Summer School in Artificial Intelligence

A quick overview of good old-fashioned AI

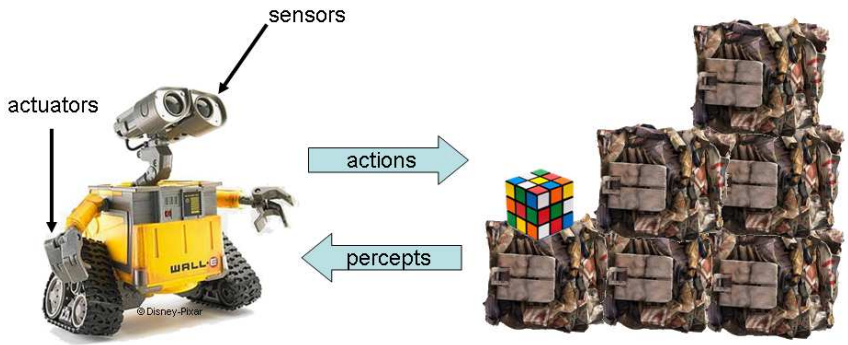
Jean-Philippe Poli

Teacher,
Centrale-Supélec

Researcher,
CEA LIST

Outline

Foreword : What is an intelligent agent ?



An intelligent agent is an entity that perceives its environment through sensors and acts on its environment through effectors.

Foreword : Rational agent

Artificial Intelligence = Computational Rationality

- Rational : maximally achieving pre-defined goals.
- Rationality only concerns what decisions are made (not the though process behing them).
- Goals : expressed in terms of utility of outcomes.
- Being rational means maximizing your expected utility

Foreword : Rational agent

Definition

- An entity that perceives and acts on its environment to accomplish its goals according to its abilities, or its beliefs (or knowledge).
- A rational agent selects actions that maximize its (expected) utility

Characteristics of the **perceits, environment** and **action space** dictate techniques for designing rational agents.

Foreword : Task Environment

PEAS

The first task to design a rational agent is to specify the task environment (PEAS) :

- Performance measure.
- Environment.
- Actuators.
- Sensors.

Foreword : Property of the environment

Observable :

- Do the sensors give at a time a complete view of the environment ?

Deterministic vs stochastic :

- The state of the environment depends on the current state and on the action of the agent ?

Episodic :

- Evaluation of the quality of an action into a cycle (perception-action) ? Next and future steps are needed ?

Static :

- Can the environment change during the agent deliberation ?

Discrete :

- Discrete states (e.g. :playing chess) or continous (e.g. : walking) ?

Foreword : Reflex agents

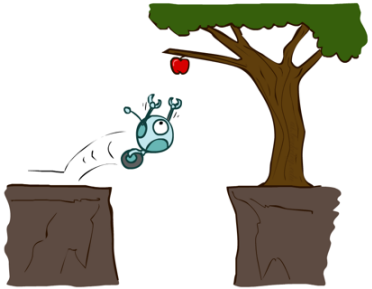


FIGURE – Source : CS188 Intro to AI - Berkeley

Foreword : Reflex agents

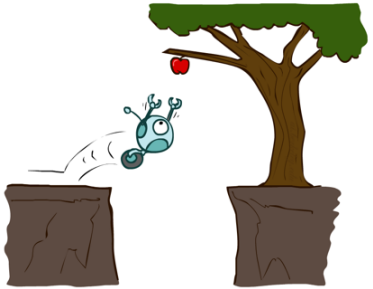


FIGURE – Source : CS188 Intro to AI - Berkeley

- Choose action (the decision) based on current percept (and maybe the memory)

Foreword : Reflex agents

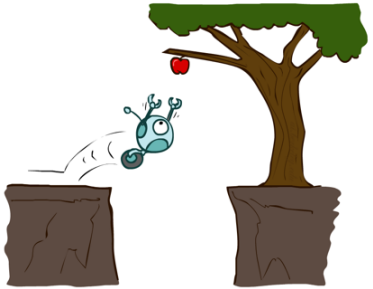


FIGURE – Source : CS188 Intro to AI - Berkeley

- Choose action (the decision) based on current percept (and maybe the memory)
- May have memory or a model of the world current state
- Consider how the world is : do not consider the future consequences of their actions.

Foreword : Reflex agents

Learning-based models

Foreword : Reflex agents

Learning-based models

- Starting point : a set of examples that partially describe the expected behavior of the system and a simple program whose parameters are not known.

Foreword : Reflex agents

Learning-based models

- Starting point : a set of examples that partially describe the expected behavior of the system and a simple program whose parameters are not known.
- The learning algorithm learns the program parameters from the examples in order to reproduce the system behavior as well as possible.

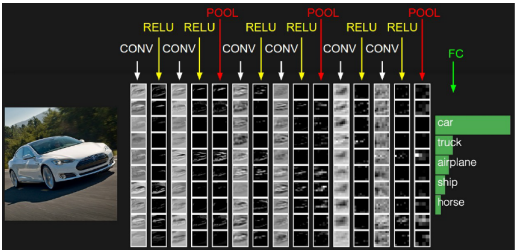
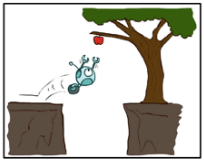


FIGURE – Source : cs221 course Stanford - Liang

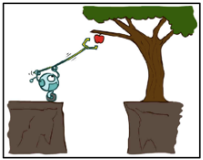
Foreword : Planning agents

Reflex agent



- Consider how the world IS

Planning agent



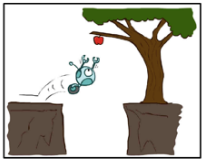
- Consider how the world WOULD BE

Planning agents

- Ask **what if** !

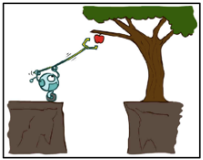
Foreword : Planning agents

Reflex agent



- Consider how the world IS

Planning agent



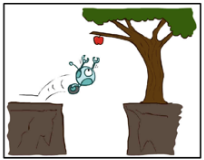
- Consider how the world WOULD BE

Planning agents

- Ask **what if** !
- Consider how the world would be.

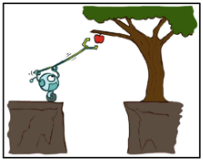
Foreword : Planning agents

Reflex agent



- Consider how the world IS

Planning agent



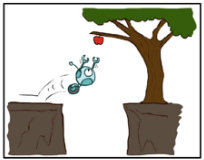
- Consider how the world WOULD BE

Planning agents

- Ask **what if** !
- Consider how the world would be.
- Decisions are based on (hypothesized) consequences of actions.

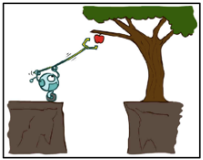
Foreword : Planning agents

Reflex agent



- Consider how the world IS

Planning agent



- Consider how the world WOULD BE

Planning agents

- Ask **what if** !
- Consider how the world would be.
- Decisions are based on (hypothesized) consequences of actions.
- Must have a model of how the world evolves in response to actions = state-based models.

Foreword : Problem-solver agents

Symbolic AI has often be used to solve different kind of problems.

Problem-solver agents

- Goal-based agents
- Decide what to do by finding sequences of actions that lead to a desire state
- In general, their environments are :
  - fully observable
  - deterministic
  - episodic
  - static
  - discrete

Outline

Problem solving agent

Main principle

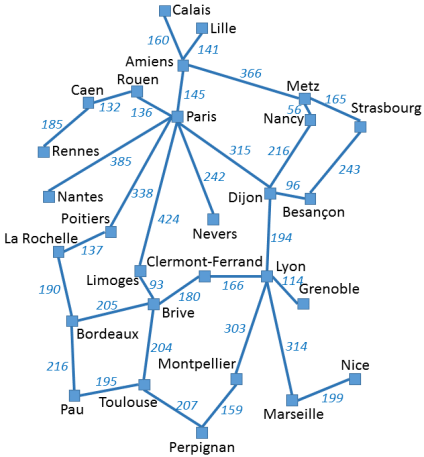
1. Problem specification :
   Set of all the states and all actions to be considered during solving.
2. Goals specification :
   Set of states to be reached.
3. Solutions searching :
   Browsing the different sequences of actions that lead to a final state and chose the best one.
4. Execution :
   Execute the chosen sequence of actions.

Problem solving agent : example

First example : go from Paris to Marseille

1. Goal : be at Marseille.
2. Problem :
   - States : cities
   - Actions : go from one city to another
3. Solutions : city sequences

Environment : static, observable, discrete and deterministic.



Problem formulation

A problem can be defined by 4 components :

- Initial state : the state x in which the agent is
  ex : $At(Paris)$
- Actions : descriptions of all the possible actions for the agent, with a successor function S which associates to a state x a set of pairs $(action, successor)$.

  ex : $S(At(Paris)) = \{(GoTo(Dijon), At(Dijon)), (GoTo(Poitiers), At(Poitiers)) \ldots\}$

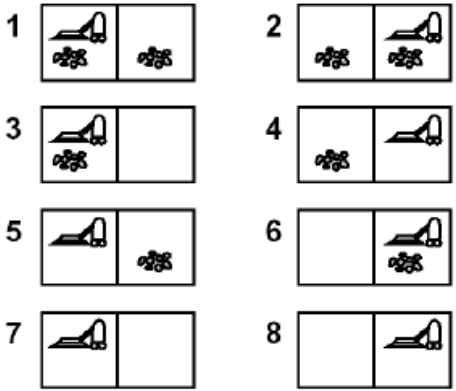
  - State space defined by the initial state and the application of the successor function. Can be represented by a directed graph whose nodes are states and edges are actions.
  - A path in this state space is a sequence of states linked by actions.

Problem formulation

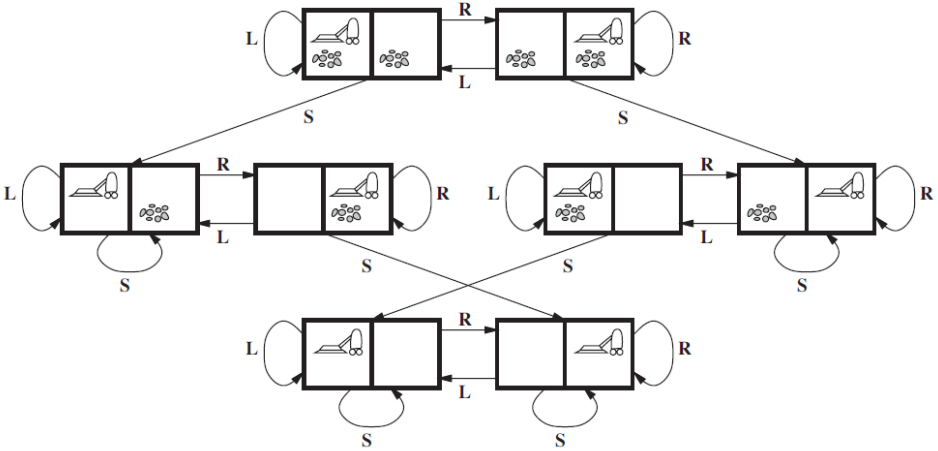
- Goal test : a test that determines whether a given state is a goal or not.
  ex : *Goal :* $\{At(Marseille)\}$
- Path cost : function that gives a cost to each path. It is a performance measure for the agent. Its values are positive.
  ex : *mileage*.

Example 1 : vacuum agent

- States : 2 positions + dirty or not : 8 possible states.
- Initial state : any of them
- Successor function : possible actions *Left*, *Right*, *Sucks*.
- Goal : all the dust is eliminated (states 7 or 8)
- Cost : each link costs 1 (i.e. number of steps)



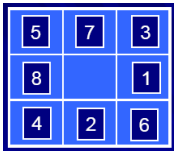
Example 1 : vacuum agent - state space



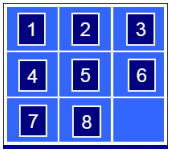
Example 2 : solving a 8-puzzle

- states : numbers of the positions of the blocks.
- successor function : actions *move blank square to the left, right, up or down*.
- goal test : current state == goal state.
- cost : each move cost 1 (the cost is thus the number of moves of the blank square).

NP-complete problem



Etat initial



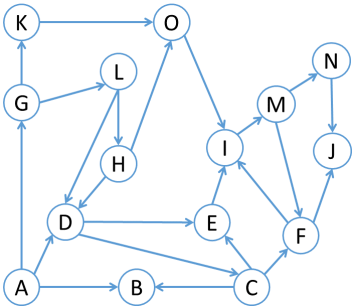
Etat but

Real world problem examples

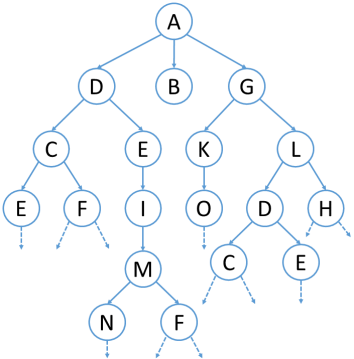
- Path finding : networks, planning of military operations, aircraft trajectory planning.
- Touristic tours or not : salesman problem.
- Robot navigation when map is known.
- Assembly of micro-electronic components

State-space graphs and search trees

General problem



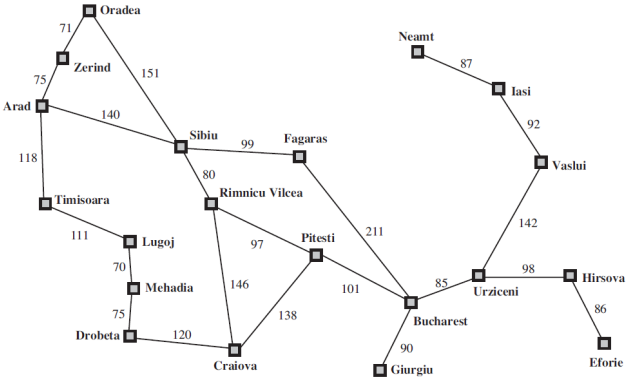
Tree representation

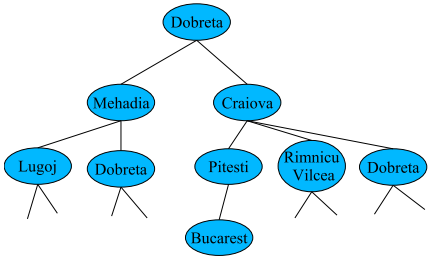


State-space graphs

- **State-space graph** : a mathematical representation of a search problem
  - Node are world configurations.
  - Arcs represent sucessors.
  - The goal test is a set of goal nodes.
- In a state graph, each state occurs only once !
- We can rarely build the full graph in memory.

Example : from Dobreta to Bucarest





Solution searching with a tree

Idea

Offline simulation of the state space by generating the successors of already explored states.

A node in such a tree is a data structure composed of :

- State : the state represented by the node.
- The root node : the start state.
- Parent node : the node in the search tree that generated the current node.
- Action : the action that has been applied on the parent node to generate the current node.
- Path cost : cost from initial state to the current node $g(n)$.
- Depth : number of actions on the path from initial state.

Nodes show states but correspond to **PLANS** that achieve those states.
For most problems, we can never actually build the whole tree

State Space graphs versus Search Trees

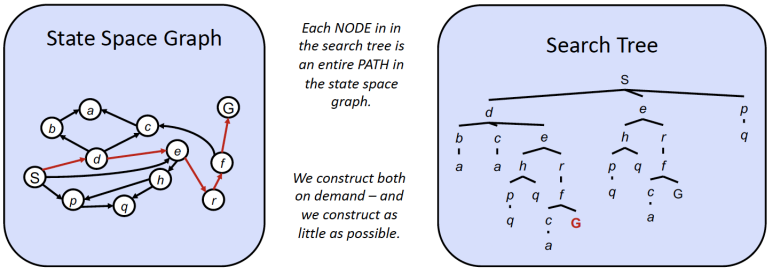


FIGURE – Source : Berkeley AI course

Search strategy

Main idea of searching in a tree

- express all the solutions
- find the best solution browsing the tree
- Which search strategy, i.e. at each step, which node to extend ?

Searching with a Search Tree

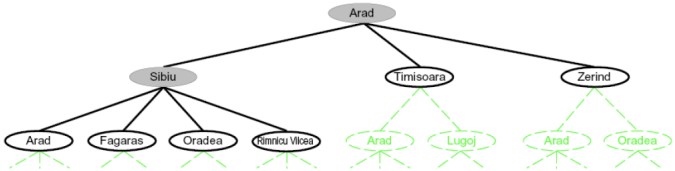


FIGURE – Source : Berkeley AI course

Search :

- Expand out potential plans (tree nodes)
- Maintain a fringe of partial plans under consideration
- Try to expand as few three nodes as possible.

General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

FIGURE – Source : Berkeley AI course

Main ideas

- Fringe
- Expansion
- Exploration strategy
- Main question : what fringe nodes to explore ?

Search performances

Criteria to evaluated the different search methods

- Completeness : does it guarantee to find a solution if it exists ?
- Optimality : does it find the best solution if several solutions exist ?
- Time complexity : time to find a solution (often regarding the number of nodes in the tree)
- Space complexity : needed memory (number of nodes in memory)
- Classicaly, the complexity is expressed regarding :
  - b : maximum branching factor (maximum number of successors for one node).
  - s : depth of the best solution.
  - m : maximal length of a path.

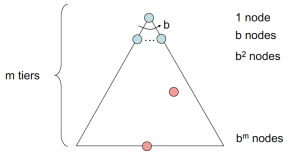


FIGURE – Source : Berkeley AI course

Different types of strategies

Uninformed solving strategies

Do not use any information of the nodes.

Informed search strategies

Use information to determine if a node is more promising than another.

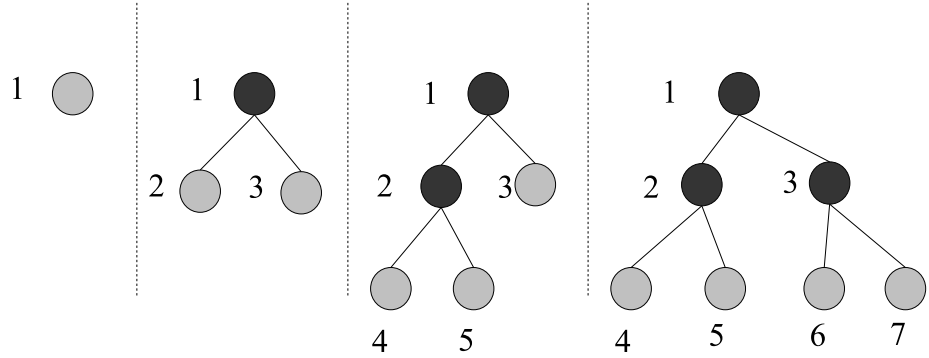
Uninformed search strategies

Examples of uninformed search strategies

- Breadth first
- Uniform cost
- Limited depth first
- Iterative depth first

Breadth-first strategy

- Strategy : expand a shallowest node first. One extends all the nodes at a level i before extending the nodes at level $i+1$.
- Implementation : Fringe as a FIFO queue



File attente = (1) File attente = (2,3) File attente = (3,4,5) File attente = (4,5, 6,7)

Breadth-first strategy

Properties

- Completeness
  Yes if b and s is finite
- Time complexity
  $1 + b + b^2 + b^3 + ... + b^s + (b^{s+1} - b)$
  i.e. $\sum_{i=1}^{s} b^i + (b^{s+1} - b) = o(b^{s+1})$
- Space complexity : same (all nodes are kept in memory)
- Optimal : no in general, yes if the cost is the same for all actions (cost are all 1)

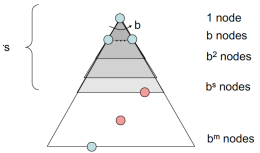
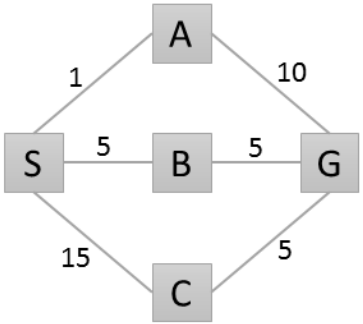


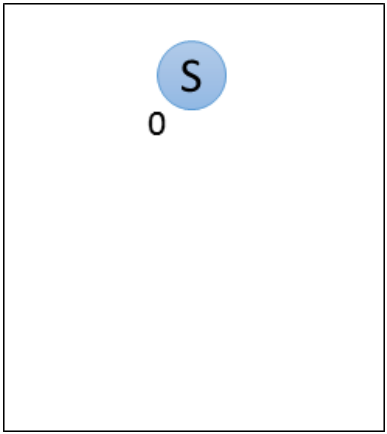
FIGURE – Source : Berkeley AI course

Uniform-cost strategy

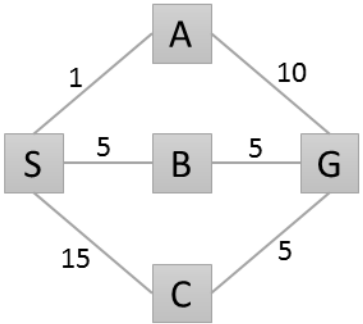
- Expand the node with the cheapest cost first.
- Implementation : Fringe as a Priority queue, the queue is sorted by cost ascending (cumulative cost).
- If cost is always the same, it's equivalent to breadth-first strategy.

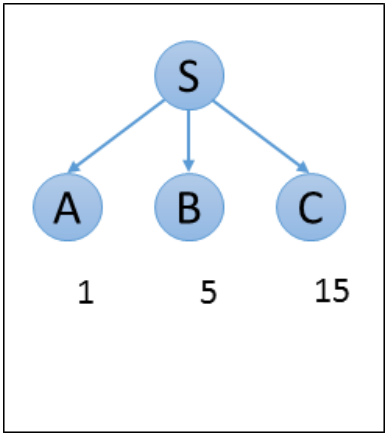
Uniform-cost strategy



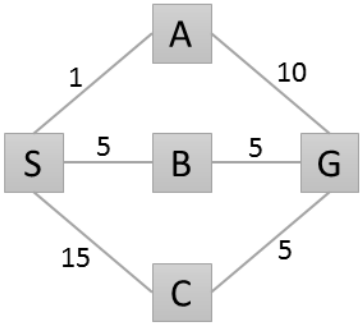


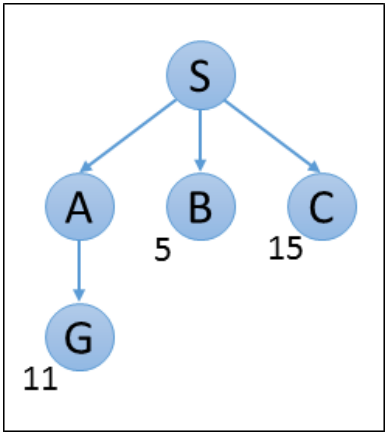
Uniform-cost strategy



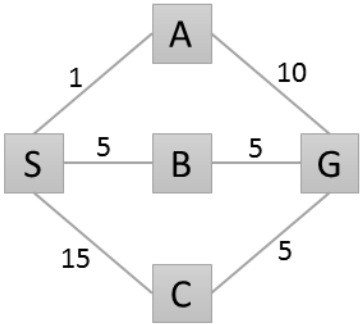


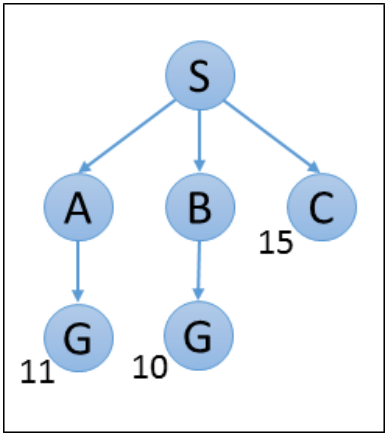
Uniform-cost strategy





Uniform-cost strategy





Uniform-cost strategy

- Completeness : yes (if cost > 0 and if the best solution has a finite cost).
- Time complexity : $o(b^s)$
- Space complexity : $o(b^s)$
- Optimal : yes

Here we can consider that the effective depth is $s = \frac{C^*}{\epsilon}$ with C^* the solution cost and that the arcs cost at least ϵ

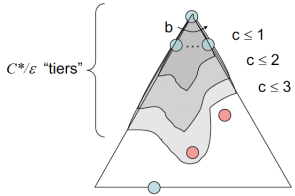


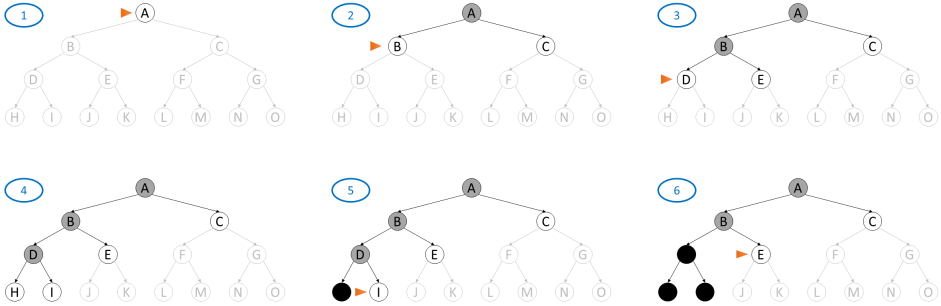
FIGURE – Source : Berkeley AI course

Depth-first strategy

- Extend the deepest node first.
- Implementation : Fringe with a LIFO stack.

Caution : loops ! State space have to be finite and acyclic.

Depth-first strategy



http ://aima.cs.berkeley.edu/

Depth-first strategy

Properties

- Completeness :
  - no, if depth is infinite, if cycles.
  - yes if state space is finite and acyclic
- Time complexity : $o(b^m)$ (costly if $m >> s$)
- Space complexity : $o(b * m)$ (linear) (a node can be deleted if we visited its descendants)
- Optimal : no (if tree is unbalanced, with a really deep left branch)
- Good for memory usage.

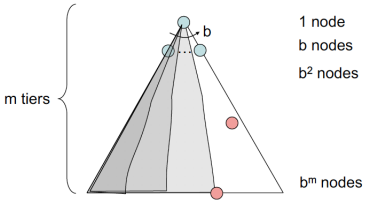


FIGURE – Source : Berkeley AI course

Limited depth-first strategy

- Depth-first strategy with a limited depth L.
- Implementation : nodes at depth L have no successors.

Limited depth-first strategy

Properties

- Completeness : Yes, if L $\geq$ s
- Time complexity : $o(b^L)$
- Space complexity : $o(b * L)$
- Optimal : no (e.g. L >s)
- Interesting strategy if the problem is well-known
- Good for memory usage.

Iterative deepening depth-first strategy

- The problem with limited depth first strategy is to find the right L.
- Iterative deepening : repeat for L=0,1,2,...
- Combines advantages of breadth-first and depth-first.
  - optimal and complete as breadth-first
  - memory saving as depth-first.

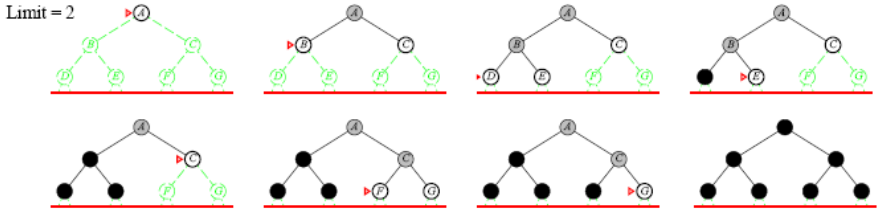
Iterative deepening depth-first strategy

Limit = 0

Iterative deepening depth-first strategy

http ://aima.cs.berkeley.edu/

Iterative deepening depth-first strategy



http ://aima.cs.berkeley.edu/

Iterative deepening depth-first strategy

Properties

- Completeness : yes
- Time complexity : $o(b^s)$
- Space complexity : $o(s)$
- Optimal : yes if cost is not decreasing

Blind search algorithm comparisons

- b = branching factor
- s = depth of solution
- m = maximum depth of the search tree
- l = depth limit

Criteria	Breadth	Cost	Depth	Depth L	Depth l
Completeness	yes	yes	yes	no if $L \geq s$	yes
Time	$o(b^{s+1})$	$o(b^s)$	$o(b^m)$	$o(b^L)$	$o(b^s)$
Space	$o(b^{s+1})$	$o(b^s)$	$o(bm)$	$o(bL)$	$o(b * s)$
Optimal	yes	yes	no	no	yes

Informed search

Principle

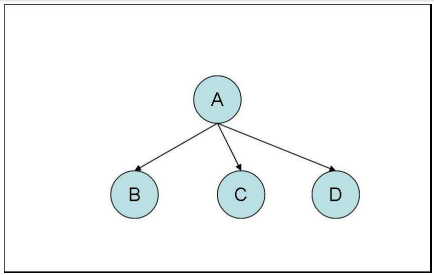
- Contrary to uninformed search strategies that do not take into account informations about the problem, inform search strategies use available information to improve the efficiency of the solving process
- A heuristic information is a rule or a method that improves always (or almost) the search.
- A heuristic is a function that *estimates* how close a state is to a goal.

Heuristic function

Definition

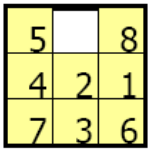
- A heuristic function $H : S \to \mathbb{R}$ associated to $x \in S$ (S state space) a number $h(x) \in \mathbb{R}$ which is an estimation of the cost to extend the current path passing by x.
- Constraint : h(solution)=0

- Node A has 3 successors with $h(B) = 0.8$, $h(C) = 2.0$ et $h(D) = 1.6$.
- The search continues with B which is the best regarding the heuristic.

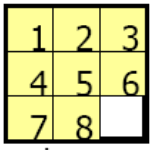


Examples of heuristic functions

$h(N)$=number of misplaced squares=6



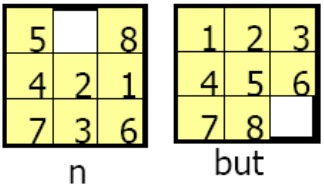
n



but

Examples of heuristic functions

$h(N)$=sum of the distances of each square to its final position.
$h(N) = 3+1+3+0+2+1+0+3 = 13$

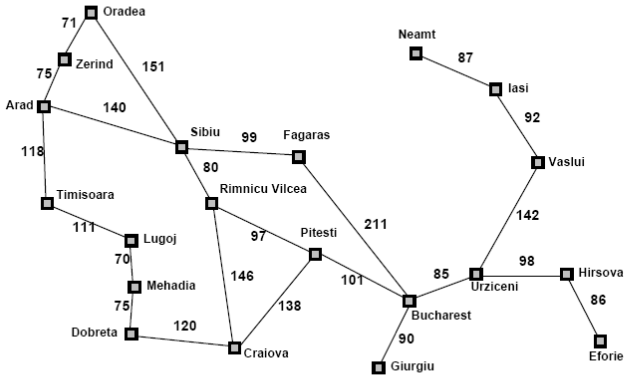


Greedy search

- Combination of breadth-first and depth-first strategies :
  - Depth : no need to compute all the nodes to find solutions.
  - Breadth : no deadend or loop.
- Principle : explores the nodes regarding the value of the heuristic function. The node with the best heuristic is expanded first (minimization of the estimated cost to get to the goal). We expand the node that you think is closest to a goal state.
- Example of heuristic function : flight distance.

Greedy search

Example : Romania travel. Heuristic function : flight distance.

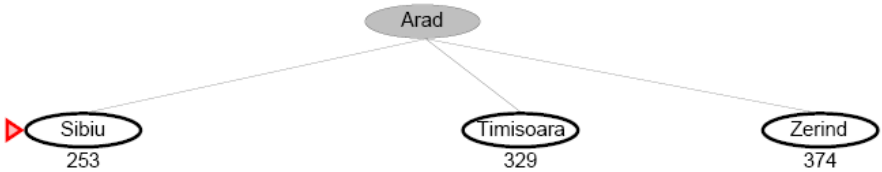


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

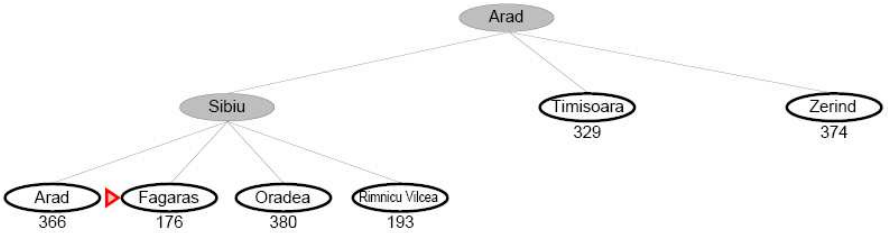
http ://aima.cs.berkeley.edu/

Greedy search



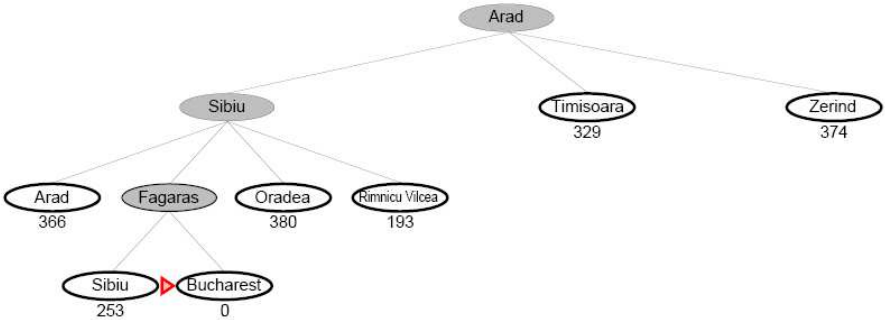
http ://aima.cs.berkeley.edu/

Greedy search



http ://aima.cs.berkeley.edu/

Greedy search



http ://aima.cs.berkeley.edu/

Greedy search

Properties

- Completeness : No, there can be a loop. Yes if state space is finite and acyclic.
- Time complexity : $o(b^m)$
- Time complexity : $o(b^m)$ (all nodes are kept in memory).
- Optimal : no

A^* strategy

Principle

Combines greedy search and uniform cost search.

- greedy search minimizes the estimated cost $h(n)$ from node n to the goal (goal proximity, forward cost) : non optimal, non complete.
- uniform cost search minimizes the cost $g(n)$ from initial state to node n (path cost, backward cost). optimal, complete, but not efficient.

Principle

Minimize the total cost $f(n)$ of the path passing by node n
$f(n) = h(n) + g(n)$

Admissible Heuristics

Definition

A heuristic h is admissible (optimistic) if :

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal.

A^* strategy

Theorem

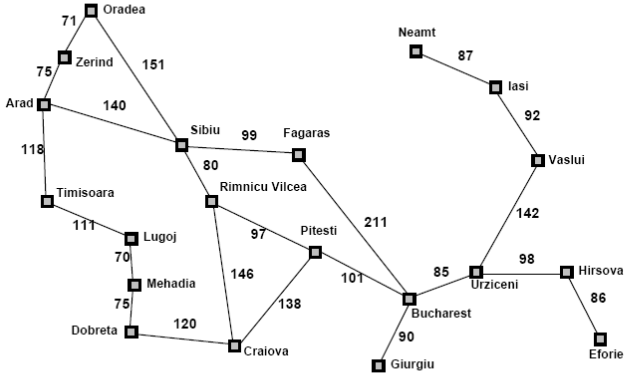
If A^* uses an admissible heuristic function, i.e. an heuristic function that never overestimates the real cost, then A^* is optimal

Creating Admissible Heursitics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics.
- Often, admissible heuristics are solutions to **relaxed problems**, where new actions are available.

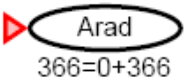
A^* search

Example : travel to Romania. Same heuristic function.



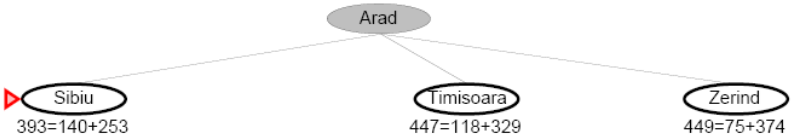
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A^* search



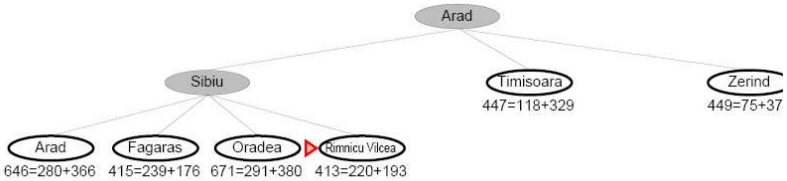
http ://aima.cs.berkeley.edu/

A^* search



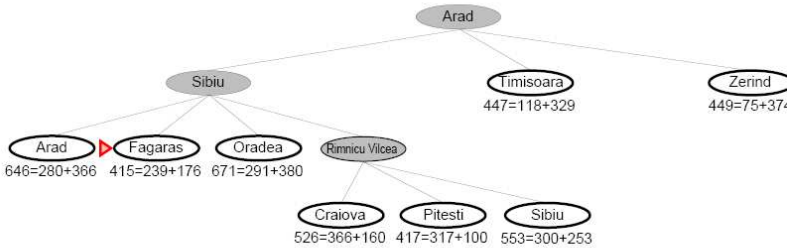
http ://aima.cs.berkeley.edu/

A^* search



http ://aima.cs.berkeley.edu/

A^* search



http ://aima.cs.berkeley.edu/

A^* search

Properties

- Complete : yes, if number of nodes is finite
- Time complexity : exponential
- Space complexity : keeps all the node in memory.
- Optimal : Yes

Search problem : Summary

A search problem consists of

- A state space
- A sucessor founction (with actions, costs)
- A start state and a goal test
- A solution which is a sequence of actions (a plan) which transforms the start state to the goal state.

Search problem : Summary

Problem solvers :

- use graph to represent a problem
- use heuristics to perform faster

Search problem : Summary

Problem solvers :

- use graph to represent a problem
- use heuristics to perform faster
- need a human intervention to get this graph
- use symbolic states (lack of concision)

Search problem : Summary

We are going to see two other approaches that are based on these principles, but extend them in two different ways :

- Considering the agent is not alone changing the environment
- Allowing the states not to be black-boxes anymore

Outline

Adversarial search

Definition

Adversarial search is a type search that arises when we try to plan actions to solve a problem (or at least to reach a goal state) but there exist other agents that are planning against us.

Adversarial search is used in many fields to find optimal solutions in a competitive environment :

- Legal system : lawyers that argue to represent their party's cases,
- Business environment : sourcing suppliers and buyers,
- Games

Why studying games ?

A proof of intelligence ?

- Almost every Human plays.
- Results are easy to popularize by researchers.

A showcase for AI

It is easy to measure the progress of an AI to a 2 players game.

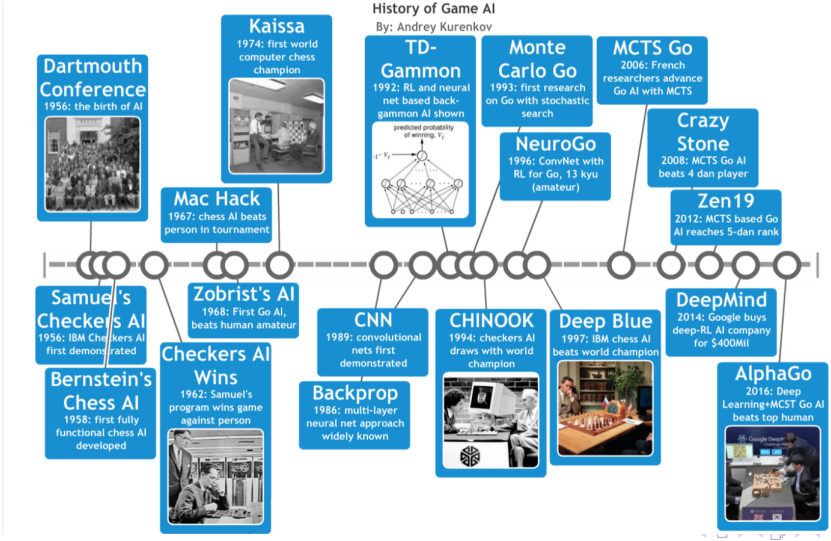
- Games are popular to demonstrate new ideas in AI ;
- But the techniques used must be applied to other domains of AI.

Games : the drosophilia of Artificial Intelligence

Some dates

- 1945 : Turing presented chess as a game a machine can play.
- 1946 : Turing talked about intelligence of machine, illustrating with chess.
- 1950 : Turing wrote the first software to play games.
- 1950 : Shannon wrote the first article about games.
- 1957 : Simon predicted : in 10 years, the champion to chess will be a computer.
- 1963 : Samuel wrote an AI to play checkers that won the world champion.
- 1989 : Creation of the World Man Machine Championship.

History of games in AI



Type of games

Classification of games

	Deterministic	**Random**
Complete information	chess, checkers, othello, go	monopoly, backgammon
Incomplete information	Battleship	poker, bridge

Games we are studying

- Turn-by-turn games
- Players have all the information about the game : the environment is deterministic and totally observable
- Zero sum : what a player wins, the other loses

Games as a search problem

- **States** : position of each item on the board + identity of the player to play
- **Initial state** : initial position for each item and identity of the first player
- **Successor function** : for a given state, return all the possible moves
- **Terminal states** : one of the players wins, or draw game
- **Utility function** : returns the value of a terminal state : +1 (wins), -1 (loses) et 0 (draw)

Definition

Game tree

Tree in which nodes represent the moves for the players. Alternates from a player to the other.

Ply

A given depth in the tree is called a ply.

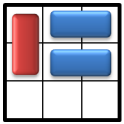
Terminal nodes

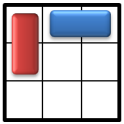
Indicate the end of the game, either a victory, a defeat or a draw.

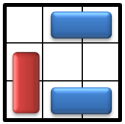
Colored dominoes

Principle

On a $n \times n$ board, you have to be the last player to lay a 2×1 domino. Player 1 uses horizontal positions and player 2 vertical positions.

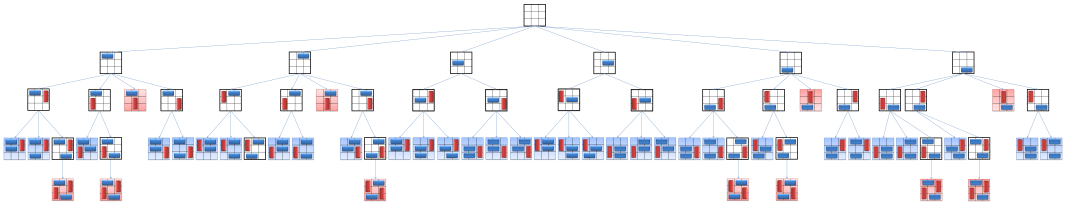




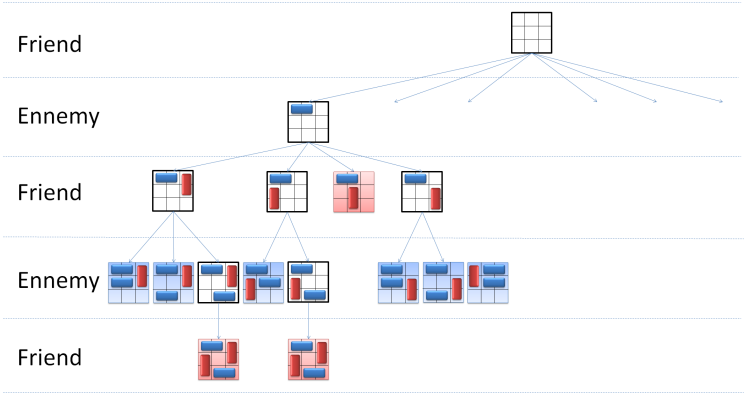


Tree game browsing

- The simulation of all the possible moves from the initial state allows to build the game tree.



Game tree browsing

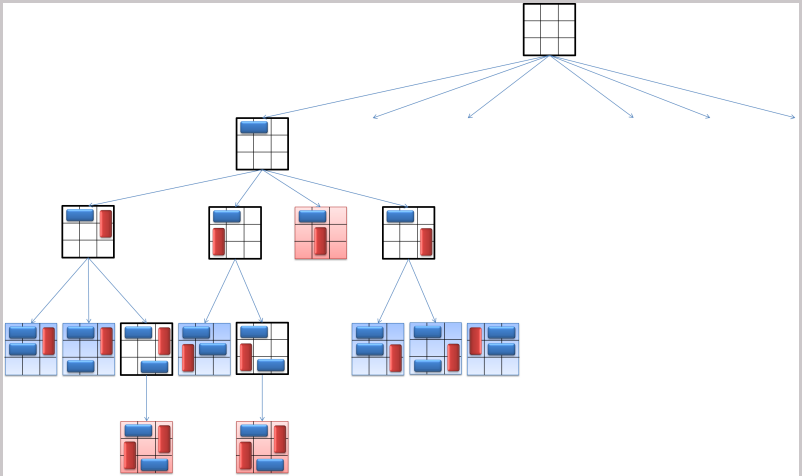


Game tree browsing

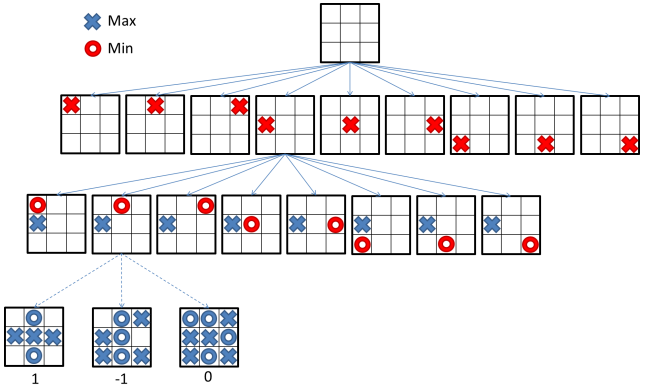
Depth first strategy.

Game tree browsing

Complete browsing of the tree game



Example : Tic Tac Toe



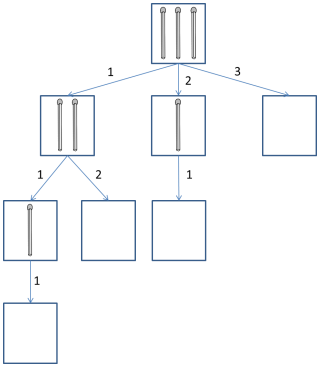
- Expand the game tree.
- Compute the values of each terminal node
- Back propagate the value to the root

Nim game

Principle

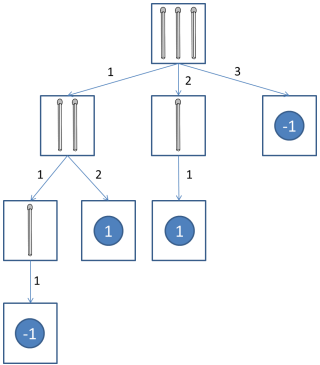
At each turn, a player can take 1, 2 or 3 matches. The player who gets the last match has lost.

We first expand the game tree.



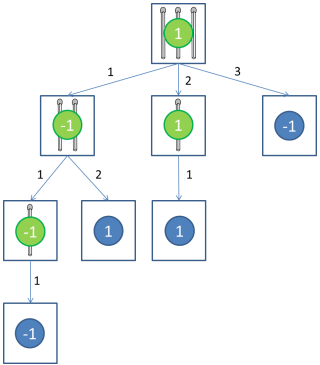
Nim game

We label the leaves regarding we won or lost.



Nim game

We back propagate the values of the leaves to the root.



Adversarial search algorithms

Two problems

1. How to backpropagate the right value ?
   This is achieved by the minimax algorithm.
2. How to prune the branches ?
   This is achieved by the alpha beta pruning

MiniMax algorithm

Principle

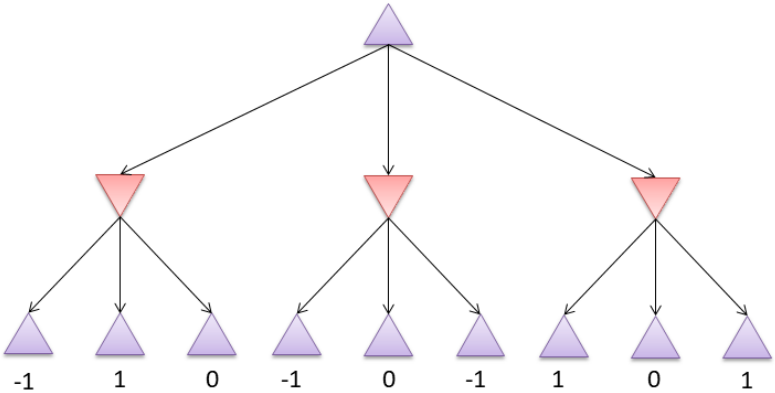
- We focus on one of the player called Max. All the heuristic values are evaluated from the point of view of Max.
- The goal is to make Max win : finding the sequence of actions to reach the terminal state with the greatest value ?
- **NOT so simple** : Max must take into account the actions of Min.
- The principle of the algorithm is to expand the tree, to assess each leaf and to back propagate those values under the hypothesis that each player chooses the best move for him :
  - MIN selects the minimum values
  - MAX selects the maximum values

MiniMax algorithm

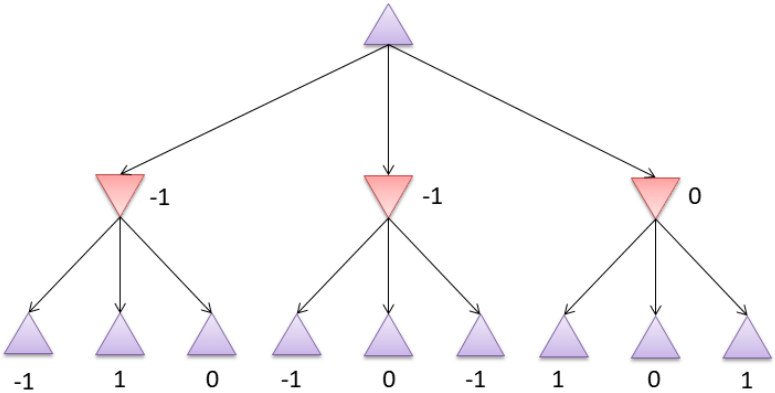
```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
  return v
```

- MiniMax-Decision returns the action that Max must perform
- Max-Value (MaxMin) returns the minimax of a state when Max has to play
- Min-Value (MinMax) returns the minimax of a state when Min has to play

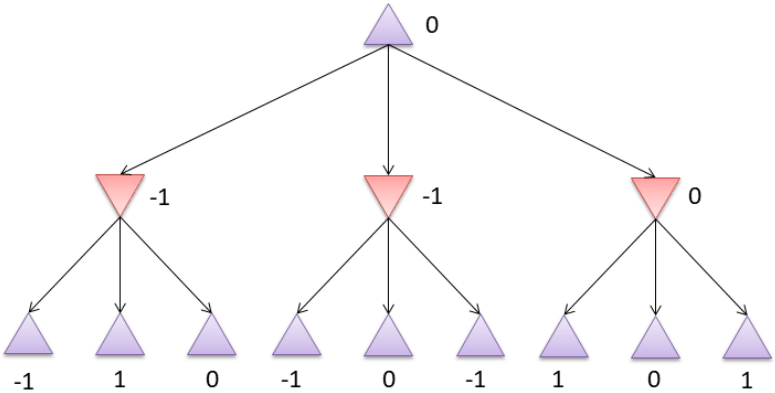
MiniMax algorithm : step by step



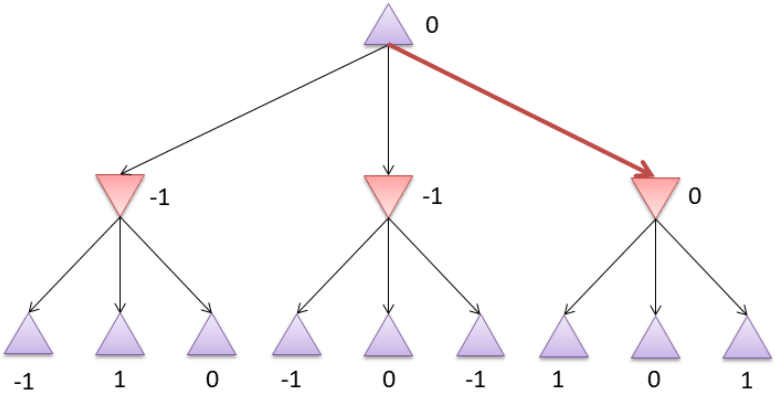
MiniMax algorithm : step by step



MiniMax algorithm : step by step



MiniMax algorithm : step by step



MiniMax algorithm : problem

Problem

In practice, it is difficult to reach the leaves of the tree (i.e. the end of the game).

MiniMax algorithm : problem

Problem

In practice, it is difficult to reach the leaves of the tree (i.e. the end of the game).

Solution

Limit the depth of the search !

Introduction of heuristics

Idea

- The tree is extended only until a given depth p : the leaves are not the final positions anymore.
- We need to evaluate the positions : the information *lost* or *won* is not available anymore.

Introduction of heuristics

Idea

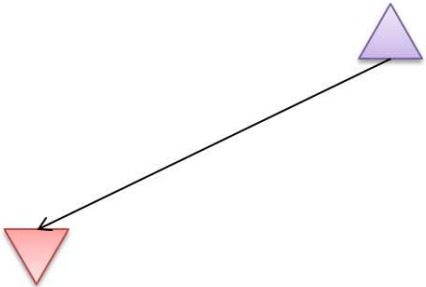
- The tree is extended only until a given depth p : the leaves are not the final positions anymore.
- We need to evaluate the positions : the information *lost* or *won* is not available anymore.

Definition

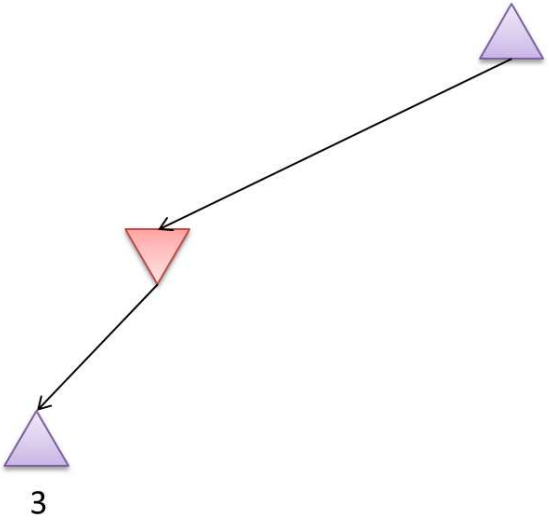
Game heuristics are functions that associates a real to a board. In adversarial games, the higher the heuristic, the closer the victory. Conversely, the more negative the heuristic, the closer the defeat.

MiniMax algorithm : step by step

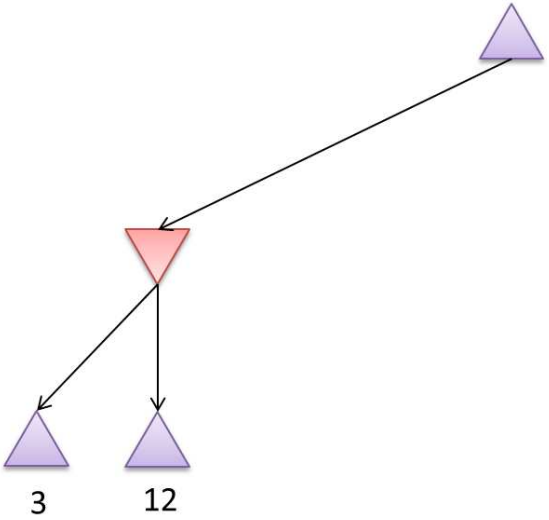
MiniMax algorithm : step by step



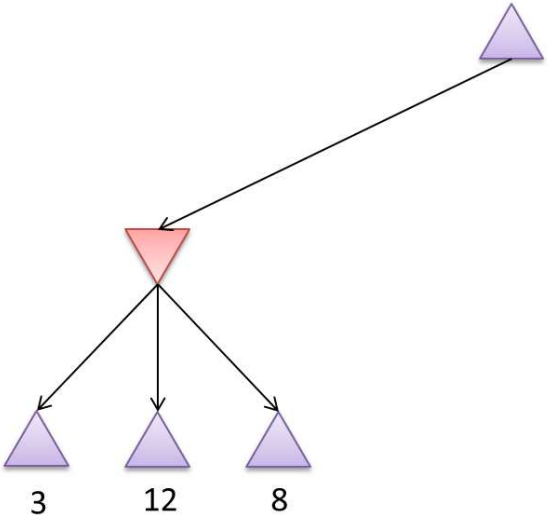
MiniMax algorithm : step by step



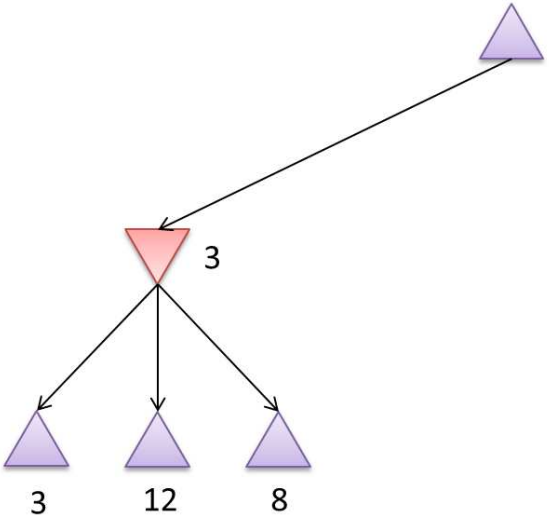
MiniMax algorithm : step by step



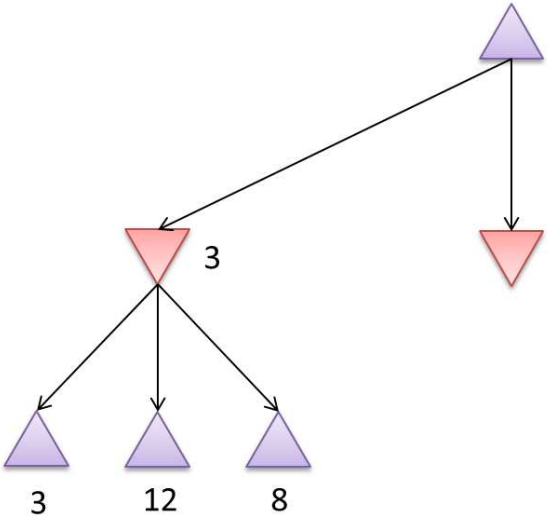
MiniMax algorithm : step by step



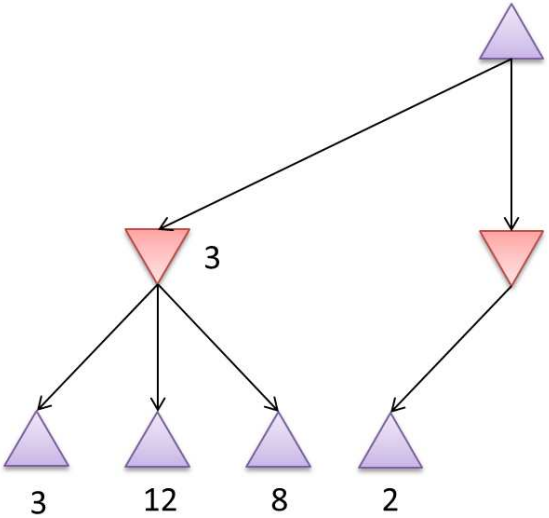
MiniMax algorithm : step by step



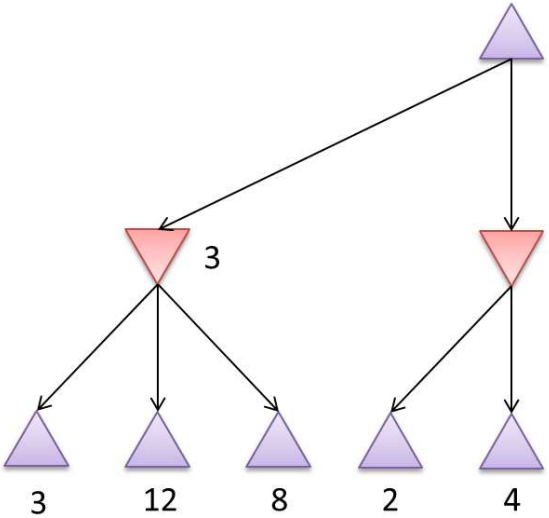
MiniMax algorithm : step by step



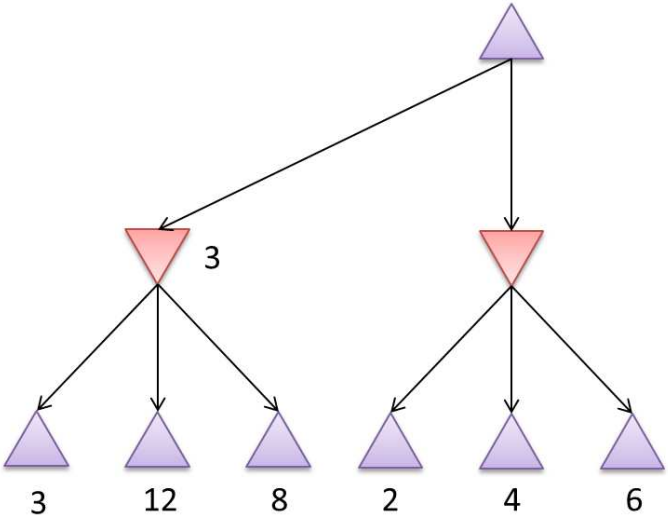
MiniMax algorithm : step by step



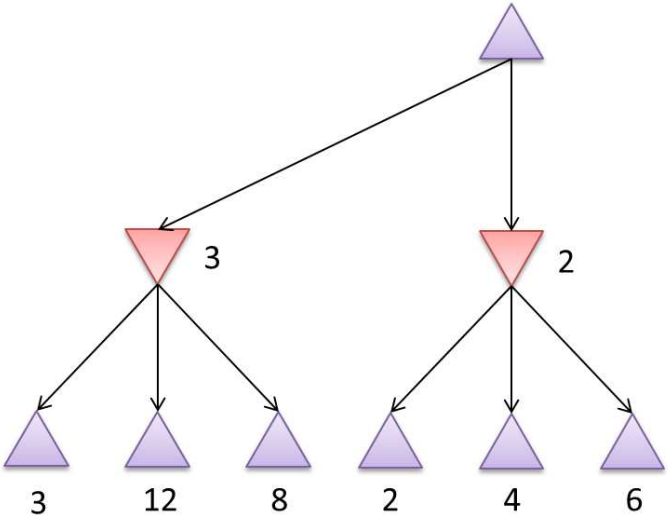
MiniMax algorithm : step by step



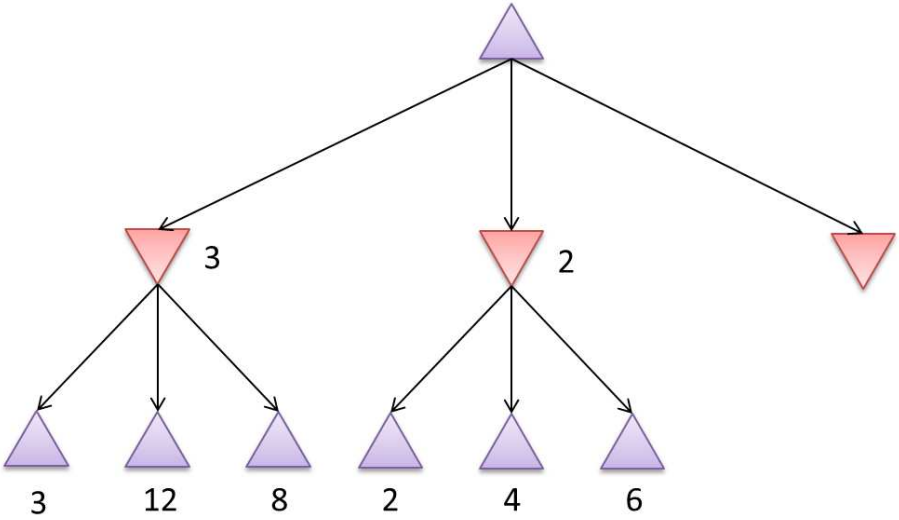
MiniMax algorithm : step by step



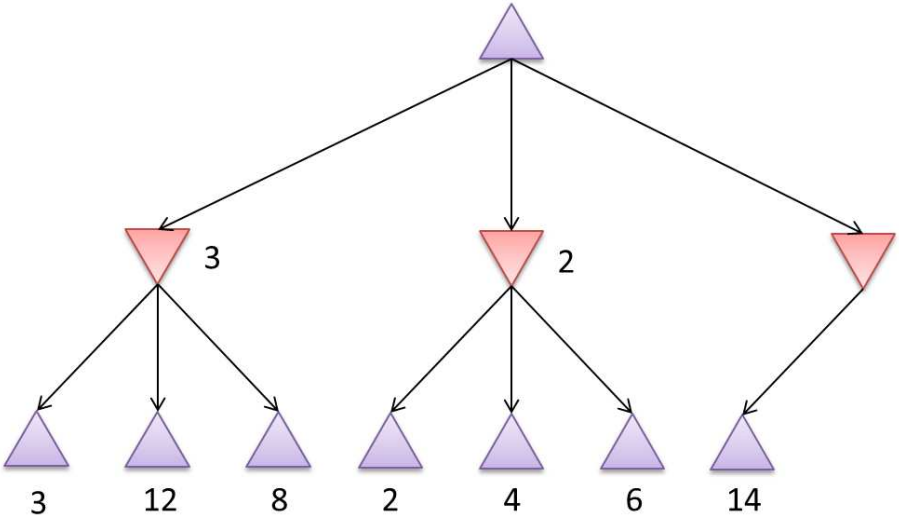
MiniMax algorithm : step by step



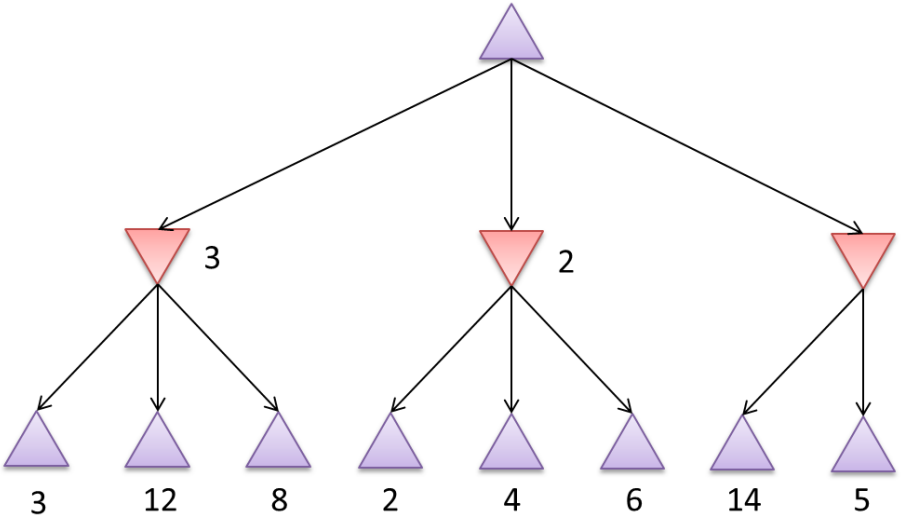
MiniMax algorithm : step by step



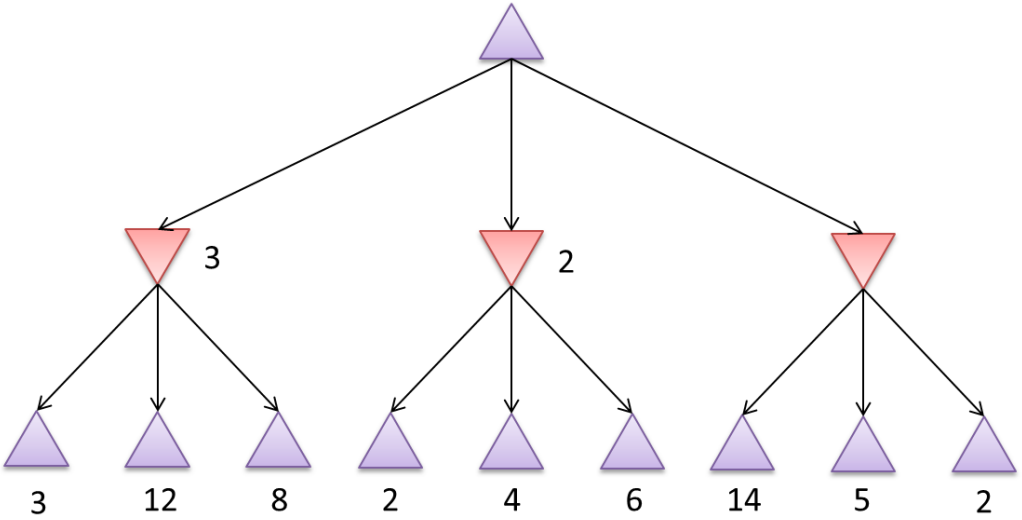
MiniMax algorithm : step by step



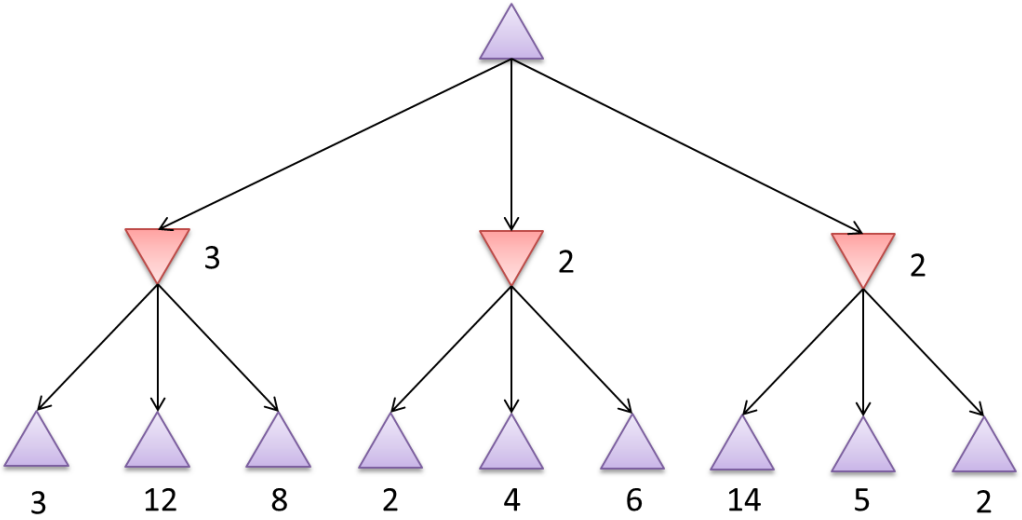
MiniMax algorithm : step by step



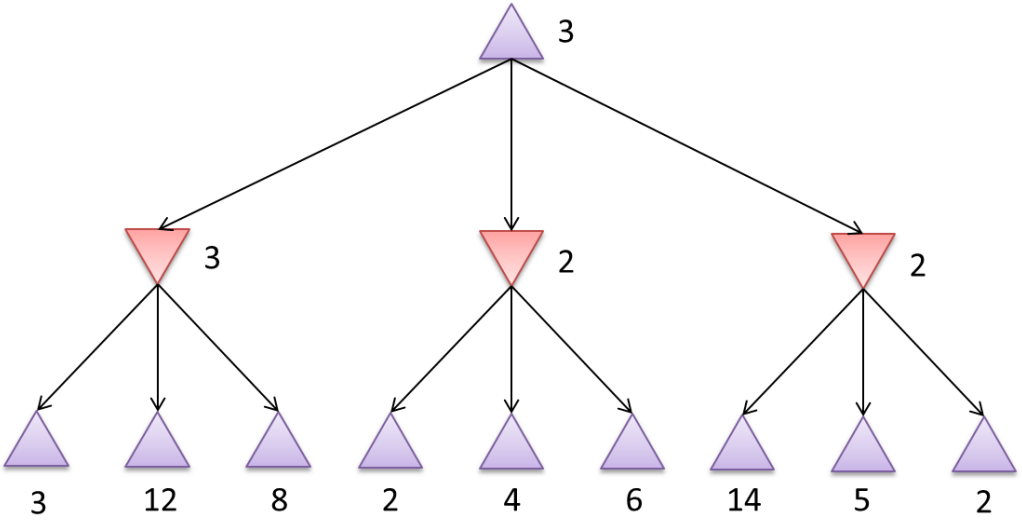
MiniMax algorithm : step by step



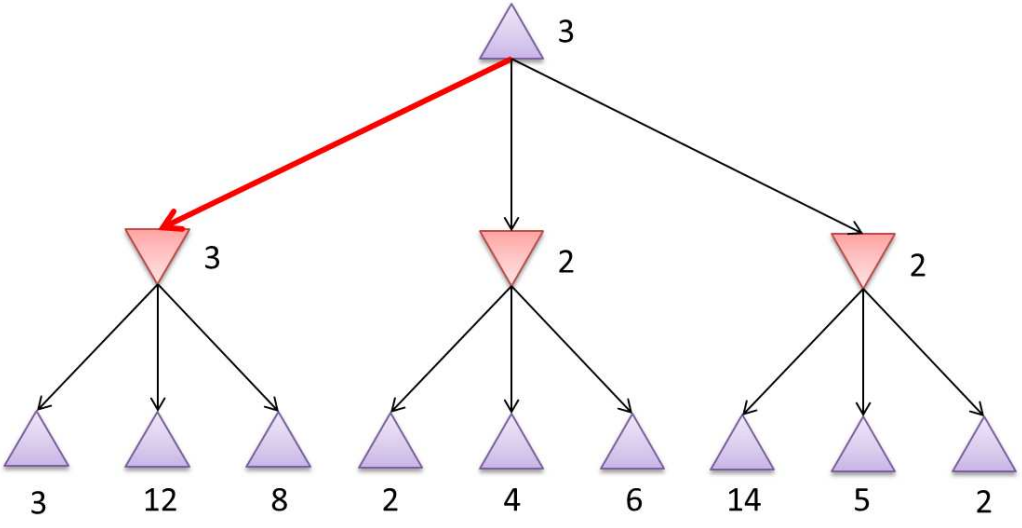
MiniMax algorithm : step by step



MiniMax algorithm : step by step



MiniMax algorithm : step by step



Algorithm properties

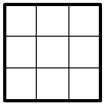
Properties

- Completeness
  Yes, if the game tree is finite
- Time complexity
  $o(b^m)$ (depth-first browsing) b : branching factor. m : maximum depth
- Space complexity : $o(bm)$
- Optimality : Yes, if the opponent is also optimal

Evaluation function for a board

- $e(s) = +\infty$ if s is a winning board for Max
- $e(s) = -\infty$ if s is a winning board for Min
- $e(s)$ measures the situation from the point of view of Max .

e(s) = # lignes/cols/diags ouvertes pour Max
- # lignes/cols/diags ouvertes pour Min

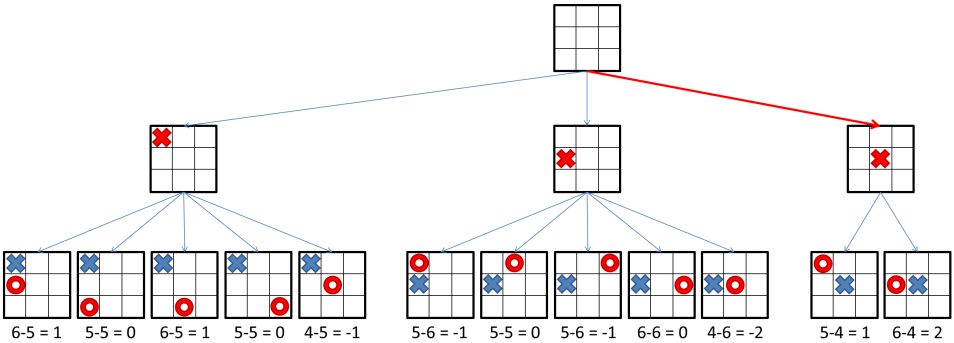


8 – 8 = 0

6 – 4 = 2

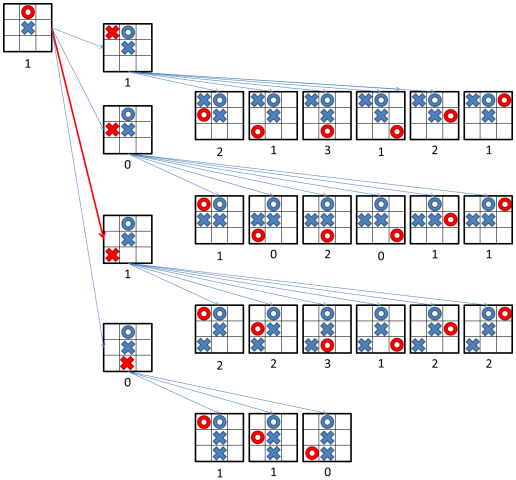
3 – 3 = 0

Example : Tic Tac Toe with depth 2



From http ://cui.unige.ch/DI/cours/IA

Example : Tic TAc Toe with depth 2



From http ://cui.unige.ch/DI/cours/IA

MiniMax algorithm with limited depth

The maximum depth depends on several criteria, as the capacity of the machine, the time to play, the moment in the game.
As an illustration, this is the depth regarding the level of the AI at chess :

- 4 plys : beginner
- 8 plys : master
- 12 plys : Deep Blue

MiniMax algortihm

Importance of the evaluation function

Classically, it's a linear function that weights the value of each item on the board.

Example of chess evaluation function

$$\begin{aligned} f(P) &= 200(K-K') + 9(Q-Q') + 5(R-R') + 3(B-B'+N-N') + (P-P') \\ &\quad -\frac{1}{2}(D-D'+S-S'+I-I') \\ &\quad +0.1(M-M') \dots \end{aligned}$$

with :

- K, Q, R, B, N, P the number of kings, queens, rooks, bishops, knights and pawns for the player (prime stands for the opponent's items) ;
- D, S, I doubled, backward and isolated pawns ;
- M is a measure of mobility (e.g., the number of legal moves for the player).

MiniMax algorithm : problem

Problem (still !)

In practice, the game tree is too wide to be browsed with MiniMax.

MiniMax algorithm : problem

Problem (still !)

In practice, the game tree is too wide to be browsed with MiniMax.

Solution

Prune some branches to decrease the branching factor.

Alpha-Beta pruning

Admissible pruning

The goal is to have the same backpropagated values at the root, but without browsing the whole game tree. The idea is to prune branches of the search tree which have no consequence for the evaluation of the nodes.

Idea

Let n be a node of the game tree such as it is player P turn. If a better choice of action has been browsed before, the node n will never be chosen.

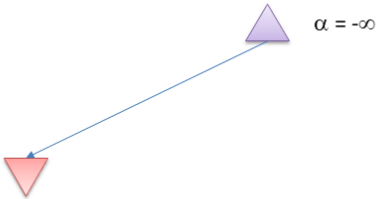
Alpha-Beta pruning

Principle

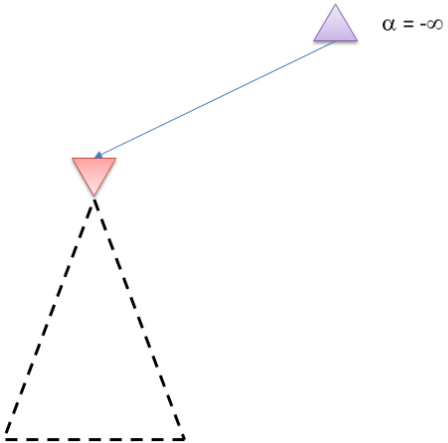
- Extend the game tree until a depth d
- Do not extend a node if it will never be chosen.
  - Each Max node keeps the trace of a α-value, i.e. the value of its best successor yet. It's the best choice for Max for the current path. The α-value is ascending.
  - Each Min node keeps the trace of a β-value, i.e. the value of its best successor yet. It's the best choice for Min for the current path. The β-value is descending.
- The branch is cut whenever α-value is greater than β-value

Alpha-Beta pruning

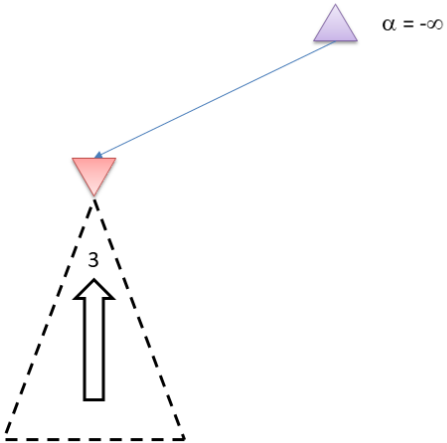
Alpha-Beta pruning



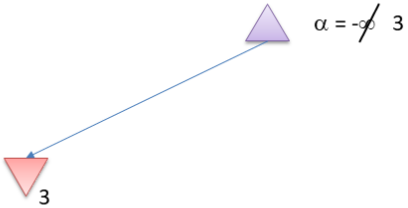
Alpha-Beta pruning



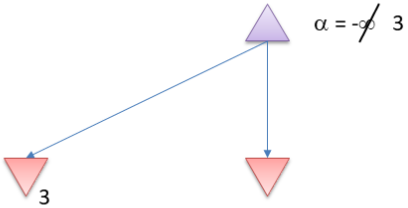
Alpha-Beta pruning



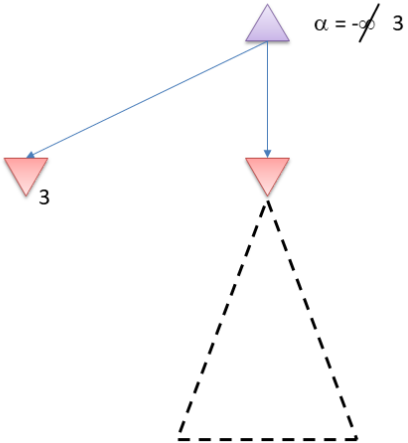
Alpha-Beta pruning



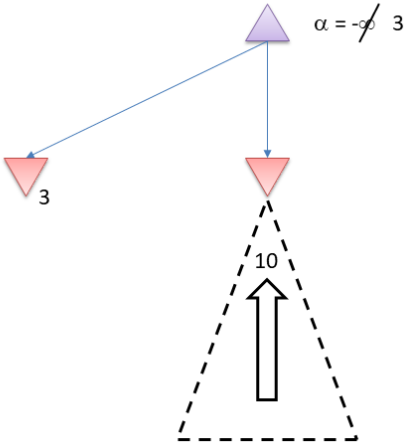
Alpha-Beta pruning



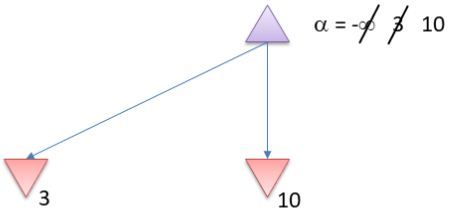
Alpha-Beta pruning



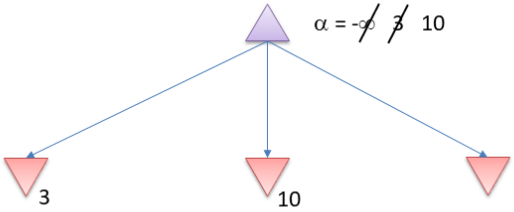
Alpha-Beta pruning



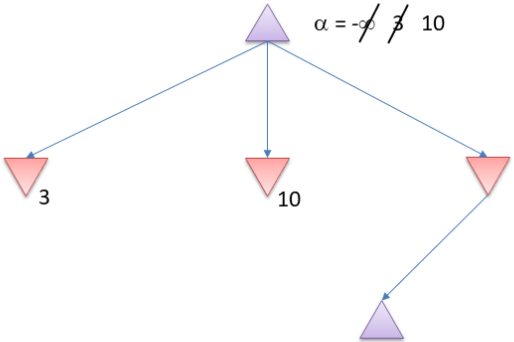
Alpha-Beta pruning



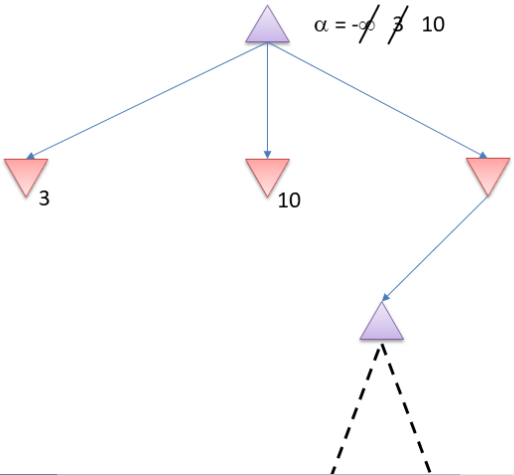
Alpha-Beta pruning



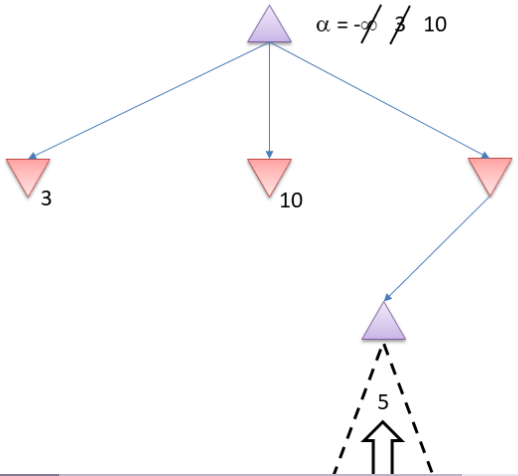
Alpha-Beta pruning



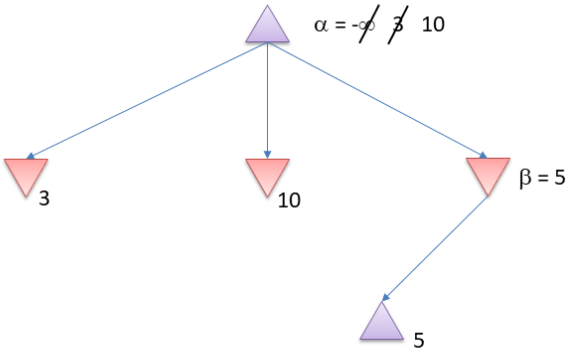
Alpha-Beta pruning



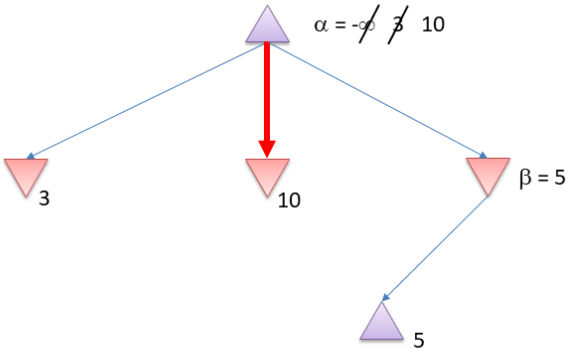
Alpha-Beta pruning



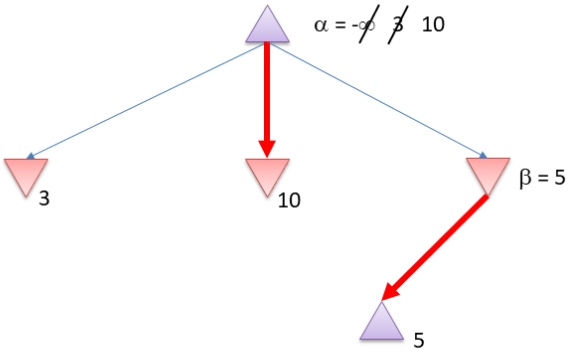
Alpha-Beta pruning



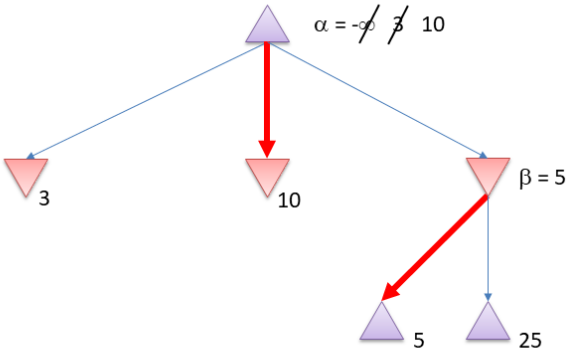
Alpha-Beta pruning



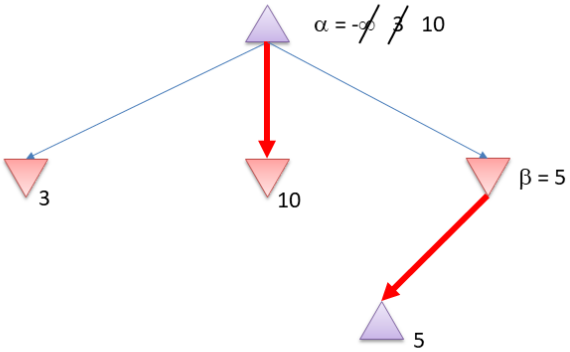
Alpha-Beta pruning



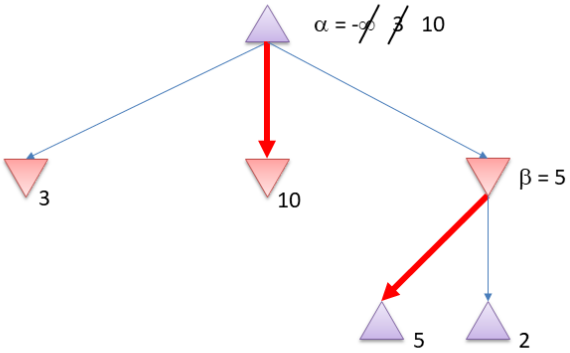
Alpha-Beta pruning



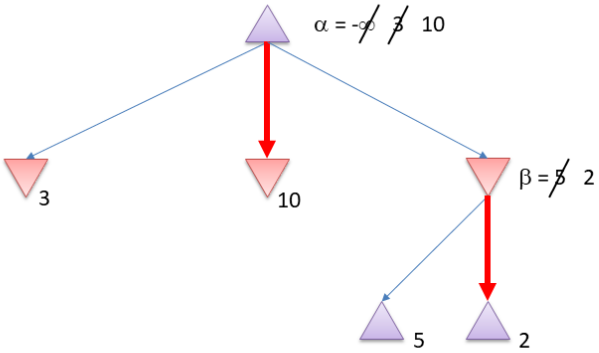
Alpha-Beta pruning



Alpha-Beta pruning

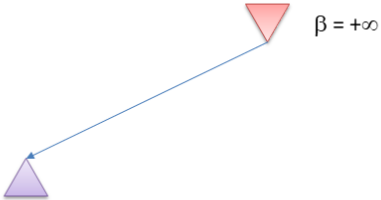


Alpha-Beta pruning

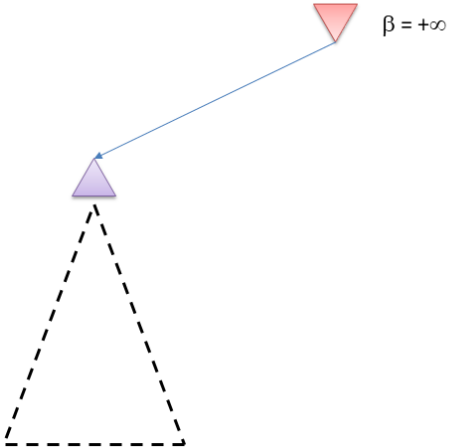


Alpha-Beta pruning

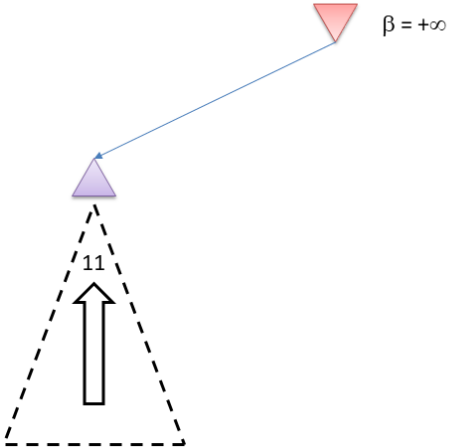
Alpha-Beta pruning



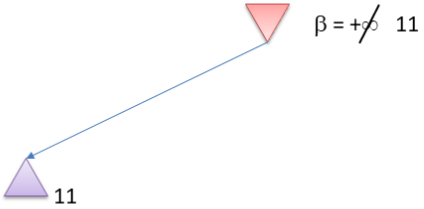
Alpha-Beta pruning



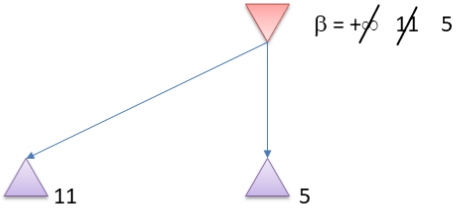
Alpha-Beta pruning



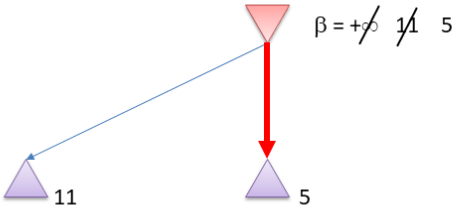
Alpha-Beta pruning



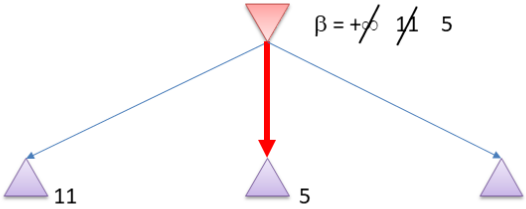
Alpha-Beta pruning



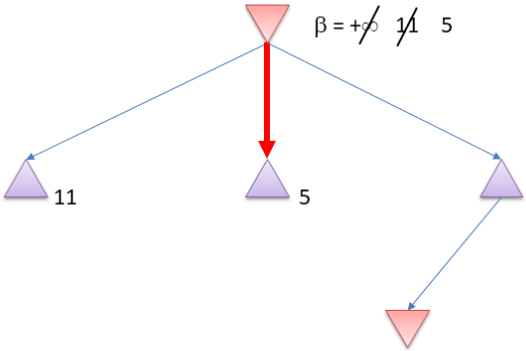
Alpha-Beta pruning



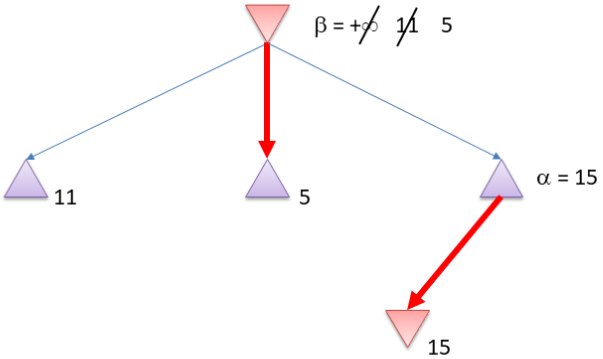
Alpha-Beta pruning



Alpha-Beta pruning



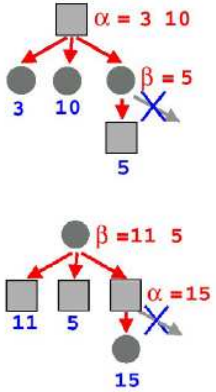
Alpha-Beta pruning



Alpha-Beta pruning

Two rules

1. Stop the search of a Max node if its α-value $\geq$ β-value of its parent node.
2. Stop the search of a Min node if its β-value $\leq$ α-value of its parent node.



Alpha-Beta pruning

Algorithm 1: MaxValue function

Function *MaxValue (state, α, β)*/* Evaluation at level FRIEND */

Data:
- *state* : current board
- α : best current evaluation of FRIEND
- β : best current evaluation of ENNEMY

begin
  if *IsLeaf(state)* **then**
    return *evaluate(state)*/* Heuristic evaluation */
  end
  $v \longleftarrow -\infty$
  forall *successor s of state* **do**
    $v \longleftarrow \max(v, \mathit{MinValue}(s, \alpha, \beta))$
    if $v \geq \beta$ **then**
      return v/* cut */
    end
    $\alpha \longleftarrow \max(\alpha, v)$
  end
  return v
end

Alpha-Beta pruning

Algorithm 2: MinValue function

Function *MinValue (etat, α, β)*/* Evaluation at level ENNEMY */

Data:
- *state* : current board
- α : best current evaluation of FRIEND
- β : best current evaluation of ENNEMY

begin
  if *IsLeaf(state)* **then**
    return *evaluate(state)*/* Heuristic evaluation */
  end
  $v \longleftarrow +\infty$
  forall *successor s of state* **do**
    $v \longleftarrow \min(v, MaxValue(s, \alpha, \beta))$
    if $v \geq \beta$ **then**
      return v/* cut */
    end
    $\beta \longleftarrow \min(\beta, v)$
  end
  return v
end

Alpha-Beta pruning

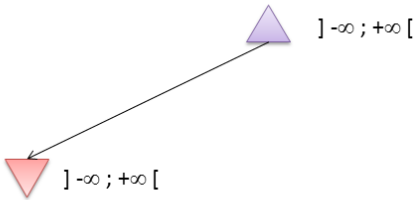
Principle

- To evaluate a board, we call *MaxValue* with the current board.
- Initially, we set $\alpha = -\infty$ and $\beta = +\infty$
- α and β are local to each function.

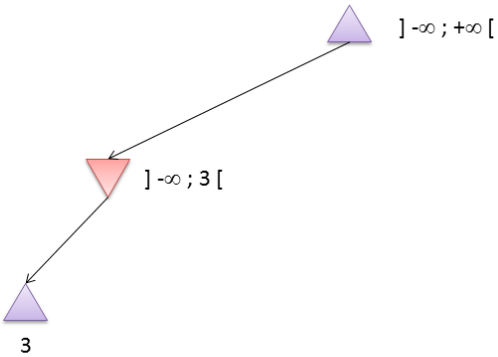
Example

] -∞ ; +∞ [

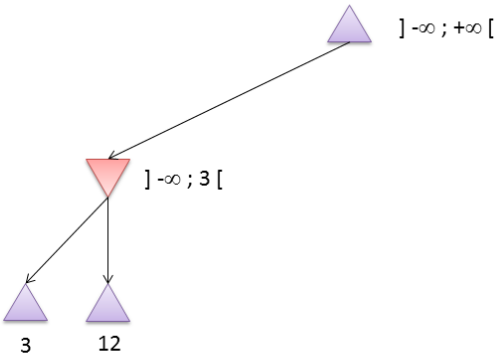
Example



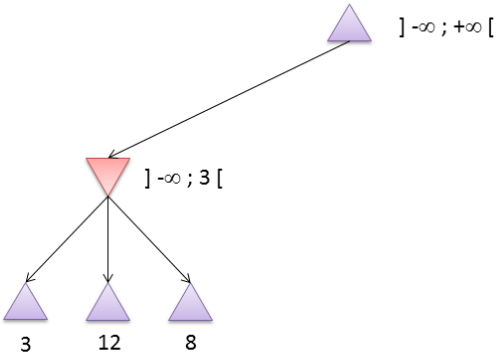
Example



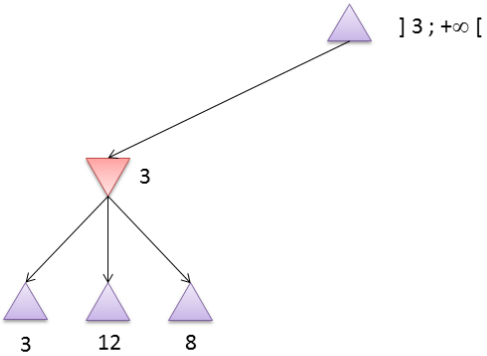
Example



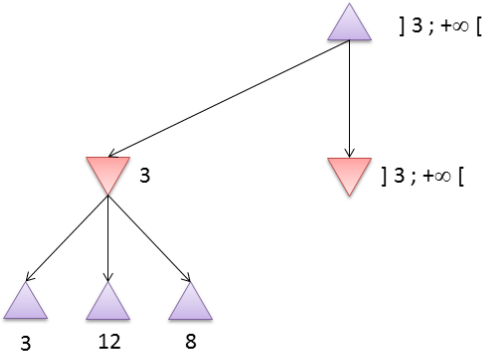
Example



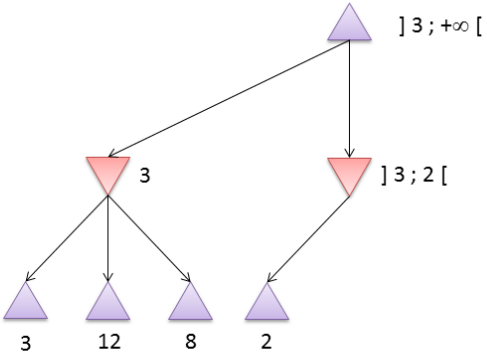
Example



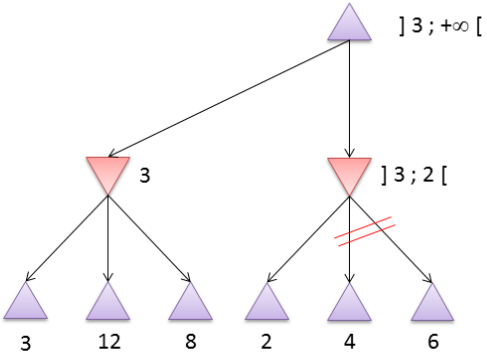
Example



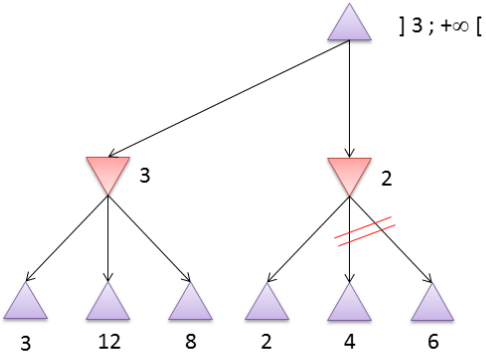
Example



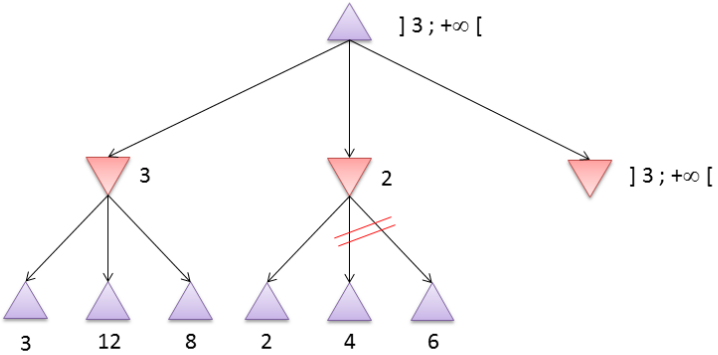
Example



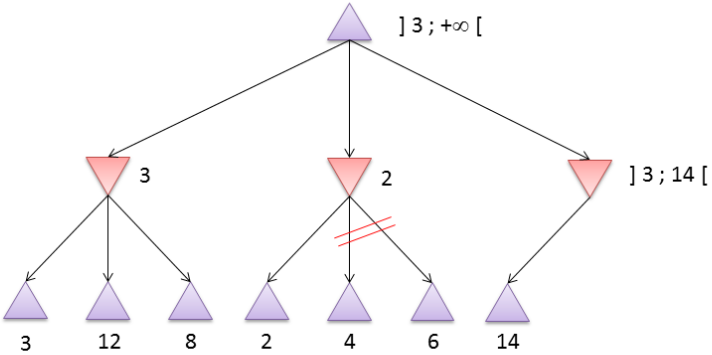
Example



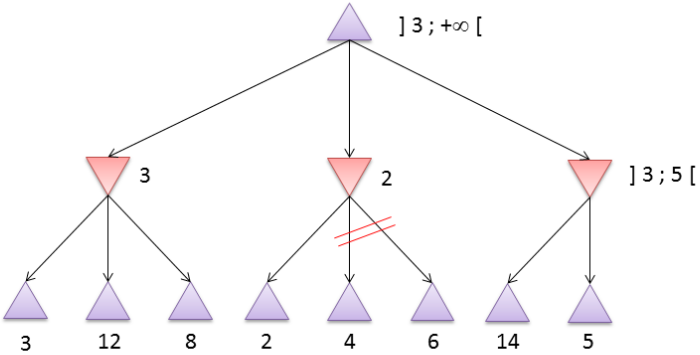
Example



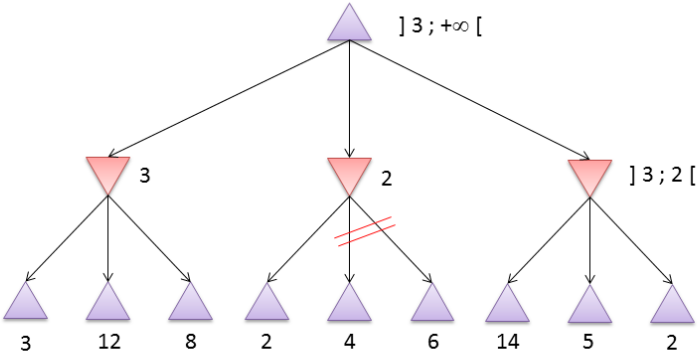
Example



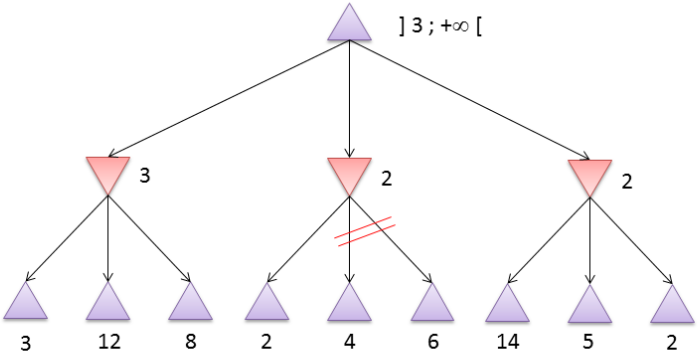
Example



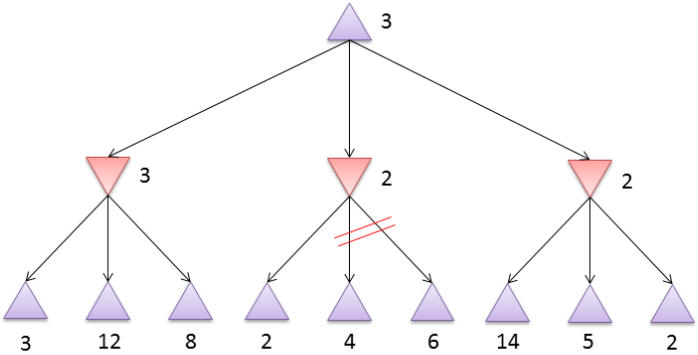
Example



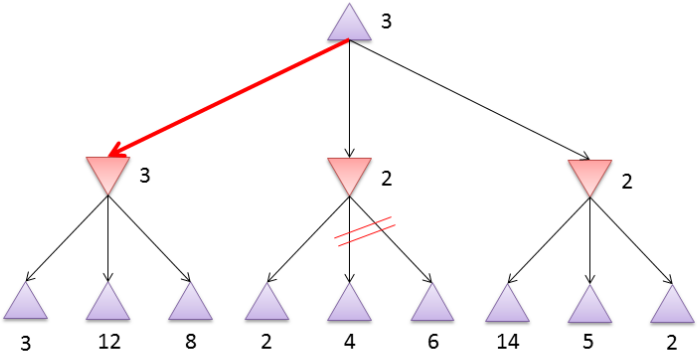
Example



Example



Example



Properties of Alpha-Beta

Efficiency

Let assume that the nodes are almost ideally sorted, the time complexity of the algorithm is $O(b^{\frac{d}{2}})$ instead of $O(b^d)$

- Theoretical branching factor $\sqrt{b}$
- We can extend twice more the game tree.

Properties of Alpha-Beta

Further reading

- Negamax : instead of having 2 functions, we use only one but we use the opposite of the value at each level
- $\alpha\beta$ pruning within a restrained window
- SSS : breath-first browsing

Comparison of the algorithms

Number of browsed nodes and execution time (without unit) for othello game.

p	minimax	$\alpha - \beta$	SSS*
1	3 - 0.37	3 - 0.36	3 - 0.36
2	14 - 1.38	13 - 1.4	11 - 1.19
3	61 - 6.0	37 - 3.9	35 - 3.7
4	349 - 32.5	150 - 14.77	95 - 10.0
5	2050 - 185.7	418 - 41.4	292 - 19.2
6	13773 - 1213.5	1830 - 172.9	1617 - 117.3

Humans VS Machines

Are the machines better at games ?

- Some examples of games at which computers are better :
  - Checkers
  - Othello
  - Scrabble
  - Chess
  - Backgammon
  - Go
- New challenge :
  - Bridge

Outline

Introducion

The last extension of searching emphasizes the expressivity of the states.
In the previous algorithms, the states are just symbols. What if now states could be described by variables ? What kind of problems can be formalized like this ?

Constraint Satisfaction Problems (CSP)

Definition

A constraint satisfaction problem CSP is defined by :

- A set of variables $\{X_1, X_2, ..., X_n\}$.
  - Each variable has a domain D_i of possible values (often discrete and finite).
- A set of constraints $\{C_1, C_2, ..., C_p\}$.
  - Each constraint involved a subset of variables and specify the combination of possible values for this subset.

Constraint Satisfaction Problems (CSP)

Constraints can be defined :

- by extension : for a given constraint, we give all the possible values for the variables
- by intension : the values are deducted from a mathematical relation, e.g. for 2 variables x and y defined on booleans, the constraint $x \neq y$

Example

Let x and y be two boolean variables. We want to reprensent the constraint $c =$ "x is different from y".

- by extension : $c = \{(x = true, y = false), (x = false, y = true)\}$
- by intension : $c = x \neq y$

Constraint Satisfaction Problems (CSP)

A state of the problem is defined by an affectation of values for a subset of variables.

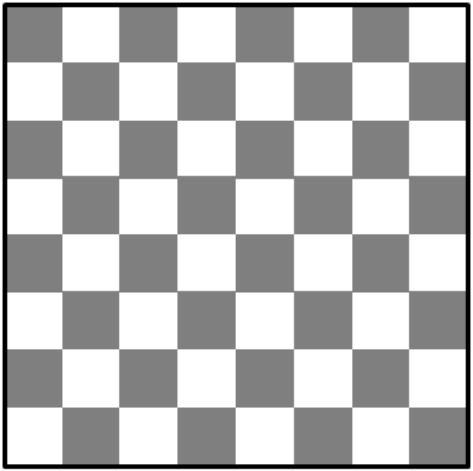
- if no constraint is violated, state is consistent.
- if all the variables are affected, state is complete.
- if the state is consistent and complete, it is a solution.

Goal

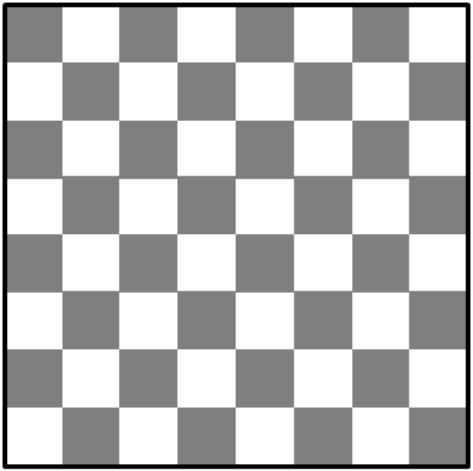
Affect a value to each variable regarding the satisfaction of all the constraints.

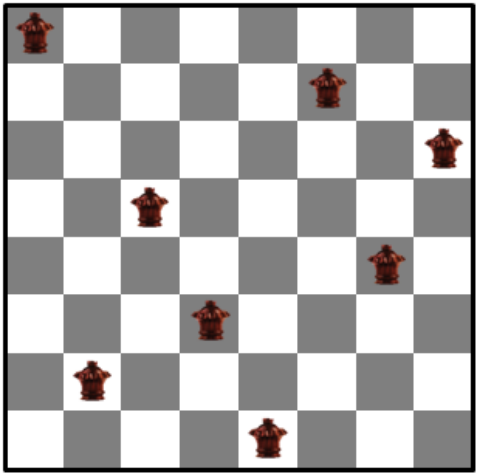
CSP allows general-purpose algorithms with more power that standard search algorithms.

Example 1 : 8-queens problem



Example 1 : 8-queens problem





Example 1 : 8-queens problem

Solution 1

- **Variables** : 64 cells $X_{i,j}$ with $1 \leq i, j \leq 8$.
- **Domains** : $D_{i,j} = \{0, 1\}$.
- **Constraints** :
  - $X_{i,j} = 1 \rightarrow X_{i,k} = 0 \; \forall k \in [1, 8], \; k \neq j$ (line)
  - $X_{i,j} = 1 \rightarrow X_{k,j} = 0 \; \forall k \in [1, 8], \; k \neq i$ (column)
  - $X_{i,j} = 1 \rightarrow X_{k,k+(j-i)} = 0 \; \forall k \in [1, 8], \; k \neq i, 1 \leq k + (j - i) \leq 8$ (diagonal)
  - $X_{i,j} = 1 \rightarrow X_{k,k+(j+i)} = 0 \; \forall k \in [1, 8], \; k \neq i, 1 \leq k + (j + i) \leq 8$ (diagonal)
  - $\sum X_{i,j} = 8$

Example 1 : 8-queens problem

Solution 2

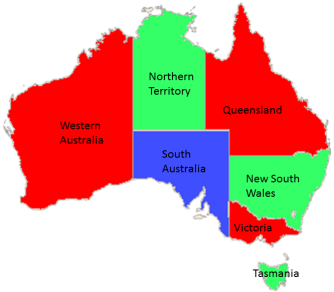
- **Variables** : 8 queens X_i with $1 \leq i \leq 8$, i is column number
- **Domains** : $D_{i,j} = \{1, 2, 3, 4, 5, 6, 7, 8\}$.
- **Constraints** :
  - $X_i = k \rightarrow X_j \neq k \; \forall j \in [1, 8],\ j \neq i$ (line)
  - $X_i = j \rightarrow X_k \neq k + (j - i) \; \forall k \in [1, 8],\ k \neq i, 1 \leq k + (j - i) \leq 8$ (diagonal)
  - $X_i = j \rightarrow X_k \neq k + (j + i) \; \forall k \in [1, 8],\ k \neq i, 1 \leq k + (j + i) \leq 8$ (diagonal)

Example 2 : Map coloration problem

Problem position

- **Goal** : colour the map.
- **Variables** : $\{WA, NT, SA, Q, NSW, V, T\}$
- **Domains** : {Red, Green, Blue }
- **Constraints** : adjacent regions must have different colors
  $\{WA \neq NT, WA \neq SA, NT \neq Q, NT \neq SA, Q \neq SA, Q \neq NSW, SA \neq NSW, SA \neq V, NSW \neq V\}$

Example 2 : Map coloration problem



Solutions are affectations that satisfy all constraints, e.g. :
$\{ WA = red, NT = green, SA = blue, Q = red, NSW = green, V = red, T = green \}$

Example 3 : houses problem

We have 5 houses :

- Variables :
  - Nationalities : $N_i = \{$english, spanish, japanese, italian, norwegian$\}$.
  - Colors : $C_i = \{$red, green, white, yellow, blue$\}$
  - Drinks : $B_i = \{$tea, coffee, milk, juice, water$\}$
  - Jobs : $P_i = \{$painter, sculptor, diplomat, violonist, doctor $\}$
  - Animals : $A_i = \{$dog, snail, fox, horse, zebra$\}$

Example 3 : houses problem

- **Constraints** :
  - The English man lives in the red house
  - The Spanish has a dog
  - The Japanese is a painter
  - The Italian drinks tea
  - The Norwegian lives in the left-most house
  - The inhabitant of the green house drinks coffee
  - The green house is to the right the white house
  - The sculptor have snails
  - The diplomat lives in a yellow house
  - The inhabitant of the middle house drinks milk
  - The Norwegian is the neighbor of the blue house
  - The violinist drink juices
  - The fox is in the house next to doctor's house
  - The horse is in the house next to the diplomat's house.

Questions

- Who's got a Zebra ?
- Who drinks water ?

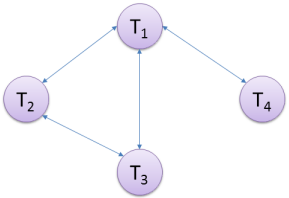
Example 3 : houses problem

- The English man lives in the red house
- The Spanish has a dog
- The Japanese is a painter
- The Italian drinks tea
- The Norwegian lives in the left-most house $\Rightarrow N_1 = Norwegian$
- The inhabitant of the green house drinks coffee
- The green house is to the right the white house
- The sculptor have snails
- The diplomat lives in a yellow house
- The inhabitant of the middle house drinks milk $\Rightarrow D_3 = Milk$
- The Norwegian is the neighbor of the blue house
- The violinist drink juices
- The fox is in the house next to doctor's house
- The horse is in the house next to the diplomat's house.

Example 3 : houses problem

- The English man lives in the red house $\Rightarrow C_1 \neq \mathit{Red}$
- The Spanish has a dog
- The Japanese is a painter
- The Italian drinks tea
- The Norwegian lives in the left-most house $\Rightarrow N_1 = \mathit{Norwegian}$
- The inhabitant of the green house drinks coffee
- The green house is to the right the white house
- The sculptor have snails
- The diplomat lives in a yellow house
- The inhabitant of the middle house drinks milk $\Rightarrow D_3 = \mathit{Milk}$
- The Norwegian is the neighbor of the blue house
- The violinist drink juices $\Rightarrow J_3 \neq \mathit{Violinist}$
- The fox is in the house next to doctor's house
- The horse is in the house next to the diplomat's house.

Example 4 : task ordering



- **Variables** : tasks T_1, T_2, T_3, T_4
- **Constraints** :
  - T_1 must be processed at the same time as T_3
  - T_2 must be finished before T_1 starts
  - T_2 must be parallel to T_3
  - T_4 must start before the end of T_1.
- **Questions :**
  - Is there a solution ?

Real world CSPs

- Assignement problems.
- Timetabling problems.
- Hardware configuration.
- Transportation scheduling.
- Factory scheduling.
- Circuit layout.
- Fault diagnosis.
- ...

many real-world problems involve real-valued variables.

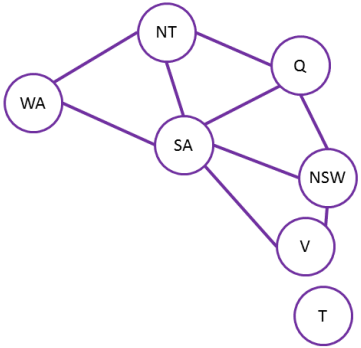
Types of problems

- Domains **finite/infinite**
  - Finite domains : set of values/symbols (boolean CSP)
  - Infinite domains : integers, strings (task ordering : variables are dates/times)
- Variables **discrete /continuous**
- Constraints **linear / non-linear**
- Cardinality of constraints :
  - Unary (*ex : SA =vert*)
  - Binary (*ex : SA* $\neq$ *WA*)
  - greater order : with 3+ variables
- Constraint **absolute / relative** : preferences over the constraints
- **Global constraints** : over all the CSP's varables (example : AllDiff).

Constraint graph

Definition

A constraint graph is a representation of CSPs. Nodes are variables and edges are constraints.



- 2 variables are adjacent if they are linked by constraint.
- Algorithms use this graph.

FIGURE – Constraint graph for map coloring

Formalization as a search problem

Problem position

- **Initial state** : no affected variable.
- **Successor function** : one variable is picked and affected regarding the constraints.
- **Goal** : all the variables must have a value.

Properties

- For n variables with domains of size d, we have $o(d^n)$ possible affectations.
- Algorithm is the same for all CSP (not problem-dependant).
- Each solution will appear at depth n
  $\rightarrow$ depth-first strategy is ok.

Formalization as a search problem

Problem position

- **Initial state** : no affected variable.
- **Successor function** : one variable is picked and affected regarding the constraints.
- **Goal** : all the variables must have a value.

Properties

- For n variables with domains of size d, we have $o(d^n)$ possible affectations.
- Algorithm is the same for all CSP (not problem-dependant).
- Each solution will appear at depth n
  $\rightarrow$ depth-first strategy is ok.
- There's no path in such a tree because the order in which variables are affected has no importance.

Algorithms for CSP

Since there is no order :

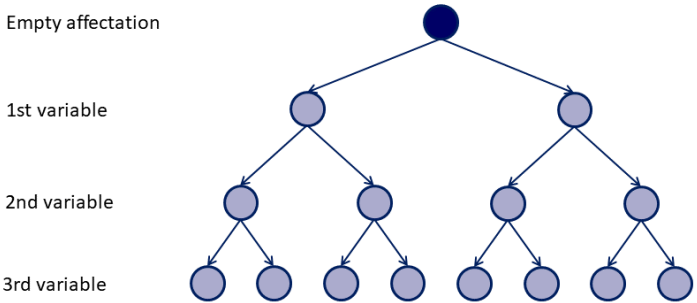
- We can extend a node by choosing one unaffected variable
- We don't memorize the path, but only the current node

The depth-first strategy for CSP is called backtracking.

Backtracking Search

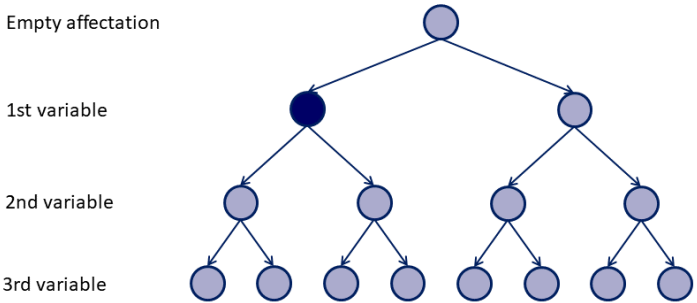
- Backtracking search is the basic uninformed algorithm for solving CSPs
- **Idea 1** : one variable at a time.
  - Variable assignments are commutative.
  - Only need to consider assigments to a single variable at each step.
- **Idea 2** : Check constraints as you go
  - Consider only values which do not conflict previous assigments

Backtracking



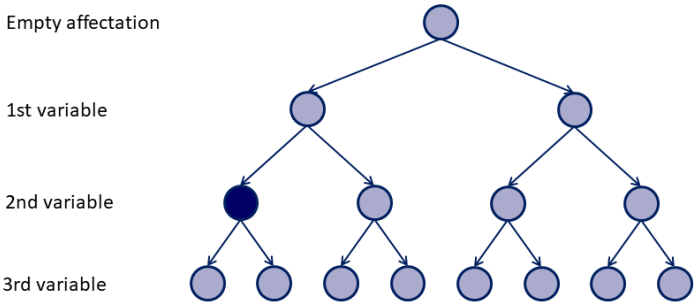
Affectation = {}

Bactracking



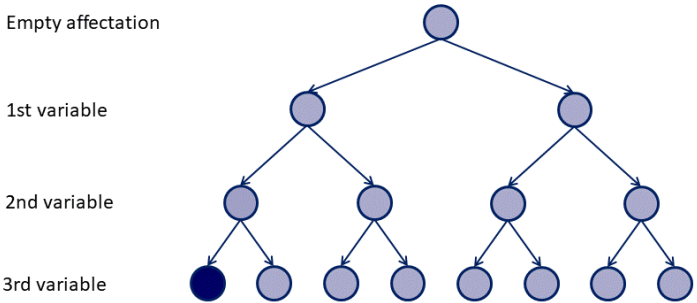
Affectation = { (var1=v11) }

Bactracking



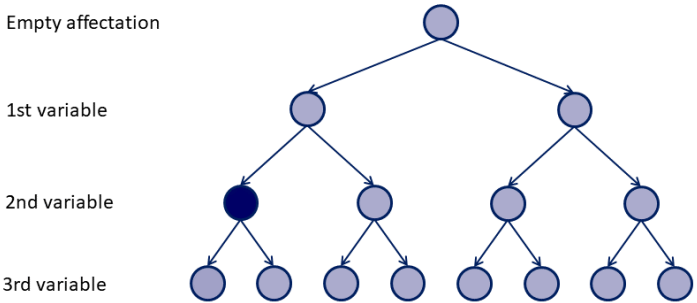
Affectation = { (var1=v11), (var2=v21) }

Bactracking



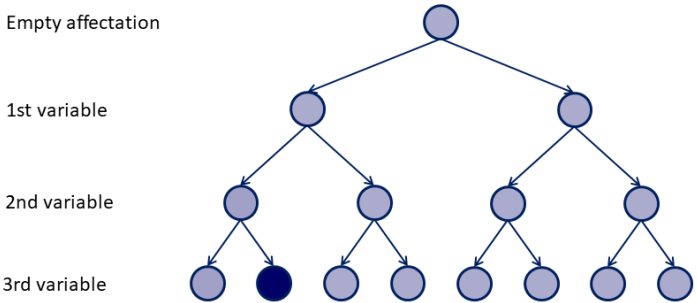
Affectation = { (var1=v11), (var2=v21), (var3=v31) }

Bactracking



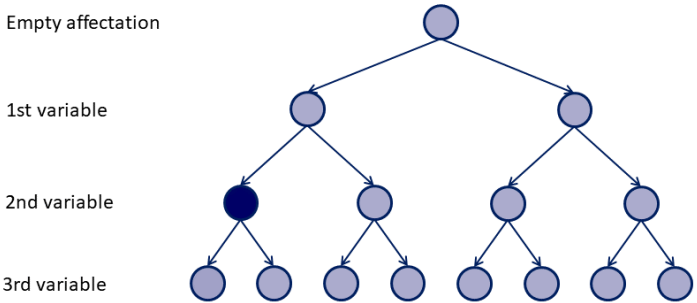
Affectation = { (var1=v11), (var2=v21) }

Bactracking



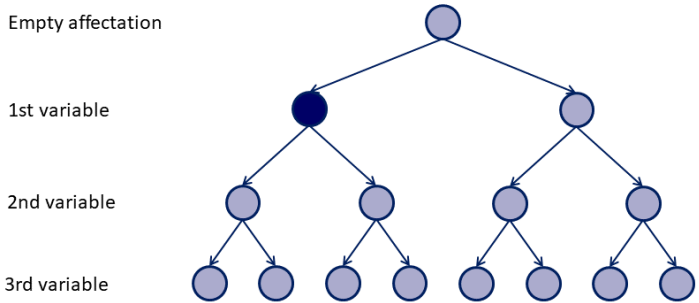
Affectation = { (var1=v11), (var2=v21), (var3=v32) }

Bactracking



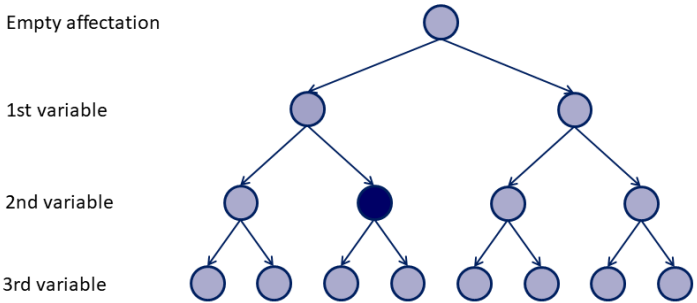
Affectation = { (var1=v11), (var2=v21) }

Bactracking



Affectation = { (var1=v11) }

Bactracking



Affectation = { (var1=v11), (var2=v22) }

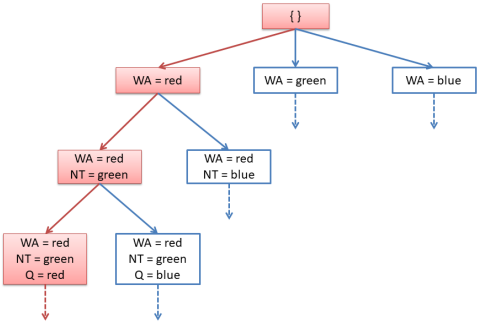
Bactracking

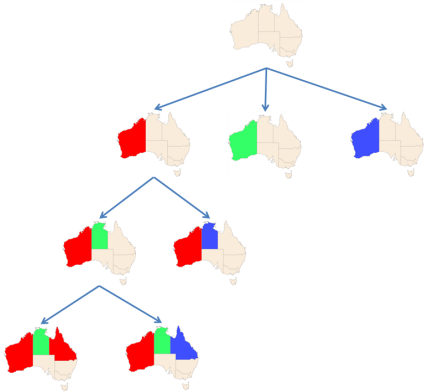
Principles

1. We affect a variable at each step
2. The affectation is commutative : the order doesn't count
3. The algorithm go back if no affectation are possible anymore.

Function *Backtrack (affectation)*
    begin
        if *affectation is complete* **then return** *affectation*
        $X \longleftarrow$ Select one unaffected variable
        $D \longleftarrow$ Define an order on D of X
        foreach $v \in D$ **do**
            if v *is consistent with affectation* **then**
                Add $X = v$ to *affectation*
                Result $\longleftarrow$ Backtrack(*affectation*)
                if *Result* $\neq$ *failure* **then return** Result
            end
        end
        return failure
    end

Bactracking : example





Bactracking : remarks

- Initial state and successor function are common to any CSP
- Is it possible to improve the performances of such an algorithm ? i.e. Can we find a generic heuristic ?

Bactracking : questions

- Which variable will be affected next ?
- In which order should we sort the values of the domains ?
- What are the implications of the last affectation on the unaffected variables ? (**constraint propagation**)
- Can we detect earlier a failure ? (**consistency**)

Minimal remaining values first

Heuristic

- **Principle** : Select the variable with the smallest number of possible values
- **Goal** : Minimize the number of remaining values (i.e, the branching factor)

Most constrained variable first

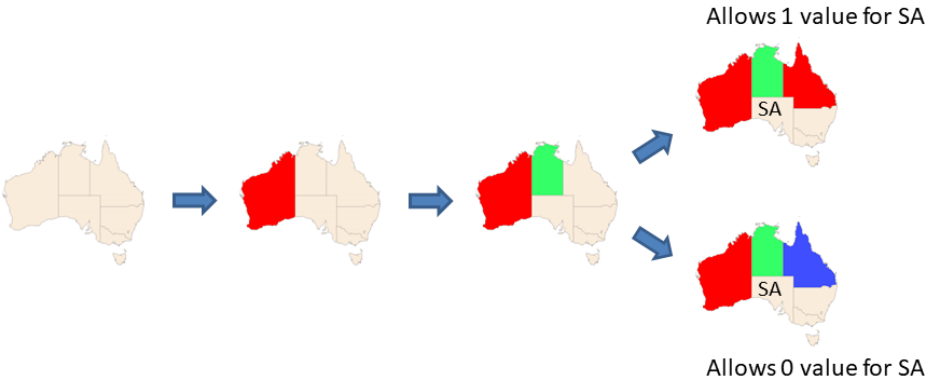
Heuristic

- **Principle** : select the variable which is implied in the greatest number of constraints.
- **Goal** : Minimize the number of remaining possible values.

Less constraining value first

Heuristic

- **Principle** : select the value that let the greatest number of values for unaffected variables.
- **Goal** : force the success



Failure detection

Principles

- Consider the constraints earlier instead of only after affectation
- Constraint propagation **Forward checking**

Forward checking

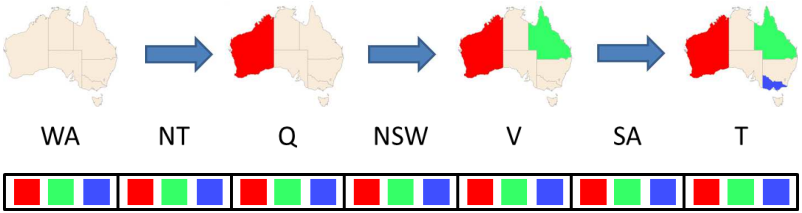
- When a value x is affected to a variable X :
  - Find every unassigned variable Y linked to X by a constraint.
  - Remove from Y's domain every value that is not consistent with x.
- Search ends when a variable has an empty domain.

Forward checking : example

We keep, at each step, all the possible values for each variable.

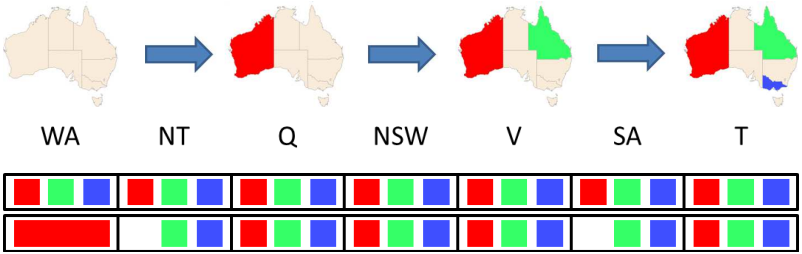
Forward checking : example

We keep, at each step, all the possible values for each variable.



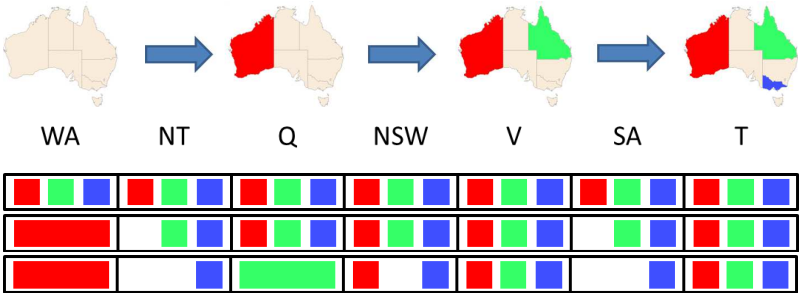
Forward checking : example

We keep, at each step, all the possible values for each variable.



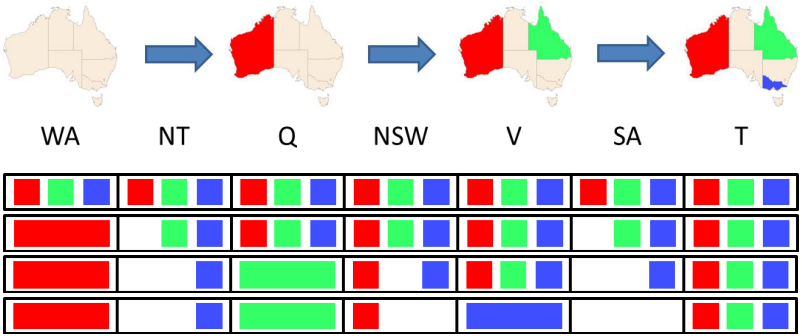
Forward checking : example

We keep, at each step, all the possible values for each variable.



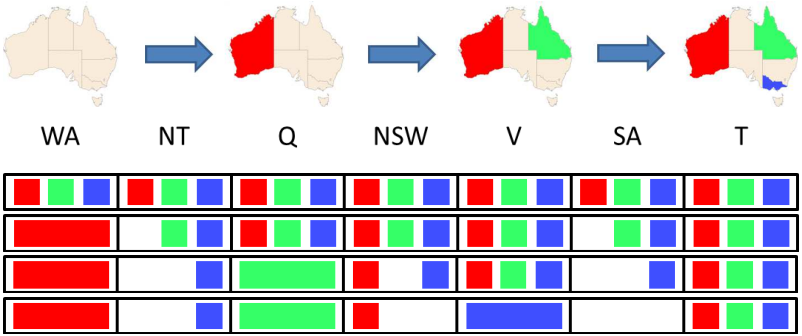
Forward checking : example

We keep, at each step, all the possible values for each variable.



Forward checking : example

We keep, at each step, all the possible values for each variable.

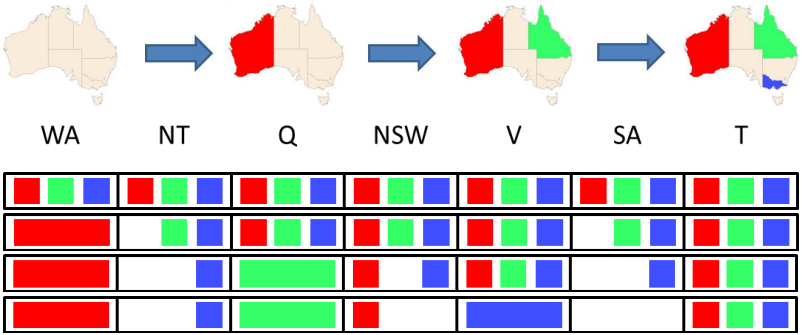


- 4th line : blocking situation : current affectation (WA =R ; Q= V ; V= B) does not satisfy all constraints. There is no color for SA anymore, so we must backtrack.

Constraint propagation

Problem

Forward checking propagates the information from assigned variables to unassigned variables, but it does not perform early-detection of failures.



- NT and SA cannot be both blue.
- Constraint propagation reinforces local constraint very fast.

Arc consistency

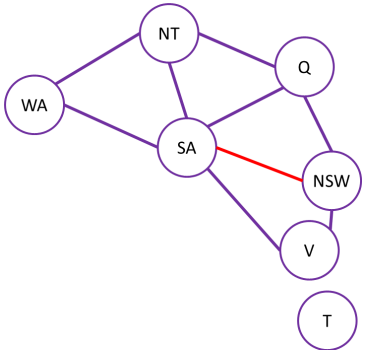
Principle

Arc consistency is another method of constraint propagation, ideal for failure early-detection.

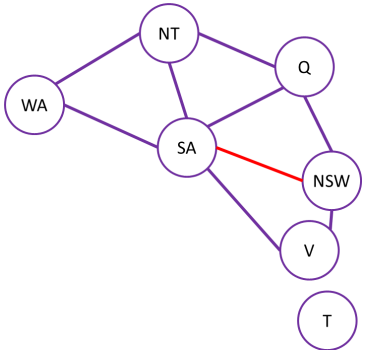
- **Arc** : oriented edge representing a constraint between two variables
- **Consistency** : the edge $X \rightarrow Y$ is **consistent** if for each remaining value of X, there exists for Y at least one value that satisfies the constraint.

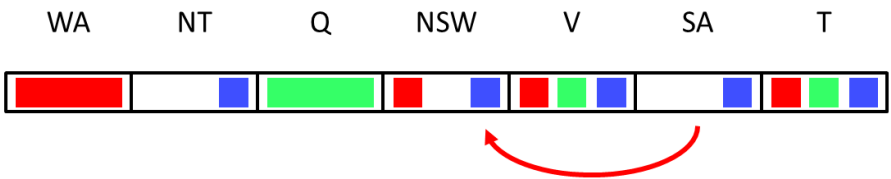
The simplest version of constraint propagation tries to make each edge of the constraint graph consistent.

Arc consistency : example

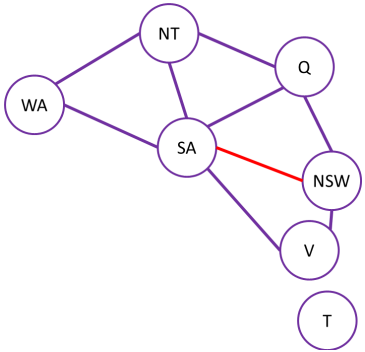


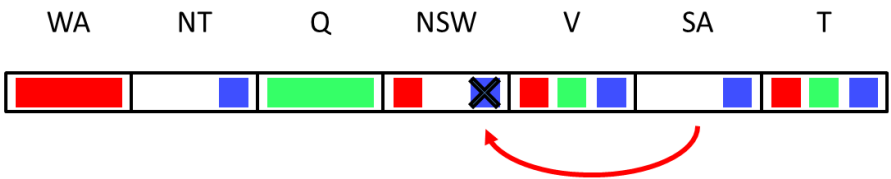
Arc consistency : example



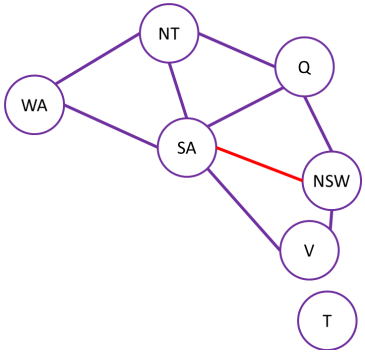


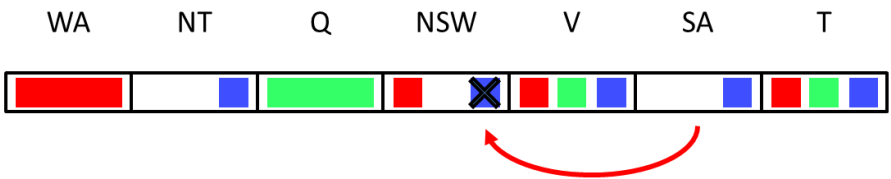
Arc consistency : example



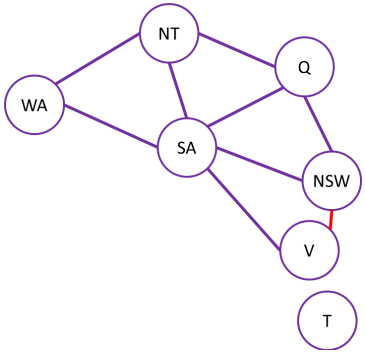


Arc consistency : example

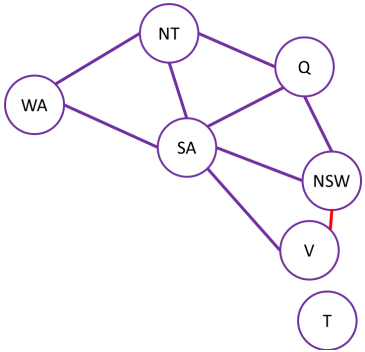


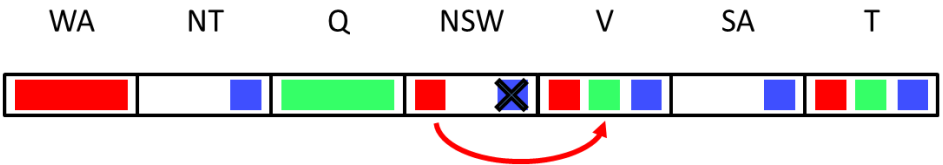


Arc consistency : example

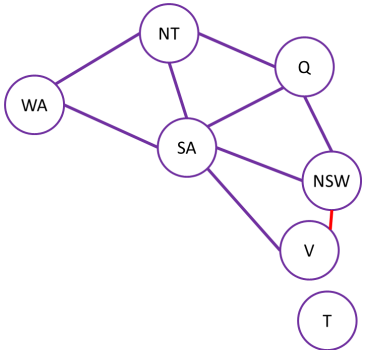


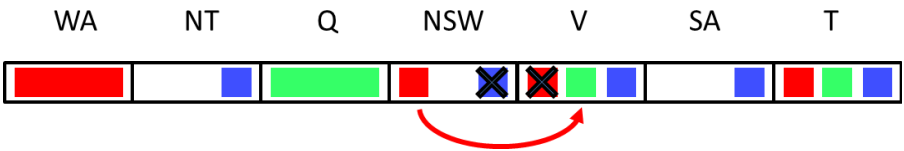
Arc consistency : example



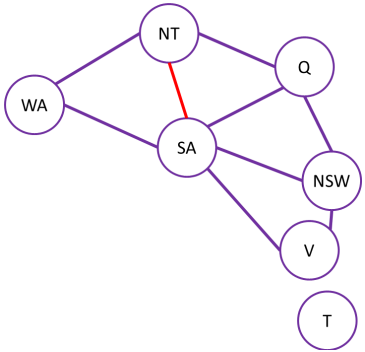


Arc consistency : example

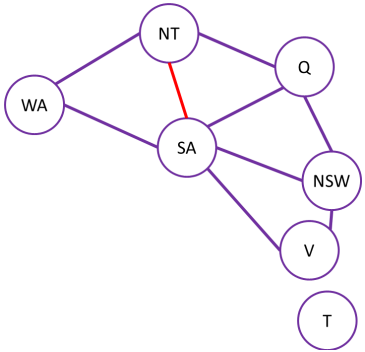


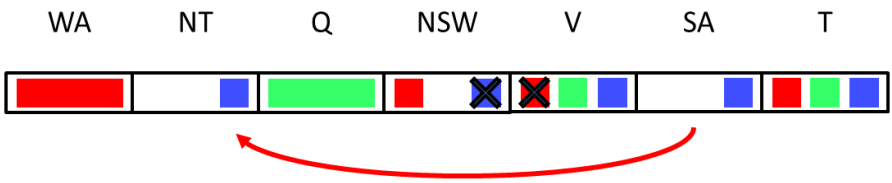


Arc consistency : example

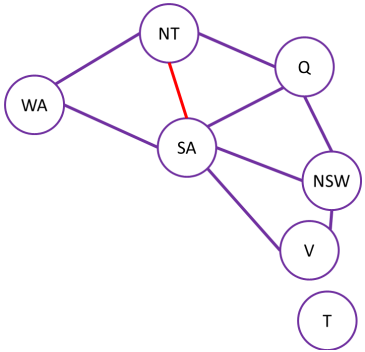


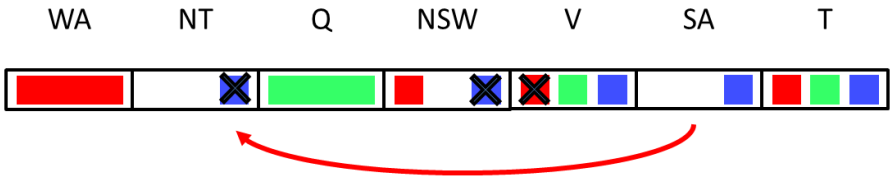
Arc consistency : example



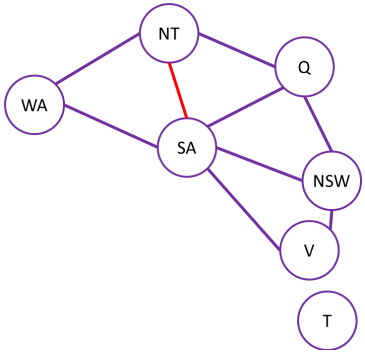


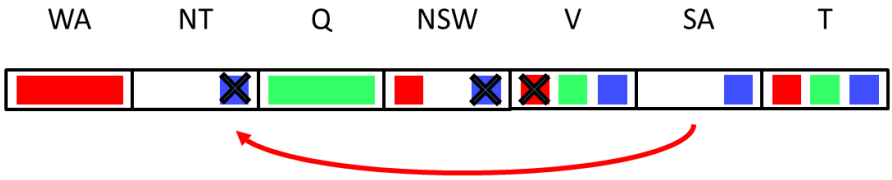
Arc consistency : example





Arc consistency : example





Arc consistency

Properties

- Arc consistency is faster than forward checking.
- Can be executed before or after each affectation.

Arc consistency : AC3 algorithm

Function *AC-3 (CSP)* `/* Returns a CSP with, eventually, reduced domains */`

Data: a binary CSP with variables X_i
begin
  queue $\longleftarrow$ all edges of CSP
  while *queue* $\neq \emptyset$ **do**
    $(X_i, X_j) \longleftarrow$ DEQUEUE(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      foreach $X_k \in$ NEIGHBORS(X_i) **do** *queue* $\longleftarrow$ Enqueue (X_k, X_i) in *queue*
    end
  end
end

Function REMOVE-INCONSISTENT-VALUES (X_i, X_j) `/* True if domain has changed */`

begin
  removed $\longleftarrow$ false
  foreach $x \in$ *domain*(X_i) **do**
    if *no value* $y \in$ *domain*(X_j) *leads to* (x, y) *satisfying constraint* $X_i \longleftrightarrow X_j$ **then**
      Remove x from domain(X_i)
      removed $\longleftarrow$ true
    end
  end
  return *removed*
end

Arc consistency : AC3 algorithm

Properties

- n = number of variables.
- d = number of values per variable.
- s = maximum number of constraints on a pair

- Each variable is inserted in the queue at most d times
- Time complexity of REMOVE-INCONSISTENT-VALUES is $o(d^2)$
- Time complexity of constraint propagation is $o(nsd^3)$

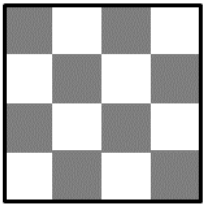
CSP solving

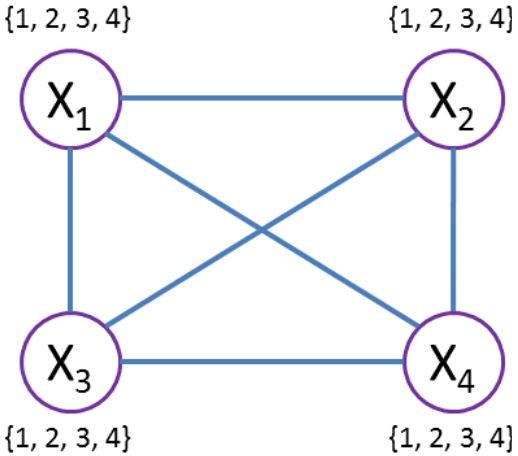
To solve a CSP :

Mix constraint propagation and backtracking, i.e. :

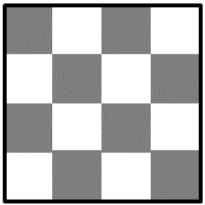
- AC 3 : to reduce domains
- Backtracking : to select a value
- Forward checking : to propagate constraints and detect contradictions

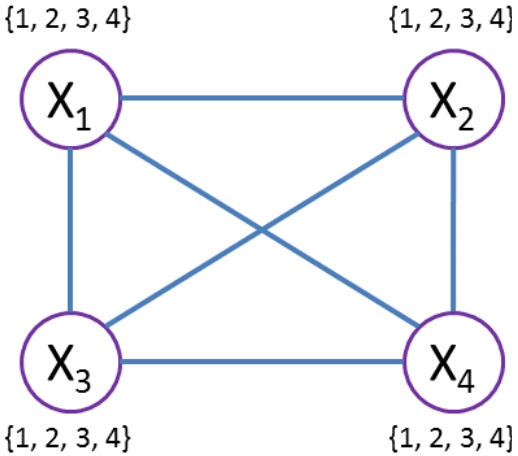
4-queens problem



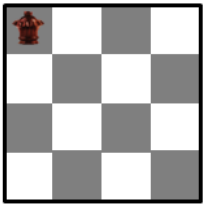


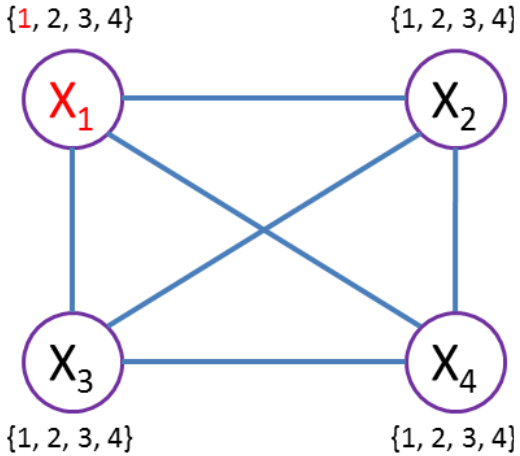
4-queens problem



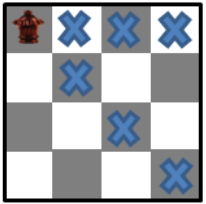


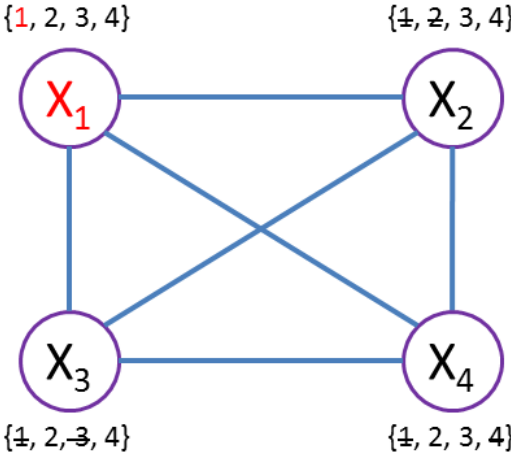
4-queens problem



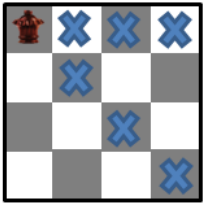


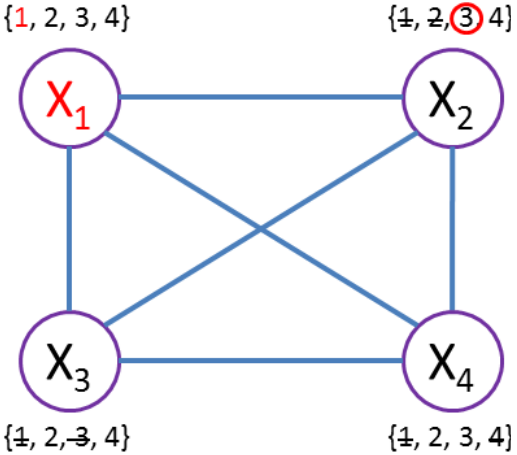
4-queens problem



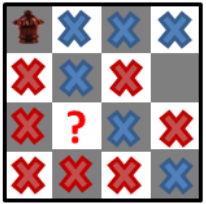


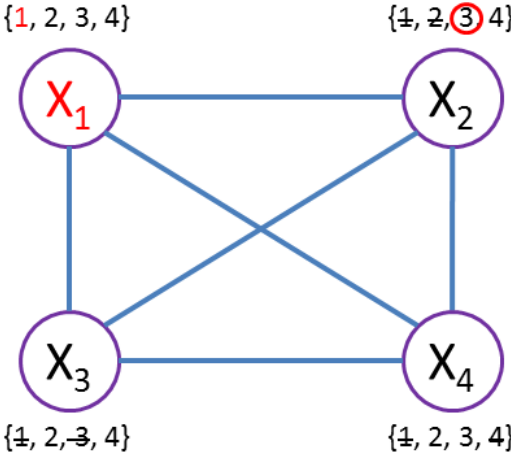
4-queens problem



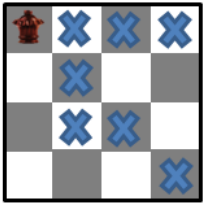


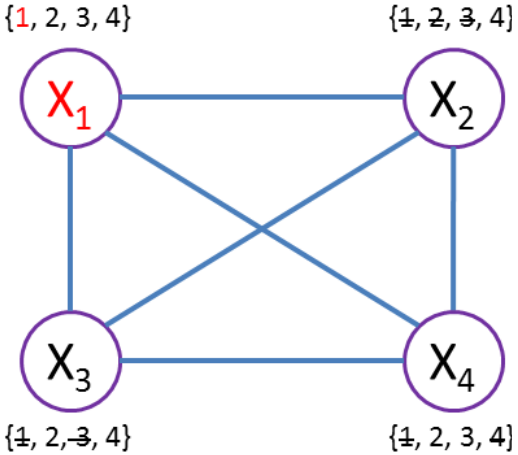
4-queens problem



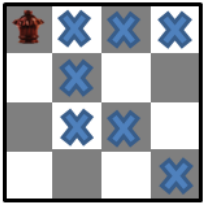


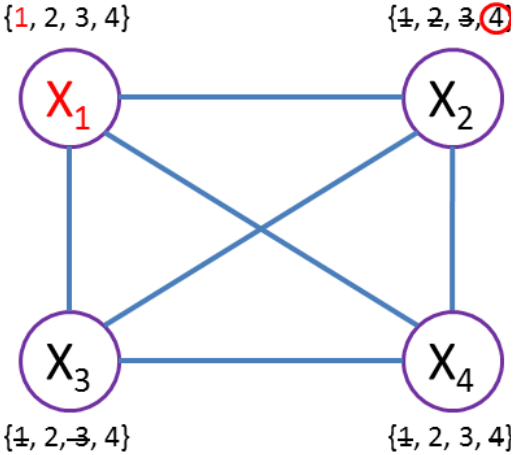
4-queens problem



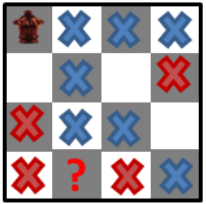


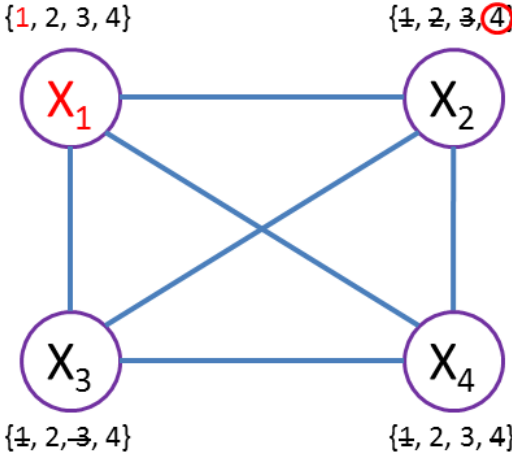
4-queens problem



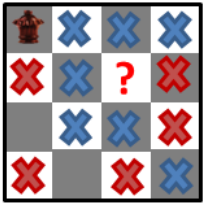


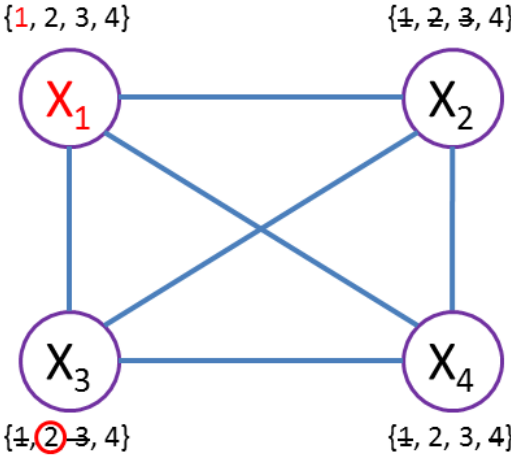
4-queens problem



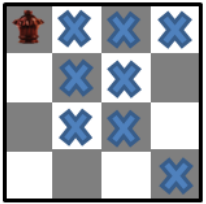


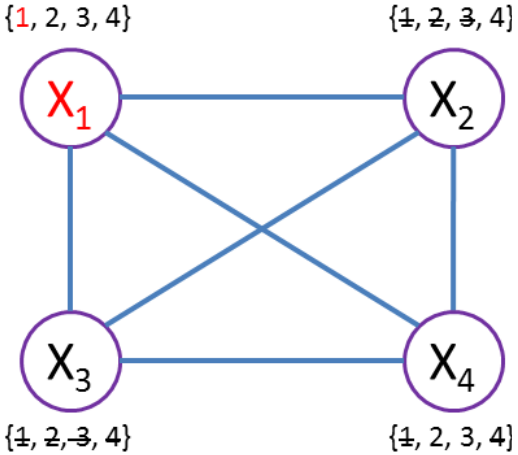
4-queens problem



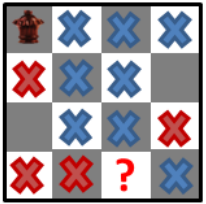


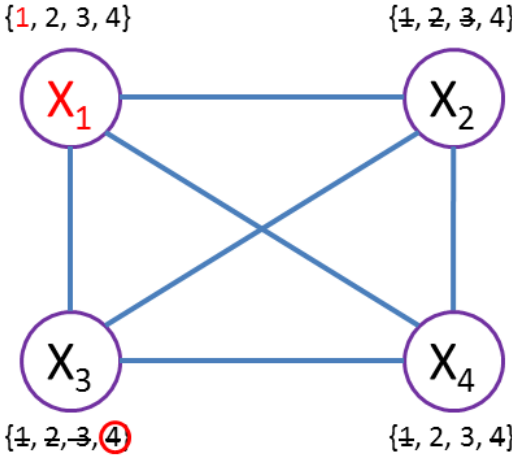
4-queens problem



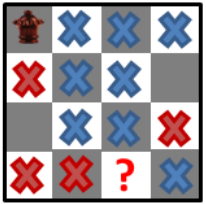


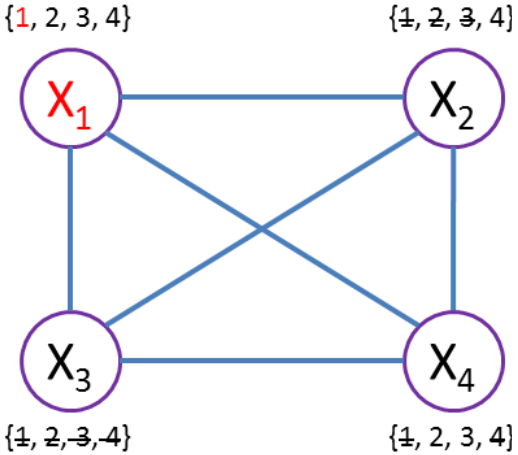
4-queens problem



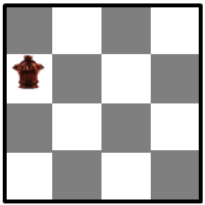


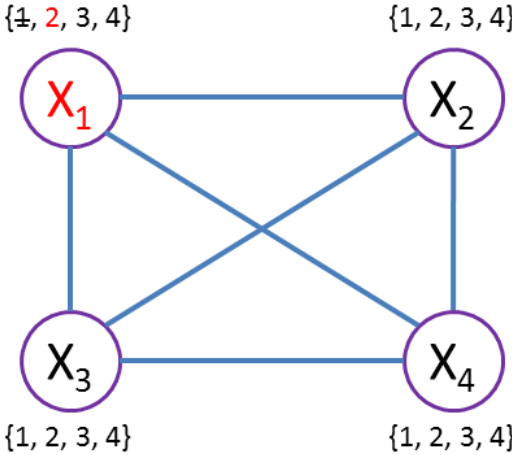
4-queens problem



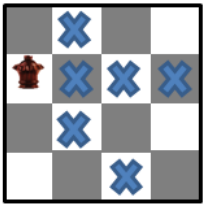


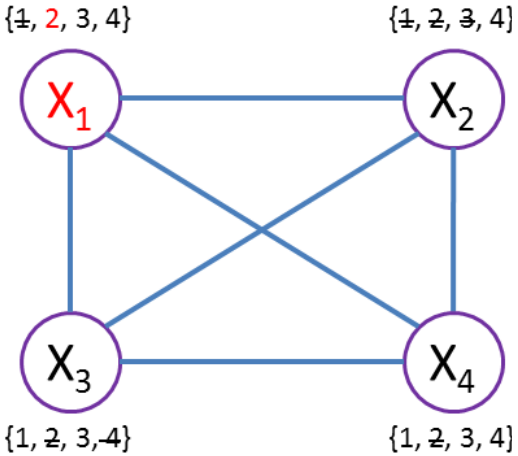
4-queens problem



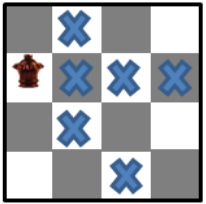


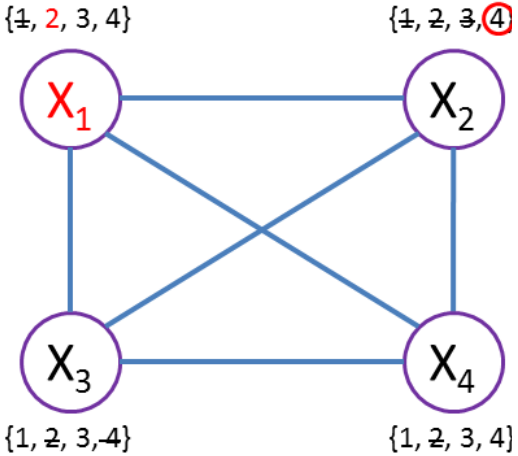
4-queens problem



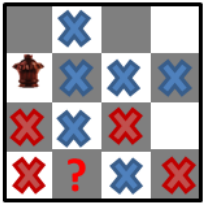


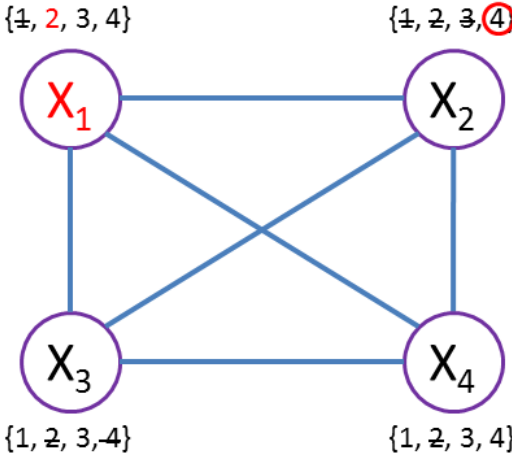
4-queens problem



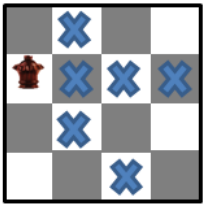


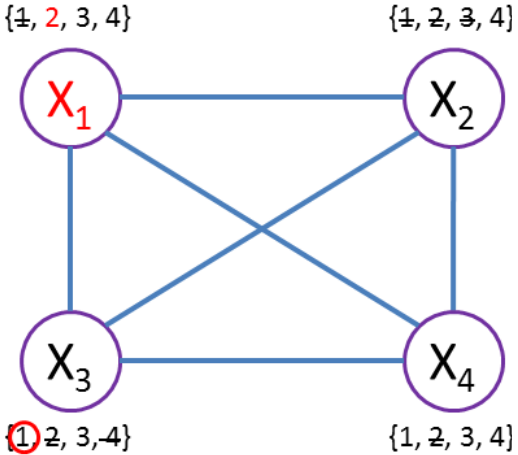
4-queens problem



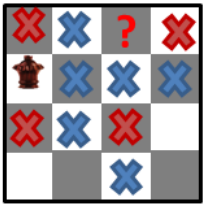


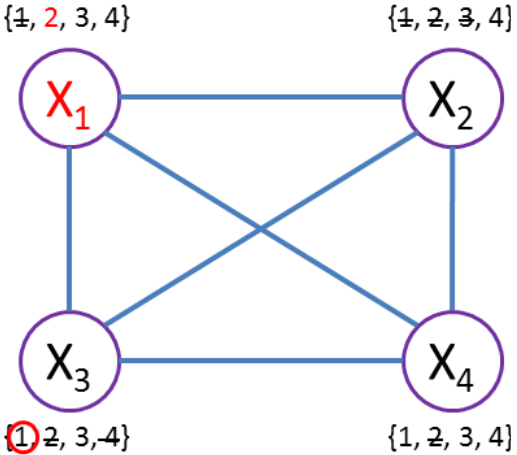
4-queens problem



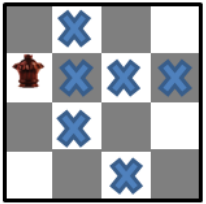


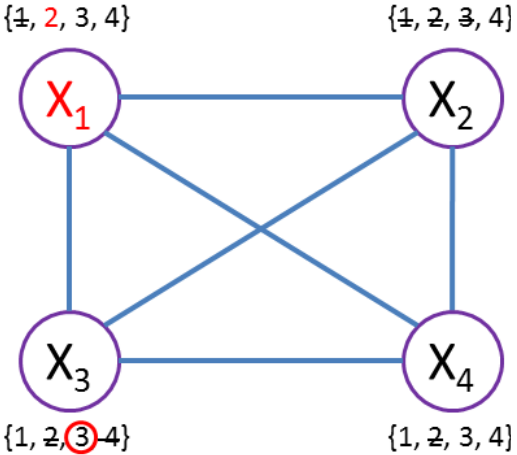
4-queens problem



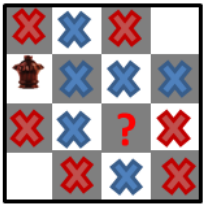


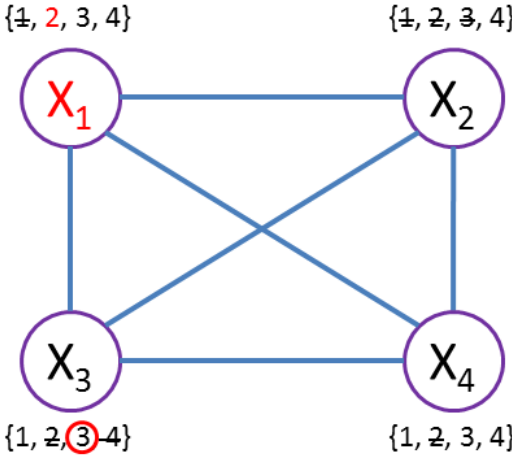
4-queens problem



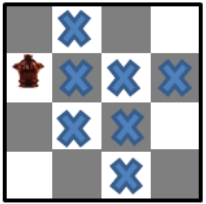


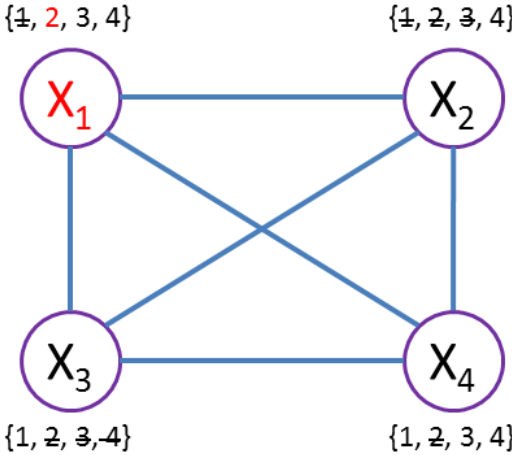
4-queens problem



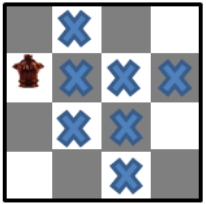


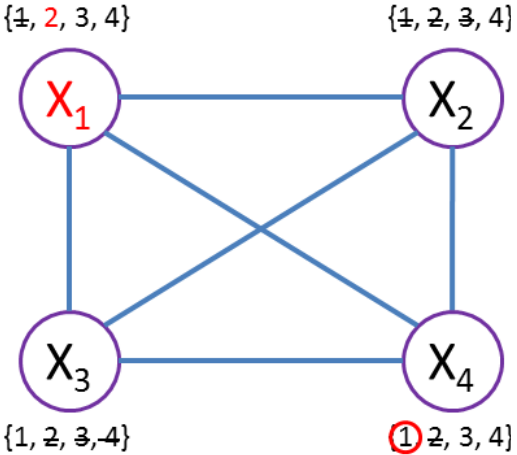
4-queens problem



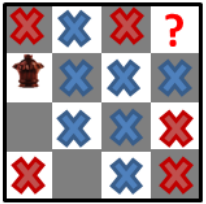


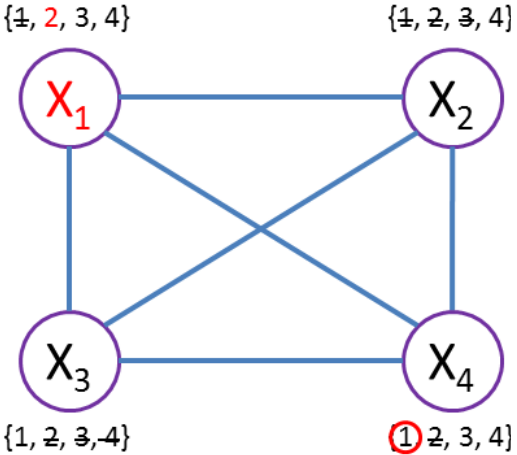
4-queens problem



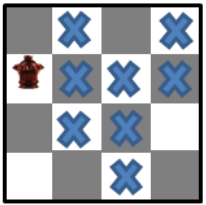


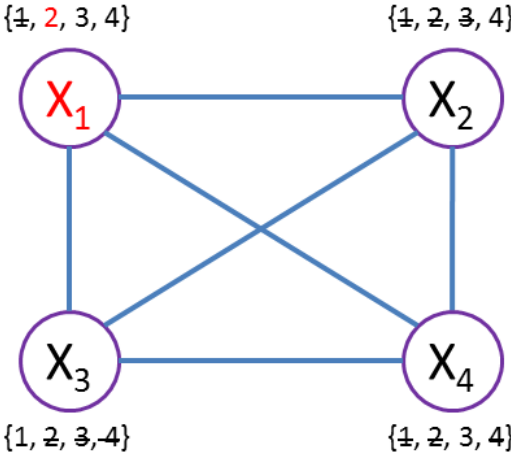
4-queens problem



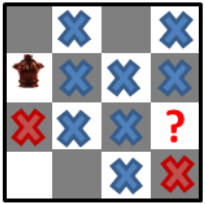


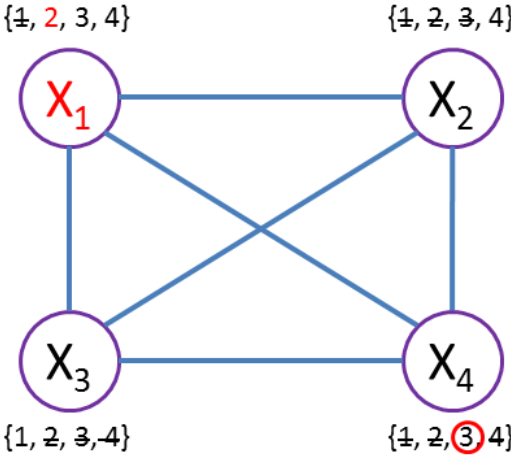
4-queens problem



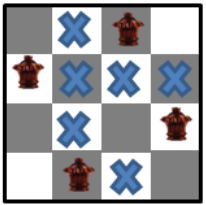


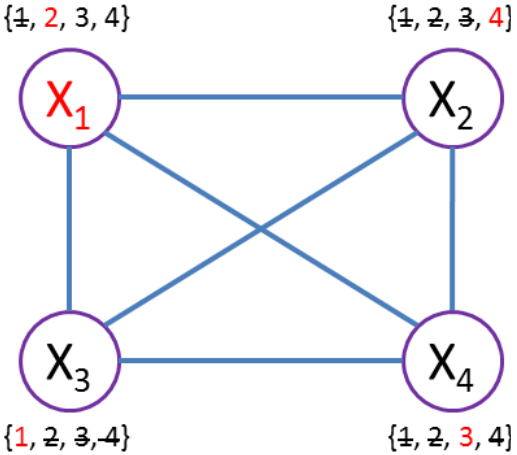
4-queens problem





4-queens problem





Outline

Conclusion

- Different algorithms to solve different problems
- All based on the same principles :
  - taking advantage of the computation power of machines to browse the different candidates
  - replace human intuition by tricks like heuristics, pruning or filtering
- With these algorithms, machine intelligence relies on their capacity to handle the combinatory.