Optimisation combinatoire du jeu Dobble

EL MAHDI OULKADI
54927

Sommaire

Le jeu de Dobble : simple en principe mais pas aussi simple à construire !

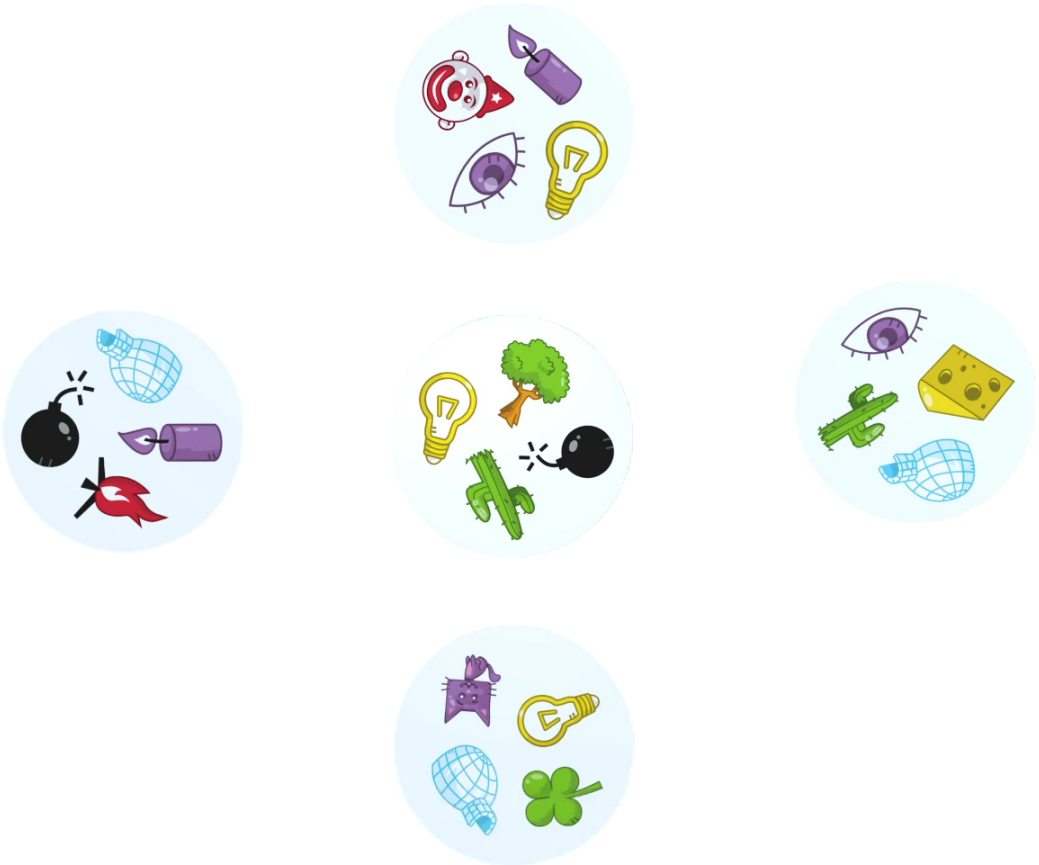
Principe du jeu Dobble

- 55 cartes
- 8 symboles différents sur chaque carte
- Jeu d'observation : trouver le symbole en commun

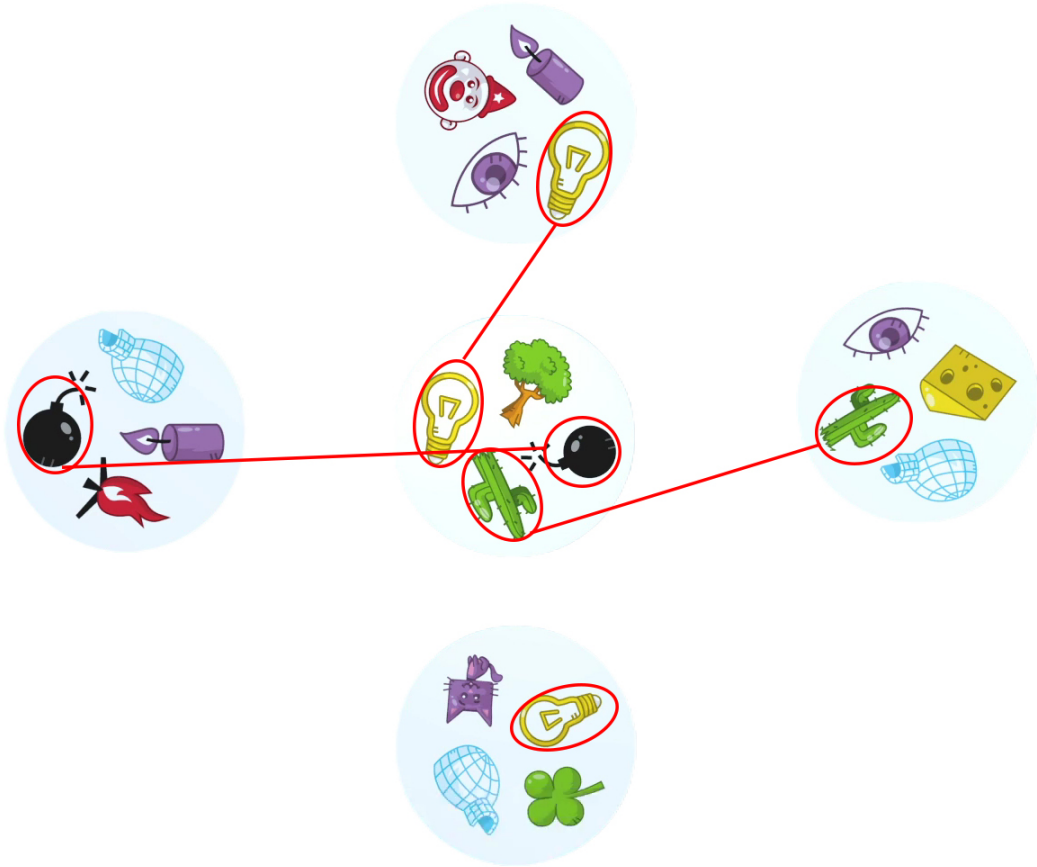
Propriété fondamentale

(P) : Deux cartes partagent toujours un et un seul symbole

Principe du jeu Dobble

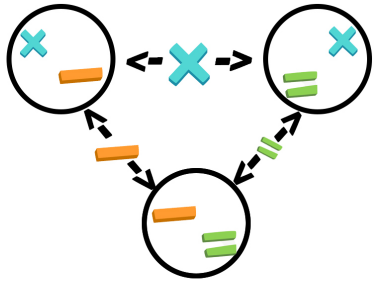


Principe du jeu Dobble

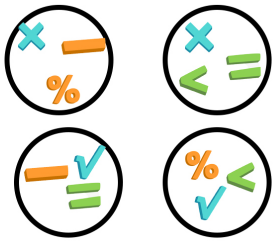


Construction de petits jeux valides

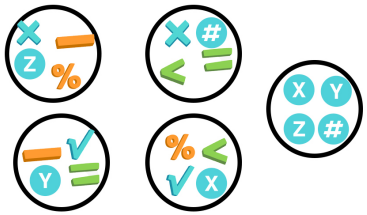
(a) 2 cartes



(b) 3 cartes



(c) 4 cartes



(d) 5 cartes

Problématique

- **Pour 55 cartes ?**
  On aura besoin de 54 symboles par carte

Contraintes :

- Un nombre de symboles par carte fixé
- Pas le même symbole en commun
- Le maximum de cartes possible

Problématique

En fixant le nombre de symboles n et le nombre de symboles par carte k , quel est le nombre maximal de cartes vérifiant (P) que l'on peut générer?

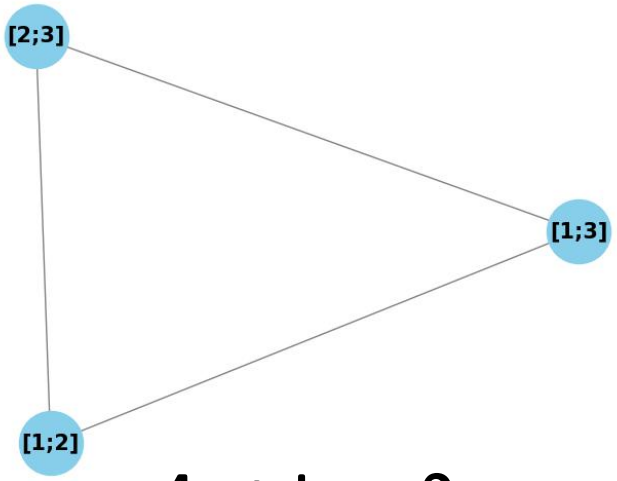
- Notations :
  - ***n* : nombre de symboles**
  - ***k* : nombre de symboles par carte**
  - **On représente les n symboles par les entiers de 1 à n**
  - **Une carte est une combinaison de k entier parmi n**

On modélise un jeu par un graphe G=(S,A) tel que :

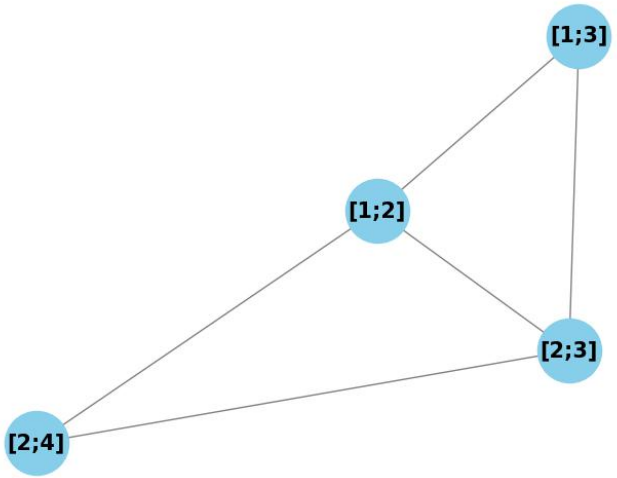
- L'ensemble des sommets S est l'ensemble des cartes
- Deux sommets sont reliés par une arête si et seulement si les deux cartes vérifient la propriété (P)

Exemple

- **Pour n = 3 et k = 2:**

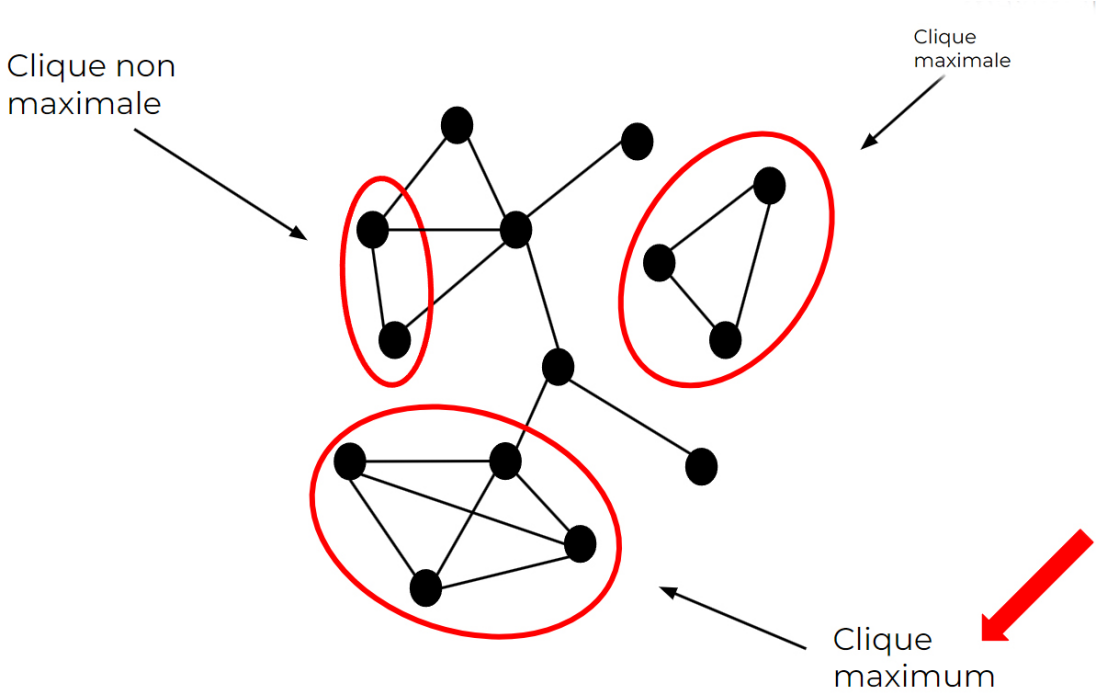


- **Pour n = 4 et k = 2:**



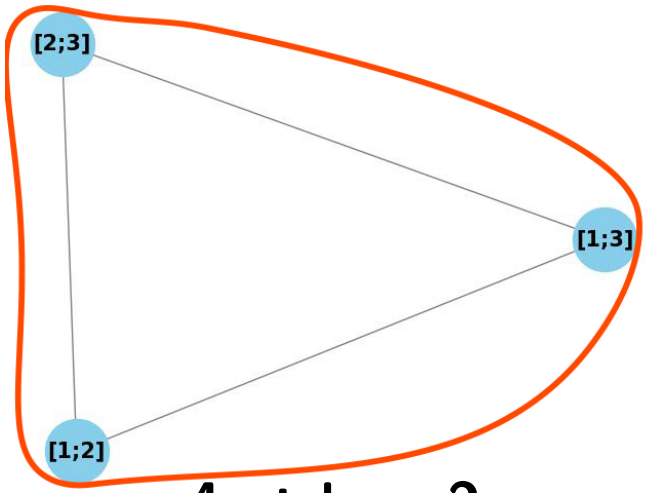
Lien avec les cliques

- Trouver le nombre maximal de cartes vérifiants *(P)* revient à trouver le maximum de sommets qui sont tous reliés entre eux par une arête
  ie : trouver la clique maximum du graphe



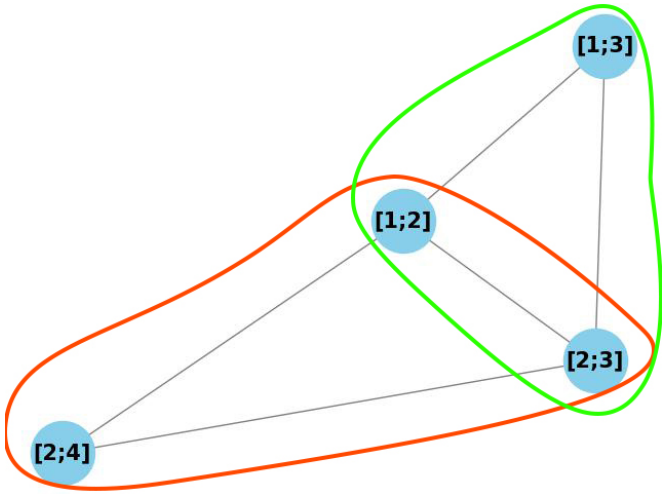
Exemple

- **Pour n = 3 et k = 2:**



- Taille du jeu : 3

- **Pour n = 4 et k = 2:**



- Taille du jeu : 3

Comment trouver la clique maximum ?

- **Utilisons la technique de retour sur trace (Backtracking) :**

- L'ensemble R est un sous-ensemble des sommets de la potentielle clique.
- L'ensemble P est l'ensemble des sommets candidats pour être ajoutés à la potentielle clique.
- L'ensemble X contient des sommets déjà traités ou appartenant déjà à une clique maximale.

1: **Fonction** $\textsc{EnumereClique}(R, P, X)$
2: **si** $P = \emptyset$ et $X = \emptyset$ **alors**
3: déclarer que R est une clique maximale
4: **fin si**
5: **pour chaque** sommet v dans P **faire**
6: $\textsc{EnumereClique}(R \cup \{v\}, P \cap N(v), X \cap N(v))$
7: $P \leftarrow P \setminus \{v\}$
8: $X \leftarrow X \cup \{v\}$
9: **fin pour chaque**
10:
11: $\textsc{EnumereClique}(\emptyset, S, \emptyset)$

- Lien du code C++ en annexe

Résultat de Moon et Moser (1962)

Chaque graphe de taille q a au plus $3^{\frac{q}{3}}$ cliques maximales.

Complexité

Le nombre total d'appels récursifs dans le pire cas est $O(3^{q/3})$, où q est le nombre de sommets du graphe.

- **Remarque :** On ne peut pas faire mieux qu'une complexité exponentielle. Le problème de recherche de clique maximale est un problème NP-Complet.

Première optimisation de l'algorithme

Propriété

- Si on note r le nombre de cartes et k le nombre de symboles 2 à 2 distincts sur chaque carte , alors :

$$r \leq k^2 - k + 1$$

- **Cette propriété permet d'arrêter le programme si le nombre maximal de cartes est atteint**

Optimisation basée sur les symétries du problème


```
2 8
taille : 7
1 2
1 3
1 4
1 5
1 6
1 7
1 8
```

(e) k = 2 et n = 8

```
3 6
taille : 4
1 2 3
1 4 5
2 4 6
3 5 6
```

(f) k = 3 et n = 6

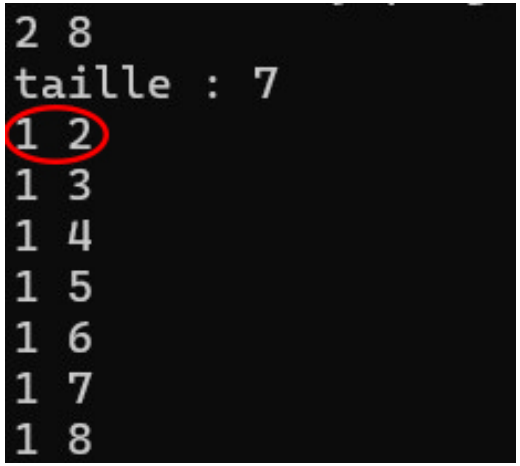
```
3 7
taille : 7
1 2 3
1 4 5
1 6 7
2 4 6
2 5 7
3 4 7
3 5 6
```

(g) k = 3 et n = 7

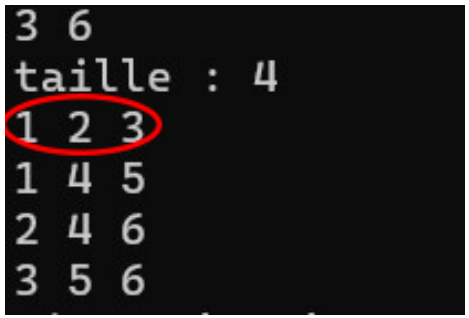
```
4 11
taille : 6
1 2 3 4
1 5 6 7
1 8 9 10
2 5 8 11
3 6 9 11
4 7 10 11
```

(h) k = 4 et n = 11

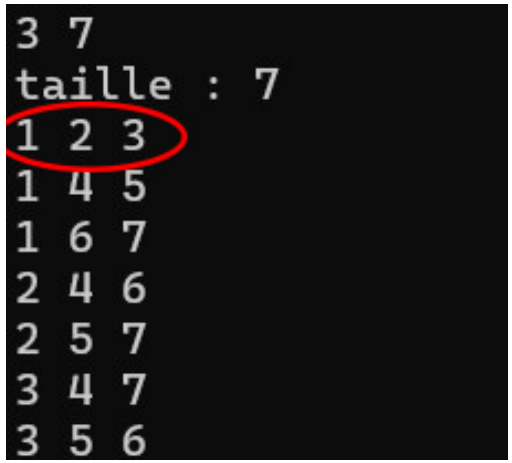
Observations ?



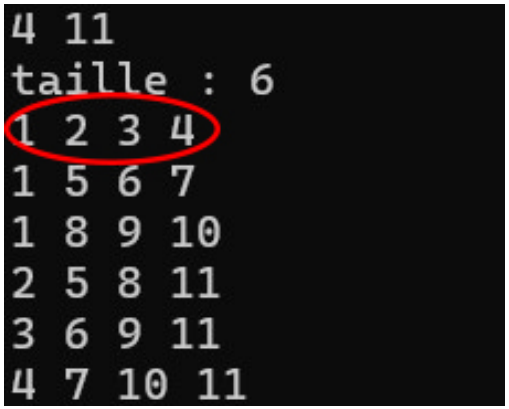
(a) k = 2 et n = 8



(b) k = 3 et n = 6



(c) k = 3 et n = 7



(d) k = 4 et n = 11

Remarque

Il existe toujours une clique maximum contenant la carte $[1, 2, ..., k]$

Conjecture

La symétrie du problème permet de conjecturer qu'en renumérotant les éléments dans tous les sommets d'une clique maximum, on peut montrer que **toute carte est contenue dans une clique maximum**

Exemple

- Pour n $= 6$ et k $= 3$:

Une clique possible est : $C_{id} = \{[2,3,4],[1,2,5],[1,3,6],[4,5,6]\}$

Renommant donc les sommets afin d'obtenir la carte $[1,2,3]$
On définit donc σ telle que :

$$\begin{cases} \sigma(2) = 1 \ ; \ \sigma(3) = 2 \\ \sigma(4) = 3 \ ; \ \sigma(5) = 5 \\ \sigma(6) = 6 \ ; \ \sigma(1) = 4 \end{cases}$$

Ainsi : $C_\sigma = \{[1,2,3],[4,1,5],[4,2,6],[3,5,6]\}$ est une clique maximum

- Soit C_{id} une clique maximum du graphe G utilisant comme symboles $c_1, c_2, ..., c_p$.
  Soit $[c_1, c_2, ..., c_k]$ une carte de C_{id}.
  Considérons la permutation σ définie par :

$$\begin{cases} \forall i \in \{1, 2, .., k\}, \sigma(c_i) = i \\ \forall i \in [k+1, p], \sigma(c_i) = \begin{cases} c_i \quad si \quad c_i \notin [1, k] \\ \min \{\{c_i \mid i \in [1, k] \ et \ c_i > k\} \setminus \{\sigma(c_j) \mid j \in [1, i-1]\}\} \end{cases} \end{cases}$$

- **Ainsi , en remplaçant chaque carte de C_{id} par son image par σ on définit une nouvelle clique maximum C_σ contenant la carte $[1, 2, ..., k]$**

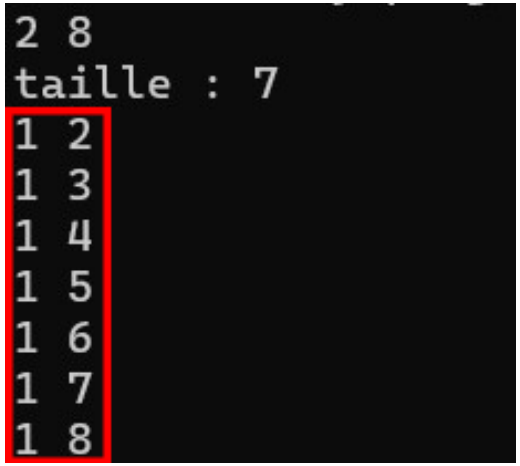
Optimisation ?

- En fixant un sommet dans notre clique , on évite du backtracking inutile.

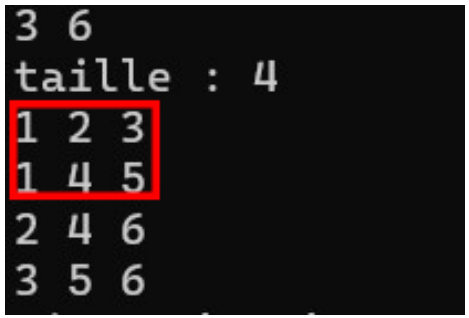
Résultat

La complexité de l'algorithme est divisée par $\binom{n}{k}$

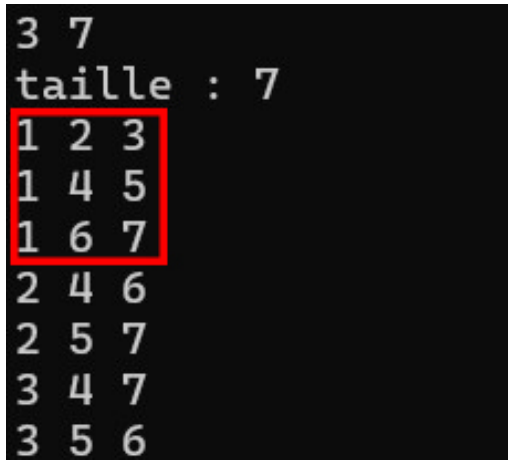
Autres observations ?



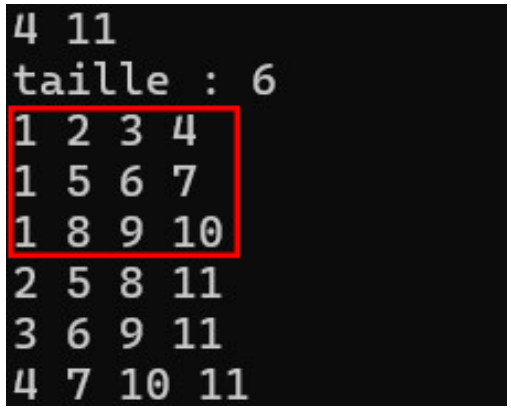
(e) k = 2 et n = 8



(f) k = 3 et n = 6



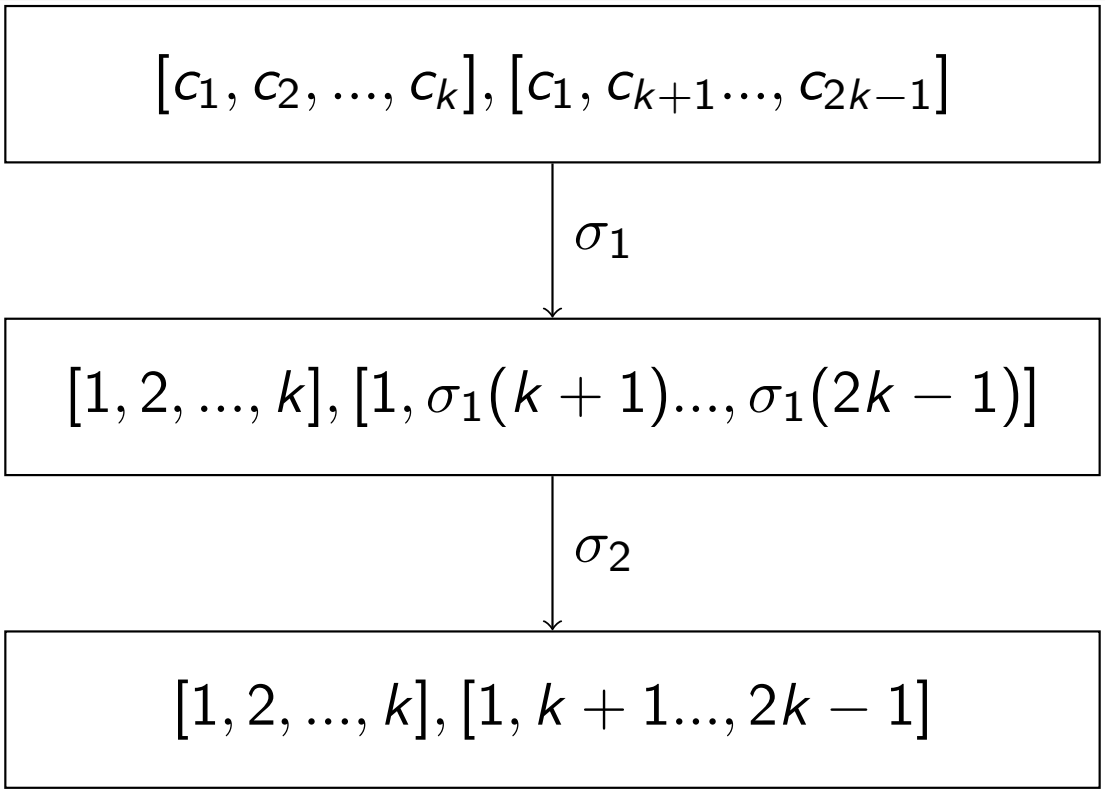
(g) k = 3 et n = 7



(h) k = 4 et n = 11

Un doublet de cartes ?

- On peut de la même façon étendre ce raisonnement pour un doublet de cartes
  ie : $[1, 2, ..., k], [1, k+1, ..., 2k-1]$
  Sous la condition : $2k - 1 \leq n$ (Preuve en annexe)



Une généralisation ?

Pour tout $q \in \mathbb{N}, si\ qk - (q-1) \leq n \Leftrightarrow k \leq \left\lfloor \frac{n+q-1}{q} \right\rfloor$
alors il existe une clique maximum contenant les cartes [1,..,k] , [1,k+1,..,2k-1],..., [1,(q-1)k,..,qk-(q-1)]

Résultat

La complexité est divisée par :

$$\binom{n}{k}^q$$

Conclusion

- Construction informatique de jeux de Dobble à l'aide de la recherche d'une clique
- Exploitation des symétries du problème pour optimiser la complexité temporelle
- **Quelques pistes de réflexion :**
  - -> Possibilité d'appliquer le même raisonnement pour des généralisations du jeu.
    Par ex : **2 cartes ont 2 et 2 UNIQUES symboles en commun.**
  - -> **Application à d'autres domaines** : cercle d'amis pour réseaux sociaux, organisation de plans de table de mariages...

Annexe : Démonstration $r \leq k^2 - k + 1$

- Pour $j \in [1, n]$ on note r_j le nombre de cartes contenant le symbole.
  On montre que : $r_j \leq k$

- Considérons un jeu de Dobble :
  En renommant les symboles on peut considérer une carte C avec les symboles 1 , 2 , . . . , k
  Notons, pour j $\in$ [| 1 , n |], Ej : l'ensemble des cartes avec le symbole j
  On a alors : $Card(E1 \cup E2 \cup \ldots \cup Ek) = r$
  De plus: $Card(E1 \cup E2 \cup \ldots \cup Ek) = r_1 + r_2 + \ldots + r_k + 1 - k$

  D'où : $r \leq k^2 - k + 1$

Annexe : Preuve 2 cartes

- Soit C_{id} une clique maximum du graphe G utilisant comme symboles $c_1, c_2, ..., c_p$.
- On réarrange les symboles afin d'obtenir les deux cartes $[c_1, c_2, ..., c_k], [c_1, c_{k+1}..., c_{2k-1}]$

Considérons les permutations σ_1 et σ_2 définies par :

- σ_1 (cf cas d'une seule carte)
- $\sigma_2 : \begin{cases} \sigma_2(1, .., k) = (1, .., k) & \\ \sigma_2(c_i) = i & si \;\; i \in [k+1, 2k-1] \\ \text{cf } \sigma_1 & \text{sinon} \end{cases}$

Fonction trouve clique maximum


```
1  void trouve_clique_maximum(ll int**
       graphe, int taille, vector<int>&
       currentClique, vector<int>
       candidats, vector<int>&
       cliqueMaximum, int&
       taille_clique_max) {
2      if(candidats.empty()){
3          if (currentClique.size() >
               taille_clique_max){
4              taille_clique_max =
                   currentClique.size();
5              cliqueMaximum =
                   currentClique;
6          }
7          return;
8      }
9      int candidat = candidats[0];
10     bool canAddCandidate = true;
11     for (int i = 0; i < currentClique.
           size(); i++) {
12         if (graphe[candidat][
               currentClique[i]] ==
               false) {
13             canAddCandidate = false;
14             break;
15         }
16     }
```

```
1      if(canAddCandidate){
2          currentClique.push_back(
               candidat);
3          vector<int> candidats_restants;
4          for(int i=1; i<candidats.size()
               ; i++){
5              if(graphe[candidat][
                   candidats[i]] == true
                   ){
6                  candidats_restants.
                       push_back(
                       candidats[i]);
7              }
8          }
9          trouve_clique_maximum(graphe,
               taille, currentClique,
               candidats_restants,
               cliqueMaximum,
               taille_clique_max);
10         currentClique.pop_back();
11     }
12     candidats.erase(candidats.begin());
13     trouve_clique_maximum(graphe,
           taille, currentClique,
           candidats, cliqueMaximum,
           taille_clique_max);
14 }
```

Fonction renvoie clique maximum


```
1  pair<int, vector<int>> renvoie_clique_max(ll int** graphe, int taille) {
2      vector<int> clique_max;
3      int taille_clique_max = 0;
4      vector<int> currentClique;
5      vector<int> candidats;
6      for(int i=0; i < taille; i++){
7          if(graphe[0][i] == 1){
8              candidats.push_back(i);
9          }
10     }
11     trouve_clique_maximum(graphe, taille, currentClique, candidats, clique_max,
           taille_clique_max);
12     return make_pair(taille_clique_max, clique_max);
13 }
```

Fonction une intersection


```
1 bool une_intersection(vector<int>& v, vector<int>& data) {
2     int cnt = 0;
3     for(int i=0; i<data.size(); i++){
4         if(find(v.begin(), v.end(), data[i]) != v.end()) {
5             cnt++;
6         }
7     }
8     return (cnt == 1);
9 }
```

Fonction combinaisons


```
1  void combinaisons(int arr[], int n, int r, int index, vector<int> data, int i, vector<
       vector<int>> &res, vector<int>& carte) {
2      if (index == r) {
3          if (une_intersection(carte, data)) {
4              res.push_back(data);
5          }
6          return;
7      }
8
9      if (i >= n)
10         return;
11
12     data[index] = arr[i];
13     combinaisons(arr, n, r, index + 1, data, i + 1, res, carte);
14     combinaisons(arr, n, r, index, data, i + 1, res, carte);
15 }
```

Fonction matrice


```
ll int** matrice(int taille, vector<vector<int>>& cartes) {
    ll int** matrice = new ll int*[taille];
    for(int i = 0; i < taille; i++){
        matrice[i] = new ll int[taille];
    }
    for(int i = 0; i < taille; i++){
        for(int j = 0; j < taille; j++){
            if(une_intersection(cartes[i], cartes[j]) == true)
                matrice[i][j] = 1;
            else
                matrice[i][j] = 0;
        }
        matrice[i][i] = 1;
    }
    return matrice;
}
```

Fonction main


```
int main() {
  time_t start, end;
  time(&start);
  ios_base::sync_with_stdio(false);
  int n ,k;
  cin >> k >> n;
  ;
  int arr[n];
  for (int i = 0; i < n; i++)
    arr[i] = i+1;

  vector<int> data(k);
  vector<vector<int>> res;
  vector<int> carte;
  for(int i=0;i<k;i++){
    carte.push_back(i+1);
  }
  res.push_back(carte);
  combinaisons(arr, n, k, 0, data, 0,
       res,carte);
```

```
    int taille = res.size();
  vector<vector<int>> vadj =
       vecteur_adjacence(taille,res);
  pair<int,vector<int>> Clique_max =
       renvoie_clique_max2(vadj,taille
       );
  cout << "taille : " << Clique_max.
       first << endl;
  for (int i = 0; i < Clique_max.second
       .size(); i++) {
     for(int j=0;j<k;j++){
          cout << res[Clique_max.
               second[i]][j] << " ";
      }
      cout << endl;
    }
    time(&end);
    double time_taken = double(end -
         start);
     cout << "Time taken by program is
          : " << fixed
      << time_taken << setprecision
           (5)<<endl;
  return 0;
}
}
```