IFT2125 - Introduction à l'algorithmique

Algorithmes voraces (gloutons) (B&B chapitres 5-6)

Pierre McKenzie

DIRO, Université de Montréal

Hiver 2018

Structures de données

À parcourir par vous-mêmes si cette matière est nouvelle pour vous :

Chapitre 5, Data structures

- Section 5.1 : Arrays, stacks and queues
- Section 5.2 : Records and pointers
- Section 5.3 : Lists
- Section 5.4 : Graphs
- Section 5.5 : Trees
- Section 5.7 : Heaps

are to be read on your own if you have not come across that material before.

Les algorithmes voraces (greedy algorithms)

- Notre première grande classe d'algorithmes.
- S'applique à une forme de problèmes d'optimisation.

Le type de problème à résoudre

Identifier, parmi un ensemble de candidats disponibles, un sous-ensemble formant une solution, optimale parmi les solutions possibles.

Exemples :

- Ensemble de poids maximum de vecteurs linéairement indépendants
- Ensemble d'arêtes formant un arbre sous-tendant minimal
- Le moins de pièces de monnaie possible totalisant un montant donné
- Valeur maximale du butin à emporter lors d'un cambriolage
- Chemin de longueur minimale entre 2 sommets d'un graphe
- Temps moyen minimum pour n tâches sur un processeur
- Sous-graphe 3-coloriable maximal d'un graphe

L'idée est simple : on a faim on bouffe !

Un candidat ajouté n'est jamais écarté ensuite

```
function greedy(C: set): set
    {C is the set of candidates}
    S ← ∅ {We construct the solution in the set S}
    while C ≠ ∅ and not solution(S) do
        x ← select(C)
        C ← C \ {x}
        if feasible(S ∪ {x}) then S ← S ∪ {x}
    if solution(S) then return S
                   else return "there are no solutions"
```

Composantes fréquentes d'un algorithme vorace

Fonction de sélection

Fonction objective : est-ce que S forme déjà une solution?

function *greedy*(C: *set*): *set*
    {C is the set of candidates}
    $S \leftarrow \varnothing$ {We construct the solution in the set S}
    while $C \neq \varnothing$ **and not** *solution*(S) **do**
        $x \leftarrow$ *select*(C)
        $C \leftarrow C \setminus \{x\}$
        if *feasible*($S \cup \{x\}$) **then** $S \leftarrow S \cup \{x\}$
    if *solution*(S) **then return** S
            else return "*there are no solutions*"

Fonction de réalisabilité

Ensemble de candidats choisis

Composantes

La fonction de sélection :

- Souvent le prochain candidat de la liste triée des candidats non encore considérés

La fonction de réalisabilité :

- Centrale à l'algorithme
- Son implantation détermine souvent la performance

La fonction objective :

- Souvent naturelle et facile à déduire du problème

Exemple

Donnée : vecteurs $v_1, \ldots, v_m \in \mathbb{R}^d$, avec poids $P[1], \ldots, P[m] \in \mathbb{R}^+$
Déterminer : ensemble de vecteurs lin. indép. de poids total maximum

- candidats : vecteurs
- fonction de réalisabilité : test d'indépendance linéaire
- fonction de sélection $= ?$

Exemple (suite)

L'algorithme

```
fonction Vecteurs(V, P[1..|V|]) : ensemble de vecteurs
      { P[a] est le poids du vecteur a }
      L[1..|V|] ← liste des vecteurs triée en ordre de poids décroissants
      S ← ∅
      tantque L non vide faire
            a ← retirer prochain vecteur de L
            si S ∪ {a} lin. indép. alors S ← S ∪ {a}
      retourner S
```

- Fonctionne ou pas ?
- Aucune fonction objective requise, ok ?

Matroïde

Définition (also in *Algorithms* by Cormen-Leiserson-Rivest)

Soit C un ensemble fini de candidats et $I \subseteq \{S : S \subseteq C\}$.

(C, I) est un matroïde ssi pour tout $X, Y \subseteq C$:

- la trivialité : $\emptyset \in I$
- l'hérédité : $[X \in I$ et $Y \subseteq X] \Rightarrow Y \in I$
- l'échange : $[X \in I$ et $Y \in I$ et $|X| < |Y|] \Rightarrow (\exists a \in Y \setminus X)[X \cup \{a\} \in I]$.

Matroïde

Définition (la même, mais cette fois en mots...)

Soit C un ensemble fini et $I = \{S_1, \ldots, S_k\}$ où les $S_i \subseteq C$.

(On appelle "indépendants" les ensembles S_i.)

(C, I) est un matroïde ssi pour tout $X, Y \subseteq C$:

- trivialité : l'ensemble vide est indépendant
- hérédité : tout sous-ensemble d'un indépendant est indépendant
- échange : si la taille d'un indépendant Y dépasse celle d'un indépendant X, alors on peut toujours trouver dans Y un candidat à ajouter à X et rester indépendant.

Matroïde

Exemples

1. $C = \{$ boeuf, fourmi, renard, grenouille, cigalle, corbeau $\}$
   $I =$ ensemble des sous-ensembles de C possédant 3 éléments ou moins

Matroïde

Exemples

1. $C = \{$ boeuf, fourmi, renard, grenouille, cigalle, corbeau $\}$
   $I =$ ensemble des sous-ensembles de C possédant 3 éléments ou moins

2. C ensemble de vecteurs
   $I =$ ensemble des sous-ensembles de C linéairement indépendants

Matroïde

Exemples

1. $C = \{$ boeuf, fourmi, renard, grenouille, cigalle, corbeau $\}$
   $I =$ ensemble des sous-ensembles de C possédant 3 éléments ou moins

2. C ensemble de vecteurs
   $I =$ ensemble des sous-ensembles de C linéairement indépendants

3. C l'ensemble des arêtes d'un graphe donné
   $I =$ ensemble des sous-ensembles d'arêtes ne créant pas de cycle
   Requiert une preuve : au tableau.

Matroïde

Deux remarques

Soit (C, I) un matroïde.

Fait

1. *Tous les $S \in I$ maximaux pour l'inclusion ont le même nombre d'éléments.*
2. *Pour tout $B \subseteq C$, $(B, \{S \cap B : S \in I\})$ est un matroïde.*

Ensemble indépendant de poids maximum d'un matroïde

Donnée : (C, I) un matroïde, chaque candidat de C ayant un poids $\in \mathbb{R}^+$
Déterminer : $S \in I$ de poids maximum

- candidats : éléments de C
- fonction de réalisabilité : ensemble courant appartient à I ?
- fonction de sélection $=$ plus lourd candidat non encore considéré

Ensemble indépendant de poids maximum d'un matroïde

L'algorithme

`fonction` *MaxIndép*$(C, I, P[1..|C|])$: ensemble de candidats

  { Retourne un $S \in I$ de poids total maximum }

  L[1..|C|] $\leftarrow$ liste des candidats triée en ordre de P décroissants

  $S \leftarrow \emptyset$

  `tantque` L non vide `faire`

    $a \leftarrow$ retirer prochain candidat de L

    `si` $S \cup \{a\} \in I$ `alors` $S \leftarrow S \cup \{a\}$

  `retourner` S

- Retourne bien un indépendant de poids maximum ?
  Au tableau.
- Intérêt ?
  - s'applique à plus d'une situation
  - capture bien la "voracité"

Retour à l'exemple des vecteurs

du transparent 9

Donnée : vecteurs $v_1, \ldots, v_m \in \mathbb{R}^d$, avec poids $P[1], \ldots, P[m] \in \mathbb{R}^+$
Déterminer : ensemble de vecteurs lin. indép. de poids total maximum

Preuve que *Vecteurs*$(V, P[1..|V|)$ est correct :

1. (C, I) est un matroïde (= un de nos exemples) lorsque
   $C = \{v_1, \ldots, v_m\}$
   $I = \{S \subseteq C : \text{ les vecteurs de } S \text{ sont linéairement indépendants } \}$
2. *MaxIndép*$(C, I, P[1..m])$ est précisément notre algorithme *Vecteurs*$(V, P[1..|V|)$ appliqué à ce cas particulier de (C, I)

Arbres sous-tendants (couvrants) minimaux

Une autre application des matroïdes

Soit (N, A) un graphe et $I = \{S \subseteq A \mid (N, S) \text{ est sans cycle}\}$.

Proposition

(A, I) est un matroïde.

Preuve : faite au tableau.

Arbres sous-tendants minimaux (suite)

Corollaire

On peut utiliser MaxIndép pour calculer un arbre sous-tendant minimal d'un graphe connexe (N, A) *avec poids* $\in \mathbb{R}^+$ *aux arêtes.*

Preuve :

Arbres sous-tendants minimaux (suite)

Corollaire

On peut utiliser MaxIndép pour calculer un arbre sous-tendant minimal d'un graphe connexe (N, A) avec poids $\in \mathbb{R}^+$ aux arêtes.

Preuve :

1. Prendre le matroïde (A, I) avec I de la proposition précédente
2. "Renverser" les poids :
   - en posant $m > \max\{\text{poids}(a) \mid a \in A\}$
   - en associant à chaque $a \in A$ le nouveau poids $(m - \text{poids}(a))$
3. Un indépendant $S \in I$ de poids maximum constituera un a.s.t.
4. Tout tel S aura la même cardinalité
5. MaxIndép aura donc maximisé $(m \times |S|) -$ (poids total de S)
6. MaxIndép aura donc minimisé (poids total de S)

Maxindép devient alors l'algorithme de Kruskal !

```
function Kruskal(G = ⟨N, A⟩: graph; length: A → ℝ⁺): set of edges
    {initialization}
    Sort A by increasing length
    n ← the number of nodes in N
    T ← ∅ {will contain the edges of the minimum spanning tree}
    Initialize n sets, each containing a different element of N
    {greedy loop}
    repeat
        e ← {u, v} ← shortest edge not yet considered
        ucomp ← find(u)
        vcomp ← find(v)
        if ucomp ≠ vcompthen
            merge(ucomp, vcomp)
            T ← T ∪ {e}
    until T contains n − 1 edges
    return T
```

- Candidats = arêtes
- Réalisabilité : arête relie des composantes connexes distinctes ?
- Fonction objective : $n - 1$ arêtes choisies ?

Autre exemple vorace : remise de la monnaie

Autre exemple vorace : remise de la monnaie

function {*make-change*}(n): *set of coins*

{Makes change for n units using the least possible number of coins. The constant C specifies the coinage}

const $C = \{100, 25, 10, 5, 1\}$

$S \leftarrow \varnothing$ {S is a set that will hold the solution}

$s \leftarrow 0$ {s is the sum of the items in S}

while $s \neq n$ **do**

$x \leftarrow$ the largest item in C such that $s + x \leq n$

if there is no such item **then**

return "*no solution found*"

$S \leftarrow S \cup$ {a coin of value x}

$s \leftarrow s + x$

return S

Mais attention !

- Une heuristique vorace peut s'appliquer à un problème sans constituer un algorithme vorace correct
  - par exemple, si l'heuristique ne garantit pas que la solution trouvée vérifie le critère d'optimalité

Au fait, l'approche vorace fonctionne-t-elle au Canada (avec ou sans la "penny" d'avant 2013) pour la remise de la monnaie ?

© J.J.'s Complete Guide to Canada.

Au fait, l'approche vorace fonctionne-t-elle au Canada (avec ou sans la "penny" d'avant 2013) pour la remise de la monnaie ?

© J.J.'s Complete Guide to Canada.

Oui ! Mais pourvu qu'un nombre illimité de pièces de chaque dénomination soient disponibles !

Remise de la monnaie

L'approche vorace fonctionne-t-elle dans tous les pays lorsqu'un nombre illimité de pièces est disponible ?

Probablement, mais pas en général !

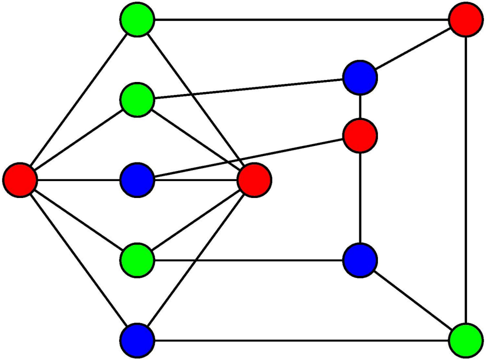
- Échoue pour certaines dénominations de pièces !
- Étonnamment difficile de démontrer le bon fonctionnement, même pour les pièces canadiennes

Retour à l'exemple

function {*make-change*}(n): *set of coins*
{Makes change for n units using the least possible
number of coins. The constant C specifies the coinage}
const $C = \{100, 25, 10, 5, 1\}$
$S \leftarrow \emptyset$ {S is a set that will hold the solution}
$s \leftarrow 0$ {s is the sum of the items in S}
while $s \neq n$ **do**
$x \leftarrow$ the largest item in C such that $s + x \leq n$
if there is no such item **then**
return "*no solution found*"
$S \leftarrow S \cup$ {a coin of value x}
$s \leftarrow s + x$
return S

- Ici les candidats sont implicites
- Fonction de réalisabilité : $s + x \leq n$, i.e., "est-ce que l'ajout de x aux candidats choisis totalise au plus le montant n à remettre ?"
- Fonction objective : $s = n$, i.e., "est-ce que les pièces totalisent exactement n ?"

L'approche vorace ne fonctionne pas pour résoudre la 3-colorabilité



- Résoudra certains exemplaires mais pas tous
- Peut servir d'heuristique en vue d'une solution approchée

Le “gloutonnoïde”

Quoi ? ?

Cette notion (greedoid) existe !
Mais nous ne l'étudierons pas (au-delà de sa définition).

(C, I) est un gloutonnoïde ssi pour tout $X, Y \subseteq C$:

- la trivialité : comme avant.
- l'accessibilité : $\emptyset \neq X \in I \Rightarrow \exists a \in X, X \setminus \{a\} \in I$
- l'échange : comme avant.

On peut démontrer...

(Matroids are treated in Cormen-Leiserson-Rivest)

C ensemble fini de candidats et $I \subseteq 2^C$

Théorème (du matroïde)

Soit (C, I) muni de la trivialité et de l'hérédité.
MaxIndép obtient $S \in I$ de poids maximum ssi (C, I) possède l'échange.

Théorème (du gloutonnoïde)

Soit (C, I) muni de la trivialité et de l'accessibilité.
MaxIndép obtient $S \in I$ de poids maximum ssi (C, I) possède l'échange.

Corollaire

L'algorithme de Prim qui suit calcule un arbre sous-tendant minimal.

Arbres sous-tendants minimaux (algo de Prim)

function $Prim(G = \langle N, A\rangle: graph; length: A \rightarrow \mathbb{R}^+)$: *set of edges*
    {initialization}
    $T \leftarrow \emptyset$
    $B \leftarrow$ {an arbitrary member of N}
    while $B \neq N$ **do**
        find $e = \{u, v\}$ of minimum *length* such that
            $u \in B$ and $v \in N \setminus B$
        $T \leftarrow T \cup \{e\}$
        $B \leftarrow B \cup \{v\}$
    return T

- Candidats $=$ arêtes
- Réalisabilité : *e* ne cause pas un cycle dans l'arbre en construction ?
- Fonction objective : tous les sommets atteints ?

Code de Huffman

But : encoder en binaire une suite w de caractères

- Bête : $\lceil \log_2 |\text{alphabet}| \rceil$ bits/caractère
  Requiert $|w| \cdot \lceil \log_2 |\text{alphabet}| \rceil$ bits

Code de Huffman

But : encoder en binaire une suite w de caractères

- Bête : $\lceil \log_2 |\text{alphabet}| \rceil$ bits/caractère
  Requiert $|w| \cdot \lceil \log_2 |\text{alphabet}| \rceil$ bits

- Intelligent : encoder les caractères de w les plus fréquents par des suites de bits courtes et les moins fréquents par de plus longues.

  Requiert $\left[\sum_{c \in \text{alphabet}} \underbrace{f(c)}_{\text{fréquence de } c} \cdot |\text{code}(c)| \right]$ bits

Code de Huffman

Exemple

w = *abracadabra*

Lettre	Fréquence	Code bête	Code possible	Code possible
a	5	000	1	1
b	2	001	001	010
r	2	010	01	011
c	1	011	0000	000
d	1	111	0001	001

$w \to$ 000001010000011000111000001010000

$w \to$ 10010110000100011001011

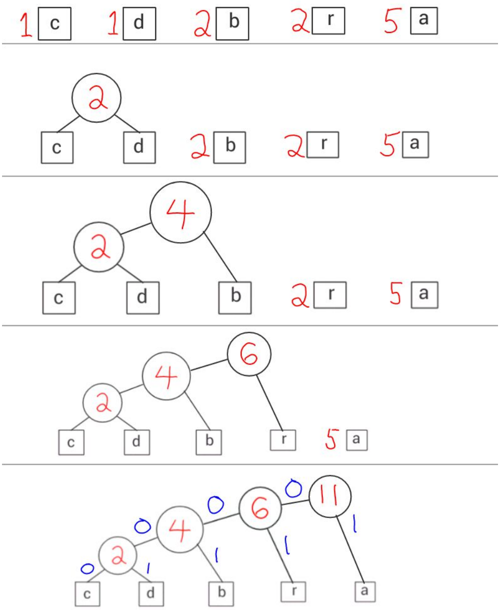
$w \to$ 10100111000100110100111

Code de Huffman

Lettre	Fréquence	Code possible	Code possible
a	5	1	1
b	2	001	010
r	2	01	011
c	1	0000	000
d	1	0001	001

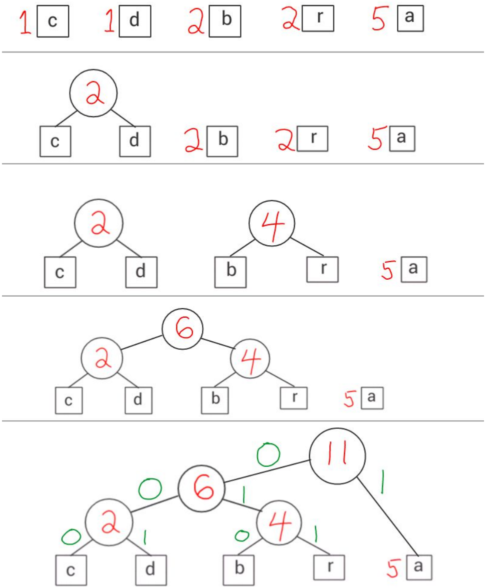
- les deux sont codes “préfixes”
- $\sum_{\sigma \in \{a,b,c,d,r\}} f(\sigma) \cdot |\text{code}(\sigma)| = \sum_{\sigma \in \{a,b,c,d,r\}} f(\sigma) \cdot |\text{code}(\sigma)|$
- comment les construire ?
- sont-ils optimaux ?

Abracadabra



Lettre	Code
a	1
b	001
r	01
c	0000
d	0001

Abracadabra



Lettre	Code
a	1
b	010
r	011
c	000
d	001

Algorithme de Huffman

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
        do z ← ALLOCATE-NODE()
           x ← left[z] ← EXTRACT-MIN(Q)
           y ← right[z] ← EXTRACT-MIN(Q)
           f[z] ← f[x] + f[y]
           INSERT(Q, z)
return EXTRACT-MIN(Q)
```


- candidats = ? ? (En réalité, mélange vorace et diviser-pour-régner)
- Réalisabilité : aucune
- Fonction objective : un seul arbre restant

Optimalité de Huffman

Notons : un arbre donne bien un code préfixe
À démontrer : Huffman donne un code préfixe qui minimize

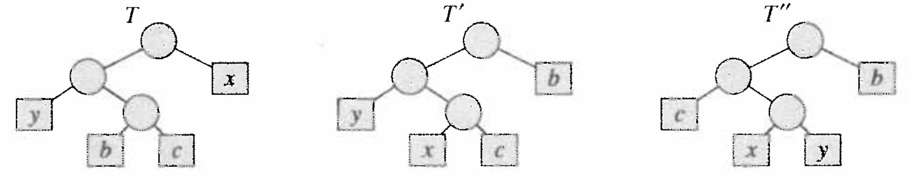
$$\left[\sum_{c \in \text{alphabet}} \underbrace{f(c)}_{\text{fréquence de } c} \cdot |\text{code}(c)|\right]$$

Nommons x et y les deux caractères les moins fréquents.

Preuve en 3 étapes :

1. tout arbre optimal est plein (i.e., toujours 2 ou 0 descendants)
2. un arbre optimal existe avec x et y feuilles de même parent
3. induction

(2) : un arbre optimal existe avec x et y feuilles de même parent



© Cormen-Leiserson-Rivest

- soient b et c les plus profonds de n'importe quel arbre
- interchanger b et x ne peut que réduire le coût
- interchanger c et y ne peut que réduire le coût

(3) : induction

x et y premiers éléments considérés

D n'importe quel arbre binaire plein

$$|D| \geqslant \left| D\begin{bmatrix} x \leftrightarrow b \\ y \leftrightarrow c \end{bmatrix} \right|$$

$$= \left| D\left[\qquad\right]\left[\boxed{x}\ \boxed{y} \rightsquigarrow \boxed{x,y} \right] \right| + f(x) + f(y)$$

$$\geqslant \left| \begin{array}{l} \text{arbre construit par notre} \\ \text{algo qui considère } \boxed{x,y} \end{array} \right| + f(x) + f(y)$$

$$= \left| \text{arbre construit par notre algo} \right|$$

Sac à dos (knapsack)

SAC À DOS

DONNÉE: capacité $W \in \mathbb{R}^{\geq 0}$ et objets $1, 2, \ldots, n$ de poids $w_1, \ldots, w_n \in \mathbb{R}^{\geq 0}$ et de valeurs $v_1, \ldots, v_n \in \mathbb{R}^{\geq 0}$

CALCULER: entiers $0 \leq x_1, \ldots, x_n \leq 1$ maximisant $\sum_1^n x_i v_i$ tels que $\sum_1^n x_i w_i \leq W$.

SAC À DOS FRACTIONNAIRE

DONNÉE: Idem

CALCULER: réels $0 \leq x_1, \ldots, x_n \leq 1$ maximisant $\sum_1^n x_i v_i$ tels que $\sum_1^n x_i w_i \leq W$.

Sac à dos (knapsack)

Un exemplaire

$$n = 5, W = 100$$

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

Figure 6.5. An instance of the knapsack problem

Sac à dos (knapsack)

Fonctions de sélection possibles

$$n = 5, W = 100$$

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

Figure 6.5. An instance of the knapsack problem

Select:	x_i					Value
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
Max v_i/w_i	1	1	1	0	0.8	164

Figure 6.6. Three greedy approaches to the instance in Figure 6.5

Sac à dos (knapsack)

Attention ! Ici la variante “fractionnaire” du problème

```
function knapsack(w[1..n], v[1..n], W): array [1..n]
    {initialization}
    for i = 1 to n do x[i] ← 0
    weight ← 0
    {greedy loop}
    while weight < W do
        i ← the best remaining object {see below}
        if weight + w[i] ≤ W then x[i] ← 1
                                  weight ← weight + w[i]
                             else x[i] ← (W − weight)/w[i]
                                  weight ← W
    return x
```

- candidats = objets
- Réalisabilité : poids des objets n’excède pas W ?
- Fonction objective : poids des objets $= W$.

Et Sac à dos NON fractionnaire ?

Mauvaises nouvelles...

L'heuristique vorace peut

- aboutir à une solution non optimale
- ne pas trouver de solution même si une solution existe

Mais, prix de consolation, aucun algorithme efficace n'est connu pour résoudre le problème NON fractionnaire !

Plus courtes distances de la source (Dijkstra)

PLUS COURTES DISTANCES

DONNÉE: graphe orienté (N, A) avec longueurs non négatives aux arcs
CALCULER: distance minimale du sommet 1 à chaque $s \in N$

Plus courtes distances de la source (Dijkstra)

function $Dijkstra(L[1..n, 1..n])$: **array** $[2..n]$
    array $D[2..n]$
    {initialization}
    $C \leftarrow \{2, 3, \ldots, n\}$ {$S = N \setminus C$ exists only implicitly}
    for $i \leftarrow 2$ **to** n **do** $D[i] \leftarrow L[1, i]$
    {greedy loop}
    repeat $n - 2$ **times**
        $v \leftarrow$ some element of C minimizing $D[v]$
        $C \leftarrow C \setminus \{v\}$ {and implicitly $S \leftarrow S \cup \{v\}$}
        for each $w \in C$ **do**
            $D[w] \leftarrow \min(D[w], D[v] + L[v, w])$
    return D

- candidats : sommets
- Réalisabilité : aucune
- Fonction objective : n candidats traités

Plus courtes distances de la source (Dijkstra)

Idée de preuve d'optimalité

$v \in S \Rightarrow D[v] = |pcc(v)|$

$v \notin S \Rightarrow D[v] = |pccs(v)|$

```
fonction Dijkstra(L[1..n, 1..n]): tableau[2..n]
    {initialisation}
    C ← {2,3,...,n}     {S = N \ C n'existe qu'implicitement}
    pour i ← 2 jusqu'à n faire D[i] ← L[1, i]
    {boucle vorace}
    répéter n–2 fois
        v ← l'élément de C qui minimise D[v]
        C ← C \ {v}  {et implicitement S ← S ∪ {v}}
        pour chaque élément w de C faire
            D[w] ← min(D[w], D[v] + L[v, w])
    retourner D
```

plus court chemin spécial ($v \notin S$):