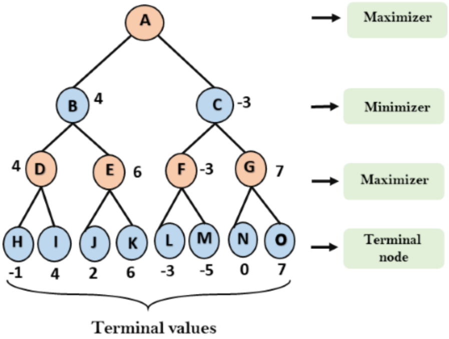
Min-Max Algorithm - example





https://www.javatpoint.com/mini-max-algorithm-in-ai

Min-Max Algorithm - example

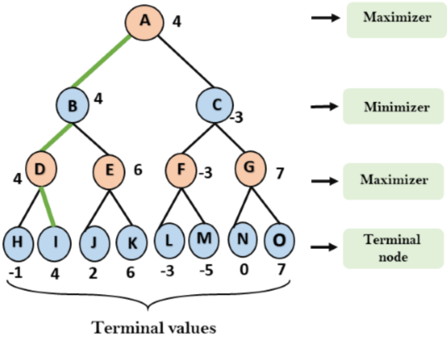


Maximizer

Minimizer

Maximizer

Terminal node



Maximizer

Minimizer

Maximizer

Terminal node

https://www.javatpoint.com/mini-max-algorithm-in-ai

Min-Max Algorithm

Algorithm Min-Max

```
Input: position, depth, isMax
Output: bestValue
Procedure MinMax(position, depth, isMax):
    if $depth = 0$ or position is terminal then
        return the value of position;
    end
    if isMax then
        bestValue $\leftarrow -\infty$; for move in possible moves from position do
            value $\leftarrow$ MinMax(position after move, depth-1, False); bestValue $\leftarrow$ max(bestValue, value);
        end
    end
    else
        bestValue $\leftarrow \infty$; for move in possible moves from position do
            value $\leftarrow$ MinMax(position after move, depth-1, True); bestValue $\leftarrow$ min(bestValue, value);
        end
    end
    return bestValue;
end
```

Limited Depth Search

Reminder

A strategy that limits the exploration of the game tree to a pre-defined depth, using an evaluation function at this terminal depth to estimate the game state's value.

Key Points

- Reduces computational demands by avoiding full tree exploration.
- Heuristic evaluation function approximates game outcomes at limited depth.
- Effectiveness hinges on appropriate depth selection and evaluation accuracy.

Advantages & Disadvantages

- Efficient computation but requires careful balance between depth and heuristic accuracy.

Iterative Deepening

Reminder

Employs repeated depth-limited searches with incrementally increasing depths, combining depth-first search's low memory footprint with breadth-first search's completeness.

Key Points

- Begins with minimal depth, increasing incrementally to find optimal solutions.
- Adaptable to time constraints, providing the best solution found within limits.
- Efficient cache usage due to frequent revisits of nodes near the root.

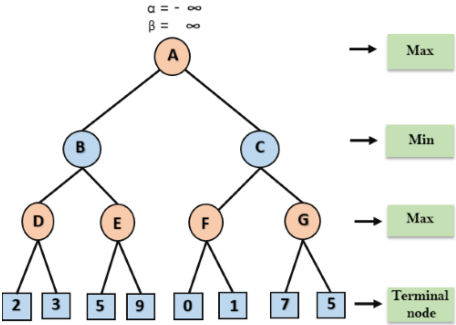
Advantages & Disadvantages

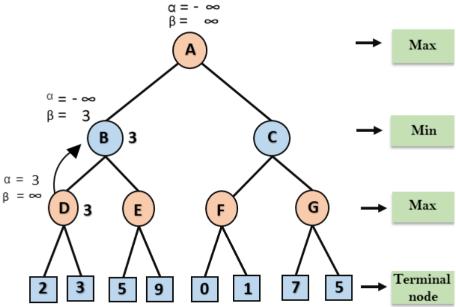
- Maximizes efficiency and flexibility, albeit with some redundant computations.

Alpha-Beta Pruning

- Alpha-Beta Pruning is a technique that improves the efficiency of the minimax algorithm for two-player games.
- The idea is to keep track of two values, α and β, which represent the best possible score for the maximizing player 1 and the minimizing player 2, respectively.
- The algorithm can prune (ignore) a branch if it finds a node where $\alpha \geq \beta$.
- By pruning branches that are irrelevant for the optimal decision, Alpha-Beta Pruning can significantly reduce the number of nodes that need to be evaluated by minimax.

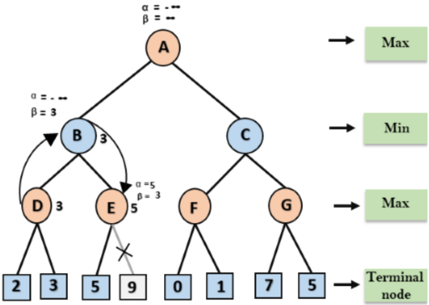
Alpha-beta-pruning - example

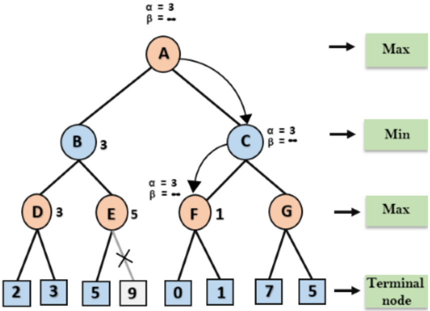




https://www.javatpoint.com/ai-alpha-beta-pruning

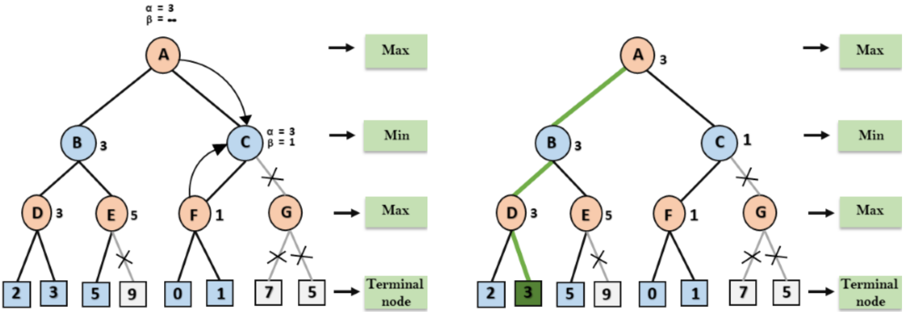
Alpha-beta-pruning- example cont.





https://www.javatpoint.com/ai-alpha-beta-pruning

Alpha-beta-pruning - example cont.



https://www.javatpoint.com/ai-alpha-beta-pruning

Alpha-Beta Pruning Algorithm

Algorithm Alpha-Beta Pruning

```
Function AlphaBeta(node, depth, $\alpha$, $\beta$, maximizingPlayer)
    if depth $= 0$ or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value $\leftarrow -\infty$
        for each child of node do
            value $\leftarrow$ max(value, AlphaBeta(child, depth - 1, $\alpha$, $\beta$, false))
            $\alpha \leftarrow$ max($\alpha$, value)
            if $\beta \leq \alpha$ then
                break
        return value
    else
        value $\leftarrow +\infty$
        for each child of node do
            value $\leftarrow$ min(value, AlphaBeta(child, depth - 1, $\alpha$, $\beta$, true))
            $\beta \leftarrow$ min($\beta$, value)
            if $\beta \leq \alpha$ then
                break
        return value
```

Heuristics for Two-Person Deterministic Games

In addition to basic game-playing algorithms like Minimax and Alpha-Beta pruning, there are several supporting methods or heuristics that can improve their performance:

- **Iterative Deepening**: An algorithm that repeatedly applies Minimax or Alpha-Beta with increasing depth limits, allowing for a more efficient use of time and resources. It can also provide useful information for move ordering and evaluation functions.
- **Move Ordering**: A technique that orders the game moves based on some heuristic, such as the likelihood of leading to a better outcome or the amount of pruning they enable. Common heuristics include killer moves (moves that have caused a cutoff in previous searches), history heuristics (moves that have been successful in the past), and static exchange evaluation (a technique that estimates the net material gain or loss of a capture sequence).
- **Transposition Tables**: A cache of previously visited states and their values, which can reduce the number of evaluations needed for similar states encountered later. In addition to storing the value of the state, transposition tables can also store the best move found so far and other search information.
- **Quiescence Search**: A procedure that extends Minimax or Alpha-Beta to consider only moves that leave the board in a quiet position, i.e., where there are no immediate captures or significant changes in material. Quiescence search can prevent the search from getting bogged down in noisy positions, where the evaluation function may be unreliable.
- **Evaluation Functions**: A heuristic function that assigns a score to a game state, allowing for the approximation of the true value function and guiding the search towards more promising moves. Evaluation functions can be based on a variety of features, such as material count, mobility, king safety, pawn structure, and piece placement. They are typically learned or hand-tuned for specific games and hardware.

These methods are often combined and tuned for specific games and hardware, and can significantly improve the performance of basic game-playing algorithms.

Key Ideas Behind Monte Carlo Method in Searching

Monte Carlo method is a family of statistical algorithms that use random sampling to solve computational problems. In game search, Monte Carlo methods are often used to approximate the value function or to guide the search towards more promising moves. Key ideas behind Monte Carlo method in searching:

- **Random Sampling**: Monte Carlo methods use random sampling to simulate many possible outcomes of the game, without explicitly searching the game tree. These outcomes are then used to estimate the value of the game state or move. The more samples that are taken, the more accurate the estimate becomes.
- **Simulation Policy**: The quality of the Monte Carlo estimate depends on the simulation policy, which determines how the random outcomes are generated. In game search, the simulation policy can be based on a variety of heuristics, such as random moves, greedy moves, or even neural networks. The simulation policy should be fast to execute and should capture the essential aspects of the game.
- **Exploration vs. Exploitation**: Monte Carlo methods balance the trade-off between exploration and exploitation, by randomly sampling both promising and unpromising moves. This allows the method to discover unexpected opportunities and to avoid getting stuck in local optima. The balance between exploration and exploitation can be controlled by adjusting the parameters of the simulation policy.
- **UCT Algorithm**: Upper Confidence Bounds for Trees (UCT) is a popular Monte Carlo search algorithm that uses a combination of random sampling and tree traversal to guide the search towards promising moves. UCT assigns each move a score that balances the estimated value and the uncertainty of the estimate, and selects the move with the highest score. UCT has been successful in a variety of games, including Go, Chess, and Poker.

Monte Carlo methods are a powerful and flexible tool for game search, and can be combined with other techniques such as tree search, deep learning, and domain-specific knowledge to achieve state-of-the-art performance.

How UCT Works

UCT employs a mathematical formula to select which node to explore next during the tree search phase. It calculates a value for each potential move (node) based on two components:

- **The win rate of the node:** This represents the *exploitation* part, favoring nodes with a high success rate in simulations.
- **The visit count of the node and its ancestors:** This represents the *exploration* part, giving a boost to less-visited nodes to ensure they are explored enough to accurately assess their value.

The UCT formula for a node j is given by:

$$\text{UCT}(j) = \frac{w_j}{n_j} + C\sqrt{\frac{\ln N_j}{n_j}}$$

Where:

- w_j is the number of wins after the j^{th} move.
- n_j is the number of simulations for the node j.
- N_j is the total number of simulations for the parent node.
- C is a constant that determines the balance between exploration and exploitation (higher values increase exploration).

Monte Carlo Tree Search (MCTS)

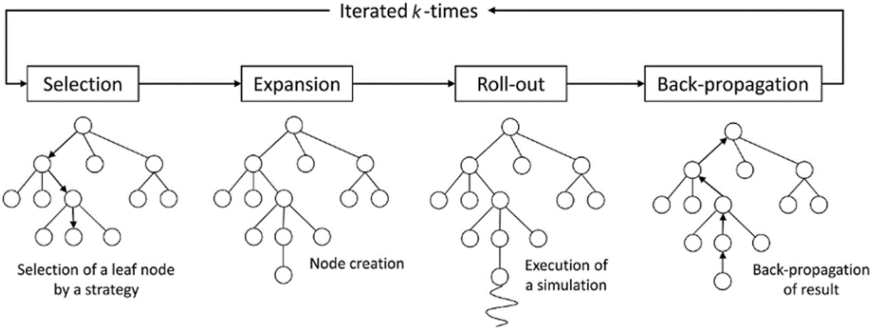
- MCTS is a search algorithm that aims to find the best move in a game
- It uses a tree structure to represent the game state and possible moves
- The algorithm consists of four steps:
  1. Selection: Choose a leaf node to expand based on a selection policy
  2. Expansion: Add one or more child nodes to the selected node
  3. Simulation: Play a simulated game from the new node to the end
  4. Backpropagation: Update the statistics of each node in the path from the selected node to the root based on the result of the simulation
- The algorithm repeats steps 1-4 until a stopping criterion is met
- The move with the highest average reward at the root node is selected as the best move

Monte Carlo Tree Search

```
Algorithm Monte Carlo Tree Search
Input: game state $s_0$, computational budget $C$
Output: best move $a^*$
Procedure MCTS($s_0$, $C$):
    Initialize empty search tree with root node containing $s_0$
    while computation budget not exhausted do
        $s_t \leftarrow$ selection phase starting at root node
        if $s_t$ is not terminal then
            $s_{t+1} \leftarrow$ expansion phase of $s_t$
            $v \leftarrow$ simulation phase of $s_{t+1}$
            backpropagation phase from $s_{t+1}$ to $s_0$ with value $v$

        end
    end
    $a^* \leftarrow$ best move from $s_0$ based on visit counts
    return $a^*$
end
```

Monte Carlo Tree Search Example


Iterated k-times
Selection
Expansion
Roll-out
Back-propagation
Selection of a leaf node by a strategy
Node creation
Execution of a simulation
Back-propagation of result

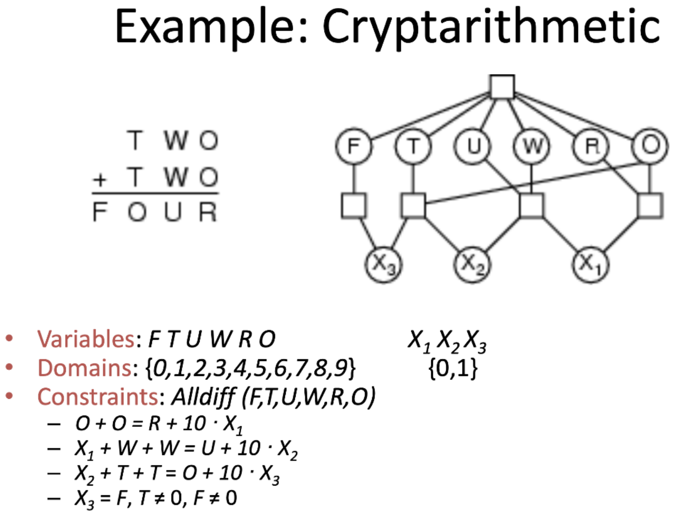
https://link.springer.com/content/pdf/10.1007/s00366-021-01338-2.pdf

CSPs as a Search Problem in AI

Constraint Satisfaction Problems (CSPs)

- A class of problems in AI that requires finding a solution that satisfies a set of constraints
- Can be modeled as a search problem where the goal is to find a solution that satisfies all constraints
- Search space can be explored using various search algorithms, such as backtracking, forward checking, and constraint propagation
- Search algorithms can be guided by heuristics, such as minimum remaining values (MRV) and degree heuristic, to improve efficiency
- Example: Sudoku, where the goal is to fill a 9x9 grid with digits while satisfying certain constraints, such as no digit can appear twice in the same row, column, or 3x3 sub-grid

Cryptarithmetic Example



Example: Cryptarithmetic

$$\begin{array}{rcccc} & & T & W & O \\ + & & T & W & O \\ \hline & F & O & U & R \end{array}$$

- Variables: *F T U W R O* $X_1 X_2 X_3$
- Domains: {*0,1,2,3,4,5,6,7,8,9*} {0,1}
- Constraints: *Alldiff (F,T,U,W,R,O)*
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$

https://aimacode.github.io/aima-exercises/csp-exercises/

Map coloring

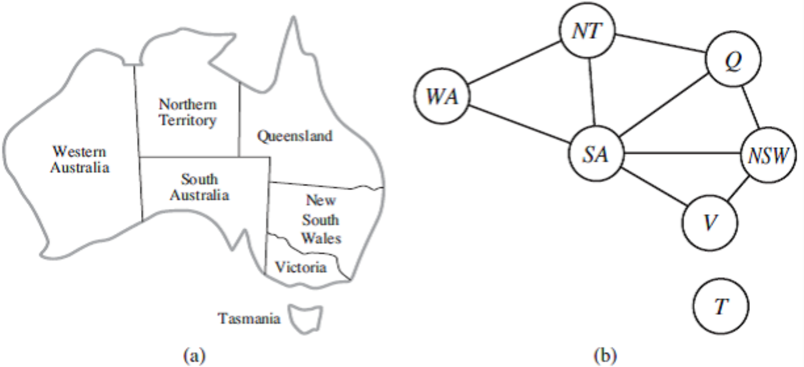
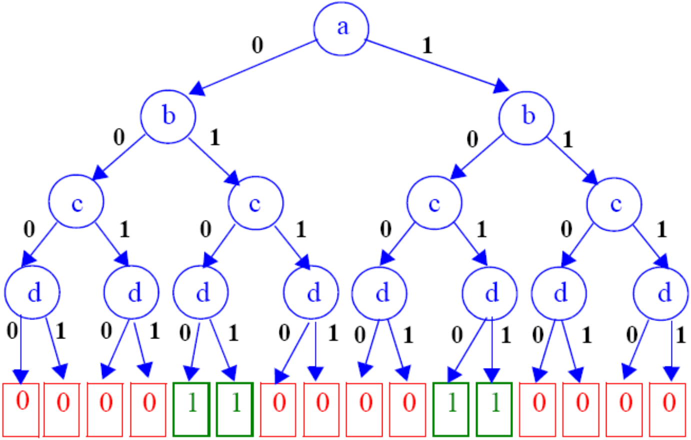


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Satisfiable SAT Example

$$f(a,b,c,d) = \begin{array}{l} (a \vee b \vee c) \cdot (a \vee b \vee \bar{c}) \cdot (\bar{a} \vee c \vee d) \\ (\bar{a} \vee c \vee \bar{d}) \cdot (\bar{b} \vee \bar{c} \vee d) \cdot (\bar{b} \vee \bar{c} \vee d) \end{array}$$



https://www.researchgate.net/publication/45267434$_S$ymmetry$_i$n$_B$oolean$_S$atisfiability

Applications of CSP in Real Life

- CSPs are used in many domains of real life, including:
    - Scheduling: Scheduling of courses, meetings, appointments, etc.
    - Resource Allocation: Allocation of resources such as staff, equipment, and materials to various tasks or projects
    - Circuit Design: Design of circuits that meet certain performance and reliability constraints
    - Robotics: Planning and control of robot motions and actions in dynamic environments
    - Bioinformatics: Analysis of biological data, such as DNA sequencing and protein folding

Formal Definition of CSP

Constraint Satisfaction Problem (CSP)

- A triple $< X, D, C >$ where
  - $X = \{X_1, X_2, ..., X_n\}$ is a set of variables
  - $D = \{D_1, D_2, ..., D_n\}$ is a set of domains, where each D_i is the domain of variable X_i
  - $C = \{C_1, C_2, ..., C_m\}$ is a set of constraints, where each C_j is a constraint over some subset of variables in X
- The goal is to find an assignment of values to variables in X that satisfies all constraints in C
- A solution to a CSP is a complete assignment of values to variables that satisfies all constraints

Basic Search Algorithms for CSPs

- **Backtracking Search:** This is the most basic form of search for solving CSPs. It involves choosing values for variables one at a time and backtracking when a variable has no legal values left to assign.
- **Forward Checking:** This enhancement to backtracking immediately checks ahead as each variable is assigned to eliminate values from the domains of unassigned variables that are inconsistent with the current partial assignment.
- **Constraint Propagation:** While AC-3 is a form of constraint propagation focused on enforcing arc consistency, there are other forms, such as path consistency and k-consistency methods, which ensure that constraints are satisfied across larger sets of variables.
- **...:** ...

CSP Algorithm

Algorithm CSP Algorithm

```
Result: $S$
Input  : $P =< X, D, C >$
Output: A solution $S$ to the CSP problem
$S \leftarrow \{\}$,$stack \leftarrow []$, $variables \leftarrow$ a list of all variables in $P$ , $domains \leftarrow$ a dictionary of domains for each variable in $P$ ,
  $constraints \leftarrow$ a list of constraints in $P$
  for $X_i$ in variables do
      initialize $X_i$ with $D_i$, push $(X_i, D_i)$ onto stack
end
while stack is not empty do
    $(X_i, D_i) \leftarrow$ pop from stack
      if $X_i$ not in $S$ then
          $S[X_i] \leftarrow D_i$ , push $(X_i, D_i)$ onto stack
    end
    if all variables have a value then
        return $S$
    end
    $X_i \leftarrow$ select a variable from variables , $D_i \leftarrow$ select a value from $domains[X_i]$
      while $D_i$ does not satisfy all constraints do
        if stack is empty then
            return Failure
        end
        $(X_j, D_j) \leftarrow$ pop from stack , $domains[X_j] \leftarrow domains[X_j] \cup \{D_j\}$ , $domains \leftarrow$ update domains using constraints
          if $X_j \neq X_i$ then
              push $(X_j, D_j)$ onto stack
        end
    end
    $S[X_i] \leftarrow D_i$ , push $(X_i, D_i)$ onto stack , $domains \leftarrow$ update domains using constraints
end
return Failure
```

Constraint Propagation and Local Search in CSP

Constraint Propagation:

- Reduces search space by enforcing constraints between variables
- Propagates effects of variable assignments to reduce domain of neighboring variables
- Helps to prune the search space and leads to faster solution times

Local Search:

- Finds solutions that satisfy a set of constraints
- Improves an initial solution iteratively by making small modifications
- Modifications guided by heuristic function that evaluates solution quality

In CSPs, combining local search with constraint propagation can further reduce the search space and improve search efficiency. One approach is to use a local search algorithm to find an initial solution, and then use constraint propagation to further prune the search space and improve solution quality.

Types of Constraint Propagation Techniques in CSP

- **Arc Consistency (AC)**: Enforces binary constraints between variables by removing inconsistent values from their domains. Repeated until no more values can be removed.
- **Path Consistency (PC)**: Extends AC to enforce n-ary constraints by maintaining consistency along paths of length two or more. Requires a stronger propagation mechanism than AC.
- **Generalized Arc Consistency (GAC)**: Enforces n-ary constraints by considering all combinations of values that satisfy the constraint. Removes inconsistent values from variable domains and can lead to further pruning of the search space.
- **Forward Checking (FC)**: Keeps track of remaining legal values for each unassigned variable. When a value is assigned to a variable, it removes incompatible values from its neighbors' domains. Stops when a variable's domain is empty.
- **Maintaining Arc Consistency (MAC)**: Combines AC and FC to enforce constraints during the search process. AC is used to prune the domains before search, and FC is used during search to ensure consistency is kept.

Forward Checking (FC) Algorithm for CSP

Algorithm Forward Checking Algorithm (initialization)

```
Input: $P = (X, D, C)$
Output: Solution to CSP problem or Failure
$S \leftarrow \{\}$
$stack \leftarrow []$
$variables \leftarrow$ a list of all variables in $P$
$domains \leftarrow$ a dictionary of domains for each variable in $P$
$constraints \leftarrow$ a list of constraints in $P$
for $X_i$ in variables do
    $D_i \leftarrow$ select a value from $domains[X_i]$
    if $D_i$ is None then
        return Failure
    end
    $S[X_i] \leftarrow D_i$
    for $X_k$ in variables do
        if $X_k \neq X_i$ then
            if $(X_i, X_k) \in constraints$ then
                $domains[X_k] \leftarrow domains[X_k] \setminus \{D_i\}$
                if $domains[X_k] = \emptyset$ then
                    return Failure
                end
            end
        end
    end
    push $(X_i, D_i)$ onto stack
end
```

Forward Checking (FC) Algorithm for CSP (cont.)

Algorithm Forward Checking Algorithm for CSP (main body)

```
while stack is not empty do
    ($X_i$, $D_i$) $\leftarrow$ pop from stack
    if $X_i$ not in $S$ then
        $S[X_i] \leftarrow D_i$
        if all variables have a value then
            return $S$
        end
        $X_j \leftarrow$ select an unassigned variable from variables
        $D_j \leftarrow$ select a value from $domains[X_j]$
        if $D_j$ is None then

        end
        $flag \leftarrow$ True
        for $X_k$ in variables do
            if $X_k \neq X_i$ then
                if $(X_j, X_k) \in constraints$ then
                    $domains[X_k] \leftarrow domains[X_k] \setminus \{D_j\}$
                    if $domains[X_k] = \emptyset$ then
                        $flag \leftarrow$ False
                    end
                end
            end
        end
        if $flag$ then
            push ($X_j$, $D_j$) onto stack
        end
    end
end
return Failure
```

Arc Consistency (AC-3) Algorithm for CSP

```
Algorithm AC-3 Algorithm
Result: Arc Consistency in CSP
Input  : A CSP problem $P = (X, D, C)$, where $X$ are variables, $D$ are domains,
         and $C$ are constraints
Output: Domains reduced to enforce arc consistency, or failure if inconsistency is
         found
Initialize a queue $Q$ with all arcs in $C$
 while $Q$ is not empty do
     $(X_i, X_j) \leftarrow$ dequeue from $Q$
      if Revise($P, X_i, X_j$) then
          if domain of $X_i$ is empty then
              return Failure
          end
          foreach neighbor $X_k$ of $X_i$ except $X_j$ do
              enqueue $(X_k, X_i)$ onto $Q$
          end
     end
end
return The reduced domains that satisfy arc consistency
```

Solving CSPs with SAT Solvers

- **SAT (Boolean satisfiability) solvers** are powerful tools for solving Boolean satisfiability problems, which can be used to encode CSPs.
- **Encoding CSPs** as Boolean formulas involves creating a set of Boolean variables and constraints that represent the CSP.
- **Translation** of a CSP to a Boolean formula can be done in various ways, such as using one variable per domain value, or using binary encoding for each variable.
- **Solving** the encoded formula with a SAT solver can lead to finding a satisfying assignment of variables that satisfies the original CSP constraints.
- **Examples** of existing SAT tools include MiniSat, Z3, Glucose, and Lingeling, among others.
- **Benefits** of using SAT solvers for CSPs include their ability to handle large, complex problems, their support for parallel processing, and their ability to handle constraints that are not easily represented using traditional CSP techniques.

Summary - Adversarial Search and Constraint Satisfaction Search

- Adversarial Search
  - Minimax Algorithm
  - Alpha-Beta Pruning
  - Other Variants (e.g. Monte Carlo Tree Search)
- Constraint Satisfaction Search
  - CSPs as a Search Problem
  - Constraint Propagation and Local Search
  - Backtracking Search and Improvements
  - Solving CSPs with SAT Solvers
  - Hybrid Approaches
- Applications of Adversarial Search and CSPs in AI

References

1. S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach”, Financial Times Prentice Hall, 2019.
2. M. Flasiński, Introduction to Artificial Intelligence”, Springer Verlang, 2016
3. M. Muraszkiewicz, R. Nowak (ed.), Sztuczna Inteligencja dla inżynierów”, Oficyna Wydawnicza PW, 2022
4. J. Prateek , Artificial Intelligence with Python”, Packt 2017