Structure de données en Text Mining

Julien Lemoine

29 mai 2008

Plan

Usage

Structure de données: liée à un/des algorithme(s)

Type de structure de données

- Structures généralistes (Tableaux, Tables de hashages, arbres de recherches binaires, ...)
- Chaque domaine a des structures de données spécifiques:
  par exemple les transducteurs en NLP

Dictionnaires

- Dans notre métier: 99% des structures sont des dictionnaires
- Objectif: associer une valeur Y à chaque clé X
- Exemples:
  - dictionnaire de mots
  - dictionnaire de n-grams
  - dictionnaire associant des meta-données à une url
  - dictionnaire associant des co-occurrences à un mot
  - ...

Structures utilisées dans notre domaine

Algorithmes

- Les algorithmes appliqués sur ces structures de données sont souvent plus complexes qu'une simple recherche exacte
- Par exemple:
  - recherche via une expression régulière
  - recherche approximative (trouver l'orthographe la plus proche)
  - recherche par préfixe(trouver toutes les urls d'un site dans un dictionnaire d'urls)
  - ...

Les bonnes questions

Comment choisir la bonne structure:

- Structures de données dynamiques ou statiques: ajout/suppression de valeurs à quelle fréquence ?
- RAM ou Disque: nombres d'éléments ?
- Quels algorithmes vont utiliser cette structure ?
- Performances ? Nombres d'accès à la seconde

Correction orthographique (1/2)

Idée: comparer deux mots à l'aide d'une distance pour savoir si ils sont proches ou pas (distance d'édition):

- Distance de Levenshtein : compter le plus petit nombre de suppression, insertion, substitution
- Distance de Damerau-Levenshtein : compter le plus petit nombre de suppression, insertion, substitution, transposition
- possibilité de donner des poids différents entre la suppression, l'insertion, la substitution et la transposition (par défault tous à 1)
- selon les travaux de Damerau (1964), 80% des erreurs d'orthographe sont corrigées avec une de ces 4 corrections
- aussi largement utilisé en génétique pour comparer les séquences d'ADN

Correction orthographique (2/2)

Il s'agit de la base statistique de tout algorithme de correction à laquelle on ajoute :

- Prise en compte de la fréquence des mots
- Phonétisation des mots (par exemple éviter que "andore" soit corrigé en "andre (andré)" plutôt que "andorre" alors que la distance est la même)

Exemples

- $D_{levenshtein}(bojnour, bonjour) = 2$ (2 substitutions)
- $D_{damerau-levenshtein}(bonjnour, bonjour) = 1$ (1 transposition)
- $D_{levenshtein}(alseimer, alzheimer) = 2$ (1 substitution + 1 insertion)
- $D_{damerau-levenshtein}(alseimer, alzheimer) = 2$ (1 substitution + 1 ajout)
- $D_{levenshtein}(bonjjour, bonjour) = 1$ (1 suppression)
- $D_{levenshtein}(dijstra, dijkstra) = 1$ (1 insertion)
- ...

Implémentation classique

- Comparaison entre deux mots
- Application d'un algorithme rapide (programmation dynamique) pour comparer deux mots (disponible dans tous les langages de programmation)

Programmation dynamique (Damerau-Levenshtein)

```
uint compute(char mot1[1..tailleMot1], char mot2[1..tailleMot2])
  uint d[0..tailleMot1, 0..tailleMot2], i, j, dist
  Pour (i = 0, i ≤ tailleMot1, ++i) d[i, 0] = i
  Pour (j = 1, j ≤ tailleMot2, ++j) d[0, j] = j

  Pour (i = 1, i ≤ tailleMot1, ++i)
    Pour (j = 1 j ≤ tailleMot2, ++j)
      Si (mot1[i] = mot2[j] alors dist = 0 sinon dist = 1
      d[i, j] = min(d[i-1, j ] + 1, // suppression
                    d[i , j-1] + 1, //insertion
                    d[i-1, j-1] + dist) //substitution
      Si (i > 1 et j > 1 et mot1[i] = mot2[j-1] et mot1[i-1] = mot2[j]) alors
        d[i, j] = min(d[i, j], d[i-2, j-2] + dist) //transposition
  retourne d[tailleMot1, tailleMot2]
```

Exemple

- Comparer le mot “crise” avec le mot “kries”
- Initialisation, d =

	0	1	2	3	4	5
0	**0**	**1**	**2**	**3**	**4**	**5**
1	**1**					
2	**2**					
3	**3**					
4	**4**					
5	**5**					

- Voir cette initialisation comme :
  - d[0, i] = comparaison de la chaîne de caractères “kries” avec une chaîne de caractères ayant *i* caractères en moins. Ex: d(“kries”, “kr”) = 3
  - d[i, 0] = comparaison de la chaîne de caractères “crise” avec une chaîne de caractères ayant *i* caractères en moins. Ex: d(“crise”, “c”) = 4

Exemple, i = 1, j = 1

- Objectif: remplir la case d[1, 1] (“crise”, “kries”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “kries”, il faut donc maintenant comparer “ries” avec “crise”. distance = d[0, 1] + 1 = 2. d[0, 1] car “ries” à uniquement quatre lettres
  - insérer une lettre devant le mot “kries”, il faut donc maintenant comparer “kries” avec “rise”, distance = d[1, 0] + 1 = 2. d[1, 0] car “rise” à uniquement quatre lettres
  - substituer la lettre “k” par “c”, il faut donc maintenant comparer “ries” avec “rise”, distance = d[0, 0] + 1 = 1
  - transposition: pas encore possible

Comparaison de deux mots

Résultat

- $d[1,1] = min(d[0,1]+1, d[1,0]+1, d[0,0]+1) = 1$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	**1**				
2	2					
3	3					
4	4					
5	5					

Exemple, i = 1, j = 2

- Objectif: remplir la case d[1, 2] (“crise”, “ries”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “ries”, il faut donc maintenant comparer “ies” avec “crise”. distance = d[0, 2] + 1 = 3
  - insérer une lettre devant le mot “ries”, il faut donc maintenant comparer “ries” avec “rise”, distance = d[1, 1] + 1 = 2
  - substituer la lettre “r” par “c”, il faut donc maintenant comparer “ies” avec “rise”, distance = d[0, 1] + 1 = 2
  - transposition: pas encore possible

Résultat

- $d[1,2] = min(d[0,2]+1, d[1,1]+1, d[0,1]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	**2**			
2	2					
3	3					
4	4					
4	5					

Exemple, i = 1, j = 3

- Objectif: remplir la case d[1, 3] (“crise”, “ies”)
- $d[1,3] = min(d[1,2]+1, d[0,3]+1, d[0,2]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	**3**		
2	2					
3	3					
4	4					
5	5					

Exemple, i = 1, j = 4

- Objectif: remplir la case d[1, 4] (“crise”, “es”)
- $d[1,4] = min(d[1,3]+1, d[0,4]+1, d[0,3]+1) = 4$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	**4**	
2	2					
3	3					
4	4					
5	5					

Exemple, i = 1, j = 5

- Objectif: remplir la case d[1, 5] (“crise”, “s”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “s”, il faut donc maintenant comparer “” avec “crise”. distance = d[0, 5] + 1 = 6
  - insérer une lettre devant le mot “s”, il faut donc maintenant comparer “s” avec “rise”, distance = d[1, 4] + 1 = 5
  - substituer la lettre “s” par “c”, il faut donc maintenant comparer “” avec “rise”, distance = d[0, 4] + 1 = 5
  - transposition: pas encore possible

Comparaison de deux mots

Résultat

- $d[1,5] = min(d[0,5]+1, d[1,4]+1, d[0,4]+1) = 5$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	**5**
2	2					
3	3					
4	4					
5	5					

Exemple, i = 2, j = 1

- Objectif: remplir la case d[2, 1] (“rise”, “kries”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “kries”, il faut donc maintenant comparer “ries” avec “rise”. distance = d[1, 1] + 1 = 2
  - insérer une lettre devant le mot “kries”, il faut donc maintenant comparer “kries” avec “ise”, distance = d[2, 0] + 1 = 3
  - substituer la lettre “k” par “r”, il faut donc maintenant comparer “ries” avec “ise”, distance = d[1, 0] + 1 = 2
  - transposition: pas encore possible

Comparaison de deux mots

Résultat

- $d[2,1] = min(d[1,1]+1, d[2,0]+1, d[1,0]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	**2**				
3	3					
4	4					
5	5					

Exemple, i = 2, j = 2

- Objectif: remplir la case d[2, 2] (“rise”, “ries”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “ries”, il faut donc maintenant comparer “ies” avec “rise”. distance = d[1, 2] + 1 = 3
  - insérer une lettre devant le mot “ries”, il faut donc maintenant comparer “ries” avec “ise”, distance = d[2, 1] + 1 = 3
  - substituer la lettre “r” par “r”, il faut donc maintenant comparer “ies” avec “ise”, distance = d[1, 1] + 0 = 1
  - transposition: tester le matche de “rkise” avec “crise” (pas possible)

Comparaison de deux mots

Résultat

- $d[2,2] = min(d[1,2]+1, d[2,1]+1, d[1,1]) = 1$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	**1**			
3	3					
4	4					
5	5					

Exemple, i = 2, j = 3

- Objectif: remplir la case d[2, 3] (“rise”, “ies”)
- $d[2,3] = min(d[1,3]+1, d[2,2]+1, d[1,2]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	**2**		
3	3					
4	4					
5	5					

Exemple, i = 2, j = 4

- Objectif: remplir la case d[2, 4] (“rise”, “es”)
- $d[2,4] = min(d[1,4]+1, d[2,3]+1, d[1,3]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	**3**	
3	3					
4	4					
5	5					

Comparaison de deux mots

Exemple, i = 2, j = 5

- Objectif: remplir la case d[2, 5] (“rise”, “s”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “s”, il faut donc maintenant comparer “” avec “rise”. distance = d[1, 5] + 1 = 6
  - insérer une lettre devant le mot “s”, il faut donc maintenant comparer “s” avec “ise”, distance = d[2, 4] + 1 = 4
  - substituer la lettre “s” par “r”, il faut donc maintenant comparer “” avec “ise”, distance = d[1, 4] + 1 = 5
  - transposition: tester le matche de “se” avec “crise” (pas possible)

Résultat

- $d[2,5] = min(d[1,5]+1, d[2,4]+1, d[1,4]+1) = 4$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	**4**
3	3					
4	4					
5	5					

Exemple, i = 3, j = 1

- Objectif: remplir la case d[3, 1] (“ise”, “kries”)
- $d[3,1] = min(d[2,1]+1, d[3,0]+1, d[2,0]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	**3**				
4	4					
5	5					

Exemple, i = 3, j = 2

- Objectif: remplir la case d[3, 2] (“ise”, “ries”)
- $d[3,2] = min(d[2,2]+1, d[3,1]+1, d[2,1]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	**2**			
4	4					
5	5					

Exemple, i = 3, j = 3

- Objectif: remplir la case d[3, 3] (“ise”, “ies”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “ies”, il faut donc maintenant comparer “es” avec “ise”. distance = d[2, 3] + 1 = 3
  - insérer une lettre devant le mot “ies”, il faut donc maintenant comparer “ies” avec “se”, distance = d[3, 2] + 1 = 3
  - substituer la lettre “i” par “i”, il faut donc maintenant comparer “se” avec “es”, distance = d[2, 2] + 0 = 1
  - transposition: tester le matche de “ires” avec “rise” (pas possible)

Comparaison de deux mots

Résultat

- $d[3,3] = min(d[2,3]+1, d[3,2]+1, d[2,2]) = 1$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	**1**		
4	4					
5	5					

Exemple, i = 3, j = 4

- Objectif: remplir la case d[3, 4] (“ise”, “es”)
- $d[3,4] = min(d[2,4]+1, d[3,3]+1, d[2,3]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	**2**	
4	4					
5	5					

Exemple, i = 3, j = 5

- Objectif: remplir la case d[3, 2] (“ise”, “s”)
- $d[3,5] = min(d[2,5]+1, d[3,4]+1, d[2,4]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	**3**
4	4					
5	5					

Exemple, i = 4, j = 1

- Objectif: remplir la case d[4, 1] (“se”, “kries”)
- $d[4,1] = min(d[3,1]+1, d[4,0]+1, d[3,0]+1) = 4$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	**4**				
5	5					

Exemple, i = 4, j = 2

- Objectif: remplir la case d[4, 2] (“se”, “ries”)
- $d[4,2] = min(d[3,2]+1, d[4,1]+1, d[3,1]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	**3**			
5	5					

Exemple, i = 4, j = 3

- Objectif: remplir la case d[4, 3] (“se”, “ies”)
- $d[4,3] = min(d[3,3]+1, d[4,2]+1, d[3,2]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	**2**		
5	5					

Exemple, i = 4, j = 4

- Objectif: remplir la case d[4, 4] (“se”, “es”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “es”, il faut donc maintenant comparer “s” avec “se”. distance = d[3, 4] + 1 = 3
  - insérer une lettre devant le mot “es”, il faut donc maintenant comparer “es” avec “e”, distance = d[4, 3] + 1 = 3
  - substituer la lettre “e” par “s”, il faut donc maintenant comparer “e” avec “s”, distance = d[3, 3] + 1 = 2
  - transposition: tester le matche de “eis” avec “ise” (pas possible)

Résultat

- $d[4,4] = min(d[3,4]+1, d[4,3]+1, d[3,3]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	**2**	
5	5					

Exemple, i = 4, j = 5

- Objectif: remplir la case d[4, 5] (“se”, “s”)
- $d[4,5] = min(d[3,5]+1, d[4,4]+1, d[3,4]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	**3**
5	5					

Exemple, i = 5, j = 1

- Objectif: remplir la case d[5, 1] (“e”, “kries”)
- $d[5,1] = min(d[4,1]+1, d[5,0]+1, d[4,0]+1) = 5$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	3
5	5	**5**				

Exemple, i = 5, j = 2

- Objectif: remplir la case d[5, 2] (“e”, “ries”)
- $d[5,2] = min(d[4,2]+1, d[5,1]+1, d[4,1]+1) = 4$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	3
5	5	5	**4**			

Exemple, i = 5, j = 3

- Objectif: remplir la case d[5, 3] (“e”, “ies”)
- $d[5,3] = min(d[4,3]+1, d[5,2]+1, d[4,2]+1) = 3$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	3
5	5	5	4	**3**		

Comparaison de deux mots

Exemple, i = 5, j = 4

- Objectif: remplir la case d[5, 4] (“e”, “es”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “es”, il faut donc maintenant comparer “s” avec “e”. distance = d[4, 4] + 1 = 3
  - insérer une lettre devant le mot “es”, il faut donc maintenant comparer “es” avec “”, distance = d[5, 3] + 1 = 4
  - substituer la lettre “e” par “e”, il faut donc maintenant comparer “s” avec “”, distance = d[4, 3] + 0 = 2
  - transposition: tester le matche de “eis” avec “se” (pas possible)

Résultat

- $d[5,4] = min(d[4,4]+1, d[5,3]+1, d[4,3]) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	3
5	5	5	4	3	**2**	

Exemple, i = 5, j = 5

- Objectif: remplir la case d[5, 5] (“e”, “s”)
- Prendre le min de plusieurs cas :
  - supprimer la première lettre du mot “s”, il faut donc maintenant comparer “” avec “e”. distance = d[4, 5] + 1 = 4
  - insérer une lettre devant le mot “s”, il faut donc maintenant comparer “s” avec “”, distance = d[5, 4] + 1 = 3
  - substituer la lettre “s” par “e”, il n’y a plus rien à comparer ensuite. distance = d[4, 4] + 1 = 3
  - transposition: tester le matche de “es” avec “se” (possible). distance = d[3, 3] + 1 = 2

Résultat

- $d[5,5] = min(d[4,5]+1, d[5,4]+1, d[4,4]+1, d[3,3]+1) = 2$
- d =

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	4	5
2	2	2	1	2	3	4
3	3	3	2	1	2	3
4	4	4	3	2	2	3
5	5	5	4	3	2	**2**

- Donc $distance(\text{“}kries\text{”}, \text{“}crise\text{”}) = d[5,5] = 2$

Conclusion

- Algorithme rapide pour comparer deux mots: $O(n \times m)$, n = taille du mot_1 et m = taille du mot_2
- La complexité ne dépend pas de la distance recherchée
- En pratique, le besoin est plutôt de chercher le mot ayant la plus petite distance avec le mot *requête* dans un grand dictionnaire
- Il n'est pas envisagable de calculer la distance avec tous les mots du dictionnaire
- Il faut donc une structure de données permetant d'obtenir le mot le plus proche rapidement.

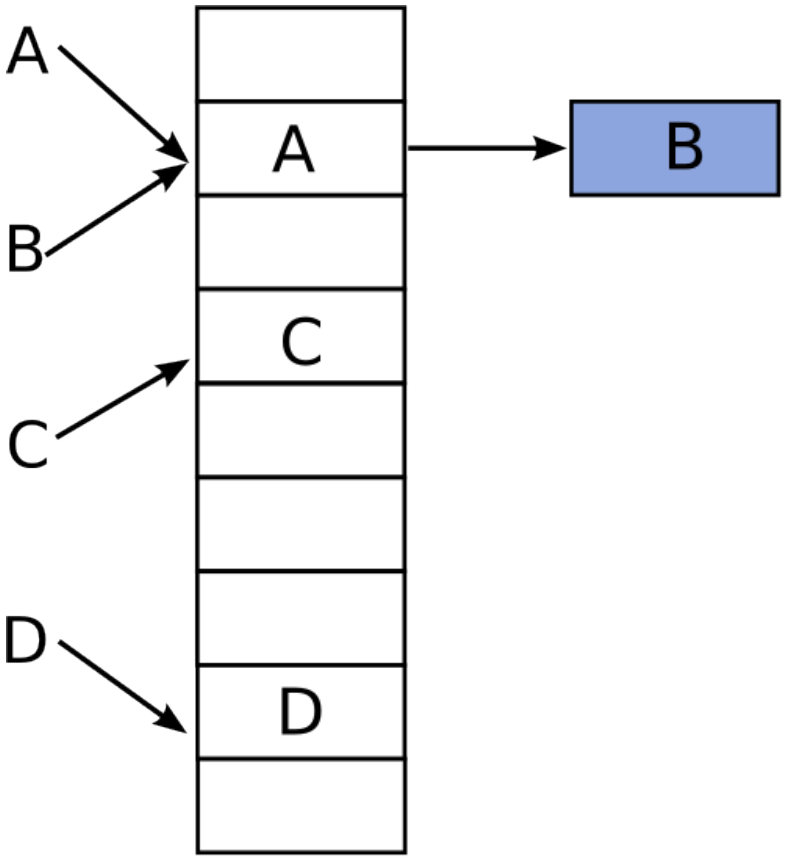
Plan

Principe

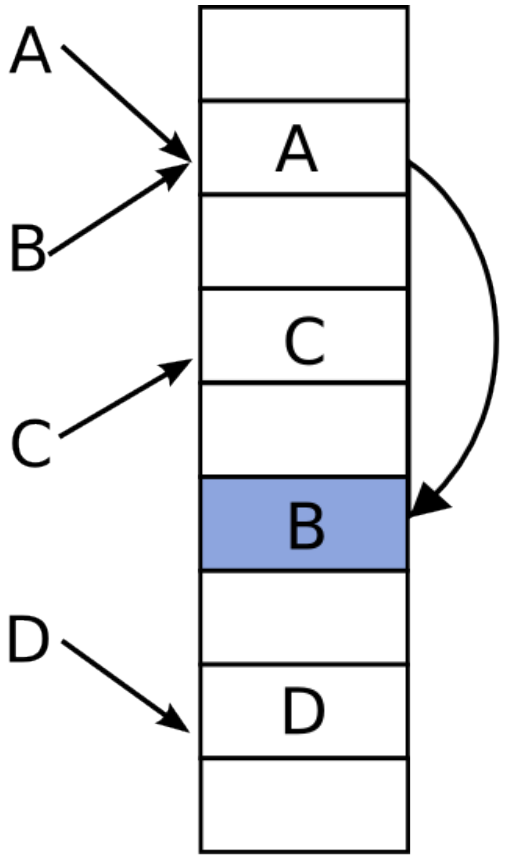
Définition

- Soit T un tableau de n éléments ($n \in \mathbb{P}$)
- On calcule une fonction de hashage (h) de chaque clé k à ajouter, la valeur de la fonction de hashage donne la position dans le tableau
- Il y a des collisions possibles: deux clés différentes k_1 et k_2 peuvent avoir la même position: $h(k_1) = h(k_2)$
- Gestion des collisions à l'extérieur du tableau (*chaînage*) ou à l'intérieur(*adressage ouvert*)

Les collisions (1/2)



Chaînage



Adressage Ouvert

Les collisions (2/2)

Résolution par chaînage

Solution la plus efficace en NLP [3]

- On stocke les collisions dans une liste chaînée
- On stocke les éléments les plus fréquemments utilisés en tête de liste

Résolution par adressage ouvert

De nombreuses méthodes : $i = 0, 1, ..., n-1 =$ numéro du sondage

- Sondage linéaire : $h(k,i) = (h'(k) + i) mod n$
- Sondage quadratique: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) mod n, c_1, c_2 \neq 0$
- Double hashage : $h(k,i) = (h_1(k) + i h_2(k)) mod n$

Birthday paradox

Définition

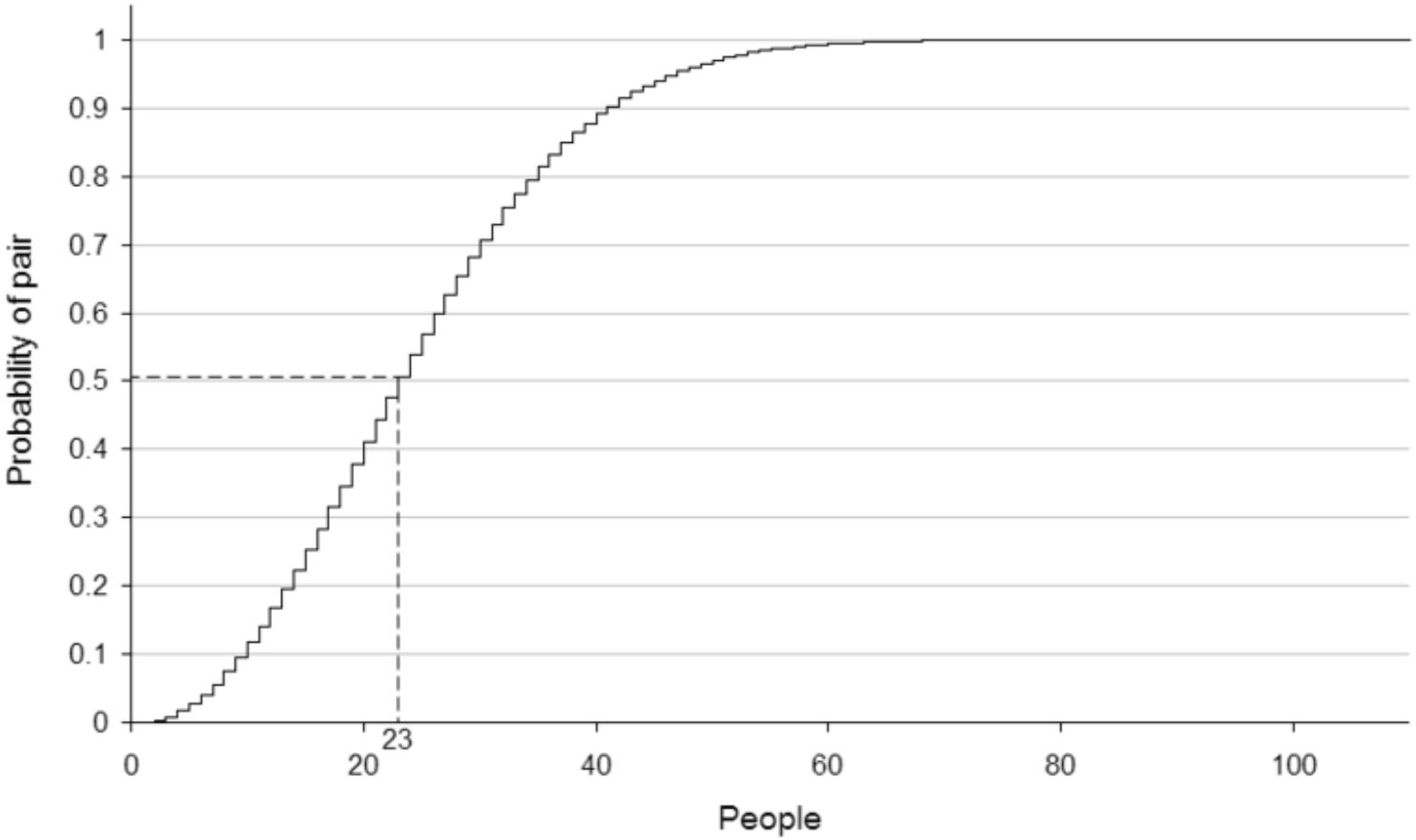
- En probabilité: le *Birthday paradox* est un cas particulier de collision (généralisable)
- Soit un groupe de N personnes choisies aléatoirement
- Quelle est la probabilité qu'au moins deux personnes aient leur anniversaire le même jour ?
- Pour 23 personnes, la probabilité est de plus de 50%
- Pour 57 personnes, la probabilité est de plus de 99%

Birthday paradox

Définition

- En probabilité: le *Birthday paradox* est un cas particulier de collision (généralisable)
- Soit un groupe de N personnes choisies aléatoirement
- Quelle est la probabilité qu'au moins deux personnes aient leur anniversaire le même jour ?
- Pour 23 personnes, la probabilité est de plus de 50%
- Pour 57 personnes, la probabilité est de plus de 99%

Birthday paradox



Tables de hashages: Caractéristiques

Complexité de la recherche/ajout

- en moyenne en $O(1)$
- dans le pire des cas en $O(N)$

Avantages

- façon simple pour représenter un ensemble (sans répétition des clés)
- Bonnes performances moyennes

Inconvénients

- Pas de compression des clés
- Pas d'ordre sur les clés, donc
  - pas de tri possible
  - pas d'algorithme autre que $get(key)$
  - pour faire une recherche via une distance d'édition, il faut énumérer le contenu de la table de hashage

Plan

Présentation

- BST: Binary Search Tree
- chaque noeud p_i contient une clé k_i de taille variable
- Beaucoup de préfixes dupliqués
- Pire des cas: liste chaînée
- Utiliser un AVL ou un Red-Back-Tree : rééquilibrage en $O(log(N))$
- Temps de recherche/ajout en $O(log(N) \times |w|$ où $|w|$ est le nombre de lettre du mot à rechercher

Conclusion

- BST pas adapté pour stocker un dictionnaire
- pour faire une recherche via une distance d'édition, il faut énumérer le contenu complet de l'arbre

Exemple

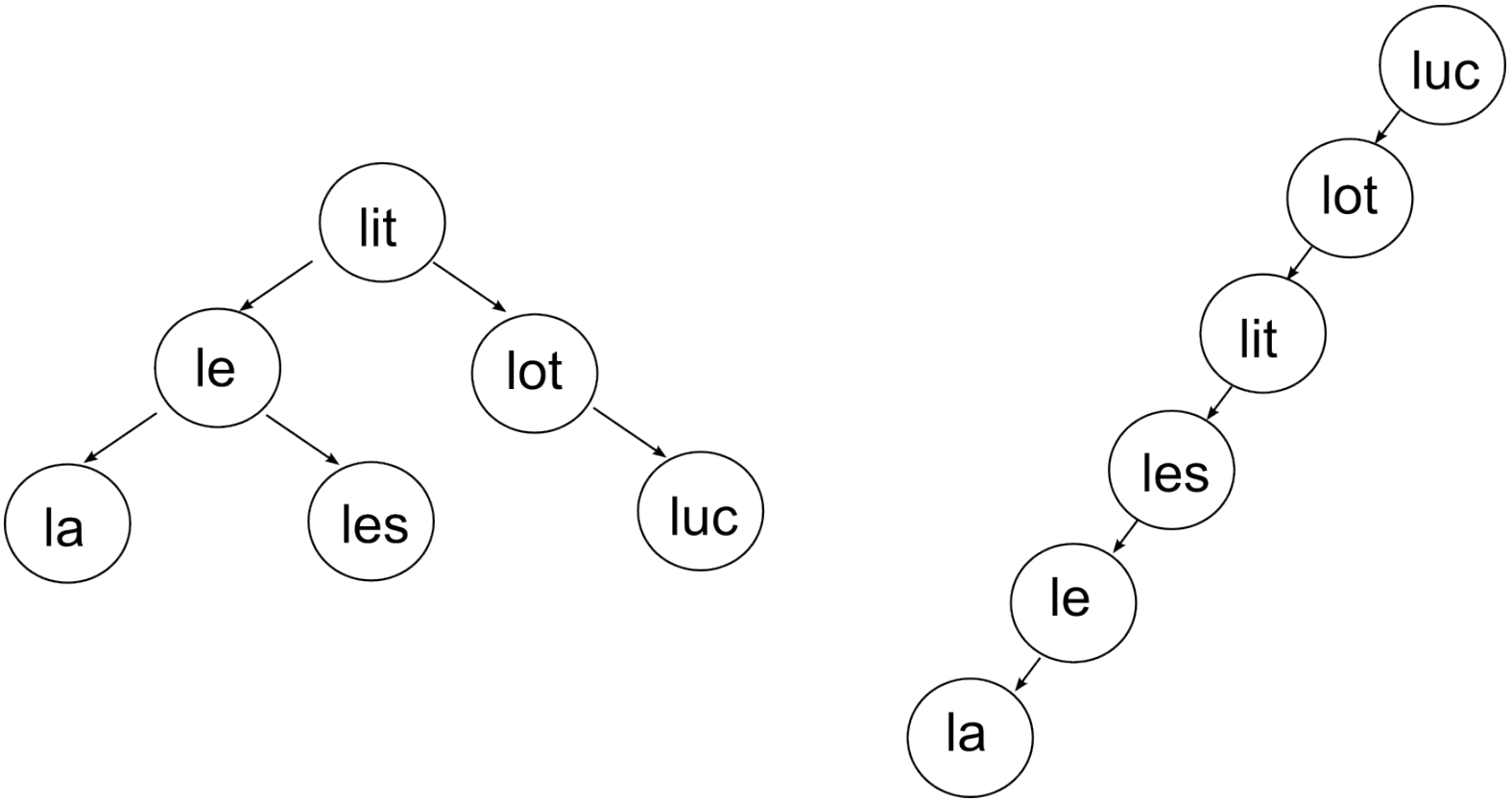


Figure: Exemple de BST avec les clés "le", "la", "les", "lit", "lot", "luc"

Plan

Principe

- Daniel D. & Tarjan, Robert E. (1985),
- Même principe qu'un arbre binaire de recherche équilibrée
- Les noeuds récemment utilisés remontent dans l'arbre (chaque lookup place l'élément recherché à la racine) : opération de *splaying*
- Peut-être vu comme un *cache*
- Plus adapté qu'un AVL/RBT pour le texte mais toujours une complexité de $O(log(N) \times |w|)$ dans le pire des cas
- Pour faire une recherche via une distance d'édition, il faut toujours énumérer le contenu complet de l'arbre

Plan

Trie: Introduction

Principe

- 1959/1960: R. de la Briandais and E. Fredkin
- Arbre N-aire adapté à la représentation du texte (Automate à états finis déterministe)
- Soit $\mathbb{A}$ un aphabet et $|\mathbb{A}|$ le nombre de symboles de cet alphabet.
- Chaque noeud contient entre 0 et $|\mathbb{A}|$ fils dont la transition est etiquetée par un symbole de $\mathbb{A}$
- Chaque préfixe est stocké une seule fois
- Plusieurs manières de stocker les pointeurs vers les noeuds fils
  - sous forme de tableau de pointeurs (contient $|\mathbb{A}|$ pointeurs). Occupe énormement de mémoire, mais offre un accès en O(1) aux noeuds fils
  - sous forme de liste chaînée
  - sous forme de table de hashage
  - ...

Exemple de Trie

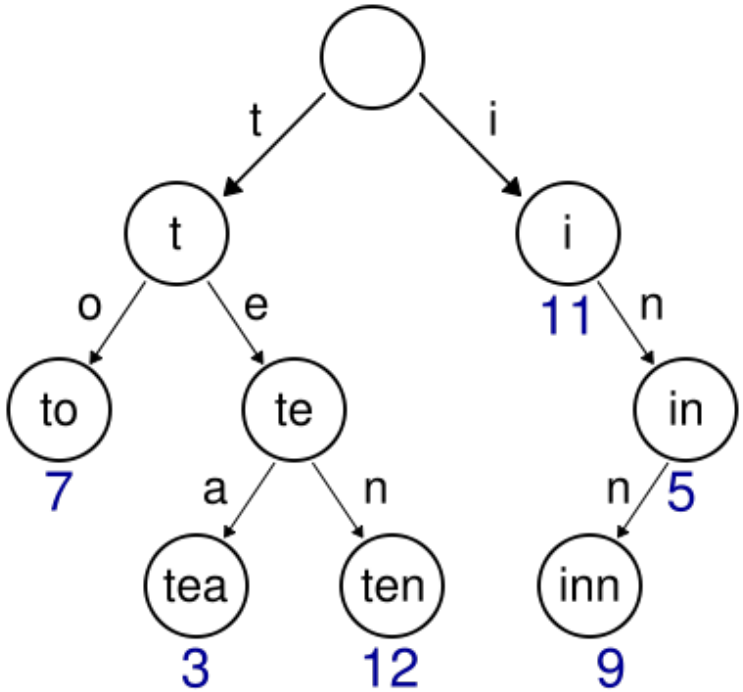


Figure: Exemple de Trie avec les clés/valeurs "to"/7, "tea"/3, "ten"/12, "i"/11, "in"/5, et "inn"/9

Caractéristiques (1/2)

Complexité

- Avec un tableau accès direct : Ajout/Recherche en $O(|w|)$ où $|w|$ est le nombre de lettre du mot à rechercher
- Avec liste chaînée : Ajout/Recherche en $O(|\mathbb{A}| \times |w|)$
- Bien faire la différence avec un arbre binaire de recherche (chaque noeud contient une clé complète : $O(|w| \times log(N))$ où N est le nombre d'éléments dans l'arbre

Questions

- Une recherche préfixe est-elle possible ? simple ?
- Une recherche approximative est-elle possible ?
- Si un noeud est à une distance $>$ seuil, Que penser de ses noeuds fils ?

Algorithme

```
int distance(noeud, char mot[1..tailleMot], float dist)
  Si (dist > maxDist) alors retourne dist
  int res = -1, mdist = -1;
  Si (noeudFinal(noeud)) alors res = tailleMot
  Si (dist + 1 < maxDist) alors
    int suppression = distance(noeud, mot[2..tailleMot], dist + 1))
    res = min(res, suppression)
    Pour (ni ∈ successeur(noeud))
      Si (tailleMot > 0 et char(ni) == mot[1]) mdist = 0 sinon mdist = 1
      int substitution = distance(ni, mot[2..tailleMot], dist + mdist)
      int insertion = distance(ni, mot[1..tailleMot], dist + 1)
      res = min(res, substitution, insertion)
  retourne res
```

recherche de “cecci” avec distance maximale de 1

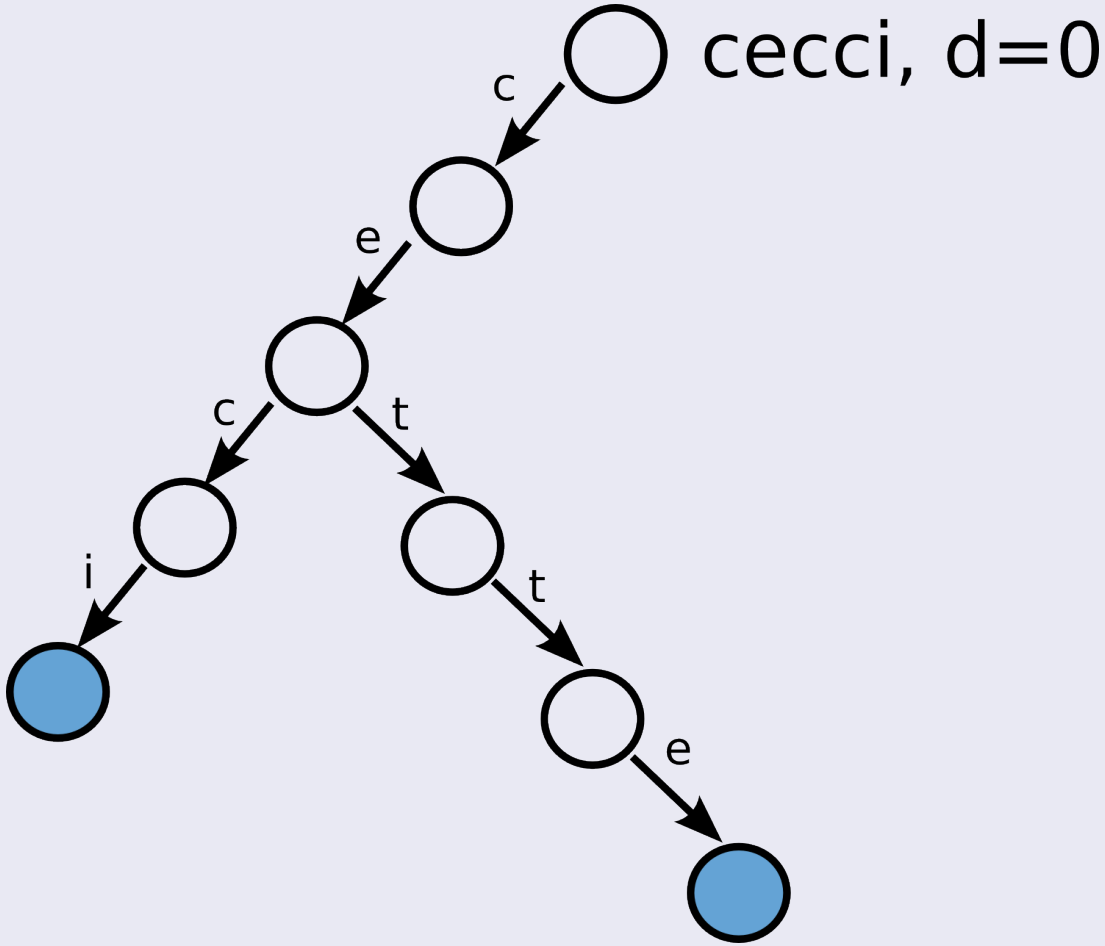


Figure: Initialisation

recherche de “cecci” avec distance maximale de 1

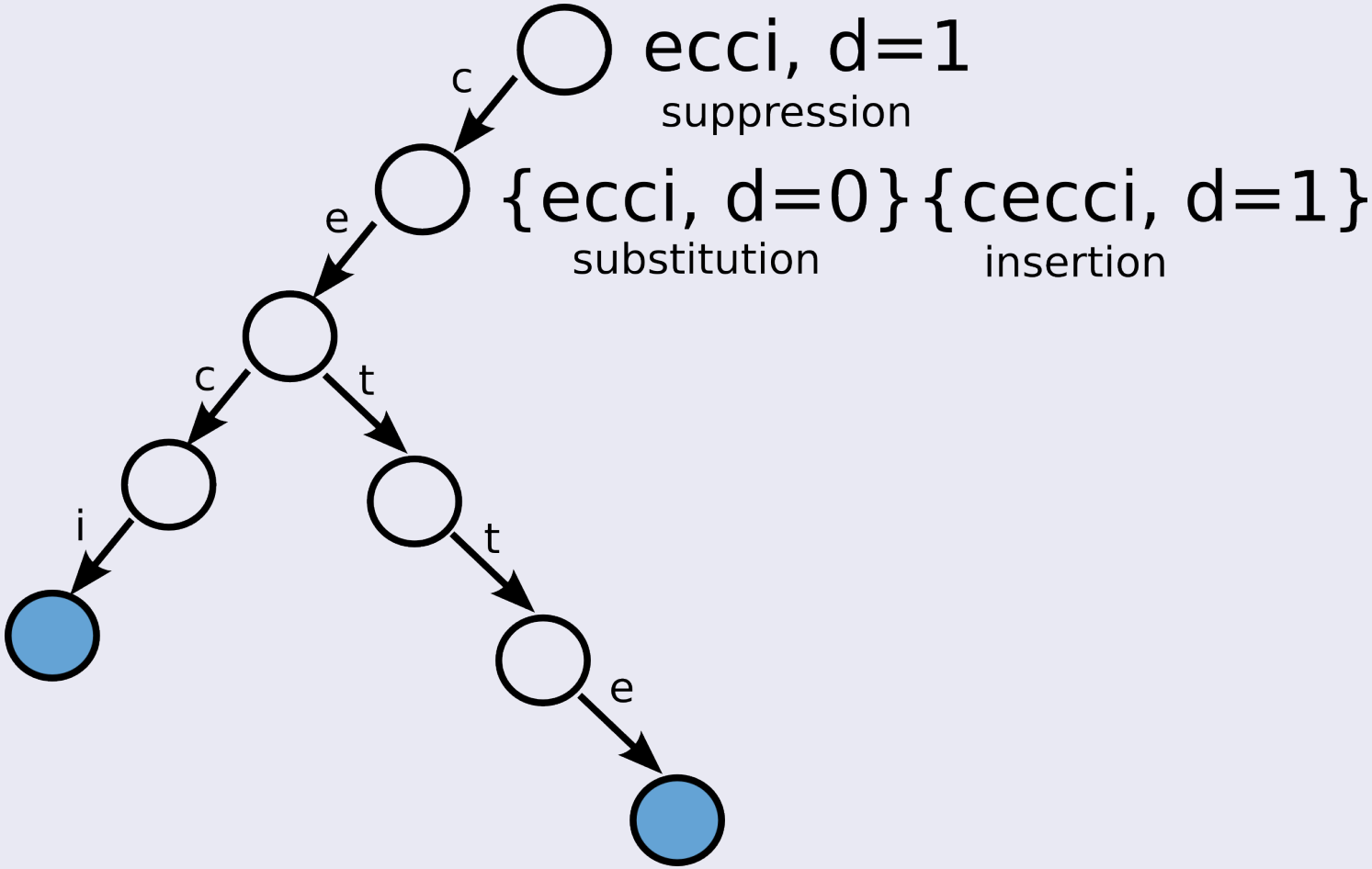


Figure: Application des 3 transformations sur la racine

recherche de “cecci” avec distance maximale de 1

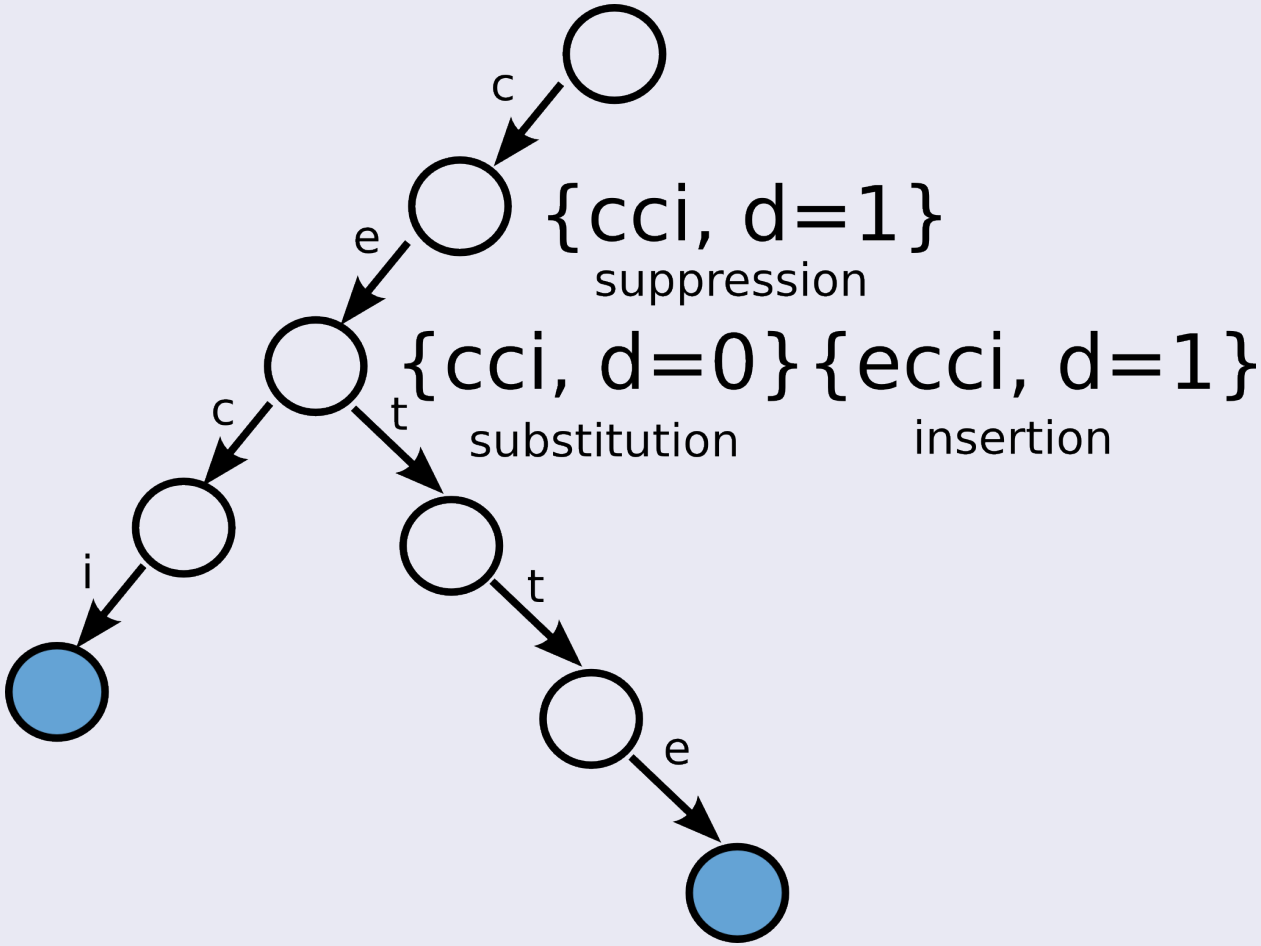


Figure: élimination des distances > 0 et applications des 3 transformations

recherche de “cecci” avec distance maximale de 1

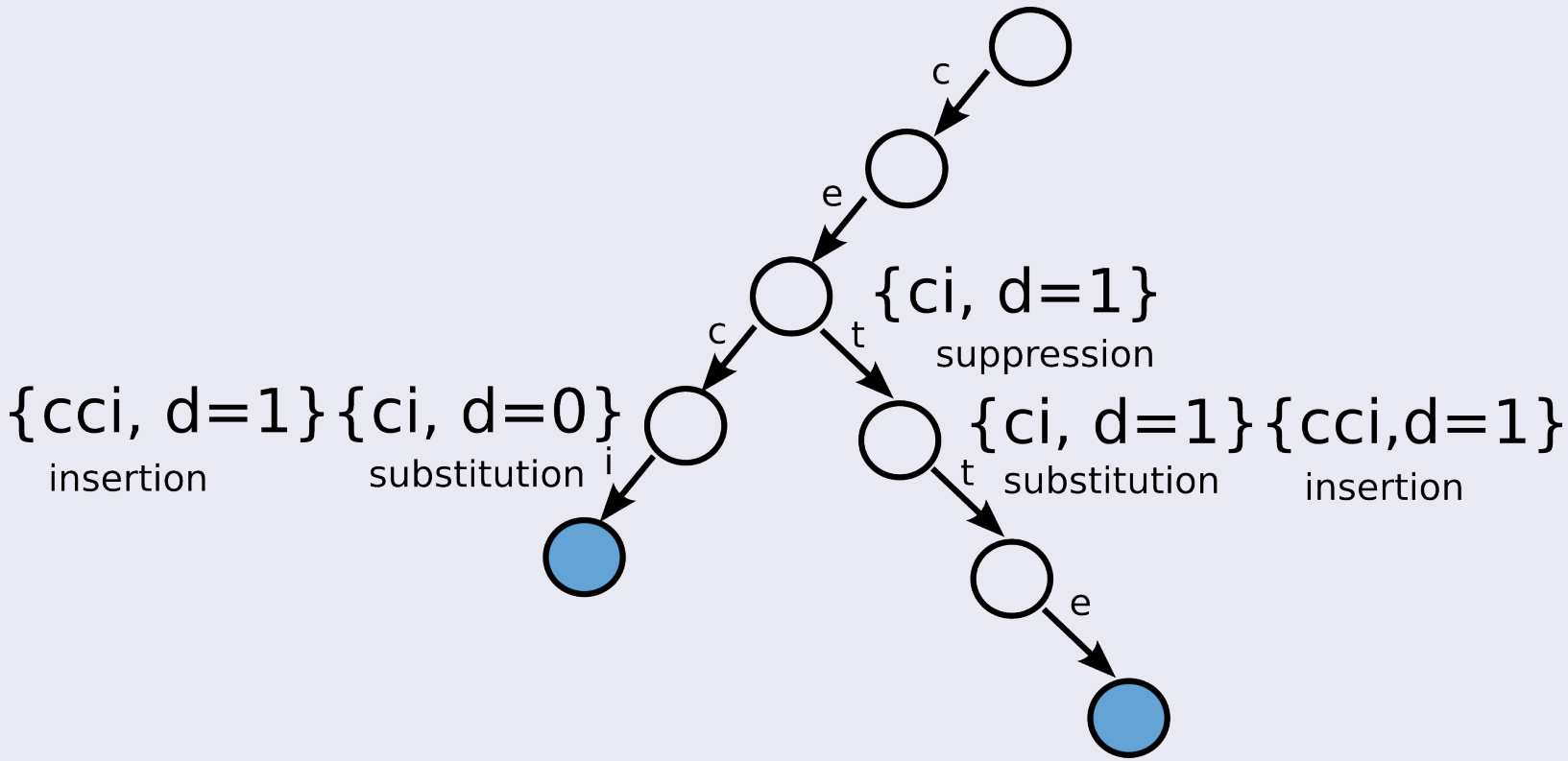


Figure: élimination des distances > 0 et applications des 3 transformations

Trie: exemple de recherche approximative

recherche de “cecci” avec distance maximale de 1

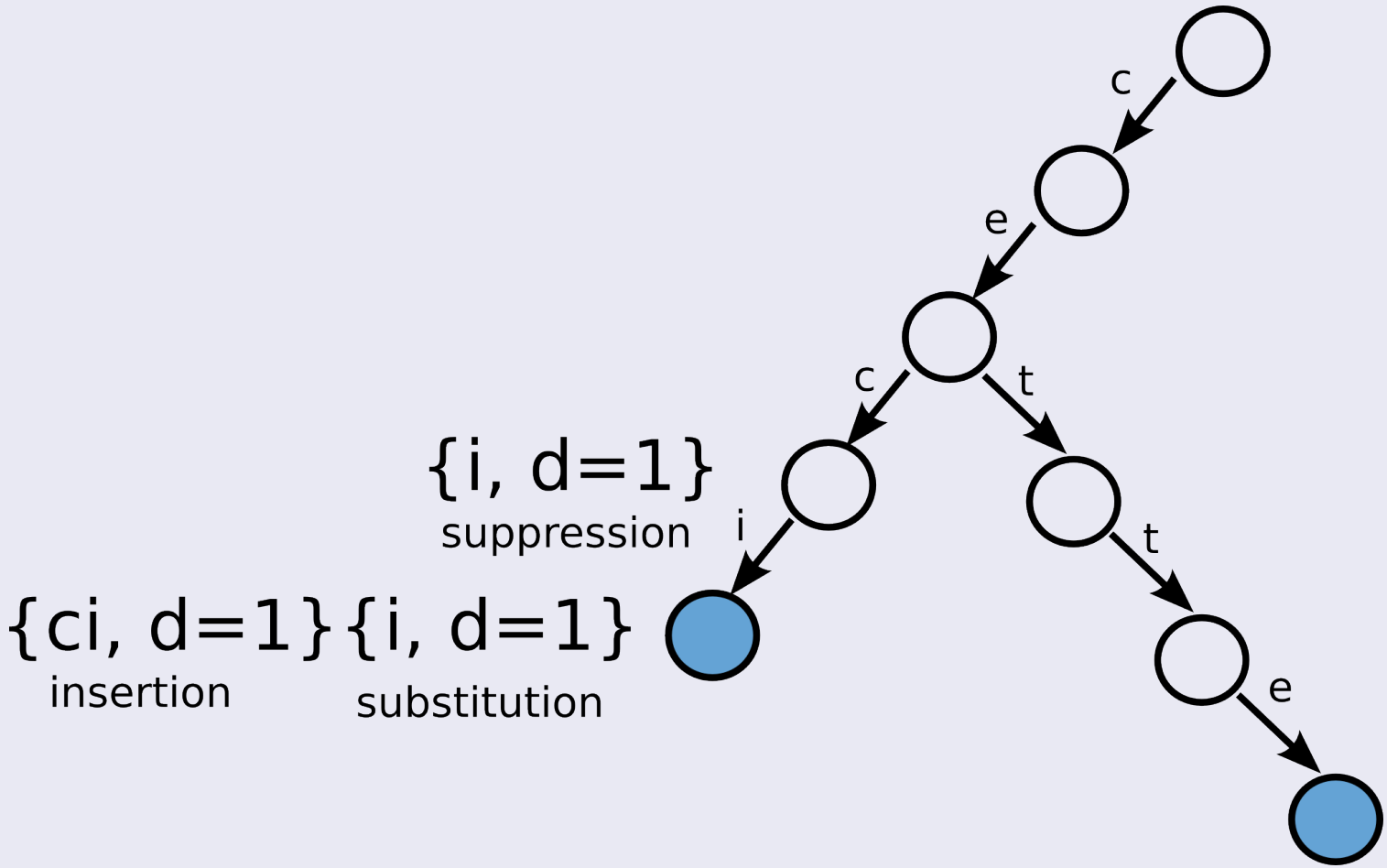


Figure: élimination des distances > 0 et applications des 3 transformations

recherche de “cecci” avec distance maximale de 1

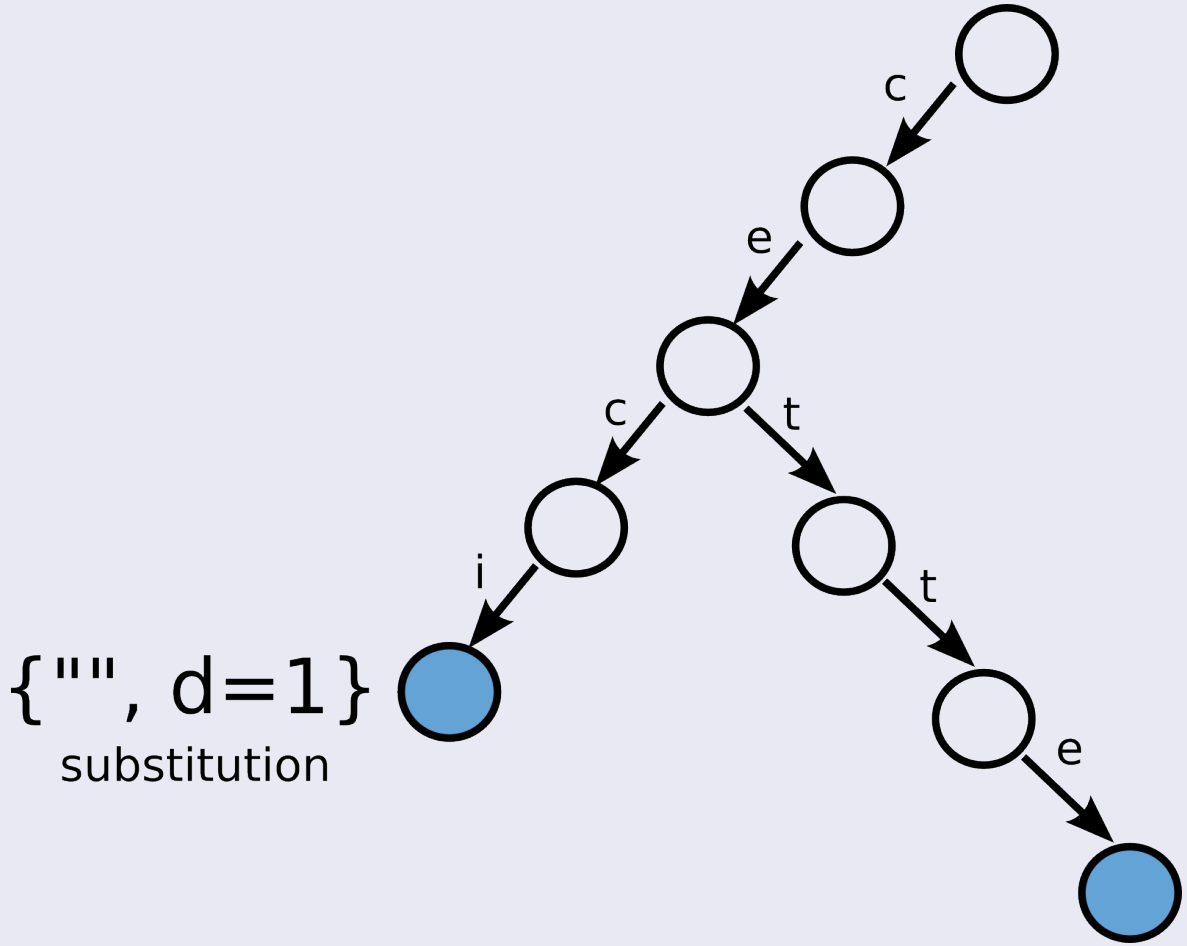


Figure: Résultat

Trie: Caractéristiques (2/2)

Avantages

- Compression préfixe : taille en $O(N)$ dans le pire des cas
- Implémentation simple
- Trié par construction
- Enumération préfixe triviale
- Nombreux algorithmes possibles : par exemple recherche approximative

Inconvénient

- Coûteux en RAM pour la version à accès direct ou lent pour la version avec liste chainée
- Coût de stockage des pointeurs importants

Suffix Tree

Définition

- Un arbre suffixe utilise une structure de type Trie pour stocker tous les suffixes des clés
- Utilisé pour pré-calculer certaines expressions régulières
- Pour une clé de taille k, on ajoute k éléments dans l'arbre suffixe

Exemple d'arbre suffixe

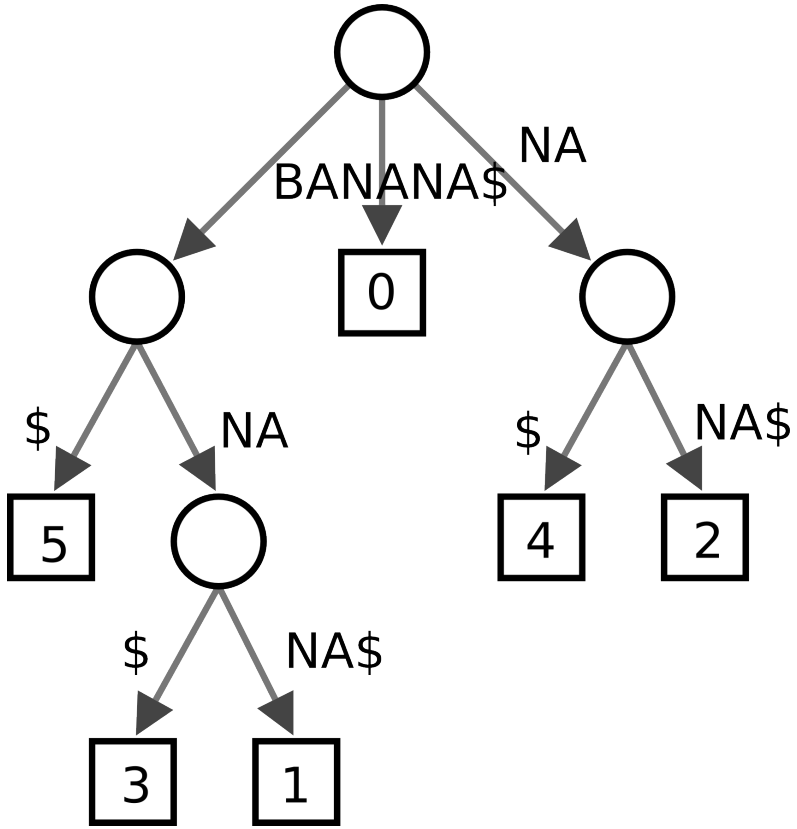


Figure: Exemple d'arbre suffixe pour la clé "BANANA"

Patricia Trie: Introduction

- Patricia Trie, radix tree ou crit bit tree (Pat Tree quand il contient des suffixes)
- Est un type de Trie
- Remplace les listes dans l'arbre par des chaînes de caractères (économie de pointeurs)
- Réduit énormément la consommation RAM sans changer les performances
- Les strings sont externes à la structure (dans un tableau). Les transitions contiennent des offset/length dans ce tableau

Patricia Trie: Exemple

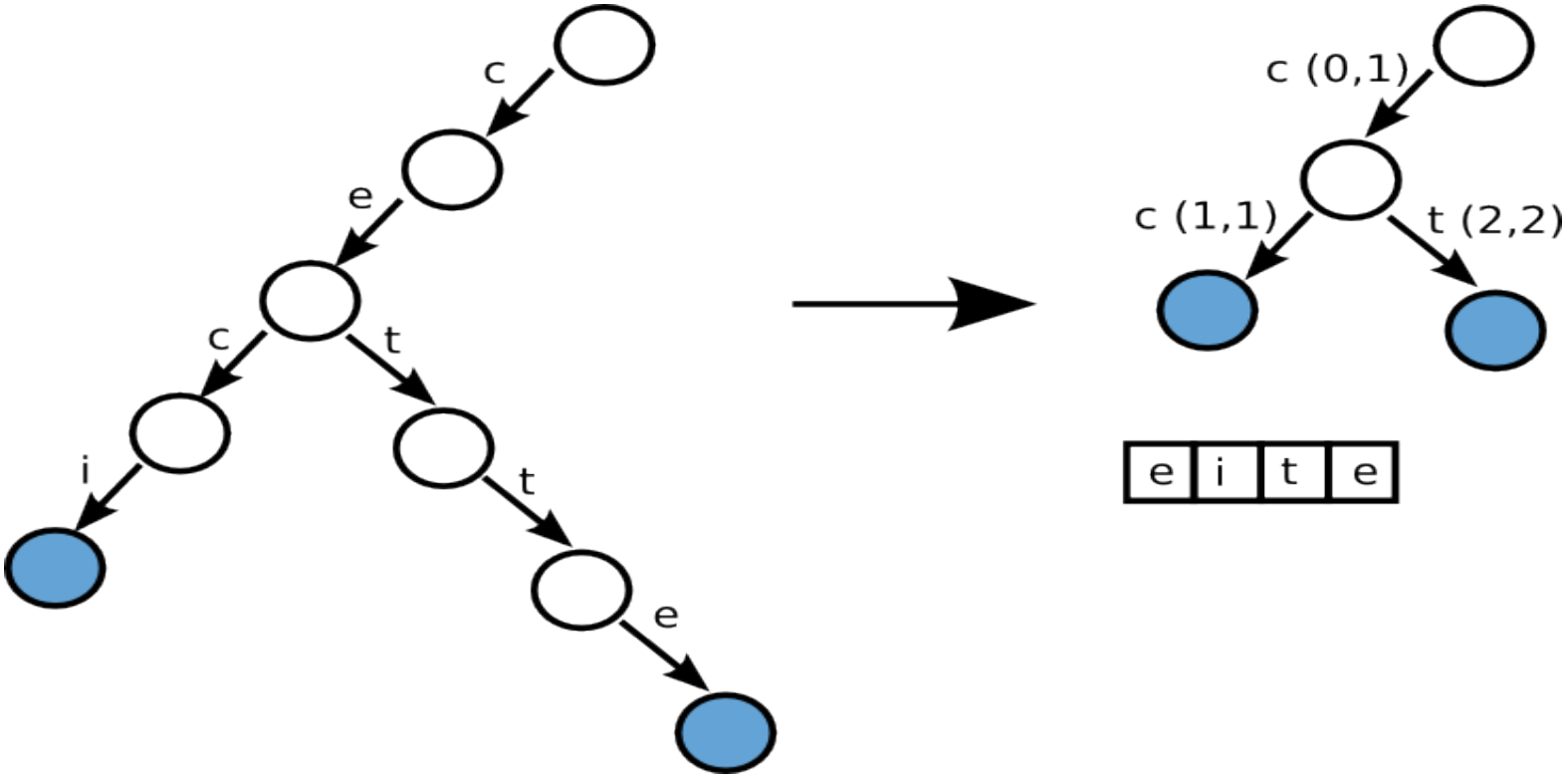


Figure: Exemple de transformation d'un Trie en Patricia Trie avec les clés "ceci" et "cette"

Patricia Trie

Avantages

- On peut stocker le tableau sur disque et la structure en RAM
- On peut contrôler le nombre de lecture dans le tableau (nombre d'I/O lorsque le vecteur est sur disque)
- Si le vecteur est sur disque, on a $O(1)$ accès disques

Inconvenients

Fragmentation du tableau

Plan

Introduction

- Jon Bentley et Robert Sedgewick, 1998
- Arbre ternaire (trois fils : f_l, f_s, f_r)
- Mélange entre un arbre binaire de recherche et un Trie
- un symbole p de $\mathbb{A}$ comme étiquette de chaque noeud
- le fils à gauche f_l signifie $< p$
- le fils à droite f_r signifie $> p$
- le fils du milieu f_s signifie $= p$
- la même compression que le Patricia Trie est possible

Exemple de TST

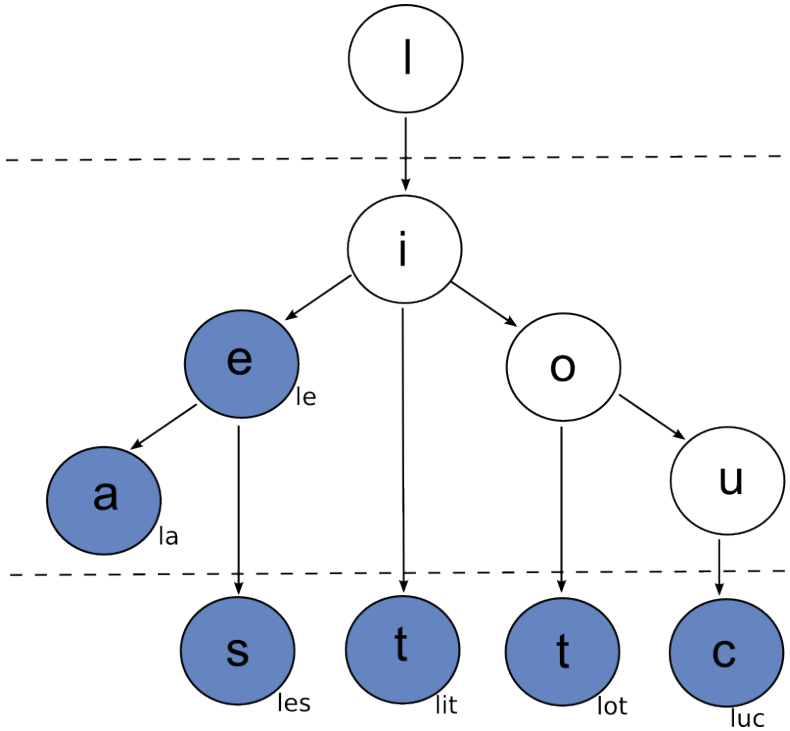


Figure: Exemple de TST avec les clés "le", "la", "les", "lit", "lot", "luc"

Caractéristiques

Complexité

Ajout/Recherche en $O(log(|(A)|) \times |w|)$ ou $|w|$ est le nombre de lettres du mot à rechercher

Avantages/Inconvénient

Les mêmes que le Patricia Trie

Plan

Introduction

- Steffen Heinz, Justin Zobel, Hugh E. Williams (2002)
- Principe:
  - Les noeuds (non feuilles) sont stockés en utilisant un Trie (basé sur un tableau de pointeurs)
  - Les feuilles sont stockées en utilisant une structure classique (le plus souvent un arbre binaire de recherche)
- Bien adapté pour stocker de nombreuses clés / valeurs
- Décision pour un noeud d'être sous forme pleine (tableau) ou creuse(BST) à partir d'une heuristique

Exemple de Burst Tree

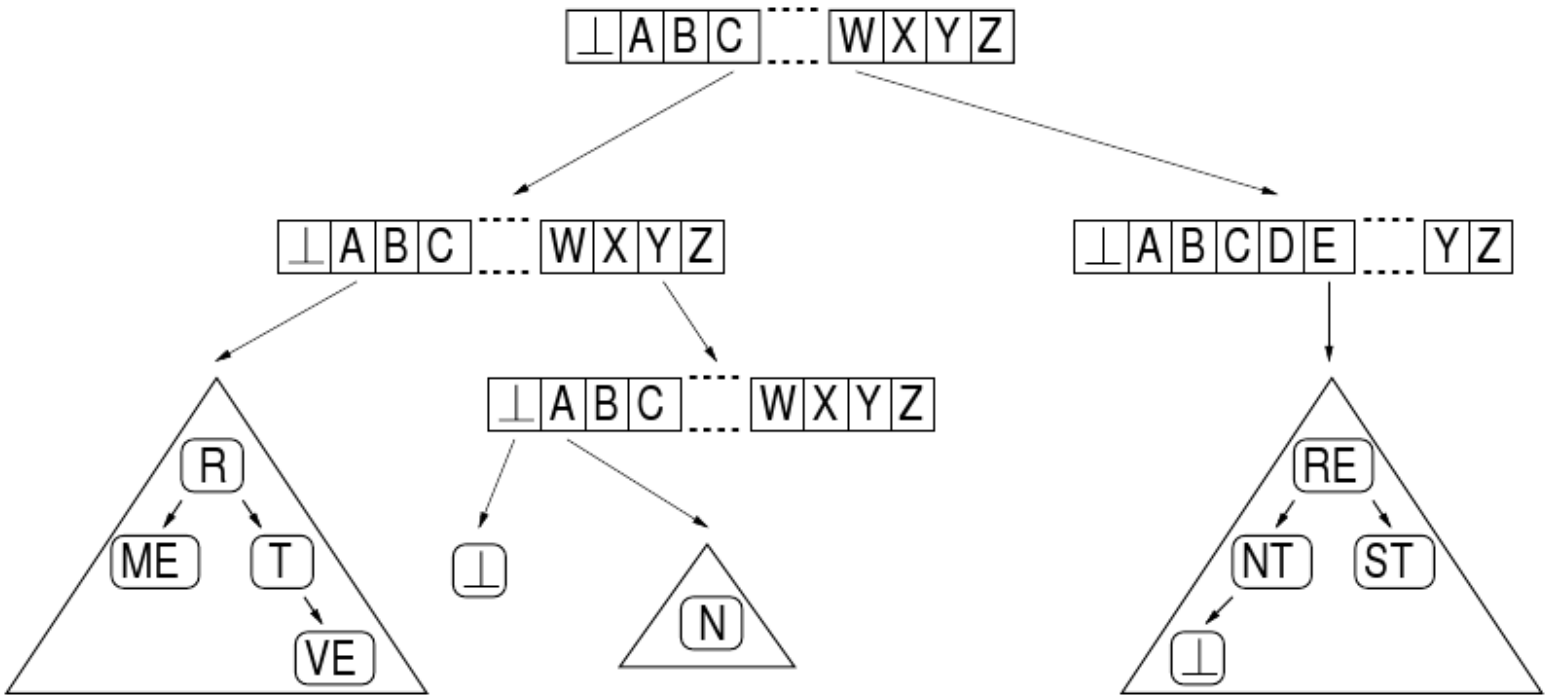


Figure: Exemple de Burst Tree avec les clés "car", "came", "cat", "cave", "cyan", "cy", "we", "were", "went", "west"

Plan

Définition

- Alan Silverstein (HP, 2002)
- Un Judy Array est un type de Trie
- Arité = 256
- Un judy Array possède trois types de noeuds :
  - Liste de pointeurs (linear)
  - vecteur de bit : 256 bits + 8 listes chaînées de 32 pointeurs maximum (bitmap)
  - vecteur de 256 pointeurs (uncompressed)
- structure complexe à implémenter (changement de type de noeud)
- Heuristiques pour déterminer le bon type de noeud

Plan

B-Tree classique

- B-Tree: possible de stocker des clés à taille variable
- noeuds contiennent des indices dans un tableau sur disque
- Il faut $O(log(B))$ accès disque par noeud !

String B-Tree

- Paolo Ferragina, Roberto Grossi (1999)
- Adapte le concept du B-Tree pour des clés de tailles variables
- Idée: chaque noeud organise les B clés sous forme de Patricie Trie
- Imaginer le matching du noeud racine sous forme de Patricia Trie, que se passe-t'il dans les cas suivants :
  - Nous sommes sur une feuile
  - Il n'y a pas de match dans le patricia trie

Patricia Trie: opérateur $<=$

Principe

- Soit un noeud intermédiaire n et ses B noeuds $n_i, i \in 0, .., B-1$ associés.
- Pour chaque noeud n_i, on extrait la clé la plus grande et on la stocke dans le noeud n avec le pointeur (RAM ou disque vers le noeud n_i)
- Il faut donc implémenter un match $<=$ dans les Patricia Trie !

Exemple de Patricia Trie dans un String B-Tree

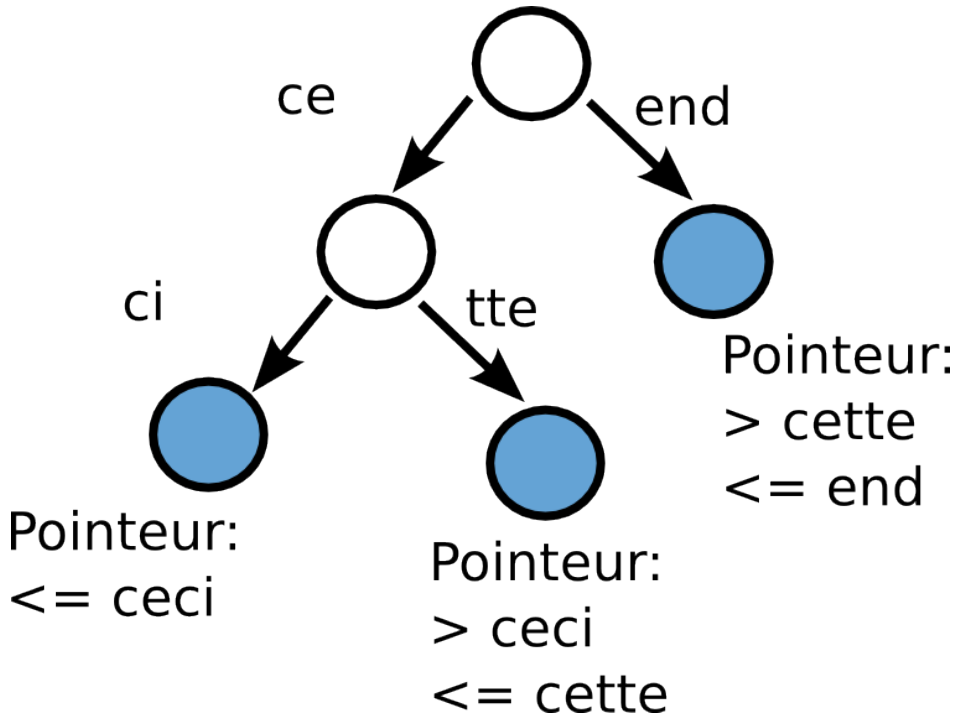


Figure: Exemple noeud intermédiaire d'un String B-Tree contenant trois fils. Comment faire la recherche *ceux* ?

Utilisation générale

- Utilise trois niveaux, les deux premiers en RAM
- Le dernier sur disque
- 1 recherche = un accès disque
- 1 ajout = deux accès disque (une lecture + une écriture)
- Ex: 1 milliard de mots: 200M en RAM + 10G sur disque

String B-Tree

Avantages

- Permet de stocker de très gros volume de clés/valeurs ($>$ 1 milliard)
- Permet de contrôler finement le nombre d'accès disque par recherche
- Garde la structure hiérarchique (ex: itérateur préfixe possible)
- Bon compromis RAM/Disque

Inconvenients

- Il faut toujours un accès disque, même quand l'élément n'existe pas

Bloom Filter

Présentation

- Burton H. Bloom (1970)
- Structure probabiliste qui stocke l'information *la clé est peut-être/pas dans la structure* sur n bits
- Au début tous les bits sont à 0
- On définit k fonctions de hashage donnant un résultat dans $[0, n-1]$

Principe

- Ajout: pour la clé c, on calcule la valeur des k fonctions de hashage et on met les bits $h_i(c), i \in [0..k-1]$ à 1
- Recherche: pour la clé c, on calcule la valeur des k fonctions de hashage :
  - Si tous les bits $h_i(c)$ sont à 1, alors la clé **est peut-être ajoutée**
  - Si un des bit $h_i(c)$ est à 0, alors la clé n'a pas été ajoutée

Quelques chiffres

- 77% de réponses correctes avec 2 bits par clé
- 90% de réponses correctes avec 3.4 bits par clé
- 99% de réponses correctes avec 9.6 bits par clé
- Cette structure ajoutée devant un String B-Tree permet de supprimer l'IO quand la clé n'existe pas dans 90% des cas avec juste 3.4 bits par clé

table de hashage parfaites

Principe

- Calculer une fonction de hashage parfaite (sans collision) pour chaque éléments (ex: gperf)
- Long à calculer pour les gros ensemble, impossibilité d'appliquer un algorithme spécial

Plan

Définition

- Objectif: serialiser le Patricia Trie dans un tableau binaire et réaliser un interpréteur
- Principe: transformer les noeuds successeurs en tableau de paires (char, offset) : recherche dichotomique
- deux implémentations possibles:
  - écriture bit par bit (compression efficace mais complexe)
  - écriture octet par octet (compression moins efficace), écriture des offset en taille variable (base 128)

Trie Compilé

Caractéristiques

- Environ 10x plus économe en RAM que la version dynamique
- Plus efficace (localité: cache CPU). Environ 4 millions de lookup par secondes
- Exemple: 45 millions de N-gram français (Clé = n-gram, valeur = fréquence), Taille: 300 MO, seulement 6.6 octets par clé/valeur

Suggest

Caractéristiques

- Objectif: commencer à tapper un préfixe et suggérer les meilleurs entrées du dictionnaire commencant par ce préfixe.
- Quelle structure de données utiliser ?

Suggest

Caractéristiques

- Trie compilé particulier
- permet de résoudre efficacement des requêtes préfixes
- Idée: avoir une table de pré-calcul dans chaque noeud

Exemple de Suggest

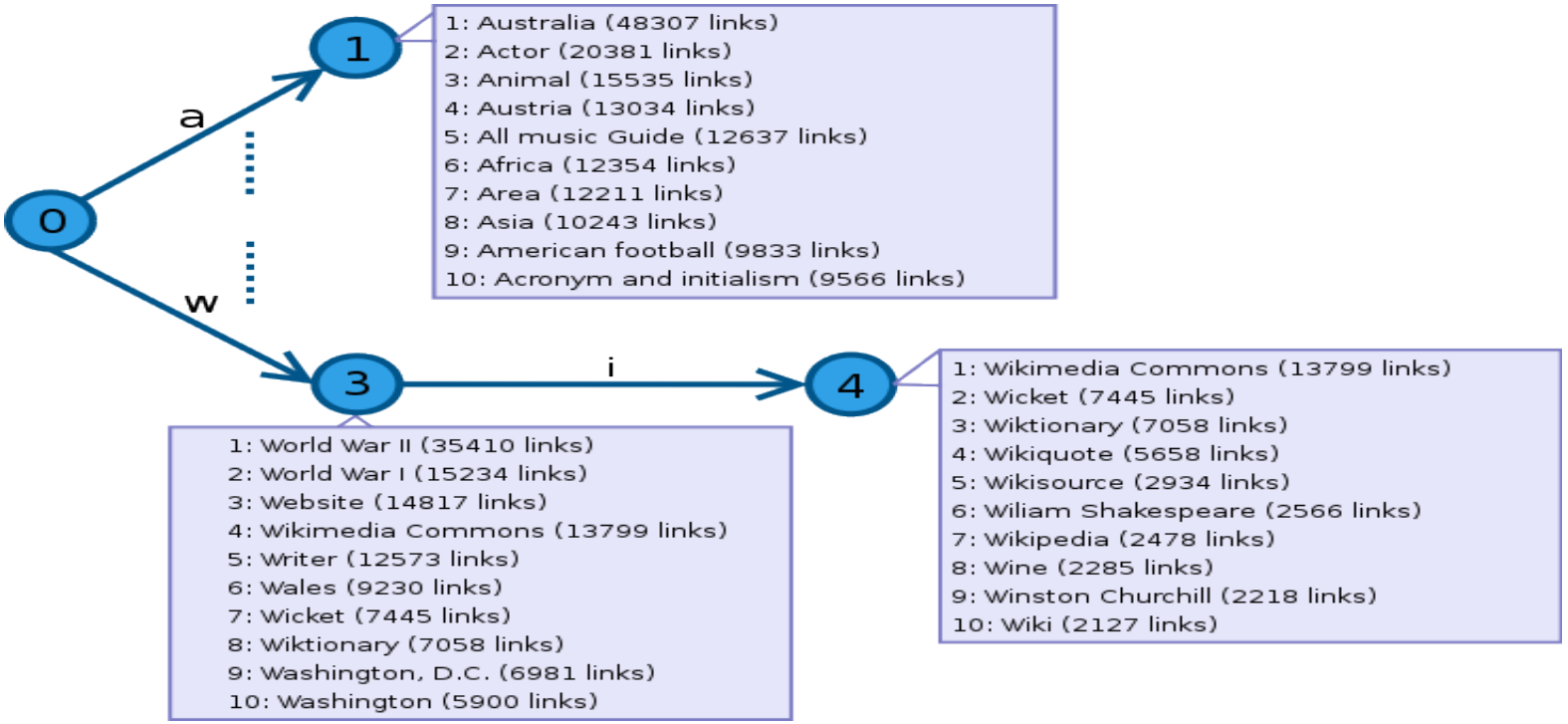


Figure: Exemple de suggest sur les titres Wikipedia (Score = fréquence)

Plan

Principe

- Même principe que la structure dynamique, mais :
  - on utilise toujours trois niveaux, les deux premiers en RAM et le dernier sur disque
  - on utilise des Trie Compilés pour représenter les noeuds (meilleur compression)
  - On peut contrôler le nombre de clés/valeurs dans chaque feuille (on contrôle la taille de l'accès disque en plus du nombre !)
  - On peut construire la structure avec très peu de RAM (à peine 10 MO)

Algorithme de construction (1/2)

- Soit K l'ensemble des clés triées à ajouter
- On part d'un noeud racine, d'un noeud intermédiaire et d'une feuille, tous vides
- On ajoute les clés par odre dans la feuille
  - Si la feuille contient maintenant P éléments, on la flush sur disque dans le fichier *feuilles* et on ajoute la dernière clé dans le noeud intermédiaire
  - Si le noeud intermédiaire contient maintenant Q éléments, on le flush dans le fichier *tête* et on ajoute la dernière clé dans le noeud racine

Algorithme de construction (2/2)

- On flush le noeud feuille, on ajoute sa dernière clé dans le noeud intermédiaire
- On flush le noeud intermédiaire et on ajoute sa dernière clé dans le noeud racine
- enfin on flush le noeud racine

Best Practices

- Essayer de ramener le problème sur une structure statique si possible (meilleurs performances)
- En fonction du volume:
  - < 1 million et uniquement besoin de faire $get(c)$: hash table
  - < 50 million: Patricia Trie / TST
  - > 50 millions: String B-Tree
- Si vous avez besoin d'un algorithme autre que $get(c)$, il faut une structure arborescente: tri, énumeration préfixe, recherche approximative, expressions régulieres ...

Bibliographie (1/2)

1. **Introduction à l'algorithmique**, Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest, Clifford Stein
2. **Automata and Dictionaries**, Denis Maurel,Franz Guenthner
3. **In-memory Hash Tables for Accumulating Text Vocabularies**, Justin Zobel. Steffen Heinz. Hugh E. Williams. Department of Computer Science, RMIT University
4. **Burst Tries: A Fast, Efficient Data Structure for String Keys**, Steffen Heinz, Justin Zobel, Hugh E. Williams
5. **Judy IV Shop Manual**, Alan Silverstein
6. **The string B-tree: a new data structure for string search in external memory and its applications**, Paolo Ferragina, Roberto Grossi
7. **Programming Pearls**, Jon Bentley

Bibliographie (2/2)

8 **A guided tour to approximate string matching.**, G. Navarro, ACM Computing Surveys (CSUR) archive 33(1), pp 31-88, 2001