IFT436 – Algorithmes et structures de données

Université de Sherbrooke

Examen final

Enseignant: Michael Blondin
Date: mercredi 22 décembre 2021
Durée: 3 heures

Directives:

- Vous devez répondre aux questions dans le **cahier de réponses**, et *non* sur ce questionnaire;
- **Une seule feuille** de notes au format 8½" × 11" est permise;
- Les **fiches récapitulatives** des chapitres 5 à 8 se trouvent à la fin de ce questionnaire;
- **Aucun matériel additionnel** n'est permis;
- **Aucun appareil électronique** (calculatrice, téléphone, montre intelligente, etc.) n'est permis;
- Vous devez donner **une seule réponse** par sous-question;
- L'examen comporte **5 questions** sur **8 pages** valant un total de **50 points**;
- La correction se base notamment sur la **clarté**, l'**exactitude** et la **concision** de vos réponses, ainsi que sur la **justification** pour les questions qui en requièrent une;
- Les indices d'une séquence **débutent à 1**; autrement dit, $s = [s[1], s[2], \ldots, s[n]]$ si $n = |s|$.

Question 1: analyse d'algorithmes récursifs

Un artiste a produit quatre exemplaires de n œuvres. Un exemplaire de l'œuvre i vaut $v[i]$ dollars et possède une aire de $a[i]$ cm^2. L'artiste désire minimiser la superficie de son kiosque du marché de Noël de Sherbrooke, mais en étant en mesure d'obtenir au moins b dollars en vendant tout ce qu'il y exposera. Formellement, il cherche donc à identifier des valeurs $x[1], \ldots, x[n] \in [0..4]$ qui minimisent $a[1] \cdot x[1] + \ldots + a[n] \cdot x[n]$ tout en satisfaisant $v[1] \cdot x[1] + \ldots + v[n] \cdot x[n] \geq b$. Une informaticienne lui propose cet algorithme:

Entrées: deux séquences v et a de $n \geq 0$ entiers positifs, et un entier positif b
Résultat: aire minimale

```
aire-min(v, a, b):
  résoudre(i, aire, valeur):
    si i = n + 1 alors
      si valeur ≥ b alors retourner aire
      sinon retourner ∞
    sinon
      m ← ∞
      pour q ∈ [0..4]
        m ← min(m, résoudre(i + 1, aire + a[i] · q, valeur + v[i] · q))
      retourner m
  retourner résoudre(1, 0, 0)
```

(a) Donnez une récurrence linéaire t où $t(n)$ dénote le nombre d'opérations élémentaires exécutées par l'appel initial « `résoudre(1, 0, 0)` » par rapport à n. Considérez ces opérations élémentaires: 2,5 pts

- arithmétique: $+$, $\cdot$, min, etc.;
- affectations;
- comparaisons: $=$, $\geq$, etc.;
- accès aux éléments d'une séquence.

$$t(n) = \begin{cases} 3 & \text{si } n = 0, \\ 5 \cdot t(n-1) + \underbrace{48}_{3+5\cdot 9} & \text{sinon.} \end{cases}$$

(b) Identifiez la forme close de t. Vous n'avez *pas* à identifier les valeurs des constantes, mais vous devez indiquer le système d'équations que vous auriez à résoudre pour les identifier. Laissez une trace de votre démarche. 3 pts

La récurrence linéaire non homogène $t(n) - 5 \cdot t(n-1) = 48 \cdot 1^n$ mène au polynôme $(x^1 - 5)(x - 1)$. La forme close est donc $t(n) = c_1 \cdot 5^n + c_2 \cdot 1^n$. On obtient le système:

$$\begin{aligned} t(0) &= 3 = c_1 + c_2 \\ t(1) &= 63 = 5c_1 + c_2 \end{aligned}$$

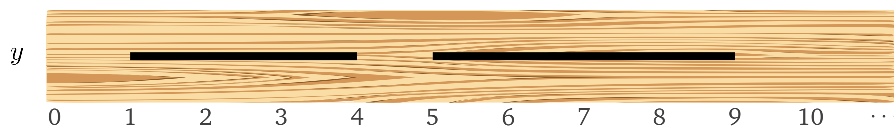
(c) Expliquez comment améliorer l'algorithme à l'aide d'une forme d'élagage. 2,5 pts

Sol. 1: si *valeur* $+ 5 \cdot (v[i] + \ldots + v[n]) < b$, on ne poursuit pas car impossible d'obtenir $\geq b$.

Sol. 2: si *valeur* $\geq b$, on ne poursuit pas car ajouter des exemplaires augmentera l'aire.

Question 2: diviser-pour-régner

Considérons une machine-outil à commande numérique. Sa librairie logicielle permet notamment de découper une séquence de traits dans une planche de bois. Formellement, un *trait* est une paire $(g, d) \in \mathbb{N}^2$ où $g < d$ indiquent le côté gauche et droite. Par exemple, sur entrée $s = [(1, 3), (5, 7), (2, 4), (6, 9)]$, la machine effectue cette découpe, où y est la position verticale actuelle de l'outil:



Sur cette entrée s, la machine effectue des découpages superflus, par ex. après avoir découpé le trait $(1, 3)$, il n'est pas nécessaire de découper $(2, 4)$ en entier car $(2, 3)$ a déjà été retiré de la planche. Comme la machine est lente, nous cherchons à implémenter un algorithme `réduire` qui retire les chevauchements d'une séquence de traits (avant de les envoyer à la machine). Par exemple, `réduire(`s`)` peut retourner $[(1, 4), (5, 9)]$.

(a) Complétez le pseudocode ci-dessous: 7 pts

Entrée: une séquence s de traits
Résultat: une séquence de traits sans chevauchements équivalente à s

```
réduire(s):
  si |s| ≤ 1 alors
    retourner s
  sinon
    m ← |s| ÷ 2
    x ← réduire(s[1 : m])
    /* Complétez ce segment                                          */
    /*     de pseudocode (sur autant de lignes que nécessaire)       */
```

Afin de rendre votre code moins cryptique, utilisez au besoin ces fonctions auxiliaires, où t est un trait:

```
gauche(t):
  retourner t[1]
droite(t):
  retourner t[2]
```

```
réduire(s):
  si |s| ≤ 1 alors
    retourner s
  sinon
    m ← |s| ÷ 2
    x ← réduire(s[1 : m])
    y ← réduire(s[m + 1 : n])
    z ← []
    i ← 1; j ← 1                                   // fusionner séquences
    tant que i ≤ |x| ∧ j ≤ |y|
      si gauche(x[i]) > gauche(y[j]) alors         // premier trait d'abord
        x ↔ y
        i ↔ j
      si droite(x[i]) < gauche(y[j]) alors         // traits disjoints?
        ajouter x[i] à z
        i ← i + 1
      sinon
        x[i] ← (gauche(x[i]), max(droite(x[i]), droite(y[j])))
        j ← j + 1
    retourner z + x[i : |x|] + y[j : |y|]          // ajouter traits restants
```

Dans l'ancien solutionnaire, la première ligne du bloc **sinon** était « $x[i] \leftarrow (\texttt{gauche}(x[i]), \texttt{droite}(y[j]))$ ». Or, c'est incorrect car c'est possible que le trait $y[j]$ soit entièrement compris dans le trait $x[i]$. Il faut donc garder le plus grand côté droit parmi les deux traits.

(b) Nous disons qu'un trait (a, b) *contient* un point x si $x \in [a..b]$. Par exemple, le trait $(5, 7)$ contient 5, 6 et 7. Complétez cet algorithme: 2 pts

Entrées: une séquence s de traits triés et sans chevauchements, et $x \in \mathbb{N}$
Résultat: *vrai* s'il y a un trait de s qui contient x, *faux* sinon

```
contient(s, x):
    contient'(lo, hi):
        si lo = hi alors
            /* à compléter */
        sinon
            i ← (lo + hi) ÷ 2
            si x ≤ droite(s[i]) alors
                retourner contient'(lo, i)
            sinon
                /* à compléter */
    retourner contient'(1, |s|)
```

```
contient(s, x):
    contient'(lo, hi):
        si lo = hi alors
            retourner gauche(s[lo]) ≤ x ≤ droite(s[lo])
        sinon
            i ← (lo + hi) ÷ 2
            si x ≤ droite(s[i]) alors
                retourner contient'(lo, i)
            sinon
                retourner contient'(i + 1, hi)
    retourner contient'(1, |s|)
```

(c) Quel est le temps d'exécution asymptotique des algorithmes obtenus en (a) et (b) dans le pire cas? Vous pouvez supposer que $|s|$ est une puissance de deux. Justifiez. 3 pts

(a) Le temps s'exprime par $t(n) = 2 \cdot t(n \div 2) + f(n)$ où $f \in \mathcal{O}(n)$. Par le théorème maître, avec $b = c = 2$ et $d = 1$, on obtient $t \in \mathcal{O}(n \log n)$.

(b) Le temps s'exprime par $t(n) = t(n \div 2) + f(n)$ où $f \in \mathcal{O}(1)$. Par le théorème maître, avec $b = 2$, $c = 1$ et $d = 0$, on obtient $t \in \mathcal{O}(\log n)$.

Question 3: force brute et programmation dynamique

Considérons un jeu où Tux se déplace vers la droite sur une banquise de n blocs. Le bloc i lui permet de sauter jusqu'au bloc $i + s[i]$ (grâce à de la nourriture qui lui offre momentanément une dose d'énergie). Par exemple, considérons la banquise ci-dessous. Sur le premier bloc, Tux peut se rendre en un seul saut au bloc 2 ou au bloc 3 (car $i = 1$ et $s[1] = 2$). Tux débute à la case 1 et désire atteindre la case n avec le moins de sauts possible.

2	3	0	1	1	0

Formellement, une banquise est représentée par une séquence s d'entiers non négatifs, par ex. $s = [2, 3, 0, 1, 1, 0]$ ci-dessus. Remarquons qu'il est possible que Tux ne puisse pas se rendre à la case n, par ex. si $s[1] = 0$. De plus, la valeur du dernier bloc n'influence pas le parcours puisque Tux a terminé (c'est de la pure gourmandise).

(a) Afin d'identifier le nombre de sauts minimal que Tux doit effectuer pour se rendre à la case n, nous pourrions explorer exhaustivement *tous* les parcours possibles. Cette procédure fonctionnerait-elle en temps polynomial dans le pire cas par rapport à n? Justifiez. 3 pts

> Non.
>
> *Sol. 1*: Si $s = [2, \dots, 2]$, il y a 2 façons de traverser 2 blocs: $[1, 1]$ ou $[2]$. Il y a donc $\geq 2^{n/2}$ parcours.
>
> *Sol. 2*: Si $s = [2, \dots, 2]$, il y a au moins 2 façons de traverser 3 blocs: $[1, 2]$ ou $[2, 1]$. Il y a donc $\geq 2^{n/3}$ parcours.
>
> *Sol. 3*: Si $s = [2, \dots, 2]$, le nombre de parcours est décrit par la suite de Fibonacci $f(n) = f(n-1) + f(n-2)$, qui est exponentielle.
>
> *Sol. 4*: Si $s = [n-1, \dots, 1, 0]$, le nombre de parcours est décrit par $f(n) = f(n-1) + \dots + f(0)$, et ainsi par la suite exponentielle $1, 1, 2, 4, 8, 16, 32, \dots$.

(b) Quel est le nombre de sauts minimal pour la banquise $[2, 3, 0, 1, 1, 0]$? Justifiez. 2 pts

> Trois sauts.
>
> *Sol. 1*: Tux doit éviter le bloc 3, car autrement il est bloqué. Tux doit donc sauter au bloc 2. Ensuite, il peut sauter au bloc 4 ou 5. Le bloc 4 doit obligatoirement passer par le bloc 5, donc on a intérêt à y aller directement.
>
> *Sol. 2*: Soit $T[i]$ le nombre de sauts minimal à partir du bloc i. On a $T[i] = \min\{1 + T[i+j] : 1 \leq j \leq s[i]\}$ où $T[n] = 0$ et les positions invalides valent ∞. On obtient donc 3 car $T = [3, 2, \infty, 2, 1, 0]$.

(c) Donnez un algorithme, décrit sous forme de pseudocode, qui identifie le nombre de sauts minimal pour une banquise de n blocs en temps polynomial. Justifiez le temps d'exécution. 5 pts

> *Sol. 1*:
>
> ```
> sauts(s):
>     initialiser un tableau T[1..n] avec ∞
>     T[n] ← 0
>     pour i de n − 1 à 1
>         pour k de i + 1 à min(i + s[i], n)
>             T[i] ← min(T[i], 1 + T[k])
>     retourner T[1]
> ```
>
> L'initialisation se fait en temps $\mathcal{O}(n)$. Comme les autres opérations sont élémentaires et qu'on exécute deux boucles imbriquées au plus n fois chacune, on obtient $\mathcal{O}(n^2)$ au total.
>
> *Sol. 2*: Même idée, mais vers l'avant.
>
> *Sol. 3–4*: Mêmes idées, mais en procédant récursivement et en utilisant la mémoïsation.

(d) Expliquez, en mots et/ou avec du pseudocode, comment adapter l'algorithme afin d'identifier une séquence 2 pts

de sauts qui mène à une solution minimale.

Sol. 1: Lors du calcul de $T[i]$, on mémorise $k - i$ tel que $T[i] = 1 + T[k]$. Cela permet de construire la séquence de sauts à partir de $T[0]$.

Sol. 2: Même idée mais en rebroussant chemin à partir de $T[n]$.

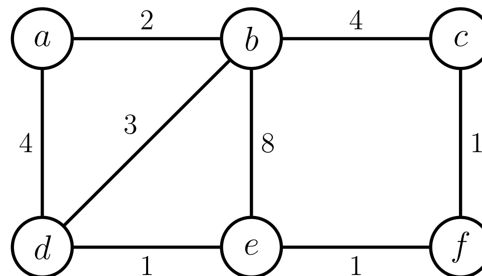
Sol. 3–4: Mêmes idées mais récursivement.

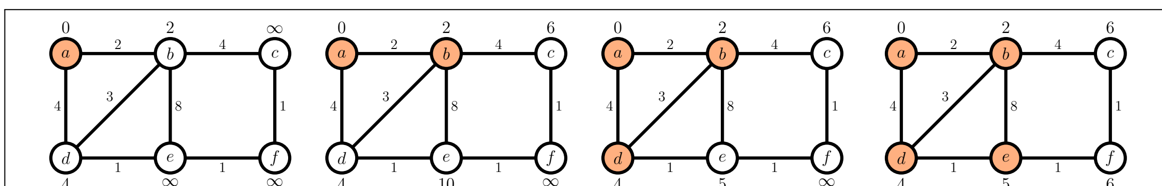
Question 4: plus courts chemins

(a) Expliquez comment traduire le problème de la question 3 en un problème de plus court chemin. Indiquez, en mots, comment obtenir le graphe résultant et les sommets de départ et d'arrivée. 2,5 pts

On cherche la distance du sommet 1 vers le sommet n dans le graphe dirigé $\mathcal{G} = (V, E)$ où $V := \{1, \ldots, n\}$, $E := \{(i, j) : 1 \leq i \leq j \leq \min(i + s[i], n)\}$, et chaque arête est pondérée par 1.

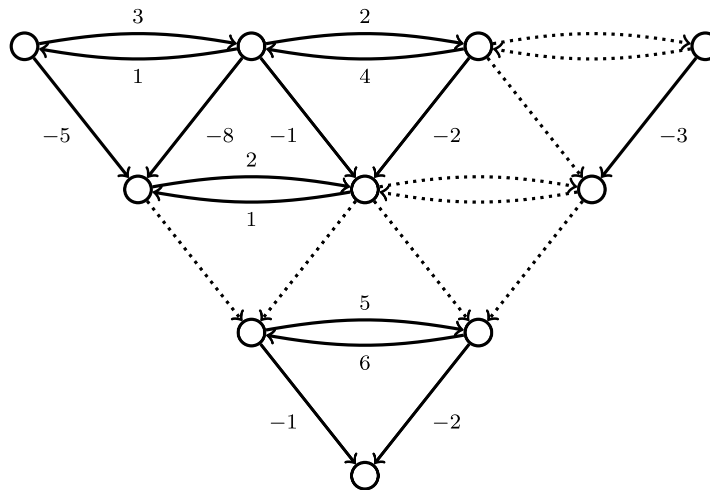
(b) Exécutez l'algorithme de Dijkstra sur le graphe pondéré ci-dessous afin de déterminer la longueur d'un plus court chemin de a vers f. Arrêtez l'exécution *dès que possible*. Laissez une trace des itérations effectuées en indiquant les sommets marqués et les distances partielles identifiées. 3 pts





À la prochaine itération, on peut marquer c ou f (même valeur: 6). On s'arrête donc en marquant f.

(c) Considérons une carte triangulaire, avec n cases de départ et une case d'arrivée, organisée de cette façon: 2,5 pts



Plus précisément, l'étage du haut possède n cases, et chaque étage subséquent en possède une de moins. Sur chaque étage on peut se déplacer librement de gauche à droite avec un coût positif. De plus, on peut descendre vers une case adjacente avec un coût négatif (mais pas remonter). On cherche à identifier une case de départ sur l'étage du haut qui minimise le coût d'atteindre la case tout en bas. Afin d'y arriver, est-ce plus efficace d'utiliser l'algorithme de Floyd–Warshall; l'algorithme de Dijkstra pour chaque case de départ; ou l'algorithme de Bellman–Ford pour chaque case de départ? Justifiez.

Rappel: leurs temps d'exécution, tels qu'analysés en classe, appartiennent à $\Theta(|V|^3)$*,* $\mathcal{O}(|V| \log |V| + |E|)$ *et* $\Theta(|V| \cdot |E|)$ *respectivement.*

Sol. 1: On ne peut pas utiliser l'algorithme de Dijkstra comme il y a des poids négatifs. On a $|V| = n(n+1)/2 \in \Theta(n^2)$ et $2(|V|-1) \leq |E| \leq 4|V|$. Les temps d'exécution sont de $\Theta((n^2)^3) = \Theta(n^6)$ et $\Theta(n \cdot n^4) = \Theta(n^5)$. L'algo. de Bellman-Ford est donc préférable.

Sol. 2 (même chose avec autre notation): On a $2(|V|-1) \leq |E| \leq 4|V|$. Les temps d'exécution sont de $\Theta(|V|^3)$ et $\Theta(\sqrt{|V|} \cdot |V| \cdot |V|) = \Theta(|V|^{2.5})$. L'algo. de Bellman-Ford est donc préférable.

Question 5: algorithmes probabilistes

Il n'est pas nécessaire d'évaluer vos résultats numériquement; des expressions symboliques suffisent.

Nous disons qu'une séquence s de n entiers est *triangulaire* si:

— n est un multiple de 6;
— s contient $(1/6)n$ occurrences d'un nombre a, $(2/6)n$ occurrences d'un nombre b, et $(3/6)n$ occurrences d'un nombre c,
— $a \neq b$, $a \neq c$ et $b \neq c$.

Par exemple, la séquence $s = [20, 10, 15, 20, 20, 10]$ est triangulaire. Considérons ces deux algorithmes probabilistes qui cherchent à identifier le minimum d'une séquence triangulaire:

Entrée: séquence triangulaire s de taille $n \in \mathbb{N}_{\geq 6}$
Résultat: $\min(s)$

```
triangulaireA(s):
    faire
        choisir i ∈ [1..n] aléatoirement de façon uniforme
        choisir j ∈ [1..n] aléatoirement de façon uniforme
        choisir k ∈ [1..n] aléatoirement de façon uniforme
    tant que |{s[i], s[j], s[k]}| ≠ 3                // trois valeurs distinctes?
    retourner min(s[i], s[j], s[k])
```

Entrée: séquence triangulaire s de taille $n \in \mathbb{N}_{\geq 6}$
Résultat: $\min(s)$

```
triangulaireB(s):
    faire 100 fois
        choisir i ∈ [1..n] aléatoirement de façon uniforme
        choisir j ∈ [1..n] aléatoirement de façon uniforme
        si |{s[1], s[i], s[j]}| = 3 alors            // trois valeurs distinctes?
            retourner min(s[1], s[i], s[j])
    retourner s[1]
```

Pour chacun des algorithmes `triangulaireA` et `triangulaireB`:

(a) Dites s'il est de Las Vegas ou de Monte Carlo. Expliquez pourquoi. 3 pts

`triangulaireA`: *Las Vegas* puisqu'il est toujours correct (on quitte la boucle avec $\{a, b, c\}$), mais que la boucle peut être arbitrairement longue (par ex. en pigeant $i = j = k$ continuellement).

`triangulaireB`: *Monte Carlo* car son temps est toujours de $\mathcal{O}(1)$ (au plus 100 itérations), et car il peut se tromper (par ex. si $s[1]$ n'est pas le minimum et qu'on pige $i = j = 1$ chaque fois).

(b) Selon votre réponse en (a), identifiez la probabilité d'erreur ou le temps espéré de l'algorithme. Justifiez. 7 pts

`triangulaireA`: On quitte une itération si les trois valeurs sont distinctes, ainsi avec probabilité

$$p = \underbrace{3!}_{\text{\# d'ordonnancements}} \cdot (1/6 \cdot 2/6 \cdot 3/6) = 1/6.$$

Le nombre d'itérations espéré est donc de $1/p = 6$ (loi géométrique). Puisque les opérations sont toutes élémentaires, le temps espéré est de $\mathcal{O}(1)$.

`triangulaireB`: Si $s[1] = \min(s)$, alors il n'y a jamais d'erreur. Sinon, la probabilité p d'obtenir un succès à une itération est

$$p = \begin{cases} 2! \cdot 2/6 \cdot 3/6 = 1/3 & \text{si } s[1] = a, \\ 2! \cdot 1/6 \cdot 3/6 = 1/6 & \text{si } s[1] = b, \\ 2! \cdot 1/6 \cdot 2/6 = 1/9 & \text{si } s[1] = c. \end{cases}$$

On considère le pire cas: $p = 1/9$. La probabilité d'erreur est donc de $(1 - 1/9)^{100} = (8/9)^{100}$.

Annexe:

Fiches récapitulatives

5. Algorithmes récursifs et approche diviser-pour-régner

Diviser-pour-régner

- *A)* découper en sous-problèmes disjoints
- *B)* obtenir solutions récursivement
- *C)* s'arrêter aux cas de base (souvent triviaux)
- *D)* combiner solutions pour obtenir solution globale
- *Exemple*: tri par fusion $\mathcal{O}(n \log n)$

Récurrences linéaires

- *Cas homogène:* $\sum_{i=0}^{d} a_i \cdot t(n-i) = 0$
- *Polynôme caractéristique:* $\sum_{i=0}^{d} a_i \cdot x^{d-i}$
- *Forme close:* $t(n) = \sum_{i=1}^{d} c_i \cdot \lambda_i^n$ où les λ_i sont les racines
- *Constantes c_i:* obtenues en résolvant un sys. d'éq. lin.
- *Cas non homo.:* si $= c \cdot b^n$, on multiplie poly. par $(x - b)$
- *Exemple*:
  - Récurrence: $t(n) = 3 \cdot t(n-1) + 4 \cdot t(n-2)$
  - Poly. carac.: $x^2 - 3x - 4 = (x-4)(x+1)$
  - Forme close: $t(n) = c_1 \cdot 4^n + c_2 \cdot (-1)^n$

Autres méthodes

- *Substitution*: remplacer $t(n), t(n-1), t(n-2), \ldots$ par sa déf. jusqu'à deviner la forme close
- *Arbres*: construire un arbre représentant la récursion et identifier le coût de chaque niveau

Quelques algorithmes

- *Hanoï*: $src[1:n-1] \to tmp, src[n] \to dst, tmp[1:n-1] \to dst$ $\mathcal{O}(2^n)$
- *Exp. rapide*: exploiter $b^n = (b^{n \div 2})^2 \cdot b^{n \bmod 2}$ $\mathcal{O}(\log n)$
- *Mult. rapide*: calculer $(a+b)(c+d)$ en 3 mult. $\mathcal{O}(n^{\log 3})$
- *Horizon*: découper blocs comme tri par fusion $\mathcal{O}(n \log n)$

Théorème maître (allégé)

- $t(n) = c \cdot t(n \div b) + f(n)$ où $f \in \mathcal{O}(n^d)$:
  - $\mathcal{O}(n^d)$ si $c < b^d$
  - $\mathcal{O}(n^d \cdot \log n)$ si $c = b^d$
  - $\mathcal{O}(n^{\log_b c})$ si $c > b^d$

6. Force brute

Approche

- *Exhaustif*: essayer toutes les sol. ou candidats récursivement
- *Explosion combinatoire*: souvent # solutions $\geq b^n, n!, n^n$
- *Avantage*: simple, algo. de test, parfois seule option
- *Désavantage*: généralement très lent et/ou avare en mémoire

Techniques pour surmonter explosion

- *Élagage*: ne pas développer branches inutiles
- *Contraintes*: élaguer si contraintes enfreintes
- *Bornes*: élaguer si impossible de faire mieux
- *Approximations*: débuter avec approx. comme meilleure sol.
- *Si tout échoue*: solveurs SAT ou d'optimisation

Problème des n dames

- *But*: placer n dames sur échiquier sans attaques
- *Algo.*: placer une dame par ligne en essayant colonnes dispo.

Sac à dos

- *But*: maximiser valeur sans excéder capacité
- *Algo.*: essayer sans et avec chaque objet
- *Mieux*: élaguer dès qu'il y a excès de capacité
- *Mieux++*: élaguer si aucune amélioration avec somme valeurs

Retour de monnaie

- *But*: rendre montant avec le moins de pièces
- *Algo.*: pour chaque pièce, essayer d'en prendre 0 à # max.

7. Programmation dynamique

Approche

- *Principe d'optimalité*: solution optimale obtenue en combinant solutions de sous-problèmes qui se chevauchent
- *Descendante*: algo. récursif + mémoïsation (ex. Fibonacci)
- *Ascendante*: remplir tableau itér. avec solutions sous-prob.

Retour de monnaie

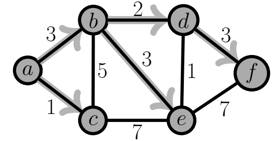
- *Sous-question*: # pièces pour rendre j avec pièces 1 à i?
- *Identité*: $T[i, j] = \min(T[i-1, j],\ T[i, j - s[i]] + 1)$
- *Exemple*: montant $m = 10$ et pièces $s = [1, 5, 7]$

	0	1	2	3	4	5	6	7	8	9	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8	9	10
2	0	1	2	3	4	1	2	3	4	5	2
3	0	1	2	3	4	1	2	1	2	3	2

Sac à dos

- *Sous-question*: val. max. avec capacité j et les objets 1 à i?
- *Identité*: $T[i, j] = \max(T[i-1, j],\ T[i-1, j - p[i]] + v[i])$

Plus courts chemins



- *Déf.*: chemin simple de poids minimal
- *Bien défini*: si aucun cycle négatif
- *Approche générale*: raffiner distances partielles itérativement
- *Dijkstra*: raffiner en marquant sommet avec dist. min.
- *Floyd-Warshall*: raffiner via sommet intermédiaire v_k
- *Bellman-Ford*: raffiner avec $\geq 1, 2, \ldots, |V| - 1$ arêtes
- *Sommaire*:

	Dijkstra	**Bellman-Ford**	**Floyd-Warshall**
Types de chemins	d'un sommet vers les autres		paires de sommets
Poids négatifs?	✗	✓	✓
Temps d'exécution	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$	$\Theta(\|V\| \cdot \|E\|)$	$\Theta(\|V\|^3)$
Temps ($\|E\| \in \Theta(1)$)	$\mathcal{O}(\|V\| \log \|V\|)$	$\Theta(\|V\|)$	$\Theta(\|V\|^3)$
Temps ($\|E\| \in \Theta(\|V\|)$)	$\mathcal{O}(\|V\| \log \|V\|)$	$\Theta(\|V\|^2)$	$\Theta(\|V\|^3)$
Temps ($\|E\| \in \Theta(\|V\|^2)$)	$\mathcal{O}(\|V\|^2)$	$\Theta(\|V\|^3)$	$\Theta(\|V\|^3)$

8. Algorithmes et analyse probabilistes

Modèle probabiliste

- *Modèle*: on peut tirer à pile ou face (non déterministe)
- *Aléa*: on peut obtenir une loi uniforme avec une pièce
- *Idéalisé*: on suppose avoir accès à une source d'aléa parfaite (en pratique: source plutôt pseudo-aléatoire)

Algorithmes de Las Vegas

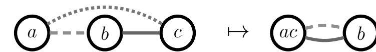
- *Temps*: varie selon les choix probabilistes
- *Valeur de retour*: toujours correcte
- *Exemple*: tri rapide avec pivot aléatoire
- *Temps espéré*: dépend de $\mathbb{E}[Y_x]$ où $Y_x = \#$ opér. sur entrée x

Algorithmes de Monte Carlo

- *Temps*: borne ne varie *pas* selon les choix probabilistes
- *Valeur de retour*: pas toujours correcte
- *Exemple*: algorithme de Karger
- *Prob. d'erreur*: dépend de $\Pr(Y_x \neq \text{bonne sortie sur } x)$

Coupe minimum: algorithme de Karger

- *Coupe*: partition (X, Y) des sommets d'un graphe non dirigé
- *Taille*: # d'arêtes qui traversent X et Y
- *Coupe min.*: identifier la taille minimale d'une coupe
- *Algorithme*: contracter itérativement une arête aléatoire en gardant les multi-arêtes, mais pas les boucles



- *Prob. d'erreur*: $\leq 1 - 1/|V|^2$ (Monte Carlo)
- *Amplification*: on peut réduire (augmenter) la prob. d'erreur (de succès) arbitrairement (en général: avec min., maj., $\vee$, etc.)

Temps moyen

- *Temps moyen*: $\sum$(temps instances de taille n) / # instances
- *Attention*: *pas* la même chose que le temps espéré
- *Hypothèse*: entrées distribuées uniformément ($\pm$ réaliste)
- *Exemple*: $\Theta(n^2)$ pour le tri par insertion