Games

Tushar B. Kute,
http://tusharkute.com

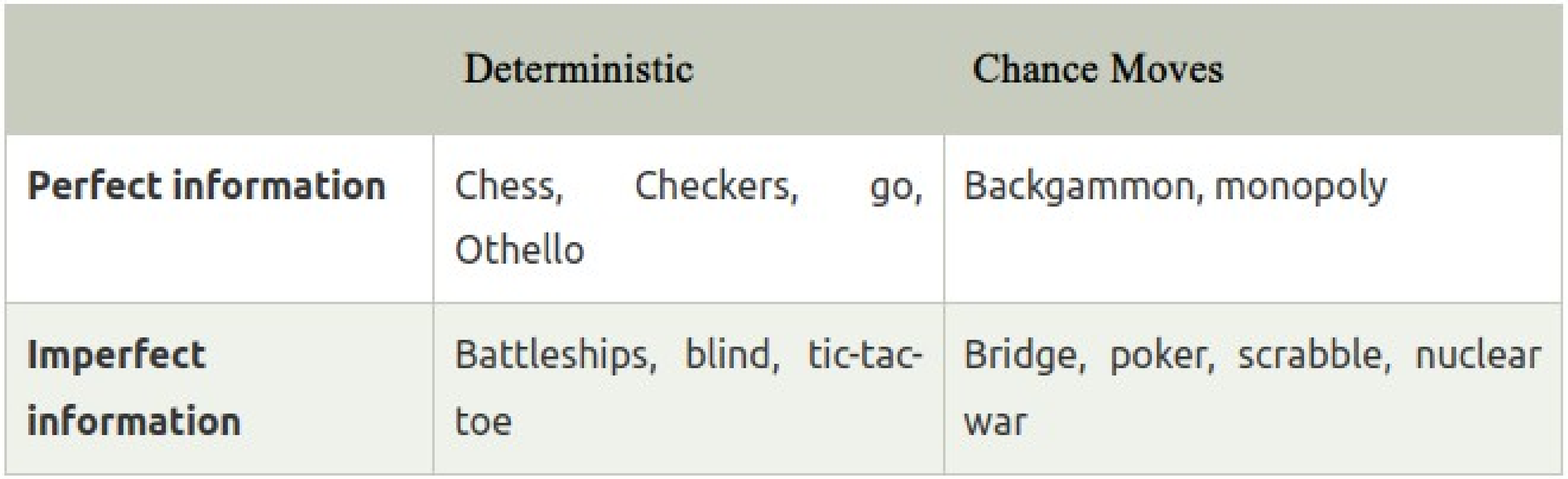
Adversarial Search

- Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.
  - In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
  - But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.

Adversarial Search

- The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI



	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Types of Games in AI

- Perfect information: A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- Imperfect information: If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.

Types of Games in AI

- Deterministic games: Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- Non-deterministic games: Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games.
- Example: Backgammon, Monopoly, Poker, etc.

Zero-sum game: Embedded thinking

- The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:
  - What to do.
  - How to decide the move
  - Needs to think about his opponent as well
  - The opponent also thinks what to do
- Each of the players is trying to find out the response of his opponent to their actions.
- This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem

- A game can be defined as a type of search in AI which can be formalized of the following elements:
  - Initial state: It specifies how the game is set up at the start.
  - Player(s): It specifies which player has moved in the state space.
  - Action(s): It returns the set of legal moves in state space.
  - Result(s, a): It is the transition model, which specifies the result of moves in the state space.

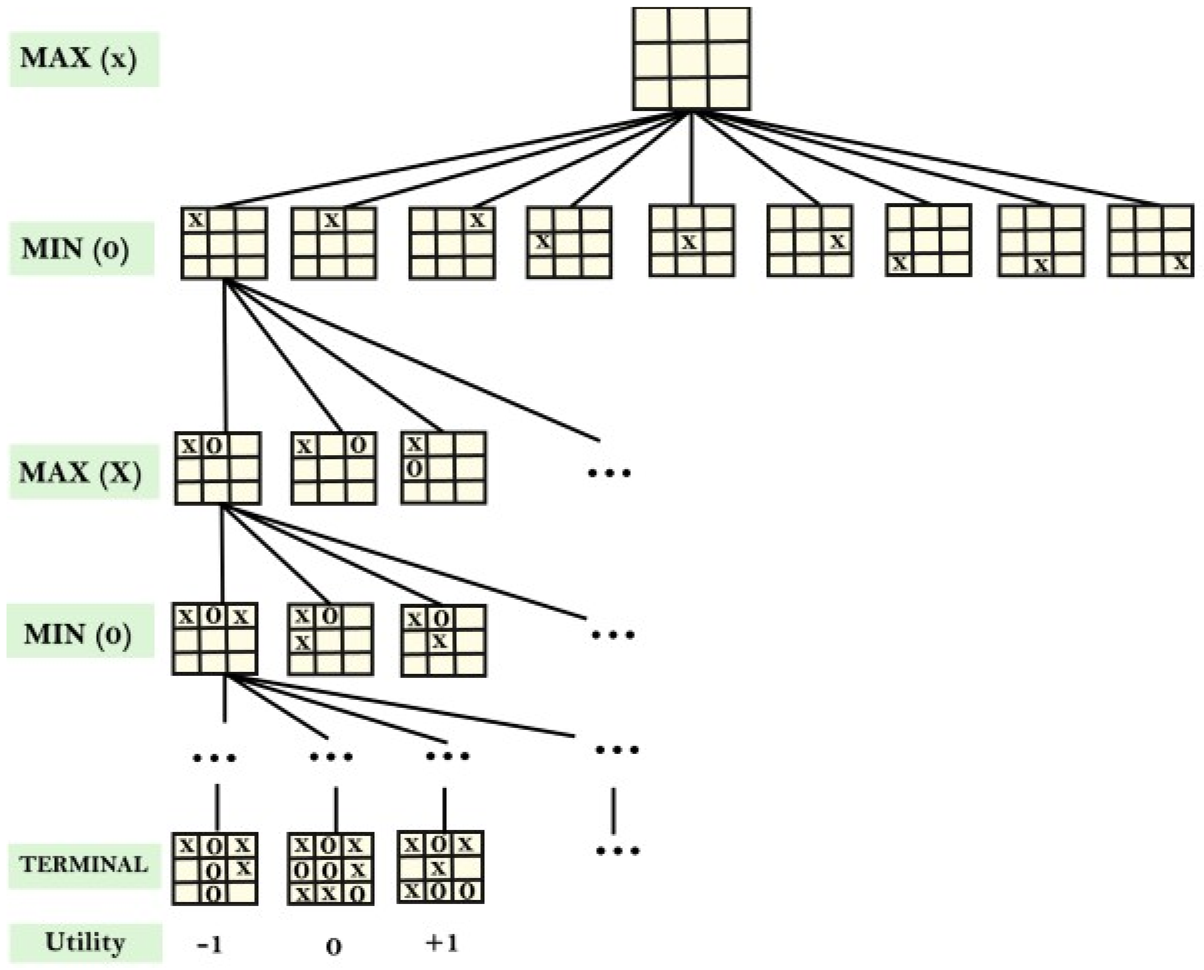
Formalization of the problem

- Terminal-Test(s): Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- Utility(s, p): A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function.
  - For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

Game tree

- A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.
- Example: Tic-Tac-Toe game tree:
- The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:
  - There are two players MAX and MIN.
  - Players have an alternate turn and start with MAX.
  - MAX maximizes the result of the game tree
  - MIN minimizes the result.

Game tree



Game tree

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.

Game tree

- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Game tree

- Hence adversarial Search for the minimax procedure works as follows:
  - It aims to find the optimal strategy for MAX to win the game.
  - It follows the approach of Depth-first search.
  - In the game tree, optimal leaf node could appear at any depth of the tree.
  - Propagate the minimax values up to the tree until the terminal node discovered.

Game tree

- In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY(s)} & \text{If TERMINAL-TEST(s)} \\ \max_{a \in \text{Actions(s)}} \text{MINIMAX(RESULT(s, a))} & \text{If PLAYER(s) = MAX} \\ \min_{a \in \text{Actions(s)}} \text{MINIMAX(RESULT(s, a))} & \text{If PLAYER(s) = MIN.} \end{cases}$$

Mini-Max Algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.

Mini-Max Algorithm

- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Pseudo-code for MinMax Algorithm

```
function minimax(node, depth, maximizingPlayer) is
    if depth ==0 or node is a terminal node then
        return static evaluation of node
    if MaximizingPlayer then      // for Maximizer Player
        maxEva= -infinity
        for each child of node do
            eva= minimax(child, depth-1, false)
            maxEva= max(maxEva,eva)       //gives Maximum of the values
        return maxEva
    else                          // for Minimizer player
        minEva= +infinity
        for each child of node do
            eva= minimax(child, depth-1, true)
            minEva= min(minEva, eva)       //gives minimum of the values
        return minEva
```

Initial Call

- Minimax(node, 3, true)
- Working of Min-Max Algorithm:
  - The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
  - In this example, there are two players one is called Maximizer and other is called Minimizer.
  - Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.

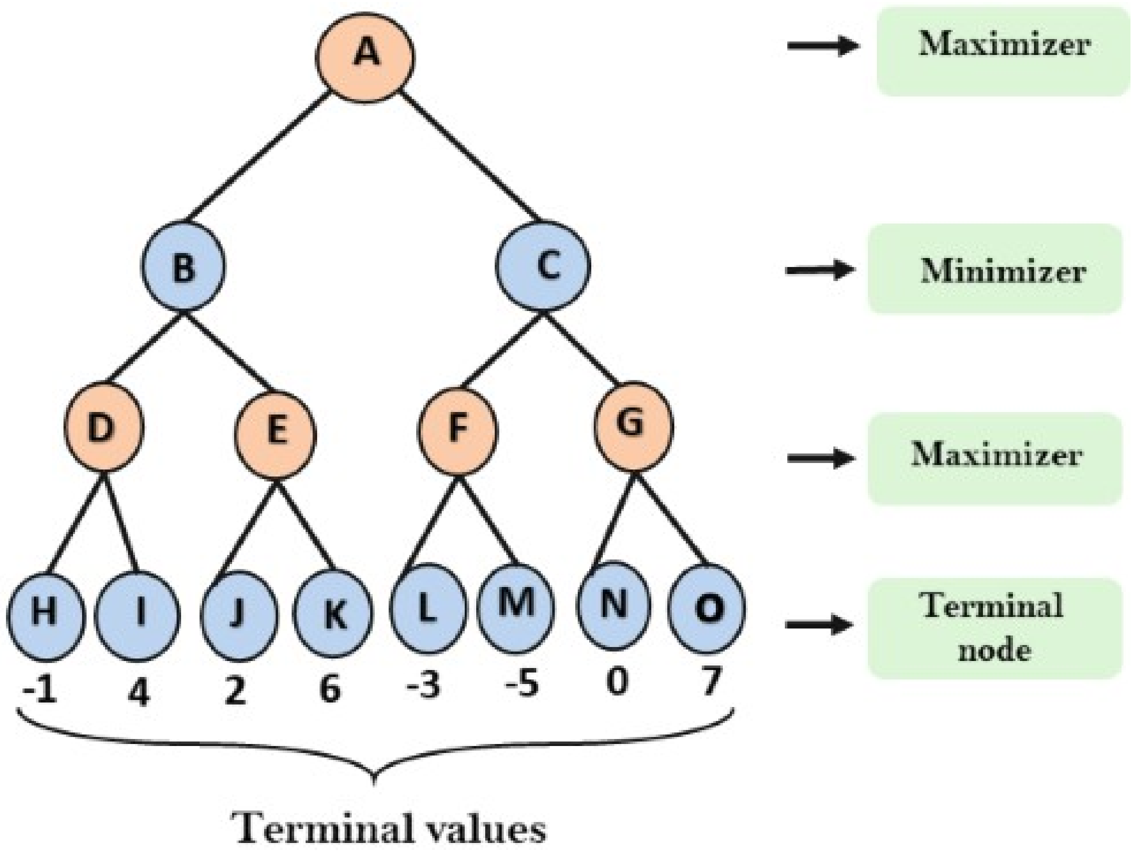
Initial Call

- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs.

Step-I

- In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.

Step-I

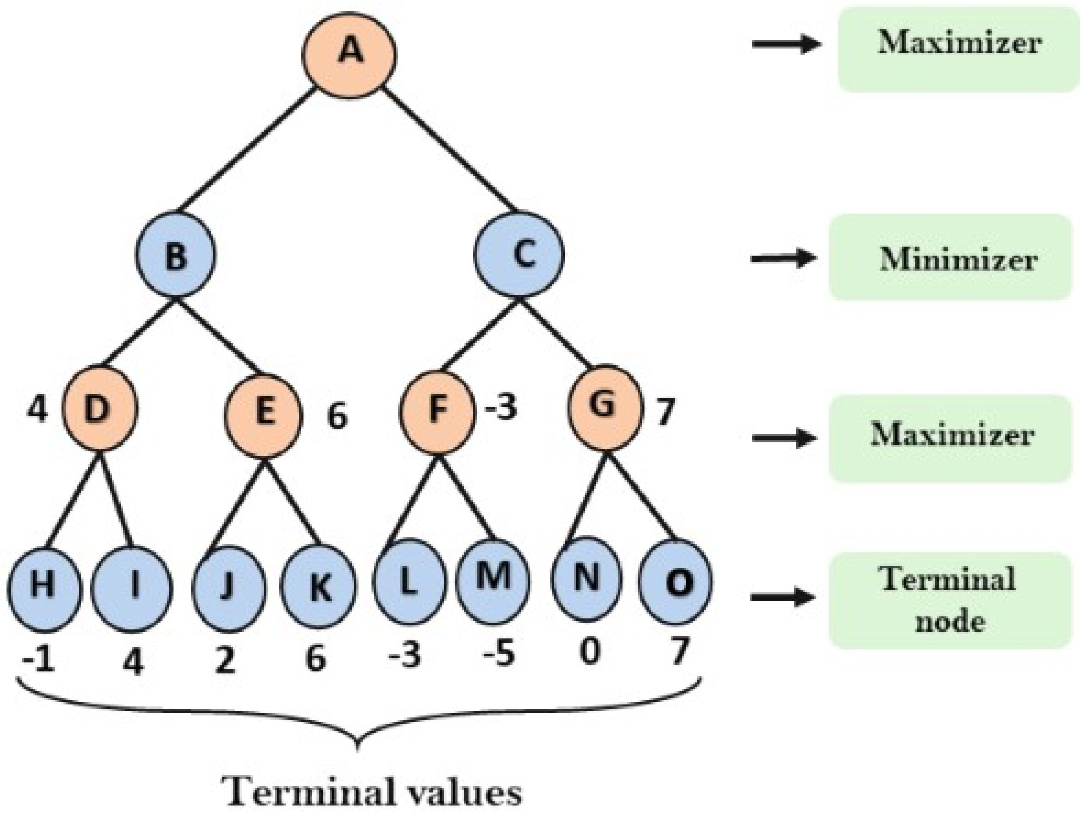

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
-1
4
2
6
-3
-5
0
7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node

Step-II

- Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

For node D	max(-1,- -∞) => max(-1,4)= 4
For Node E	max(2, -∞) => max(2, 6)= 6
For Node F	max(-3, -∞) => max(-3,-5) = -3
For node G	max(0, -∞) = max(0, 7) = 7

Step-II


A
B
C
4 D
E 6
F -3
G 7
H
I
J
K
L
M
N
O
-1
4
2
6
-3
-5
0
7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node

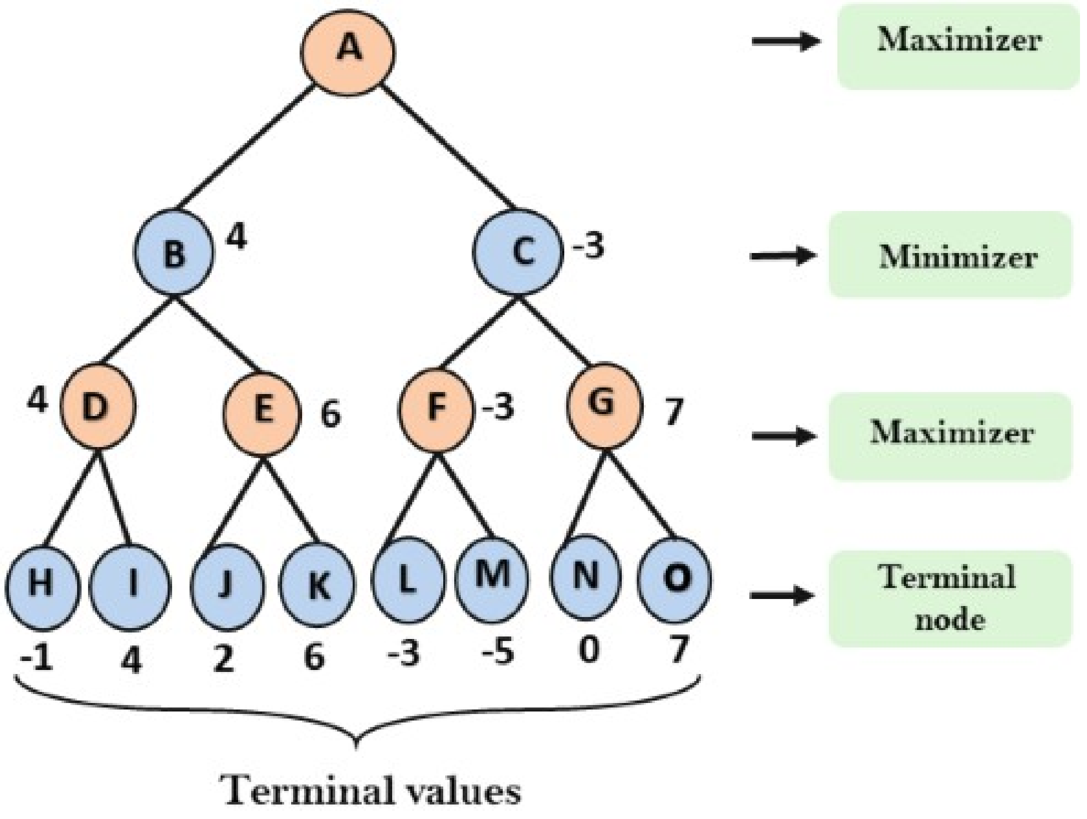
Step-III

- In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

For node B= min(4,6) = 4

For node C= min (-3, 7) = -3

Step-III



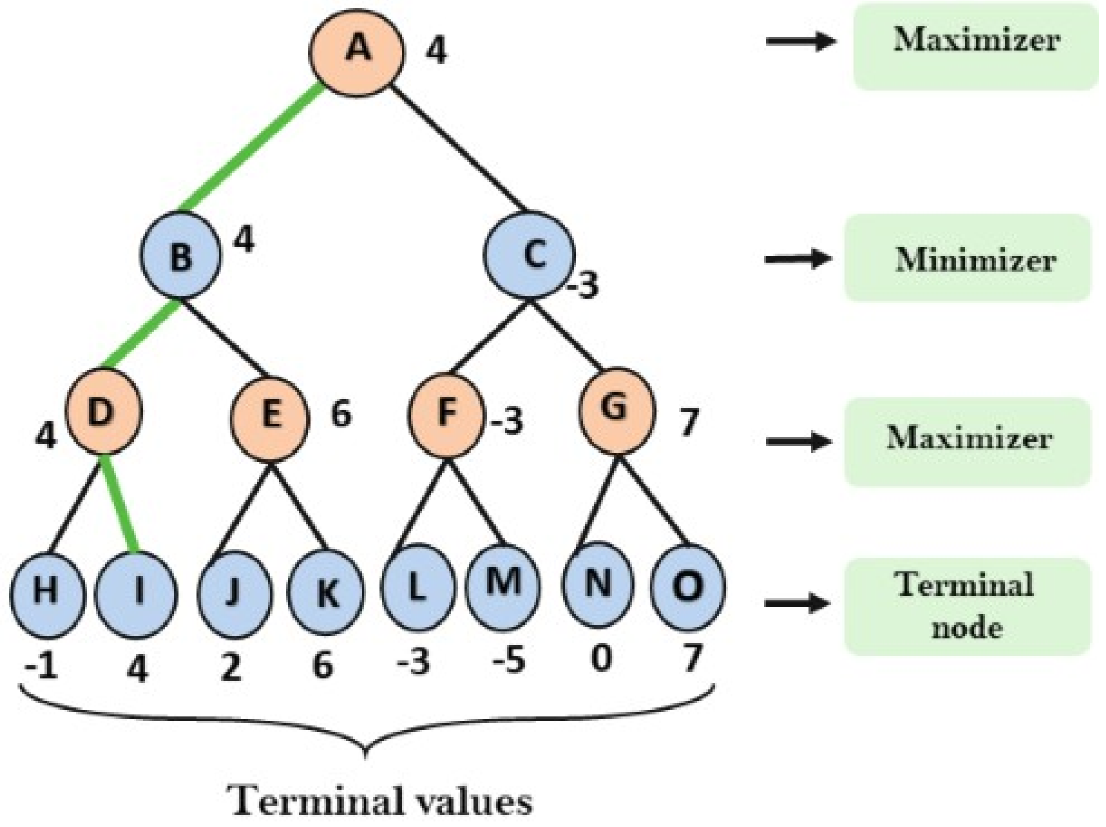
A
B
4
C
-3
D
4
E
6
F
-3
G
7
H
I
J
K
L
M
N
O
-1
4
2
6
-3
-5
0
7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node

Step-IV

- Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

For node A max(4, -3)= 4

Step-IV



That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm

- Complete- Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- Optimal- Min-Max algorithm is optimal if both opponents are playing optimally.
- Time complexity- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is O(bm), where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

Limitation of the minimax Algorithm

- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc.
- This type of games has a huge branching factor, and the player has lots of choices to decide.
- This limitation of the minimax algorithm can be improved from alpha-beta pruning

Inference in CSPs

- Path consistency
  - Path consistency: A two-variable set {Xi, Xj} is path-consistent with respect to a third variable Xm if, for every assignment {Xi = a, Xj = b} consistent with the constraint on {Xi, Xj}, there is an assignment to Xm that satisfies the constraints on {Xi, Xm} and {Xm, Xj}.
  - Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

Inference in CSPs

- K-consistency
  - K-consistency: A CSP is k-consistent if, for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.
  - 1-consistency = node consistency; 2-consisency = arc consistency; 3-consistensy = path consistency.
  - A CSP is strongly k-consistent if it is k-consistent and is also (k - 1)-consistent, (k – 2)-consistent, … all the way down to 1-consistent.
  - A CSP with n nodes and make it strongly n-consistent, we are guaranteed to find a solution in time O(n2d). But algorithm for establishing n-consitentcy must take time exponential in n in the worse case, also requires space that is exponential in n.

Inference in CSPs

- Global constraints
  - A global constraint is one involving an arbitrary number of variables (but not necessarily all variables).
  - Global constraints can be handled by special-purpose algorithms that are more efficient than general-purpose methods.

Global constraints

- 1) inconsistency detection for Alldiff constraints
  - A simple algorithm: First remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more vairables than domain values left, then an inconsistency has been detected.
  - A simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constrains.

Global constraints

- 2) inconsistency detection for resource constraint (the atmost constraint)
- We can detect an inconsistency simply by checking the sum of the minimum of the current domains;
- e.g. Atmost(10, P1, P2, P3, P4): no more than 10 personnel are assigned in total.
- If each variable has the domain {3, 4, 5, 6}, the Atmost constraint cannot be satisfied.
- We can enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains.
- e.g. If each variable in the example has the domain {2, 3, 4, 5, 6}, the values 5 and 6 can be deleted from each domain.

Global constraints

- 3) inconsistency detection for bounds consistent
- For large resource-limited problems with integer values, domains are represented by upper and lower bounds and are managed by bounds propagation.
- e.g. suppose there are two flights F1 and F2 in an airline-scheduling problem, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are
- D1 = [0, 165] and D2 = [0, 385].

Global constraints

- Now suppose we have the additional constraint that the two flight together must carry 420 people: F1 + F2 = 420. Propagating bounds constraints, we reduce the domains to

  D1 = [35, 165] and D2 = [255, 385].
- A CSP is bounds consistent if for every variable X, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y.

Sudoku

- A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row.
- The empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the pre-filled squares have a domain consisting of a single value.

Sudoku

- There are 27 different Alldiff constraints: one for each row, column, and box of 9 squares:
-
- Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)
-
- Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)
-
- …
-
- Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)
-
- Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)
-
- …
-
- Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)
-
- Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)
-
- …

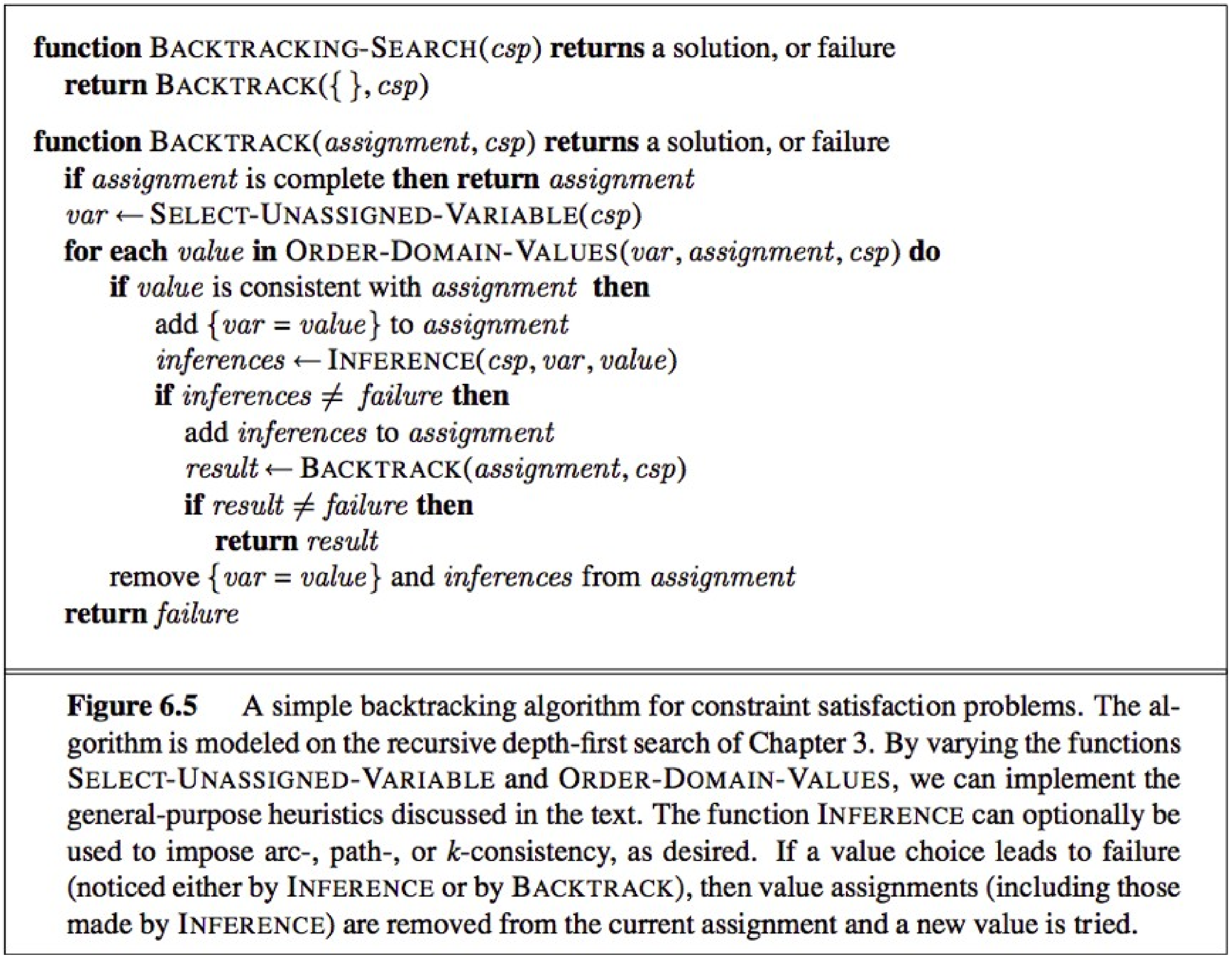
Backtracking search for CSPs

- Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome.
- Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

Backtracking search for CSPs

- Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.
- There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.
- BACKTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

Backtracking search for CSPs



```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
          add inferences to assignment
          result ← BACKTRACK(assignment, csp)
          if result ≠ failure then
            return result
     remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

Backtracking search for CSPs

- To solve CSPs efficiently without domain-specific knowledge, address following questions:
- 1)function SELECT-UNASSIGNED-VARIABLE: which variable should be assigned next?
- function ORDER-DOMAIN-VALUES: in what order should its values be tried?
- 2)function INFERENCE: what inferences should be performed at each step in the search?
- 3)When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

Intelligent backtracking

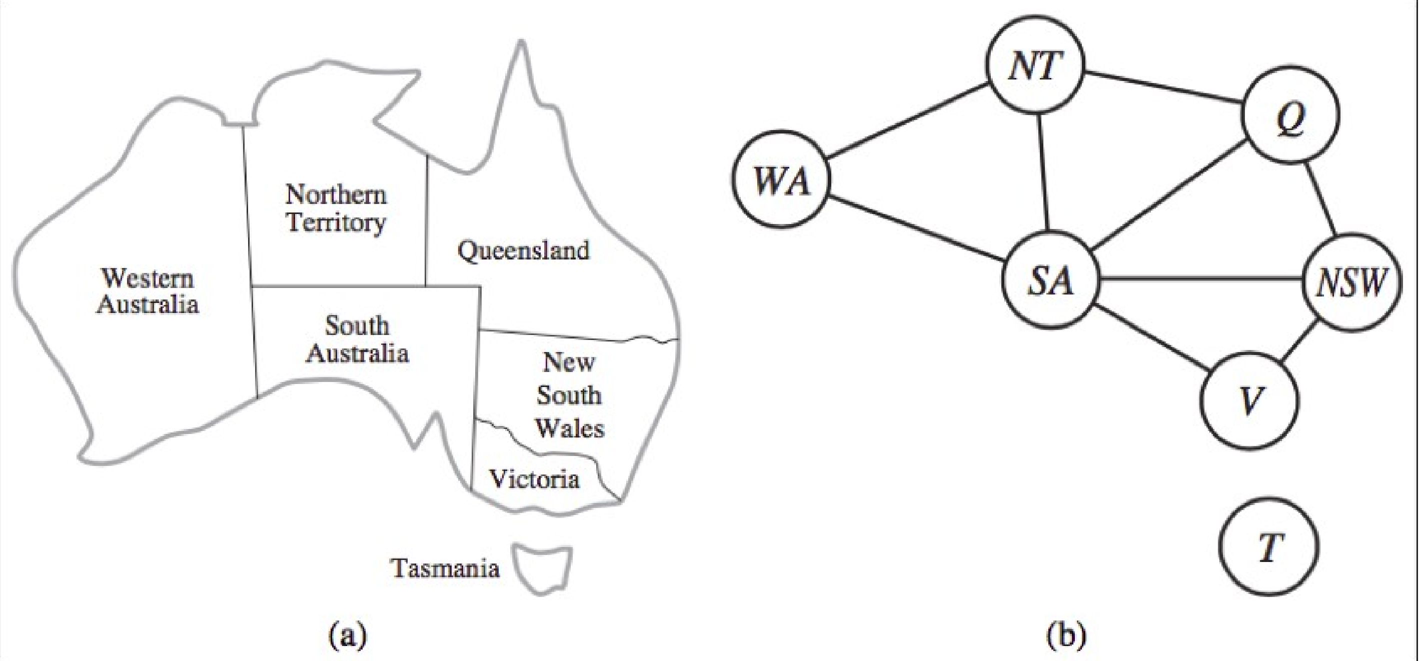
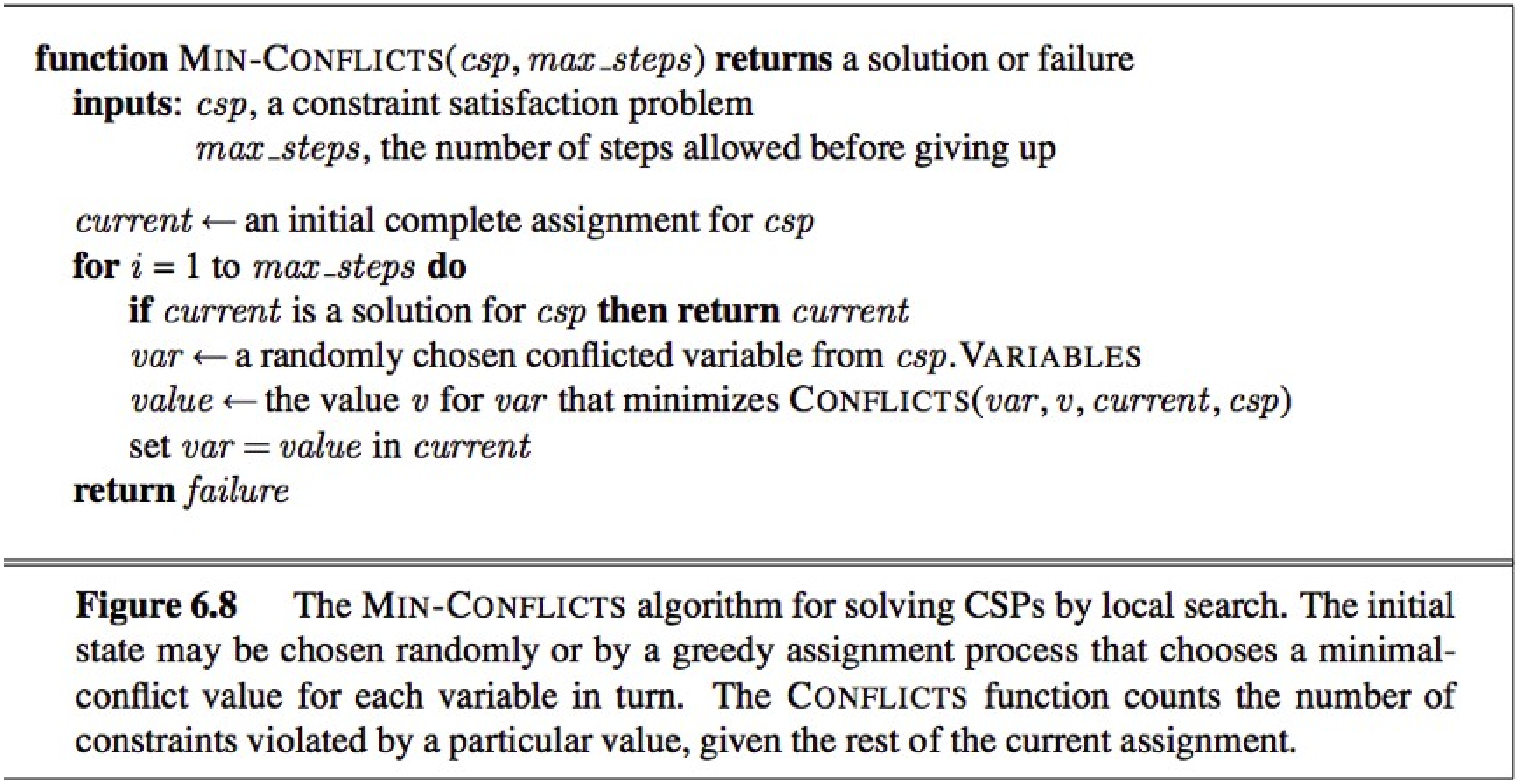


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Local search for CSPs

- Local search algorithms for CSPs use a complete-state formulation: the initial state assigns a value to every variable, and the search change the value of one variable at a time.
- The min-conflicts heuristic: In choosing a new value for a variable, select the value that results in the minimum number of conflicts with other variables.

Local search for CSPs



```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
     if current is a solution for csp then return current
     var ← a randomly chosen conflicted variable from csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
     set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Local search for CSPs

- The landscape of a CSP under the mini-conflicts heuristic usually has a series of plateau. Simulated annealing and Plateau search (i.e. allowing sideways moves to another state with the same score) can help local search find its way off the plateau.
- This wandering on the plateau can be directed with tabu search: keeping a small list of recently visited states and forbidding the algorithm to return to those tates.

Local search for CSPs

- Constraint weighting: a technique that can help concentrate the search on the important constraints.
- Each constraint is given a numeric weight Wi, initially all 1.
- At each step, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.
- The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.
- Local search can be used in an online setting when the problem changes, this is particularly important in scheduling problems.

Structure of Problem

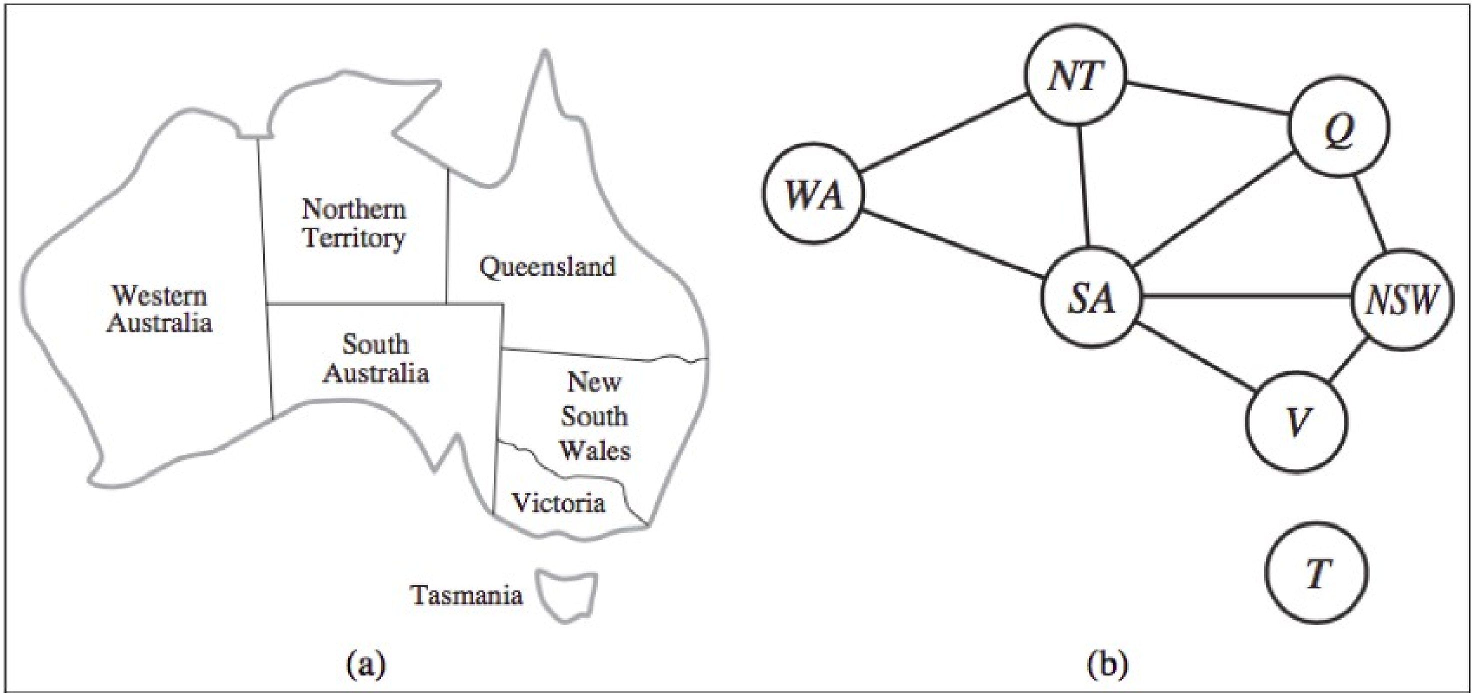


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

The structure of constraint graph

- The structure of the problem as represented by the constraint graph can be used to find solution quickly.
- e.g. The problem can be decomposed into 2 independent subproblems: Coloring T and coloring the mainland.
- Tree: A constraint graph is a tree when any two varyiable are connected by only one path.
- Directed arc consistency (DAC): A CSP is defined to be directed arc-consistent under an ordering of variables X1, X2, … , Xn if and only if every Xi is arc-consistent with each Xj for j>i.
- By using DAC, any tree-structured CSP can be solved in time linear in the number of variables.

The structure of constraint graph

- Pick any variable to be the root of the tree;
- Choose an ordering of the variable such that each variable appears after its parent in the tree. (topological sort)
- Any tree with n nodes has n-1 arcs, so we can make this graph directed arc-consistent in O(n) steps, each of which must compare up to d possible domain values for 2 variables, for a total time of O(nd2)
- Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value.
- Since each link from a parent to its child is arc consistent, we won't have to backtrack, and can move linearly through the variables.

The structure of constraint graph

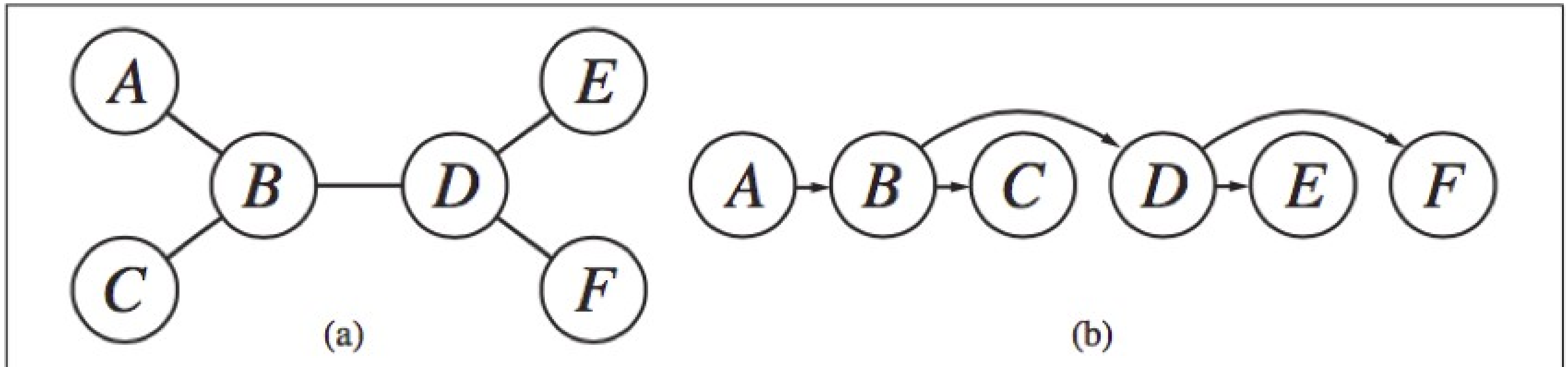
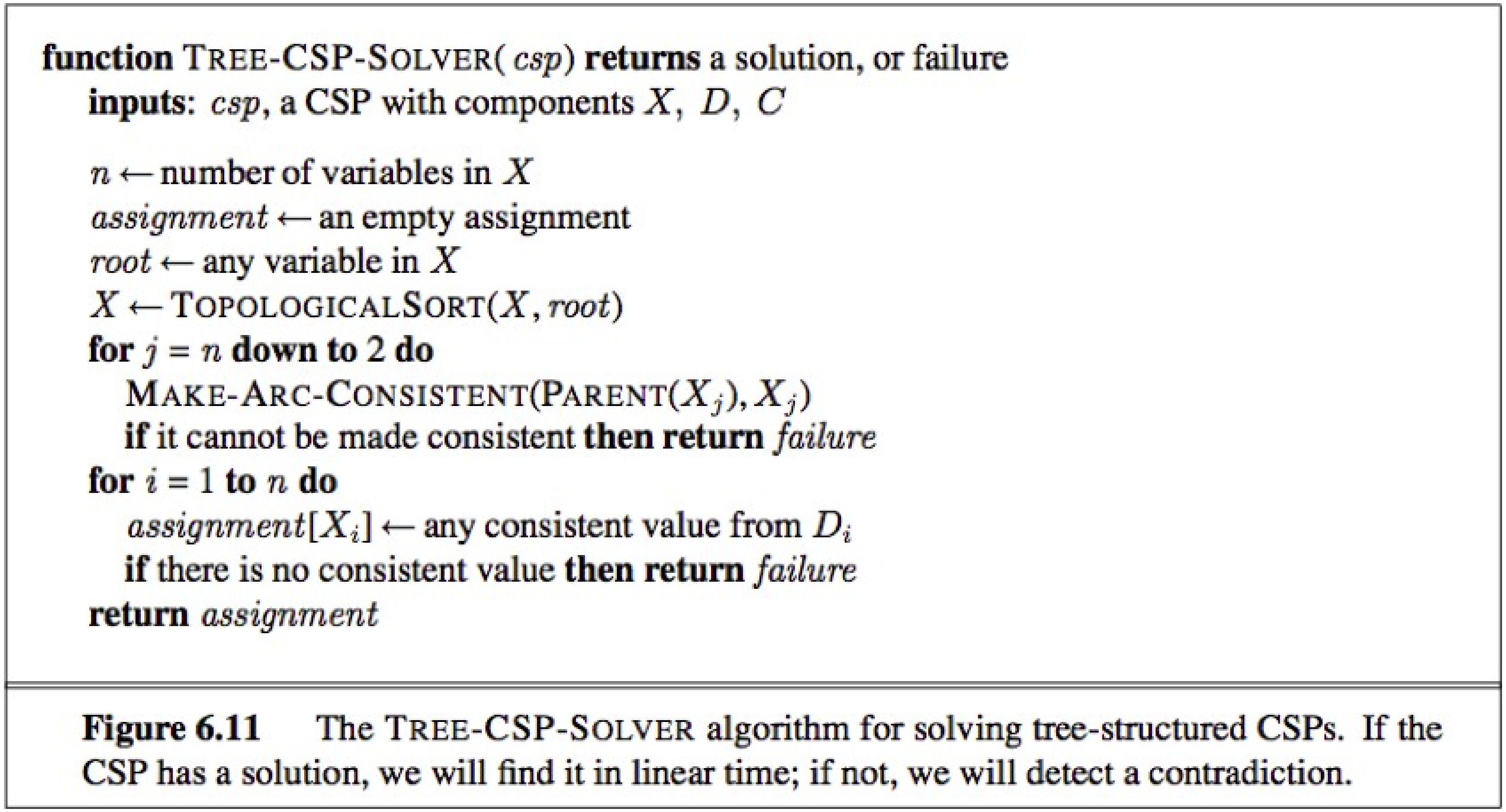


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

The structure of constraint graph



```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
    inputs: csp, a CSP with components $X$, $D$, $C$

    $n \leftarrow$ number of variables in $X$
    assignment $\leftarrow$ an empty assignment
    root $\leftarrow$ any variable in $X$
    $X \leftarrow$ TOPOLOGICALSORT($X$, root)
    for $j = n$ down to 2 do
        MAKE-ARC-CONSISTENT(PARENT($X_j$), $X_j$)
        if it cannot be made consistent then return failure
    for $i = 1$ to $n$ do
        assignment[$X_i$] $\leftarrow$ any consistent value from $D_i$
        if there is no consistent value then return failure
    return assignment
```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

Thank you

@mitu_skillologies

/mITuSkillologies

@mitu_group

/company/mitu-skillologies

MITUSkillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

contact@mitu.co.in

tushar@tusharkute.com