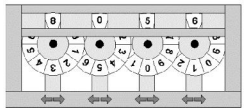
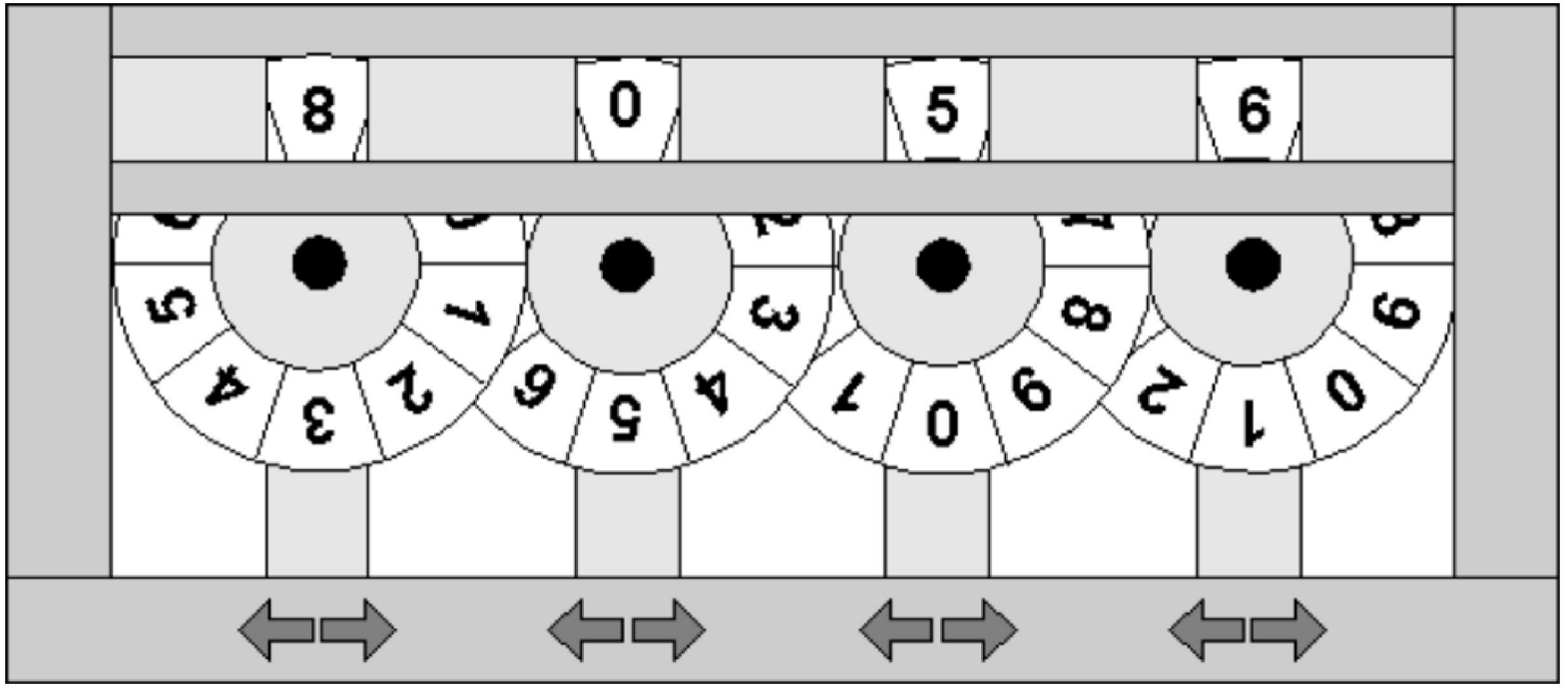
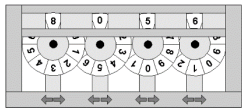




Les roues tournent

Représentation d'une configuration

```
private static class Config {
    private final byte[] digits = new byte[NB_WHEELS];

    private Config(String s) {
        for(int k = 0; k < s.length(); ++k)
            digits[k] = (byte)(s.charAt(k) - '0');

    // ...
}
```



Un problème d'exploration.

Quels sont les successeurs de 8056?

A priori: 9056, 7056, 8156, 8956, 8066
8046, 8057, 8055.

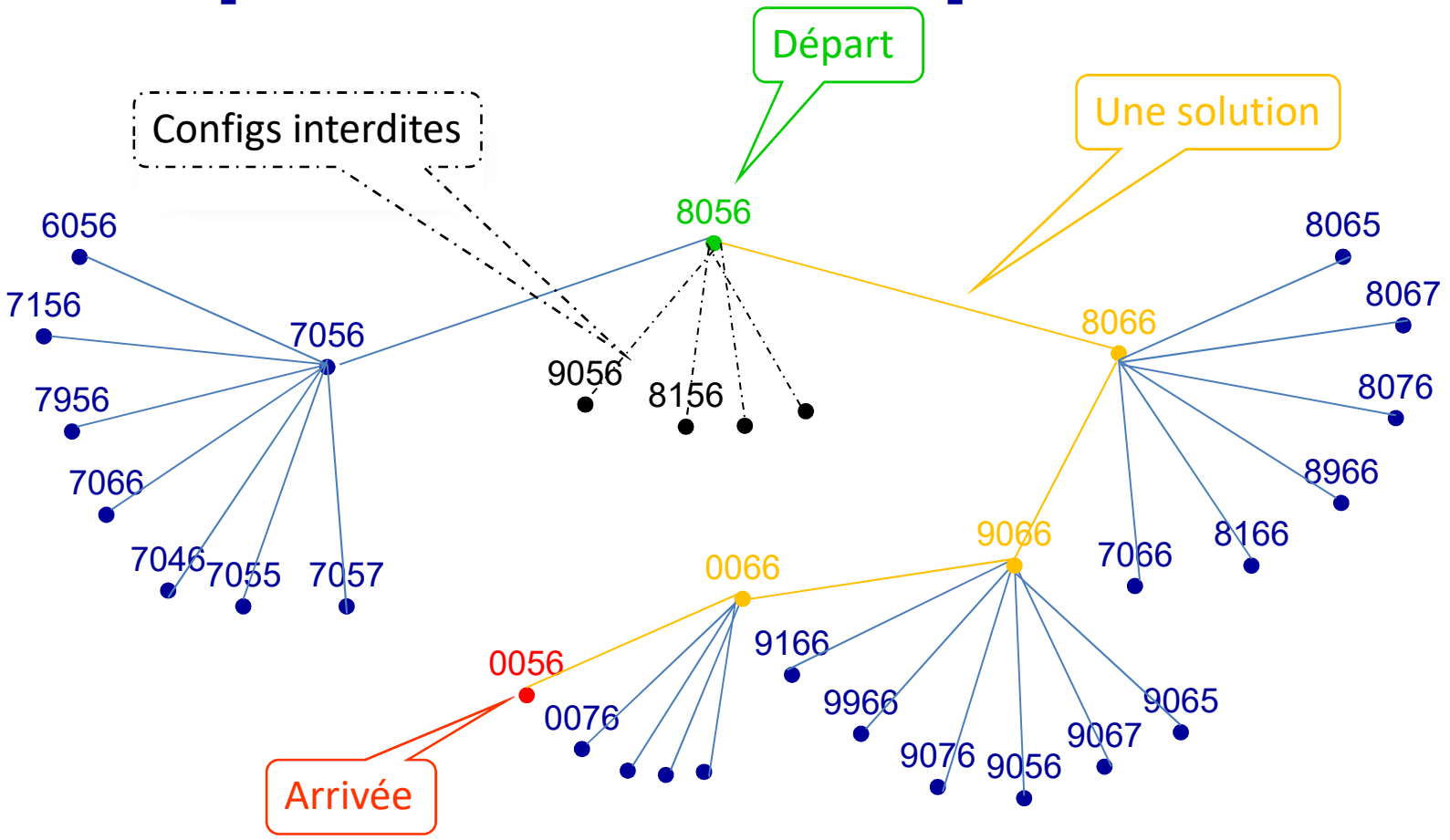
Certains ne sont peut-être pas permis comme
9056, 8156, 8956, 8046, 8055, 8057

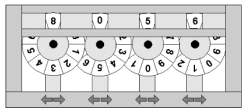
Donc restent 7056, 8066.

Et de là?

…

Un problème d'exploration.

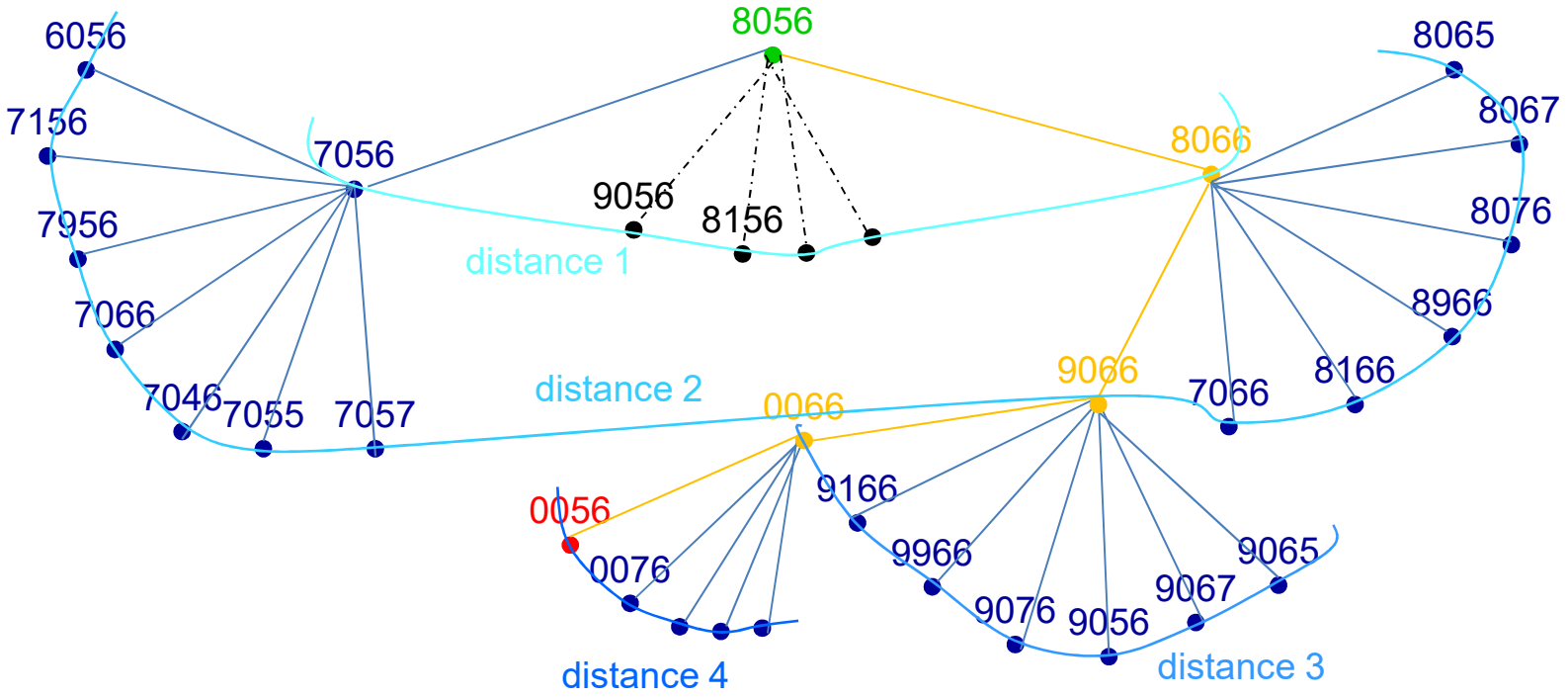


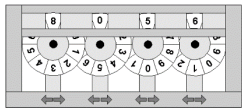


Un problème d'exploration.

Pour trouver le plus court chemin on fait une exploration en largeur: Tous ceux à distance 1 du départ puis ceux à distance 2 (distance 1 de ceux qui sont à distance 1)...

Un problème d'exploration.

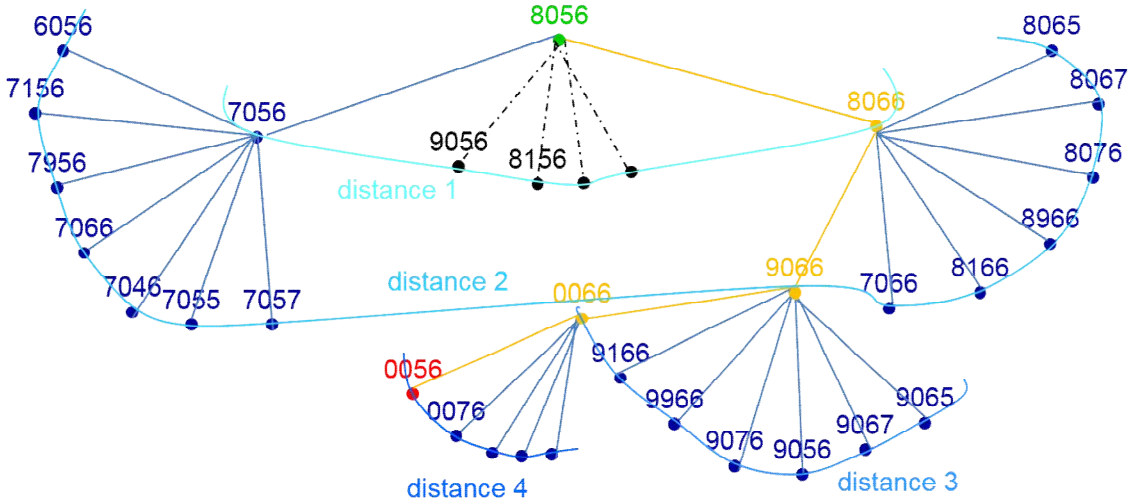


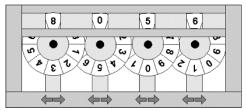


Un problème d'exploration.

Pour parcourir en largeur, on utilise une File (Queue).

- Y mettre la config de départ
- Boucler:
  - Prendre une config dans la file
  - Vérifier si on a trouvé l'arrivée
  - Sinon, ajouter ses successeurs (ses voisins à distance 1) à la file



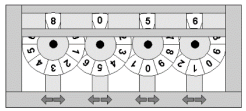


Un problème d'exploration.

Pour parcourir en largeur, on utilise une File (Queue).

```
private String solve() {
    // ...
    Queue<Config> queue = new LinkedList<>();
    queue.add(start);
    used.add(start);
    while(!queue.isEmpty()) {
        Config c = queue.remove();
        if(c.equals(stop))
            return assembleSolution();
        for(Config next : c.successors()) {
            queue.add(next);
            used.add(next);
        }
    }
    return "";
}
```

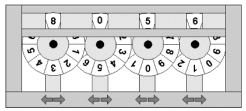
Explication plus loin



Un problème d'exploration.

Les successeurs

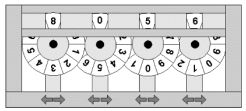
- Pour chacun des 4 chiffres
- Générer la configuration avec +1 et -1 modulo 10



Les successeurs d'une configuration

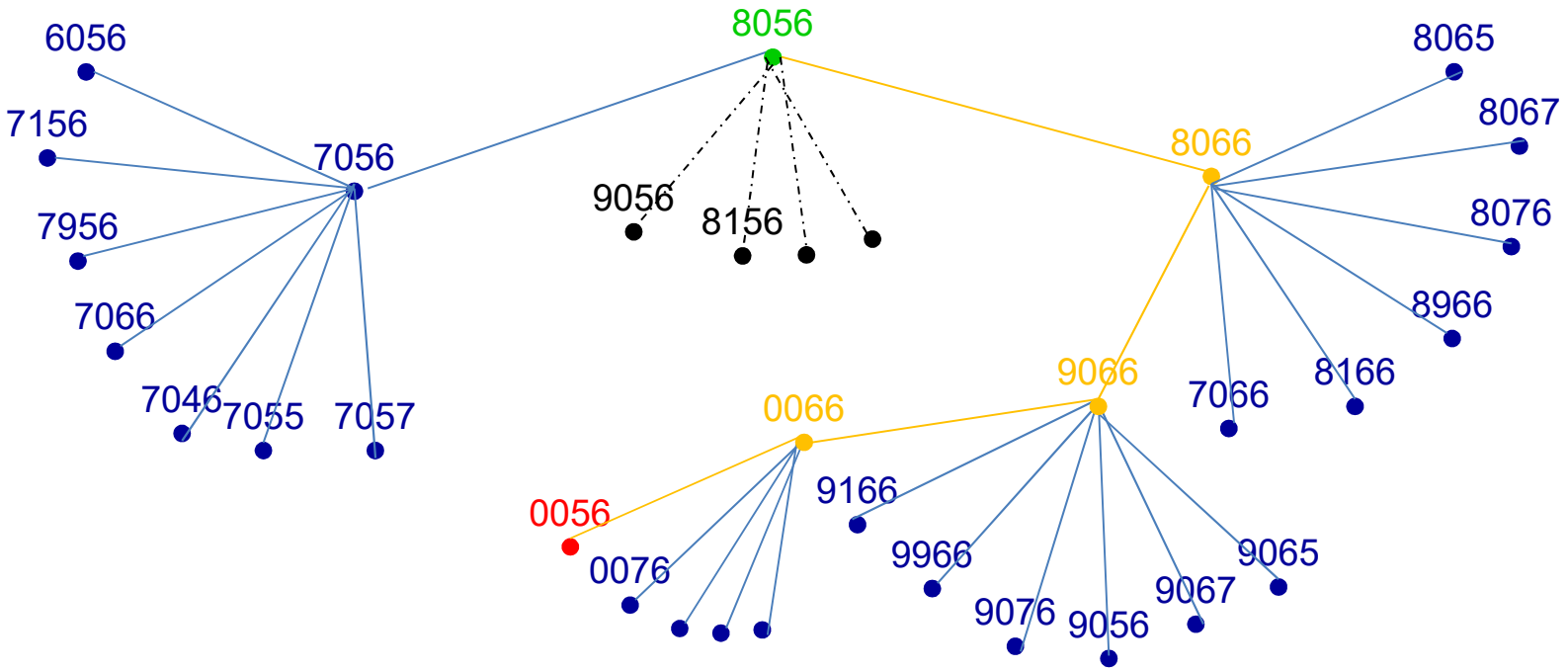
```
private static class Config {

    // ...

    private List<Config> successors(Set<Config> used) {
        List<Config> succs = new ArrayList<>();
        for(int i = 0; i < NB_WHEELS; ++i)
            for(int sign = -1; sign <= 1; sign += 2) {
                Config succ = new Config(this);
                succ.digits[i] =
                    (byte)((succ.digits[i] + sign + 10) % 10);
                if(!used.contains(succ))
                    succs.add(succ);
            }
        return succs;
    }
}
```

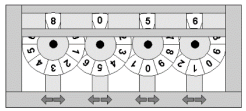
Explication plus loin



Ne pas tourner en rond

Dès qu'une config a été mise dans la file, il devient impossible d'y repasser.

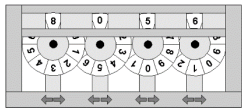




Ne pas tourner en rond

Utiliser un Set des configs impossibles

```
// Les Config par lesquelles il ne faut pas (re)passer
private final Set<Config> used;
// ...
    private String solve() {
        // ...
        Queue<Config> queue = new LinkedList<>();
        queue.add(start);
        used.add(start);
        while(!queue.isEmpty()) {
            Config c = queue.remove();
            if(c.equals(stop))
                return assembleSolution();
            for(Config next : c.successors(used)) {
                queue.add(next);
                used.add(next);
            }
        }
        return "";
    }
```

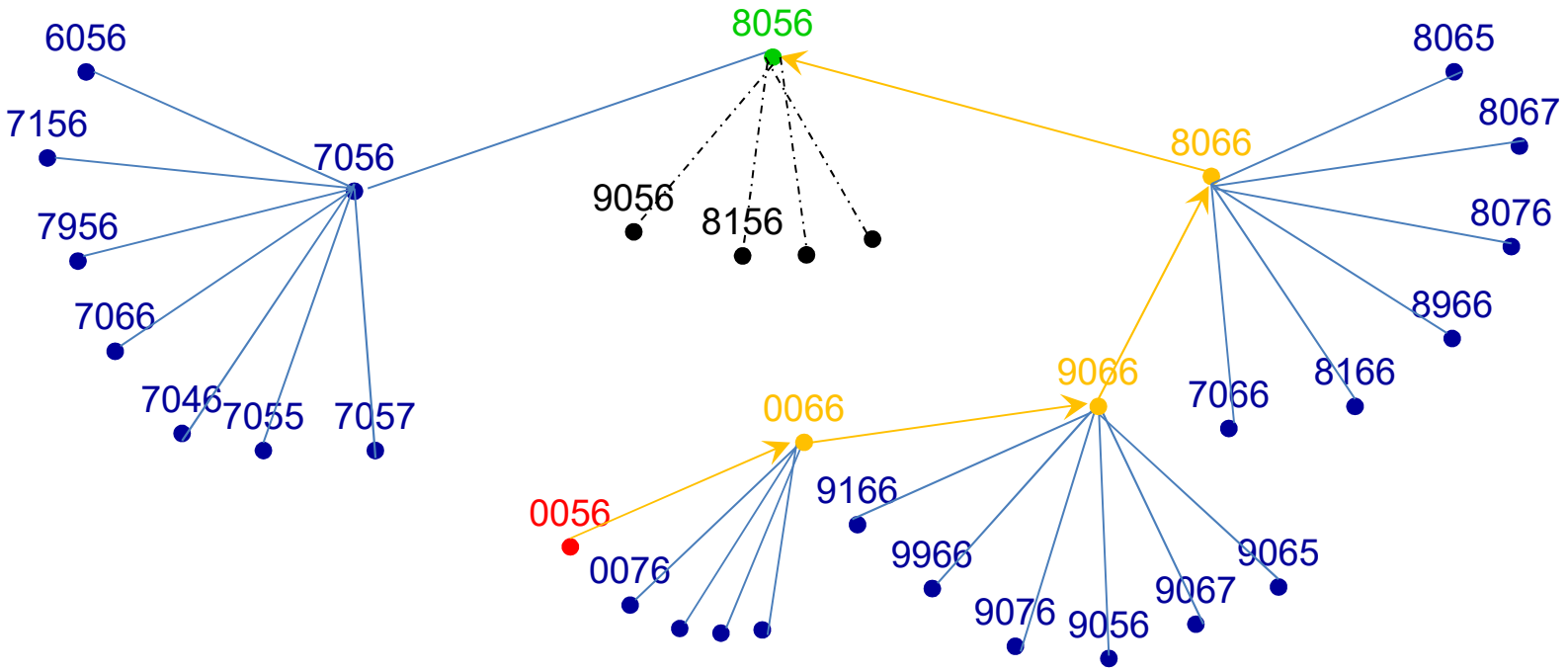


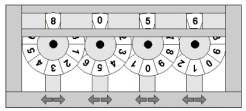
Les successeurs d'une configuration

```
private static class Config {

    // ...

    private List<Config> successors(Set<Config> used) {
        List<Config> succs = new ArrayList<>();
        for(int i = 0; i < NB_WHEELS; ++i)
            for(int sign = -1; sign <= 1; sign += 2) {
                Config succ = new Config(this);
                succ.digits[i] =
                    (byte)((succ.digits[i] + sign + 10) % 10);
                if(!used.contains(succ))
                    succs.add(succ);
            }
        return succs;
    }
}
```

Mémoriser le chemin

Chaque nouvelle config doit savoir de laquelle elle est obtenue

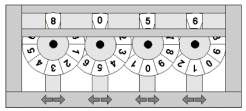




Mémoriser le chemin

Un objet *SolutionNode* mémorise une config et un lien vers le précédent

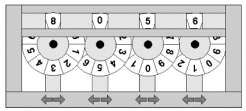
```
private static class SolutionNode {

    public SolutionNode(Config c, SolutionNode prec) {
        this.current = c;
        this.prec = prec;
    }
    public final Config current;
    public final SolutionNode prec;
}
```



Mémoriser le chemin

Dans la File, au lieu de mémoriser les *Config*, on mémorise les *SolutionNode*

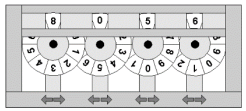
```
private String solve() {
    // ...
    Queue<SolutionNode> queue = new LinkedList<>();
    queue.add(new SolutionNode(start, null));
    used.add(start);
    while(!queue.isEmpty()) {
        SolutionNode sl = queue.remove();
        if(sl.current.equals(stop)) {
            return assembleSolution(sl);
        }
        for(Config next : sl.current.successors(used)) {
            queue.add(new SolutionNode(next, sl));
            used.add(next);
        }
    }
    return "";
}
```



Construire la solution

La solution est obtenue **à l'envers** en suivant les liens *prec*.

```
private static String assembleSolution(SolutionNode sl) {
    StringBuilder sol = new StringBuilder();
    while(sl != null) {
        sol.append(sl.current);
        sol.append(' ');
        sl = sl.prec;
    }
    return sol.toString();
}
```



Construire la solution

Mais pourquoi à l'envers? Il suffit de partir de stop et d'aller vers start

```
private String solve() {
    // ...
    Queue<SolutionNode> queue = new LinkedList<>();
    queue.add(new SolutionNode(stop, null));
    used.add(stop);
    while(!queue.isEmpty()) {
        SolutionNode sl = queue.remove();
        if(sl.current.equals(start)) {
            return assembleSolution(sl);
        }

    // ...
}
```