

Graph Theory

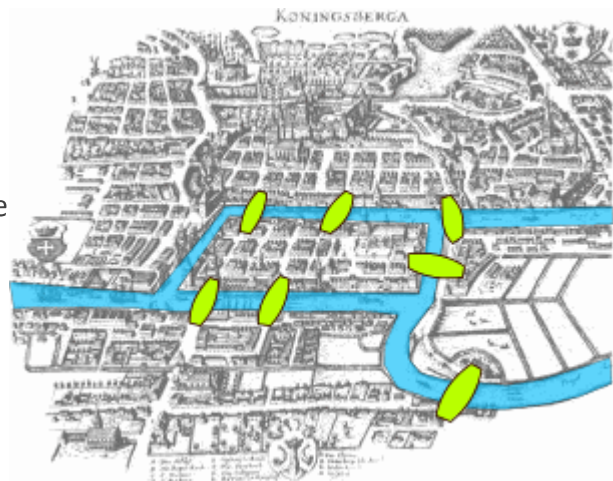
- Voir [Wes22; Chapter 1-2]
- ou [CLRS22; Chapter 22-24]

[Wes22] D. B. West. *Introduction to Graph Theory* (2022).

[CLRS22] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms, Fourth Edition* (MIT Press, 2022).

Seven Bridges of Königsberg ⇌

Est-il possible de prendre tous les ponts de la ville une et une seule fois (le point de départ est au choix) ?

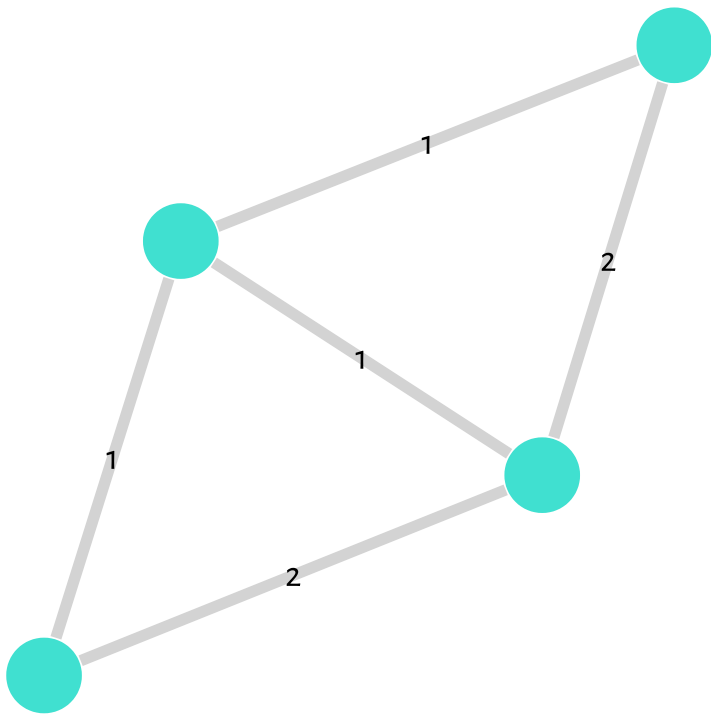


Définition ⇄

Un graph a un ensemble V de noeuds (nodes) / sommets (vertices) et E d'arêtes (edges) / arcs.

Math	Informatique	Non-dirigé	Dirigé
V	Nodes	Sommet / vertex	Noeud / node
E	Edges	Arête / edge	Arc

Le graphe du problème de Königsberg test:



Graphes et polyèdres ⇔

Prenons un polyèdre, où se cache le graphe ?

On associe un sommet du graphe à chaque sommet du polyèdre et une arête à chaque arête (c'est la même terminologie, coïncidence ?).

Relation d'Euler: le nombre de faces est égal à $2 - |V| + |E|$.

Terminologie des parcours

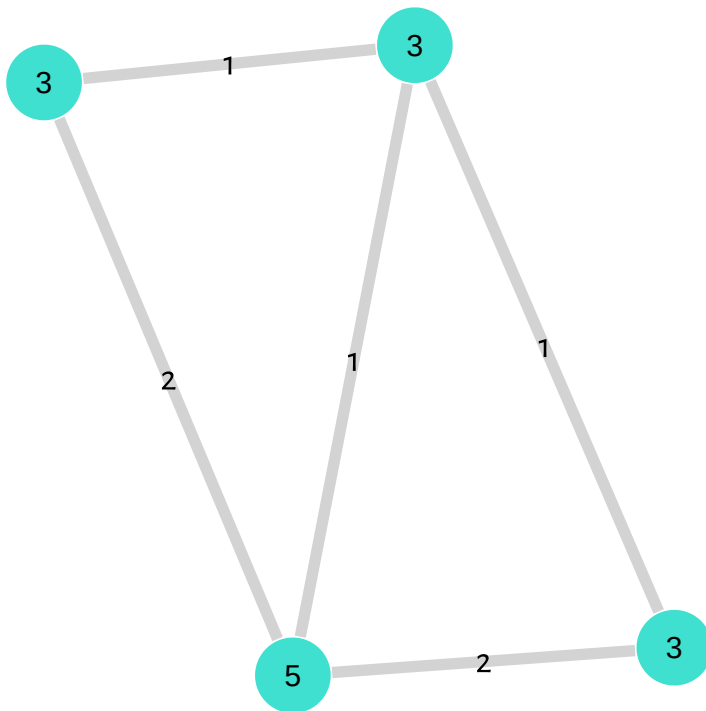
Un parcours / walk de longueur k d'un graphe $G = (V, E)$ est une suite de $k + 1$ noeuds $v_0, v_1, \dots, v_k \in V$ et k arêtes $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k) \in E$. Si $v_0 = v_k$, le parcours est fermé.

	Noeuds distincts	Arêtes distinctes
Ouvert	Chemin	Piste / Trail
Fermé	Cycle	Circuit

Piste (resp. circuit) *Eulérienne* : Une piste (resp. circuit) qui visite **toutes** les arêtes.

Degré d'un noeud ⇔

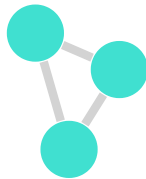
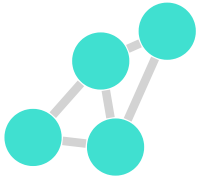
Le degré d'un noeud v est le nombre d'arête qui lui sont incidentes.



```
1 let
2   g = Graph(4)
3   add_edge!(g, 1, 2)
4   add_edge!(g, 1, 4)
5   add_edge!(g, 2, 3)
6   add_edge!(g, 2, 4)
7   add_edge!(g, 3, 4)
8   gplot(g, nodelabel = [3, 3, 3, 5], edgelabel = [1, 2, 1, 1, 2])
9 end
```

Composante connexe

Une composante connexe C est un ensemble de noeuds tels que toutes paires de noeuds est connectée par un chemin.



Calcul de composantes connexes

On démarre en assignant une composant connexe différente pour chaque noeud. Pour chaque arête, on fusionne les composantes connexes des deux noeuds liés par l'arête. Comment calculer cette fusion efficacement ?

Supposons qu'on ait 4 noeuds. On démarre avec le tableau $(1, 2, 3, 4)$ signifiant que chaque noeud est dans une composante connexe différente.

- En commençant avec l'arête $(1, 2)$, on fusionne et on arrive au tableau $(1, 1, 3, 4)$.
- Si on voit ensuite l'arête $(3, 4)$, on arrive alors au tableau $(1, 1, 3, 3)$.
- Si on voit ensuite l'arête $(1, 3)$, on update le tableau à $(1, 1, 1, 1)$.

► Quelle est la complexité de cet algorithm ?

Disjoint-Set datastructure ⇄

Pour améliorer l'algorithme, pour la fusion de l'algorithme **(1, 3)**, on peut updater le tableau à **(1, 1, 1, 3)**. Pour le noeud 4, on encode donc qu'il est dans la même connected component que le noeud 3 qui est dans la même connected component que le noeud 1. Le noeud 4 est donc lié indirectement au noeud 1 qui est appelé sa racine.

► **Quelle est complexité après cette amélioration ?**

De façon surprenante, si on met à jour la valeur dans le tableau pour mettre directement le root après l'avoir calculé, la complexité passe à $O(|V|\alpha(|V|))$ où α est la réciproque de la fonction d'Ackermann, une fonction qui augmente extrêmement lentement donc en pratique, c'est presque $O(|V|)$.

Piste Eulérienne

Théorème [Wes22; Theorem 1.2.26] Considérons un graphe avec une seule composante connectée. Il existe un circuit Eulérienne si et seulement si tous les noeuds ont un degré pair. Si tous les noeuds ont un degré pair sauf 2 alors il n'existe pas de circuit mais une piste Eulérienne.

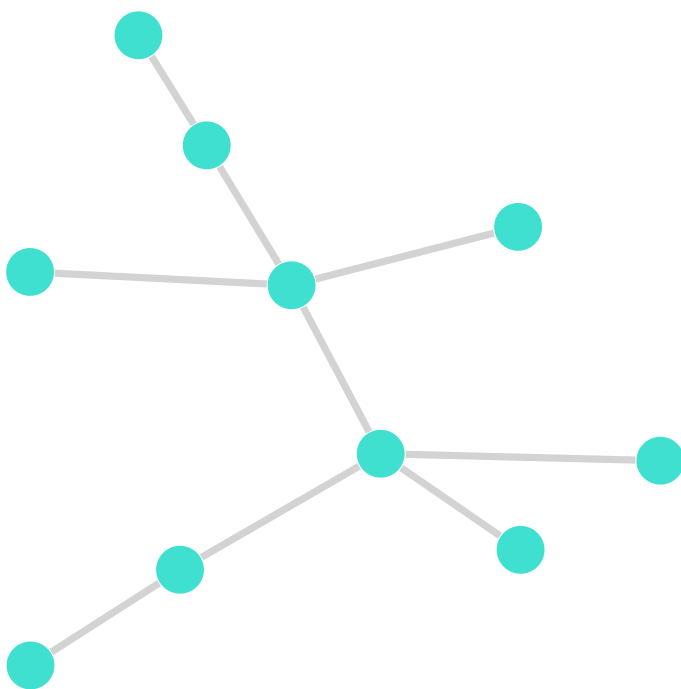
► **Proof**

Arbres et forêts ⇄

Définition

- Une *forêt* (forest) est un graphe qui n'a **pas** de **cycle**.
- Un *arbre* (tree) est une forêt avec **une** seule **composante** connexe.

Par abus de langage, on parle souvent d'arbre sans se soucier de si le graphe est connecté ou pas.



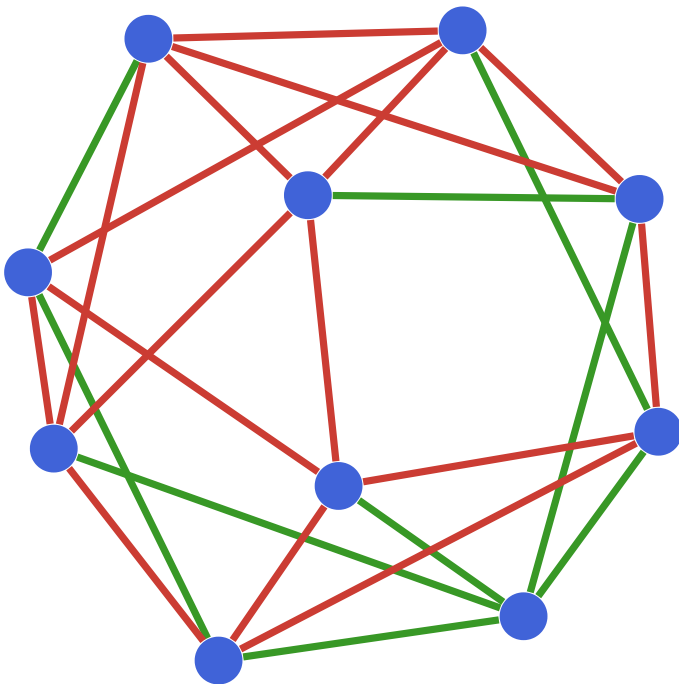
```
1 gplot(uniform_tree(10))
```

Spanning tree ⇔

Étant donné un graphe $G(V, E)$, sa *forêt sous-tendante* (spanning forest) est une forêt $G'(V, E')$ où $E' \subseteq E$. Comment la trouver ?

spanning_forest (generic function with 2 methods)

```
1 function spanning_forest(g, edges_it = edges(g))
2   component = IntDisjointSets(nv(g))
3   function create_loop!(edge)
4     loop = in_same_set(component, src(edge), dst(edge))
5     union!(component, src(edge), dst(edge))
6     return loop
7   end
8   return [edge for edge in edges_it if !create_loop!(edge)]
9 end
```

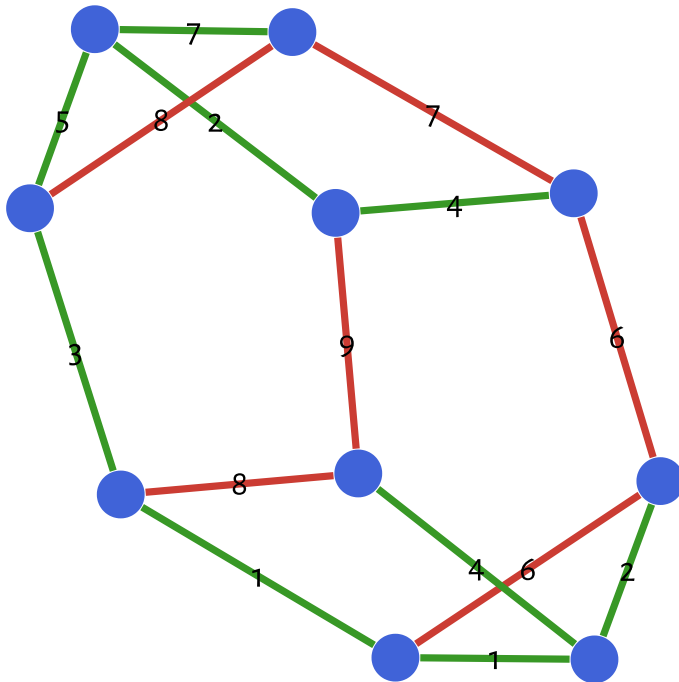


```
1 let
2   g = random_regular_graph(10, 5)
3   cols = [colors.red, colors.green]
4   tree = Set(spanning_forest(g))
5   gplot(g, edgestrokec = [cols[(edge in tree) + 1] for edge in edges(g)],
6         nodefillc = colors.blue)
7 end
```

Minimum spanning tree ↺

kruskal (generic function with 1 method)

```
1 kruskal(g) = spanning_forest(g, sort(collect(edges(g)), by = weight))
```

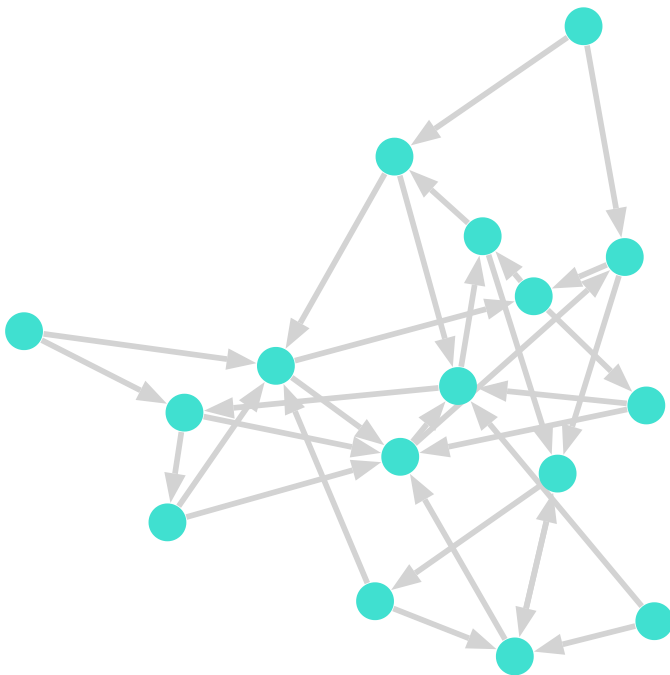


► Quelle est la complexité de Kruskal ?

Graphes dirigés ⇔

Dans un graphe dirigé, une arête (i, j) a un **sens**. Intuitivement, on peut aller de i vers j mais on ne peut qu'aller de j vers i si il y a aussi une autre arête de (j, i) .

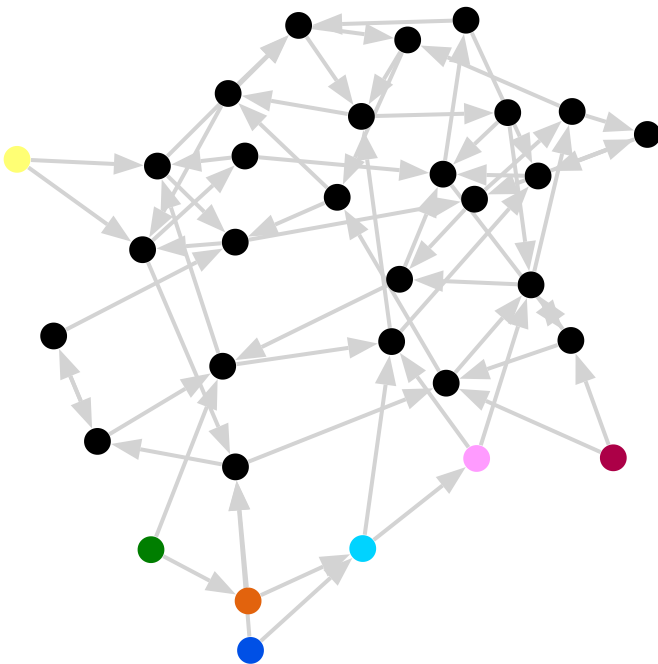
Que deviennent les notions de **connected components**, **spanning tree**, etc...



```
1 plot(random_regular_digraph(16, 2))
```

Strongly connected components ⇔

Définition Une **composante fortement connexe** d'un graphe dirigé $G(V, E)$ est un ensemble $V' \subseteq V$ tel qu'il existe un chemin de u vers v pour tous $u, v \in V'$.



```
1 let
2   Random.seed!(32)
3   g = random_regular_digraph(32, 2)
4   c = Set.(strongly_connected_components(g))
5   cols = distinguishable_colors(length(c))
6   tree = Set(spanning_forest(g))
7   gplot(g, nodefillc = [cols[findfirst(comp -> v in comp, c)] for v in
8     vertices(g)])
8 end
```

Directed Acyclic Graph (DAG) ⇄

Si un graph contient un cycle, tous les noeuds de ce cycle sont contenus dans la même composante fortement connexe. Si chaque composante fortement connexe est fusionnée en un noeud, il ne reste plus de cycle. Le graphe résultat n'a donc plus de cycle, il est dit *acyclique* (DAG).

On va voir que beaucoup de problème peuvent se voir comme un problème de calcul de chaque noeud d'un graphe où les arêtes représentent les dépendences de calcul.

Suite de Fibonacci ↺

$$F_n = F_{n-1} + F_{n-2} \quad F_2 = F_1 = 1$$

fib (generic function with 1 method)

```
1 function fib(n)
2     if n <= 2
3         return 1
4     else
5         return fib(n - 1) + fib(n - 2)
6     end
7 end
```

n =  10

55

```
1 @time fib(n)
```

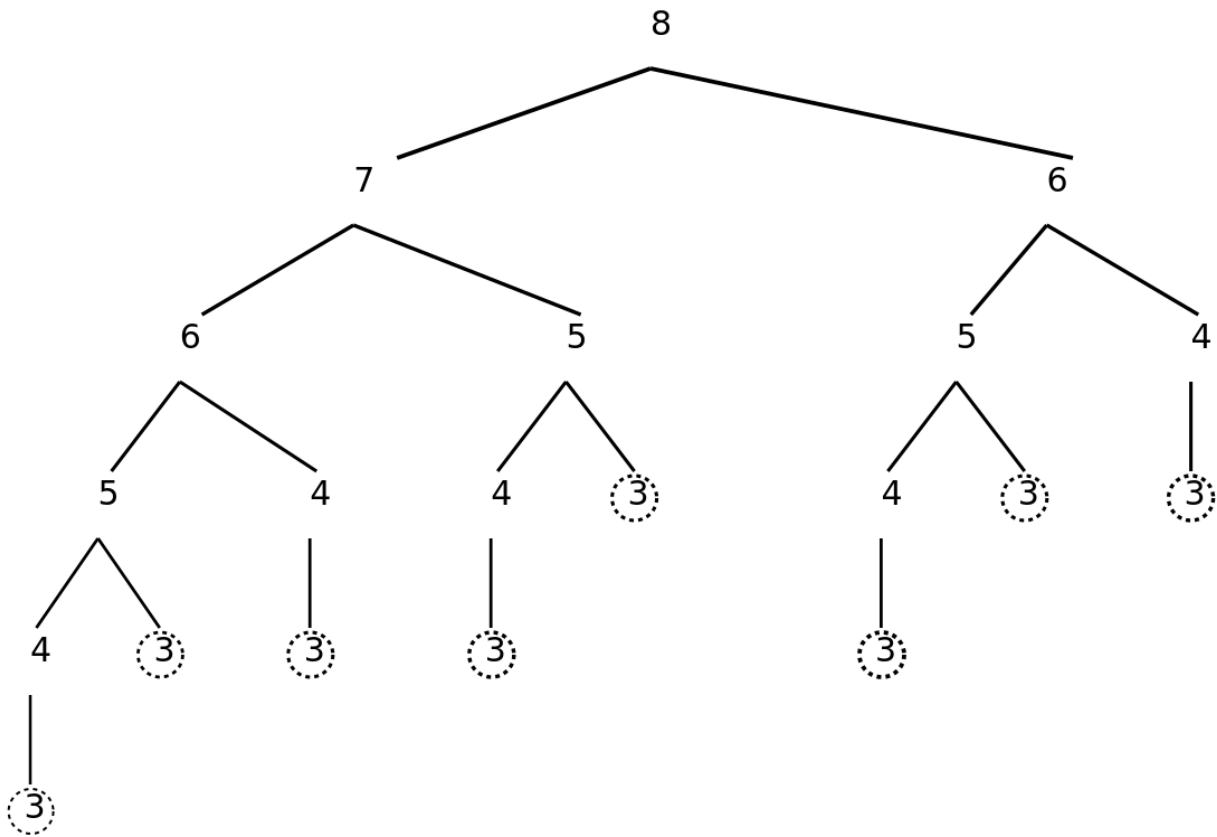


0.000001 seconds



Combien de **fois** fib(3) est appelé quand l'utilisateur appelle fib(8) ?

Visualisation des appels récursifs ↗

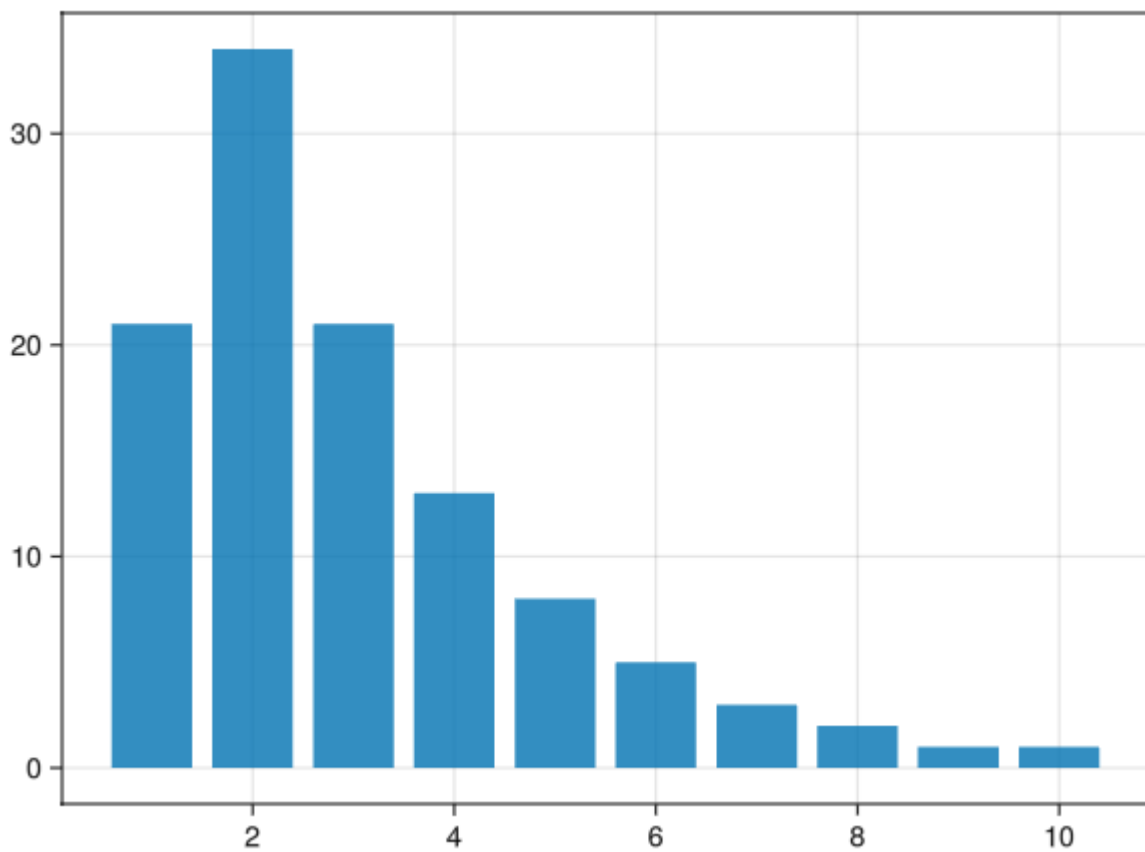


```
1 draw_fibo(8)
```

Nombre d'appels ⇄

n = 10

Nombre d'appels de `fib(k)` venant de `fib(n)`



```
1 Makie.barplot(1:n, called)
```

Bottom-Up / Dynamic Programming

Définition *Ordre topologique*: Ordre des noeuds dans lequel chaque noeud u apparait après tous les noeuds v tels qu'il y a un chemin de u vers v .

n =  10

fib_bottom_up (generic function with 1 method)

```
1 function fib_bottom_up(n)
2     fib = zeros{Int, n}
3     fib[1] = fib[2] = 1
4     for k in 3:n
5         fib[k] = fib[k - 1] + fib[k - 2]
6     end
7     return fib[n]
8 end
```

55

```
1 @time fib_bottom_up(n)
```



0.000002 seconds (2 allocations: 144 bytes)



Top-Down / Mémoïzation

n =  10

fib! (generic function with 1 method)

```
1 function fib!(cache, n)
2     if cache[n] == 0
3         cache[n] = fib!(cache, n - 1) + fib!(cache, n - 2)
4     end
5     return cache[n]
6 end
```

cached_fib (generic function with 1 method)

```
1 function cached_fib(n)
2     cache = zeros{Int, n}
3     cache[1] = cache[2] = 1
4     return fib!(cache, n)
5 end
```

55

```
1 @time cached_fib(n)
```



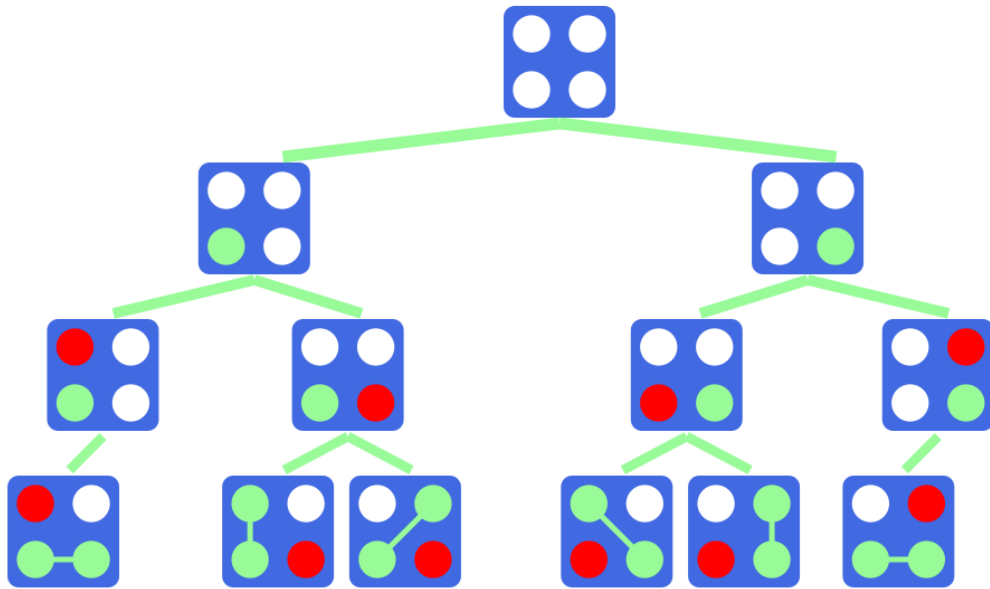
0.000002 seconds (2 allocations: 144 bytes)



Une IA pour le jeu Puissance 4

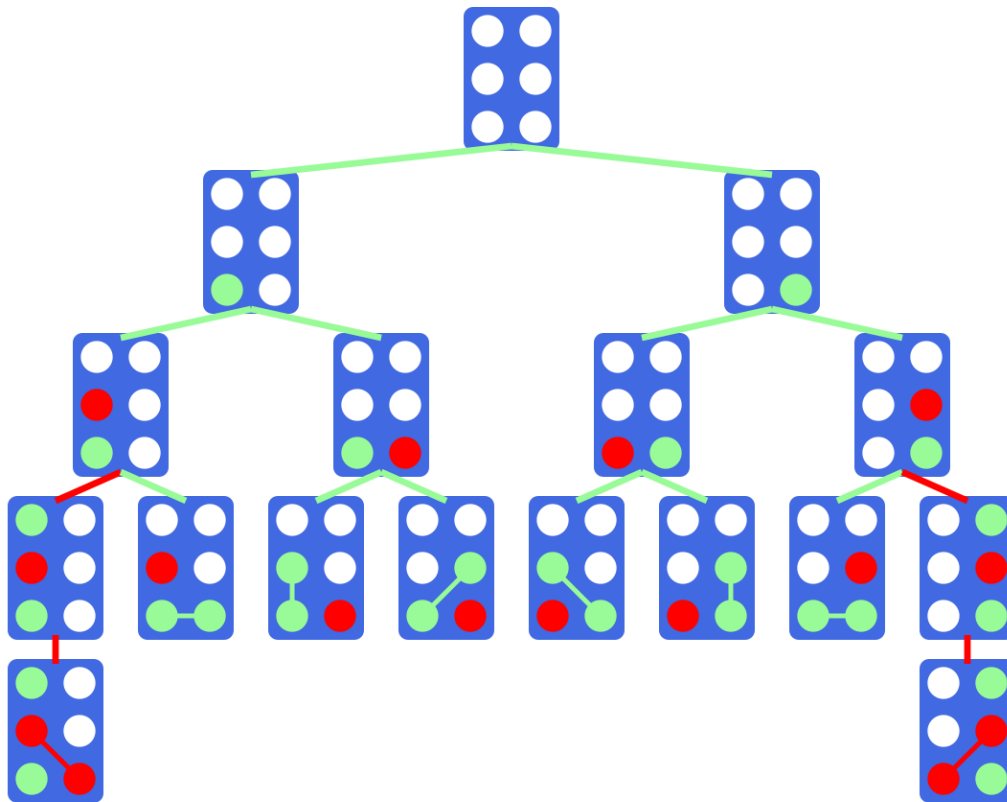
- Le Puissance 4 de 7 colonnes de hauteur 6 est résolu par ordinateur en 1988
- Utilisation de l'algorithme **minimax** vu au cours S4...
- ...mais avec l'aide de la programmation dynamique!

Puissance 2 de 2 colonnes de hauteur 2 ⇌



```
1 minimax(2, 2, 2, scaling = 1.7)
```

Puissance 2 de 2 colonnes de hauteur 3 $\Leftrightarrow$



Puissance 2 : sans mémoïzation

Colonnes =  3

Hauteur =  3

Longueur =  2

`cols = ▶ [1, 1, 2]`

```
1 cols = @time game(nrows, ncols, longueur)
```



```
0.295667 seconds (281.12 k allocations: 14.057 MiB, 99.90% compilation time)
```



tour =  3



Mémoïsation pour Puissance 4

Colonnes =  3

Hauteur =  3

Observations clés:

- Arbre de recherche:
 - **colonnes** = 3 sous-branches par noeuds
 - profondeur : **colonnes** × **hauteur** donc ...
 - ... $3^{\text{colonnes} \times \text{hauteur}}$ = 19 683 feuilles !
- Stratégie optimale dépend de la **grille** mais **pas du passé**
- Chaque case a 3 états possibles: ,  et 
- $3^{\text{colonnes} \times \text{hauteur}}$ = 19 683 grilles différentes...
- ... mais beaucoup de ces grilles sont invalides ou **pas explorées**

Quelle **structure de donnée** utiliser pour la mémoïsation : état du jeu → action optimale ?

Mémoïsation pour Puissance 2

Colonnes =  3

Hauteur =  3

```
mem_cols = ▶ [1, 1, 2]
```

```
1 mem_cols = @time game(mem_nrows, mem_ncols, mem_longueur, memoization = true)
```

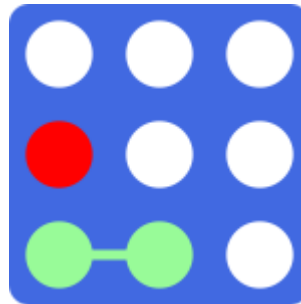


```
0.000044 seconds (18 allocations: 976 bytes)
```



Longueur =  2

tour =  3



Pièces de monnaie ⇔

Comme on a vu avec Fibonacci et le puissance 4, beaucoup de problème de calcul peuvent se représenter par un graphe où le but est de calculer la valeur d'un noeud du graphe et chaque arête (u, v) signifie que pour calculer la valeur du noeud u , il faut d'abord calculer la valeur du noeud v .

Considérons le problème de calculer de combien de manières différentes x_n , on sait faire une certaine somme avec des pièces de monnaie. Par exemple, pour faire 3 € avec les pièces de 1 € et 2 €, on peut soit utiliser 3 pièces de 1 € ou 1 pièce de 2 € et une pièce de 1 € donc il y a 2 possibilités.

Quel graphe peut-on utiliser pour modéliser ce problème ? On peut utiliser un noeud par somme en €, représentant la solution du problème pour cette somme là. Si on ne travaille pas avec les cents, et que avec les billets jusque 10 €, on a

$$x_n = x_{n-10} + x_{n-5} + x_{n-2} + x_{n-1}$$

On a donc les arêtes $(n, n-10)$, $(n, n-5)$, $(n, n-2)$ et $(n, n-1)$.

► Est-ce que cette formule donne le bon résultat ?

Considérons la valeur $y_{n,k}$ qui correspond à la façon de faire la somme n avec des billets/pièces de valeur max k .

► Quelle est la formule pour $y_{n,10}$? Quel est le graphe ? Est-ce que la bonne valeur est calculée ?

Exercice: Implémenter le calcul avec l'approche top-down et bottom-up. Laquelle est la plus rapide ? Pourquoi ?

Supplémentaire: On peut aussi utiliser la formule

$$y_{n,10} = y_{n-10,5} + y_{n-20,5} + \dots + y_{n-5,2} + y_{n-10,2} + \dots$$

Modifier l'implémentation pour utiliser cette formule. Est-ce plus rapide ? Pourquoi ?

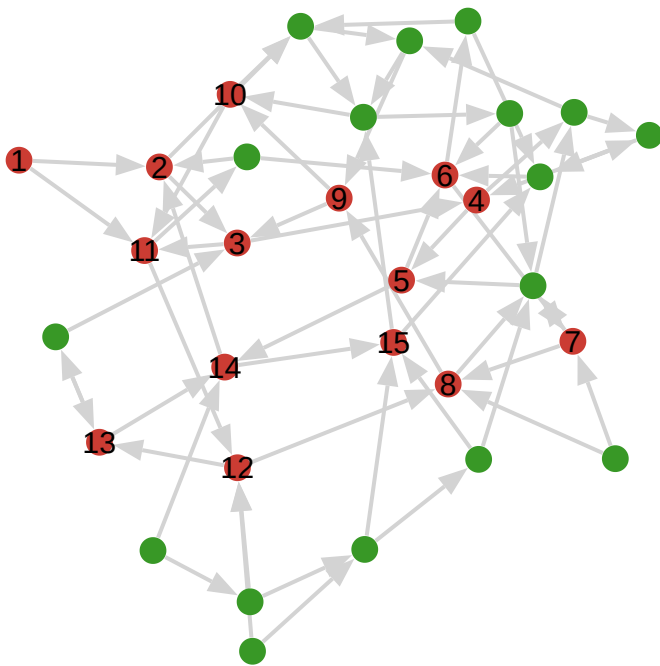
Depth-First Search (DFS) ⇄

dfs! (generic function with 1 method)

```
1 function dfs!(path, g::DiGraph, current, target)
2     if current in path
3         return false
4     end
5     push!(path, current)
6     if current == target
7         return true
8     end
9     for next in Graphs.outneighbors(g, current)
10         if dfs!(path, g, next, target)
11             return true
12         end
13     end
14     pop!(path)
15     return false
16 end
```

dfs (generic function with 1 method)

```
1 function dfs(g, current, target)
2     path = Int[]
3     dfs!(path, g, current, target)
4     return path
5 end
```



```

1 let
2   Random.seed!(32)
3   n = 32
4   g = random_regular_digraph(n, 2)
5   path = dfs(g, 1, n)
6   plot_path(g, path)
7 end

```

plot_path (generic function with 1 method)

```

1 function plot_path(g, path)
2   cols = [colors.red, colors.green]
3   gplot(g, nodefillc = [cols[v in path ? 1 : 2] for v in vertices(g)], nodelabel
4   = [something(findfirst(isequal(v), path), "") for v in vertices(g)])
5 end

```

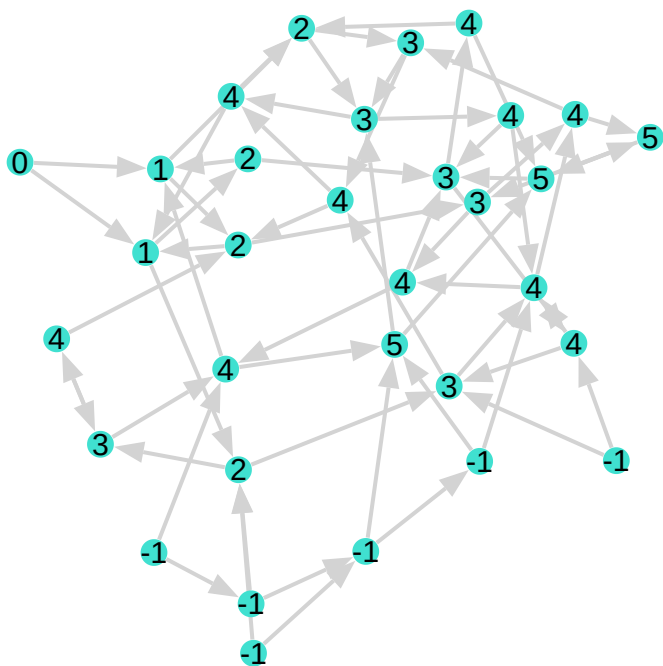
► Quelle est la complexité de cet algorithme ?

► Trouve-t-il le plus court chemin ?

Breadth-First Search (BFS) ⇄

L'algorithme BFS résout le problème de shortest path dans un graphe dirigé dont tous les poids sont 1.

```
bfs (generic function with 1 method)
1 function bfs(g::DiGraph, source)
2     dist = fill(-1, nv(g))
3     queue = Queue{Tuple{Int,Int}}{ }()
4     dist[source] = 0
5     enqueue!(queue, (source, 0))
6     while !isempty(queue)
7         current, d = dequeue!(queue)
8         for next in outneighbors(g, current)
9             if dist[next] == -1
10                 dist[next] = d + 1
11                 enqueue!(queue, (next, d + 1))
12             end
13         end
14     end
15     return dist
16 end
```



```

1  let
2      Random.seed!(32)
3      n = 32
4      g = random_regular_digraph(n, 2)
5      dist = bfs(g, 1)
6      gplot(g, nodelabel = string.(dist))
7  end

```

► Quelle est la complexité de l'algorithme BFS ?

Dijkstra ⇄

L'algorithme de Dijkstra résout le problème de shortest path dans un graph dirigé quand tous les poids sont positifs.

dijkstra (generic function with 1 method)

```
1 function dijkstra(g::SimpleWeightedGraph, source)
2     dist = fill(-1, nv(g))
3     dist[source] = 0
4     queue = PriorityQueue{Int,Int}{}
5     push!(queue, source => 0)
6     while !isempty(queue)
7         current = dequeue!(queue)
8         for next in outneighbors(g, current)
9             d = dist[current] + g[current, next, Val{:weight}]
10            if dist[next] == -1
11                dist[next] = d
12                enqueue!(queue, next => d)
13            elseif d < dist[next]
14                dist[next] = d
15                queue[next] = d
16            end
17        end
18    end
19    return dist
20 end
```


Max Flow / Min Cut

- Le problème de Max Flow détermine le flot le plus important d'un noeud source à un noeud target étant donné une capacité maximale de chaque arête.
- Le problème de Min Cut détermine la somme des capacité minimale d'une coupe du graphe séparant la source de la target.

Théorème Pour tout graphe, le flot maximal est égal à la cut minimale !

Utils

```
1 using PlutoUI, Graphs, GraphPlot, Colors, DataStructures, SimpleWeightedGraphs
```



```
Precompiling Graphs...
 6778.1 ms ✓ StaticArrays
  647.3 ms ✓ StaticArrays → StaticArraysStatisticsExt
1015.9 ms ✓ ArnoldiMethod
3680.2 ms ✓ Graphs
4 dependencies successfully precompiled in 12 seconds. 26 already precompiled.
Precompiling GraphPlot...
2264.8 ms ✓ Compose
1273.1 ms ✓ GraphPlot
2 dependencies successfully precompiled in 4 seconds. 40 already precompiled.
Precompiling SimpleWeightedGraphs...
 952.8 ms ✓ SimpleWeightedGraphs
1 dependency successfully precompiled in 1 seconds. 30 already precompiled.
```



```
1 import DocumenterCitations, Polyhedra, Makie, WGLMakie, StaticArrays, Bonito, Luxor
   , Formatting, Random
```

```
Precompiling DocumenterCitations...
 477.4 ms ✓ Expat_jll
 796.3 ms ✓ BibParser
 551.0 ms ✓ Git_jll
 428.0 ms ✓ Git
1349.9 ms ✓ Bibliography
26397.3 ms ✓ Documenter
 2636.0 ms ✓ DocumenterCitations
 7 dependencies successfully precompiled in 31 seconds. 61 already precompiled.
Precompiling Polyhedra...
 754.8 ms ✓ CodecBzip2
 893.6 ms ✓ EarCut_jll
 877.7 ms ✓ OpenSpecFun_jll
 923.5 ms ✓ StructArrays → StructArraysStaticArraysExt
1039.5 ms ✓ ConstructionBase → ConstructionBaseStaticArraysExt
1545.7 ms ✓ BenchmarkTools
2354.0 ms ✓ SpecialFunctions
5465.9 ms ✓ GeometryBasics
3604.1 ms ✓ ForwardDiff
 687.7 ms ✓ ForwardDiff → ForwardDiffStaticArraysExt
84435.1 ms ✓ MathOptInterface
18372.9 ms ✓ JuMP
 4207.6 ms ✓ Polyhedra
13 dependencies successfully precompiled in 114 seconds. 74 already precompiled.
Precompiling Makie...
 752.4 ms ✓ Rmath_jll
 761.1 ms ✓ Lmath_jll
```

qa (generic function with 2 methods)

```
1 include("utils.jl")
```

```
1 begin
2     slider_n = @bind n Slider(1:42, default = 10, show_value = true);
3     slider_k = @bind k Slider(1:42, default = 10, show_value = true);
4     nothing
5 end
```

draw_fib (generic function with 1 method)

```
1 function draw_fib(n, depth)
2     if n < 3
3         return 0
4     end
5     dy = 140
6     δ = round(Int, dy * 0.9^depth) * 1.2
7     w = fib(n - 2)
8     Δy = 30
9     if depth != 1
10         Luxor.translate(δ * w/2, 0)
11     end
12     if n == 3
13         Luxor.setdash("dashed")
14         Luxor.circle(Luxor.Point(6, -6), 20, :stroke)
15     end
16     Luxor.sethue("black")
17     Luxor.text(string(n))
18     Luxor.translate(-δ * w/2, 0)
19     cur = 0
20     for m in [n - 1, n - 2]
21         if m < 3
22             continue
23         end
24         dx = cur * δ
25         Luxor.translate(dx, dy)
26         dw = draw_fib(m, depth + 1)
27         Luxor.sethue("black")
28         Luxor.setdash("solid")
29         Luxor.setline(δ/40)
30         Luxor.line(Luxor.Point(-dx+δ * w/2, -dy+Δy), Luxor.Point(δ*dw/2, -Δy),
31                     :stroke)
32         Luxor.translate(-dx, -dy)
33         cur += dw
34     end
35     return w
36 end
```

draw_fibo (generic function with 1 method)

```
1 function draw_fibo(n)
2     Luxor.@draw begin
3         Luxor.translate(0, -350)
4         Luxor.fontsize(30)
5         Luxor.fontface("Alegreya Sans SC")
6         draw_fib(n, 1)
7     end 1200 800
8 end
```

random_weighted (generic function with 1 method)

```
1 function random_weighted(n, k, weights)
2     g = random_regular_graph(n, k)
3     wg = SimpleWeightedGraph(n)
4     for e in edges(g)
5         add_edge!(wg, src(e), dst(e), rand(weights))
6     end
7     return wg
8 end
```

colors = ▶ (red = , green = , blue = , purple = )

```
1 colors = Colors.JULIA_LOGO_COLORS
```

biblio =

▶ CitationBibliography("/home/runner/work/LSINC1113/LSINC1113/Lectures/biblio.bib", AlphaSt

```
1 biblio = load_biblio!()
```

① Loading bibliography from `/home/runner/work/LSINC1113/LSINC1113/Lectures/biblio.bib`...

⊗ Entry west2022Introduction is missing the publisher field(s).

① Loading completed.

polyhedron =

Polyhedron Polyhedra.DefaultPolyhedron{Float64, Polyhedra.MixedMathRep{Float64, Matrix{Float64, 24-element iterator of Vector{Float64}}:

```
[0.0, 1.0, 2.0, 3.0]
[0.0, 2.0, 1.0, 3.0]
[1.0, 0.0, 2.0, 3.0]
[1.0, 2.0, 0.0, 3.0]
[2.0, 0.0, 1.0, 3.0]
[2.0, 1.0, 0.0, 3.0]
[0.0, 1.0, 3.0, 2.0]
[0.0, 3.0, 1.0, 2.0]
[1.0, 0.0, 3.0, 2.0]
[1.0, 3.0, 0.0, 2.0]
[3.0, 0.0, 1.0, 2.0]
[3.0, 1.0, 0.0, 2.0]
[0.0, 3.0, 2.0, 1.0]
[0.0, 2.0, 3.0, 1.0]
:
```

```
1 polyhedron = Polyhedra.polyhedron(Polyhedra.vrep(Float64[
2     0 1 2 3; 0 2 1 3; 1 0 2 3; 1 2 0 3; 2 0 1 3; 2 1 0 3;
3     0 1 3 2; 0 3 1 2; 1 0 3 2; 1 3 0 2; 3 0 1 2; 3 1 0 2;
4     0 3 2 1; 0 2 3 1; 3 0 2 1; 3 2 0 1; 2 0 3 1; 2 3 0 1;
5     3 1 2 0; 3 2 1 0; 1 3 2 0; 1 2 3 0; 2 3 1 0; 2 1 3 0
6 ]))
```

projected =

Polyhedron Polyhedra.DefaultPolyhedron{Float64, Polyhedra.MixedMathRep{Float64, Matrix{Float64, 24-element iterator of Vector{Float64}}:

```
[-0.7071067811865501, -1.2247448713915914, -1.7320508075688754]
[0.7071067811865492, -1.2247448713915912, -1.7320508075688754]
[-1.4142135623730974, -5.10702591327572e-15, -1.7320508075688759]
[-0.7071067811865478, 1.224744871391588, -1.7320508075688759]
[1.4142135623730963, -4.3298697960381105e-15, -1.7320508075688756]
[0.7071067811865472, 1.2247448713915883, -1.7320508075688756]
[-0.7071067811865465, -2.0412414523193156, -0.5773502691896244]
[-2.121320343559643, 0.40824829046386146, -0.5773502691896253]
[0.7071067811865529, -2.0412414523193156, -0.5773502691896242]
[2.121320343559643, 0.40824829046385874, -0.5773502691896244]
[-1.4142135623730931, 1.6329931618554543, -0.5773502691896262]
[1.4142135623730934, 1.6329931618554512, -0.5773502691896251]
[2.1213203435596464, -0.40824829046386274, 0.5773502691896222]
[0.7071067811865501, 2.0412414523193165, 0.5773502691896217]
:
```

```
1 projected = Polyhedra.project(polyhedron, [1 1 1; -1 1 1; 0 -2 1; 0 0 -3])
```

```
1 struct Config{N}
2     libre::StaticArrays.SVector{N, UInt8}
3     player::StaticArrays.SVector{N, UInt8}
4     max::UInt8
5 end
```

_copy (generic function with 3 methods)

```
1 begin
2   _num_rows(mat::AbstractMatrix) = size(mat, 1)
3   _num_rows(config::Config) = _shift_bit(config.max) - 1
4   _reverse(config::Config) = Config(reverse(config.libre), reverse(config.player),
   config.max)
5   Base.getindex(c::Config, i::UInt8, j::Int) = (!iszero(c.libre[j] & i),
   !iszero(c.player[j] & i))
6   _compare(player::Int, a, b) = player * a > player * b
7   _compare(player::Bool, a, b) = player ? a < b : a > b
8   _sign(player::Int, a) = player * a
9   _sign(w::Tuple{Bool,Bool}, a) = w[2] ? -a : a
10  _first_player(::AbstractMatrix) = 1
11  _first_player(::Config) = false
12  _next_player(player::Int) = -player
13  _next_player(player::Bool) = !player
14  _columns(m::AbstractMatrix) = axes(m, 2)
15  _columns(config::Config) = eachindex(config.libre)
16  function _next(mat::AbstractMatrix, col)
17      for r in axes(config, 1)
18          if iszero(config[r, col])
19              return r
20          end
21      end
22      return 0
23  end
24  function _next(config::Config, col)
25      mask = config.libre[col] + 0b1
26      if mask == config.max
27          return 0b0
28      end
29      return mask
30  end
31  function _assign(mat::StaticArrays.SMatrix{I,J}, r, c, value) where {I,J}
32      target = r + (c - 1) * I
33      return StaticArrays.SMatrix{I,J}(ntuple(i -> i == target ? value : mat[i],
   Val(I * J)))
34      #return @SMatrix [(row, col) == (r, c) ? value : mat[row, col] for row in
   1:nrows, col in 1:ncols]
35  end
36  function _assign(v::StaticArrays.SVector{N}, mask, col) where {N}
37      StaticArrays.SVector{N}(ntuple(c -> c == col ? (v[c] | mask) : v[c], Val(N)))
38  end
39  function _assign(config::Config, mask, col, value::Bool)
40      libre = _assign(config.libre, mask, col)
41      if value
42          player = _assign(config.player, mask, col)
43      else
44          player = config.player
```

```

45     end
46     return Config(libre, player, config.max)
47 end
48 function _assign(mat::Matrix, row, col, value)
49     mat[row, col] = value
50     return mat
51 end
52 _unassign(::StaticArrays.SMatrix, _, _) = nothing
53 _unassign(::Config, _, _) = nothing
54 function _unassign(mat::Matrix, row, col)
55     mat[row, col] = 0
56     return
57 end
58 _copy(m::StaticArrays.SMatrix) = m
59 _copy(m::Config) = m
60 _copy(m::Matrix) = copy(m)
61 end

```

_has_winner (generic function with 2 methods)

```

1 begin
2 _no_winner(::AbstractMatrix) = 0
3 _no_winner(::Config) = (false, false)
4 _has_winner(x::Int) = !iszero(x)
5 _has_winner(x::Tuple{Bool,Bool}) = x[1]
6 end

```

```
const Δs = ▶ [(-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
```

```
1 const Δs = [(-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
```

winner (generic function with 1 method)

```

1 function winner(config, longueur, pos)
2     for Δ in Δs
3         win = winner(config, pos, Δ, longueur)
4         if _has_winner(win)
5             return win, Δ
6         end
7     end
8     return _no_winner(config), (0, 0)
9 end

```

winner (generic function with 2 methods)

```
1 function winner(config, longueur)
2   for col in _columns(config)
3     row = _first_row(config)
4     while true
5       pos = (row, col)
6       win, Δ = winner(config, longueur, pos)
7       if _has_winner(win)
8         return win, pos, Δ
9       end
10      row = _next_row(row)
11      if _done(config, row)
12        break
13      end
14    end
15  end
16  return _no_winner(config), (0, 0), (0, 0)
17 end
```

_done (generic function with 2 methods)

```
1 begin
2   _first_row(::AbstractMatrix) = 1
3   _next_row(row::Int) = row + 1
4   _done(m::AbstractMatrix, row) = row > size(m, 1)
5   _first_row(::Config) = 0b1
6   _next_row(row::UInt8) = row << 1
7   _done(c::Config, row) = row == c.max
8 end
```

winner (generic function with 3 methods)

```
1 function winner(config, x, Δ::Tuple{Int,Int}, longueur)
2   if !_inrows(config, x[1], (longueur::Int - 1) * Δ[1]) ||
3     !in(x[2] + (longueur::Int - 1) * Δ[2], _columns(config))
4     #if !all(in.(x .+ (longueur - 1) .* Δ, axes(config)))
5       return _no_winner(config)
6     end
7     cur = config[x[1], x[2]]
8     if iszero(cur[1])
9       return _no_winner(config)
10    end
11    for i in 1:(longueur::Int - 1)
12      if config[_shift(x[1], i * Δ[1]), _shift(x[2], i * Δ[2])] != cur
13        #if config[(x .+ i .* Δ)...] != cur
14          return _no_winner(config)
15        end
16      end
17    end
18  end
```

_inrows (generic function with 2 methods)

```
1 begin
2   _shift(x::Int, y::Int) = x + y
3   _shift(x::UInt8, y::Int) = x << y
4   _inrows(m::AbstractMatrix, x::Int, y::Int) = in(x + y, axes(m, 1))
5   function _inrows(config::Config, x::UInt8, y::Int)
6       if y == 0
7           return true
8       elseif y < 0
9           return x >= (0b1 << (UInt8(-y)))
10      else
11          return (config.max >> UInt8(y)) > x
12      end
13  end
14  end
```

play! (generic function with 1 method)

```
1 function play!(config, col::Int, player)
2     row = _next(config, col)
3     #row = @time findfirst(row -> iszero(config[row, col]), axes(config, 1))
4     #if !isnothing(row)
5     if !iszero(row)
6         config = _assign(config, row, col, player)
7     end
8     return row, config
9 end
```

`_draw` (generic function with 2 methods)

```
1 function _draw(config, longueur, δ = 50)
2   colors = Dict(
3     (false, false) => "white", (true, true) => "red", (true, false) =>
4     "palegreen",
5     0 => "white", 1 => "palegreen", -1 => "red",
6   )
7   sz = (length(_columns(config)), _num_rows(config))
8   c = (sz .+ 1) ./ 2 .* δ
9   screen = sz .* δ
10  pos(row, col) = Luxor.Point(δ * col, δ * (sz[2] - _shift_bit(row) + 1))
11  Luxor.translate(-Luxor.Point(c...))
12  Luxor.sethue("royalblue")
13  Luxor.polysmooth(Luxor.box(Luxor.Point(c...), screen..., vertices=true), δ *
14  0.2, :fill)
15  for col in _columns(config)
16    row = _first_row(config)
17    while true
18      Luxor.sethue(colors[config[row, col]])
19      Luxor.circle(pos(row, col), δ / 3, :fill)
20      row = _next_row(row)
21      if _done(config, row)
22        break
23      end
24    end
25  end
26  win, x, Δ = winner(config, longueur)
27  if _has_winner(win)
28    Luxor.sethue(colors[win])
29    #polysmooth(Luxor.box(pos((x .+ Δ .* (longueur - 1) ./ 2)...), reverse((Δ
30    .* δ .* (longueur - 0.4)) .+ δ/5)...), 4, :fill)
31    Luxor.setline(δ/10)
32    Luxor.line(pos(x...), pos((_shift_bit.(x) .+ Δ .* (longueur-1))...),
33    :stroke)
34  end
35  Luxor.translate(Luxor.Point(c...))
36 end
```

`_shift_bit` (generic function with 2 methods)

```
1 begin
2   _shift_bit(x::Int) = x
3   function _shift_bit(x::UInt8)
4     y = findfirst(i -> (0b1 << (i-1)) == x, 1:7)
5     return y
6   end
7 end
```

best_move_rec (generic function with 1 method)

```
1 function best_move_rec(config, player, longueur, depth, cache; patient, maxturn)
2   if depth > maxturn + 1
3     error("$depth > $maxturn + 1")
4   end
5   if cache != nothing
6     if haskey(cache, config)
7       return convert(Tuple{Int,Int}, cache[config])
8     end
9     reversed = _reverse(config)
10    if haskey(cache, reversed)
11      return convert(Tuple{Int,Int}, cache[reversed])
12    end
13  end
14  config = _copy(config)
15  best = 0
16  best_col = 0
17  for col in _columns(config)
18    row, new_config = play!(config, col, player)
19    if !iszero(row)
20      win, _ = winner(new_config, longueur, (row, col))
21      if _has_winner(win)
22        best = _sign(win, 2maxturn - depth)
23        best_col = col
24      end
25    end
26  end
27  if iszero(best)
28    for col in _columns(config)
29      row, new_config = play!(config, col, player)
30      if !iszero(row)
31        cur, _ = best_move_rec(new_config, _next_player(player), longueur,
32          depth + 1, cache; patient, maxturn)
33        if best_col == 0 || _compare(player, cur, best)
34          best = cur
35          best_col = col
36          if patient && _compare(player, best, 0)
37            break
38          end
39        end
40        _unassign(config, row, col)
41      end
42    end
43  end
44  if cache != nothing
45    cache[config] = (best, best_col)
46  end
47  return best, best_col
48 end
```

game (generic function with 1 method)

```
1 game(nrows, ncols, longueur; memoization = false, patient = true) = game!  
  (_init(nrows, ncols), Int[], longueur, memoization; patient, maxturn = nrows *  
  ncols)
```

game! (generic function with 1 method)

```
1 function game!(config, cols, longueur, memoization; patient, maxturn)  
2   if memoization  
3     cache = Dict{typeof(config), Tuple{Int8, Int8}}{ }()  
4   else  
5     cache = nothing  
6   end  
7   player = _first_player(config)  
8   for i in 1:maxturn  
9     _, col = best_move_rec(config, player, longueur, 1, cache; patient, maxturn)  
10    if col == 0  
11      break  
12    end  
13    push!(cols, col)  
14    _, config = play!(config, col, player)  
15    w = winner(config, longueur)  
16    if _has_winner(w[1])  
17      break  
18    end  
19    player = _next_player(player)  
20  end  
21  return cols  
22 end
```

draw (generic function with 2 methods)

```
1 function draw(config, longueur, δ = 50)  
2   sz = (length(_columns(config)), _num_rows(config))  
3   screen = sz .* δ  
4   Luxor.@draw begin  
5     _draw(config, longueur, δ)  
6   end screen[1] screen[2]  
7 end
```

play (generic function with 1 method)

```
1 function play(nrows, ncols, cols)  
2   config = _init(nrows, ncols)  
3   player = _first_player(config)  
4   for col in cols  
5     _, config = play!(config, col, player)  
6     player = _next_player(player)  
7   end  
8   return config  
9 end
```

`_init` (generic function with 1 method)

```
1 function _init(nrows, ncols)
2     z = StaticArrays.SVector{ncols}(ntuple(_ -> zero(UInt8), Val(ncols)))
3     Config(z, z, (0b1 << nrows))
4 end
```

`minimax` (generic function with 1 method)

```
1 function minimax(nrows::Integer, ncols, longueur; vert = -200, w = 1200, h = 600,
dy = 140, damping = 0.9, Δy = 55, scaling)
2     config = _init(nrows, ncols)
3     Luxor.@draw minimax(config, _first_player(config), longueur, 1; dy, damping,
vert, Δy, scaling) w h
4 end
```

```
1 function width(config, player, longueur, damping, depth)
2     if _has_winner(winner(config, longueur)[1])
3         return damping^depth
4     end
5     w = 0
6     for col in _columns(config)
7         row, new_config = play!(config, col, player)
8         if !iszero(row)
9             w += width(new_config, _next_player(player), longueur, damping, depth +
1)
10         end
11     end
12     return max(w, 1)
13 end;
```

```

1 function minimax(config, player, longueur, depth; dy, damping, vert, Δy, scaling)
2     δ = round(Int, dy * damping^depth) * scaling
3     w = width(config, player, longueur, damping, depth)
4     if all(iszero(config.libre))
5         Luxor.translate(0, vert)
6     else
7         Luxor.translate(δ * w/2, 0)
8     end
9     _draw(config, longueur)
10    Luxor.translate(-δ * w/2, 0)
11    if _has_winner(winner(config, longueur)[1])
12        return w
13    end
14    cur = 0
15    for col in _columns(config)
16        row, new_config = play!(config, col, player)
17        if !iszero(row)
18            dx = cur * δ
19            Luxor.translate(dx, dy)
20            win, _, _ = winner(new_config, longueur)
21            score, _ = best_move_rec(new_config, _next_player(player), longueur,
22            depth + 1, nothing, patient = true, maxturn = 30)
23            dw = minimax(new_config, _next_player(player), longueur, depth + 1; dy,
24            damping, vert, Δy, scaling)
25            if _has_winner(win)
26                if player
27                    Luxor.sethue("red")
28                else
29                    Luxor.sethue("palegreen")
30                end
31            elseif score == 0
32                Luxor.sethue("black")
33            elseif score > 0
34                Luxor.sethue("palegreen")
35            else
36                Luxor.sethue("red")
37            end
38            Luxor.setline(δ/20)
39            Luxor.line(Luxor.Point(-dx+δ * w/2, -dy+Δy), Luxor.Point(δ*dw/2, -Δy),
40            :stroke)
41            Luxor.translate(-dx, -dy)
42            cur += dw
43        end
44    end
45    return w
46 end;

```

`mesh =`

► `Mesh(Polyhedron Polyhedra.DefaultPolyhedron{Float64, Polyhedra.MixedMathRep{Float64, Matrix{Float64}}, 24-element iterator of Vector{Float64}}:`

```
1 mesh = Polyhedra.Mesh(projected)
```

`cite` (generic function with 1 method)

```
1 cite(args...) = bibcite(biblio, args...)
```

`refs` (generic function with 1 method)

```
1 refs(args...) = bibrefs(biblio, args...)
```