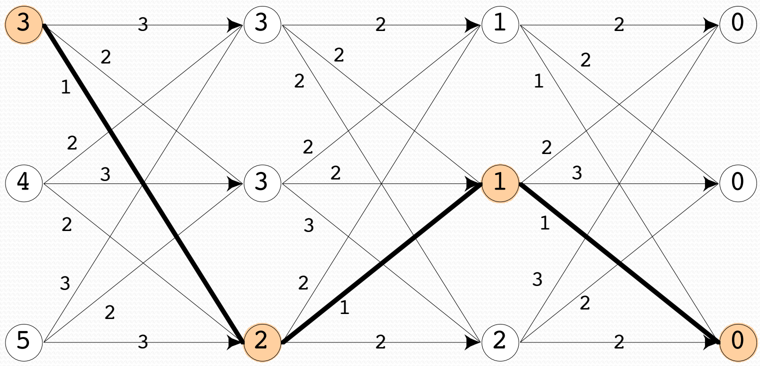
Dynamic programming

Chapter Summary

- Introduction to dynamic programming
- Application to shortest-path distance in a graph
- Application to edit-distance

Dynamic programming

- Suppose we have a lattice with N levels (acyclic directed graph with levels):



3

Dynamic programming

- The problem is to reach level N
- From level 0
- With minimum cost = shortest-path problem

4

Dynamic programming

- Some definitions

$$s_k = \text{variable containing the state at level } k$$

- The local cost associated to the decision to jump to the state $s_k = j$ at level k

$$d(s_k = j \mid s_{k-1} = i)$$

- Given that we were in state $s_{k-1} = i$ at level $k-1$

5

Dynamic programming

- The total cost of a path $(s_0, s_1, \ldots, s_N)$

  Is: $D(s_0, s_1, \ldots, s_N) = \sum_{i=1}^{N} d(s_i \mid s_{i-1})$

- The optimal cost when starting from state s_0 is

$$D^*(s_0) = \min_{(s_1,\ldots,s_N)} \{D(s_0, s_1, \ldots, s_N)\}$$
$$= \min_{(s_1,\ldots,s_N)} \left\{ \sum_{i=1}^{N} d(s_i \mid s_{i-1}) \right\}$$

- The minimum is taken with respect to the set of values of each variable (index of nodes in each level)

6

Dynamic programming

- The optimal cost, whatever the initial state, is

$$D^* = \min_{s_0} \{D^*(s_0)\}$$

- And the optimal cost when starting from some intermediate state s_k:

$$D^*(s_k) = \min_{(s_{k+1},\ldots,s_N)} \left\{ \sum_{i=k+1}^{N} d(s_i \mid s_{i-1}) \right\}$$

7

Dynamic programming

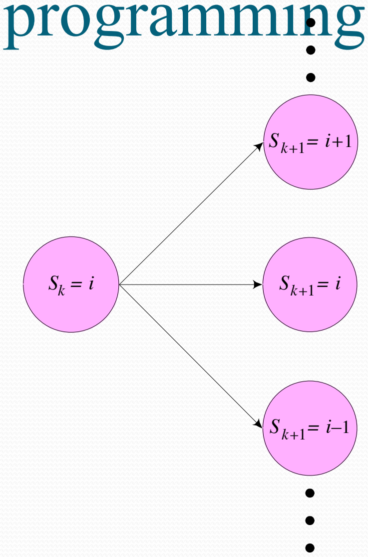
- Here is the backward recurrence relation allowing to obtain the optimal cost:

$$\boxed{\begin{aligned} &D^*(s_N) = 0 \\ &D^*(s_k = i) = \min_{s_{k+1}} \{d(s_{k+1} \mid s_k = i) + D^*(s_{k+1})\} \\ &D^* = \min_{s_0} \{D^*(s_0)\} \end{aligned}}$$

8

Dynamic programming

- Graphically:



9

Dynamic programming

- Proof:

$$\begin{aligned}
D^*(s_k) &= \min_{(s_{k+1},\ldots,s_N)} \left\{ \sum_{i=k+1}^{N} d(s_i \mid s_{i-1}) \right\} \\
&= \min_{(s_{k+1},\ldots,s_N)} \left\{ d(s_{k+1} \mid s_k) + \sum_{i=k+2}^{N} d(s_i \mid s_{i-1}) \right\} \\
&= \min_{s_{k+1}} \left\{ \min_{(s_{k+2},\ldots,s_N)} \left\{ d(s_{k+1} \mid s_k) + \sum_{i=k+2}^{N} d(s_i \mid s_{i-1}) \right\} \right\} \\
&= \min_{s_{k+1}} \left\{ d(s_{k+1} \mid s_k) + \underbrace{\min_{(s_{k+2},\ldots,s_N)} \left[\sum_{i=k+2}^{N} d(s_i \mid s_{i-1}) \right]}_{D^*(s_{k+1})} \right\} \\
&= \min_{s_{k+1}} \left\{ d(s_{k+1} \mid s_k) + D^*(s_{k+1}) \right\}
\end{aligned}$$

Dynamic programming

- In a symmetric way, here is the forward recurrence relation (used, e.g., in edit-distance):

$$D^*(s_0) = 0$$

$$D^*(s_k = i) = \min_{s_{k-1}} \{d(s_k = i \mid s_{k-1}) + D^*(s_{k-1})\}$$

$$D^* = \min_{s_N} \{D^*(s_N)\}$$

11

Dynamic programming

- Now, if there are jumps bypassing some levels, the backward recurrence becomes:

$$D^*(s_N) = 0$$

$$D^*(s_k = i) = \min_{\{s\}|s_k = i \to s} \{d(s \mid s_k = i) + D^*(s)\}$$

$$D^* = \min_{s_0} \{D^*(s_0)\}$$

- where $\{s\}\,|s_k = i \to s = Succ(i)$ is the set of states to which there is a direct transition from $s_k = i$
- That is, all the succesors of $s_k = i$

12

Dynamic programming

- To find the optimal path, we need to keep track of
  - the previous state in each node (a pointer from the previous state to the current state)
- And use backtracking from the last node
  - in order to retrieve the optimal path

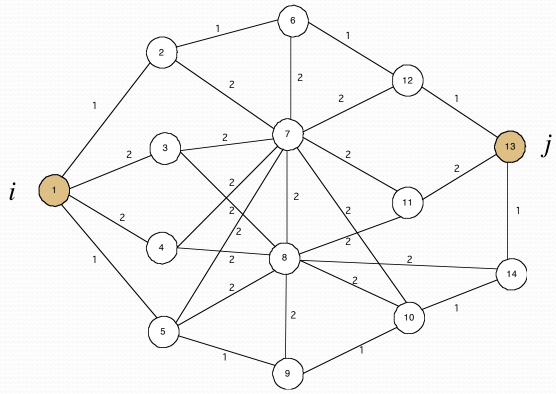
13

Formule de Bellman-Ford (distance du plus court chemin dans un graphe)

14

Coûts minimal entre deux noeuds d'un réseau

- Imaginons que nous avons un réseau ou graphe



Coûts minimal entre deux noeuds d'un réseau

- Nous souhaitons calculer le coût minimal pour se rendre à un état 0 (état destination) à partir de tout autre état du réseau
  - Il existe plusieurs algorithmes performants résolvant ce problème
  - Nous choisissons une variante de l'algorithme de Bellman-Ford, utilisant la programmation dynamique

Coûts minimal entre deux noeuds d'un réseau

- Notons $c_{ij} = d(j \mid i)$ le coût immédiat entre le noeud i et le noeud j
- Nous avons, pour $i \neq j$:
  - c_{ij} = coût du lien, s'il existe un lien entre i et j et $i \neq 0$ (i n'est pas le noeud destination !)
  - $c_{ij} = \infty$ s'il n'y a pas de lien entre i et j
  - $c_{0j} = \infty$ pour tout $j \neq 0$ càd que, une fois qu'on a atteint l'état final 0, on est arrivé à destination et on y reste (pas moyen de s'échapper)

Coûts minimal entre deux noeuds d'un réseau

- Un état dont on ne peut pas s'échapper est appelé absorbant
- Un coût infini implique que la transition ne sera jamais choisie
  - Et donc c'est comme si elle n'existait pas (aucun lien)

Coûts minimal entre deux noeuds d'un réseau

- Nous devons considérer les “self-loops”, pour $i = j$ (= boucle) :
  - $c_{ii} = \infty$ pour tout $i \neq 0$ (on n'autorise pas de boucler dans un état non-destination – c'est sous-optimal)
  - $c_{00} = 0$: quand on arrive à l'état destination, on y reste sans coût additionnel (on est arrivé à destination)

Coûts minimal entre deux noeuds d'un réseau

- Supposons que l'état 0 est l'état destination
- Nous déployons le réseau dans le temps, en niveaux
  - chaque niveau k correspond à un “time step”, ou transition, ou étape
  - Si n est le nombre de noeuds, pour calculer la distance la plus courte, il ne peut pas y avoir plus de n niveaux
    - Sinon, on passerait plusieurs fois par le même état, ce qui est sous-optimal

Coûts minimal entre deux noeuds d'un réseau

- On se retrouve ainsi dans les conditions d'un problème de programmation dynamique à n niveaux et n états (graphe dirigé acyclique)
- Pour l'implémentation, définissons un tableau $D^*(i, k)$ de dimension n x n où
  - i est l'indice de l'état (0 à n–1)
  - situé au niveau k (0 à n–1)
- Les éléments (i, k) de ce tableau représentent:

$$D^*(s_k = i)$$

Coûts minimal entre deux noeuds d'un réseau

- Le tableau $D^*(i, k)$ contiendra donc le coût minimal pour atteindre l'état destination 0
  - A partir de l'état d'indice i
  - Au niveau k
- Il est initialisé de la manière suivante:
  - $D^*(0, n–1) = 0$ (état destination 0 au dernier niveau)
  - $D^*(i, n–1) = \infty$ pour tout autre noeud $i \neq 0$ du dernier niveau (ces états ne sont pas permis – uniquement l'état 0 est un état destination)

Coûts minimal entre deux noeuds d'un réseau

- L'algorithme est itératif: on examine, à chaque itération, à partir de *i*,
  - par quel état intermédiaire *j* il est le plus intéressant de passer

Coûts minimal entre deux noeuds d'un réseau

- Appliquons la formule backward de programmation dynamique (niveaux de 0 à $N = (n-1)$):

$$D^*(s_N) = 0$$

$$D^*(s_k = i) = \min_{s_{k+1}} \{d(s_{k+1} \mid s_k = i) + D^*(s_{k+1})\}$$

$$D^* = \min_{s_0} \{D^*(s_0)\}$$

$$\begin{cases} D^*(0, n-1) = 0 \\ D^*(i, n-1) = \infty \text{ for } i \neq 0 \\ D^*(i,k) = \min_{j \in Succ(i)} \{c_{ij} + D^*(j, k+1)\} \end{cases}$$

Coûts minimal entre deux noeuds d'un réseau

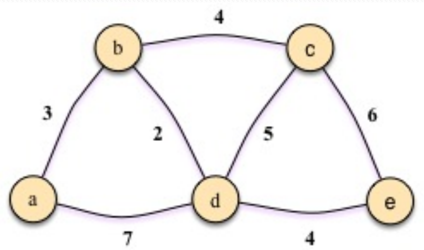
- Après initialisation (niveau n–1),
- Il faut donc itérer n–1 fois, pour les n–1 niveaux inférieurs (pour k de n–2 à 0):
- For i = 0 to n–1

$$D^*(i, k) = \min_{j=1}^{n} \{c_{ij} + D^*(j, k+1)\}$$

- End
  - puisque $c_{ij} = \infty$ pour les transitions impossibles

Coûts minimal entre deux noeuds d'un réseau

- Soit l'example suivant contenant 5 noeuds {a, b, c, d, e}



Coûts minimal entre deux noeuds d'un réseau

- La matrice de coûts associée est

$$\mathbf{C} = \begin{bmatrix} \infty & 3 & \infty & 7 & \infty \\ 3 & \infty & 4 & 2 & \infty \\ \infty & 4 & \infty & 5 & 6 \\ 7 & 2 & 5 & \infty & 4 \\ \infty & \infty & 6 & 4 & \infty \end{bmatrix}$$

Coûts minimal entre deux noeuds d'un réseau

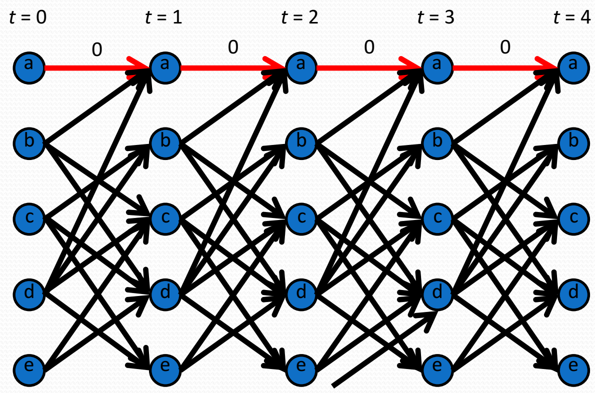
- Pour connaître la distance de chaque noeud vers le noeud a, celui-ci est rendu absorbant:

noeud absorbant

$$\mathbf{C} = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ 3 & \infty & 4 & 2 & \infty \\ \infty & 4 & \infty & 5 & 6 \\ 7 & 2 & 5 & \infty & 4 \\ \infty & \infty & 6 & 4 & \infty \end{bmatrix}$$

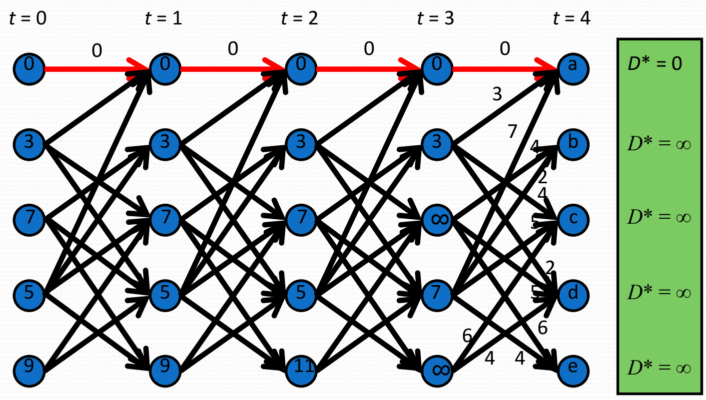
Coûts minimal entre deux noeuds d'un réseau

- Nous formons le graphe dirigé acyclique



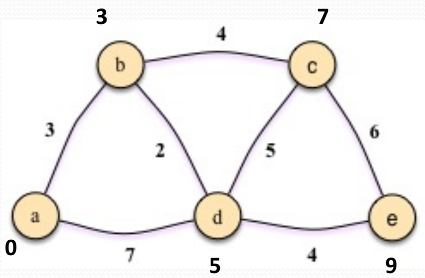
Coûts minimal entre deux noeuds d'un réseau

- Nous appliquons la programmation dynamique en tenant compte des coûts:



Coûts minimal entre deux noeuds d'un réseau

- Nous obtenons donc pour le coût vers a:

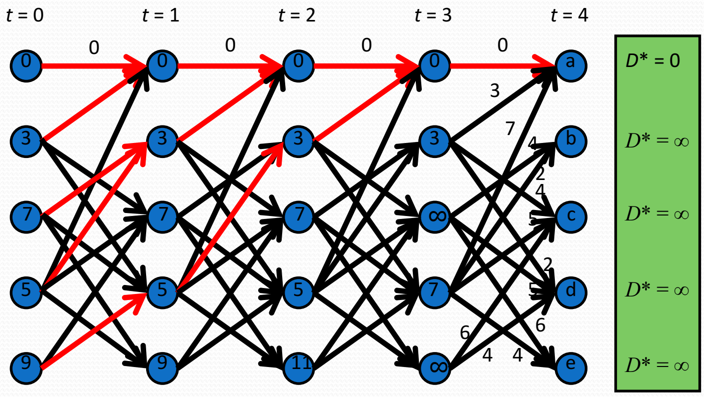


Coûts minimal entre deux noeuds d'un réseau

- Cet algorithme calcule le coût minimal pour rejoindre le noeud d'indice o à partir de tout autre noeud
- Si l'on veut retrouver le chemin le plus court, il faut retenir, pour chaque noeud *i*,
  - le noeud successeur *j* qui produit la distance minimale
  - Ainsi, pour chaque noeud, on peut retrouver le noeud successeur qui se trouve sur le trajet le plus court et suivre ce fil conducteur jusqu'au noeud destination o

Coûts minimal entre deux noeuds d'un réseau

- Calcul du chemin le plus court:



Application à l'edit-distance

Application à l'edit-distance

- Calcul de la distance entre deux chaînes de caractères
- Calcule le nombre minimal d'insertions, de suppressions et de substitutions
- Pour passer d'une chaîne de caractères **x** à une autre chaîne de caractères **y**

35

Application à l'edit-distance

- Egalement appelé "distance de Levenstein"
- Soient deux chaînes de caractères

$$\begin{cases} \mathbf{y} = y_1 y_2 \dots y_{|\mathbf{y}|} \\ \mathbf{x} = x_1 x_2 \dots x_{|\mathbf{x}|} \end{cases}$$

- x_i étant le *i*-ème caractère de la chaîne **x**

36

Application à l'edit-distance

- La longueur de la chaîne **x** est notée |**x**|
- En général, nous avons

$$|\mathbf{x}| \neq |\mathbf{y}|$$

- Une sous-chaîne est définie par

$$\mathbf{x}_i^j = x_i x_{i+1} \dots x_j$$

37

Application à l'edit-distance

- Nous allons lire un à un les caractères de la chaîne **x** pour construire progressivement la chaîne **y**
- Nous définissons pour ce faire trois opérations d'édition:
  - Insertion d'un caractère dans **y** (sans piocher dans **x**)
  - Suppression d'un caractère de **x** (sans la concaténer à **y**)
  - Substitution d'un caractère de **y** par celui de **x**

38

Application à l'edit-distance

- Première convention:

$$\mathbf{x}_i^{|\mathbf{x}|}$$

  - Signifie que nous avons lu les i–1 premiers caractères de **x**
  - Et donc **x** est amputé de ses i–1 premiers caractères
  - Nous les avons piochés pour alimenter **y**

39

Application à l'edit-distance

- Deuxième convention:

$$\mathbf{y}_0^j$$

  - Signifie que les j premiers caractères de **y** ont été transcrits
- Nous lisons donc progressivement des caractères de **x** pour écrire **y**

40

Application à l'edit-distance

- Il s'agit d'un processus à niveaux (étapes) et états qui donne lieu à a graphe dirigé acyclique
  - Nous pourrons donc appliquer la programmation dynamique

$$\begin{cases} \text{Un état correspond à } (\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^j) \\ \text{Niveau } k \text{ correspond à } i + j = const = k \end{cases}$$

  - Un état est donc caractérisé par le couple (i, j)
    = i –1 premiers caractères de $\mathbf{x}$ lus et
    j caractères transcrits dans $\mathbf{y}$

41

Application à l'edit-distance

- Voici la définition des trois opérations

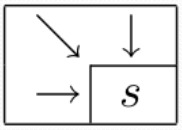
$$\begin{cases} \text{Insertion par rapport à } \mathbf{x}\text{: } (\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^{j-1}) \rightarrow (\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^j) \\ \text{Suppression par rapport à } \mathbf{x}\text{: } (\mathbf{x}_{i-1}^{|\mathbf{x}|}, \mathbf{y}_0^j) \rightarrow (\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^j) \\ \text{Substitution par rapport à } \mathbf{x}\text{: } (\mathbf{x}_{i-1}^{|\mathbf{x}|}, \mathbf{y}_0^{j-1}) \rightarrow (\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^j) \end{cases}$$

- Pour les deux premières opérations, on passe du niveau k au niveau k+1
- Pour la dernière, on passe au niveau k+2 (on "saute" un niveau)

42

Application à l'edit-distance

- Nous représentons la situation par un tableau à deux dimensions
  - Un niveau est représenté par $(i + j)$ = constante = diagonale
  - Un état est représenté par (i, j)
    - i est l'indice de ligne
    - j est l'indice de colonne
  - Une opération est représentée par une transition dans ce tableau



43

Application à l'edit-distance

- Exemple de tableau correspondant au calcul de la distance entre "livre" et "lire"

y

		∅	l	i	v	r	e
	∅	0	1	2	3	4	5
	l	1	0	1	2	3	4
x	i	2	1	0	1	2	3
	r	3	2	1	1	1	2
	e	4	3	2	2	2	1

44

Application à l'edit-distance

- Un niveau correspond à une diagonale:
  - $(i + j) = \text{cte}$

$$
\begin{array}{cc|cccccc|}
 & & \multicolumn{6}{c}{\mathbf{y}} \\
 & & \varnothing & \text{l} & \text{i} & \text{v} & \text{r} & \text{e} \\
\hline
 & \varnothing & 0 & 1 & 2 & 3 & 4 & 5 \\
 & \text{l} & 1 & 0 & 1 & 2 & 3 & 4 \\
\mathbf{x} & \text{i} & 2 & 1 & 0 & 1 & 2 & 3 \\
 & \text{r} & 3 & 2 & 1 & 1 & 1 & 2 \\
 & \text{e} & 4 & 3 & 2 & 2 & 2 & 1 \\
\hline
\end{array}
$$

45

Application à l'edit-distance

- Nous introduisons un coût (ou pénalité) associé à chaque opération (insertion, suppression, substitution)

$$
\begin{cases}
\delta_{\text{ins}}(y_i) = 1 \\
\delta_{\text{sup}}(x_i) = 1 \\
\delta_{\text{sub}}(x_i, y_j) = \delta_{ij}
\end{cases}
$$

- Avec

$$
\begin{cases}
\delta_{ij} = 1 \text{ si } x_i \neq y_j \\
\delta_{ij} = 0 \text{ si } x_i = y_j
\end{cases}
$$

46

Application à l'edit-distance

- Nous pouvons appliquer les formules de programmation dynamique forward:
  - Initialement, aucune opération n'a encore été effectuée (coditions initiales):

$$D^*(\mathbf{x}_0^{|\mathbf{x}|}, \mathbf{y}_0^0) = 0$$

  - Ensuite:

$$\boxed{D^*(\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^j) = \min \begin{cases} D^*(\mathbf{x}_i^{|\mathbf{x}|}, \mathbf{y}_0^{j-1}) + 1 \\ D^*(\mathbf{x}_{i-1}^{|\mathbf{x}|}, \mathbf{y}_0^j) + 1 \\ D^*(\mathbf{x}_{i-1}^{|\mathbf{x}|}, \mathbf{y}_0^{j-1}) + \delta_{ij} \end{cases}}$$

Application à l'edit-distance

- Et finalement l'édit-distance est la valeur du dernier élément du tableau:

$$\mathrm{dist}(\mathbf{x}, \mathbf{y}) = D^*(\mathbf{x}_{|\mathbf{x}|}^{|\mathbf{x}|}, \mathbf{y}_0^{|\mathbf{y}|})$$

- Cette valeur est appelée edit-distance ou distance de Levenstein

48

Application à l'edit-distance

- Exemple à traiter

		y					
		∅	l	i	v	r	e
	∅	0	1	2	3	4	5
	l	1	0	1	2	3	4
x	i	2	1	0	1	2	3
	r	3	2	1	1	1	2
	e	4	3	2	2	2	1

- dist(lire, livre) = 1

Application à l'edit-distance

- Si l'on veut connaître le chemin optimal (séquence optimale des opérations)
  - Il faut maintenir, pour chaque état, un pointeur vers l'état d'où l'on vient dans un tableau
- Les informaticiens ont développé de nombreuses améliorations de cet algorithme (plus rapides)

Deux livres d'algorithmique