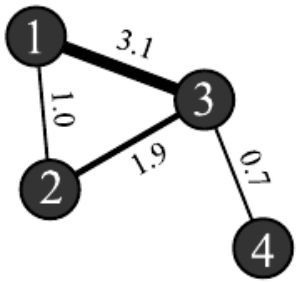
Efficient data structure for sparse network representation

Jörkki Hyvönen, Jari Saramäki et Kimmo Kaski

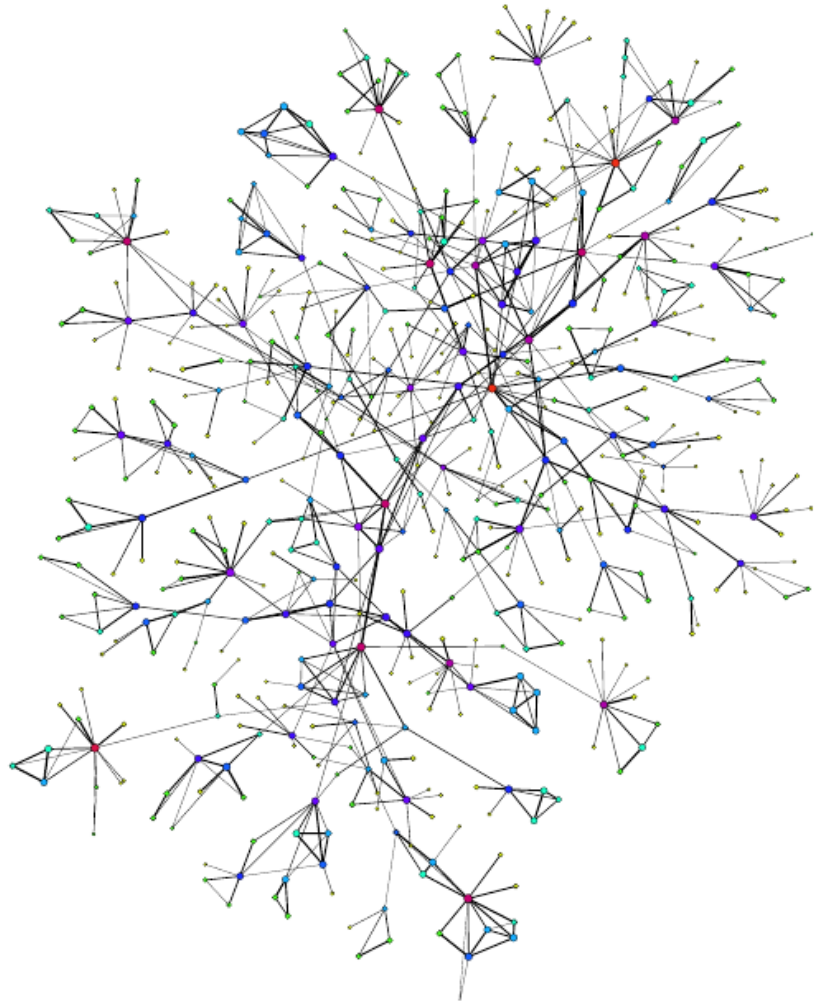
2007

Introduction



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$W = \begin{pmatrix} 0.0 & 1.0 & 3.1 & 0.0 \\ 1.0 & 0.0 & 1.9 & 0.0 \\ 3.1 & 1.9 & 0.0 & 0.7 \\ 0.0 & 0.0 & 0.7 & 0.0 \end{pmatrix}$$



Plan

Première partie I

Cahier des charges

Opérations de base 1/2

Opérations sur les graphes de terrain :

- études statistiques
- détection de structures (communautés, ...)
- restructuration du réseau (ajout, suppression de nœuds, d'arêtes)
- ...

Taille d'un graphe de terrain : 10^6 nœuds, 10^7 arêtes.

Opérations de base 2/2

Opérations sur la structure de donnée :

- ajout, suppression de nœuds, d'arêtes
- parcourt du graphe
- recherche d'arêtes
- ...

$\Rightarrow$ les opérations usuelles sur les matrices creuses sont insuffisantes.

Considérations matérielles

Utilisation optimisée du cache :

- localité temporelle
- localité spatiale

Dans un graphe, la localité correspond au voisinage $\Rightarrow$ stocker les sommets voisins d'un sommet dans des cases adjacentes de la mémoire (ce qui n'est pas réalisé avec une matrice d'adjacence).

Point de départ

Structure de donnée efficace :

- pour des objets creux
- pour l'utilisation du cache

Représentation des objets de base :

- nœuds indicés par des entiers de 0 à $N - 1$
- nœuds représentés par un tableau (cases contiguës en mémoire)
- arêtes ? ?

Mesure de qualité

Ce qui fait une bonne structure de donnée :

- occupation mémoire
- temps nécessaire aux opérations . . .

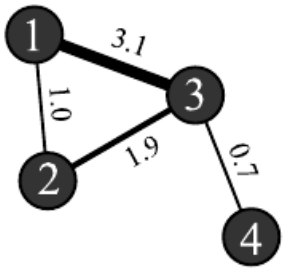
. . .ces opérations étant :

- ajout et suppression d'une arête
- recherche de l'existence et/ou du poids d'une arête
- itération d'une fonction sur tous les voisins d'un nœud

Deuxième partie II

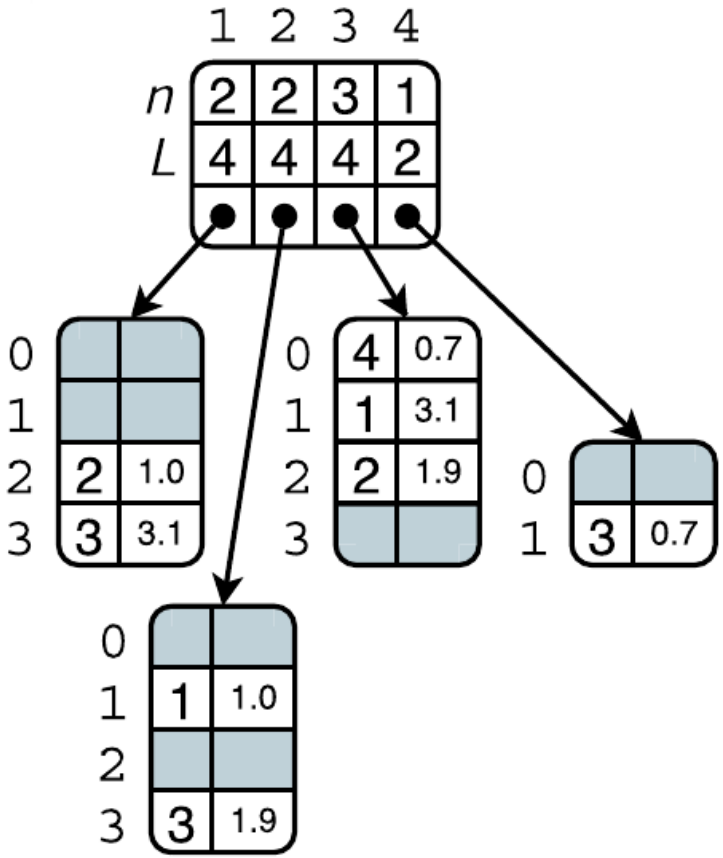
Tables de hachage

Idée



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$W = \begin{pmatrix} 0.0 & 1.0 & 3.1 & 0.0 \\ 1.0 & 0.0 & 1.9 & 0.0 \\ 3.1 & 1.9 & 0.0 & 0.7 \\ 0.0 & 0.0 & 0.7 & 0.0 \end{pmatrix}$$



Pourquoi des tables de hachage ?

- pointeurs : pas de localité spatiale
- temps moyen des opérations optimal (pas de temps réel)

Fonctions de hachage

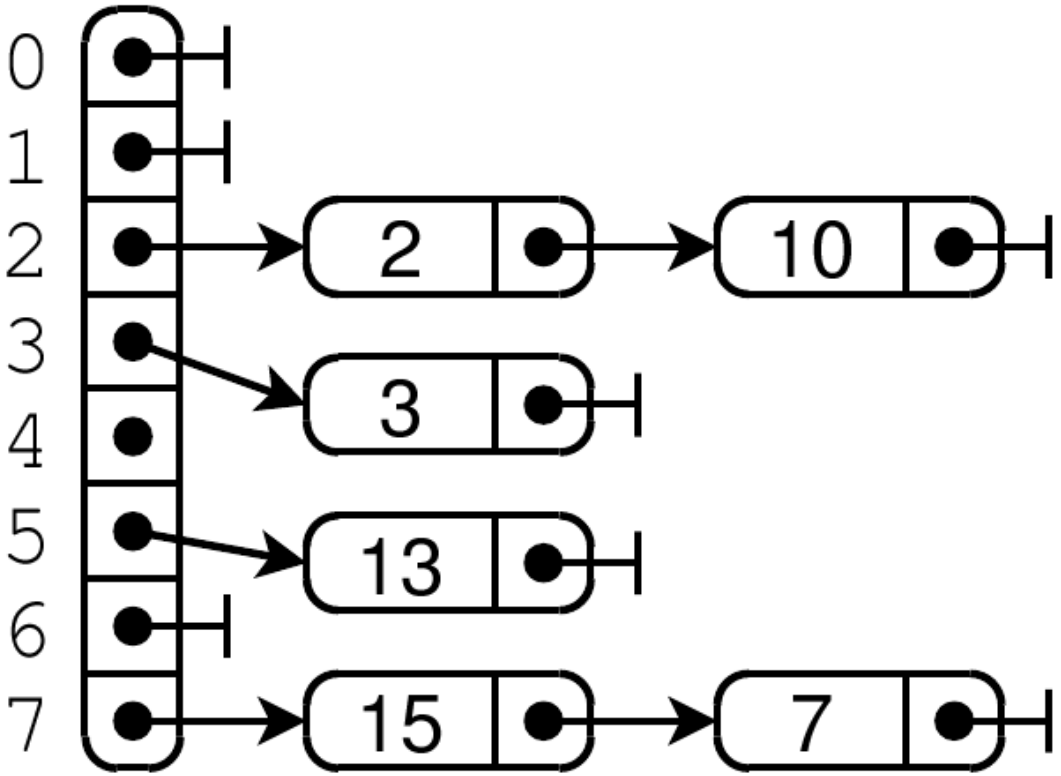
- on choisit une taille L pour chaque table de hachage
- fonction de hachage : h_L qui associe à chaque clé un entier de $[\![0, L-1]\!]$
- si la table devient trop petite, on double L
- gestion des collisions ?

Exemples de fonction de hachage :

- modulo : $h_L(k) = k \ [L]$
- fonction multiplicative :

$$h_L(k) = \frac{M}{W}(ak \ [W])$$

Gestion des collisions : liste chaînée



Gestion des collisions : adressage libre

Au lieu d'avoir **une** fonction de hachage, on en a L : $h_{1,L}, \ldots, h_{L,L}$, qui envoient chacune la même clé k sur des entrées toutes distinctes de la table.

- ajout d'un élément : on essaie $h_{1,L}, h_{2,L}, \ldots$ jusqu'à trouver une entrée libre
- suppression d'un élément : on met une marque spéciale considérée comme :
  - une entrée non libre lors d'une recherche
  - une entrée libre lors d'une insertion

Toutes les opérations (de base) ont un coût amorti en $O(1)$.

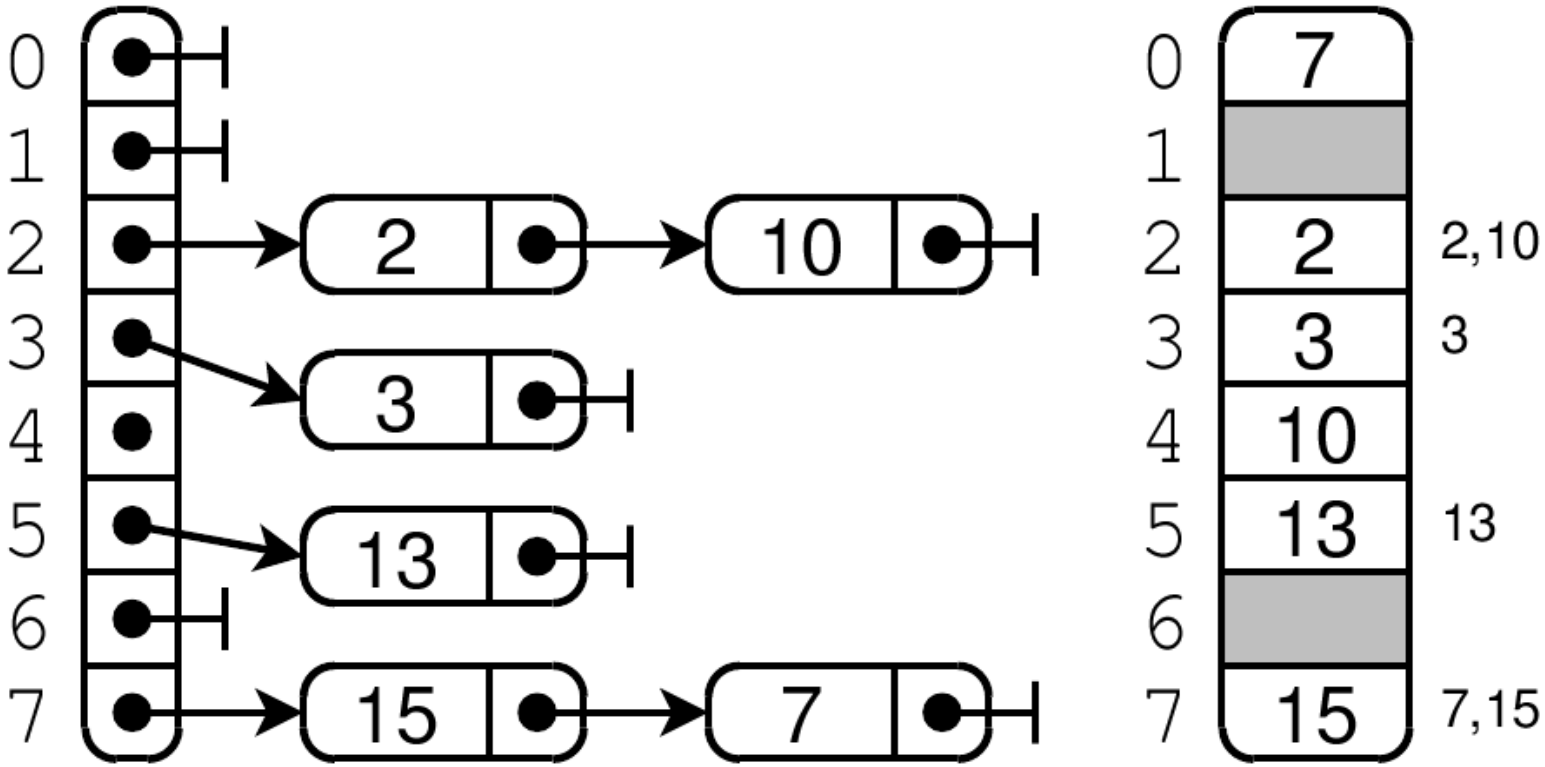
Exemple

$$L = 3$$

$$h_1 : \begin{array}{ccc} 0 & \mapsto & 0 \\ 1 & \mapsto & 0 \\ 2 & \mapsto & 2 \end{array} \qquad h_2 : \begin{array}{ccc} 0 & \mapsto & 1 \\ 1 & \mapsto & 2 \\ 2 & \mapsto & 1 \end{array} \qquad h_3 : \begin{array}{ccc} 0 & \mapsto & 2 \\ 1 & \mapsto & 1 \\ 2 & \mapsto & 0 \end{array}$$

- insertion de $0, 1, 2$
- suppression de 1
- recherche de 2

Recherche linéaire : présentation

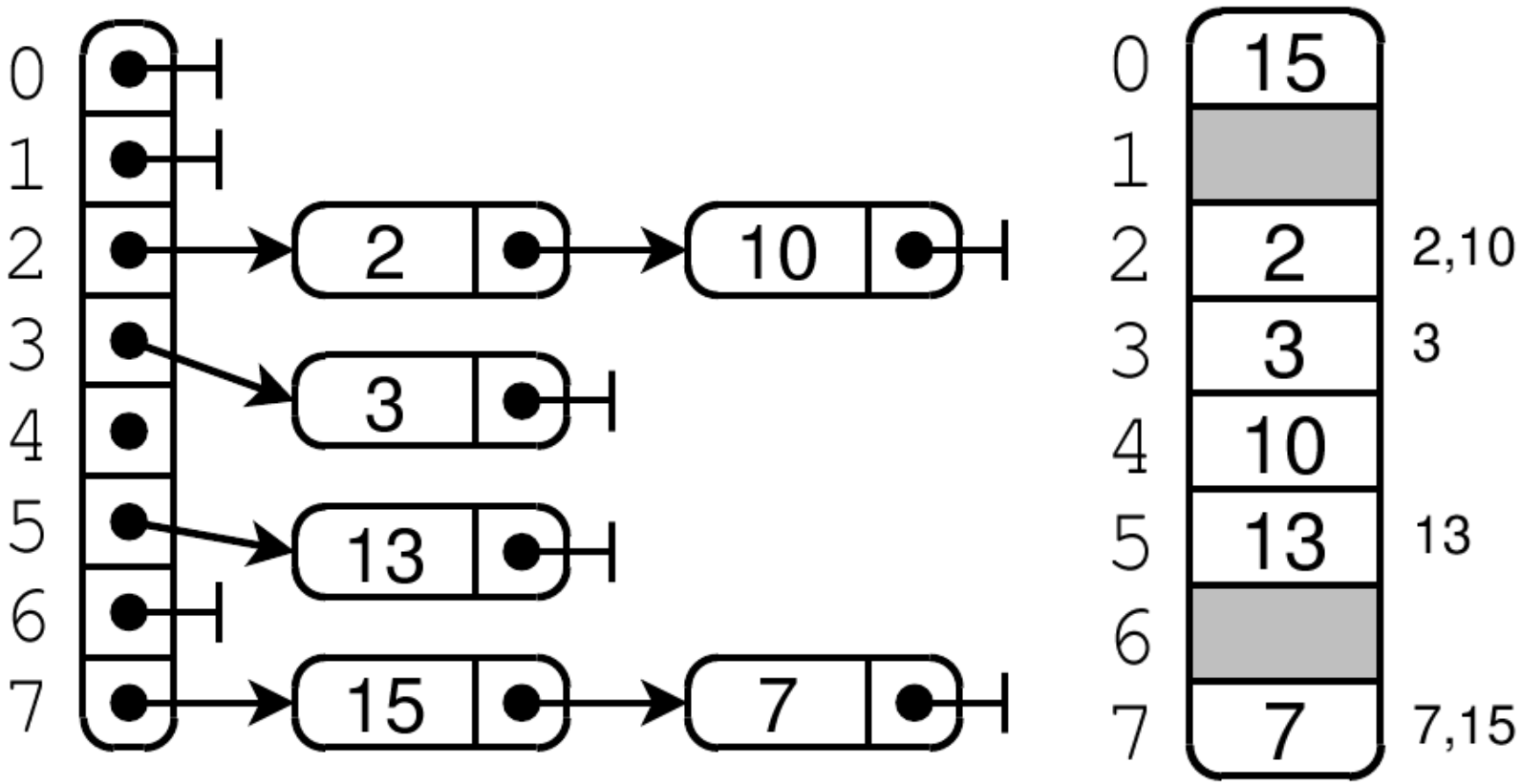


Recherche linéaire : implantation

$$\begin{cases} h_{1,L}(k) = h_L(k) \\ h_{n+1,L}(k) = (h_{n,L}(k) + 1) \ [L] \end{cases}$$

- bonne localité spatiale
- recherches :
  - réussies : $O(\sqrt{n})$
  - non réussies : $O(n)$

Recherche linéaire avec tri : présentation



Recherche linéaire avec tri : implantation

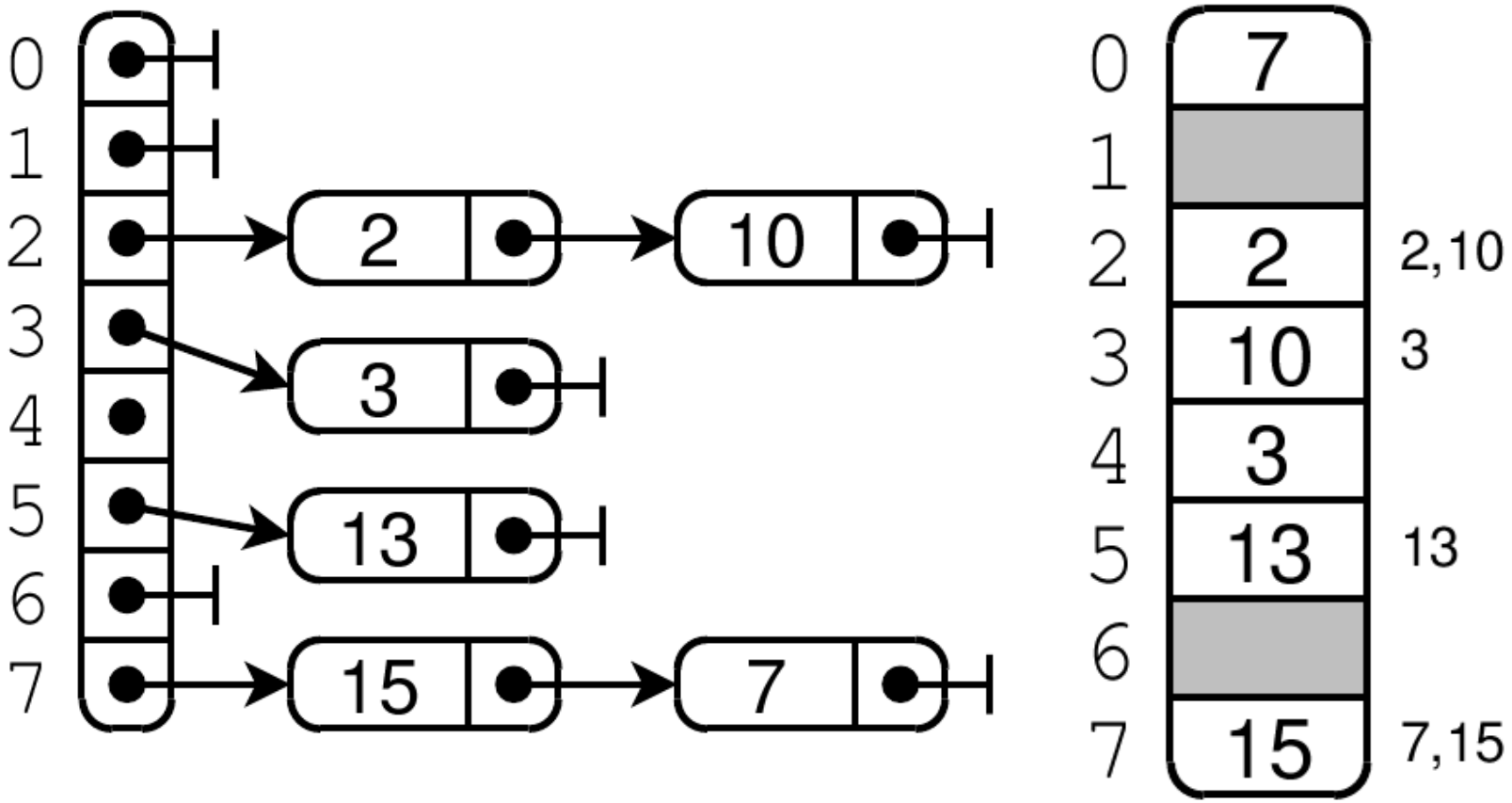
Intérêts :

- recherches réussies et non réussies avec le même temps
- on obtient la même table quel que soit l'ordre d'insertion

Inconvénient :

- insertion plus longue

Stratégie Robin Hood : présentation



Stratégie Robin Hood : implantation

On place les éléments le moins loin possible de la place initiale $h_L(k)$.

Intérêts :

- très bonne localité spatiale
- bibliothèques efficaces

Inconvénient :

- algorithmes plus complexes

Troisième partie III

Résultats expérimentaux

Dispositif expérimental

Dispositif :

- 3 GHz Pentium 4, 3 Go de mémoire
- g++ version 3.3.5 avec option -O

Fonction de hachage : multiplicative.

Comparaisons

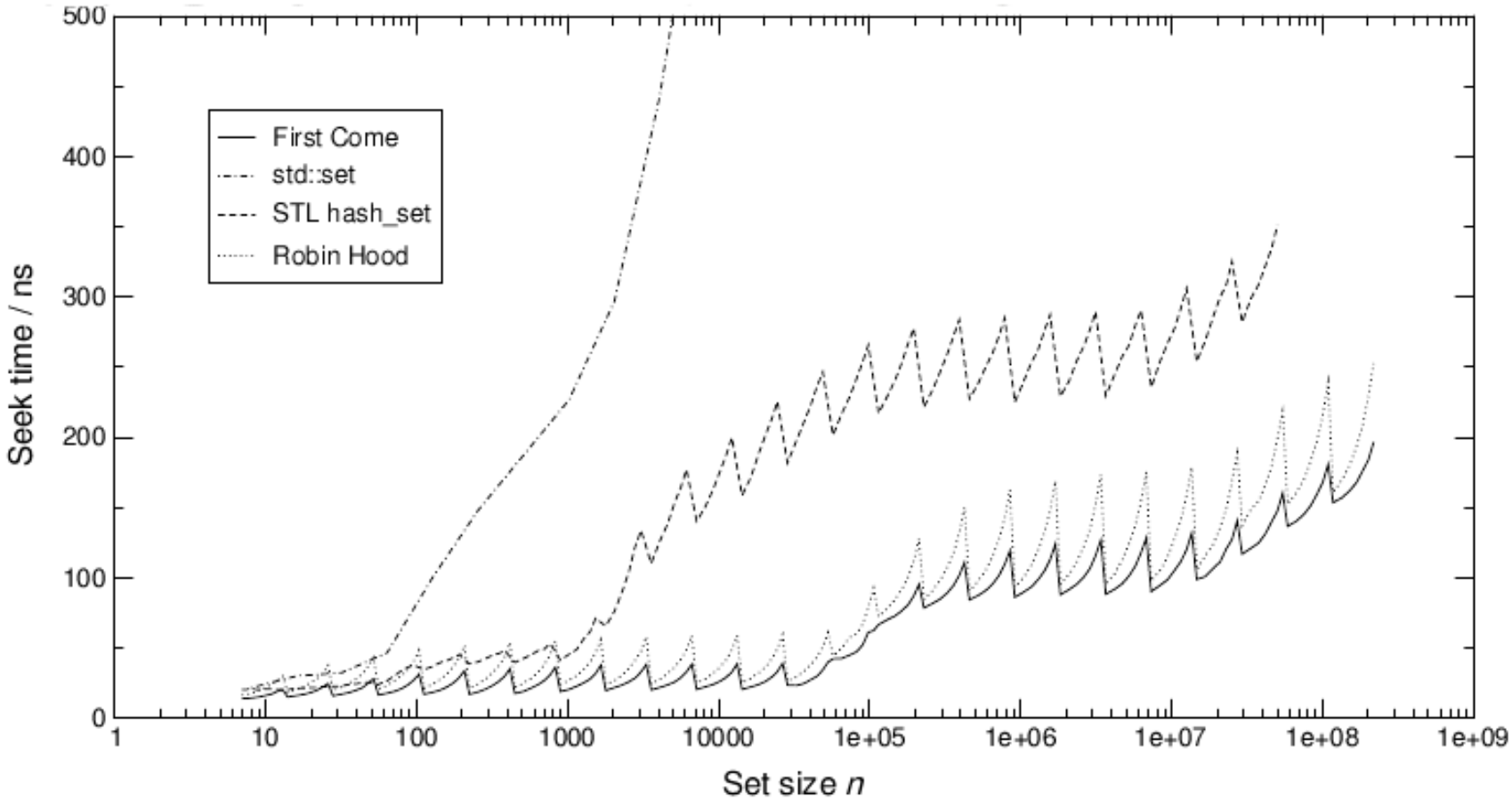
Structures de données :

- arbres rouges et noirs
- tables de hachage de la bibliothèque standard (gestion des collisions : listes chaînées)
- tables de hachage avec recherche linéaire
- tables de hachage avec la stratégie Robin Hood

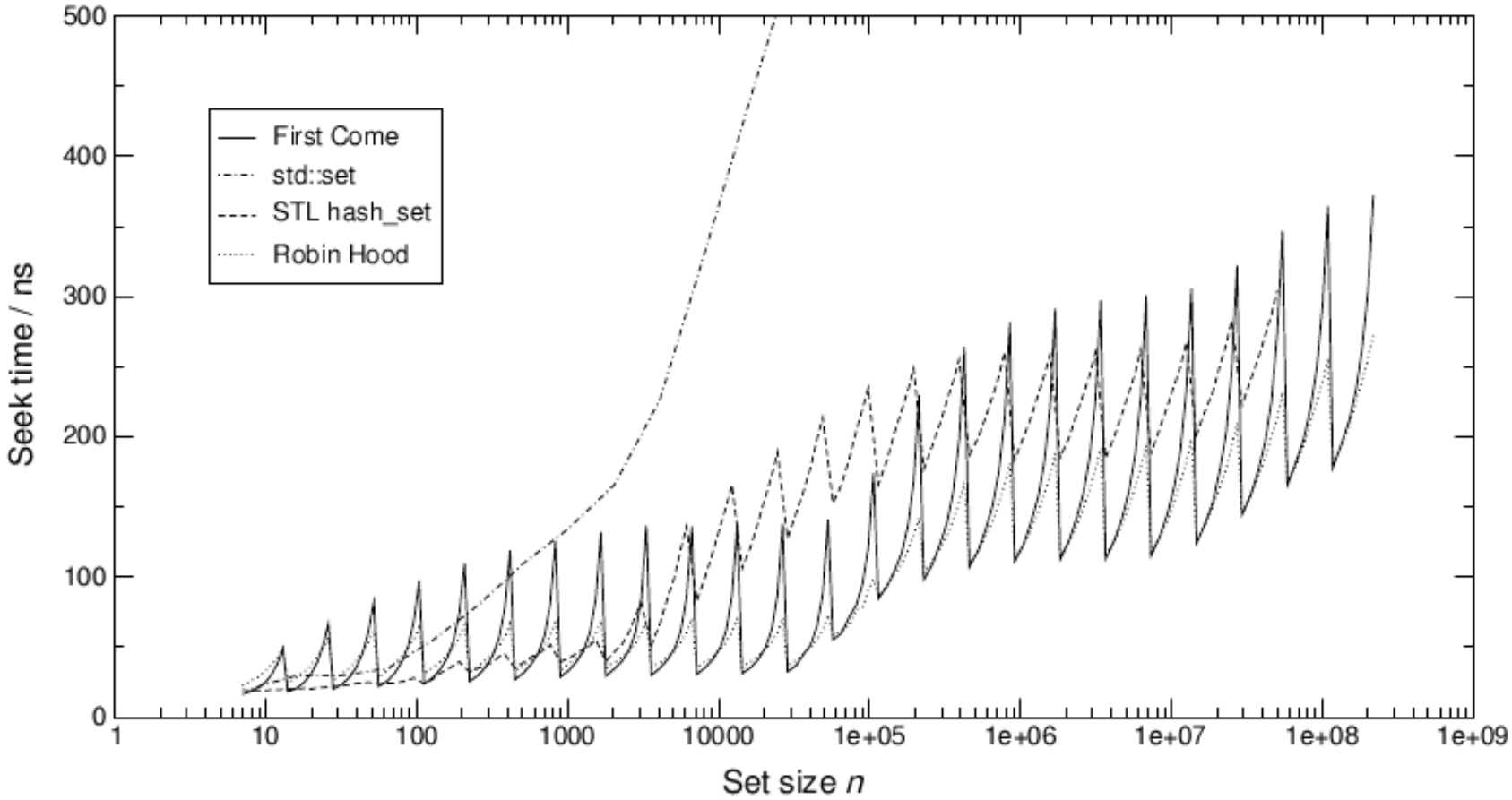
Tests :

- recherches réussies
- recherches non réussies
- encombrement mémoire

Recherches réussies



Recherches non réussies



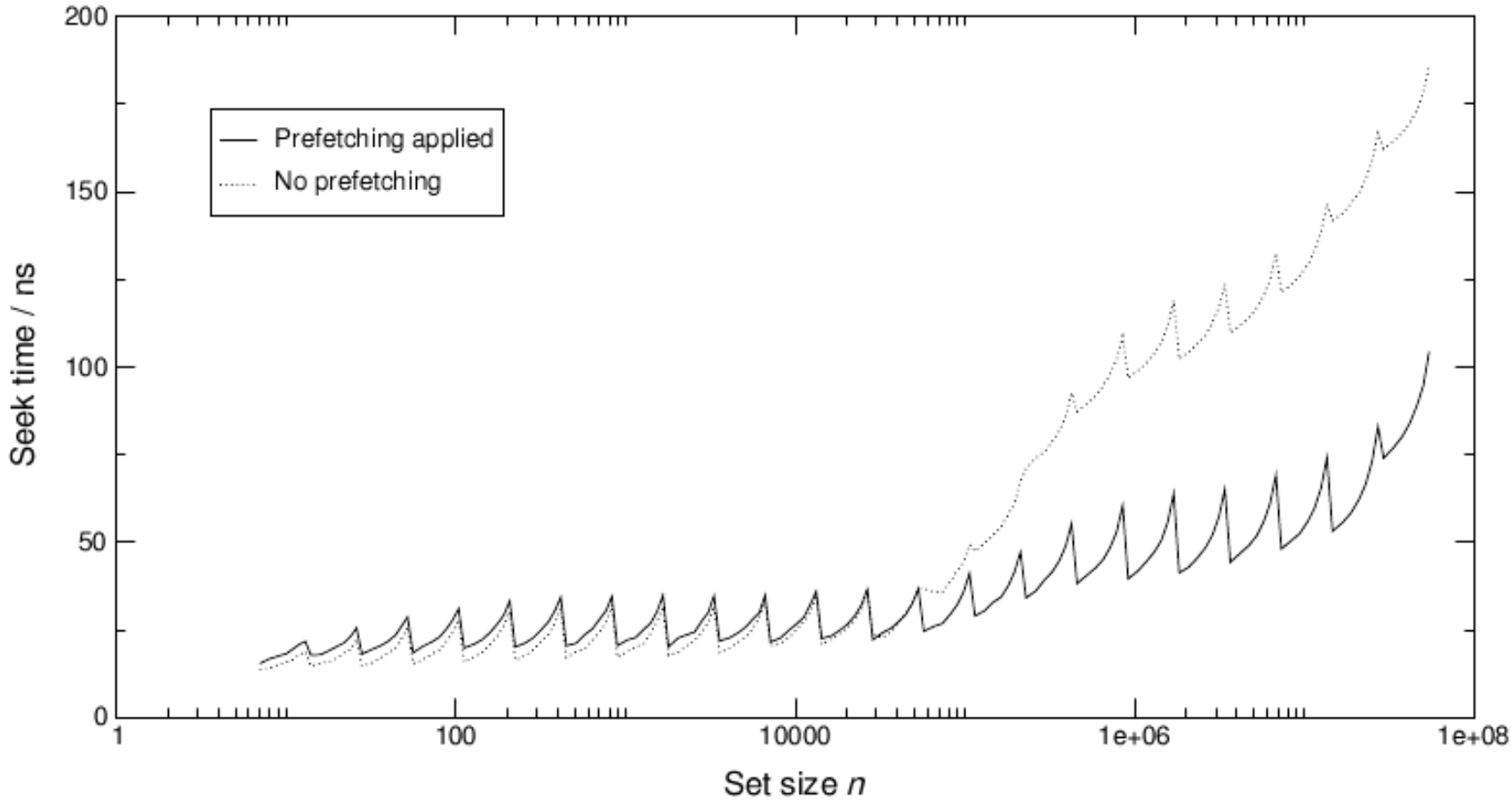
Préchargement : idée

On a constaté que, lors d'une recherche, toutes les entrées visitées tiennent dans une ligne de cache.

$\Rightarrow$ On précharge toutes les entrées visitables dans le cache.

Cela n'a d'intérêt qu'avec de l'adressage libre, pas avec les listes chaînées.

Préchargement : résultats



Encombrement mémoire

On mesure l'encombrement moyen en termes de mots machine :

Implantation	Nœuds	Arêtes
Arbres rouges et noirs	8	5
Listes chaînées	10	$6 \leqslant 5 + 1/\alpha \leqslant 7$
Recherche linéaire	5	$2 \leqslant 2/\alpha \leqslant 4$

α est le coefficient de remplissage : $\alpha = n/L$

Conclusion

Structure de donnée :

- parfaitement adaptée aux opérations à effectuer
- bonnes performances spatiales et temporelles
- tenant compte de l'architecture de l'ordinateur

Inconvénient :

- portabilité difficile

Remarques

Points positifs :

- bonne étude qui va mener sur une bibliothèque et l'ouverture à d'autres architectures (CELL)
- article de référence pour savoir comment implanter des algorithmes sur des graphes de terrain

Points négatifs :

- d'autres structures de données performantes existent, elles sont évoquées trop rapidement (en particulier, elles ne sont pas comparées lors des tests)
- dépendance à l'architecture et au langage de programmation

Merci !

Des questions ?