Projet de complexité et algorithmique appliqué

Résolution du problème du voyageur de commerce

Tom Solacroup et Jérémy Turon

I) Les algorithmes implémentés

1) Brute force
2) Backtracking
3) Utilisation d'arbre couvrant
4) Réduction vers SAT

II) Nos résultats

1) Résultats sur des graphes complets pondérés
2) Résultats sur des graphes à poids d'arêtes lourds ou léger

I) Les algorithmes utilisés

1) Brute force

Pseudo-code :
Entrée : un graphe G=(V,E).
début
    retourner C une permutation de V tel que poids(C) = min(poids(A) | A est une permutation de V) ;
fin

Il s'agit juste de faire un parcours exhaustif de tout les cycles possibles (permutations de V) de calculer leur poids et de retourner celle qui a le plus petit poids possible. Ayant n!, |V| = n, permutations possibles et le temps de calcul du poids d'un cycle de taille n étant n nous avons une complexité de l'ordre de O(n x n!).

2) Backtracking

Pseudo-code :

```
Entrée : un graphe G = (V,E),
          A un cycle,
          l un entier positif,
          lengthSoFar un entier positif,
          Sol un cycle.
début
        n = taille(A);
        minCost = poids(Sol);
        si l == n alors
                newCost = lengthSoFar +dist(A[n-1],[A[0]) ;
                si newCost < minCost alors
                        Sol = A ;
                fin si
        sinon
                 pour i allant de l à n faire
                         swap(A, l, i) ;
                         newLength = lengthSoFar +dist(A[l-1],[A[l]) ;
                        si newLength ≤ minCost  alors
                                newSol = backTracking(graph, A, l+1, newLength, Sol) ;
                                 si poids(newSol) < minCost alors
                                        Sol = newSol ;
                                 fin si ;
                         fin si
                         swap(&A, l, i) ;
                 fin pour
        fin si
        retourner Sol ;
fin
```

Cette algorithme est très similaire à celui de la force brute, le principe est ici aussi d'essayer tout les permutations possibles la différence est qu'ici on essaye de détecter lesquels ne vont conduire à aucun cycle plus intéressant. Pour cela on ordonne les permutations de la façon suivant, soit V = { 1, 2, …, n}, on teste toutes les permutations commençant par 1, puis par 2, …, et enfin par n, et pour chacun d'eux on teste toutes les permutations avec un sommet j en position i et avant de passer à j + 1 on vérifie que le chemin du premier sommet au sommet est plus léger que le meilleure cycle actuellement trouvé, sinon il ne sert à rien de tester les cycles découlant de ce chemin.
Dans le pire des cas nous ne supprimerons aucune possibilités et nous retrouverons donc avec les mêmes performances que pour le brute force, c'est-à-dire une compléxité en O(n x n!).

Lors de la première exécution, A est une permutation quelconque de V, l = 0, lengthSoFar = 0, et Sol = A.

3) Utilisation d'arbre couvrant

Pseudo-code :
Entrée : un graph G = (V,E).
début
	C= {} ;
	on trouve un arbre couvrant T de G en appliquant l'algorithme de Kruskal ;
	on effectue un parcours en profondeur de T et à chaque fois qu'on passe par un sommet on le rajoute dans C ;
	retourner C ;
fin

La complexité de la recherche de l'arbre couvrant est en O(m log(m)) avec m = |E|, et celle de la création du cycle est en O(|T|) soit o(n), on a donc un algorithme avec une complexité en O(n + m x log(m)).
Il s'agit d'une 2-approximation de la solution.
Preuve :
Soit S la meilleure solution, T l'arbre couvrant de poids minimal, et C la solution trouvé avec notre algorithme pour un graphe G.
poids(T) < poids(S), on toujours créer un arbre couvrant de G plus léger que S en prenant S et en retirant l'arête entre deux de ces sommets consécutifs, on obtient ainsi un chemin hamiltonien et donc un arbre couvrant.
Et dans notre cas, le cycle que nous créons passe deux fois par chaque arête de T, on a donc poids(C) = 2 x poids(T) < 2 x poids(S), nous avons donc bien une 2-approximation.

4) Réduction vers SAT

Nous n'avons pas réussi à mettre en place une réduction vers SAT, celle que nous avons essayé, et qui est décrite ci-dessous, est déclaré tout le temps insatisfiable par glucose, nous n'avons pas réussi à trouver notre erreur.

Il s'agit de réduire le problème du voyageur de commerce avec k arêtes de poids 10 vers le problème SAT pour ensuite utiliser un SAT solveur (dans notre cas glucose).

Pseudo-code :
Entrée : un graphe G = (V,E).
début
    pour k allant de 0 à |V| - 1 faire
        réduire le problème du voyageur de commerce avec k arêtes vers SAT;
        résoudre la formule SAT générer avec un SAT solveur;
        si le SAT solveur trouve une solution
            retourner k ;
        fin si
    fin pour
    retourner n ;
fin

$\vee\vee \quad \wedge\wedge$

La réduction est la suivante :
Soit P = {1,2,...,n}.
Soit K = {1,2,...,k}.
Les variables sont x_{ie} tels que x_{ie} est vrai si l'arête e est en ième position dans le cycle. Et x_{ij} si l'arête en position j est de poids 10, i étant juste un compteur.

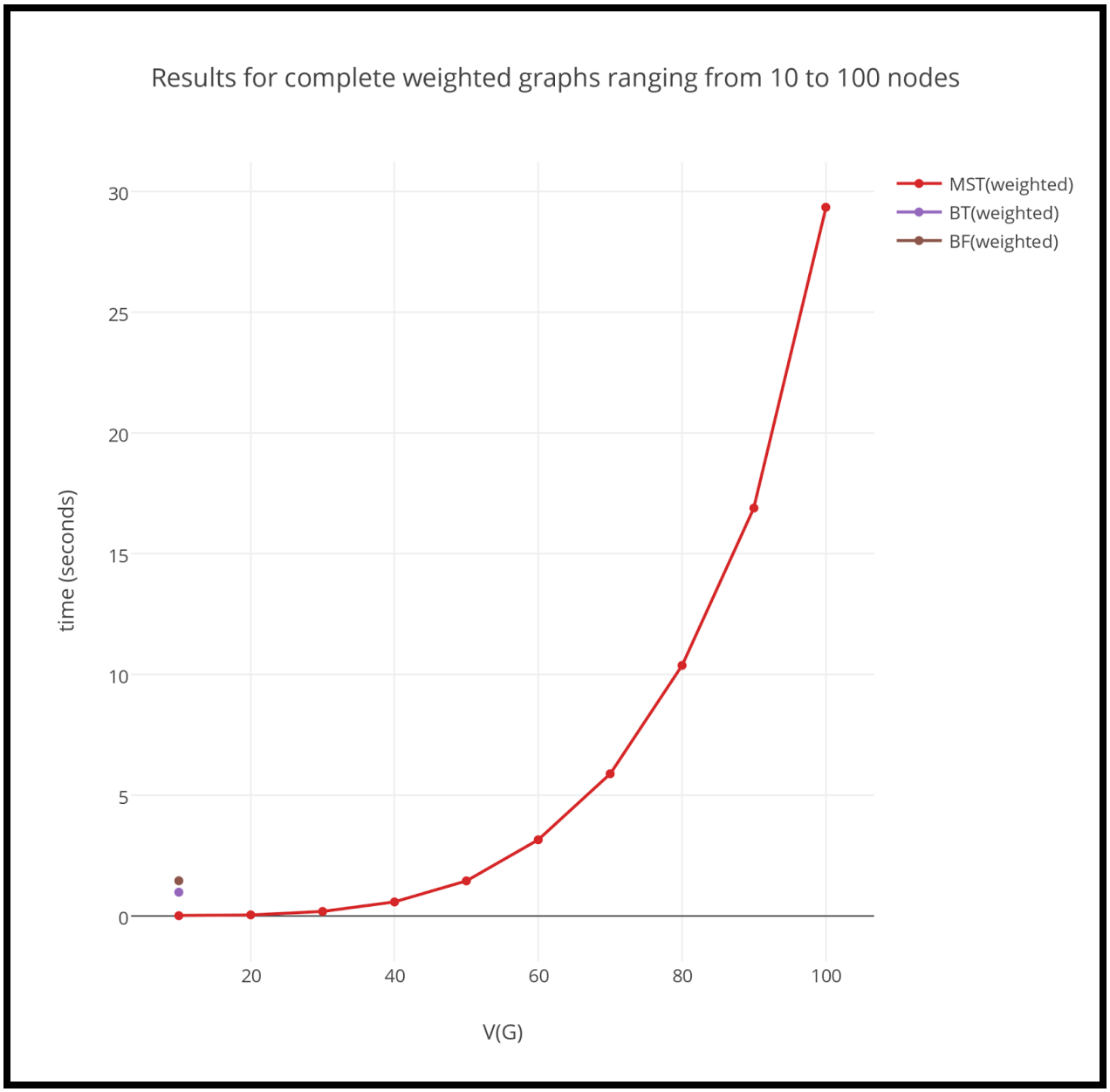
1) $\wedge_{i \in P} \ (\vee_{e \in E} x_{ie})$, on veut que chaque position est au moins une arête associée, on a n clauses de cette forme.
2) $\wedge_{i \in P} \wedge_{e \in E,\, e' \in E,\, e' \neq e} (\neg x_{ie} \vee \neg x_{ie'})$, on veut au maximum une arête par position, on a n * m * (m-1) / 2 clauses de cette forme.
3) $\wedge_{e \in E} \wedge_{i \in P,\, i' \in P,\, i' \neq i} (\neg x_{ie} \vee \neg x_{i'e})$, on veut au maximum une position par arête, on a m * n * (n-1) clauses de cette forme.
4) $\wedge_{i \in P} \wedge_{(u,v) \in E} (\neg x_{i(u,v)} \vee (\vee_{v' \in V} x_{((i+1) \bmod n)(v,v')})) \wedge (\neg x_{i(u,v)} \vee (\vee_{u' \in V} x_{((i-1) \bmod n)(u,u')}))$, on veut que les arêtes choisies forment un cycle, on a n * m * 2 clauses de cette forme.
5) $\wedge_{i \in P} \wedge_{j \in K} (\neg x_{ji} \vee (\vee_{e \in E,\, poids(e) = 10} x_{ie}))$, soit il y a un j tel que x_{ji} est vrai soit la position i n'a pas d'arête de poids 10, on a n * k clauses de cette forme.
6) $\wedge_{j \in K} (\vee_{i \in P} x_{ji})$, on a au une position par j dans K, on a k clauses de cette forme.
7) $\wedge_{i \in P} \wedge_{j \in K,\, j' \in K,\, j' \neq j} (\neg x_{ji} \vee \neg x_{j'i})$, chaque j possède sa propre arête, on a n * k * (k-1) / 2 clauses d cette forme.
8) $\wedge_{j \in K} \wedge_{i \in P,\, i' \in P,\, i' \neq i} (\neg x_{ji} \vee \neg x_{ji'})$, un j a au plus une position associé, on a k * n * (n-1) / 2 clauses de cette forme.

Soit φ la formule SAT généré. La complexité est en O(|φ| * n) + O(SAT-Solveur(φ)), avec |φ| étant le nombre de variables plus le nombre de clauses.

II) Résultats

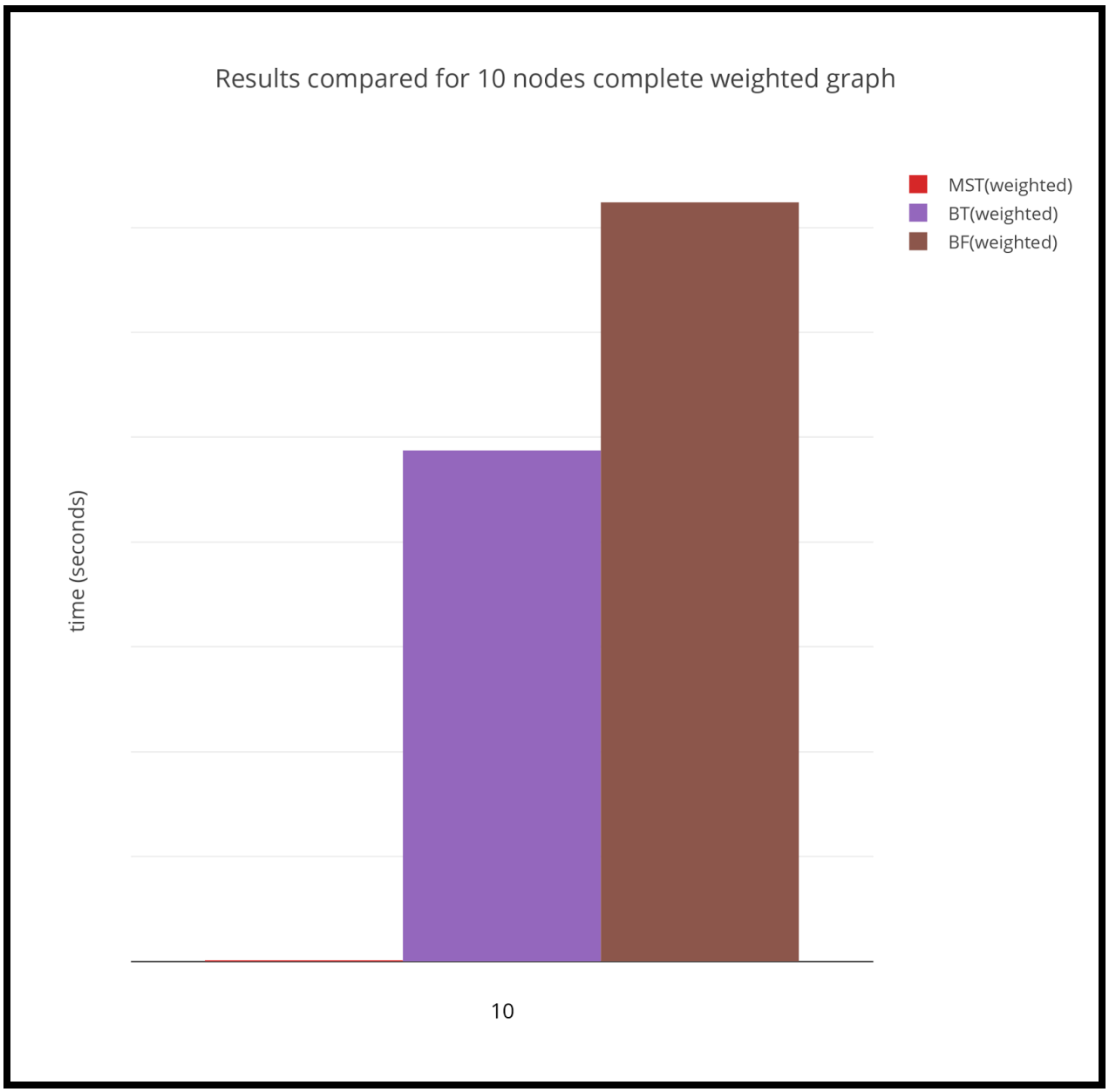
Note : les résultats décrits dans cette partie ont été produits en exécutant chacun des algorithmes implémentés de notre programme sur la machine du CREMI nommée “infini1” pendant plusieurs heures. Nos jeux de tests sont des graphes générés à la volée par le programme gengraph de Cyril Gavoille. Nous générons 10 graphes de 10 sommets, nous récupérons le temps d’éxécution nécessaire au traitement d’un graphe, puis nous réitérons avec 10 sommets de plus (jusqu’à atteindre des graphes de 100 sommets. De plus, notre programme invoque la version parallèle de Glucose 4 de Gille Audemard et Laurent Simon.

2) Résultats sur des graphes complets pondérés



Pour les graphes complets pondérés, l'utilisation d'un arbre couvrant (MST) a été la plus efficace. En effet, alors que nous n'avons pas pu avoir de résultats avec le bruteforce et le backtracking pour des graphes ayant 20 sommets ou plus, l'utilisation d'un arbre couvrant a permis de traiter en 29 secondes un graphe de 100 sommets.

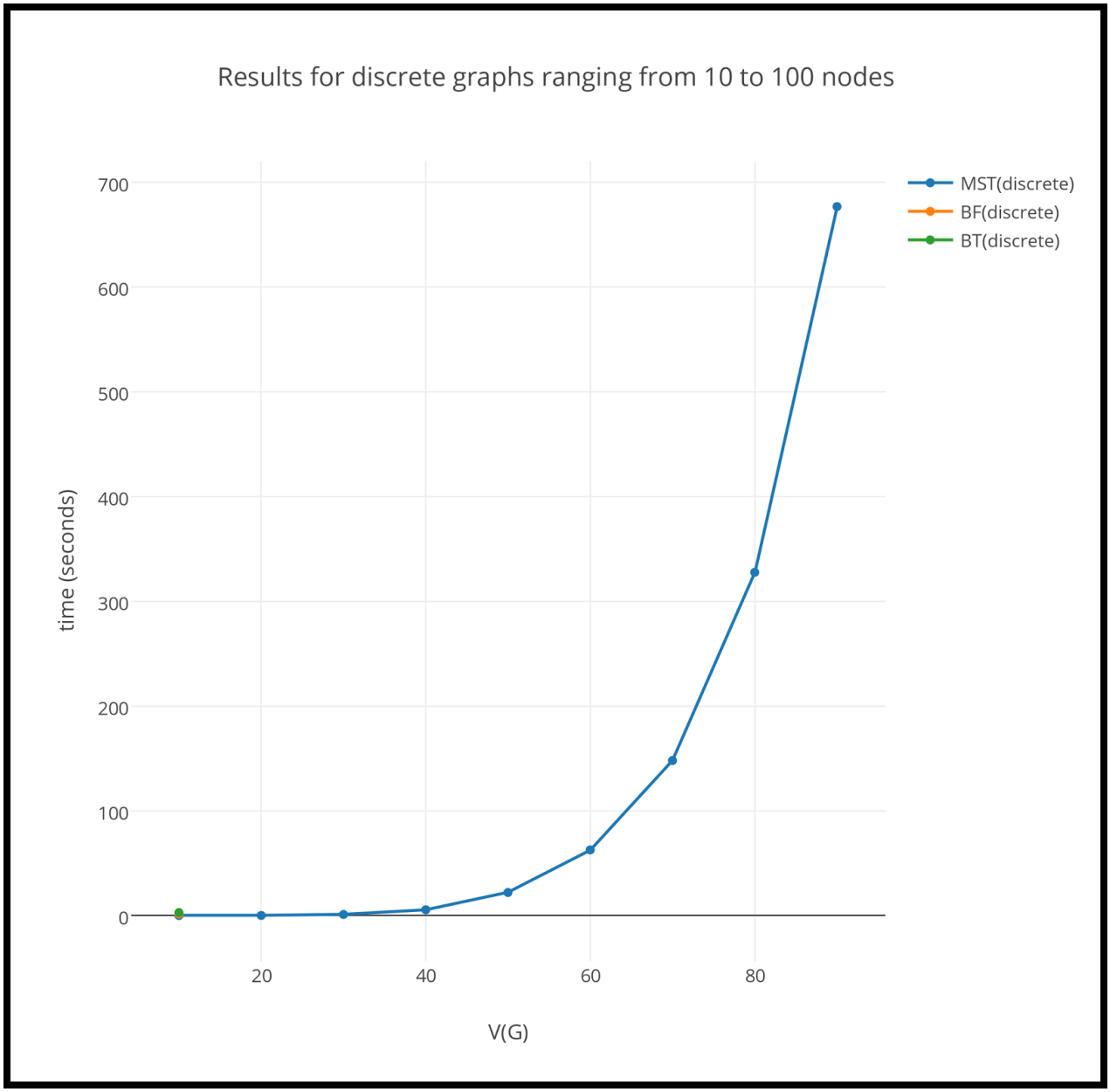
Voyons en détail les temps de traitement pour des graphes de dix sommets :



On peut observer que le temps de traitement d'un graphe complet pondéré de dix sommets avec un arbre couvrant est négligeable. L'utilisation de backtracking en revanche

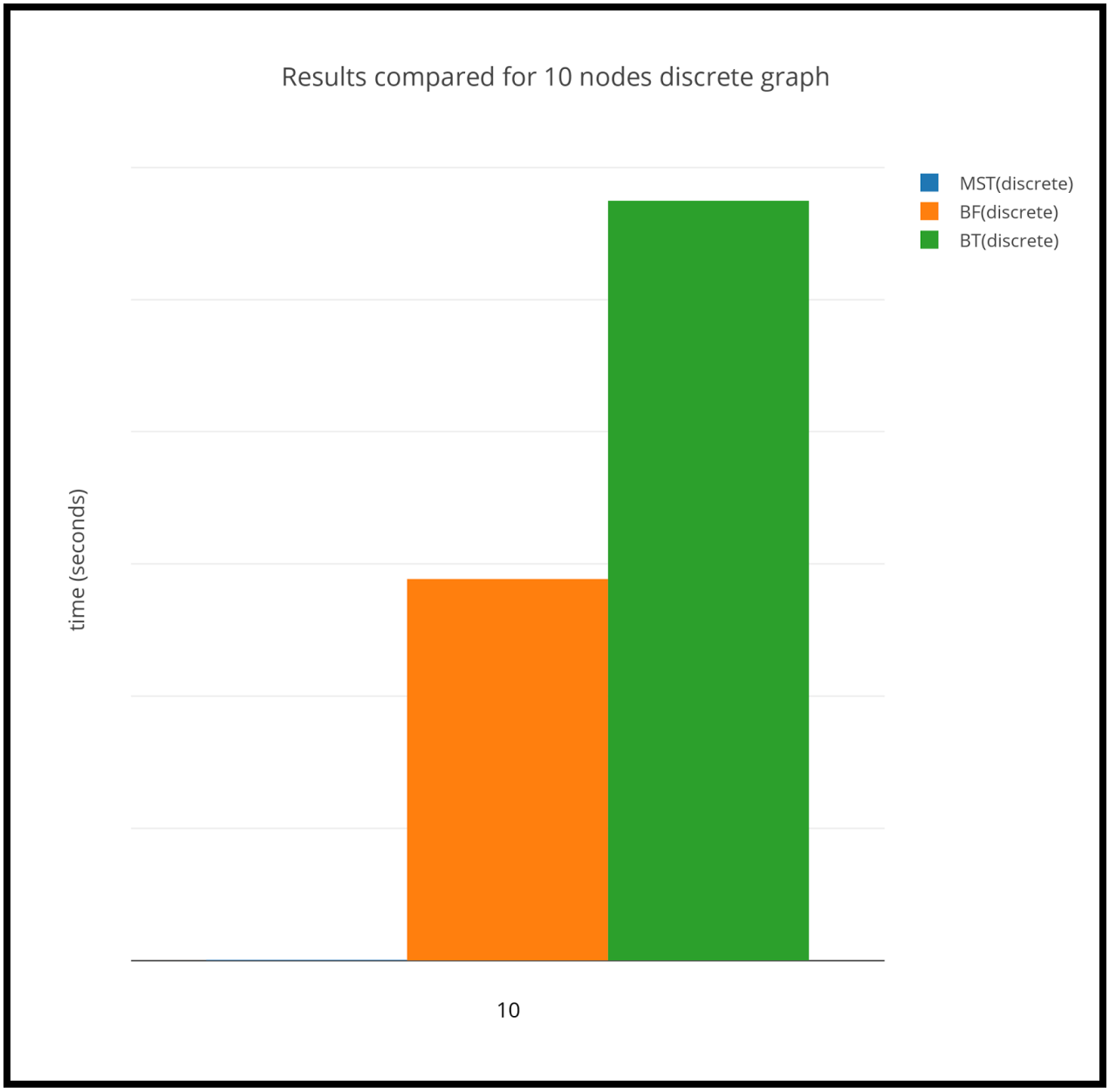
repousse le temps d'exécution à 0.9 seconde, tandis que le temps d'exécution lorsqu'on utilise le bruteforce est de 1.4 secondes.

2) Résultats sur des graphes à poids d'arêtes lourds ou léger



Sur des graphes à poids d'arêtes lourds ou léger, l'arbre couvrant est encore une fois l'algorithme le plus efficace. Nous avons pu obtenir des résultats pour des graphes ayant jusqu'à 90 sommets, avec un temps d'exécution de 676 secondes. Les algorithmes bruteforce et backtracking n'ont à nouveau pas permis d'obtenir des résultats dans des temps raisonnables pour des graphes de plus de 10 sommets.

Voyons en détail les temps de traitement pour des graphes de dix sommets :



L’arbre couvrant est encore une fois l’algorithme le plus efficace avec un temps d'exécution négligeable. En revanche, contrairement à ce que nous avons pu voir pour les graphes pondérés, le bruteforce semble ici plus efficace que le backtracking, avec un temps d'exécution de 1.4 secondes contre 2.8 secondes.