Introduction aux journées dynamique/online/streaming

Louis Esperet

CNRS, Laboratoire G-SCOP, Grenoble, France

Journées CALAMAR

15 janvier 2024

Graphes dynamiques

Modèle principal (fully dynamic): les n sommets du graphe sont fixés et à chaque étape une arête est ajoutée ou enlevée (à partir du graphe vide ou d'un graphe donné... un temps de préprocessing peut être nécessaire si le graphe de départ n'est pas vide).

Graphes dynamiques

Modèle principal (fully dynamic): les n sommets du graphe sont fixés et à chaque étape une arête est ajoutée ou enlevée (à partir du graphe vide ou d'un graphe donné... un temps de préprocessing peut être nécessaire si le graphe de départ n'est pas vide).

On veut maintenir une solution à un problème donné tout au long de l'évolution du graphe. par exemple, est-ce que le graphe appartient à une classe donnée, la taille d'un couplage max, un couplage max, ...

Graphes dynamiques

Modèle principal (fully dynamic): les n sommets du graphe sont fixés et à chaque étape une arête est ajoutée ou enlevée (à partir du graphe vide ou d'un graphe donné... un temps de préprocessing peut être nécessaire si le graphe de départ n'est pas vide).

On veut maintenir une solution à un problème donné tout au long de l'évolution du graphe. par exemple, est-ce que le graphe appartient à une classe donnée, la taille d'un couplage max, un couplage max, ...

On pourrait recalculer une solution optimale à chaque mise à jour mais on veut éviter ça. Idéalement, si le temps pour calculer la dernière solution directement est $T(n)$ on voudrait que le temps total de l'algo sur toute l'évolution soit $O(T(n))$.

Variantes

On peut se restreindre au

Variantes

On peut se restreindre au

- modèle incrémental (ajout d'arêtes uniquement), ou

Variantes

On peut se restreindre au

- modèle incrémental (ajout d'arêtes uniquement), ou
- modèle décrémental (suppression d'arêtes uniquement).

Variantes

On peut se restreindre au

- modèle incrémental (ajout d'arêtes uniquement), ou
- modèle décrémental (suppression d'arêtes uniquement).

On peut aussi décider d'enlever ou d'ajouter des sommets.

Variantes

On peut se restreindre au

- modèle incrémental (ajout d'arêtes uniquement), ou
- modèle décrémental (suppression d'arêtes uniquement).

On peut aussi décider d'enlever ou d'ajouter des sommets.

Pour les graphes pondérés on peut ajouter une opération où on met à jour le poids d'une arête.

Variantes

On peut se restreindre au

- modèle incrémental (ajout d'arêtes uniquement), ou
- modèle décrémental (suppression d'arêtes uniquement).

On peut aussi décider d'enlever ou d'ajouter des sommets.

Pour les graphes pondérés on peut ajouter une opération où on met à jour le poids d'une arête.

On peut considérer qu'on reçoit les modifications par paquet (batch-dynamic)

Variantes

On peut se restreindre au

- modèle incrémental (ajout d'arêtes uniquement), ou
- modèle décrémental (suppression d'arêtes uniquement).

On peut aussi décider d'enlever ou d'ajouter des sommets.

Pour les graphes pondérés on peut ajouter une opération où on met à jour le poids d'une arête.

On peut considérer qu'on reçoit les modifications par paquet (batch-dynamic)

On distingue les algos où on doit maintenir toute la solution de manière explicite à chaque modification, par exemple toute la coloration (explicit dynamic algorithm), et les algos où on la maintient de manière implicite, et on y accède par des requêtes (queries): implicit dynamic algorithm.

Objectif

- preprocessing time : le temps de calcul sur le graphe initial

Objectif

- preprocessing time : le temps de calcul sur le graphe initial
- update time : le temps (amorti ou pire cas) du maintien de la solution après une modification du graphe, et

Objectif

- preprocessing time : le temps de calcul sur le graphe initial
- update time : le temps (amorti ou pire cas) du maintien de la solution après une modification du graphe, et
- query time : le temps de calcul (amorti ou pire cas) par requête à la solution courante (dans le cas implicite).

Objectif

- preprocessing time : le temps de calcul sur le graphe initial
- update time : le temps (amorti ou pire cas) du maintien de la solution après une modification du graphe, et
- query time : le temps de calcul (amorti ou pire cas) par requête à la solution courante (dans le cas implicite).

On cherche à minimiser le temps de calcul par opération (=update+query), ou individuellement par update ou query en fonction des situations, ou on cherche des trade-off entre update et query time.

Objectif

- preprocessing time : le temps de calcul sur le graphe initial
- update time : le temps (amorti ou pire cas) du maintien de la solution après une modification du graphe, et
- query time : le temps de calcul (amorti ou pire cas) par requête à la solution courante (dans le cas implicite).

On cherche à minimiser le temps de calcul par opération (=update+query), ou individuellement par update ou query en fonction des situations, ou on cherche des trade-off entre update et query time.

L'idéal (classe Dynamic P) est un temps par modification polylog$n$, vu que l'entrée de chaque modification est de taille $O(\log n)$ (le nom de l'arête qu'on ajoute ou qu'on enlève).

Algorithmes randomisés

Pour mesurer la performance d'un algorithme il est utile d'imaginer que c'est un adversaire qui fait les modifications dans le graphe, et son but est de maximiser le temps total de l'algorithme.

Algorithmes randomisés

Pour mesurer la performance d'un algorithme il est utile d'imaginer que c'est un adversaire qui fait les modifications dans le graphe, et son but est de maximiser le temps total de l'algorithme.

On peut imaginer plusieurs types d'adversaire de plus en plus retors.

Algorithmes randomisés

Pour mesurer la performance d'un algorithme il est utile d'imaginer que c'est un adversaire qui fait les modifications dans le graphe, et son but est de maximiser le temps total de l'algorithme.

On peut imaginer plusieurs types d'adversaire de plus en plus retors.

- un adversaire oblivious, qui connaît l'algo (mais pas les bits aléatoires) et qui doit décider de toutes les updates avant que l'algo démarre

Algorithmes randomisés

Pour mesurer la performance d'un algorithme il est utile d'imaginer que c'est un adversaire qui fait les modifications dans le graphe, et son but est de maximiser le temps total de l'algorithme.

On peut imaginer plusieurs types d'adversaire de plus en plus retors.

- un adversaire oblivious, qui connaît l'algo (mais pas les bits aléatoires) et qui doit décider de toutes les updates avant que l'algo démarre
- un adversaire adaptive, qui en plus voit le résultat de l'algo sur toutes les updates $< i$ avant de décider l'update i

Algorithmes randomisés

Pour mesurer la performance d'un algorithme il est utile d'imaginer que c'est un adversaire qui fait les modifications dans le graphe, et son but est de maximiser le temps total de l'algorithme.

On peut imaginer plusieurs types d'adversaire de plus en plus retors.

- un adversaire oblivious, qui connaît l'algo (mais pas les bits aléatoires) et qui doit décider de toutes les updates avant que l'algo démarre
- un adversaire adaptive, qui en plus voit le résultat de l'algo sur toutes les updates $< i$ avant de décider l'update i
- un adversaire qui connait en plus les bits aléatoires (on en reparlera plus tard)

$(\Delta + 1)$-coloration

On veut maintenir une $(\Delta + 1)$-coloration d'un graphe de degré maximum Δ.

$(\Delta + 1)$-coloration

On veut maintenir une $(\Delta + 1)$-coloration d'un graphe de degré maximum Δ.

C'est facile de faire en temps $O(\Delta)$ par modification, de manière déterministe (dans le cas de l'ajout de l'arête uv, si par malheur $c(u) = c(v)$ il suffit de choisir pour u une couleur différente de celle de tous ses voisins.

$(\Delta + 1)$-coloration

On veut maintenir une $(\Delta + 1)$-coloration d'un graphe de degré maximum Δ.

C'est facile de faire en temps $O(\Delta)$ par modification, de manière déterministe (dans le cas de l'ajout de l'arête uv, si par malheur $c(u) = c(v)$ il suffit de choisir pour u une couleur différente de celle de tous ses voisins.

Théorème (Bhattacharya, Grandoni, Kulkarni, Liu et Solomon 2022) : Il existe une structure de donnée randomisée qui permet de maintenir une $(\Delta + 1)$-coloration d'un graphe dynamique qui après t modifications met un temps total $O(n \log n + \Delta n + t)$ en espérance et avec grande probabilité.

$(\Delta+1)$-coloration

On veut maintenir une $(\Delta+1)$-coloration d'un graphe de degré maximum Δ.

C'est facile de faire en temps $O(\Delta)$ par modification, de manière déterministe (dans le cas de l'ajout de l'arête uv, si par malheur $c(u) = c(v)$ il suffit de choisir pour u une couleur différente de celle de tous ses voisins.

Théorème (Bhattacharya, Grandoni, Kulkarni, Liu et Solomon 2022) : Il existe une structure de donnée randomisée qui permet de maintenir une $(\Delta+1)$-coloration d'un graphe dynamique qui après t modifications met un temps total $O(n \log n + \Delta n + t)$ en espérance et avec grande probabilité.

On dit que le temps amorti est $O(1)$ (ici en espérance et avec grande proba).

$(\Delta+1)$-coloration

On veut maintenir une $(\Delta+1)$-coloration d'un graphe de degré maximum Δ.

C'est facile de faire en temps $O(\Delta)$ par modification, de manière déterministe (dans le cas de l'ajout de l'arête uv, si par malheur $c(u) = c(v)$ il suffit de choisir pour u une couleur différente de celle de tous ses voisins.

Théorème (Bhattacharya, Grandoni, Kulkarni, Liu et Solomon 2022) : Il existe une structure de donnée randomisée qui permet de maintenir une $(\Delta+1)$-coloration d'un graphe dynamique qui après t modifications met un temps total $O(n \log n + \Delta n + t)$ en espérance et avec grande probabilité.

On dit que le temps amorti est $O(1)$ (ici en espérance et avec grande proba).

L'algo ne marche que contre un adversaire oblivious.

Couplage maximum

Solomon a un résultat similaire qui maintient un couplage maximal en temps amorti $O(1)$ (en espérance).

Couplage maximum

Solomon a un résultat similaire qui maintient un couplage maximal en temps amorti $O(1)$ (en espérance).

En particulier ça donne une 2-approximation pour couplage de cardinalité max.

Couplage maximum

Solomon a un résultat similaire qui maintient un couplage maximal en temps amorti $O(1)$ (en espérance).

En particulier ça donne une 2-approximation pour couplage de cardinalité max.

Si on veut un couplage maximum exact (ou même juste la taille d'un couplage maximum), alors il y a une borne inf de $\Omega(n^{1/2-\epsilon})$ par modification (sous l'hypothèse de requêtes polylogarithmiques, et moyennant une conjecture classique du domaine, OMv, sur laquelle reposent la plupart des bornes inf).

Couplage maximum

Solomon a un résultat similaire qui maintient un couplage maximal en temps amorti $O(1)$ (en espérance).

En particulier ça donne une 2-approximation pour couplage de cardinalité max.

Si on veut un couplage maximum exact (ou même juste la taille d'un couplage maximum), alors il y a une borne inf de $\Omega(n^{1/2-\epsilon})$ par modification (sous l'hypothèse de requêtes polylogarithmiques, et moyennant une conjecture classique du domaine, OMv, sur laquelle reposent la plupart des bornes inf).

OMv = online matrix-vector multiplication. On a une matrice booléenne M de taille $n \times n$, et on reçoit un par un n vecteurs booléens $v_1, \ldots, v_n$, et il faut calculer à chaque fois Mv_i immédiatement. La conjecture est qu'on ne peut pas faire ça en temps total $O(n^{3-\epsilon})$ pour une constante $\epsilon > 0$.

Treewidth

Korhonen, Majewski, Nadara, Pilipczuk, Sokolowski (2023) donnent un algorithme fully dynamic qui maintient une décomposition arborescente d'un graphe G de largeur au plus $6k + 5$, sous la promesse que la treewidth de G ne dépasse jamais k. Le temps amorti de mise à jour est $O_k(2^{\sqrt{\log n} \log \log n})$. De plus, pour toute propriété ϕ expressible en logique CMSO2, l'algorithme peut maintenir si G vérifie ϕ.

Treewidth

Korhonen, Majewski, Nadara, Pilipczuk, Sokolowski (2023) donnent un algorithme fully dynamic qui maintient une décomposition arborescente d'un graphe G de largeur au plus $6k + 5$, sous la promesse que la treewidth de G ne dépasse jamais k. Le temps amorti de mise à jour est $O_k(2^{\sqrt{\log n} \log \log n})$. De plus, pour toute propriété ϕ expressible en logique CMSO2, l'algorithme peut maintenir si G vérifie ϕ.

Résultats similaires pour la treedepth par Dvořák, Kupec, Tuma (2014), avec temps de mise à jour constant (amorti).

Algorithmes online

Load balancing. On a n tâches identiques et n processeurs en parallèle, et à chaque fois qu'une tâche arrive on doit l'envoyer sur un processeur.

Algorithmes online

Load balancing. On a n tâches identiques et n processeurs en parallèle, et à chaque fois qu'une tâche arrive on doit l'envoyer sur un processeur.

Difficulté : on ne peut demander la charge que de $O(1)$ processeurs à chaque fois, et on veut minimiser la charge maximale des processeurs.

Algorithmes online

Load balancing. On a n tâches identiques et n processeurs en parallèle, et à chaque fois qu'une tâche arrive on doit l'envoyer sur un processeur.

Difficulté : on ne peut demander la charge que de $O(1)$ processeurs à chaque fois, et on veut minimiser la charge maximale des processeurs.

- Solution 1. envoyer chaque tâche sur un processeur au hasard (charge $\Theta(\log n / \log \log n)$)

Algorithmes online

Load balancing. On a n tâches identiques et n processeurs en parallèle, et à chaque fois qu'une tâche arrive on doit l'envoyer sur un processeur.

Difficulté : on ne peut demander la charge que de $O(1)$ processeurs à chaque fois, et on veut minimiser la charge maximale des processeurs.

- Solution 1. envoyer chaque tâche sur un processeur au hasard (charge $\Theta(\log n / \log \log n)$)
- Solution 2. tirer 2 processeurs au hasard et envoyer la tâche sur le moins chargé des deux (charge $\Theta(\log \log n)$)

Algorithmes online

Load balancing. On a n tâches identiques et n processeurs en parallèle, et à chaque fois qu'une tâche arrive on doit l'envoyer sur un processeur.

Difficulté : on ne peut demander la charge que de $O(1)$ processeurs à chaque fois, et on veut minimiser la charge maximale des processeurs.

- Solution 1. envoyer chaque tâche sur un processeur au hasard (charge $\Theta(\log n / \log \log n)$)
- Solution 2. tirer 2 processeurs au hasard et envoyer la tâche sur le moins chargé des deux (charge $\Theta(\log \log n)$)

"Balls and bins", "Power of two choices"

Problème des secrétaires

On a n candidats pour un seul travail, à qui on fait passer des entretiens dans un ordre aléatoire uniforme.

Problème des secrétaires

On a n candidats pour un seul travail, à qui on fait passer des entretiens dans un ordre aléatoire uniforme.

À la fin de chaque entretien on doit décider si on prend ou non le candidat, et le poste est alors pourvu immédiatement. Notre but est de trouver une stratégie qui maximise la probabilité de sélectionner le meilleur candidat.

Problème des secrétaires

On a n candidats pour un seul travail, à qui on fait passer des entretiens dans un ordre aléatoire uniforme.

À la fin de chaque entretien on doit décider si on prend ou non le candidat, et le poste est alors pourvu immédiatement. Notre but est de trouver une stratégie qui maximise la probabilité de sélectionner le meilleur candidat.

Solution optimale : écouter poliment les n/e premiers candidats puis ensuite sélectionner le premier candidat qui est meilleur que tous les précédents (ou le dernier candidat sinon). Proba de succès $1/e \approx 37\%$ (et c'est optimal)

Problème des secrétaires

On a n candidats pour un seul travail, à qui on fait passer des entretiens dans un ordre aléatoire uniforme.

À la fin de chaque entretien on doit décider si on prend ou non le candidat, et le poste est alors pourvu immédiatement. Notre but est de trouver une stratégie qui maximise la probabilité de sélectionner le meilleur candidat.

Solution optimale : écouter poliment les n/e premiers candidats puis ensuite sélectionner le premier candidat qui est meilleur que tous les précédents (ou le dernier candidat sinon). Proba de succès $1/e \approx 37\%$ (et c'est optimal)

(note. si on veut sélectionner le 2ème meilleur candidat à la place, la meilleure stratégie ne garantit qu'une proba 1/4 de réussir.)

Algorithmes online de graphes

On peut voir le problème des secrétaires comme un problème online de graphe: on a une étoile pondérée avec les n feuilles qui arrivent une par une, et on veut sélectionner un couplage de poids max avec la meilleure proba possible (en devant décider immédiatement si on prend l'arête qui arrive dans le couplage ou pas).

Algorithmes online de graphes

On peut voir le problème des secrétaires comme un problème online de graphe: on a une étoile pondérée avec les n feuilles qui arrivent une par une, et on veut sélectionner un couplage de poids max avec la meilleure proba possible (en devant décider immédiatement si on prend l'arête qui arrive dans le couplage ou pas).

Objectif plus traditionnel : maximiser le competitive ratio, le ratio entre la qualité de la solution obtenue online et la meilleure solution offline.

Algorithmes online de graphes

On peut voir le problème des secrétaires comme un problème online de graphe: on a une étoile pondérée avec les n feuilles qui arrivent une par une, et on veut sélectionner un couplage de poids max avec la meilleure proba possible (en devant décider immédiatement si on prend l'arête qui arrive dans le couplage ou pas).

Objectif plus traditionnel : maximiser le competitive ratio, le ratio entre la qualité de la solution obtenue online et la meilleure solution offline.

On dit qu'un algo online (pour un problème de maximisation) est c-compétitif si OPT de la solution trouvée par l'algo online est au moins c fois OPT dans le graphe final.

Couplages bipartis online

On a un graphe biparti avec bipartition (U, V), qui a un couplage parfait (pour simplifier). Les sommets de V arrivent un par un (avec leurs arêtes incidentes) et on doit décider immédiatement à quel sommet de U les matcher (ou ne pas les matcher du tout).

Couplages bipartis online

On a un graphe biparti avec bipartition (U, V), qui a un couplage parfait (pour simplifier). Les sommets de V arrivent un par un (avec leurs arêtes incidentes) et on doit décider immédiatement à quel sommet de U les matcher (ou ne pas les matcher du tout).

L'ago glouton donne un couplage maximal, donc de taille au moins $1/2$ fois le couplage maximum (et on peut construire des instances ou le $1/2$ est atteint).

Couplages bipartis online

On a un graphe biparti avec bipartition (U, V), qui a un couplage parfait (pour simplifier). Les sommets de V arrivent un par un (avec leurs arêtes incidentes) et on doit décider immédiatement à quel sommet de U les matcher (ou ne pas les matcher du tout).

L'ago glouton donne un couplage maximal, donc de taille au moins $1/2$ fois le couplage maximum (et on peut construire des instances ou le $1/2$ est atteint).

On peut faire mieux en commençant par ordonner de manière aléatoire uniforme les sommets de U et ensuite matcher chaque sommet de V au meilleur voisin dans U. On obtient un competitive ratio $(1 - 1/e) \approx 63\%$ et c'est optimal (Ranking Algorithm, de Karp, Vazirani, Vazirani 1990).

Couplages bipartis online

On a un graphe biparti avec bipartition (U, V), qui a un couplage parfait (pour simplifier). Les sommets de V arrivent un par un (avec leurs arêtes incidentes) et on doit décider immédiatement à quel sommet de U les matcher (ou ne pas les matcher du tout).

L'ago glouton donne un couplage maximal, donc de taille au moins $1/2$ fois le couplage maximum (et on peut construire des instances ou le $1/2$ est atteint).

On peut faire mieux en commençant par ordonner de manière aléatoire uniforme les sommets de U et ensuite matcher chaque sommet de V au meilleur voisin dans U. On obtient un competitive ratio $(1 - 1/e) \approx 63\%$ et c'est optimal (Ranking Algorithm, de Karp, Vazirani, Vazirani 1990).

On peut aussi regarder le cas où les sommets de V arrivent de manière aléatoire uniforme. Dans ce cas l'ago glouton est équivalent à l'algo ci-dessus et ça donne du $(1 - 1/e)$.

Remarques finales sur les algos online

Ben-David, Borodin, Karp, Tardos, Wigderson ont montré en 1994 que

- l'adversaire qui connaît les bit aléatoires est tellement fort que l'aléatoire est inutile.
- si on a un algo c-compétitif contre un adversaire oblivious, et un algo d-compétitif contre un adversaire adaptive, on a un algo déterministe cd-compétitif.
- On peut avoir des moyens plus ou moins fiables de prédire l'avenir et on veut utiliser ça pour faire des meilleurs choix (dans le sens ou si la prédiction est correcte notre solution est meilleure, et si la prédiction n'est pas correcte on ne fait pas pire) (cf exposé de Bertrand Simon).

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

$F_0 =$ nombre de symboles différents et $F_1 = m$. F_2 ?

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

$F_0 =$ nombre de symboles différents et $F_1 = m$. F_2 ?

Alon, Matias, Szegedy (1999) : On tire au hasard $x_1, \ldots, x_n = \pm 1$. On initialise notre compteur $r := 0$ et à chaque fois qu'un a_i est lu, on définit $r := r + x_i$.

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

$F_0 =$ nombre de symboles différents et $F_1 = m$. F_2 ?

Alon, Matias, Szegedy (1999) : On tire au hasard $x_1, \ldots, x_n = \pm 1$. On initialise notre compteur $r := 0$ et à chaque fois qu'un a_i est lu, on définit $r := r + x_i$.

$E(r^2) = E(\sum_{i=1}^{n} x_i m_i)^2$ (où m_i est le nombre d'occurrence de a_i).

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

$F_0 =$ nombre de symboles différents et $F_1 = m$. F_2 ?

Alon, Matias, Szegedy (1999) : On tire au hasard $x_1, \ldots, x_n = \pm 1$. On initialise notre compteur $r := 0$ et à chaque fois qu'un a_i est lu, on définit $r := r + x_i$.

$E(r^2) = E(\sum_{i=1}^{n} x_i m_i)^2$ (où m_i est le nombre d'occurrence de a_i).

En développant, c'est égal à $E(\sum_{i=1}^{n} x_i^2 m_i^2) + 2E(\sum_{i<j} x_i x_j m_i m_j)$.

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

F_0 = nombre de symboles différents et $F_1 = m$. F_2 ?

Alon, Matias, Szegedy (1999) : On tire au hasard $x_1, \ldots, x_n = \pm 1$. On initialise notre compteur $r := 0$ et à chaque fois qu'un a_i est lu, on définit $r := r + x_i$.

$E(r^2) = E(\sum_{i=1}^{n} x_i m_i)^2$ (où m_i est le nombre d'occurrence de a_i).

En développant, c'est égal à $E(\sum_{i=1}^{n} x_i^2 m_i^2) + 2E(\sum_{i<j} x_i x_j m_i m_j)$.

Donc $E(r^2) = F_2$ (et en calculant la variance on vérifie que r^2 est concentré autour de F_2.

Data streams

Data stream. une suite $S = v_1, \ldots, v_m$ de lettres d'un alphabet $A = \{a_1, \ldots, a_n\}$.

On veut faire des statistiques, par exemple calculer le k-ième moment

$$F_k = \sum_{i=1}^{n} (\text{nombre d'occurrence de } a_i \text{ dans } S)^k.$$

F_0 = nombre de symboles différents et $F_1 = m$. F_2 ?

Alon, Matias, Szegedy (1999) : On tire au hasard $x_1, \ldots, x_n = \pm 1$. On initialise notre compteur $r := 0$ et à chaque fois qu'un a_i est lu, on définit $r := r + x_i$.

$E(r^2) = E(\sum_{i=1}^{n} x_i m_i)^2$ (où m_i est le nombre d'occurrence de a_i).

En développant, c'est égal à $E(\sum_{i=1}^{n} x_i^2 m_i^2) + 2E(\sum_{i<j} x_i x_j m_i m_j)$.

Donc $E(r^2) = F_2$ (et en calculant la variance on vérifie que r^2 est concentré autour de F_2.
De plus le calcul de la variance demande juste que les x_i soient indépendants 4 à 4, et on peut stocker une telle famille avec $O(\log n)$ bits au lieu de n.

Algorithmes de streaming dans les graphes

Pour les graphes: c'est comme dans le modèle dynamique (les arêtes sont ajoutées et supprimées au fûr et à mesure, et on veut maintenir une solution (par ex. le graphe est-il connexe?).

Algorithmes de streaming dans les graphes

Pour les graphes: c'est comme dans le modèle dynamique (les arêtes sont ajoutées et supprimées au fûr et à mesure, et on veut maintenir une solution (par ex. le graphe est-il connexe?).

La difficulté est que l'espace est limité, typiquement $O(n)$ pour un graphe à n sommet (facile pour les graphes sparse, mais dur pour les graphes denses où il faut trouver un moyen d'approximer le graphe avec un objet sparse).
"semi-streaming algorithm".

Algorithmes de streaming dans les graphes

Pour les graphes: c'est comme dans le modèle dynamique (les arêtes sont ajoutées et supprimées au fûr et à mesure, et on veut maintenir une solution (par ex. le graphe est-il connexe?).

La difficulté est que l'espace est limité, typiquement $O(n)$ pour un graphe à n sommet (facile pour les graphes sparse, mais dur pour les graphes denses où il faut trouver un moyen d'approximer le graphe avec un objet sparse).
"semi-streaming algorithm".

Différence majeure avec les deux autres modèles: dans le modèle streaming on s'autorise parfois plusieurs passes.

Résumé

- Graphes dynamiques : on cherche à minimiser le temps de mise à jour de la solution à chaque mise à jour du graphe (et de manière secondaire, le temps de requêtes à la solution si elle est stockée de manière implicite).

Résumé

- Graphes dynamiques : on cherche à minimiser le temps de mise à jour de la solution à chaque mise à jour du graphe (et de manière secondaire, le temps de requêtes à la solution si elle est stockée de manière implicite).
- Online : On prend des décisions irrévocables (ou alors ça coûte cher de faire des modifications), et on cherche à optimiser la qualité de la solution obtenue par rapport à la meilleure solution si on avait eu toutes les infos en main

Résumé

- Graphes dynamiques : on cherche à minimiser le temps de mise à jour de la solution à chaque mise à jour du graphe (et de manière secondaire, le temps de requêtes à la solution si elle est stockée de manière implicite).
- Online : On prend des décisions irrévocables (ou alors ça coûte cher de faire des modifications), et on cherche à optimiser la qualité de la solution obtenue par rapport à la meilleure solution si on avait eu toutes les infos en main
- Streaming : on cherche à minimiser l'espace (et de manière secondaire, le temps de calcul). Il peut y avoir plusieurs passes.