LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

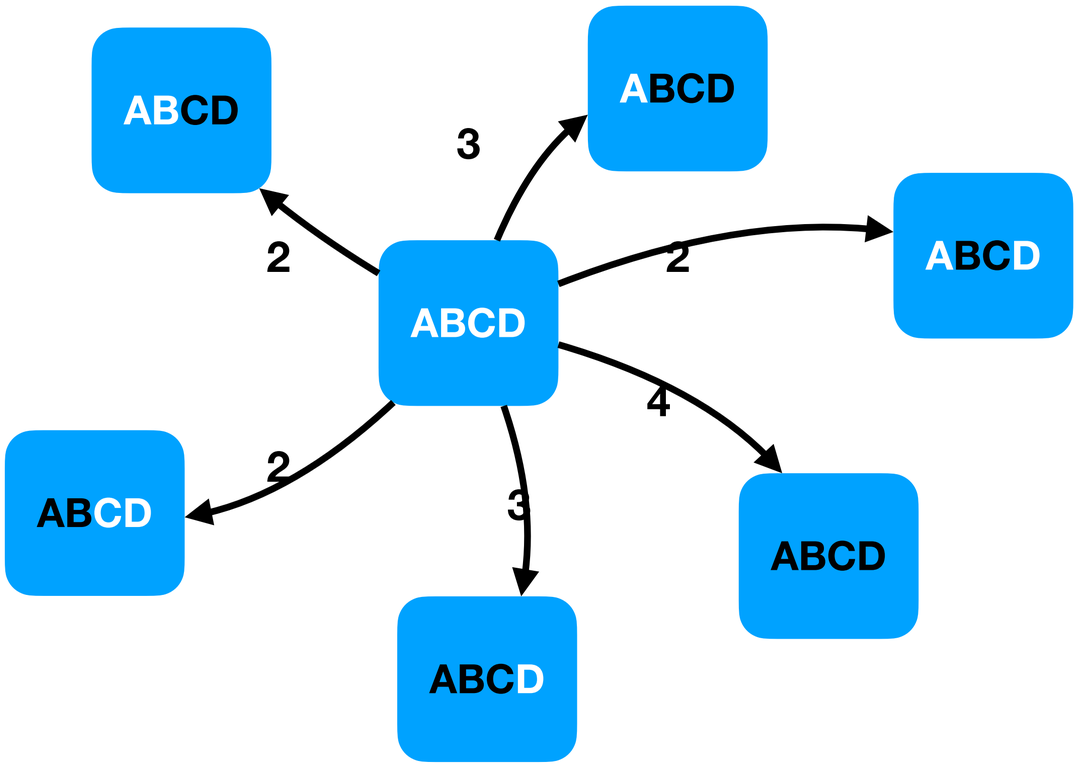
Graphs

Pierre Schaus

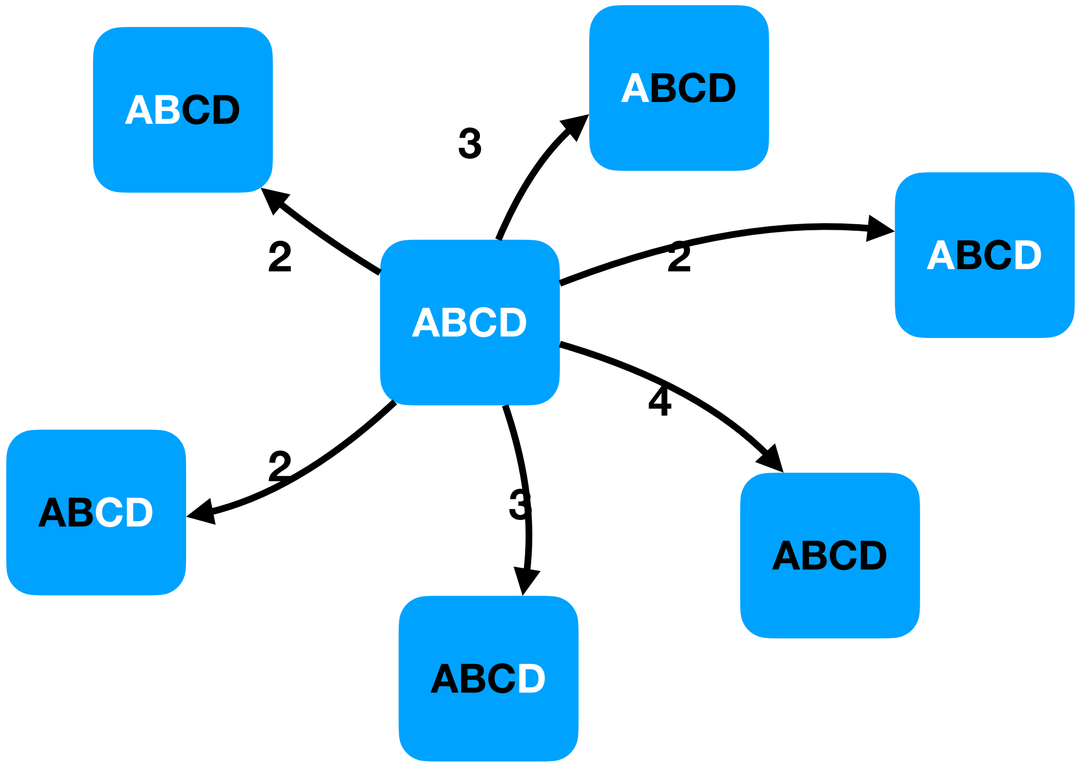
String problem

rotation(HAMBURGER, 4, 8)= HAMBEGRUR

Chaque String est un noeud possible dans un graphe. Pour un string de taille n, il y a donc n! noeuds possibles.



- Comment représenter ce graphe ? Structure de donnée ?



- Combien de voisins ? (n-1+n-2…+1)=n*(n-1)/2

Deux possibilités

- Graph explicite
  - Le graphe complet est construit à priori
- Graph implicite
  - Le graph est “lazy”, les voisins sont recalculés à chaque fois dynamiquement

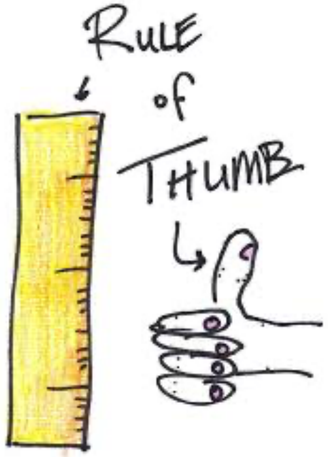
Implementation

- Construire une structure de graphe explicite est difficile car
  - Il faut a priori décider de V, construire tous les edges, etc.
  - Chaque noeud doit avoir un mapping vers un entier entre 0 et |V-1| (hash map ?)

- Nous optons donc pour une structure de graphe implicite.
  - Les String dans une hashmap sont les noeuds (on va les ajouter de manière “lazy”).
  - Les Edges ne sont jamais stockées mais recalculées grâce à la méthode **rotation**.

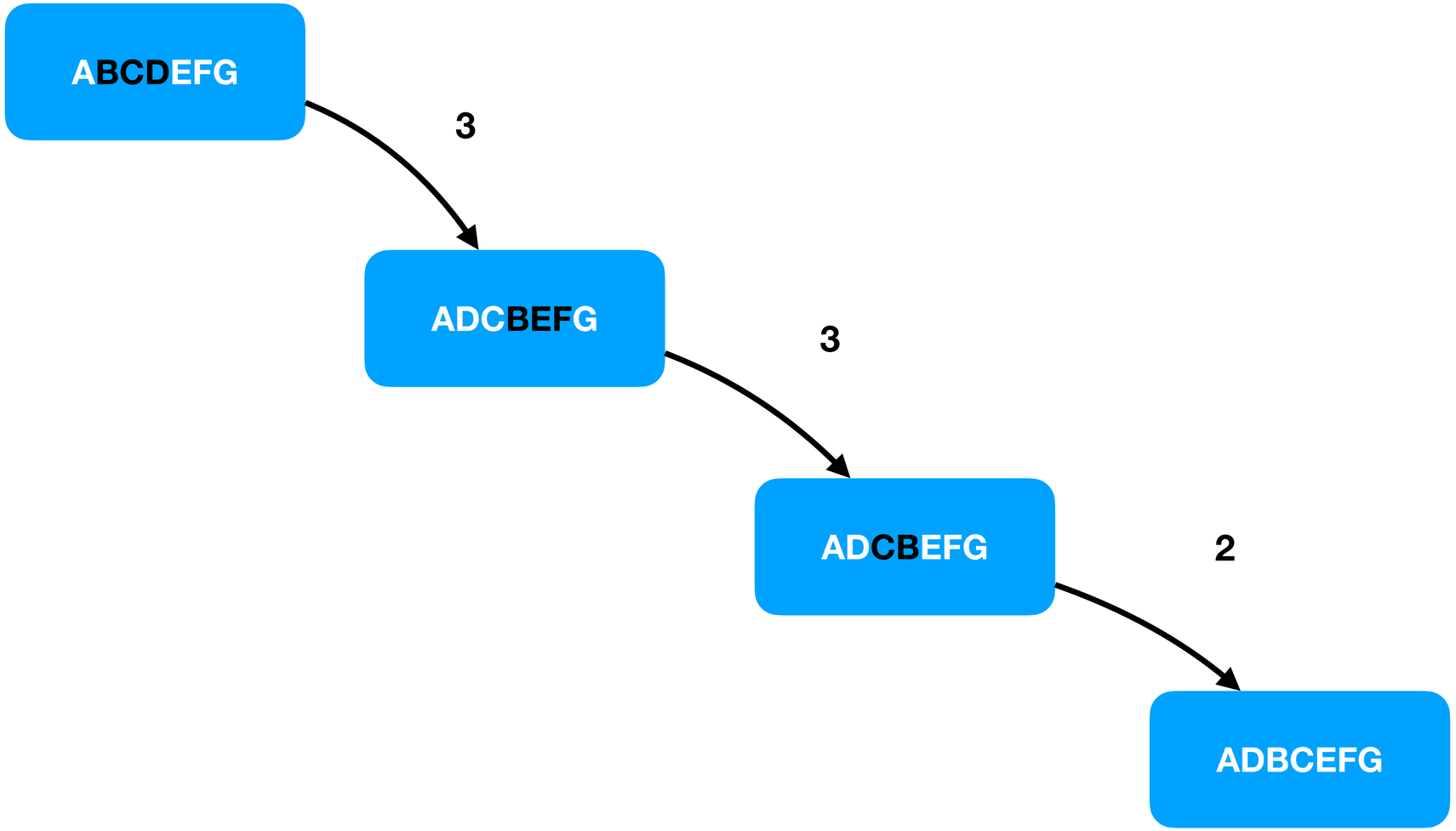
Comment choisir

- La structure implicite est préconisée lorsque
  - générer les edges sortant d'un noeud v se fait en O(degré(v)) (car elle ne prend pas de mémoire et n'induit pas d'overhead de calcul).
  - il n'est pas trivial de faire le mapping entre les identifiants des objets modélisés par les noeuds.
  - Construire le graph explicite risque de prendre plus de temps l'algorithme directement sur la structure implicite (est-ce que l'algorithme v les edges et tous les noeuds ?)



Quel algorithme pour trouver le plus court chemin ?

Chemin **ABCDEFG vers ADBCEFG ?**



Dijkstra

```
public class WordTransformationSP {

    public static String rotation(String s, int start, int end) {
        . . .
    }

    static class Entry implements Comparable<Entry> {
        String value;
        int dist;
        Entry(String content, int value) {
            this.value = content;
            this.dist = value;
        }
        public int compareTo(Entry o) {
            return this.dist - o.dist;
        }
    }

    . . .
}
```

Such that you can sort the string according to an integer value (distance)

Natural order on integers


```java
public static int minimalCost(String from, String to) {
    HashMap<String, Integer> distTo = new HashMap<>();
    PriorityQueue<Entry> PQ = new PriorityQueue<>();
    PQ.add(new Entry(from, 0));
    distTo.put(from, 0);
    while (!PQ.isEmpty()) {
        Entry n = PQ.poll();
        String v = n.value;
        for (int i = 0; i < v.length() - 1; i++) {
            for (int j = i + 2; j <= v.length(); j++) {
                String w = rotation(v, i, j);
                if (!distTo.containsKey(w) ||
                    distTo.get(w) > distTo.get(v) + (j - i)) {
                    distTo.put(w, distTo.get(v) + (j - i));
                    PQ.add(new Entry(w, distTo.get(v) + (j - i)));
                }
            }
        }
    }
    return distTo.get(to);
}
```

Les noeuds de mon graphe + la distance à ceux-ci

Edge relaxation

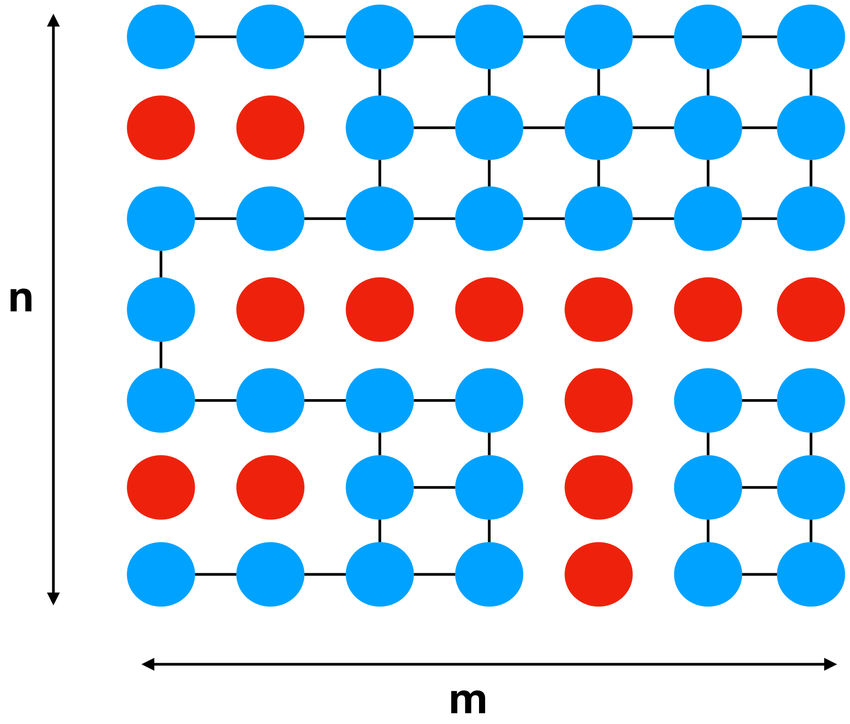
Génération des $O(n^2)$ successeurs

J'utilise la PQ non adaptable de Java. Comme nous avons vu, cela ne change pas la complexité Big-Oh

Maze Problem

```
public int [][] maze = new int[][] {
        //  --y-->
        {0,0,0,0,0,0,0}, // |
        {1,1,0,0,0,0,0}, // x
        {0,0,0,0,0,1,0}, // |
        {0,1,1,1,1,1,1}, // V
        {0,0,0,0,1,0,0},
        {1,1,0,0,1,0,0},
        {0,0,0,0,1,0,0}
};
```

origin: x1=0,y1=1
destination: x2=2,y2=1




```
import java.util.LinkedList;
public class Maze {
  public static Iterable<Integer> shortestPath(int [][] maze, int x1, int y1, int x2, int y2) {
      // TODO
      return new LinkedList<>();
  }
}
```

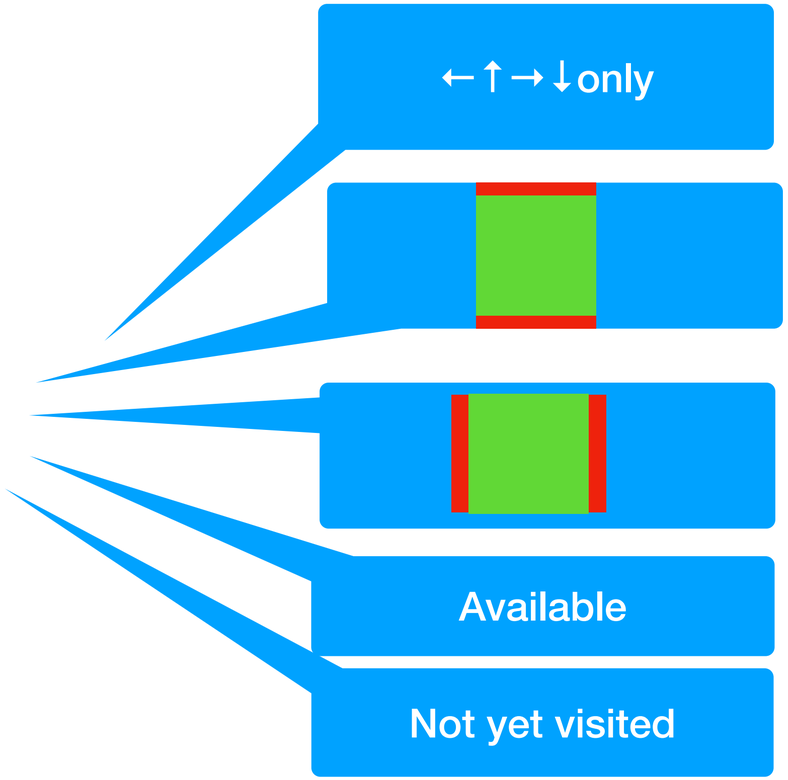
Identifiant unique pour x,y => id= x*m+y
On peut retrouver x = id/m et y = id%m

Solution = BFS

```
public static Iterable<Integer> shortestPath(int [][] maze,  int x1, int y1, int x2, int y2) {

    int n = maze.length;
    int m = maze[0].length;
    boolean [][] marked = new boolean[n][m];
    int [][] prev = new int[n][m];
    Queue<Integer> open = new ArrayDeque<>();
    if (maze[x1][y1] != 1) {
        open.add(m*x1+y1);
        marked[x1][y1] = true;
    }
    while (!open.isEmpty()) {
        int p = open.remove();
        int x = row(p, m);
        int y = col(p, m);
        if (x == x2 && y == y2) break;

        for (int dx : new int[]{-1, 0, 1}) {
            for (int dy : new int[]{-1, 0, 1}) {
                if ((Math.abs(dx) + Math.abs(dy)) == 1 &&
                        x + dx < n && x + dx >= 0 &&
                        y + dy < m && y + dy >= 0 &&
                        maze[x + dx][y + dy] == 0 &&
                        !marked[x + dx][y + dy]) {
                    open.add((x + dx) * m + (y + dy));
                    prev[x + dx][y + dy] = p;
                    marked[x + dx][y + dy] = true;
                }
            }
        }
    }
    if (!marked[x2][y2]) return new LinkedList<>();
    return extractPath(x1,y1,x2,y2,prev);
}
```



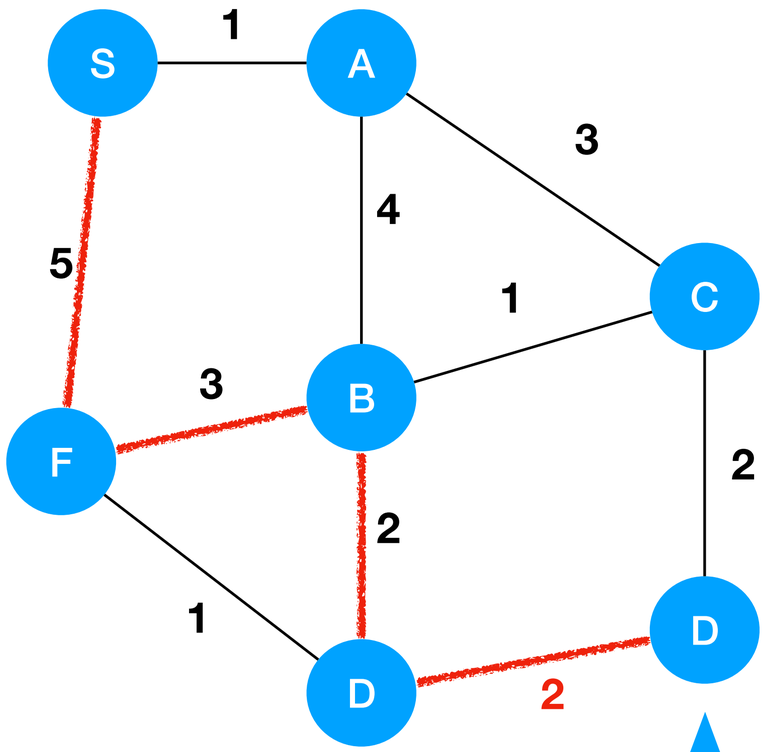
```java
public static Iterable<Integer> extractPath(int x1, int y1, int x2, int y2, int [][] prev) {
    int m = prev[0].length;
    LinkedList<Integer> path = new LinkedList<>();
    int x = x2;
    int y = y2;
    path.addFirst(x2*m+y2);
    while (x != x1 || y != y1) {
        int p = prev[x][y];
        x = row(p,m);
        y = col(p,m);
        path.addFirst(p);
    }
    return path;
}
```

J’utilise une LinkedList car l’itérateur sur un Stack de Java est FIFO :-((erreur de design, Stack extends Vector)

J’ajoute en tête pour avoir un iterator LIFO

Maximisation du plus petit poids sur le chemin

Soit un graphe G dirigé et pondéré positivement. Etant donné une origine S, on veut trouver un chemin vers chaque noeud qui maximise le poids de l'arête minimum sur ce chemin (bottleneck-distance).



Bdistance(S,D) =min(5,3,2,2)=2

Application:
Maximisation de la bande passante dans un réseau télécom

Problème de chemin nous fait penser à Dijkstra

- Rappel: Dijkstra ne fonctionne pas pour trouver le plus long chemin car dès l'instant où il y a un cycle, on peut allonger le chemin.

- Ici, ça n'est pas la même chose, la distance se calcule comme le minimum, donc il n'y a pas d'intérêt à circuler le long d'un cycle. Intuitivement, un algorithme type Dijkstra pourrait fonctionner, mais il faut le prouver …

1930-2002

Cherchons des conditions d'optimalité (adaptation p650)

- Soit bdist[v] la bottleneck-distance d'un chemin de S vers v. Ces valeurs correspondent à maximum **SSI**

$$\forall\ e=(v,w) : bdist[w] \geqslant \min(bdist[v], e.weight)$$

- Nécessaire: Optimalité => Condition, Facile de voir que si la condition n'est pas satisfaite, on peut améliorer la solution.

- Suffisante: Condition => Optimalité. Soit un noeud w et un chemin optimum vers ce noeuds s=v0,v1,v2,…,vk=w dont la b-distance est OPT_{sw}. Le long de ce chemin j'ai bien

$$dist[w] = dist[v_k] \geq \min(dist[v_{k-1}], e_k\,.\,weight)$$

$$dist[v_{k-1}] \geq \min(dist[v_{k-2}], e_{k-1}\,.\,weight)$$

$$\dots$$

$$dist[v_2] \geq \min(dist[v_1], e_2\,.\,weight)$$

$$dist[v_1] \geq \min(dist[s], e_1\,.\,weight)$$

$$dist[w] = dist[v_k] \geq \min(dist[v_0], e_1\,.\,weight, e_2\,.\,weight, \dots, e_k\,.\,weight)$$

Donc = OPT_{sw}

∞

OPT_{sw}

Adaptation Dijkstra

```
public DijkstraBandWidth(EdgeWeightedDigraph G, int s) {
    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];
    for (int v = 0; v < G.V(); v++)
        distTo[v] = -Double.POSITIVE_INFINITY;
    distTo[s] = Double.MaxDouble;
    // relax vertices in order of distance from s
    pq = new IndexMaxPQ<Double>(G.V());
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMax();
        for (DirectedEdge e : G.adj(v))
            relax(e);
    }
}
// relax edge e and update pq if changed
private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] < min (distTo[v] , e.weight())) {
        distTo[w] = min (distTo[v] , e.weight());
        edgeTo[w] = e;
        if (pq.contains(w)) pq.increaseKey(w, distTo[w]);
        else                pq.insert(w, distTo[w]);
    }
}
```

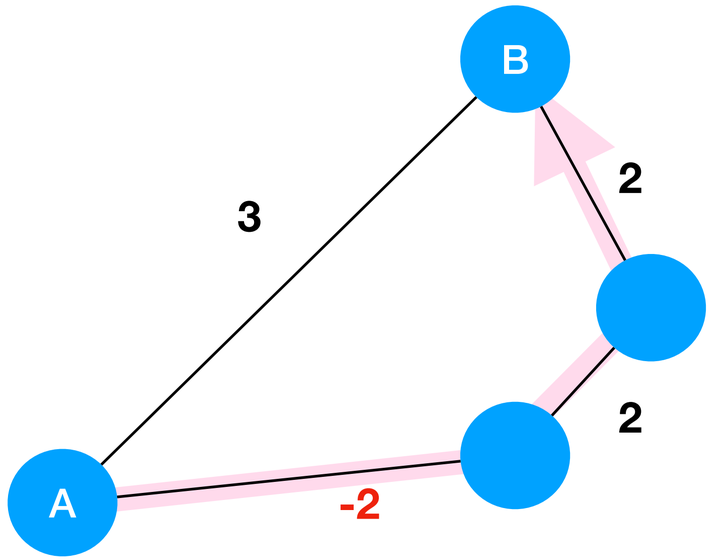
One more thing …

- Est-ce que cet algorithme fonctionne pour:
  - Les graphes avec des cycles négatifs ?
  - Les graphes sans cycle négatifs mais avec des poids négatifs ?

Réponse

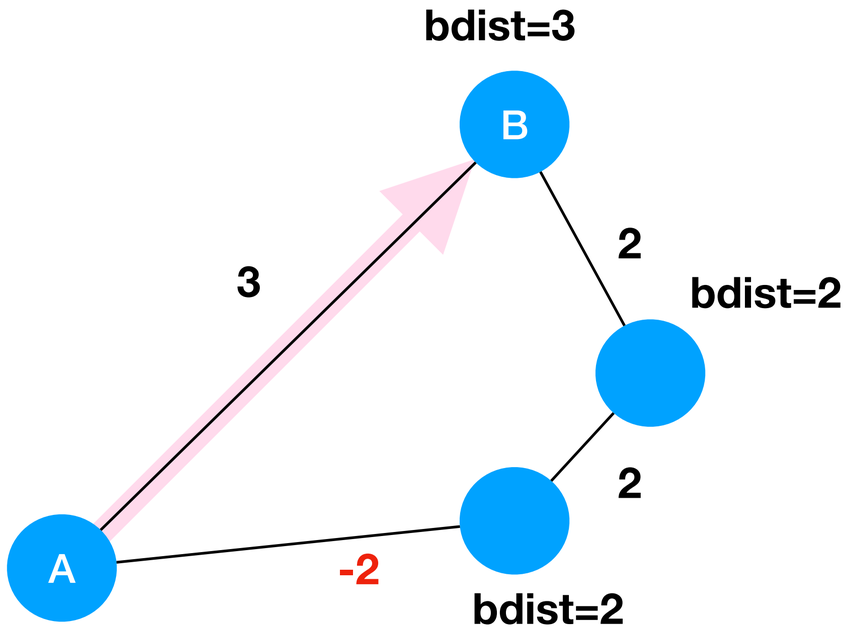
- Oui! Ça fonctionne, pour s'en convaincre, il faut se souvenir pourquoi ça ne fonctionne pas pour le plus court chemin classique.

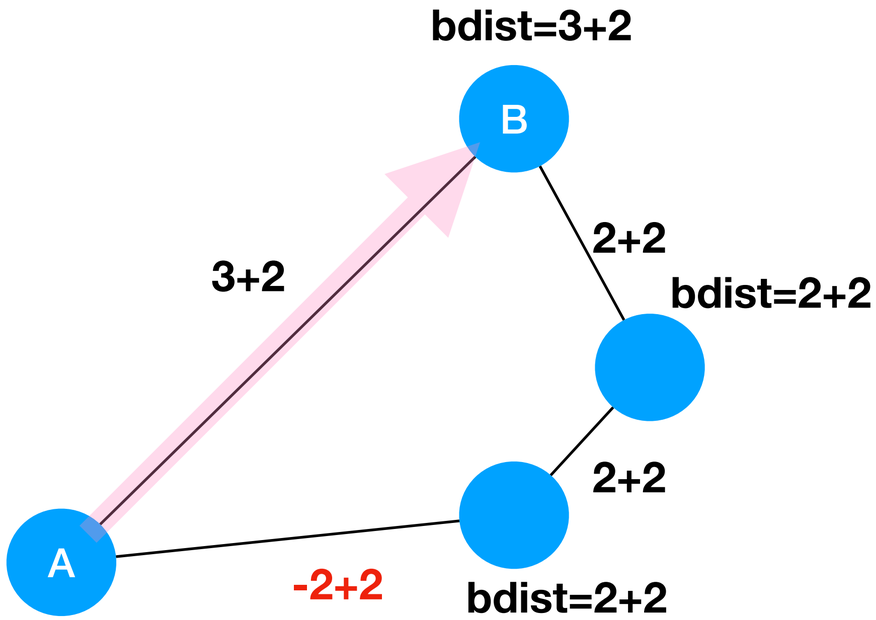
- Pour le plus court chemin classique, nous avions eu l'idée d'augmenter le poids de chaque arête par le poids en valeur absolue de l'arête la plus négative => mauvaise idée car cela change la solution



Réponse

- Mais augmenter d'une valeur constante δ ne change pas la solution. Les conditions d'optimalités restent satisfaites, simplement chaque bottleneck distance est augmentée de δ





Autre approche: Les Arbres Sous-tendants

- Première observation:

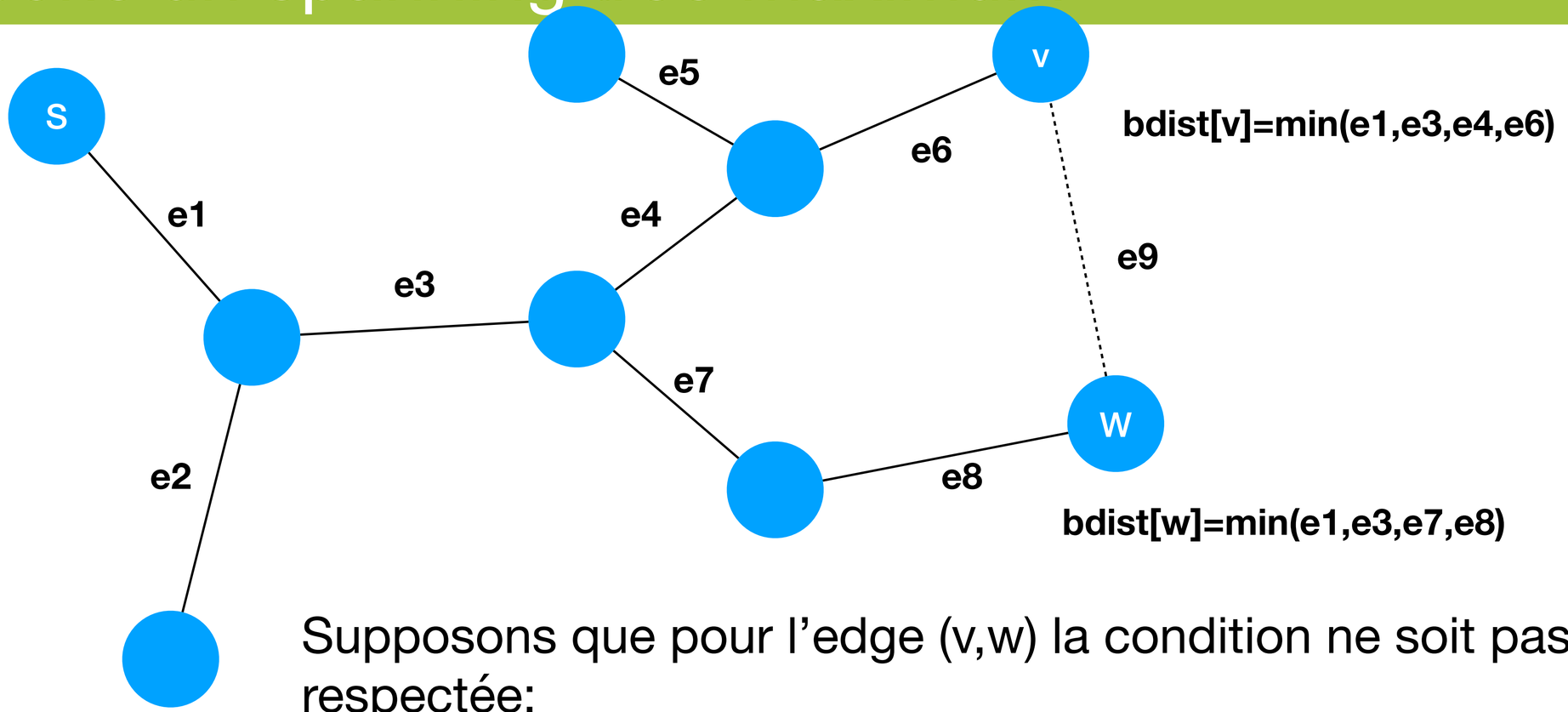
- On peut calculer un maximum spanning tree:
  - Adaptation de Kruskal: Considérer les edges dans l'ordre décroissant. On peut aussi adapter Prim.

- Pour le prouver, il faut vérifier que pour toutes les edges du maximum spanning respectent les conditions

$$\forall\ e=(v,w) : bdist[w] \geqslant \min(bdist[v], e.weight)$$

- Voyons pourquoi …

Supposons un spanning tree maximum



Supposons que pour l'edge (v,w) la condition ne soit pas respectée:

min(bdist[v],e9) > bdist[w] $\Longrightarrow$

min(e1,e3,e4,e6,e9) > min(e1,e3,e7,e8) $\Longrightarrow$

min(e4,e6,e9) > min(e7,e8) $\Longrightarrow$

on peut faire un spanning tree encore plus lourd en incluant e9 et en supprimant l'edge la plus légère entre e7 et e8 (contradiction avec notre hypothèse de départ)

All Pairs Maximum Bottle-Neck Distance

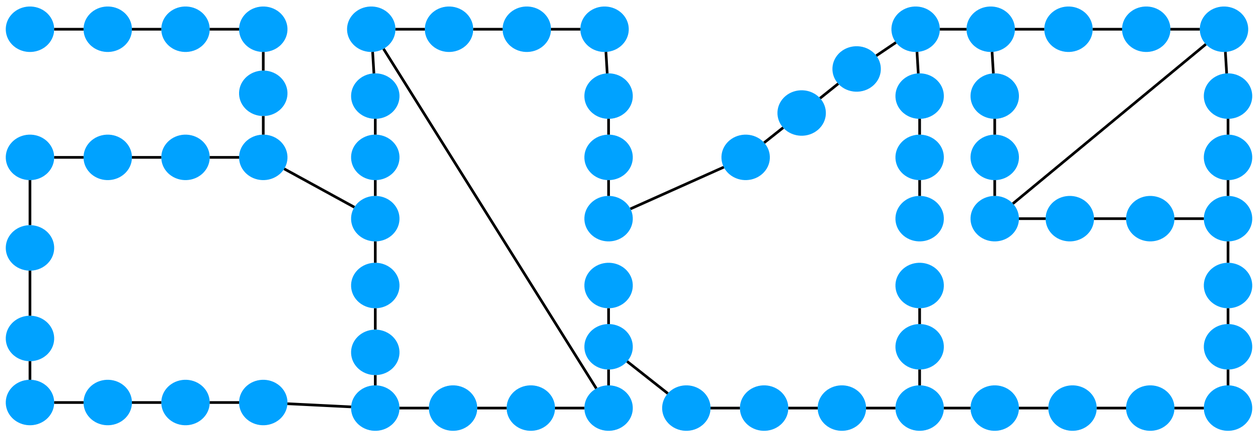
- C’est intéressant car pour construire le spanning tree, nous n’avons pas tenu compte de S (noeud origine).

- On peut donc calculer le “all-pair maximum bottle-neck distance”:
  - O(E log(E)) pour obtenir le spanning tree
  - Calculer toutes les maximum b-distance au départ d’un noeud: O(V), donc $O(V^2)$ au total. C’est mieux que Bellman-Ford.

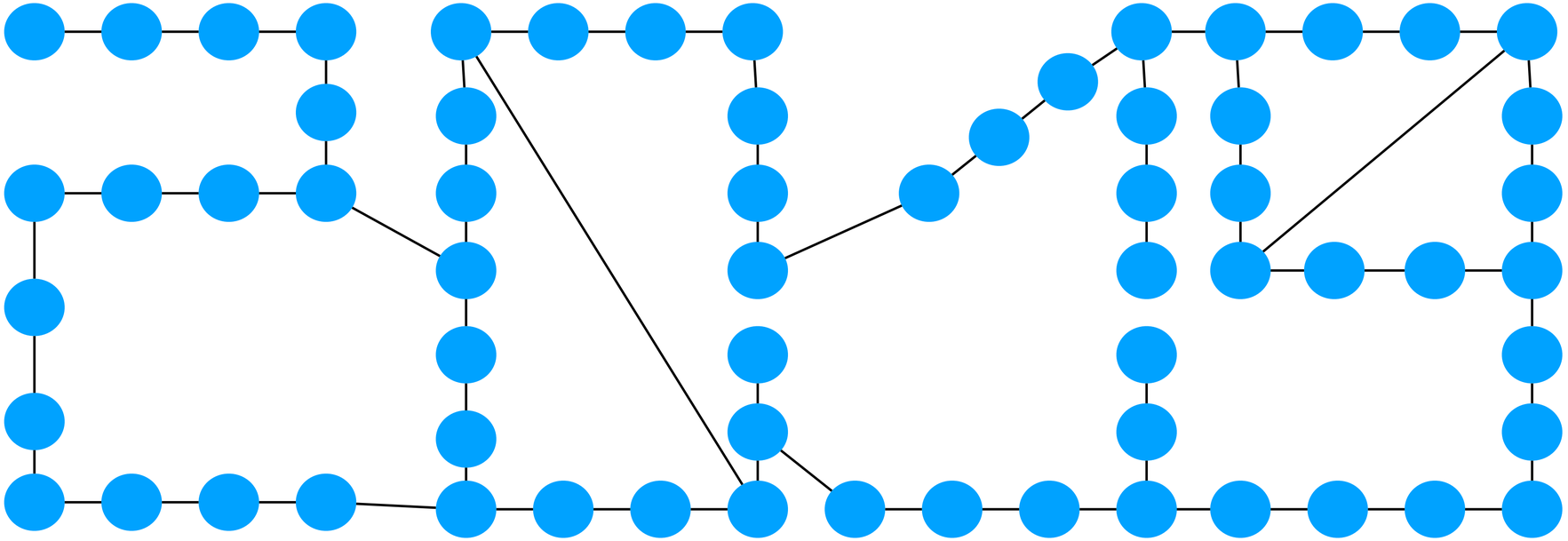
- Etant donné un noeud choisi aléatoirement qui s’allume en premier, combien de temps faudra-t-il au minimum pour allumer toute la guirlande ?

- Comment calculer le temps nécessaire pour allumer toute la guirlande étant donné un noeud choisi au départ ?

- Est-ce q’on ne peut pas réutiliser cet algorithme pour trouver le noeud de départ qui donnera le plus petit temps ?

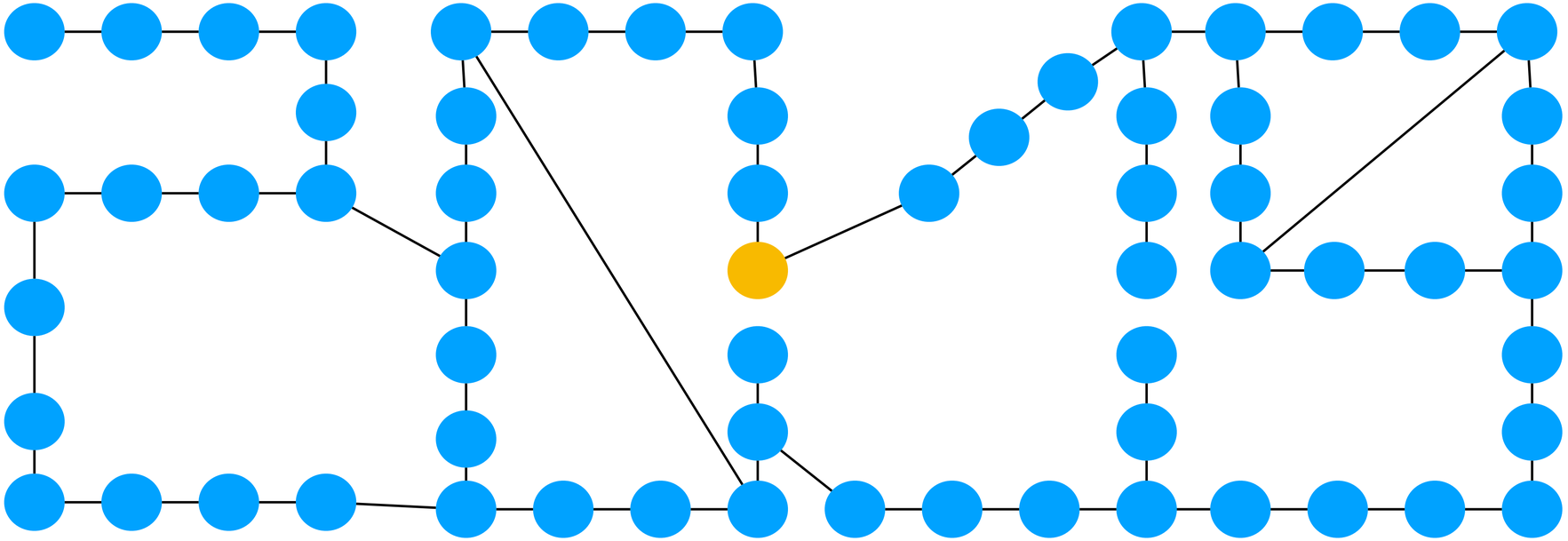


Le problème de la Guirlande de Noel



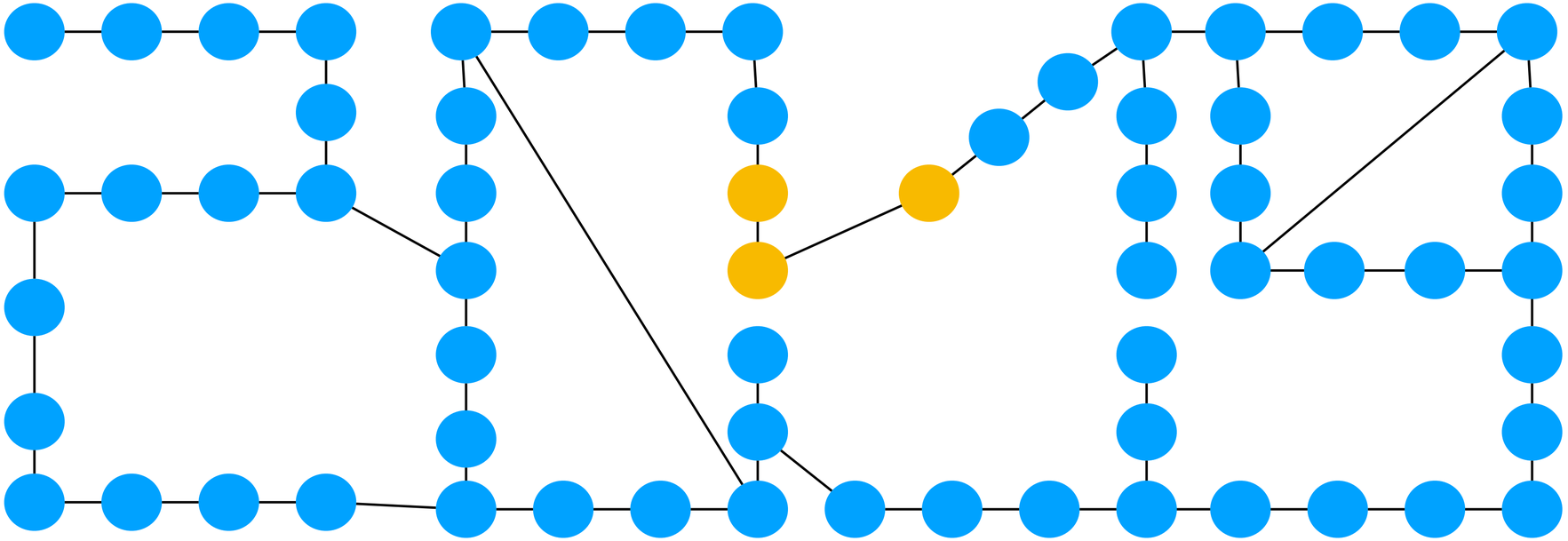
Le problème de la Guirlande de Noel

t=0



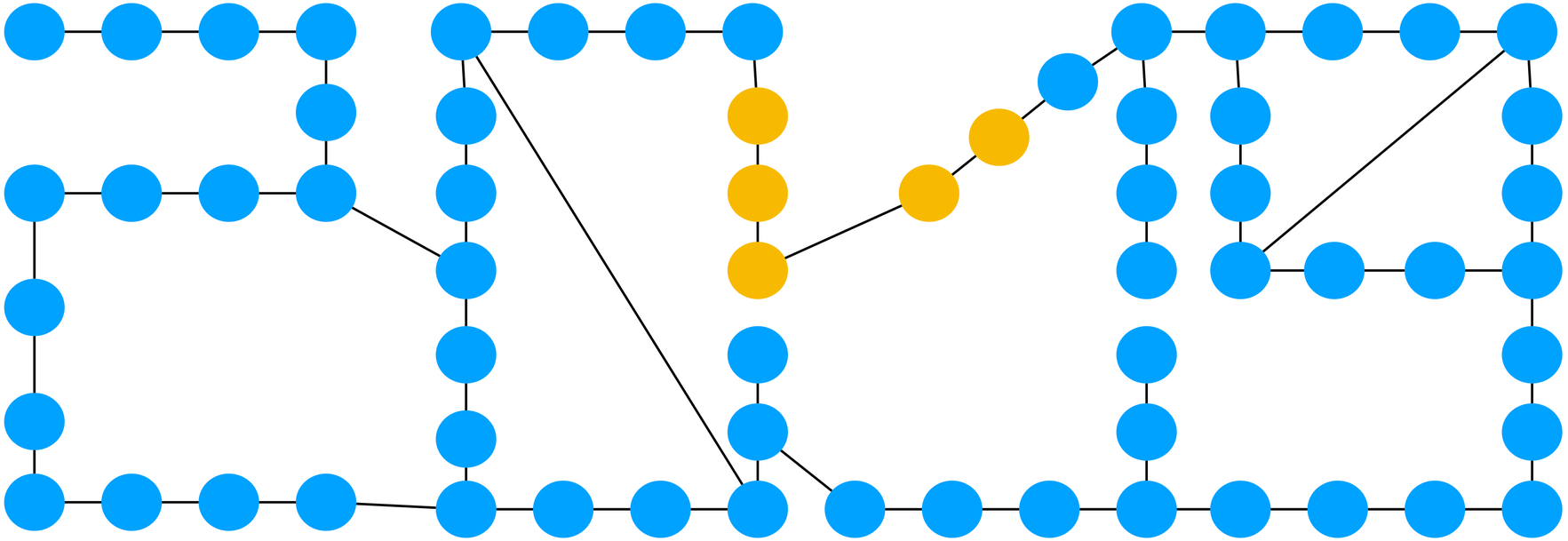
Le problème de la Guirlande de Noel

t=1



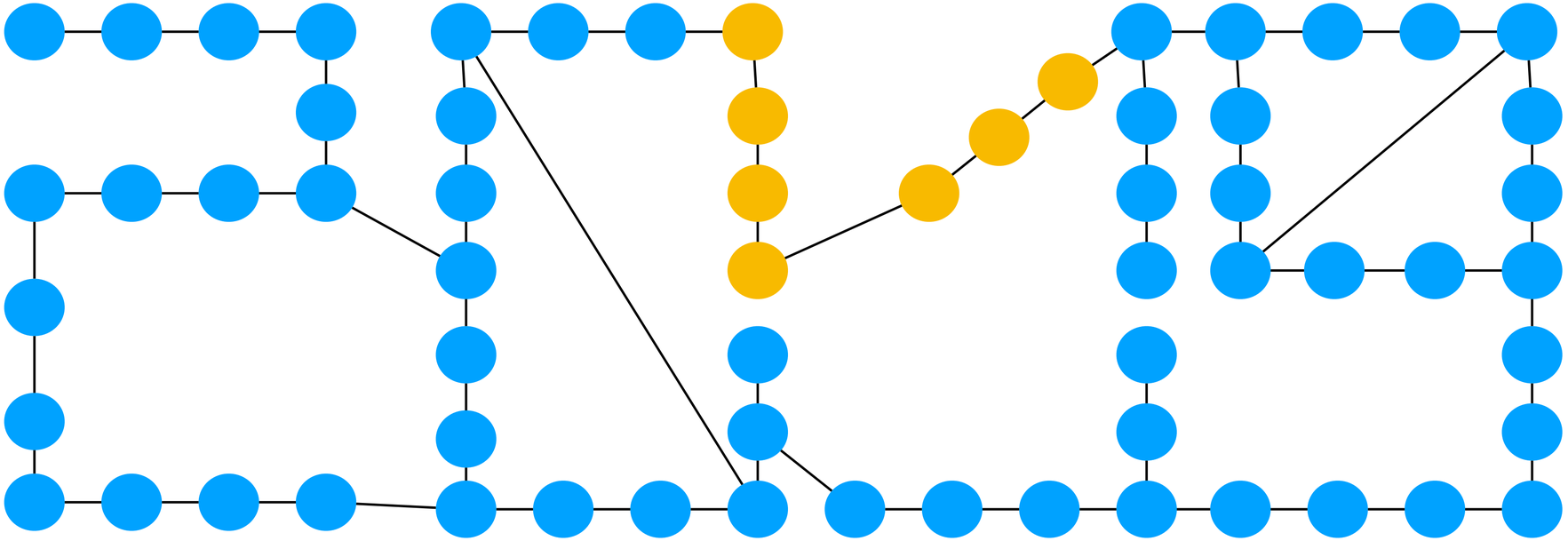
Le problème de la Guirlande de Noel

t=2



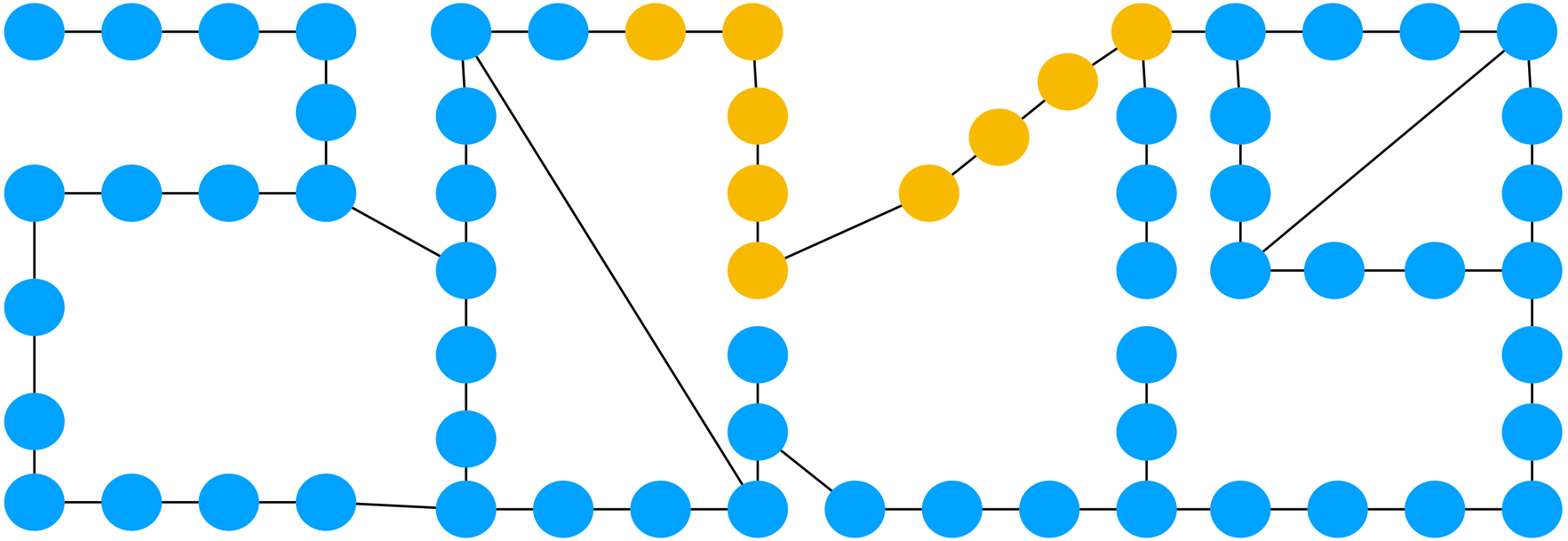
Le problème de la Guirlande de Noel

t=3



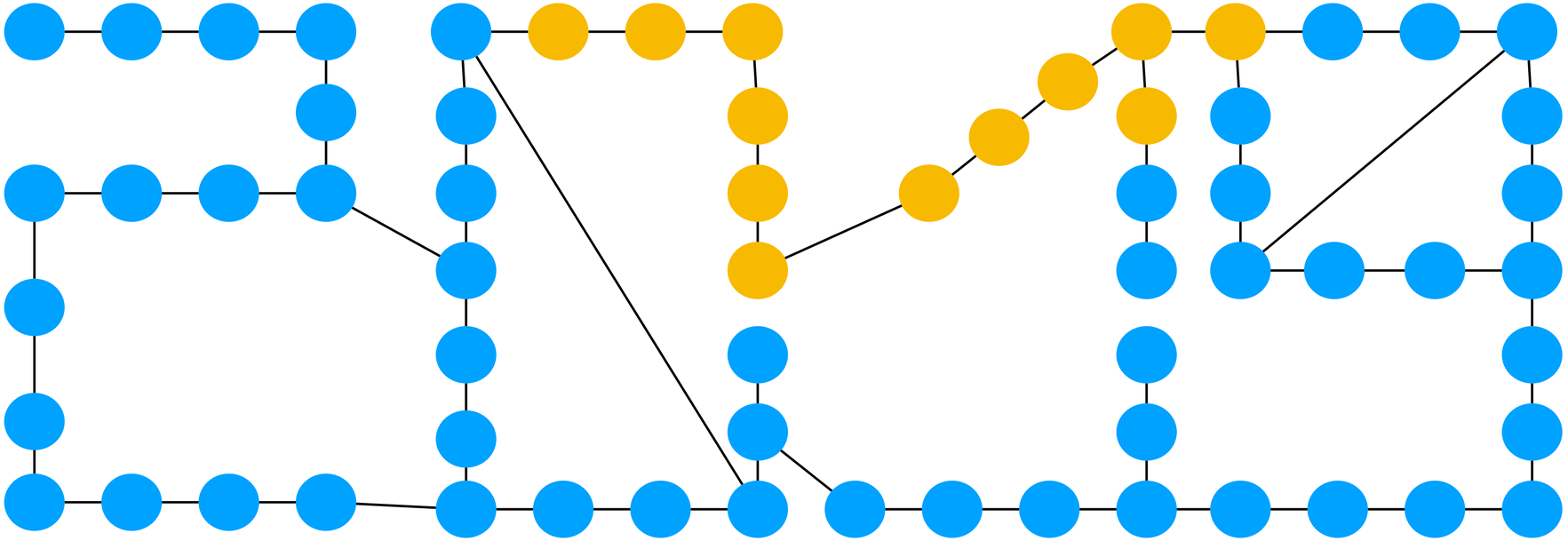
Le problème de la Guirlande de Noel

t=4



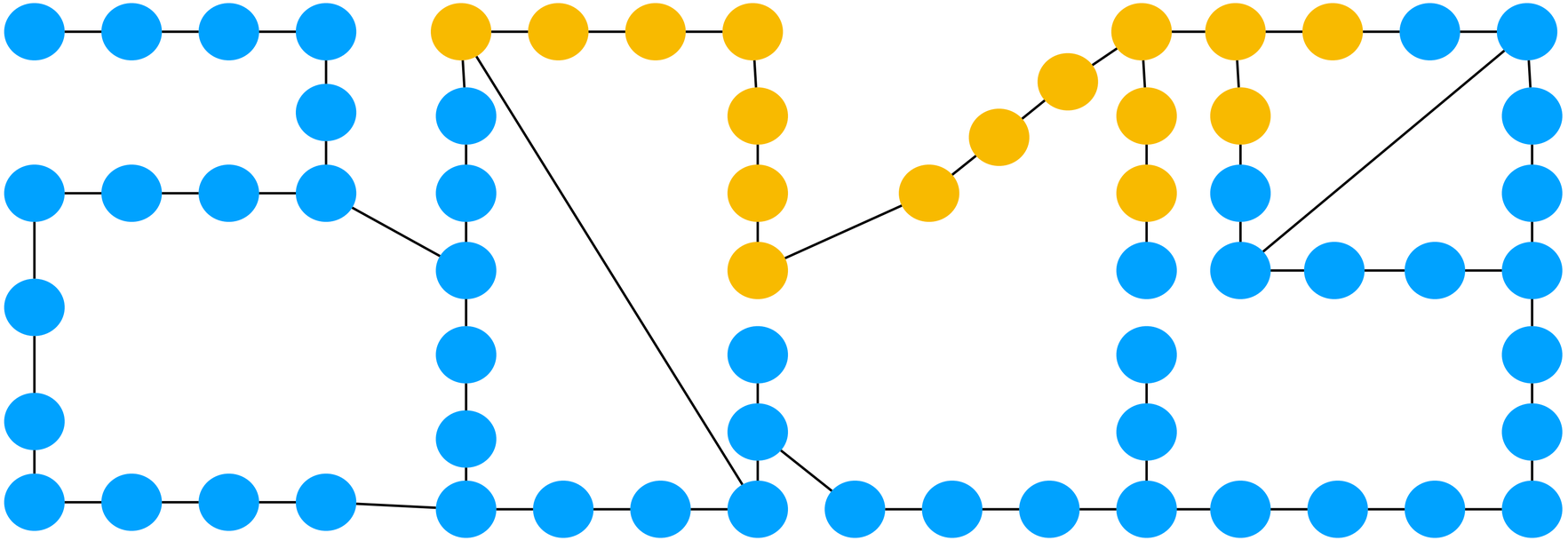
Le problème de la Guirlande de Noel

t=5



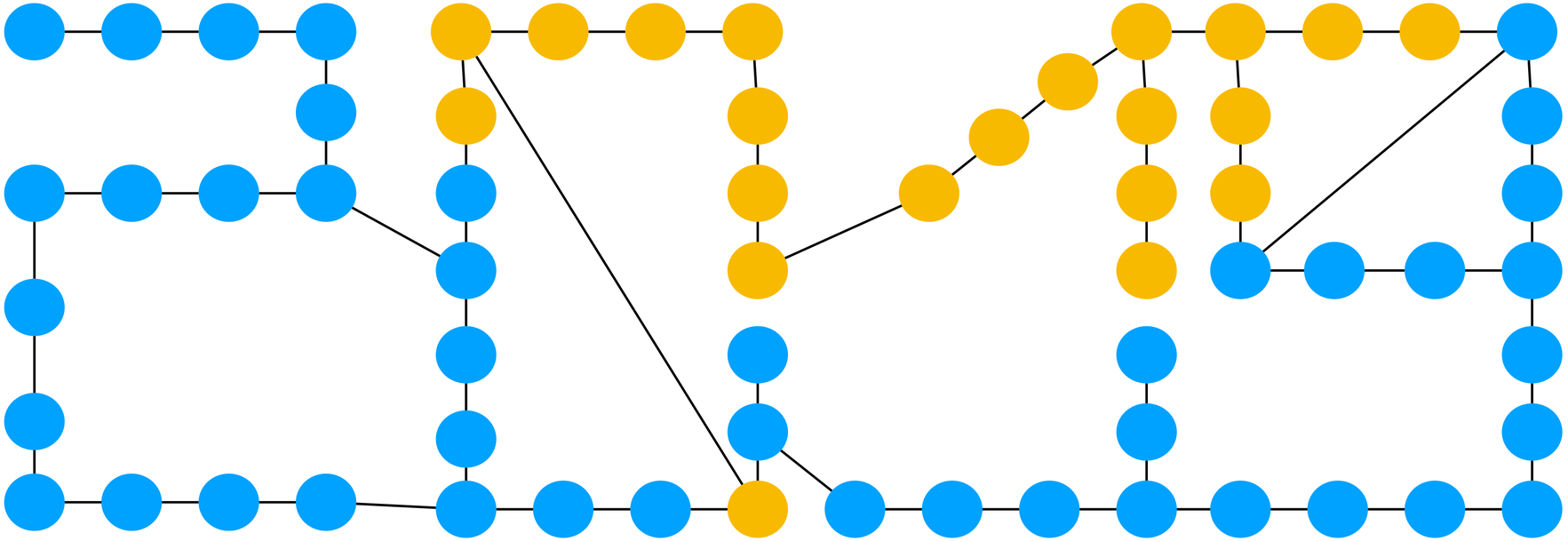
Le problème de la Guirlande de Noel

t=6



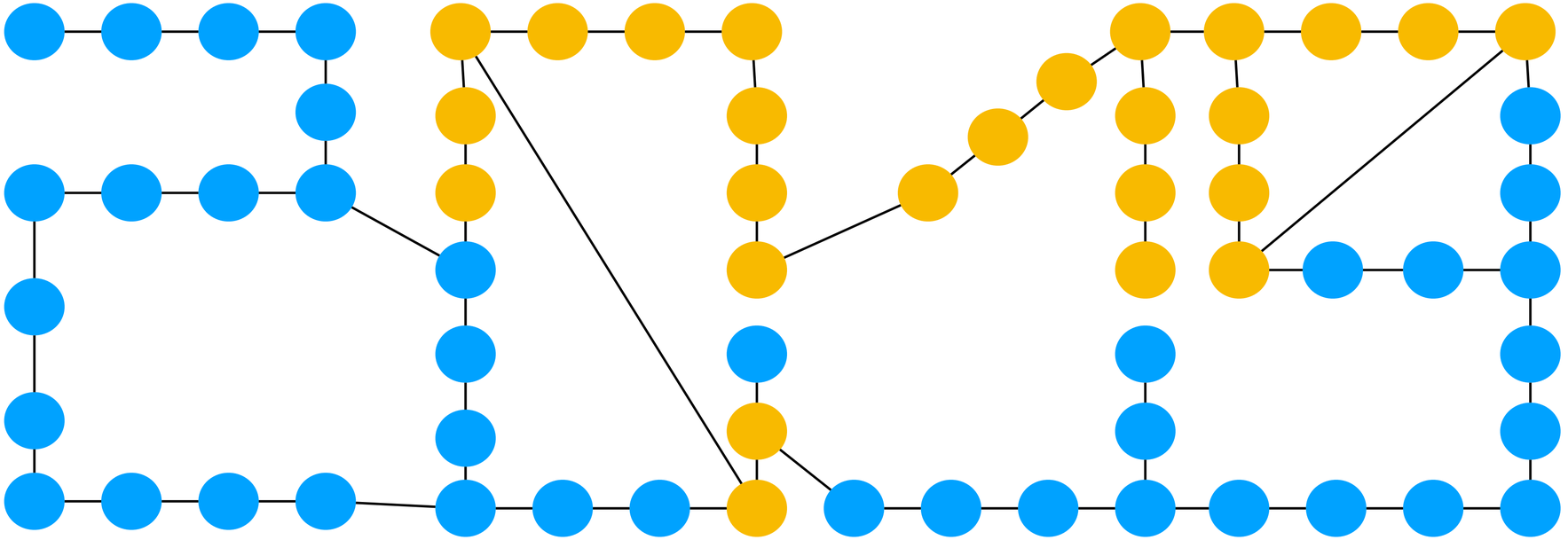
Le problème de la Guirlande de Noel

t=7



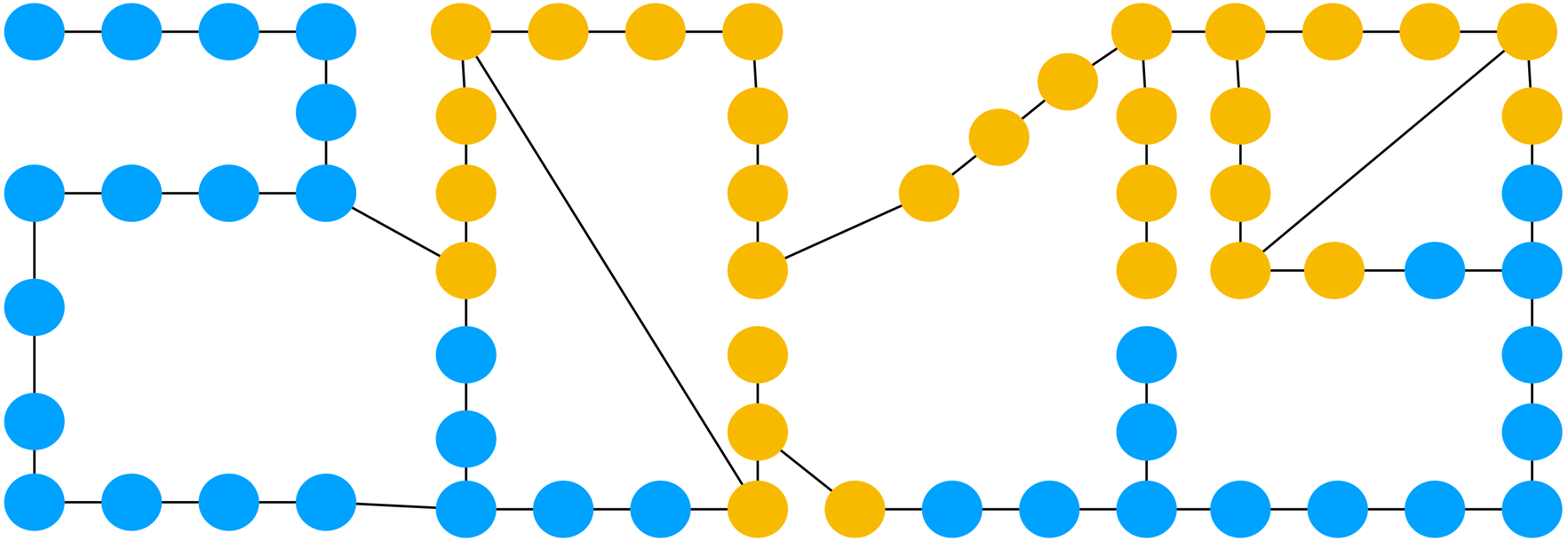
Le problème de la Guirlande de Noel

t=8



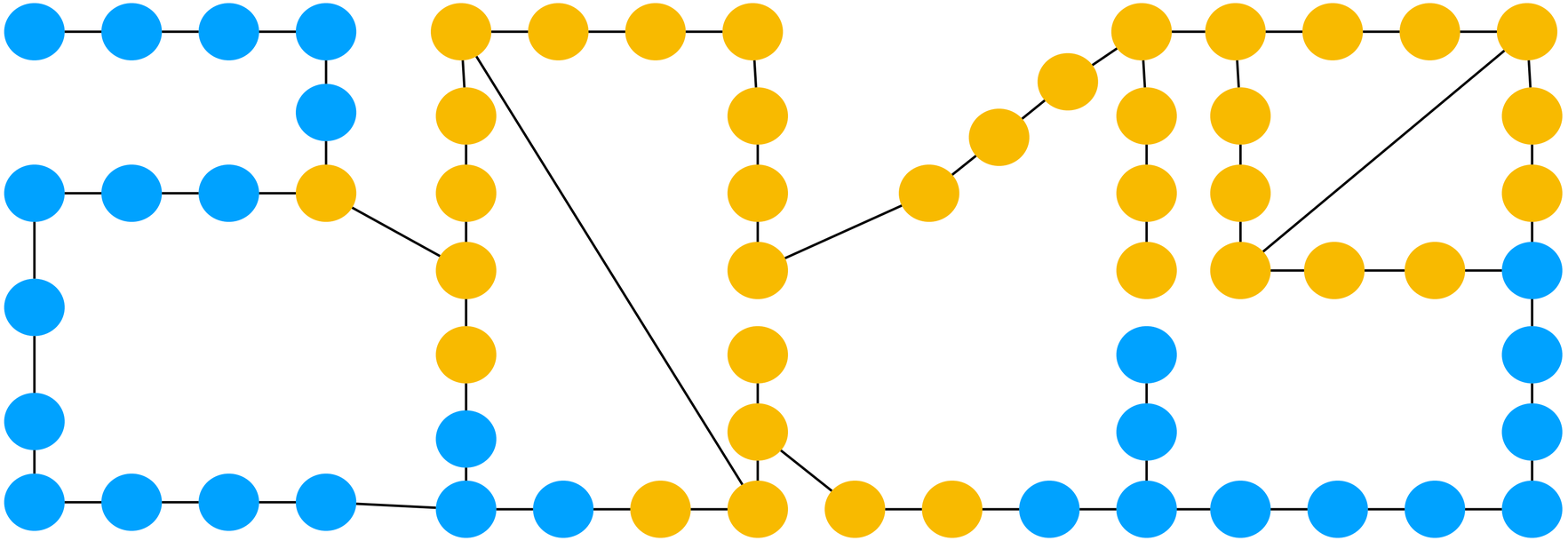
Le problème de la Guirlande de Noel

t=9



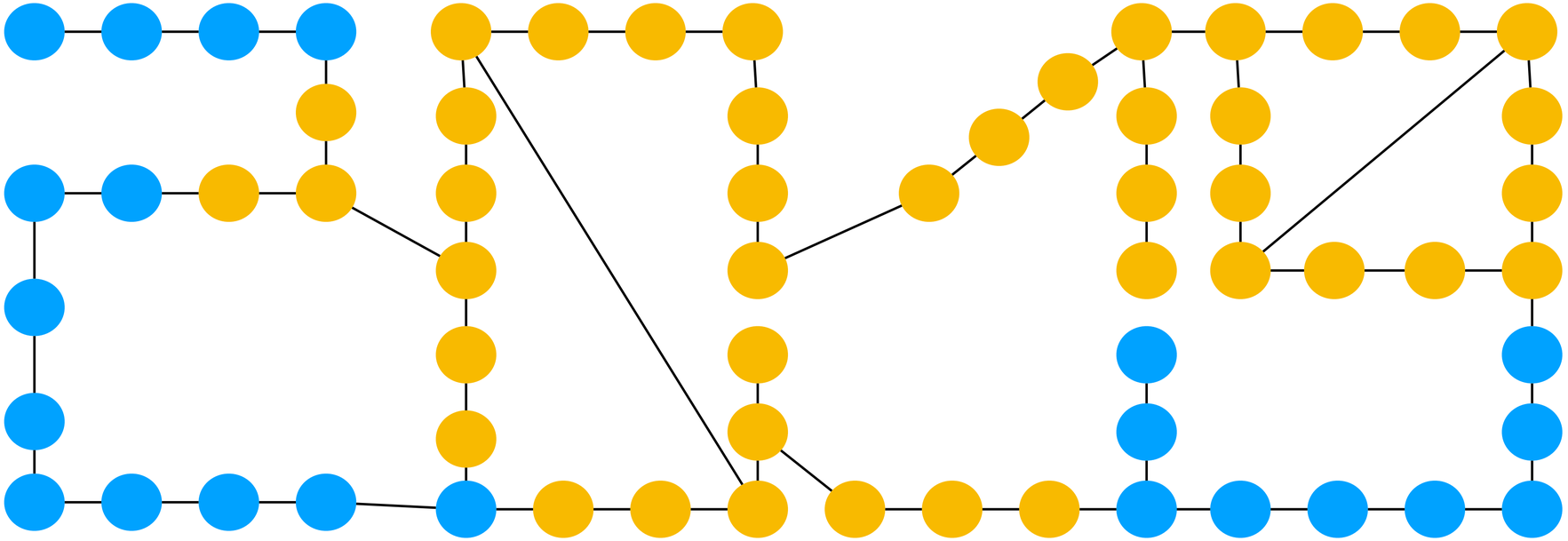
Le problème de la Guirlande de Noel

t=10



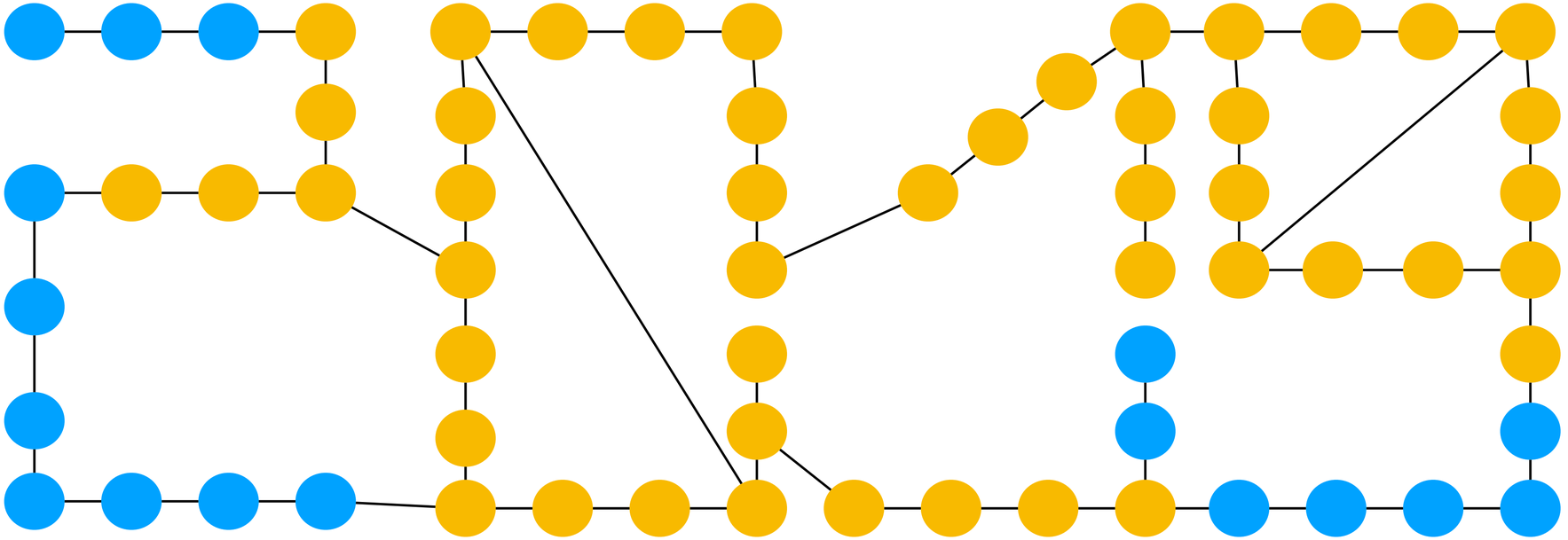
Le problème de la Guirlande de Noel

t=11



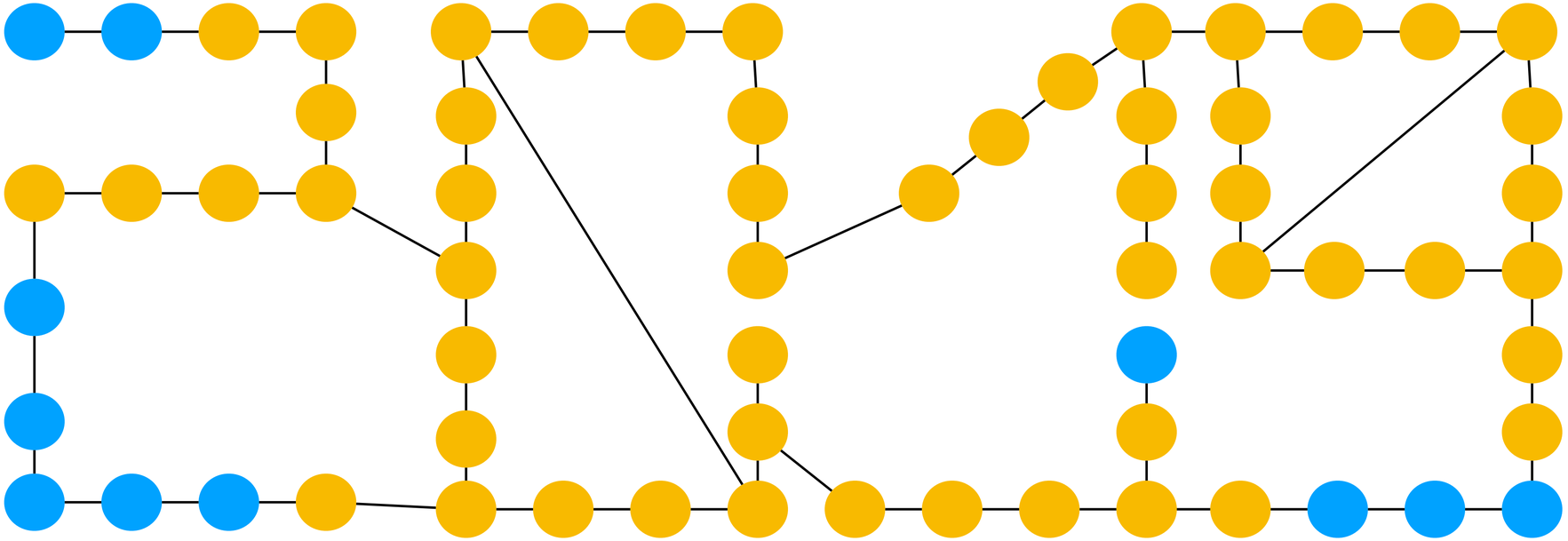
Le problème de la Guirlande de Noel

t=12



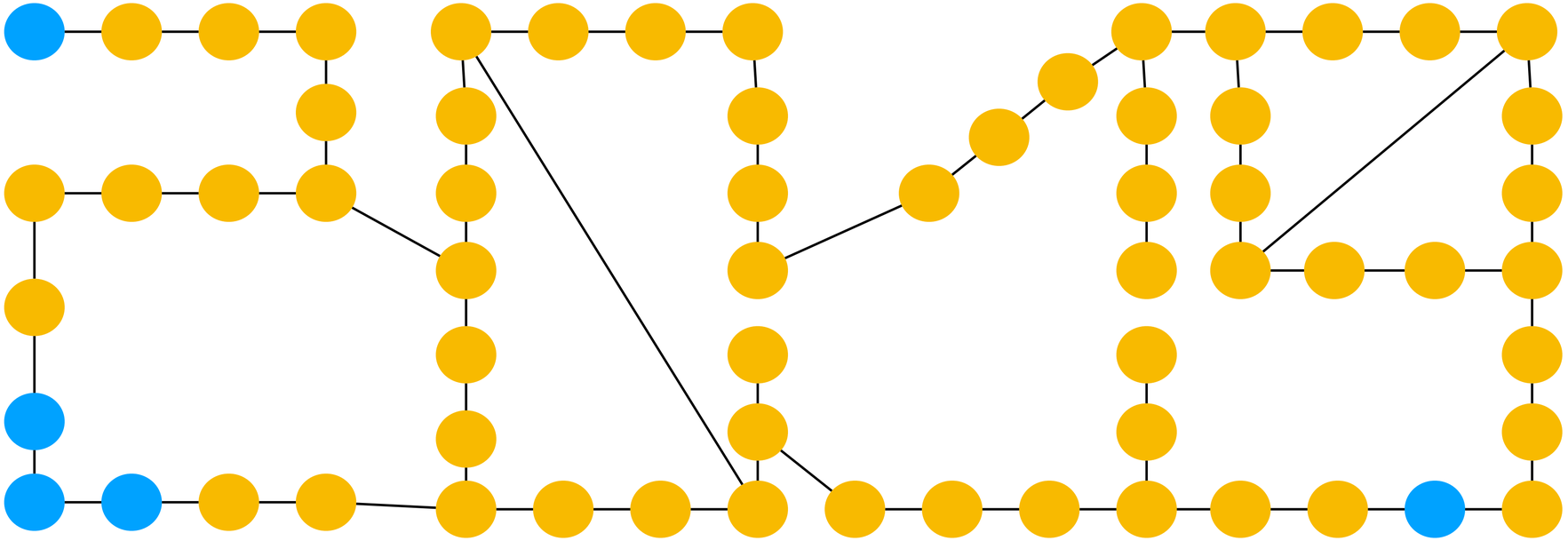
Le problème de la Guirlande de Noel

t=13



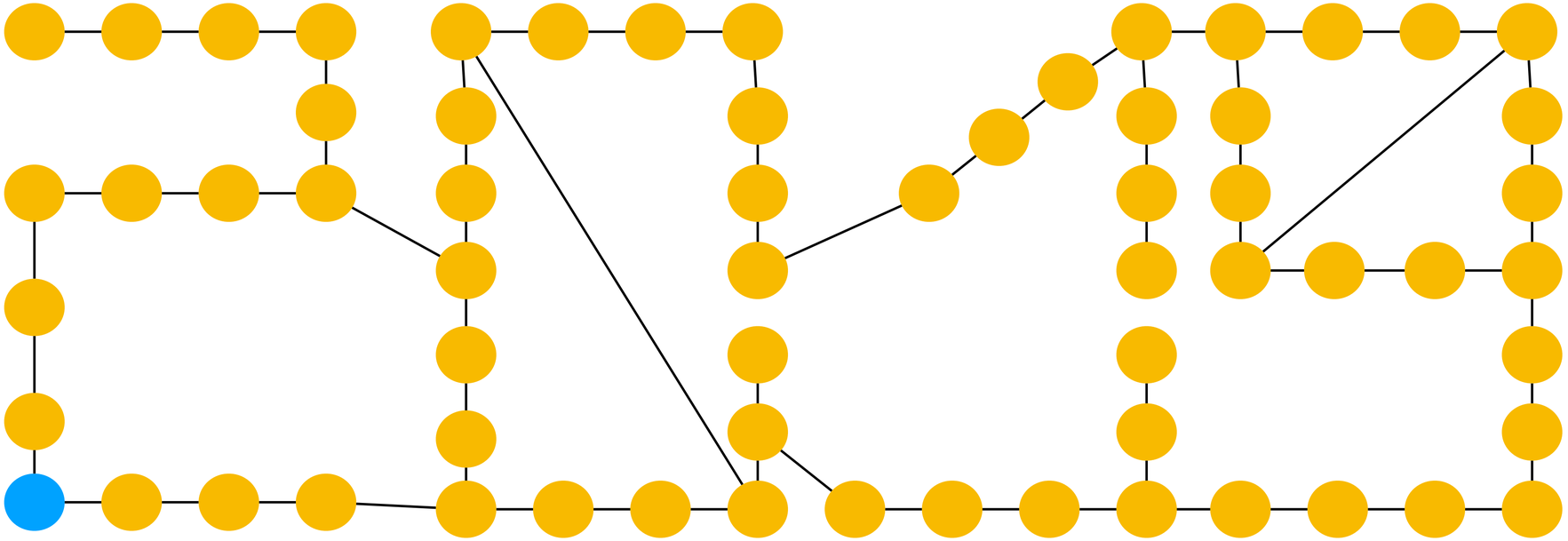
Le problème de la Guirlande de Noel

t=14



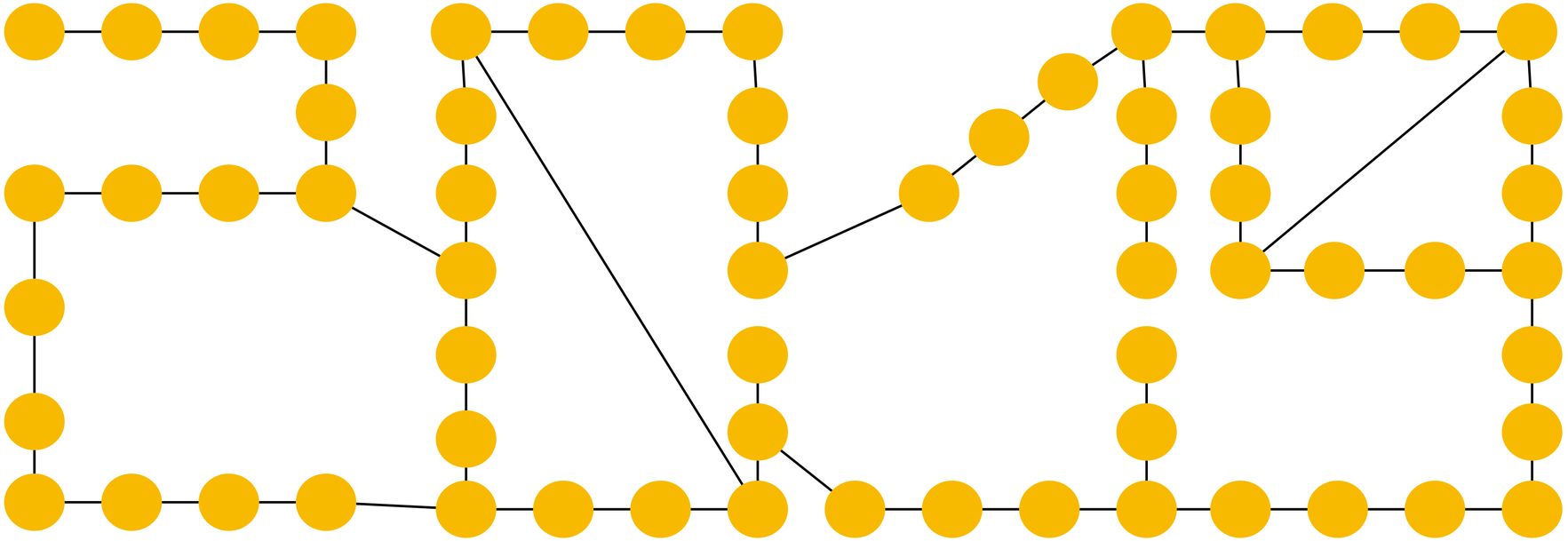
Le problème de la Guirlande de Noel

t=15



Le problème de la Guirlande de Noel

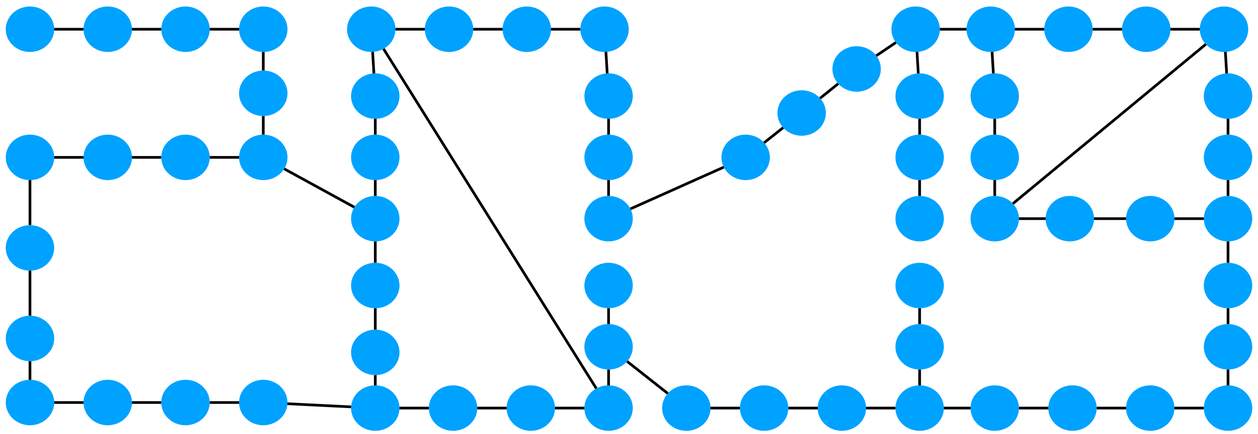
t=16



- Etant donné un noeud choisi aléatoirement qui s’allume en premier, combien de temps faudra-t-il au minimum pour allumer toute la guirlande ?

- Comment calculer le temps nécessaire pour allumer toute la guirlande étant donné un noeud choisi au départ ?

- Est-ce q’on ne peut pas réutiliser cet algorithme pour trouver le noeud de départ qui donnera le plus petit temps ?



- Etant donné un noeud qui s'allume, comment calculer le temps nécessaire pour allumer toute la guirlande ?
  - Oui, c'est du BFS qui s'exécute en O(V+E)

- Est-ce q'on ne peut pas réutiliser cet algorithme pour trouver le noeud de départ qui donnera le plus petit temps ?
  - Oui: un BFS au départ de chaque noeud => O(V*(V+E))
  - Pour aller plus vite:
    - Commencer d'abord sur les noeuds avec un grand degré (heuristique)
    - Maintenir la meilleure solution courante et arrêter un BFS dès que la distance devient moins bon que la meilleure solution courante.

The End …

Best Wishes for