1.1

AI Game Programming

Jean-Charles Régin

Master Informatique 1st Year

Remerciements

- Wikipédia
- Patrick Lester (images A*)
- Fabien Torre (Minimax and Alpha-Béta)

Goal

- I would like you to understand that computers can be used for solving complex problems
  - Web and video are not the only one usage
- Clever use of the computation power
- It is quite important to program without any bug (or with only a limited number of bugs)

- Some knowledge in order to be ready to begin to try to understand alphaGo ☺

Plan

- Introduction
- Path based problems (state graph)
- Two players problems

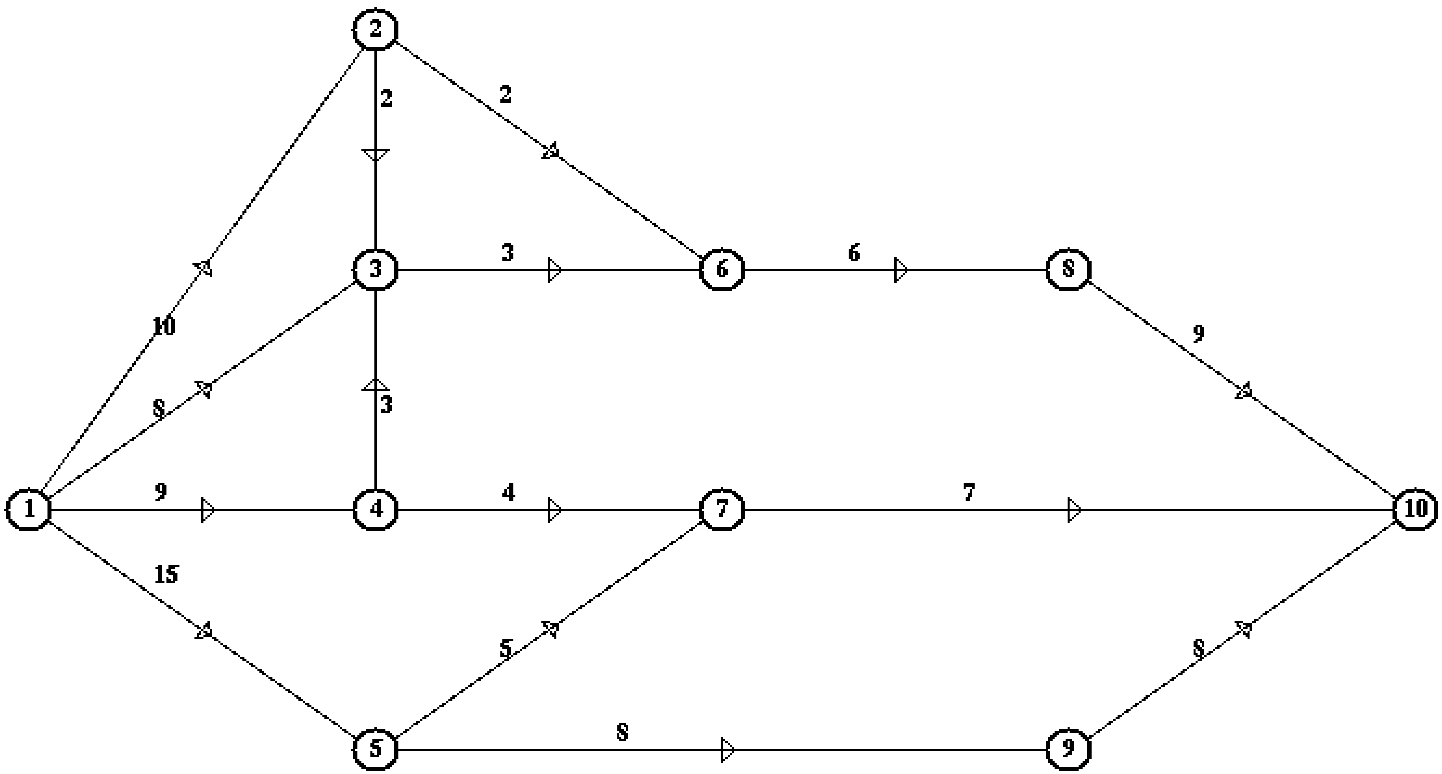
1.5 Introduction

Graph : definitions

- A directed Graph G=(X,U) is defined by
  - A set of **vertices** or **nodes** X
  - A set of nodes couple U, where a couple is called an arc
- If u=(i,j) is an arc of G then i is the initial extremity of u and j the terminal extremity of u.
- The arcs have a direction. The arc u=(i,j) is from i to j.
- Arcs may have a cost, a capacity, a length, a weight etc…

Graph

1.7



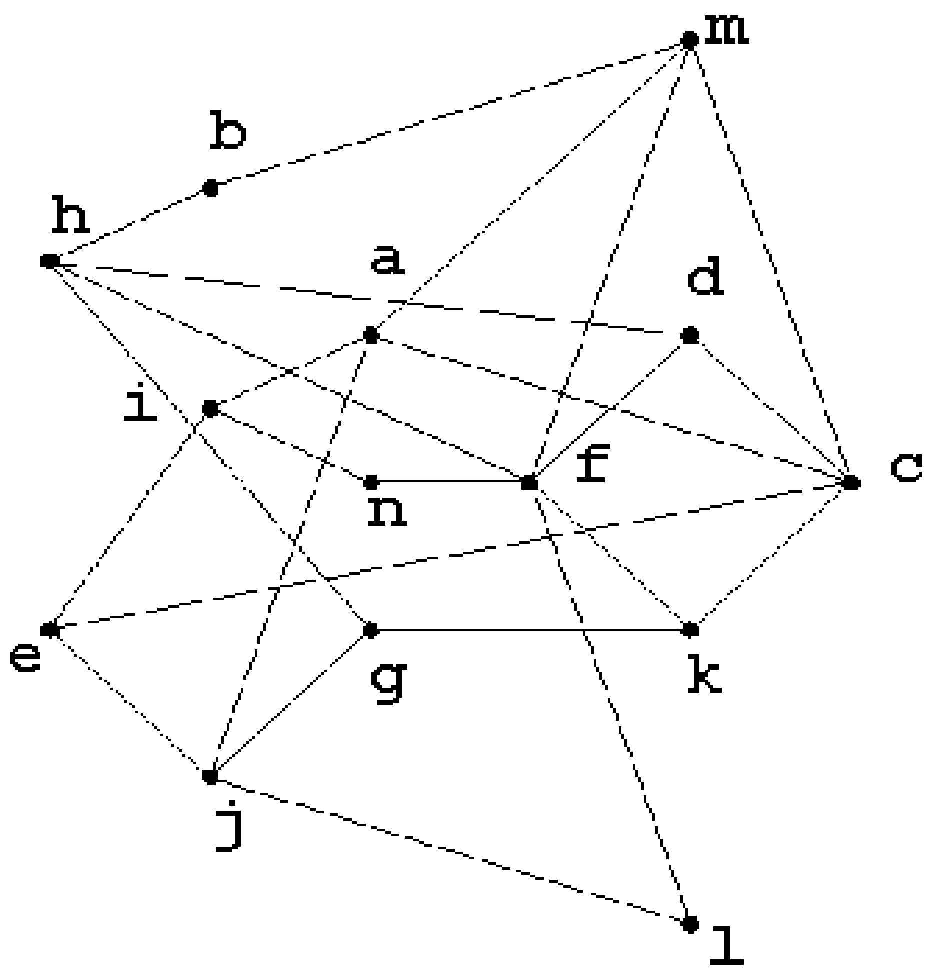
Graph

- We will denote by $\omega(i)$: the set of arcs having i as an extremity
- We will denote by $\omega^+(i)$: the set of arcs having i as an initial extremity = set of arcs outgoing i.
- We will denote by $\omega^-(i)$: the set of arcs having i as a terminal extremity = set of arcs incoming i
- N(i) : set of neighbors of i : set of nodes j such that it exists an arc from i to j

Undirected Graph

- An undirected Graph G=(X,E) is defined by
  - A set of **vertices** or **nodes** X
  - A set of pairs of nodes called edges
- The edges are not oriented

Undirected Graph



Graph : definitions

- Two nodes are neighbor if there are linked by an arc or an edge
- Incoming degree of i : number of incoming arcs of i
- Outgoing degree of i : number of outgoing arcs of i

Graph : definitions

- Directed path of length q : sequence de q arcs $\{u_1,u_2,\ldots,u_q\}$ such that
  - $u_1=(i_0,i_1)$
  - $u_2 = (i_1,i_2)$
  - $u_q=(i_{q-1},i_q)$
- Directed path : all the arcs are oriented in the same direction
- Directed cycle : directed path having the same extremities

Computers are very fast

- Chess game :
  - A player has about 10 possible moves in average,
  - I have 10 moves, my opponent has 10 moves for each of my 10 moves.
  - If I play twice then 10(me)x10(him)x10(me) combinations to evaluate.
  - If I play 3 times then 10(me)x10(him)x10(me)x10(him)x10(me) combinations.
- If I play k times then 10^{2k-1} combinations

Chess game

- If I play k times then 10^{2k-1} combinations
- If I play 5 times then 10 000 000 = 10 millions combinations
- 20 years ago (since 1993)
  - The best program was playing like an Intl Master
  - The frequency of the computer was 100Mhz
- Today
  - The best program is the best player in the world. It costs almost nothing ($50 or $100)
  - Computer runs at 3Ghz

Chess game

- Increasing of the computer power
  - frequency : 3Ghz vs 100 Mhz
  - multicores vs mono core
  - Best generated code (compilers)
  - Best architecture of processors (CPU manufacturer)
  - Dhrystone benchmark (from 1984) : calculations made only for integers
    - Pentium (100Mhz) Dhry2 opt = 122,
    - Core i7 930 (3Ghz) Dhry 2 opt = 8684
    - Ratio : 8684/122 = 71,18. (30 for frequency)

Chess game

- 70 time faster in 20 years (30 for the frequency, 2,5 for the architecture)
  - 1 move = 10 (me) * 10 (him) = 100 combinations
  - Roughtly in 20 ans we save 1 move for a monocore
- 2 move (2 for me and 2 for) = 100*100=10000.
  - It requires 100 cores !
- **Thus we will not solve a problem by expecting only a progress of the computer**

1.17 Path problems

Plan

- State Graph
- Path in a graph (DFS, BFS)
- Shortest path between two nodes
  - Dijkstra’s algorithm
- Very large Graph: algorithm A*
- Applications

State Graph

- A lot of problems can be solved by defining a state graph
- A node = a possible state
- An arc = a change between two states
- Usually there are 2 particular states
  - Initial state
  - Final state
- The solution of the problem becomes simple: this a path between an initial state to a final state
- The difficulty is to define the state graph

State Graph

- Sailor Cat needs to bring a wolf, a goat, and a cabbage across the river.
  - The boat is tiny and can only carry one passenger at a time.
  - If he leaves the wolf and the goat alone together, the wolf will eat the goat.
  - If he leaves the goat and the cabbage alone together, the goat will eat the cabbage.
- How can he bring all three safely across the river?

State graph

- We define the states:
- We have 3 animals, 2 sides of the river and the location of the boat. Wolf (W), Goat (G), Cabbage (C)
  - (L=(B,G,W) R=(C)) sens ?
- Some states are forbidden
  - (L=(G,W) R=(B,C)) the wolf will eat the goat
  - (L=(B,G,W) R=(C)) is fine because the sailor is witht he wolf and the goat

State graph

- We define all the states and represent all the allowed states. We kink them when it is possible to go from one to another one
- (L=(B,G,W) R=(C)) can be linked to
  - (L=(G,W) R=(B,C)) the boat changed of side
  - (L=(W) R=(B,G,C)) the goat crossed the river
  - (L=(G) R=(B,W,C)) the wolf crossed the river

State Graph

- 2 sides et 4 objects. For each object we have 2 possibilities, there are $2^4=16$ possible states:
  - (L=(B,W,G,C) D=()) ok INITIAL STATE
  - (G=(B,G,W) D=(C)) ok
  - (G=(B,G,C) D=(W)) ok
  - (G=(B,W,C) D=(G)) ok
  - (G=(G,W,C) D=(B)) forbidden
  - (G=(B,G) D=(W,C)) ok
  - (G=(B,W) D=(G,C)) forbidden
  - G=(B,C) D=(G,W)) forbidden
  - (G=(G,W) D=(B,C)) forbidden
  - (G=(G,C) D=(B,W)) forbidden
  - (G=(W,C) D=(B,G)) ok
  - (G=(B) D=(G,W,C)) forbidden
  - (G=(G) D=(B,W,C)) ok
  - (G=(W) D=(B,G,C)) ok
  - (G=(C) D=(B,G,W)) ok
  - (G=() D=(B,G,W,C)) ok FINAL STATE

State Graph

- We draw the state graph!

State Graph

- A group of soldiers arrive at a river. The bridge has been destroyed and the river cannot be crossed by swimming. The captain thinks and sees a small boat that is handled by two boys. He took the boat but he discovers that the boat
  - Is fine for a maximum of one soldier OR two boys
  - Is too small for one soldier and one boy.
- The captain finds a solution. Which one?

State Graph

- We define the state graph
- However, the number of soldiers is not defined, so how can we define this graph?

State Graph

- Try to work with only one soldier. That is to find a way for transporting the soldier from the left side to the right side and to have the boat and the two boys on the left side
- So we start with 2 boys and 1 soldier and we try to help the soldier to croos the river and to have the two boys back

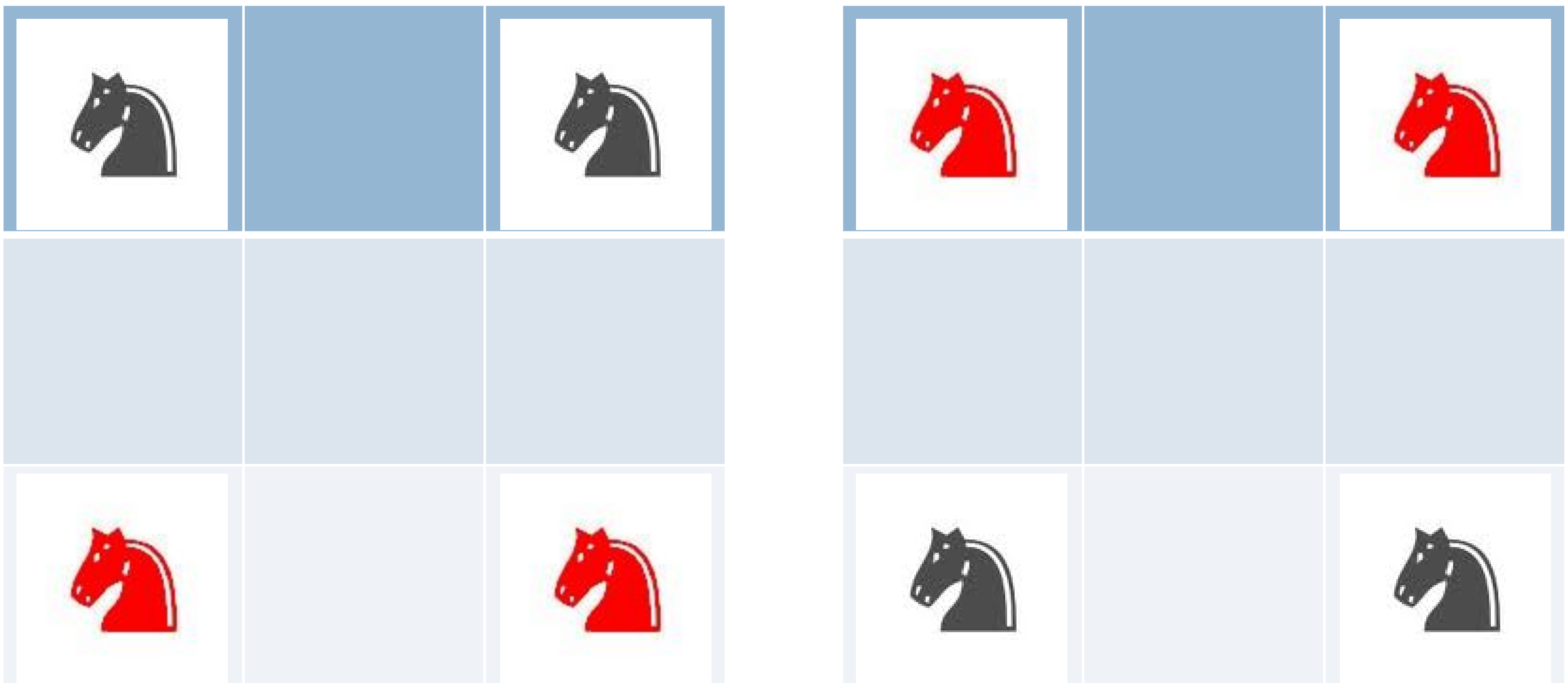
State Graph

- We define the states
- (L=(B,S,E1,E2) R=()) initial state
- (L=(B,E1,E2) R(S)) final state

- We enumerate all the states, we draw the grph and we search for a path from the initial state to the final state.
- Then, we repeat this solution for all the soldiers!

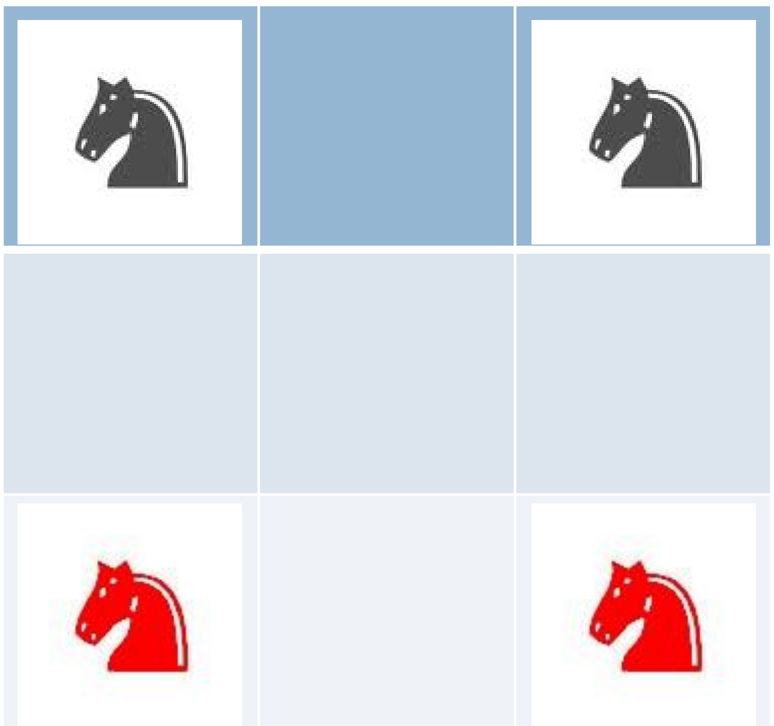
State Graph

- Problem of the exchange of two knights (chess game)
- We want to go from the left situation to the right



State Graph

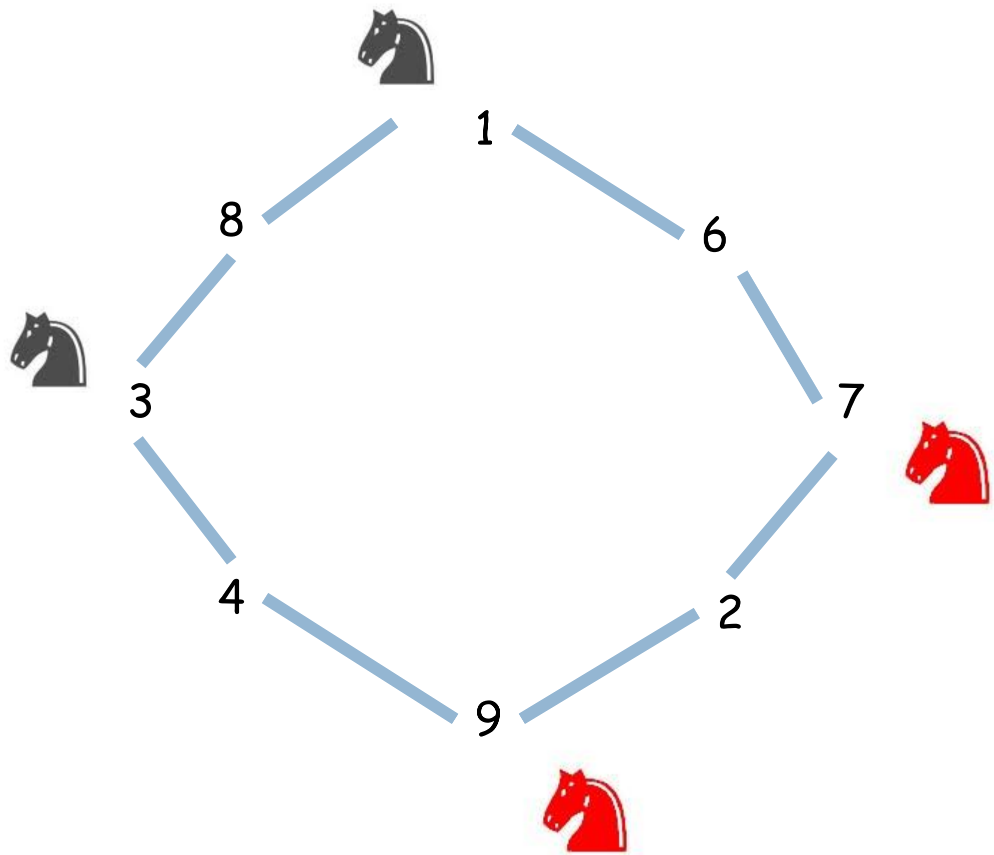
- Problem of the exchange of two knights (chess game)
- We want to go from the left situation to the right



1	2	3
4	5	6
7	8	9

State Graph

- We draw the possible move from square to square



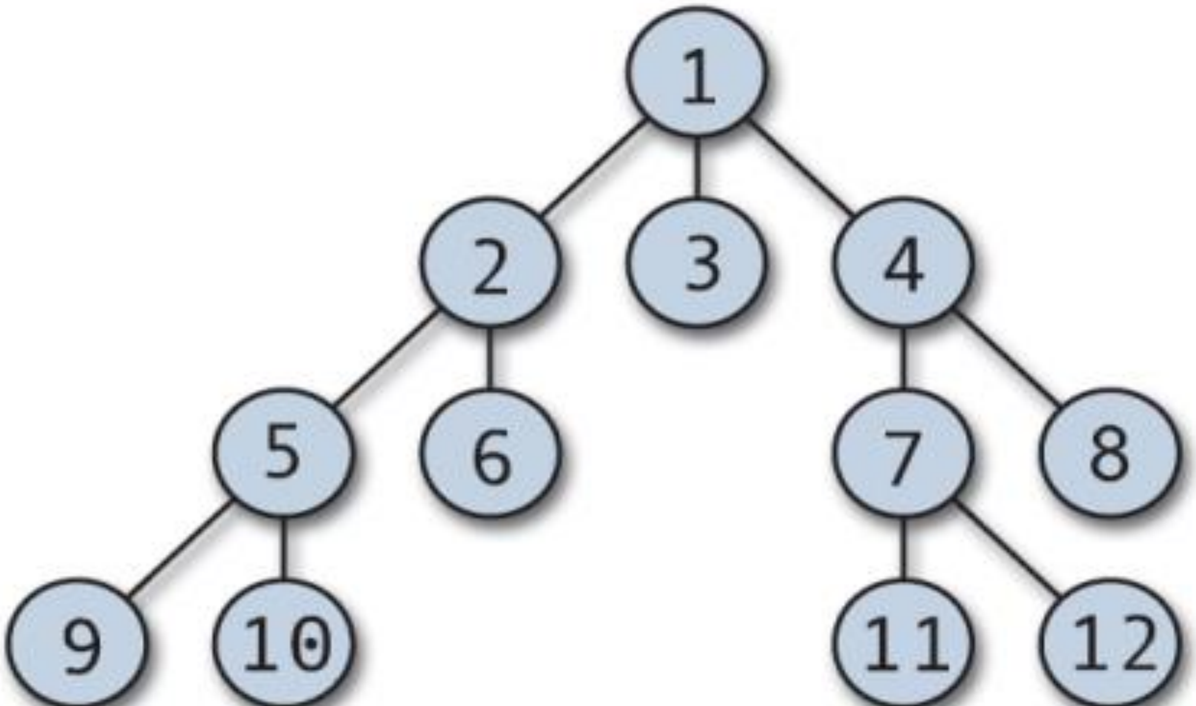
Plan

- State Graph
- **Path in a graph (DFS, BFS)**
- Shortest path between two nodes
  - Dijkstra's algorithm
- Very large Graph: algorithm A*
- Applications

Tree

- A tree is an undirected connected graph without cycle

Tree

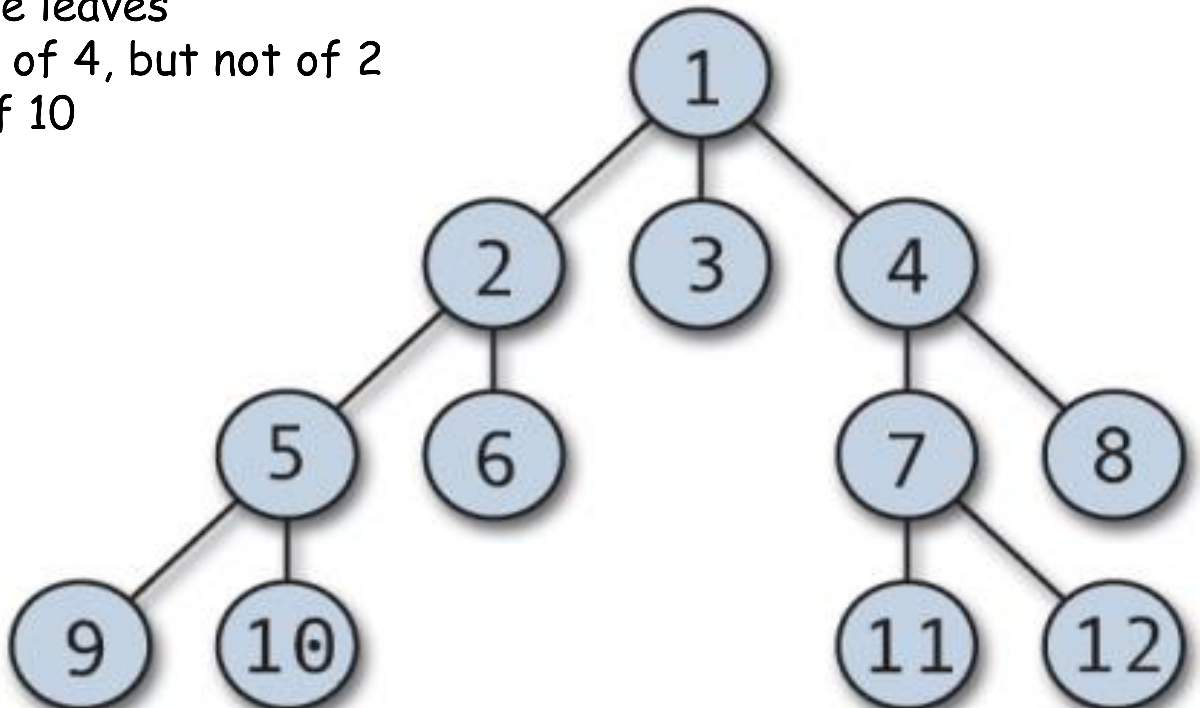


Tree

- The **root** r of the tree is the unique node that does not have any parent
- Each node which is not the root has
  - a unique parent, denoted by parent(x)
  - 0 or several children. child(x) represents teh set of children of x
- If x and y are nodes such that x is on the path from r to y then
  - x is an ancestor of y
  - y is a descendant of x
- A node without child is a leaf

Tree

1 is the root
9,10,6,3,11,12,8 are leaves
11 is a descendant of 4, but not of 2
2 is an ancestor of 10

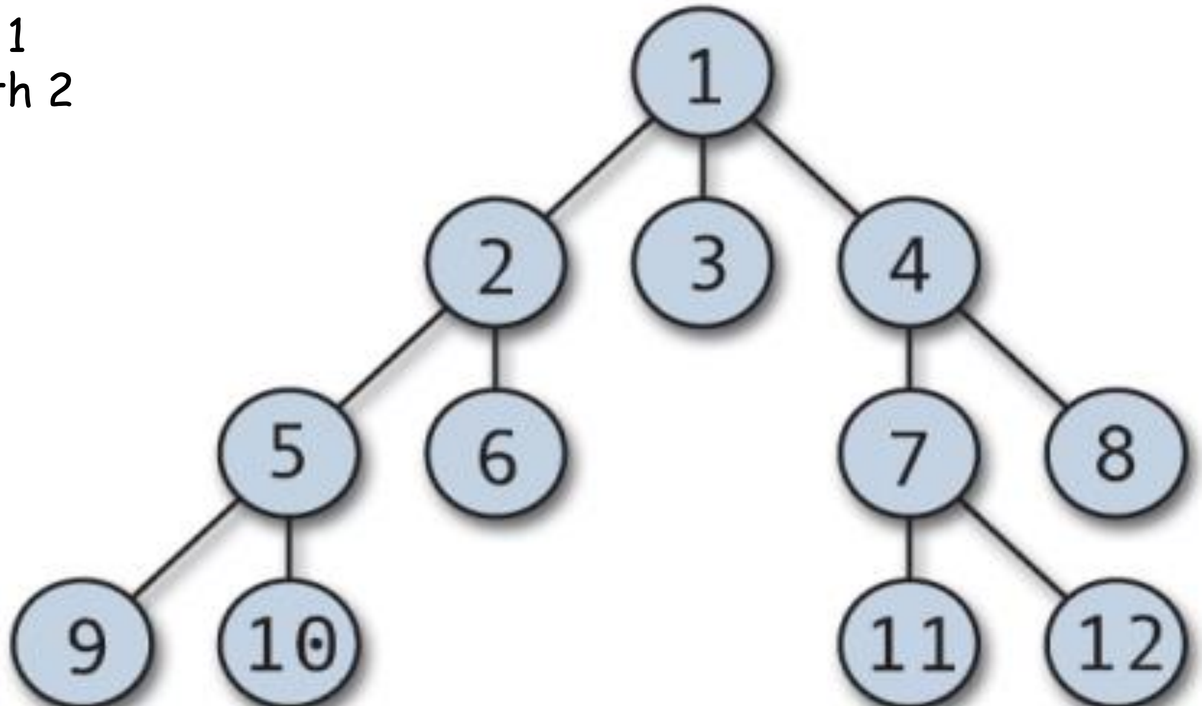


Tree

- The **depth** of a node is revursively defined by
  - depth(v) = 0 if v is the root
  - depth(v) = depth(parent(v)) + 1

Tree

1 is the root
2,3,4 are at depth 1
5,6,7,8 are at depth 2



Tree traversal

- We traverse the set of nodes of the tree

- Breadth first search
- Depth first search
  - Prefixed
  - Infixed
  - Postfixed

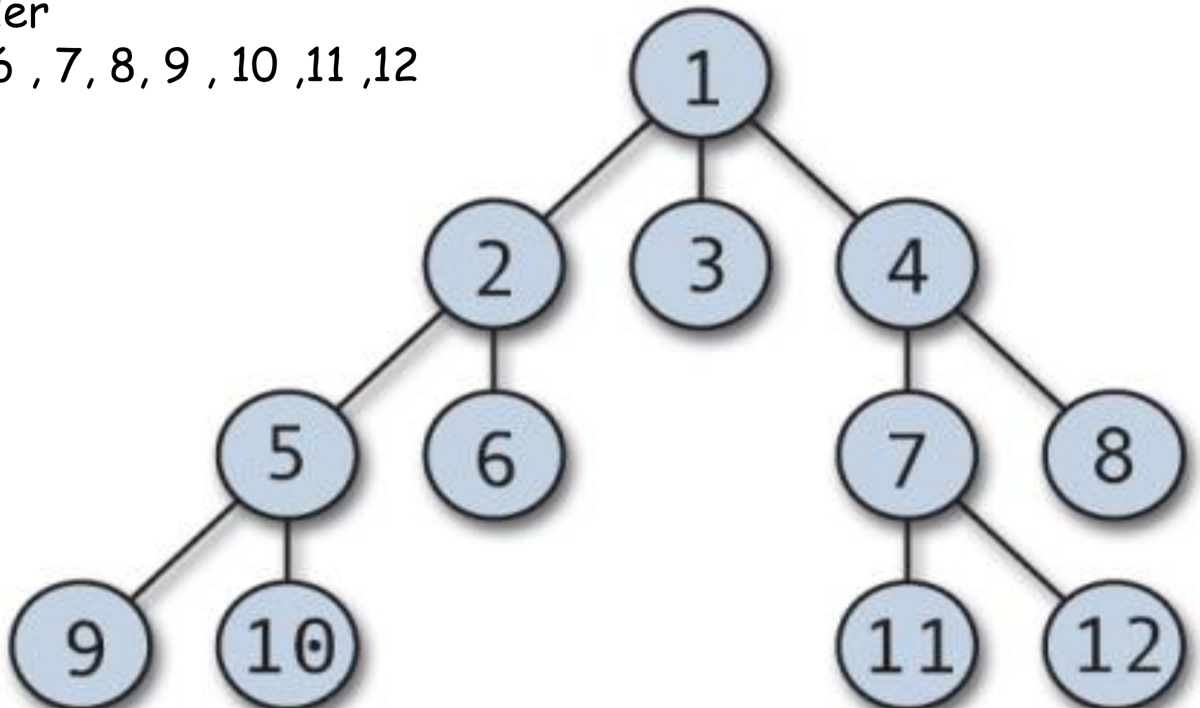
Tree: Breadth First Search (BFS)

- We visit the root and we repeat the following process until each node is visited:
  visit the children of the least recently visited node

- We visit all the nodes at depth 1, then all the nodes at depth 2 ,…

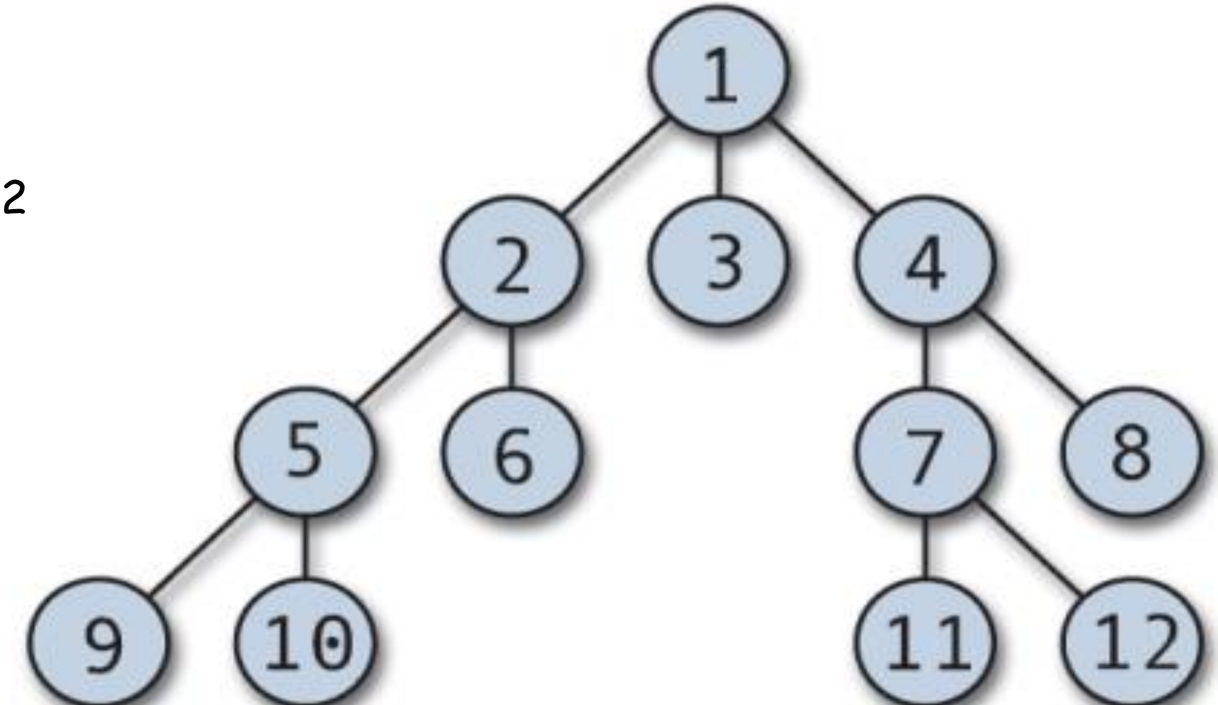
Tree: BFS

BFS visit order
1, 2 ,3 ,4 ,5, 6 , 7, 8, 9 , 10 ,11 ,12



Tree: BFS with passes

BFS with passes: visit order
Pass 1 : 1
Pass 2 : 2,3,4
Pass 3 : 5,6,7,8
Pass 4 : 9,10,11,12



Tree: Depth First Search (DFS)

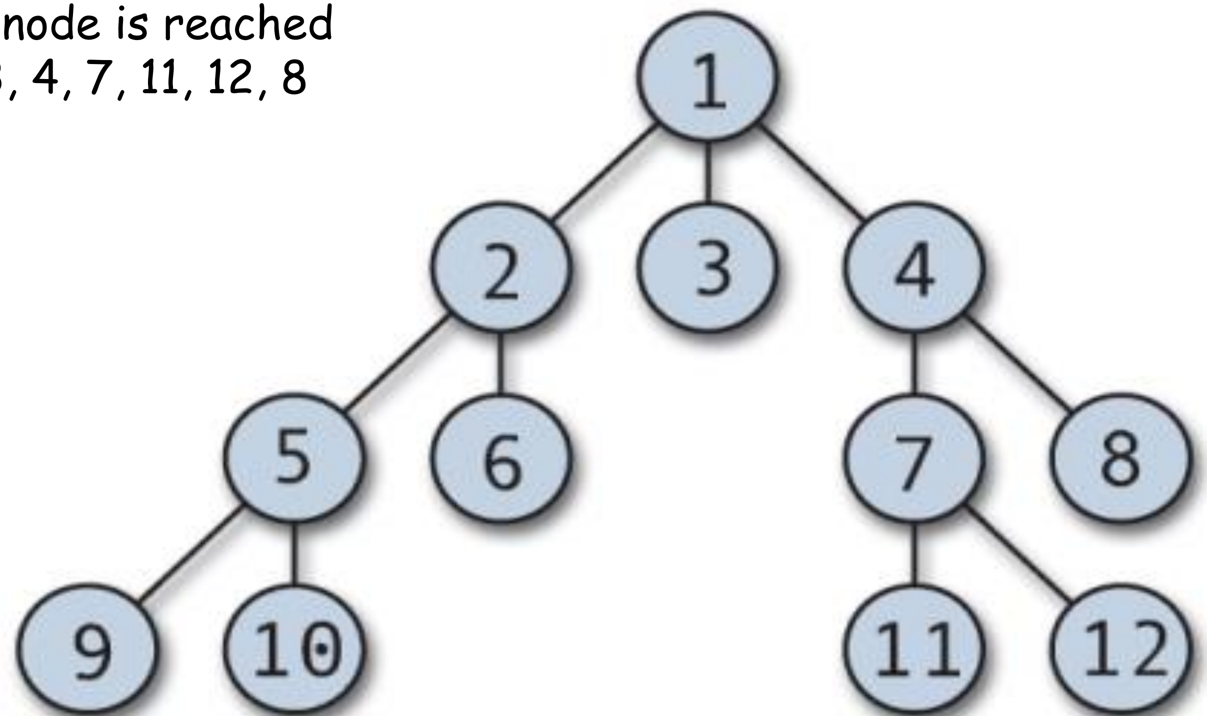
- Recursive definition
- visit(node x)
  previsit(x)
  for each child y of x
      visit(y)
  postvisit(x)
- First call: visit(root(T))

Tree: Depth First Search (DFS)

- Prefixed or postfixed order depends on functions previsit and postvisit
- If previsit(x) marks i and add it to the order, then we have a prefixed order
- If postvisit(x) marks i and add it to the order, then we have a postfixed order

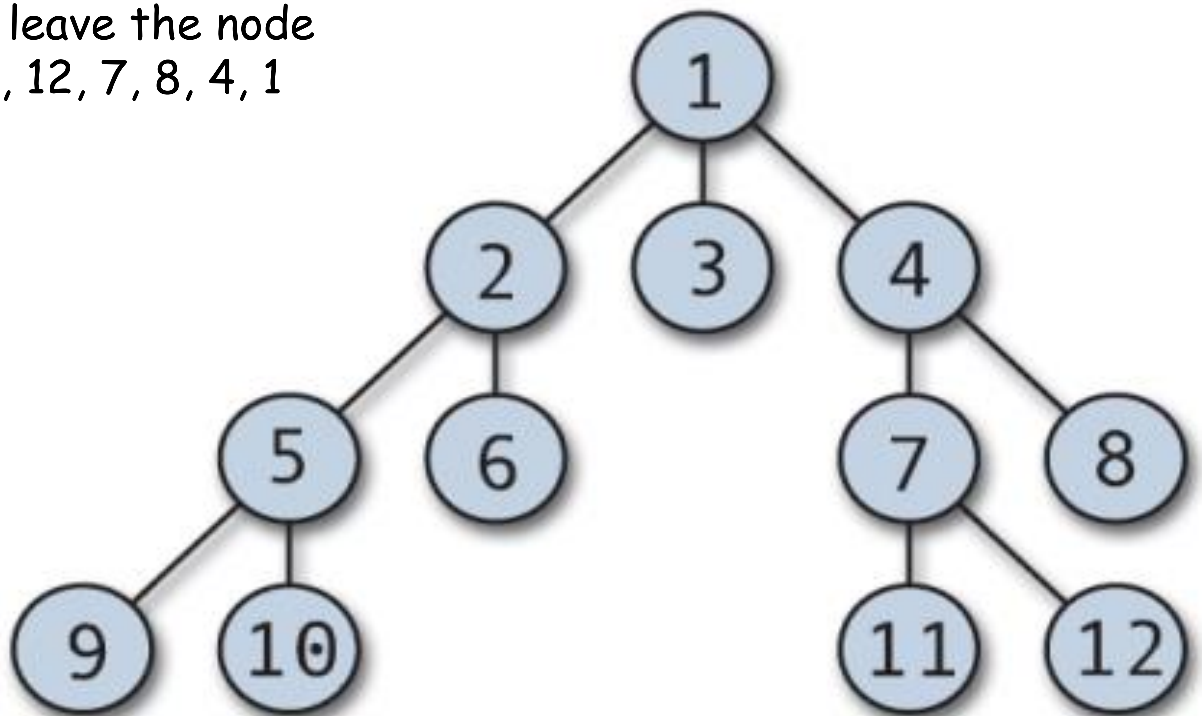
Tree

DFS: prefixed order
We mark when a node is reached
1 ,2 ,5, 9, 10, 6, 3, 4, 7, 11, 12, 8



Tree

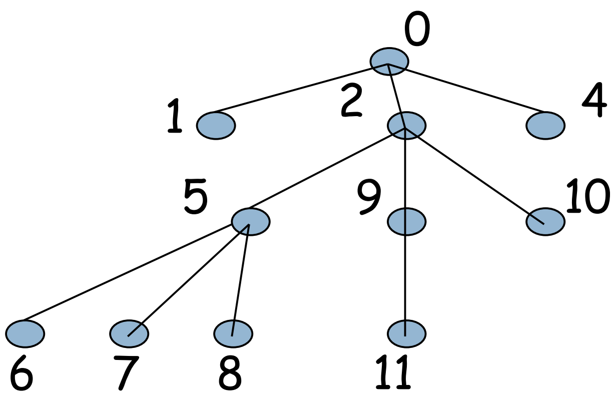
DFS: postfixed order
We mark when we leave the node
9, 10, 5, 6, 2, 3, 11, 12, 7, 8, 4, 1



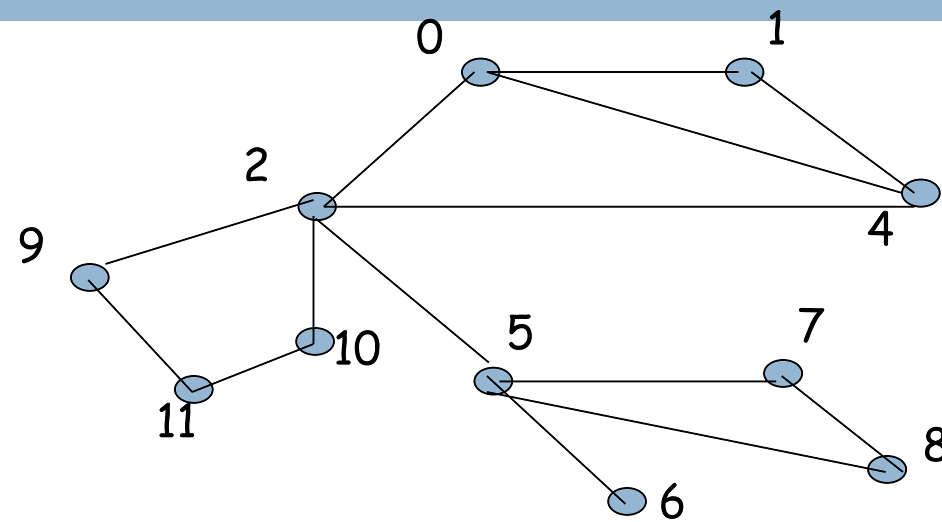
Paths in graph

- The algorithms for the trees can be applied for graphs
- We just need to be careful
  - **We have to avoid visiting twice the same node**
  - Why?

Illustration of BFS



BFS Tree



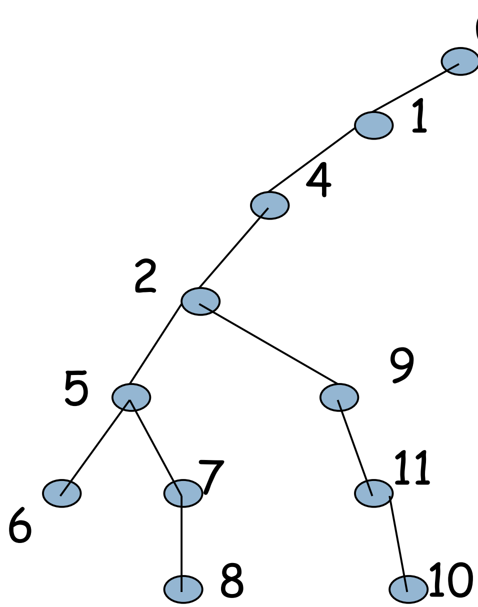
Graph G

Graph: BFS

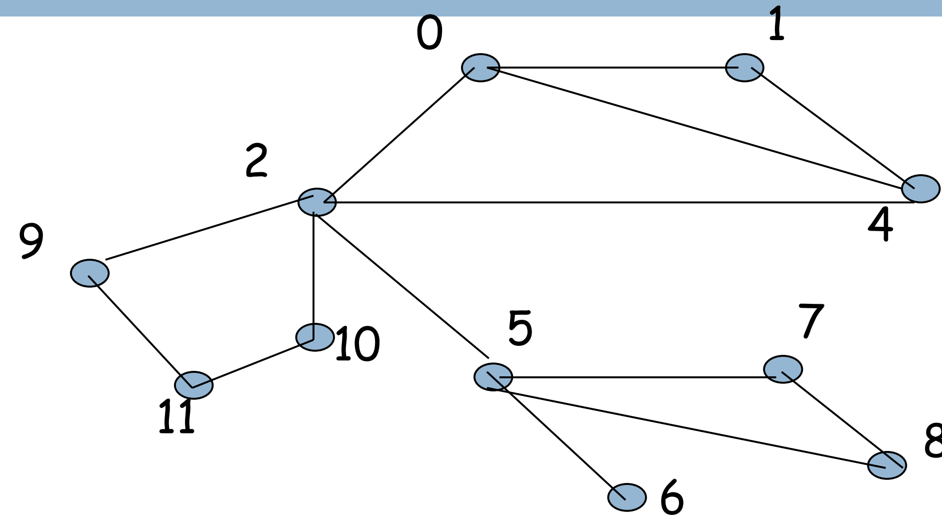
- Bfs(G,s) : array

```
r ← s
pour tous les sommets x marque[x] ← faux
créer un file F; enfiler(F,r); marque[r] ← vrai
i ← 0
tant que (F n'est pas vide)
        x ← premier(F); defiler(F)
        array[i] ← x
        i++
        pour chaque voisin y de x
                si marque[y] est faux
                alors   marque[y] ← vrai
                        enfiler(F,y)
        fin pour
fin tant que
```

Illustration of DFS



DFS Tree



Graph G

Graph: DFS

- Dfs(G,s) : array

```
r ← s
pour tous les sommets x marque[x] ← faux
créer un pile P; push(P,r); marque[r] ← vrai
i ← 0
tant que (P n'est pas vide)
        x ← top(P); pop(P);
        array[i] ← x
        i++
        pour chaque voisin y de x
                si marque[y] est faux
                alors   marque[y] ← vrai
                        push(P,y)
        fin pour
fin tant que
```

Plan

- State Graph
- Path in a graph (DFS, BFS)
- **Shortest path between two nodes**
  - Dijkstra's algorithm
- Very large Graph: algorithm A*
- Applications

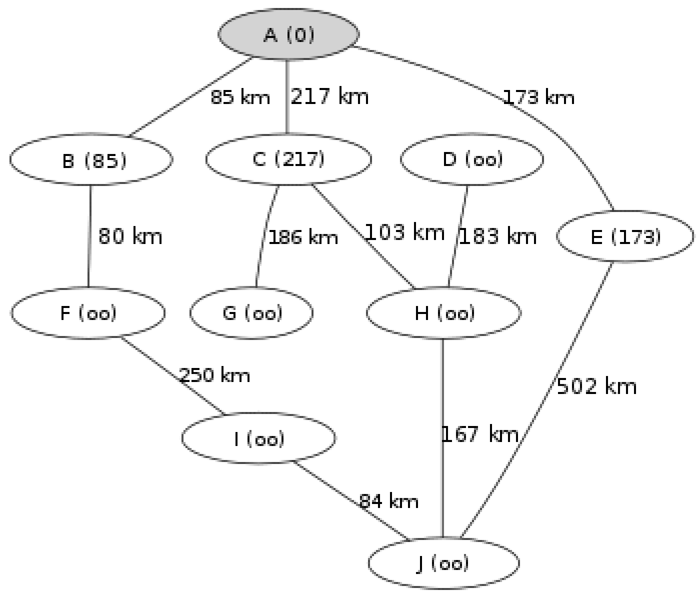
Shortest path between two nodes

- We introduce length on arcs
- This length may be named:
  - Distance
  - Weight
  - Cost
  - …
- The important fact is that **the length of a path is equal to the sum of the length of its arcs**

Shortest path between two nodes

- We assume that all costs are non negative
- This is not mandatory : we can compute the shortest path with negative costs, but we need different algorithms

Path from A to J



Shortest path between two nodes

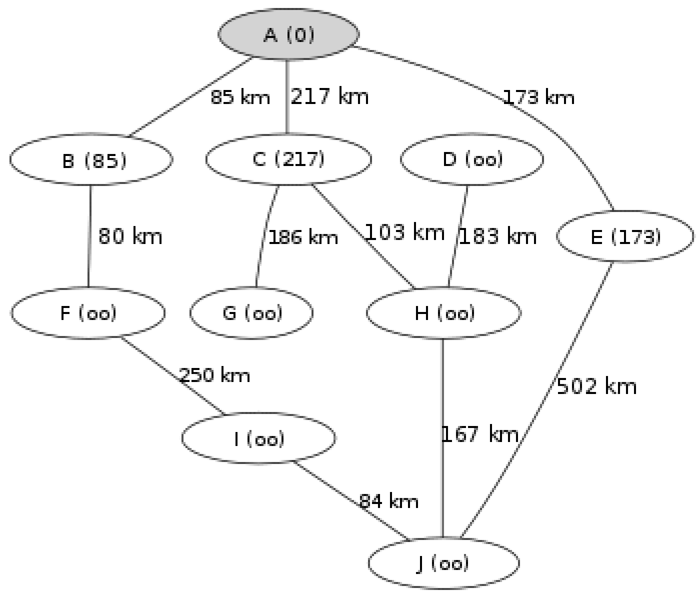
- We cannot enumerate all the paths
- A greedy algorithm exists!
- **This is NOT a DFS based algorithm**

Dijkstra's algorithm

- We maintain the distance between any node and the source
- We have 3 kinds of nodes
  - **Open** (these are candidate for the next step)
    - We can reach them by traversing nodes that have been selected
  - **Closed** : these are nodes that have already been chosen
  - **Undefined** : currently we cannot reach them
- At each step: **we select the open node with the smallest distance to the source**
  - We consider the nodes that are linked to this node
    - The undefined become open and we compute their distance
    - The open nodes may have their distance modified
  - We close the node

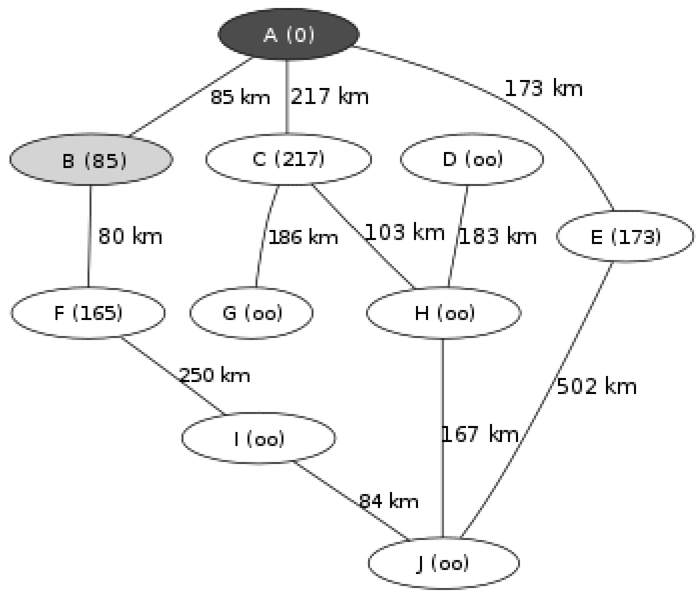
Path from A to J

A is open
Neighbors of A : B,C and E
Open with
D(A,B) ≤ 85
D(A,C) ≤ 217
D(A,E) ≤ 173
OpenNodes={B,E,C}



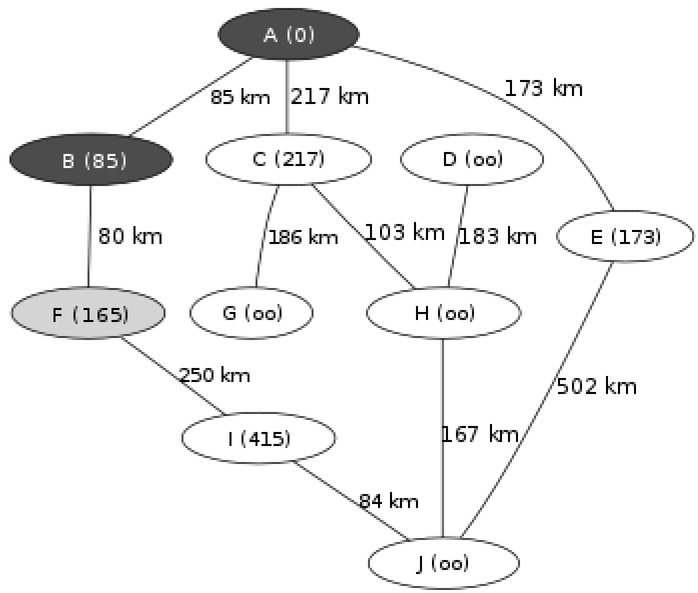
Path from A to J

We add the open node
Having the smallest distance.
Here, it is B.
We look at the neighbors of B
We open F
D(A,F) ≤ d(A,B) + d(B,F) ≤ 165
We close B
OpenNodes = {F,E,C}



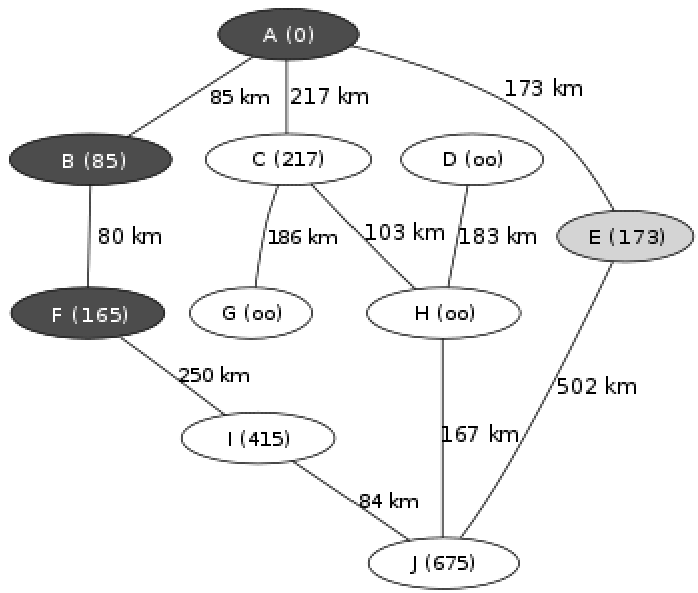
Path from A to J

We select F.
We open I
D(A,I) ≤ d(A,F) + d(F,I) ≤ 415
We close F
OpenNodes = {E,C,I}



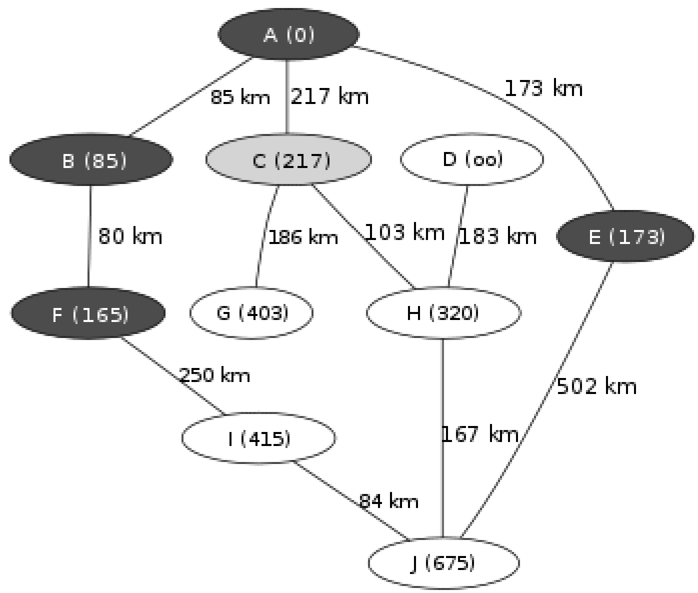
Path from A to J

We select E.
We open J J
$D(A,J) \leq d(A,E) + d(E,J) \leq 675$
We close E
OpenNodes = {C,I,J}



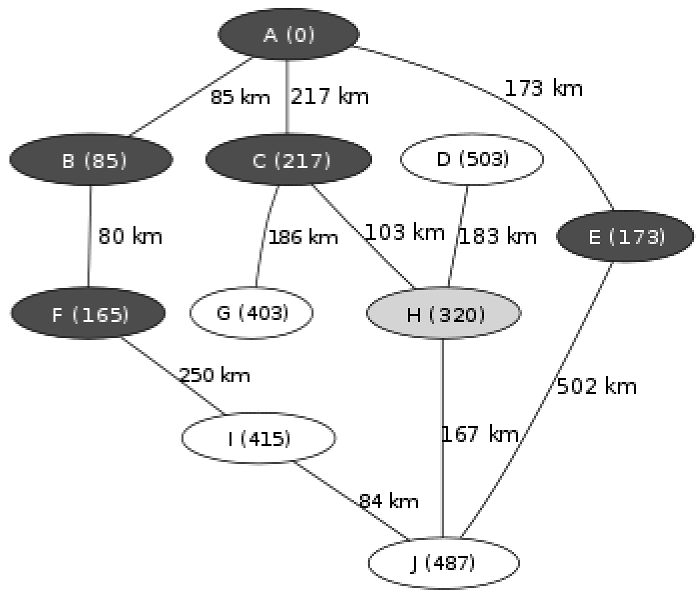
Path from A to J

We select C
We open G and H
OpenNodes = {H,G,I,J}



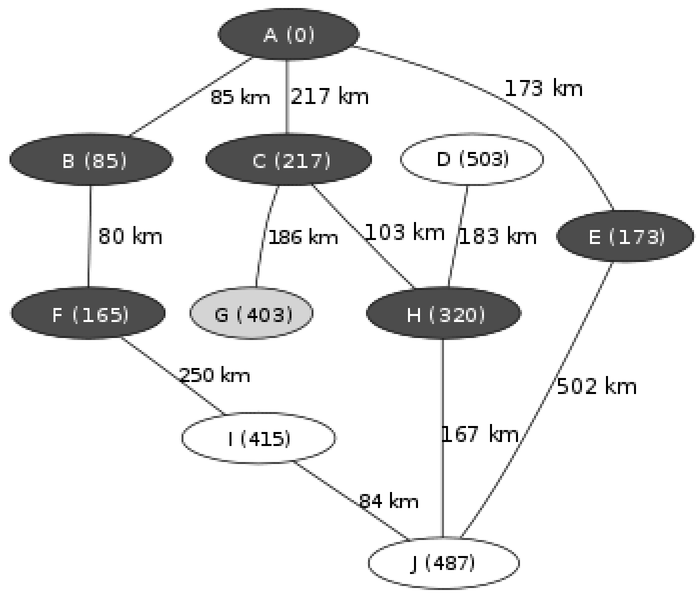
Path from A to J

We select H
We update J



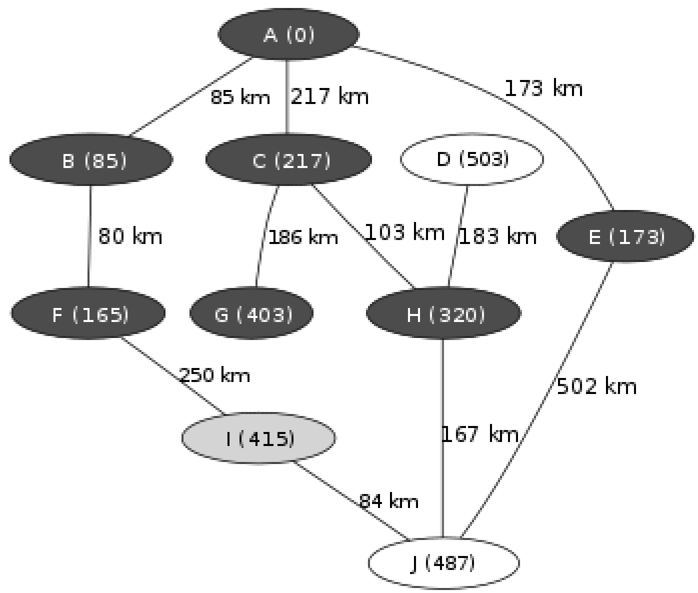
Path from A to J

We select G



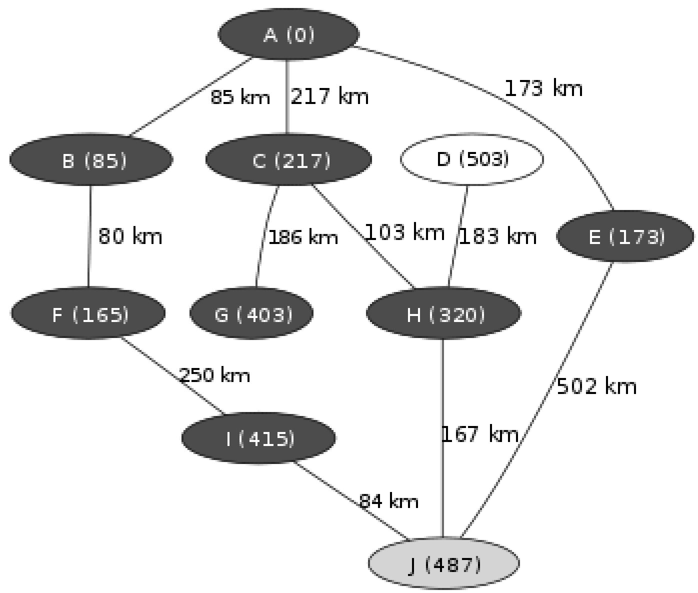
Path from A to J

We select I



Path from A to J

We select J
End of the algorithm



Dijkstra

- update_distances(s1,s2)
  if d[s2] > d[s1] + length(s1,s2)
  then d[s2] ← d[s1] + length(s1,s2)
- Dijkstra(G,Lengths,sdeb)
  OpenNodes ← {s}
  while OpenNodes ≠∅ and t is not closed
  do i ← Find_min(OpenNodes)
  remove i from OpenNodes
  close(i)
  for each node k neighbor of i
  do **if** k is not closed
  then add k to OpenNodes if it was not there
  updates_distances(i,k)
  done
  done
- To find the path we need to keep the predecessor (the one which update the distance)

Dijkstra's algorithm

- The main difficulty is the computation of the node which is the closest to the source
  - Priority Queue

Plan

- State Graph
- Path in a graph (DFS, BFS)
- Shortest path between two nodes
  - Dijkstra's algorithm
- **Very large Graph: algorithm A***
- Applications

Graph and solution

- A lot of problems can be solved easily if we are able to find a shortest path from a node s to a node t.
- In some cases we cannot define the graph because
  - it is too large
  - it is dynamic
  - we do not have a complete information

Some issues with the graph

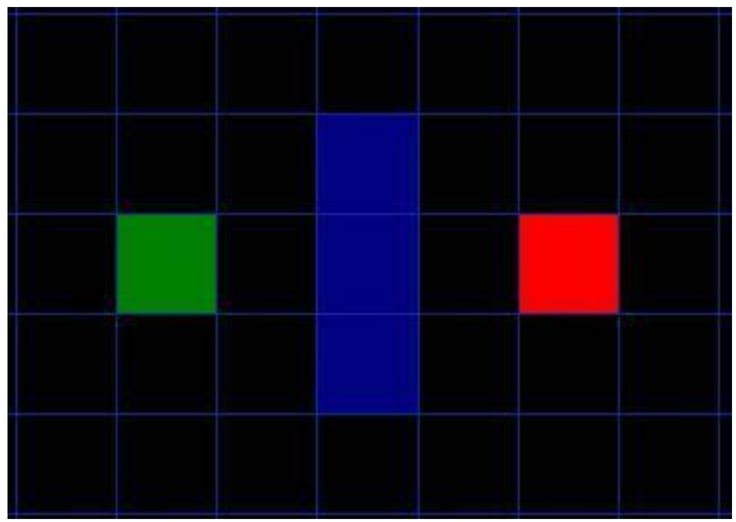
- We will try to limit the combinatorial explosion
- We will traverse the graph and build it when needed
  - For Dijkstra we just need the neighbors of a node
  - We would like to limit the number of open nodes

- How to limit the number of open nodes?
  - **With a guide!**

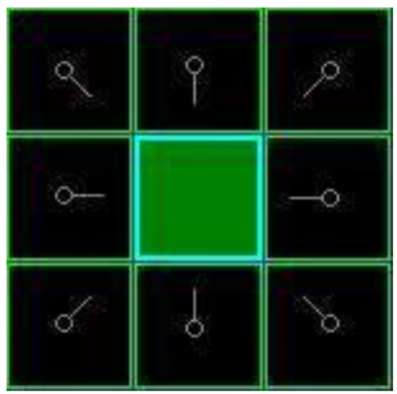
- Consider a function *f* capable of evaluate a node and defined as follows:
- $f(s,n,t) = g(s,n) + h(n,t)$ with
  - $g(s,n)$ the length of the best known path to reach n from s
  - $h(n,t)$ the estimation of the length of the path from n to t
- $f(s,n,t)$ is an estimation of the shortest path from s to n traversing n.

- What is the advanatge of this?
  - With Dijkstra we consider only the distance from s
    - We select the node the closest to the source
    - We update this distance
  - With the algorithm A*
    - We select the node having the smallest f
    - We update g
- We prevent the search from going into all directions: we focus its attention to the goal

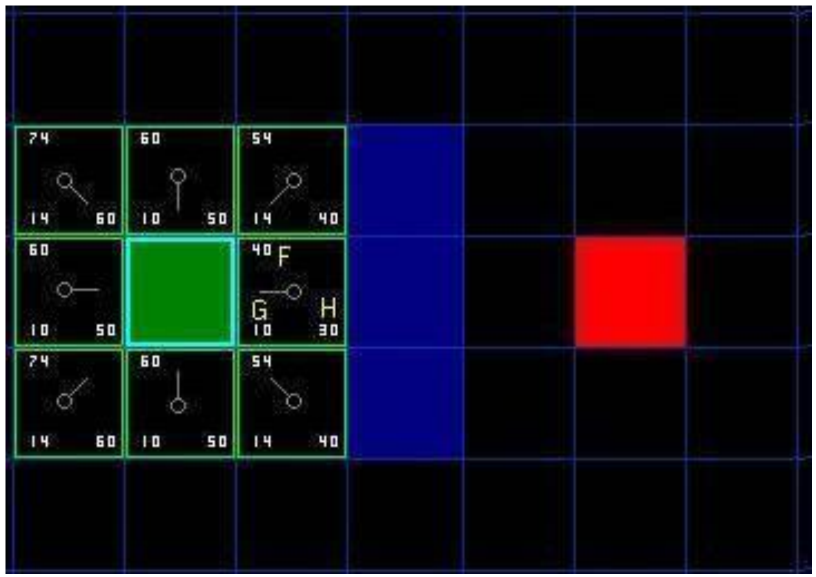
Algorithm A*



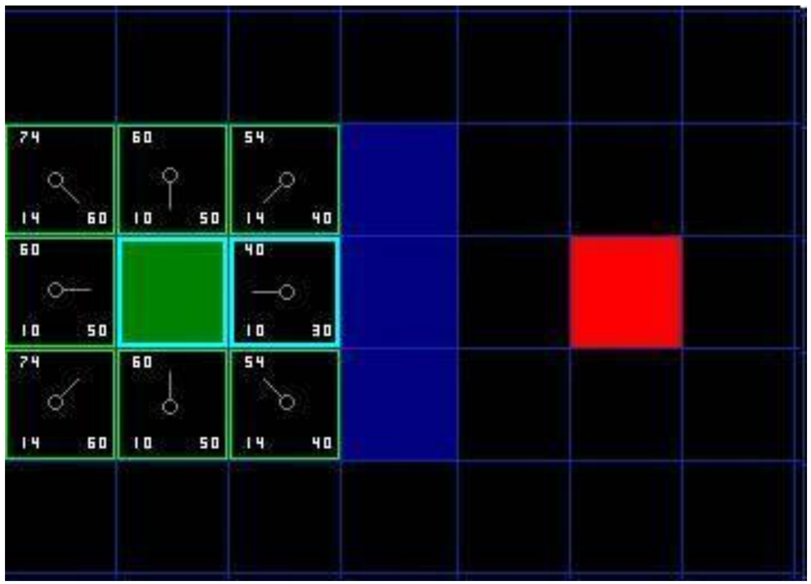
Algorithm A*



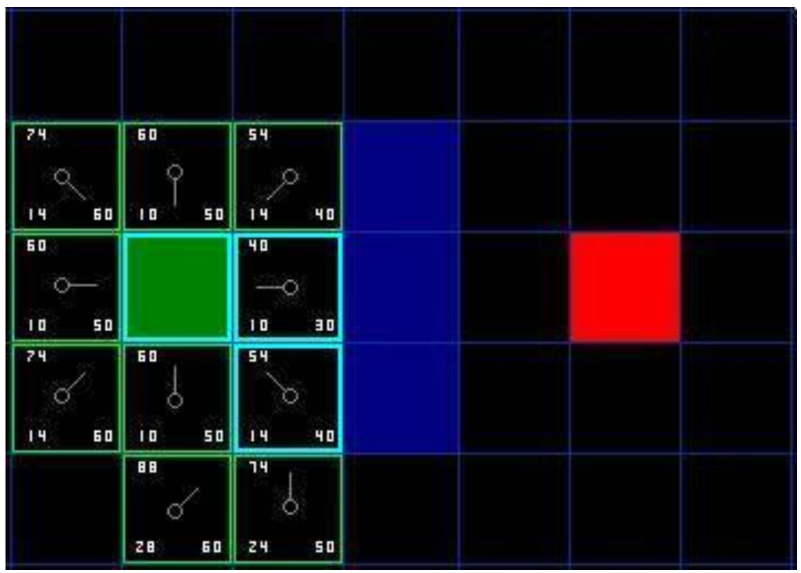
Algorithm A*


74
14
60
60
10
50
54
14
40
60
10
50
40
F
G
H
10
30
74
14
60
60
10
50
54
14
40

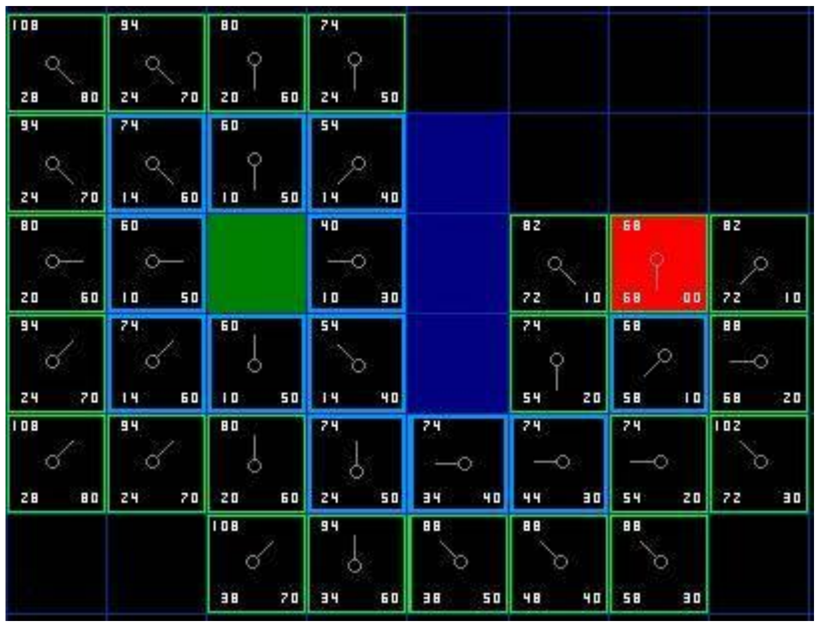
Algorithm A*



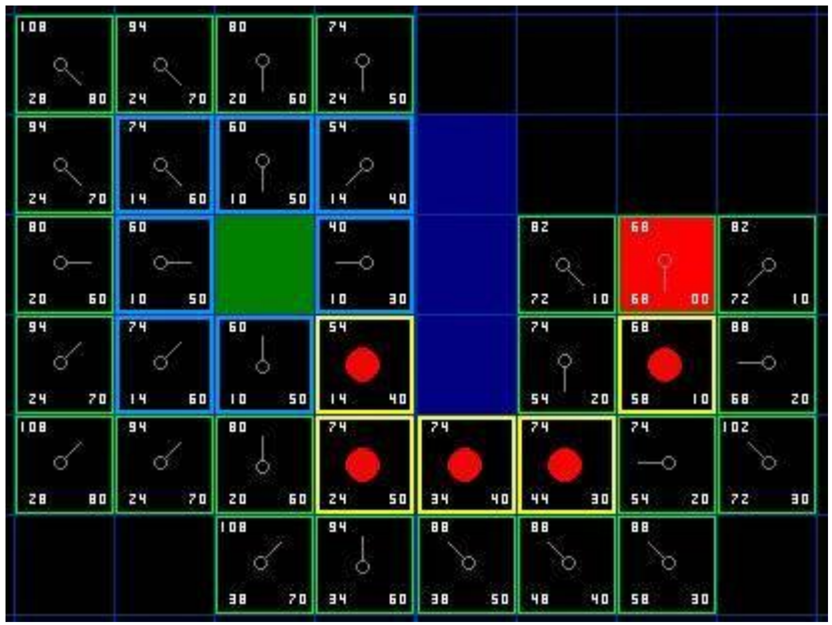
Algorithm A*


74
14
60
60
10
50
54
14
40
60
10
50
40
10
30
74
14
60
60
10
50
54
14
40
88
28
60
74
24
50

Algorithm A*


108 28 80
94 24 70
80 20 60
74 24 50
94 24 70
74 14 60
60 10 50
54 14 40
80 20 60
60 10 50
40 10 30
82 72 10
68 68 00
82 72 10
94 24 70
74 14 60
60 10 50
54 14 40
74 54 20
68 58 10
88 68 20
108 28 80
94 24 70
80 20 60
74 24 50
74 34 40
74 44 30
74 54 20
102 72 30
108 38 70
94 34 60
88 38 50
88 48 40
88 58 30

Algorithm A*


108 28 80
94 24 70
80 20 60
74 24 50
94 24 70
74 14 60
60 10 50
54 14 40
80 20 60
60 10 50
40 10 30
82 72 10
68 68 00
82 72 10
94 24 70
74 14 60
60 10 50
54 14 40
74 54 20
68 58 10
88 68 20
108 28 80
94 24 70
80 20 60
74 24 50
74 34 40
74 44 30
74 54 20
102 72 30
108 38 70
94 34 60
88 38 50
88 48 40
88 58 30

Algorithm A*

- update_distances_g(s1,s2)
```
        if g[s2] > g[s1] + length(s1,s2)
        then g[s2] ← g[s1] + length(s1,s2)
```
- Dijkstra(G,Lengths,sdeb)
```
        OpenNodes ← {s}
        while OpenNodes ≠∅ and t  is not closed
        do  i ← Find_min_f(OpenNodes)
                remove i from OpenNodes
                close(i)
                for each node k neighbor of i
                do      if k is not closed
                        then    add k to OpenNodes if it was not there
                                updates_distances_g(i,k)
                done
        done
```
- To find the path we need to keep the predecessor (the one which update the distance)

Algorithm A*

- Comment trouver et relier f, g et h ?
  - si h=0, alors f(n)=g(n) et l'algorithme se comporte comme une largeur d'abord
  - si f(n) = h(n), l'algorithme ressemble à une profondeur d'abord, on parle de *gradient*
- Propriétés de h et de l'algorithme A*
  - h est dite *minorante* si pour tout noeud *n*, on a *h(n)* inférieure ou égale à *h*(n)* ;
- si h est minorante alors l'algorithme A* est admissible (il trouvera la solution optimale)

Algorithm A* : Beam Search

- Even if the algorithm A* does not developp systematically all the nodes, the number of open nodes may become prohibitive.
- A solution is to keep only the **k** best nodes.
- This is a ***beam search.***

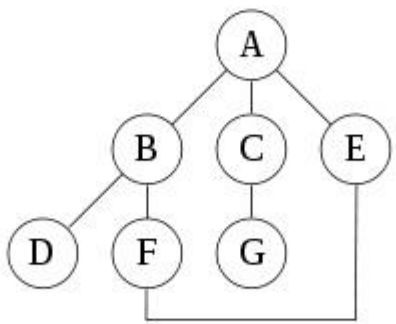
- Inconvénients
  - We may not find the optimal solution
  - We may not find a solution
- Avantage
  - We reduce the combinatorial explosion

IDDFS: Iterative Deepening DFS

- This is a repeted DFS with a boudned depth
  - We perform a DFS with a depth limit equals to 1
  - We perform a DFS with a depth limit equals to 2
  - …
  - We perform a DFS with a depth limit equals to k

- This is a way to mix a DFS and a BFS
- This is interesting for the game algorithm because we augment our knowledge with a BFS before going to deeply with a DFS

IDDFS



Depth max = 0
A

Depth max = 1
A, B, C, E

Depth max = 2
A, B, D, F, C, G, E

DFS : A, B, D, F, E, C, G (C is visited late)
BFS : A, B, C, E, D, F, G

1.86

Two players games

Two players games

- Minimax Algorithm
- Alpha-Beta cuts

Two players games

- We consider games in which players play **successively**
- We consider games with a complet information: each player has all the information when he plays. Card based games are incomplete games.
- In the following
  - we will denote by J_1 and J_2 the two players.
  - We will consider that this is the turn of J_1. He has to play

Two players games

- A game can be seen as a tree:
  - The root corresponds to the current position
  - Nodes having an even depth = nodes where J_1 has to play
  - Nodes having an odd depth = nodes where J_2 has to play
  - an arc = a move
  - leaves = ends of the game: the winning and the loosing states for J_1, or the blocking states (draws)
- An arc link a state where the move is allowed to the state where the move is effectively done
- A pth from the root to a leave describes a possible game.

Two players games

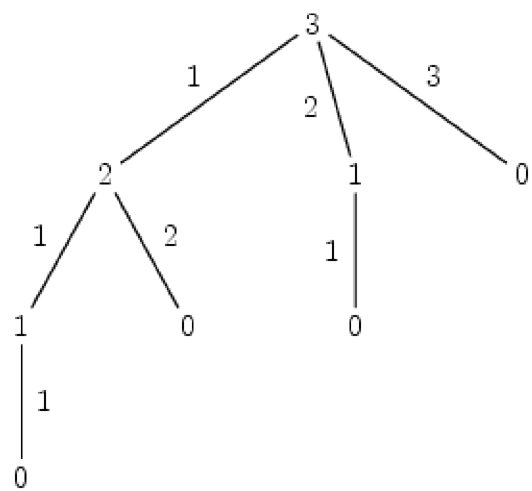
- Similarity with the state graph:
  - The initial state is the current situation
  - The final states are the ends of the game
  - The arcs corresponds to legal moves
- However, one important information is missing: the fact that there are 2 players who play successively. The moves alternate => specific algorithms

Two players games

- We will consider first a simple problem:
  - The Nim game (match game in France): we have a set of matches. At each move, a player can take 1, 2 or 3 matches. The player who takes the last one loses.

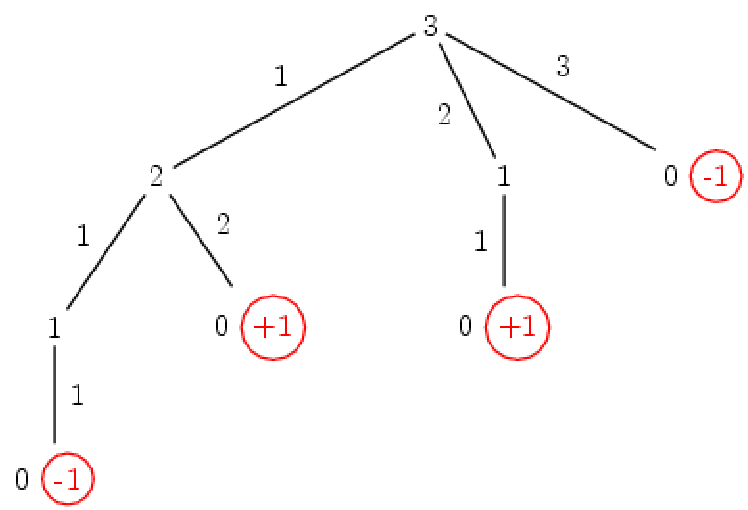
Two players games

- We build the tree game



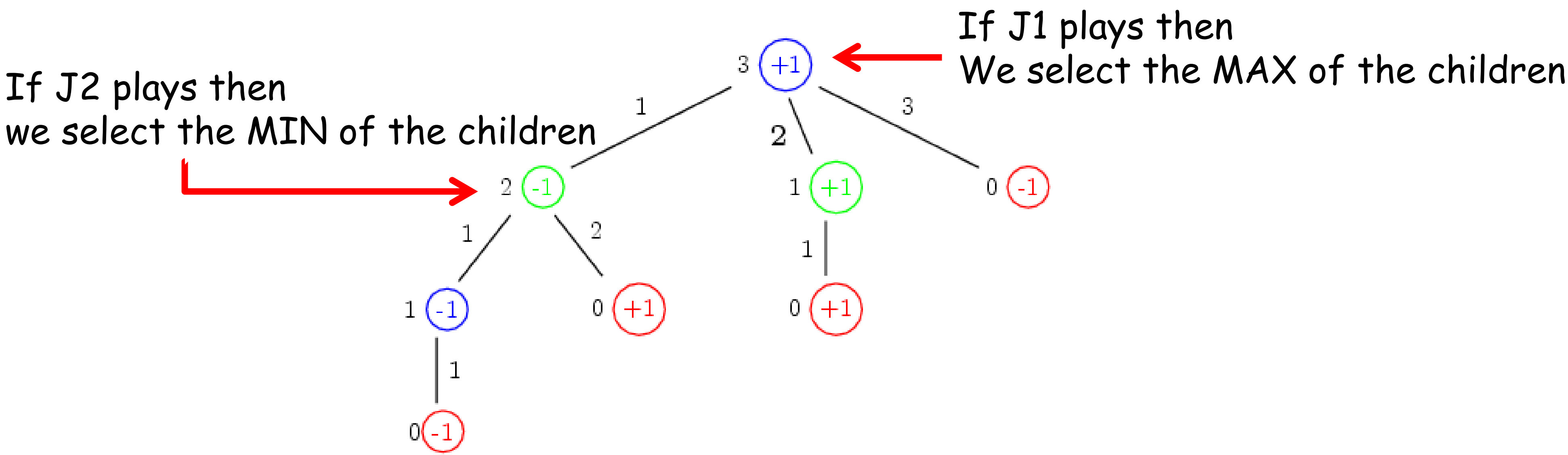
Two players games

- We evaluate the terminal positions



Two players games

- The evaluations are moving up



Minimax

- **DecisionMinMax** (e,J)

```
// Decide the best move of J in position e

For each move m of PossibleMoves(e,J)
      value[m] = MinMaxValue(Apply(m,e),J,false)
return (m such that value[m] is maximal)
```

- **MinMaxValue** (e,J,IsMax)

```
// Compute the value of e for the player J depending whether e IsMax or not

If WinningPosition(e,J) then return (+1)
If LoosingPosition (e,J) then return(-1)
If DrawPosition(e,J) then return(0)

vals = empty
For each move m of PossibleMoves(e,J)
      add MinMaxValue(Apply(m,e),opponent(J),not(IsMax)) to vals

If IsMax then return (maximum of vals)
Else return (minimum of vals)
```

Minimax

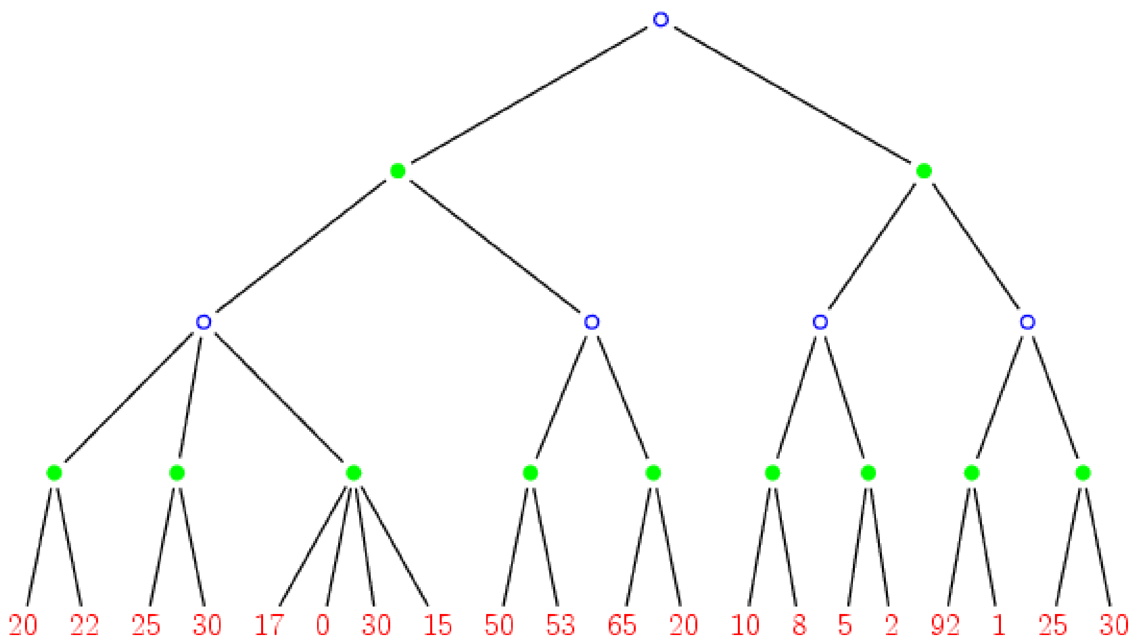
- In practice, the Minimax algorithm is not applied in that way because it is very rare to be able to develop entirely the game tree.
- Two adaptations are common
  - limit the exploration depth;
  - Make some cuts in the game tree

Minimax with bounded depth

- Stop the exploration of the game tree before reaching ends of game implies that we can evaluate the state of the game when we stop the exploration. In other words, at any moment.
- This is possible provided that we have
  - an **evaluation function h(e,J)** which evaluates the position e for the player J.
- The game is always evaluated against the **same player**
  - Positive if it is in favor of the player (good position)
  - Negative if it is in disfavor of the player (bad position)

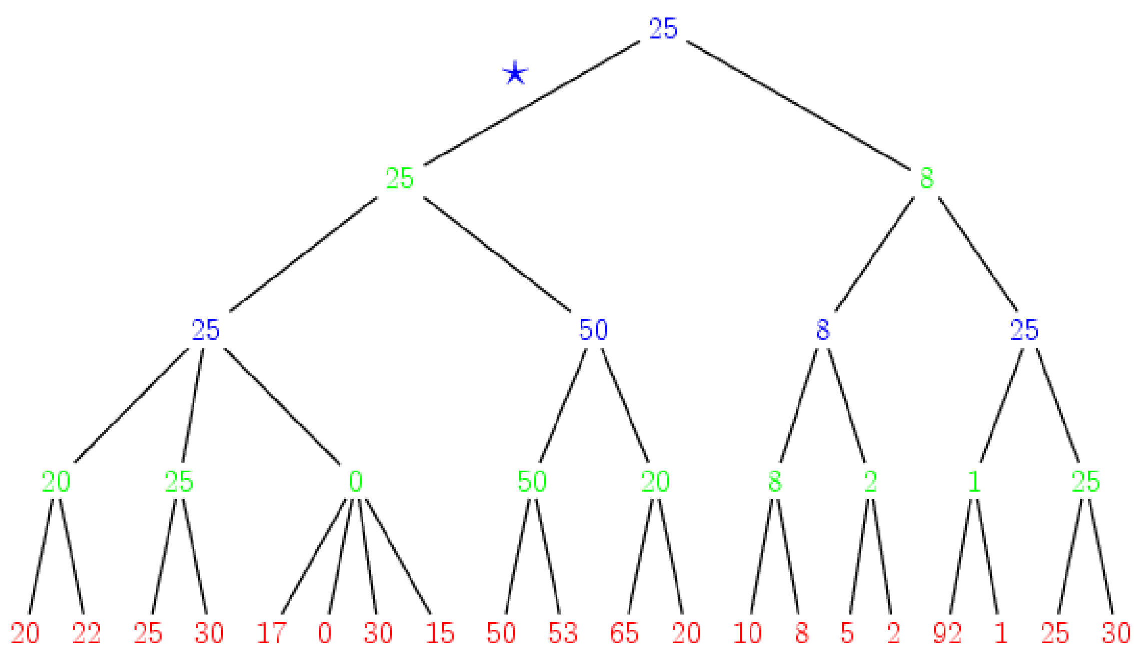
Minimax with bounded depth

- We limit the exploration at the depth 4. We evaluate the leaves with the evaluation function



Minimax with bounded depth

- The evaluations are moving up



Minimax with bounded depth

- **DecisionMinMax** (e,J,**pmax**)

```
// Decide the best move of J in position e

For each move m of PossibleMoves(e,J)
      value[m] = MinMaxValue(Apply(m,e),J,false,pmax)
return (m such that value[m] is maximal)
```

- **MinMaxValue** (e,J,IsMax, **pmax**)

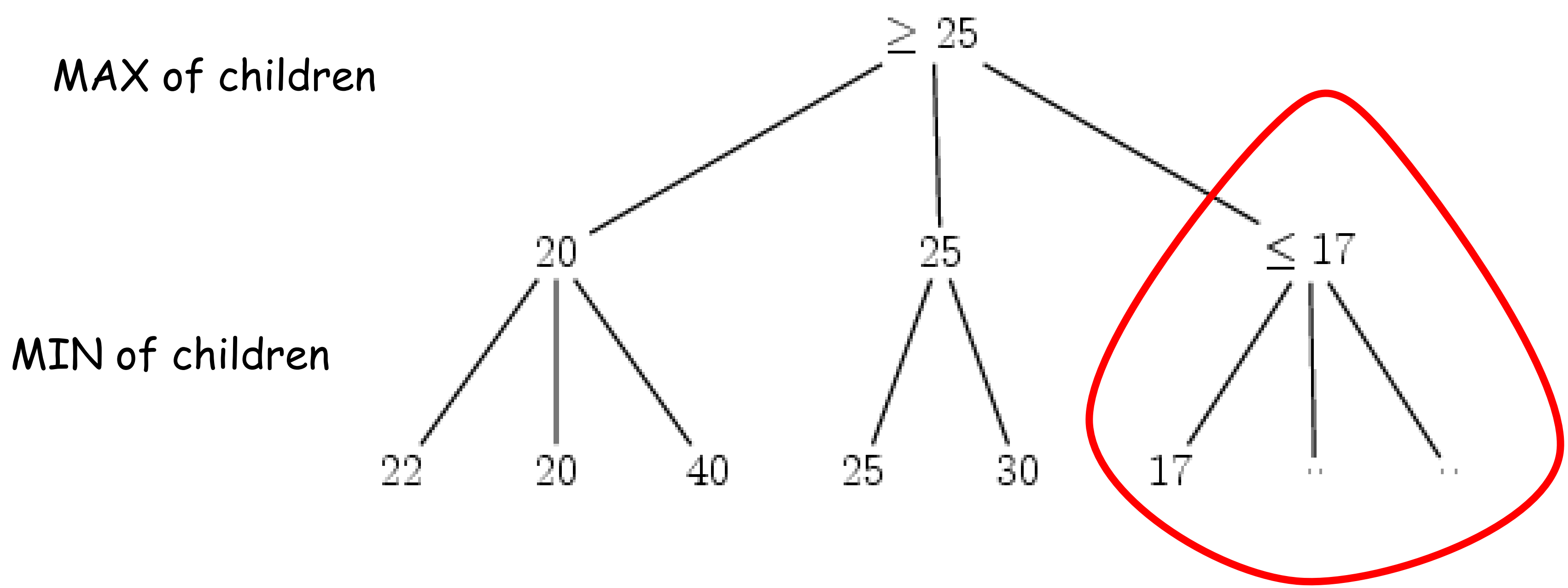
```
// Compute the value of e for the player J depending whether e IsMax or not
// pmax is the maximal depth

If WinningPosition(e,J) then return (+valMax)
If LoosingPosition (e,J) then return(-valMax)
If DrawPosition(e,J) then return(0)

If pmax=0 then return h(s,J)
vals = empty
For each move m of PossibleMoves(e,J)
      add MinMaxValue(Apply(m,e),opponent(J),not(IsMax),pmax-1)) to vals
If IsMax then return (maximum of vals)
Else return (minimum of vals)
```

Minimax with bounded depth

- The **evaluation function** is quite important
- We can obtain very different results depending on that function
- Programmation: two things
  - Enumerate the possible moves (and valid)
  - Define an evaluation function

Alpha Beta Cuts

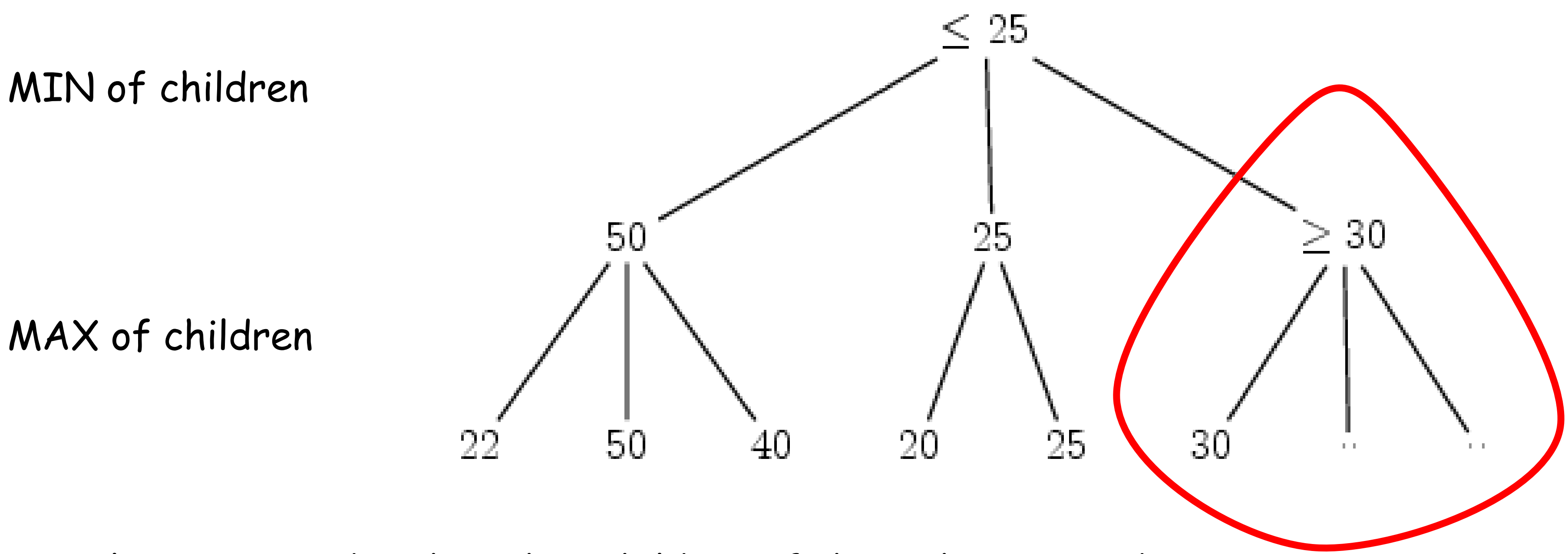
- We can avoid exploring some subtrees
- Alpha Cut (MAX)



It is useless to consider the other children of the rightmost node because 17 will never be selected

Alpha Beta Cuts

- We can avoid exploring some subtrees
- Beta Cut (MIN)



It is useless to consider the other children of the rightmost node because 30 will never be selected

Minimax with Alpha-Beta cuts

- **DecisionAlphaBeta** (e,J,pmax)
```
DecisionAlphaBeta (e,J,pmax)
// Decide the best move to play for J in the position e
alpha ← -ValMax
For each move m of PossibleMoves (e,J)
       val ← AlphaBetaValue(Apply(m,e),J,alpha,+MaxVal,false,pmax)
       If (val>alpha) then
                  action ← m
                  alpha = val
return action
```

- **AlphaBetaValue** (e,J,alpha,beta,IsMax,pmax)
```
AlphaBetaValue (e,J,alpha,beta,IsMax,pmax)
// Compute the value e for the player J depending on e.pmax is the maximal depth
If WinningPosition(e,J) then return (+valMax)
If LoosingPosition (e,J) then return(-valMax)
If DrawPosition(e,J) then return(0)

If pmax=0 then return h(s,J)

If IsMax then
    For each move m of PossibleMoves(e,J)
       alpha ← MAX(alpha,AlphaBetaValue(Apply(m,e),opponent(J),alpha,beta,not(IsMax),pmax-1)
       If alpha >= beta then return alpha /* beta cut */
    return alpha

/* Min */
For each move m of PossibleMoves(e,J)
    beta ← MIN(beta,ValeurAlphaBeta(Apply(m,e),opponent(J),alpha,beta,not(IsMax),pmax-1)
   If beta <= alpha then return beta /* alpha cut */
return beta
```

Transposition Table

Symmetries

- Tic-tac-toe
- Chess
- Checkers

Tic-tac-toe

- Compute the number of possible games in tic-tac-toe and the number of different nodes
- Draw the tree search graph from a position where the two players have played once
- Use the symmetries, transposition table and cut to reduce the number of nodes

NegaMax

- This is a simplification of the implementation of the minimax algorithm based on the fact that we have
  - max(a,b) = -min (-a,-b)

SSS*

Computers vs Humans

- **Puissance 4**
  - Le jeu est résolu. Cela signifie que l'on a montré qu'il existait une stratégie gagnante pour le joueur qui commence à jouer. Cette stratégie étant stockée dans une base de données, il est facile pour un joueur artificiel de suivre cette stratégie et de gagner systématiquement.
- **Othello**
  - Pour ce jeu, il n'y a pas de confrontation entre joueurs humains et joueurs artificiels. Il est clair que les joueurs artificiels sont supérieurs.
- **Les dames**
  - Le programme Chinook basé sur un alpha-bêta est devenu champion des États-Unis en 1992, puis champion du monde en 1994. Ce joueur utilise également une bibliothèque de fins de partie (tous les damiers comportant 8 pièces ou moins).
- **Les échecs**
  - DeepBlue bat Kasparov en 1997. Depuis, toutes les confrontations tournent systématiquement à l'avantage de la machine. Utilisation de l'algorithme alpha-bêta, le facteur de branchement vaut ici 40.
  - Deep Junior coûte € 60 http://www.hiarcs.com/pc-chess-junior-buy.htm
- **Le Go**
  - C'est le jeu où les machines ne sont pas du tout compétitives. Les joueurs artificiels sont de niveau amateur. L'algorithme alpha-bêta n'est pas utilisable dans ce cas (le facteur de branchement au Go est de 360). Une prime importante (2 millions de dollars) est promise pour le premier programme qui battra un champion humain.