Algorithmique avancée DO5

Marin Bougeret
marin.bougeret@lirmm.fr

POLYTECH
MONTPELLIER

Outline

1. Administration !
2. Contexte
3. Programmation Dynamique (DP)
4. Application au Sac à dos (KP: KnaPsack)
5. Application au plus court chemin

Plan du module S1

Volume

- programmation dynamique : environ 6h
- complexité paramétrée : environ 4,5h
- examen sur programmation dynamique et complexité paramétrée
- algorithme de page Rank : 2 séances + examn (fait par Alexandre Pinlou)

Notation

Un exam final (avec questions de cours + exos style TD, une fuille A4 recto/verso de note manuscrites autorisée).

Organisation

Ressources pédagogiques sur l'ENT

Outline

1. Administration !
2. Contexte
3. Programmation Dynamique (DP)
4. Application au Sac à dos (KP: KnaPsack)
5. Application au plus court chemin

Problème d'optimisation Vs Problème de décision

Les "problèmes de nature combinatoire" que nous abordons dans ce cours sont de deux natures : optimisation ou décision.

Définition d'un problème de décision Π:

- entrée : instance $I \in \mathcal{I}$
- question : décider si I vérifie propriété p

3-SAT :

- entrée : une formule3 SAT F (ex $F = (x_1 \vee x_3 \vee \bar{x_2}) \wedge (\bar{x_1} \vee x_4 \vee \bar{x_2}) \wedge (...)$)
- question : décider si F est satisfiable

Problème d'optimisation Vs Problème de décision

Les "problèmes de nature combinatoire" que nous abordons dans ce cours sont de deux natures : optimisation ou décision.

Définition d'un problème de décision Π:

- entrée : instance $I \in \mathcal{I}$
- question : décider si I vérifie propriété p

3-COL :

- entrée : un graphe G
- question : décider si G est 3 colorable (si on peut colorer les sommets de G avec 3 couleurs de telle sorte que les sommets adjacents aient des couleurs différentes)

Problème d'optimisation Vs Problème de décision

Les "problèmes de nature combinatoire" que nous abordons dans ce cours sont de deux natures : optimisation ou décision.

Définition d'un problème de décision Π:

- entrée : instance $I \in \mathcal{I}$
- question : décider si I vérifie propriété p

VC_{dec} :

- entrée : un graphe G, un entier k
- question : décider si G admet un vertex cover de taille k (c'est à dire un ensemble S de k sommets tel que pour tout $e \in E, e \cap S \neq \emptyset$)

Problème d'optimisation Vs Problème de décision

Définition d'un problème d'optimisation Π:

- entrée : instance $I \in \mathcal{I}$
- sortie : une solution S faisable (c'est à dire vérifiant des contraintes C)
- fonction objectif : maximiser/minimiser $c(I, S)$ (à valeur positive)

Notation : $opt(I) = max_{S \text{ faisable}} c(I, S)$ (pour un problème de maximisation)

Rmq : attention, on évite de parler de "la solution optimale" : il peut y avoir plusieurs S atteignant $opt(I)$ (mais la valeur $opt(I)$ elle est bien unique).

Problème d'optimisation Vs Problème de décision

VC :

- entrée : un graphe $G = (V, E)$
- sortie : un sous ensemble de sommets S tel que pour tout $e \in E, e \cap S \neq \emptyset$
- fonction objectif : minimiser $|S|$ (ici $c(I, S) = |S|$)

Donner exemple de S, et de $opt(I)$.

Rmq très souvent c ne dépend que de la solution, mais parfois cela n'est pas le cas, par exemple :

VC pondéré :

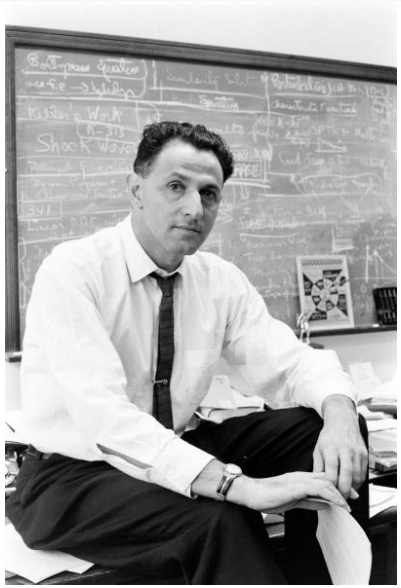
- entrée : un graphe $G = (V, E)$, un poids w_v pour tout $v \in V$
- sortie : un sous ensemble de sommets S tel que pour tout $e \in E, e \cap S \neq \emptyset$
- fonction objectif : minimiser $\sum_{v \in S} w_v$ (ici $c(I, S)$ dépend bien de S et de I)

Outline

1. Administration !
2. Contexte
3. Programmation Dynamique (DP)
4. Application au Sac à dos (KP: KnaPsack)
5. Application au plus court chemin

Histoire et Contexte

- la programmation dynamique est une technique générale permettant de réduire drastiquement (typiquement de exponentiel à polynomial) la complexité d'un algorithme récursif.
- c'est une technique extrêmement répandue (algos poly, FPT, d'approximation..) et très puissante!
- elle permet de calculer des fonctions, résoudre des problèmes d'optimisation, de décision..



- Dans les années 50 Richard Bellman choisit le terme programmation dynamique pour plaire à son supérieur
- .. mais cela n'a pas grand chose à voir avec le dynamisme ;)

Exemple 1 : Fibonacci

Définition de la suite de Fibonacci :

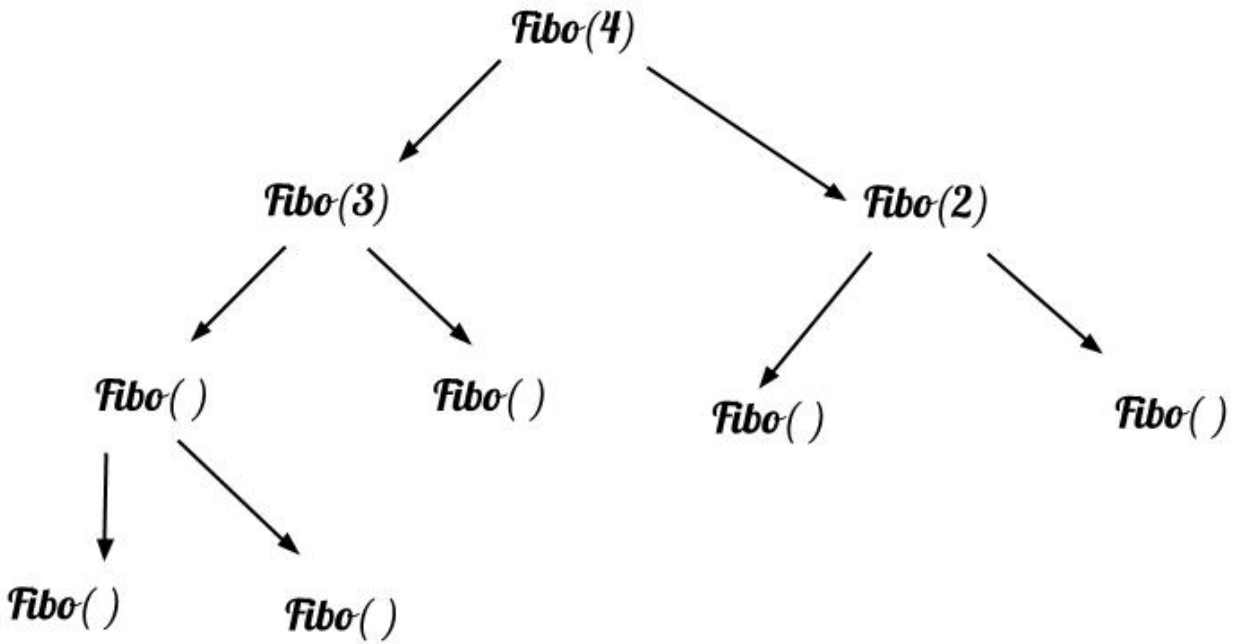
$$F(0) = 1;\ F(1) = 1; \quad F(n) = F(n-1) + F(n-2)$$

```
int Fib(n){
  if (n<=1) return 1;
  else return Fib(n-1)+Fib(n-2);
}
```

Complexité : $Fib(n)$ est exponentielle en n.
Cause : Multiplicité de calcul d'un même nombre

Où est le problème ?

La figure représente le calcule de l'algorithme récursif de Fib(4). Completez-la et marquez les calculs faits en double



Clé : mémoïsation

solution

- pour éviter de refaire des calculs déjà faits, mémoriser les résultats à l'aide d'un tableau.
- étant donné n, on déclare (à l'exterieur de la fonction) un tableau t de n cases qu'on initialise à $-\infty$
- puis, on appelle FibDP (Dynamic Programming)

Clé : mémoïsation

```
int FibClient(n){
    t = new int[n]; //var globale
    pour tout i, t[i]= -infini;
    return FibDP(n);
}

int FibDP(n){
  if(t[n]==-inf){
    int res;
    ancien code en remplacant (return truc ;)
        par (res = truc;)
    t[n]=res;
  }
  return t[n];
}
```

On peut prouver facilement que FibDP(n) = Fib(n)

Clé : mémoïsation

```
int FibClient(n){
    t = new int[n]; //var globale
    pour tout i, t[i]= -infini;
    return FibDP(n);
}

int FibDP(n){
  if(t[n]==-inf){
    int res;
    if (n<=1) res = 1;
    else res = FibDP(n-1)+FibDP(n-2);
    t[n]=res;
  }
  return t[n];
}
```

On peut prouver facilement que FibDP(n) = Fib(n)

Complexité d'une DP

Théorème : temps

La complexité d'une DP (qui termine) basée sur un tableau t est

$$\mathcal{O}(\mathit{taille}(t) \times c)$$

avec c la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

Autrement dit, on compte comme si l'on calculait chaque case du tableau une seule fois, et que les appels récursifs comptaient $\mathcal{O}(1)$

Complexité d'une DP

Théorème : temps

La complexité d'une DP (qui termine) basée sur un tableau t est

$$\mathcal{O}(\mathit{taille}(t) \times c)$$

avec c la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

Autrement dit, on compte comme si l'on calculait chaque case du tableau une seule fois, et que les appels récursifs comptaient $\mathcal{O}(1)$
Preuve : dessin :)

Complexité d'une DP

Théorème : temps

La complexité d'une DP (qui termine) basée sur un tableau t est

$$\mathcal{O}(taille(t) \times c)$$

avec c la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

Thoérème : temps (version raffinée)

La complexité d'une DP (qui termine) basée sur un tableau t est

$$\mathcal{O}(\sum_{\text{i case de t}} c(i))$$

avec $c(i)$ la complexité de l'appel qui remplit la case i en comptant $\mathcal{O}(1)$ pour chaque appel recursif.

Application à FibDP

```
int FibDP(n){
  if(t[n]==-inf){
    int res;
    if (n<=2) res = 1;
    else res = FibDP(n-1)+FibDP(n-2); //O(1)
    t[n]=res;
  }
  return t[n];
}
```

Complexité de FibDP(n): taille(t)=$\mathcal{O}(n)$, $c = 1 \Rightarrow \mathcal{O}(n)$
Mémoire utilisée : $\mathcal{O}(n)$

Pour la suite

Le seul travail à faire est donc de trouver une (bonne) façon récursive de résoudre le problème. Attention, on écrira plus la instructions liées au "tableau de marquage".

Outline

1. Administration !
2. Contexte
3. Programmation Dynamique (DP)
4. Application au Sac à dos (KP: KnaPsack)
5. Application au plus court chemin

Sac à dos (KP: KnaPsack)

Définition

- entrée : un entier C, et deux tableaux d'entiers p, v de taille n
  - C représente la taille du sac à dos
  - l'objet i occupe une place $p[i]$ dans le sac, mais rapporte une valeur $v[i]$
- sortie : un sous ensemble S d'objets tenant dans le sac ($p(S) \leq C$, avec $p(S) = \sum_{i \in S} p[i]$)
- fonction objectif : maximiser $v(S)$, avec $v(S) = \sum_{i \in S} v[i]$

- Ce problème est NP difficile
- Brute force (essayer tous les sous ensembles) : couteraît (au moins) 2^n
- On va le résoudre en $\mathcal{O}(nC)$
- Ca n'est que pseudo polynomial (il faudrait $log(C)$ pour être polynomial), mais c'est tout de même beaucoup mieux!

Ex 2: Sac à dos (KP: KnaPsack)

Prendre ou ne pas prendre, là n'est pas la question (on essaye les 2!)

- on considère le premier objet ..
- si il est trop gros, on passe au suivant
- sinon, on ne sait pas à priori si il faut le prendre ..
- on essaye donc les deux :
  - si on le prend, on gagne $v[0]$, mais il reste $c = C - p[0]$, et on passe à l'objet 1
  - sinon, on passe à l'objet 1

D'où la spécification (pour l'instant on calcule juste la valeur max)

```
int sac(int c,int i){
// prerequis
  // 0 <= c <= C
  // 0 <= i <= n (n: nb d'objets)
// action
  // calcule la valeur maximale qu'on peut
     mettre dans un sac de taille c avec les
     objets >= i
```

Ex 2: Sac à dos (KP: KnaPsack)

```
int sac(int c,int i){
  if(i==n){ return 0}
  else
    if(p[i] > c){
      return sac(c,i+1)
    }
    else{
      return max(v[i]+sac(c-p[i],i+1),sac(c,i
          +1))
    }
}
```

Ex 2: Sac à dos (KP: KnaPsack)

Allez, une dernière fois : DP associée

```
int sacDP(int c,int i){
  if(t[c,i]==-inf){
    int res;
    if(i==n){ res= 0}
    else{
      if(p[i] > c){
        res = sacDP(c,i+1)
      }
      else{
        res = max(v[i]+sacDP(c-p[i],i+1),sacDP
            (c,i+1))
      }
    }
    t[c,i] = res;
  }
  return t[c,i];
}
```

Preuve d'optimalité

Correction de sac

sacDP retourne la même valeur que sac, étudions donc sac !
Pourquoi sac retourne bien la valeur annoncée ?

Notations

- Soit E l'ensemble des entrées (c, i) possibles de l'algo
- Soit $Sol(c, i)$ l'ensemble des S tq $S \subseteq \{i, \ldots, n\}$ et $p(S) \leq c$

Problème KP-aux

On suppose n, C, p, v fixés.

- entrée : $(c, i) \in E$
- sortie : un $S \in Sol(c, i)$
- objectif : maximiser $v(S)$

Remarques :

- KP-aux est bien un problème de maximisation "classique"
- $opt(c, i)$ denote $\max_{S \in Sol(c,i)} v(S)$

Preuve d'optimalité

Prouvons par rec sur i que $\forall (c, i) \in E$, $sac(c, i) = opt(c, i)$.

Cas 1 (le plus dur) : si $p[i] \leq c$.

Soit $S^* \in Sol(c, i)$ une sol optimale ($p(S^*) = opt(c, i)$).
Cas 1.1 si $i \in S^*$.

- $v(S^*) = v[i] + v(S^* \setminus \{i\})$
- $sac(c, i) \geq v[i] + sac(c - p[i], i + 1)$ (car on garde le max)

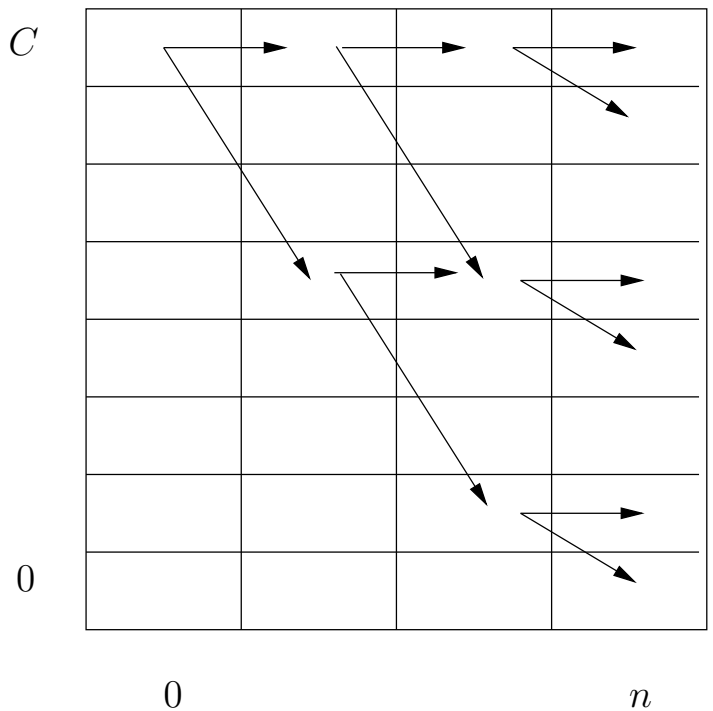
But : $sac(c - p[i], i + 1) \geq v(S^* \setminus \{i\})$

- par rec, $sac(c - p[i], i + 1) \geq opt(c - p[i], i + 1)$
- $S^* \setminus \{i\}$ est une sol faisable de $(c - p[i], i + 1)$
- donc $v(S^* \setminus \{i\}) \leq opt(c - p[i], i + 1)$ (on a même l'égalité, mais pas utile)

Cas 1.2 si $i \notin S^*$ même idée

D'après le Théorème, sacDP s'execute en temps $(C+1)(n+1) \times \mathcal{O}(1) = \mathcal{O}(Cn)$.

Que se passe t-il vraiment quand on lance sacDP(C,0) avec $C = 7$, $n = 3$, $p[0] = 3, p[1] = 3, p[2] = 1$?



Complexité

D'après le Théorème, sacDP s'execute en temps $(C+1)(n+1) \times \mathcal{O}(1) = \mathcal{O}(Cn)$.

Que se passe t-il vraiment quand on lance sacDP(C,0) avec $C = 7$, $n = 3$, $p[0] = 3, p[1] = 3, p[2] = 1$?

- $\mathcal{O}(Cn)$ est donc un pire cas (comme si on avait calculé une fois toutes les cases), en pratique on en fait moins (mais on a du mal à dire combien exactement)
- si l'on devait éxecuter à la main .. on utiliserait aussi un tel tableau pour éviter de recalculer les appels !

Ex 2: Sac à dos (KP: KnaPsack)

Remarques

On a l'impression de faire un brute force .. pourquoi intuitivement a-t-on une complexité faible ? Car on a réussi à encoder efficacement la situation restante après chaque choix.

Ex de mauvais encodage : se rappeler de tous les objets j<i que l'on a déjà pris

```
int sac(list l,int i){
  //prerequis :
    // 0 <= i <= n (n: nb d'objets)
    // l ne contient que des j < i
    // le sac contient deja tous les objets de
        l
  //action :
    // calcule la valeur maximale qu'on peut
       mettre dans la place restante avec les
       objets >= i
```

Ex 2: Sac à dos (KP: KnaPsack)

Remarques

Intuitivement : le (bon) encodage est efficace car de nombreuses "trajectoires" aboutissent à la même situation. Par exemple, on peut avoir :

```
C->C-p[0]->C-p[0]-p[2]->C-p[0]-p[2]-p[5] = c
C-> ..   -> ..     ->C-p[0]-p[2]-p[3]-p[4] = c
C-> ..   -> ..           ->C-p[0]-p[1]-p[4] = c
   ...
C-> ..   -> ..           ->C-p[3]-p[4]-p[5] = c
```

Rmq 1 : obtenir aussi une solution

Comment obtenir une solution (and not only la valeur optimale) ?

Réecrire l'algorithme pour obtenir une solution est en général trivial.

Exemple

```
list sacV2(int c,int i){
  if(i==n){ return []}

  if(p[i] > c){
   return sacV2(c,i+1)
  }
  else{
    l1 = {i} U sacV2(c-p[i],i+1);
    l2 = sacV2(c,i+1);
    if(v(l1) > v(l2))
      return l1;
    else
      return l2;
  }
}
```

Rmq 2 : dérecursification

On peut déduire d'une DP un algorithme itératif qui remplit le tableau "dans le bon ordre".
On peut ensuite généralement réduire la mémoire utilisée en n'utilisant qu'un tableau plus petit.

Rmq 2 : dérecursification

Par exemple pour sac : (penser au tableau qui se remplit colonne par colonne en partant de la droite plutôt que de lire ce code!)

```
void sacIteV1(C,p,v){
  declarer t de taille nC
  for(i from n to 0)
    for(c from 0 to n)
      if(i==n){ t[c,i]=0}
      else{
        if(p[i] > c){
          t[c,i]=t[c,i+1]
        }
        else{
          t[c,i]=max(v[i]+t[c-p[i],i+1],t[c,i
              +1]);
        }
      }
}
```

Complexité : $\mathcal{O}(nC)$
Mémoire utilisée : $\mathcal{O}(nC)$

Rmq 2 : dérecursification

V2 : 2 tableaux de taille C seulement : on a besoin que de la colonne courante et de celle juste à sa droite.

```
void sacIteV2(C,p,v){
  declarer tprec et tcour de taille C
  init tprec avec 0
  for(i from n-1 to 0)
    for(c from 0 to n)
      if(p[i] > c){
          tcour[c,i]=tprec[c,i+1]
        }
        else{
          tcour[c,i]=max(v[i]+tprec[c-p[i],i
              +1],tprec[c,i+1]);
        }
      }
      tprec <- tcour
}
```

Complexité : $\mathcal{O}(nC)$
Mémoire utilisée : $\mathcal{O}(C + n)$

DP Récursive Vs ou DP Iterative ?

DP Itérative

- Avantage : on gagne en mémoire
- Inconvénients : on est sûr de remplir l'équivalent de tout le tableau, et il faut réfléchir pour trouver le bon ordre de remplissage

DP Récursive

- Avantage : il faut juste écrire l'algo récursif (l'ajout du tableau est trivial), on ne calculera que les cases dont on aura besoin (mais il faut tout de même initialiser le tableau!)
- Inconvénients : place mémoire

Ce qu'il faut retenir DP = Rec + tableau. La version itérative n'est qu'une petite optimisation.

Rmq3 : DP basée sur une structure de donnée quelconque

Remarque

Au lieu de prendre un tableau pour t, on peut prendre n'importe quelle structure de donnée (liste, hashtable, ..).
Avantages possibles :

- taille de t plus petite (égale au nombre d'entrées réellement calculées)
- initialisation de t plus rapide

Inconvénients possibles :

- temps d'ajout/de recherche d'une entrée plus long

- Le théorème sur le temps d'une DP serait alors énnoncé en fonction des performances de cette structure de donnée.
- En pratique, une hashtable est souvent très bien!

Outline

1. Administration !
2. Contexte
3. Programmation Dynamique (DP)
4. Application au Sac à dos (KP: KnaPsack)
5. Application au plus court chemin

Exemple 2 : Déterminer le plus court chemin dans un graphe

Nous allons étudier l'algorithme de Bellman-Ford pour le plus court chemin.

Imho, ce n'est pas le meilleur exemple pour se familiariser avec les DP (le sac à dos est beaucoup mieux).

Mais .. c'est une tradition ! (et puis on est un peu obligés non ? il faut rendre hommage à Monsieur Bellman!)

Exemple 2 : Déterminer le plus court chemin dans un graphe

Soit $G = (V, A)$ un graphe, où V est l'ensemble des sommets et A l'ensemble des arcs.

- Le poids de l'arc a est un entier naturel $l(a)$.
- La longueur d'un chemin P est égale à la somme des longueurs des arcs qui les composent (notée $l(P)$)

Problème SP (shortest-path)

- entrée : un graphe G, et deux sommet s et y
- sortie : un s-y chemin P
- objectif : minimiser $l(P)$

Remarque : pour l'instant, on suppose qu'il n'y a pas de cycles négatifs (sinon problème mal défini .. pourquoi ?)

Exemple 2 : Déterminer le plus court chemin dans un graphe

- fixons s, et partons de y, en cherchant à reculer vers s
- c'est à dire considérons le problème suivant

Problème SP-aux

G et s sont fixés.

- entrée : un sommet y
- sortie : un s-y chemin P
- objectif : minimiser $l(P)$

Exemple 2 : Déterminer le plus court chemin dans un graphe

- fixons s, et partons de y, en cherchant à reculer vers s
- c'est à dire considérons le problème suivant

Problème SP-aux

G et s sont fixés.

- entrée : un sommet y
- sortie : un s-y chemin P
- objectif : minimiser $l(P)$

- quel prédecesseur de y devrait on utiliser ?
- on ne sait pas ! Donc on branche ! D'où la tentative d'algo suivant

Exemple 2 : Déterminer le plus court chemin dans un graphe

```
int A(y){
  //A(y) calcule opt(y)
  if(y==s) return 0;
  else{
    let P(y) = ensemble des predecesseurs de y
    return min_{z \in P(y)}(A(z)+l(zy))
  }
}
```

Quel est le problème de A ?

- boucle infinie potentielle (à cause des cycles) !
- il n'y a rien qui devient "plus petit" lors des appels récursifs

Solution ?

- on s'impose un nombre maximum k d'arêtes autorisées ..
- .. et c'est ce paramètre qui va diminuer dans les appels récursifs

Exemple 2 : Déterminer le plus court chemin dans un graphe

Problème SP-aux-V2

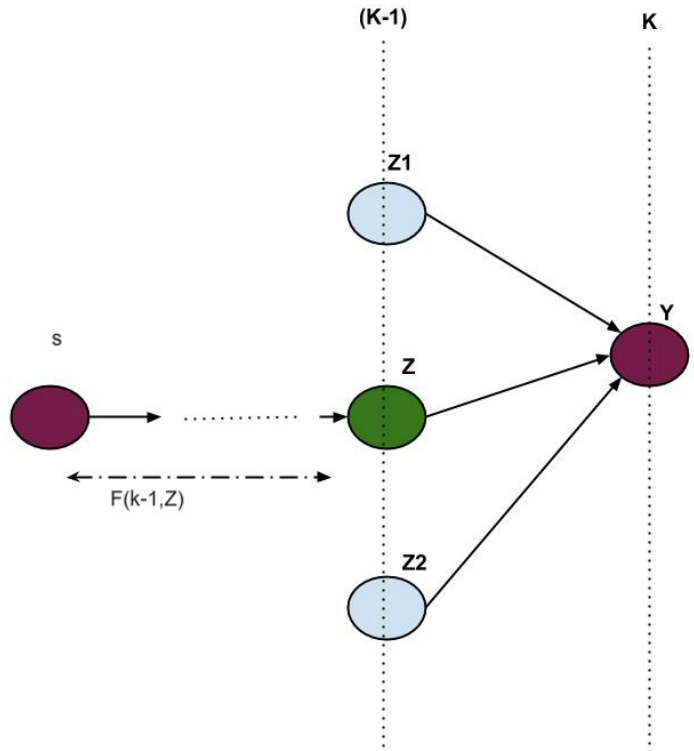
G et s sont fixés.

- entrée : (y, k) avec y un sommet, et k un entier $(0 \leq k < |V(G)|)$
- sortie : un s-y chemin P ayant au plus k arêtes
- objectif : minimiser $l(P)$

Si on sait résondre SP-aux-V2, alors il suffira de demander à le résoudre sur $(y, n-1)$ pour avoir le s-y chemin le plus court (car un tel chemin utilise bien au plus $n-1$ arêtes).

Exemple 2 : Déterminer le plus court chemin dans un graphe

```
void A2(k,y){
 //A2(k,y) calcule opt(k,y)
 if(k==0){
   if(y==s) return 0;
   else return infini;
 }
 else{ //k != 0
   if(y==s) return 0 (car pas
        cycle negatifs)
   else
     let P(y) = ensemble des
         predecesseurs de y
     return min_{z \in P(y)}(
         A2(k-1,z)+l(zy))
   }
}
```



Complexité (taille tab $= n^2$, $c = n$) $\mathcal{O}(n^3)$.

Exemple 2 : Déterminer le plus court chemin dans un graphe

Une meilleure analyse de complexité en utilisant le théorème raffiné

Soit $c(k, y)$ le temps de calcul de $A2(k, y)$ (en comptant $\mathcal{O}(1)$ pour les appels recs). On peut majorer plus finement $c(k, y)$ (que simplement n) :

- on précalcule tous les $P(y)$ (en $\mathcal{O}(n + m)$)
- $c(k, y) = \mathcal{O}(|P(y)|)$
- Théorème : temps $= \sum_k \sum_y c(k, y) = \sum_k \mathcal{O}(m) = \mathcal{O}(nm)$

Conclusion

A retenir

- DP = algo recursif + tableau (même si dérecursifiable)
- Complexité = #Cases x temps d'un appel (même si impression de brute force)
- Pour quels problèmes la DP est-t-elle un bon outil ? Situations caractéristiques où la DP est un bon outil :
  - la structure dans laquelle on cherche une solution à un ordre naturel pour la parcourir (graphe d'intervalles : de gauche à droite, arbre : de la racine vers les feuilles, pb du sac à dos : la liste d'objets, ..)
  - quand de nombreuses trajectoires (une trajectoire étant une suite de décisions locales, par pour KP, traj = $d_1, .., d_i$, avec d_i vrai ssi on prend l'objet i) différentes peuvent aboutir au même état