Recommandation opportuniste de trajectoires pour l'accomplissement de tâches dans les systèmes crowdsourcing

André Sales Fonteles, Sylvain Bouveret, Jérôme Gensel

Dans Document numérique 2016/1 Vol. 19 , pages 103 à 126
Éditions JLE

ISSN 1279-5127
ISBN 9782746247697
Date de mise en ligne : 07/07/2016

Recommandation opportuniste de trajectoires pour l'accomplissement de tâches dans les systèmes crowdsourcing

André Sales Fonteles, Sylvain Bouveret, Jérôme Gensel

Univ. Grenoble Alpes, CNRS, LIG, F-38000 Grenoble, France
{andre.sales-fonteles,sylvain.bouveret,jerome.gensel}@imag.fr

RÉSUMÉ. *Les systèmes de marché crowdsourcing (CMS) sont des plateformes qui permettent à une personne de publier des tâches afin qu'elles soient accomplies par d'autres. Récemment, un type de CMS est apparu dans lequel des tâches spatio-temporelles doivent être accomplies dans une fenêtre de temps et un lieu précis. Dans cet article, nous présentons le problème de recommandation de trajectoires utiles (*PRTU*), qui permet à une personne en situation de mobilité d'accomplir des tâches spatio-temporelles pour lesquelles elle montre une grande affinité et/ou aptitude, sans compromettre son arrivée à destination dans les temps. Nous démontrons que le* PRTU *est NP-complet (dans sa version décisionnelle) et proposons, pour y répondre, un algorithme exact accompagné de cinq heuristiques d'approximation. En outre, nous proposons une architecture de référence pour mettre en oeuvre la recommandation de ces trajectoires dans un CMS réel. Enfin, nos expérimentations montrent que l'algorithme exact proposé peut être une solution acceptable pour certaines instances de* PRTU *et que les heuristiques constituent des alternatives plus performantes qui permettent d'attendre jusqu'à 77 % de rendement optimal.*

ABSTRACT. *Crowdsourcing market systems (CMS) are platforms that allow one to publish tasks in order to be accomplished by others. Recently, a type of CMS has appeared where spatio-temporal tasks are to be accomplished by persons at a specific time-window and location. We present the Useful Trajectories Recommendation Problem (*PRTU*), that allows a person to accomplish tasks he has affinity and/or ability to, without compromising his arrival in time at the destination. We prove that* PRTU *is NP-complet (in its decision version) and propose an exact algorithm and five heuristics for it. Further, we propose a reference architecture for the deployment of the recommendation of such trajectories in a CMS. Our experiments have shown that our algorithm can be feasible solution for some instances of* PRTU *and that, otherwise, the heuristics are alternatives that run faster and can provide up to 77% of the optimal utility.*

MOTS-CLÉS : *ordonnancement de tâches spatio-temporelles, recommandation de tâches.*

KEYWORDS: *spatial task assignment, task recommendation, spatial crowdsourcing.*

DOI:10.3166/DN.19.1.103-126

1. Introduction

Au cours des dernières années, de nombreux systèmes de *crowdsourcing* (CMS, pour *Crowdsourcing Market Systems*) sont apparus, dans lequel un groupe d'utilisateurs, appelés exécutants, contribuent pour atteindre des objectifs ou résoudre des tâches. Ces systèmes s'appuient sur l'intelligence humaine pour accomplir des tâches que les ordinateurs ne sont pas capables de réaliser seuls, ou pour lesquelles ils ne sont pas toujours les plus performants (Kittur *et al*., 2011), telles que : la traduction de texte, la transcription de l'audio vers le texte, l'étiquetage sémantique, *etc*. Amazon Mechanical Turk est probablement l'exemple le plus connu de ce type de système, auquel se rattachent également des noms comme CrowdFlower [1] et oDesk [2].

Plus récemment, un autre type de CMS est apparu, dans lesquels les commanditaires peuvent publier des tâches spatio-temporelles qui requièrent d'être accomplies dans un lieu et à un moment spécifique. Par exemple, une tâche spatio-temporelle peut consister à demander à quelqu'un de se rendre à l'intersection de deux rues à un moment donné de la journée pour prendre une courte vidéo qui permettra à quelqu'un d'autre, ou même à un système, d'analyser le trafic en ce lieu, à cette heure. Sereale [3], TaskRabbit [4] et Medusa (Ra *et al*., 2012) sont des exemples de systèmes qui acceptent des tâches spatio-temporelles.

Dans ce travail, nous nous concentrons sur les CMS qui comportent des tâches spatio-temporelles. Plus spécifiquement, nous nous concentrons sur le scénario suivant : une personne, prête à contribuer au CMS, souhaite se déplacer d'un lieu à l'autre et le CMS, de façon opportuniste, recommande une trajectoire pour l'accomplissement de tâches spatio-temporelles sans compromettre son arrivée dans les temps à destination. En plus de respecter l'heure limite d'arrivée, d'autres aspects doivent être considérés. Par exemple, chaque tâche a une fenêtre de temps lors de laquelle elle peut être réalisée. Ainsi, une trajectoire doit garantir que pour chaque tâche proposée sur le parcours en un lieu donné, l'heure d'arrivée en ce lieu est bien située dans la fenêtre temporelle associée à cette tâche. De plus, au lieu d'essayer de recommander le chemin le plus court à l'exécutant, le but du système est de recommander une trajectoire avec des tâches pour lesquelles l'exécutant montre la plus grande affinité ou aptitude [5], et, par conséquent, qui ont une grande probabilité d'être acceptées. Nous appelons le problème de recherche d'une telle trajectoire, un problème de recommandation de trajectoires utiles, ou PRTU.

La suite de cet article est organisée comme suit. La section 2 est consacrée aux travaux voisins. Dans la section 3, nous décrivons formellement le problème de re-

1. http://www.crowdflower.com
2. http://www.odesk.com
3. https://www.sereale.fr
4. https://www.taskrabbit.com
5. L'affinité (ou l'intérêt) et l'aptitude d'un exécutant pour une tâche s sont représentées par une fonction d'utilité $u(s)$

commandation de trajectoires utiles et étudions sa complexité. Dans la section 4, nous présentons une méthode naïve pour résoudre le PRTU, avant de proposer un algorithme exact plus performant. Puis, nous proposons des heuristiques pour trouver des solutions approximatives aux instances du PRTU que l'approche exacte n'est pas en mesure de traiter. Dans la section 5, nous proposons une architecture de référence pour l'utilisation de la recommandation de trajectoires utiles dans un CMS réel. La section 6 présente les résultats de nos expérimentations. La section 7 conclut l'article et esquisse nos travaux futurs.

2. Travaux voisins

Dans cette section, nous présentons tout d'abord un aperçu de la littérature sur la recommandation de tâches, puis nous discutons des travaux voisins traitant de l'ordonnancement de tâches dans les CMS.

Afin d'augmenter le nombre total de tâches accomplies dans un CMS, certains chercheurs ont proposé l'utilisation de systèmes de recommandation pour suggérer à une personne des tâches ayant une utilité élevée. Ces approches ne se préoccupent toutefois pas de l'ordonnancement de ces tâches. Bien qu'au premier coup d'œil il puisse sembler simple de recommander des tâches selon leur utilité, le véritable défi de cette problématique se trouve dans l'estimation de la valeur de chaque utilité. En général, la littérature (Lin *et al*., 2014 ; Ambati *et al*., 2011 ; Yuen *et al*., 2012 ; Fonteles *et al*., 2014) propose d'estimer ces valeurs en s'appuyant sur les intérêts/préférences et compétences de l'utilisateur qui sont implicitement découverts par l'analyse de ses interactions avec le CMS. Certains travaux prônent une autre approche. Par exemple, bien que Difalla *et al.* (2013) utilisent une approche similaire d'estimation selon les préférences et compétences d'un exécutant, ces informations sont découvertes à partir de son profil dans des réseaux sociaux, et non par l'analyse de son utilisation du CMS. La différence principale entre notre problème et celui traité dans ce type de recommandation de tâches est que l'ordre dans lequel les tâches doivent être accomplies n'est pas pris en compte : lorsqu'un CMS ne traite pas de tâches spatio-temporelles, cette séquence n'est pas pertinente. Cependant, lorsque les tâches proposées présentent des contraintes de temps et localisation, l'ordonnancement des tâches joue un rôle essentiel. Dans ce travail, nous utilisons le concept d'utilité de tâche pour estimer l'utilité d'une trajectoire. Nous faisons ici l'hypothèse que l'utilité associée à chaque tâche a déjà été estimée par ailleurs (par exemple en utilisant des méthodes similaires à celles proposées dans les références mentionnées ci-dessus), et que cette fonction d'utilité est donc une donnée d'entrée du problème.

Un autre domaine lié à notre travail est celui de l'ordonnancement de tâches spatio-temporelles dans les systèmes de *crowdsourcing*. Kazemi et Shahabi (2012) proposent une approche dans laquelle le système, à partir d'un ensemble d'utilisateurs ou exécutants potentiels localisés, et d'un ensemble de tâches spatio-temporelles, tente d'attribuer une séquence de tâches aux utilisateurs qui maximise le nombre global de tâches accomplies. Un autre travail de Kazemi *et al.* (2013) a comme but la maximisation

du nombre de tâches spatio-temporelles effectuées en attribuant le plus possible de tâches. Cette approche s'appuie sur la localisation des tâches et des exécutants, ainsi que sur la notion de réputation des exécutants et de réputation requise pour l'accomplissement d'une tâche. Enfin, Deng *et al.* (2013) proposent une façon de recommander un ordonnancement à un exécutant, étant donnés sa localisation et un ensemble de tâches spatio-temporelles, qui maximise le nombre de tâches à accomplir. L'originalité de notre travail réside dans les caractéristiques suivantes. Premièrement, aucun de ces travaux ne prend en compte, dans la recommandation des tâches, les préférences/aptitudes d'un exécutant, représentées par la notion d'utilité d'une tâche pour un exécutant. Au contraire, ces approches tentent de satisfaire seulement le système (CMS) et les contraintes de ses tâches. De plus, les tâches spatio-temporelles traitées dans ces travaux ne sont pas associées à des périodes de temps durant lesquelles elles doivent être réalisées (le plus souvent elles ne possèdent qu'une échéance donnée). Enfin, aucun de ces travaux ne se concentre sur un scénario particulier mais fréquent dans lequel l'utilisateur se déplace d'un point origine à un point destination qu'il doit atteindre avant une échéance fixée.

3. Problème de recommandation de trajectoires utiles

Une tâche s correspond à la demande (ou requête) d'un service qui doit être accompli en un lieu et une fenêtre de temps. Nous caractérisons une tâche par un quadruplet $s = (l^s, \tau_1^s, \tau_2^s, \delta^s)$, où l^s est le lieu où la tâche doit être réalisée (pris dans un ensemble prédéfini L de lieux), $\tau_1^s \in \mathbb{N}$ et $\tau_2^s \in \mathbb{N}$ sont respectivement la date de début au plus tôt et la date de début au plus tard de la tâche (en d'autres termes, sa fenêtre de temps), et $\delta^s \in \mathbb{N}$ est la durée de la tâche [6]. Nous définissons ci-dessous une instance du PRTU.

DÉFINITION 1. — *Une* instance *du problème de recommandation de trajectoires utiles* PRTU *est un tuple* $(\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G})$*, où :*

– $\mathcal{S} = \{s_1, \ldots, s_n\}$ *est un ensemble de* tâches *;*

– $u : \mathcal{S} \to \mathbb{N}$ *est une* fonction d'utilité *qui associe à chaque tâche* s_i *une utilité,* $u(s_i)$*, qui traduit l'intérêt de l'exécutant pour la tâche ;*

– l_o *est le lieu initial (origine) de l'exécutant ;*

– l_d *est le lieu final (destination) que l'exécutant souhaite atteindre ;*

– $\tau_o \in \mathbb{N}$ *est le temps au plus tôt auquel l'exécutant peut quitter le lieu* l_o *;*

– $\tau_d \in \mathbb{N}$ *est le temps au plus tard (échéance) auquel l'exécutant doit atteindre son lieu de destination* l_d *;*

– $\mathcal{G} = (V, E, d)$ *est un graphe orienté pondéré, où* $V = (\{l^s | s \in \mathcal{S}\} \cup \{l_o, l_d\})$ *est l'ensemble des sommets (ou bien de lieux),* $E \subset V^2$ *est l'ensemble des arcs avec*

6. Le fait de considérer que les dates sont définies sur les entiers se fait sans perte de généralité et permet d'analyser la complexité du problème de manière rigoureuse. Dans la suite de l'article, pour simplifier le discours, les dates pourront être notées sous une forme plus usuelle (hh:mm par exemple).

$v \neq v'$ pour chaque $(v, v') \in E$ et d est une fonction de coût qui associe à chaque $(v, v') \in E$ un nombre dans $\mathbb{N}$ qui spécifie le temps nécessaire pour voyager de v à v'.

Dans ce modèle, l'aspect spatial est pris en compte de manière indirecte sous la forme du graphe sous-jacent au problème. Cela permet de s'abstraire du calcul des trajectoires exactes dans l'espace (qui peut être réalisé dans une étape préliminaire au PRTU, aboutissant au graphe des coûts), tout en gardant un formalisme suffisamment général, expressif et abstrait pour englober toute une famille de problèmes pour lesquels on peut associer un coût de passage d'une tâche à l'autre à chaque paire de tâches.

Soit $\mathcal{I} = (\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G})$ une instance de PRTU. Une trajectoire pour $\mathcal{I}$ est une séquence $T = \langle (s_1^T, t_1), \ldots, (s_m^T, t_m) \rangle$ de tâches distinctes devant être accomplies, chacune étant associée à un temps d'arrivée sur son lieu d'exécution. Le temps d'arrivée est défini comme suit :

$$t_i = \begin{cases} \max(\tau_1^{s_1^T}, \tau_o + d(l_o, l^{s_1^T})) & \text{si } i = 1 \\ \max(\tau_1^{s_i^T}, t_{i-1} + \delta^{s_{i-1}^T} + d(l^{s_{i-1}^T}, l^{s_i^T})) & \text{si } i > 1, \end{cases}$$

où $\max$ représente le fait que l'exécutant doive attendre l'instant au plus tôt $\tau_1^{s_i^T}$ afin d'accomplir la tâche s_i^T.

Une trajectoire T est *valide* si et seulement si elle satisfait toutes les conditions suivantes :

1. pour tout $(s_i^T, t_i) \in T$, $t_i \in [\tau_1^{s_i^T}, \tau_2^{s_i^T}]$ (une tâche doit être accomplie dans sa fenêtre de temps), et

2. $t_m + \delta^{s_m^T} + d(l^{s_m^T}, l_d) \leq \tau_d$ (l'exécutant doit arriver à sa destination avant l'échéance).

Pour souci de simplicité d'écriture, dans la suite de cet article, nous omettrons en général les temps d'arrivée dans la représentation des trajectoires. Ainsi, par exemple, la trajectoire $\langle (A, t_A), (C, t_C), (B, t_B) \rangle$ sera simplement représentée par la séquence $\langle A \to C \to B \rangle$.

Étant donnée une trajectoire T, l'utilité de T correspond alors simplement à la somme de toutes les utilités associées aux tâches présentes dans la trajectoire, à savoir :

$$u(T) = \sum_{(s_i^T, t_i) \in T} u(s_i^T)$$

Nous pouvons maintenant rassembler tous les concepts précédents et définir le Problème de Recommandation de Trajectoires Utiles de la manière suivante :

PROBLÈME DE RECOMMANDATION DE TRAJECTOIRES UTILES [PRTU]	
Entrée :	Une instance du PRTU $(\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G})$.
Sortie :	Une trajectoire T valide d'utilité maximale parmi l'ensemble des trajectoires valides.

3.1. Complexité du problème

La complexité du problème PRTU dépend directement de celle des fonctions u et d. Sous l'hypothèse raisonnable que ces fonctions sont calculables en temps déterministe polynomial (hypothèse que nous ferons implicitement dans toute la suite de cet article), nous pouvons montrer que la version décisionnelle du PRTU est NP-complète, c'est-à-dire que sous l'hypothèse $\mathsf{P} \neq \mathsf{NP}$, la résolution du PRTU est théoriquement difficile.

PROPOSITION 2. — *Étant donnés une instance du* PRTU $(\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G})$ *et un entier k, décider s'il existe une trajectoire valide T, telle que $u(T) \geq k$, est un problème NP-complet.*

PREUVE 3. — L'appartenance à NP est immédiate. On peut voir simplement qu'étant donné que d et u sont calculables en temps polynomial, on peut vérifier en temps polynomial également qu'une trajectoire T est valide et a une utilité supérieure à un entier k.

La NP-difficulté du problème peut se montrer par réduction depuis le problème du voyageur de commerce (TSP pour *Travelling Salesperson Problem*), dont la version décisionnelle peut s'énoncer comme suit :

TSP	
Entrée :	Un ensemble $\mathcal{C} = c_1, \ldots, c_m$, une fonction de distance $d : C \times C \to \mathbb{N}$, et un nombre entier k.
Question :	Existe-t-il une permutation σ de $[\![1, m]\!]$ telle que $d(c_{\sigma(m)}, c_{\sigma(1)}) + \sum_{i=1}^{m-1} d(c_{\sigma(i)}, c_{\sigma(i+1)}) \leq k$?

À partir d'une instance $((\mathcal{C}, d), k)$ de TSP, nous pouvons créer une instance $((\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G}), k')$ de la version décisionnelle du PRTU comme suit :

– $\mathcal{G} = (V, E, d')$ est un graphe complet entre m sommets $(v_1, \ldots, v_m)$, et $d'(v_i, v_j) = d(c_i, c_j)$;

– $\mathcal{S} = \{s_1, ..., s_m\}$, où $l^{s_i} = v_i$, $\tau_1^{s_i} = 0$, $\tau_2^{s_i} = k$ et $\delta^{s_i} = 0$;

– $u(s_i) = 1$ pour tout i ;

– l_o peut être n'importe quel sommet du graphe ;

– $l_d = l_o$;

– $\tau_o = 0$;

– $\tau_d = k$;
– $k' = m$.

Supposons qu'il existe une trajectoire valide T pour cette instance de PRTU ayant une l'utilité supérieure ou égale à $k' = m$. C'est-à-dire que toutes les tâches dans T sont présentes dans la trajectoire. Étant données la définition d'une trajectoire valide et de son utilité, on peut affirmer que (i) la trajectoire passe sur tous les sommets du graphe, et (ii), la durée totale de la trajectoire est inférieure à k (l'échéance). Cela correspond à la solution pour l'instance initiale $((\mathcal{C}, d), k)$ du problème TSP.

Inversement, supposons qu'il existe une solution σ pour l'instance initiale du TSP. Maintenant, supposons, sans perte de généralité que $v_{\sigma(1)} = l_o$ et que $s_{\sigma(i)}$ soit une tâche dont la localisation est $v_{\sigma(i)}$. Alors, on peut facilement voir que la trajectoire :

$$\langle s_{\sigma(1)} \rightarrow s_{\sigma(2)} \rightarrow s_{\sigma(3)} \rightarrow \cdots \rightarrow s_{\sigma(m)} \rangle$$

est une trajectoire valide (ayant une durée totale d'au plus k). ■

4. Algorithmes et heuristiques

Une approche de type « force brute » pour résoudre le PRTU consiste à parcourir l'ensemble des trajectoires possibles pour en extraire celles qui sont valides, et déterminer, parmi ces trajectoires, celle qui a l'utilité la plus élevée. Bien que l'algorithme de force brute garantisse la détermination d'une solution exacte au problème, son exécution reste très coûteuse en temps. En effet, le nombre total de trajectoires qui doivent être créées et analysées par l'ordinateur est la permutation de k tâches parmi un total de n, où $n = |S|$ et k variant à partir de 1 à n, depuis une trajectoire à une seule tâche jusqu'à une trajectoire à n tâches.

$$\sum_{k=1}^{n} \frac{n!}{(n-k)!} = \frac{n!}{0!} + \frac{n!}{1!} + \cdots + \frac{n!}{(n-1)!}$$

Par exemple, à partir d'un ensemble $S = \{s_1, s_2\}$ peut être dérivé un total de 4 trajectoires possibles : 2 avec deux tâches ($k = 2$) : $\langle s_1 \rightarrow s_2 \rangle$ et $\langle s_2 \rightarrow s_1 \rangle$; et 2 avec seulement une tâche ($k = 1$) : $\langle s_1 \rangle$ et $\langle s_2 \rangle$.

4.1. Algorithme exact

Dans ce scénario, nous présentons une approche améliorée qui donne aussi la réponse exacte du PRTU lorsque l'on considère qu'aucune tâche n'a d'utilité négative. L'algorithme proposé s'appuie sur le lemme suivant, dont la preuve, évidente, est omise.

LEMME 4. — *Pour toute trajectoire $T \subset T'$, $u(T') \geq u(T)$ lorsque l'utilité d'une tâche ne peut pas être négative.*

Par exemple, la trajectoire $T' = \langle A \rightarrow B \rightarrow C \rangle$ dérivée de la trajectoire $T = \langle A \rightarrow B \rangle$, i.e., $T \subset T'$, a une utilité $u(T')$ au moins aussi élevée que $u(T)$, puisque $u_A + u_B + u_C \geq u_A + u_B$. Par conséquent, si une trajectoire valide T peut être étendue par l'ajout d'une nouvelle tâche telle que la trajectoire résultante soit encore valide, la première peut être éliminée en tant que réponse au PRTU, car il existe une trajectoire $T' \supset T$ telle que $u(T') \geq u(T)$. Ainsi, pour définir la réponse au PRTU, l'algorithme trouve l'ensemble des trajectoires valides qui ne peuvent plus être étendues par l'ajout de nouvelles tâches et compare l'utilité de ces éléments. L'essence de cette approche consiste en un algorithme de parcours en profondeur qui utilise des stratégies d'élagage spécifiques pour le PRTU. Nous présentons dans l'algorithme 1 une vue d'ensemble (la fonction principale) de l'algorithme proposé.

Algorithme 1. Algorithme proposé

Entrée : Une instance de PRTU
Sortie : Une trajectoire d'utilité maximale

```
   /* supprime les tâches en dehors de [τ_o, τ_d]          */
1  pour s ∈ S faire
2  |  si τ_1^s > τ_d OU τ_2^s < τ_o alors
3  |  |  S = S − {s} ;
4  |  fin
5  fin
   /* exécute expand pour chaque tâche conservée           */
6  pour s ∈ S faire
7  |  Trajectoire T = new Trajectoire() ;
8  |  Ensemble tâchesCand = S.cloner() ;
9  |  expand(T, s, tâchesCand, u, l_o, τ_o, l_d, τ_d, *meilleureT, G);
10 fin
11 retourner *meilleureT;
```

Dans la fonction principale, la première action effectuée, lignes 1 à 5, consiste à élaguer l'ensemble de tâches $\mathcal{S}$ en supprimant toutes celles qu'il est impossible d'accomplir pendant la période $[\tau_o, \tau_d]$. Puis, pour chaque tâche conservée dans l'ensemble, la fonction *expand* est exécutée (lignes 6 à 10). Cette fonction est très important dans l'algorithme et se charge de créer, ou trouver, de manière récursive, toutes les trajectoires valides dérivées à partir d'une trajectoire de base. L'intuition derrière *expand* est simple. Tout d'abord, elle tente d'ajouter une nouvelle tâche à la fin de la trajectoire de base et puis vérifie si la nouvelle trajectoire est valide. Si elle l'est, la fonction tente de l'étendre davantage récursivement. Si elle ne peut pas être étendue, l'algorithme la compare avec la meilleure trajectoire trouvée jusqu'alors et élague toutes les trajectoires dérivées, puisqu'elles sont invalides. Nous présentons dans l'algorithme 2 la fonction *expand* en détail.

Tout d'abord, la fonction *expand* vérifie, dans la ligne 4, s'il est possible pour la personne (l'exécutant) de se déplacer à partir de la localisation, appelée *position*, de

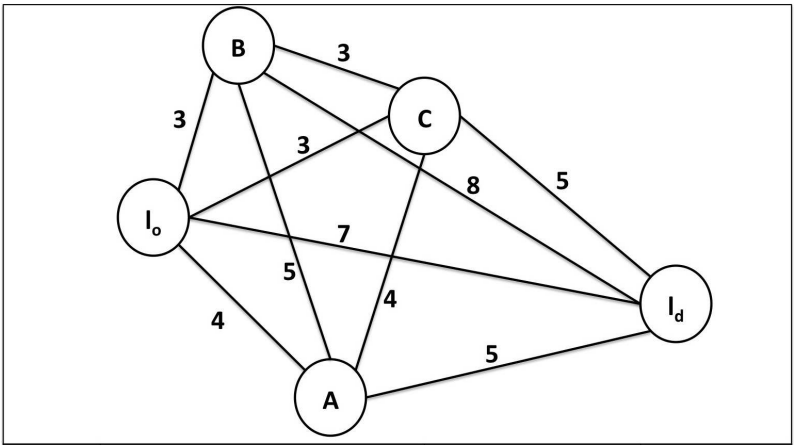
Algorithme 2. Fonction expand

```
1  Fonction expand(T, s, tâchesCand, u, position, τ, l_d, τ_d, *meilleureT, G)
       /* le temps d'arrivé en s ne peut pas être plus
          tôt que τ_1^s                                              */
2      τ = max((τ + G.distance(position, l^s)), τ_1^s);
3      dTâcheDestination = G.distance(l^s, l_d);
       /* vérifie si l'exécutant peut accomplir s et
          arriver à sa destination finale avant
          l'échéance                                                 */
4      si τ + δ^s + dTâcheDestination ≤ τ_d alors
           /* vérifie si l'exécutant peut arriver en s
              avant la fin de la fenêtre de temps                    */
5          si τ ≤ τ_2^s alors
               /* la trajectoire reste valide et s peut
                  être ajoutée.                                      */
6              position = l^s;
7              T.ajouter(s, τ);
8              τ = τ + δ^s;
9              tâchesCand = tâchesCand.cloner();
10             tâchesCand = tâchesCand −{s};
11             peutÉtendre = faux;
               /* essaie d'étendre T récursivement en
                  ajoutant d'autres tâches                           */
12             pour s' ∈ tâchesCand faire
13                 peutÉtendre = peutÉtendre OU expand(T.cloner(), s',
                   tâchesCand, u, position, τ, l_d, τ_d, *meilleureT, G);
14             fin
15             si NON peutÉtendre alors
16                 si *meilleureT = null OU u(T) > u(*meilleureT) alors
17                     *meilleureT = T;
18                 fin
19             fin
20             retourner vrai;
21         fin
22     sinon
23         tâchesCand = tâchesCand −{s};
24     fin
25     retourner faux;
26 fin
```

la dernière tâche de la trajectoire jusqu'à la nouvelle tâche, de l'accomplir et d'arriver à destination avant l'échéance τ_d. Si ce n'est pas possible, la tâche analysée est retirée de l'ensemble des tâches candidates (*tâchesCand*) et de ses dérivées dans la ligne 23. Cette décision est prise car si une tâche E ne peut pas être ajoutée à la fin d'une trajectoire $\langle A \rightarrow C \rightarrow B \rangle$ sans compromettre l'échéance, alors elle ne peut pas non plus être ajoutée à la fin d'une version étendue $\langle A \rightarrow C \rightarrow B \rightarrow D \rangle$. Ainsi, cette tâche n'est jamais analysée à nouveau lorsque l'algorithme tente plus tard d'étendre la trajectoire de base ou ses dérivées.

Dans le cas où une nouvelle tâche s' est ajoutée à la fin d'une trajectoire T sans compromettre l'échéance, la fonction vérifie si l'exécutant qui suit la trajectoire T est capable d'arriver dans la fenêtre de temps $[\tau_1^{s'}, \tau_2^{s'}]$ (ligne 5). Si oui, la nouvelle tâche est ajoutée à la fin de la trajectoire, retirée de *tâcheCand* et la fonction est exécutée à nouveau récursivement pour chaque tâche encore dans *tâcheCand*. Finalement, si la trajectoire n'est étendue par aucune tâche candidate (ligne 15), l'algorithme compare son utilité avec l'utilité de la meilleure trajectoire (*meilleureT*) trouvée jusqu'alors (ligne 16). Dans le cas où la nouvelle trajectoire a une utilité plus élevée, elle devient la meilleure et ainsi de suite.

On peut observer que d'autres stratégies d'élagage peuvent être considérées dans l'algorithme. Par exemple, la meilleure utilité possible est celle d'une trajectoire qui passe par toutes les tâches $s \in \mathcal{S}$ (dans n'importe quel ordre). Lorsqu'une telle trajectoire est trouvée, l'algorithme peut s'arrêter.



Tâche	Fenêtre de temps ($[\tau_1,\tau_2]$)	Duration (δ)	Utilité (u)
A	[1:00 PM, 1:50 PM]	5 minutes	5
B	[1:00 PM, 2:10 PM]	15 minutes	5
C	[1:30 PM, 2:40 PM]	10 minutes	6

Figure 1. Configuration de tâches dans un exemple d'instance de PRTU

Exemple : considérons une instance de PRTU comme le montre la figure 1 où $\mathcal{S} = \{A, B, C\}$, τ_o = 1:15 pm et τ_d = 2:00 pm. Dans ce cas, l'algorithme tente d'abord de retirer de $\mathcal{S}$ toutes les tâches ayant une fenêtre qui n'a pas d'intersection avec [1:15 pm, 2:00 pm] ($[\tau_o, \tau_d]$), aucune dans cet exemple. Ensuite, pour chaque tâche de $\mathcal{S}$, la fonction *expand* est exécutée avec la tâche elle-même et une trajectoire vide comme paramètre. Lorsque la fonction est exécutée avec la tâche A, l'algorithme vérifie si la trajectoire $\langle A \rangle$, résultat de l'ajout de A à la fin de la trajectoire vide, est valide. Comme c'est le cas, il essaie d'étendre la trajectoire en exécutant *expand* récursivement pour chaque tâche $s \in \{B, C\}$ en utilisant $\langle A \rangle$ comme trajectoire de base. La trajectoire $\langle A \rightarrow B \rangle$ est aussi valide et par conséquent l'algorithme essaie de l'étendre à son tour. Cependant, cette fois, $\langle A \rightarrow B \rightarrow C \rangle$ est invalide parce que l'exécutant ne peut pas suivre cette trajectoire et arriver en l_d avant l'instant t_d. Puisque $\langle A \rightarrow B \rangle$ ne peut pas être étendue et qu'aucune autre trajectoire n'a encore été analysée, elle devient la meilleure trajectoire trouvée jusque là. Puis, la récursion revient à $\langle A \rangle$ et essaie de l'étendre en $\langle A \rightarrow C \rangle$, qui est aussi valide et ne peut plus être étendue. Son utilité est donc comparée avec celle de $\langle A \rightarrow B \rangle$ et comme elle est plus élevée, $\langle A \rightarrow C \rangle$ devient la meilleure trajectoire. Ensuite, la récursion amène à considérer une nouvelle recherche avec $\langle B \rangle$ comme trajectoire de base. Une nouvelle meilleure trajectoire est trouvée : $\langle B \rightarrow C \rightarrow A \rangle$. Finalement, l'algorithme cherche des trajectoires meilleures en utilisant $\langle C \rangle$ comme trajectoire de base, mais il n'en trouve aucune et $\langle B \rightarrow C \rightarrow A \rangle$ est renvoyée comme réponse au PRTU. La figure 2 présente l'espace complet de recherche de trajectoires pour cet exemple. Les tâches grises foncées sont des tâches qui font partie de trajectoires invalides qui ont été élaguées alors que les grises claires font partie aussi de trajectoires invalides, mais qui ont été analysées. Bien que dans cet exemple seulement une trajectoire invalide a été élaguée, nos expérimentations avec des ensembles de données synthétiques nous ont montré que cet algorithme est bien plus performant que l'approche de type force brute lorsque le nombre de tâches est élevé.

4.2. Heuristiques

Bien que l'algorithme exact proposé donne une réponse optimale au problème, dans certaines circonstances le nombre de trajectoires valides dans une instance du PRTU peut augmenter considérablement, ce qui rend l'algorithme inefficace ou inutilisable (la stratégie d'élagage ne serait pas performante). Ainsi, afin de pouvoir traiter de telles instances de PRTU, des algorithmes d'approximation et des heuristiques plus performantes, en termes de temps d'exécution, doivent être conçus. Dans cette section, nous proposons cinq heuristiques différentes pour résoudre le PRTU. Bien que quelques heuristiques proposées soient très similaires à des heuristiques déjà bien connues (par example, l'heuristique gloutonne et celle au plus proche voisin), les contraintes spatiotemporelles intrinsèques au PRTU jouent un rôle déterminant dans leur efficacité. C'est pourquoi, il convient de les étudier.

Nous séparons les heuristiques en deux groupes selon leur implémentation. Le premier est composé de trois heuristiques statiques (la séquence des tâches est fixée

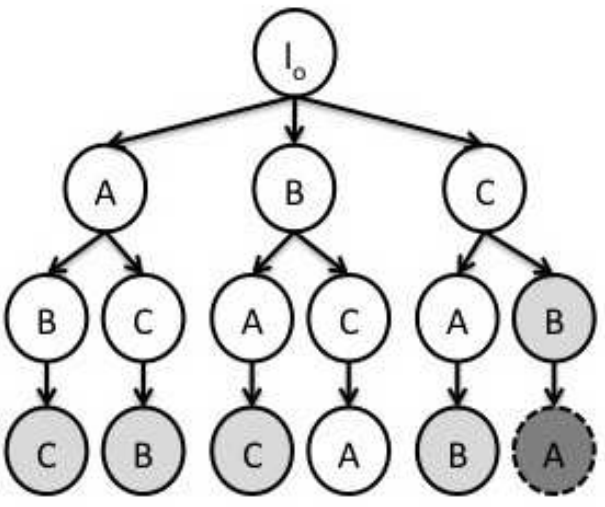


Figure 2. Espace de recherche de trajectoires

à l'avance). Le deuxième groupe est composé de deux heuristiques dynamiques (la séquence est dynamique et dépend de la dernière tâche choisie). Nous pensons que les heuristiques présentées ici sont des solutions acceptables pour le PRTU et que l'ensemble offre un cadre utile à de futures recherches à des approches plus complexes d'approximation pour le problème.

4.2.1. Heuristiques statiques

L'heuristique *gloutonne avec fenêtre de temps* (GFT) est une heuristique qui donne la priorité à l'accomplissement des tâches ayant la plus petite fenêtre de temps. L'intuition est que les tâches qui ont les plus petites fenêtres de temps sont plus difficiles à s'intégrer dans une trajectoire et doivent donc être insérées d'abord. Cela pourrait augmenter le nombre de tâches dans une trajectoire et, par conséquent, son utilité. L'heuristique GFT est présentée en détail dans l'algorithme 3. Tout d'abord, l'ensemble de tâches $\mathcal{S}$ est ordonné selon la taille des fenêtres de temps et de façon croissante. Puis, à partir d'une trajectoire vide, l'heuristique essaie d'étendre la trajectoire en ajoutant la première tâche de la liste ordonnée. Si la trajectoire reste valide, la tâche est ajoutée à titre définitif. Si elle n'est plus valide, la tâche est enlevée de la trajectoire. Après cela, l'heuristique vérifie si la trajectoire serait valide après ajout de la deuxième tâche dans l'ordre, et ainsi de suite jusqu'à la dernière tâche. Un inconvénient de cette approche est la possibilité que survienne un phénomène de famine pour les tâches qui ont des grandes fenêtres de temps : ces tâches sont susceptibles de ne jamais être incluses dans les trajectoires recommandées.

L'heuristique avec *échéance proche* (EP) est une heuristique qui se concentre sur l'augmentation du nombre de tâches accomplies en ajoutant à la trajectoire d'abord

Algorithme 3. Algorithme de l'heuristique gloutonne avec fenêtre de temps (GFT)

Entrée : Une instance de PRTU
Sortie : La meilleure trajectoire trouvée

```
1  Liste tâchesOrdonnées = créeListe(S);
2  ordonnerParFenêtreDeTemps(tâchesOrdonnées);
3  Trajectoire trajectoire = ⟨⟩;
4  pour s ∈ tâchesOrdonnées faire
5  |  append(trajectoire, s);
6  |  si NON estValide(trajectoire) alors
7  |  |  pop(trajectoire);
8  |  fin
9  fin
10 retourner trajectoire;
```

celles qui sont sur le point d'expirer. Cette heuristique essaie d'augmenter indirectement l'utilité d'une trajectoire en augmentent le nombre de tâches accomplies. L'intuition est que les tâches qui sont sur le point d'expirer (c'est-à-dire, dont l'échéance est très proche) doivent être accomplies aussitôt que possible, autrement elles ne le seront jamais. L'heuristique EP est similaire à l'heuristique GFT présentée dans l'algorithme 3. La différence réside sur la ligne 2, où la méthode *ordonnerParFenêtreDeTemps()* est remplacée par une autre qui ordonne selon les échéances. Un inconvénient de cette approche est qu'une tâche est susceptible de ne jamais faire partie d'une recommandation de trajectoires si son échéance n'est pas proche.

L'heuristique *gloutonne avec utilité* (GU) est une heuristique qui se concentre directement sur l'augmentation de l'utilité d'une trajectoire. C'est une heuristique gloutonne classique qui essaie d'augmenter l'utilité d'une trajectoire en sélectionnant d'abord les tâches qui ont les plus hautes utilités. La GU crée également une liste ordonnée de tâches, mais cette fois à partir de la tâche qui a la plus haute utilité jusqu'à celle ayant la plus basse. Le reste de l'heuristique fonctionne comme les deux autres précédemment expliquées. Un inconvénient de cette approche est la possibilité de famine par rapport à des tâches ayant des utilités faibles. Toutefois, étant donné que l'utilité, et par conséquent également l'utilité d'une trajectoire, change selon le profil d'un exécutant, cet événement est moins susceptible de se produire que dans les heuristiques précédemment présentées.

4.2.1.1. Complexité des algorithmes

Étant donné que les trois heuristiques présentées ci-dessus suivent essentiellement le même algorithme, seul change le critère de classement de tâches, leurs complexités sont les mêmes. En effet, pour ces heuristiques, la complexité est déterminée par l'algorithme de tri implémenté. La fonction *estValide* peut être ignorée en toute sécurité parce qu'elle consiste en deux opérations. La première vérifie si l'exécutant est capable d'accomplir la nouvelle tâche dans sa fenêtre de temps et la deuxième s'as-

sure qu'il arrive à sa destination à temps après avoir accompli toutes les tâches qui ont déjà été proposées jusqu'alors. Les heuristiques GFT, EP et GU que nous avons implémentées dans ce travail pour expérimentation utilisent l'algorithme de tri *merge sort* offert par l'API Java, qui a une complexité de $O(n \log n)$ dans le pire des cas, où n représente le nombre de tâches.

4.2.2. Heuristiques dynamiques

L'heuristique au *plus proche* (PP) est une heuristique dynamique et gloutonne qui se concentre sur l'augmentation du nombre de tâches dans une trajectoire et, par conséquent, aussi de son utilité. Ici, l'intuition est qu'en sélectionnant la tâche qui est la plus proche [7] de la dernière tâche ajoutée à la trajectoire, moins de temps est dépensé en voyageant et plus à travailler effectivement sur les tâches. Étant données une trajectoire vide T et une origine l_o, l'heuristique cherche d'abord la tâche la plus proche de l_o qui ajoutée à T ne l'invalide pas. Si aucune tâche répondant à l'exigence n'est trouvée, il n'y a pas de trajectoire valide pour le problème et l'algorithme s'arrête. Par contre, si une tâche est trouvée, elle est ajoutée à T. Puis, l'heuristique cherche la tâche qui ne rend pas la trajectoire invalide et qui est la plus proche de la dernière tâche ajoutée. L'heuristique répète ce dernier pas jusqu'au moment où l'on ne peut plus concaténer de nouvelle tâche sans invalider la trajectoire T. La PP est présentée dans le détail dans l'algorithme 4.

L'heuristique au *Plus Vite* (PV) est une modification de l'heuristique PP qui prend en compte aussi la durée d'une tâche. Plutôt que de simplement chercher la tâche la plus proche, cette heuristique cherche la tâche qui demande le moins de temps pour que l'exécutant puisse voyager jusqu'à sa localisation et l'accomplir. Par exemple, étant données deux tâches B et C, telles que une parmi elles, mais pas les deux, peut être ajoutée au bout d'une trajectoire $T = \langle A \rangle$ sans l'invalider. Supposons que le temps de voyage entre A et B est de 5 minutes et que la durée de la tâche B est de 15 minutes. Supposons aussi que la distance entre A et C est de 10 minutes et que la durée de C est de 5 minutes. Dans ce cas là, bien que la tâche B soit plus proche de A que la tâche C, le temps nécessaire pour accomplir la tâche C (*c.-à-d.*, le temps de voyage plus le temps de travail) est plus court que celui que l'exécutant doit dépenser pour accomplir B. Donc, l'heuristique au Plus Vite ajoute C plutôt que B après A. L'algorithme de PV et celui de PP, présenté par l'algorithme 4, sont essentiellement les mêmes. La différence entre eux est que la durée de la tâche la plus proche, représentée par $tachePlusProche.\delta$, est ajoutée à la variable $nouveauTemps$ dans la ligne 9 au lieu d'être ajoutée à variable $temps$ dans la ligne 20. Un inconvénient de PV est que les tâches qui ont une longue durée sont peu susceptibles d'être incluses dans une trajectoire recommandée. De plus, les tâches plus courtes, qui sont les plus susceptibles d'être incluses, peuvent avoir une petite utilité. Par exemple, si l'utilité est basée principalement sur la récompense des tâches, les tâches accomplies le plus

7. Cette notion de proximité s'entend au sens de la fonction de coût d associée au graphe $\mathcal{G}$.

Algorithme 4. Algorithme de l'heuristique au plus proche (PP)

Entrée : Une instance de PRTU
Sortie : La meilleure trajectoire trouvée

```
1  Ensemble S' = cloner(S);
2  Trajectoire trajectoire = ⟨⟩;
3  Sommet position = l_o;
4  entier temps = t_o;
5  faire
6      Tâche tachePlusProche = null;
7      entier meilleurTemps = ∞;
8      pour s ∈ S' faire
9          entier nouveauTemps = max(temps + G.distance(position, s), s.τ_1);
10         si nouveauTemps < meilleurTemps alors
11             append(trajectoire, s);
12             si estValide(trajectoire) alors
13                 meilleurTemps = nouveauTemps;
14                 tachePlusProche = s;
15             fin
16             pop(trajectoire);
17         fin
18     fin
19     si tachePlusProche est différente de null alors
20         temps = meilleurTemps + tachePlusProche.δ;
21         S' = S' - {tachePlusProche};
22         append(trajectory, tachePlusProche);
23         position = tachePlusProche;
24     fin
25 tant que tachePlusProche est différente de null;
26 retourner trajectoire;
```

vite demandent en principe moins d'efforts, et probablement offrent aussi une plus petite récompense/utilité.

4.2.2.1. Complexité des algorithmes

Encore une fois, les deux heuristiques présentées ici ont la même complexité. Dans le pire cas, la première fois que l'heuristique passe par la structure de répétition *faire tant que* dans l'algorithme 4, il parcourt toutes les n tâches, celle la plus proche ou celle que peut être la plus vite accomplie (cela dépend de quelle heuristique est en cours d'exécution) et l'ajoute à la trajectoire. Ensuite, le même comportement se répète pour les $(n-1)$ tâches restantes jusqu'au moment où il n'y a plus de tâches qui puisse être ajoutée sans rendre la trajectoire invalide. Au total, ce pas se reproduira n

fois, c'est-à-dire : les deux algorithmes s'exécutent $\sum_{k=1}^{n} k = O(n^2)$ fois dans le pire cas.

5. Architecture de référence pour l'emploi de la recommandation

Dans cette section, nous proposons une architecture de référence pour l'emploi de la recommandation de trajectoires utiles dans un CMS réel. La figure 3 présente cette architecture, qui comporte le Gestionnaire de Contexte, le Gestionnaire de RTU, la Base de Données et l'exécutant. Ces éléments constituent les trois principaux composants d'un système de recommandation de trajectoires.

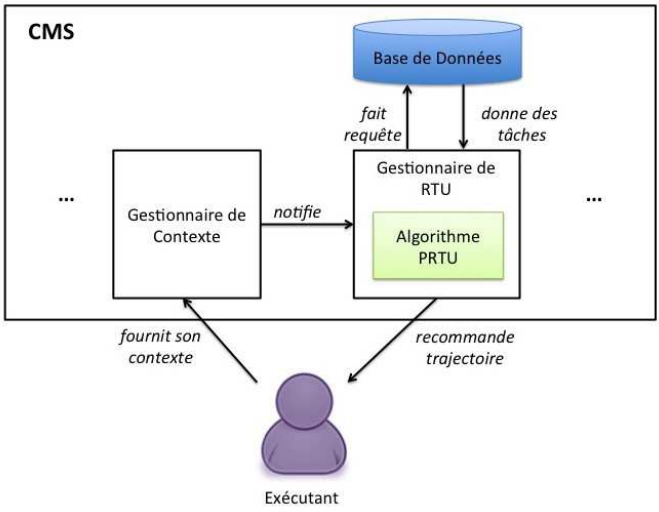


Figure 3. Architecture de référence pour l'emploi de la recommandation

Afin de proposer une recommandation, un algorithme pour le PRTU doit être muni d'une quantité considérable d'informations. Une partie de cette information peut être facilement récupérée à partir de la Base de Données du CMS. Par exemple, pour récupérer l'ensemble de tâches $\mathcal{S}$, une requête peut être effectuée pour demander toutes les tâches qui sont géographiquement et/ou temporellement pertinentes (une tâche au Japon ne concerne pas une trajectoire qui se passe dans une ville située en France). Les informations qui concernent l'exécutant requièrent des opérations plus sophistiquées pour être découvertes. Par exemple, l'intention de voyage $(l_o, \tau_o, l_d, \tau_d)$ d'un utilisateur peut être obtenue par l'analyse d'un agenda virtuel, ou inférée par l'histoire de ses positions sur le GPS ou d'autres techniques. Le rôle du Gestionnaire de Contexte est de faire face à ces complexités en observant l'utilisateur en permanence afin de détecter toute nouvelle intention de voyage (c'est-à-dire, un changement de contexte) et de notifier le Gestionnaire de RTU en passant l_o, τ_o, l_d, τ_d comme paramètres.

Le gestionnaire de RTU (**r**ecommandation de **t**rajectoires **u**tiles) est le composant ayant l'implémentation d'un algorithme pour le PRTU. Lorsque ce composant est

notifié à propos d'une intention de voyage en recevant l_o, τ_o, l_d, τ_d, il fait une requête à la Base de Données en demandant toutes les autres informations pour l'exécution de l'algorithme, (c'est-à-dire, les autres paramètres de l'instance PRTU). Puis, il exécute l'algorithme et envoie une alerte qui recommande le résultat à l'exécutant, qui à son tour peut accepter ou non la proposition.

6. Expérimentations

Afin d'analyser l'algorithme exact et les heuristiques proposées pour le PRTU, nous avons réalisé quelques expérimentations sur des jeux de données synthétiques. Ces données ont été obtenues à travers un programme qui, à partir d'un nombre de tâches donné en entrée, produit en sortie une instance de PRTU générée de façon aléatoire. Dans tous les jeux de données, toutes les fenêtres de temps de toutes les tâches générées avaient une intersection non vide avec la fenêtre de temps de l'exécutant $[\tau_o, \tau_d]$. En outre, la durée des tâches varie de 5 à 30 minutes et la fenêtre de temps entre 20 et 120 minutes. Toutes les tâches sont mises aléatoirement dans un carré de 20 *unités de distance* de côté, où chaque *unité de distance* nécessite 1 minute pour être parcourue. Le coût (temps) de voyage d'une tâche à une autre est donné par la distance euclidienne entre les deux tâches convertie en minutes. Nos expérimentations ont été effectuées sur un Mac Book Pro Retina avec un processeur Intel Core i7 3 GHz et une mémoire vive de 8 Go 1 600 MHz DD3. Le langage de programmation utilisé est Java.

Dans notre première expérimentation, nous avons comparé le temps d'exécution de l'approche de type force brute à celui de l'algorithme exact. Pour cela, nous avons exécuté chacun des deux algorithmes sur 5 000 instances aléatoires de PRTU. Plus spécifiquement, nous avons exécuté chaque algorithme 1 000 fois avec des entrées aléatoires de 5 tâches, puis la même procédure pour des entrées de 6 tâches, et ainsi de suite jusqu'à 9 tâches. Pour cette expérimentation, nous avons fixé la fenêtre de temps entre l'origine et la destination $[\tau_o, \tau_d]$ à 1 heure. La figure 4 montre une comparaison entre les temps d'exécution moyens, en millisecondes, des deux approches. On peut voir qu'avec 9 tâches, le temps d'exécution pour résoudre le PRTU en utilisant la force brute est déjà de 250 millisecondes, alors qu'avec notre approche il est encore proche de zéro (0,064 millisecondes). Bien que dans le pire cas (quand toutes les permutations possibles de tâches sont valides), notre approche peut générer autant de trajectoires que la force brute, dans la pratique, la stratégie d'élagage se montre sensiblement plus performante dans la réduction de l'espace de recherche, ce qui explique la différence entre les deux temps d'exécution.

Dans notre deuxième expérimentation, nous avons analysé le temps d'exécution de notre approche prise isolément. Nous avons exécuté l'algorithme 500 fois avec des entrées aléatoires composées de 10 tâches, puis 500 fois pour des entrées de 20 tâches, et ainsi de suite jusqu'à des entrées de 100 tâches. Au total, l'algorithme a été exécuté 5 000 fois en faisant varier la taille des entrées. La fenêtre de temps entre l'origine et la destination $[\tau_o, \tau_d]$ a été fixée à 1 heure. La figure 5 présente le résultat de l'expérimentation en millisecondes. Même avec 100 tâches, notre algorithme s'est montré

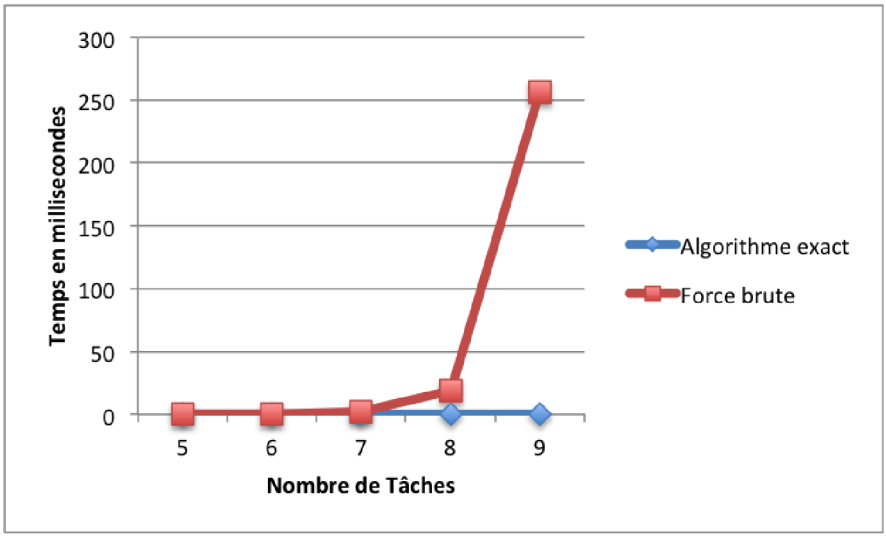


Figure 4. Comparaison entre notre approche et la force brute

capable de donner une réponse au PRTU en un temps moyen de 80 millisecondes. Nous avons également soumis notre algorithme à des instances de PRTU composées de 200 tâches, mais le temps d'exécution moyen s'est déjà élevé à 2,5 secondes.

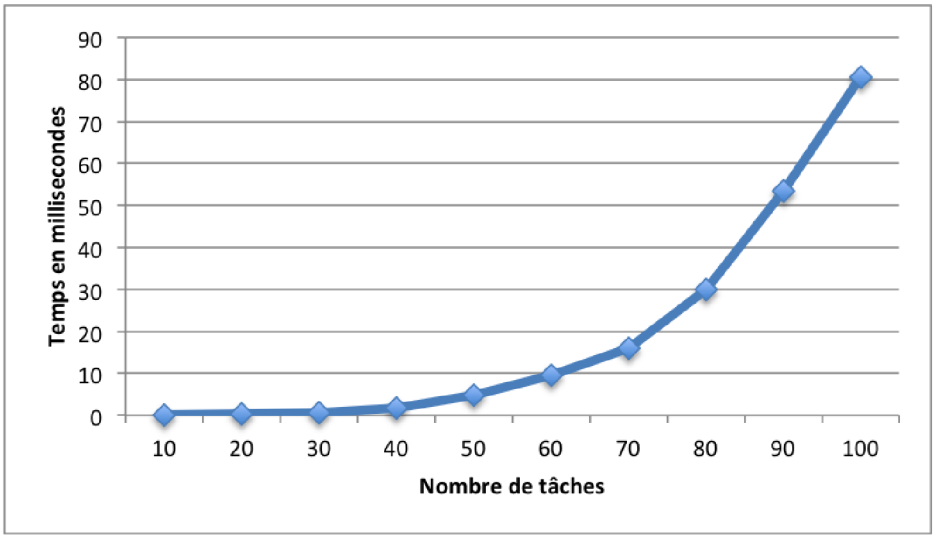


Figure 5. Temps d'exécution de notre algorithme

Bien que l'algorithme exact proposé soit une alternative faisable pour PRTU dans certaines circonstances, lorsque le nombre de trajectoires valides augmente, l'efficacité de la stratégie d'élagage diminue. Dans ce cas, le temps d'exécution peut deve-

nir trop long et l'utilisation de l'algorithme n'est plus conseillée. Alors, nous avons répété l'expérimentation en changeant la fenêtre de temps entre l'origine et la destination $[\tau_o, \tau_d]$ pour une durée de 2 heures. En outre, le nombre de tâches varie de 10 à 40. Dans cette expérimentation, nous avons aussi analysé le temps d'exécution de nos heuristiques en comparaison de l'algorithme exact. En fait, les heuristiques statiques (GFT, EP et GU) partagent la même complexité algorithmique ainsi que les heuristiques dynamiques (PP et PV), nous avons donc choisi une seule heuristique de chaque groupe pour représenter la totalité des heuristiques. Plus précisément, nous avons expérimenté l'algorithme exact et les heuristiques GFT et PP. La figure 6 montre le résultat. Avec la nouvelle fenêtre de temps $[\tau_o, \tau_d]$ de 2 heures, le temps moyen requis par l'algorithme exact pour donner une réponse a nettement augmenté. Lorsqu'il traite des entrées de 40 tâches, le temps d'exécution est déjà de plus d'une seconde. La raison de ce changement par rapport aux expérimentations précédentes est que, lorsque l'exécutant a 1 heure de fenêtre de temps $[\tau_o, \tau_d]$ pour accomplir des tâches, le nombre de trajectoires valides n'est pas si grand. Toutefois, lorsque le temps est augmenté jusqu'à 2 heures, un nombre plus grand de combinaison de tâches peut exister et, par conséquent, le nombre de trajectoires valides augmente, ce qui diminue l'efficacité de la stratégie d'élagage. Le même phénomène peut être reproduit en diminuant le temps moyen des durées de tâches, ce qui a pour effet d'augmenter le nombre de tâches pouvant s'intégrer dans une trajectoire et par conséquent aussi le nombre des trajectoires valides. La figure 7 montre le temps d'exécution des heuristiques proposées dans la même expérimentation.

Même lorsqu'une instance analysée a 40 tâches, le temps moyen d'exécution est de moins de 1 milliseconde (un résultat attendu, compte tenu du fait que la complexité algorithmique des deux groupes d'heuristiques est très faible). Nous avons également expérimenté l'algorithme et les heuristiques en utilisant des instances de PRTU composées de 60 tâches et avec une fenêtre de temps $[\tau_o, \tau_d]$ de 2 heures. Le temps moyen pour que l'approche exacte donne une réponse a été déjà 31 secondes alors qu'aucune des heuristiques n'a dépassé 1 milliseconde.

Notre dernière expérimentation, la plus importante pour évaluer les heuristiques, a été menée pour analyser la qualité des résultats fournis. Dans cette expérience, nous avons généré 100 entrées de 10 tâches, puis 100 de 30 tâches, et ainsi de suite jusqu'à des entrées de 170 tâches, toujours variant de 20 en 20. Ici, pour des raisons de performance, la fenêtre de temps $[\tau_o, \tau_d]$ a été à nouveau fixée à 1 heure. Après cela, pour chaque entrée, nous avons exécuté toutes les heuristiques et l'algorithme exact. Ensuite, nous avons calculé l'utilité moyenne des trajectoires générée par l'algorithme exact pour chaque ensemble d'entrées ayant le même nombre de tâches et comparé ce résultat avec l'utilité moyenne des trajectoires proposées par les heuristiques. La figure 8 présente le graphe des résultats.

Étant donné que l'algorithme exact garantit une solution optimale, son utilité est toujours 100 % de l'utilité moyenne optimale. Quand un petit nombre de tâches est traité, l'heuristique GU se montre la plus performante, fournissant en moyenne 77 % de l'utilité optimale. Cependant, lorsque le nombre de tâches d'une entrée augmente,

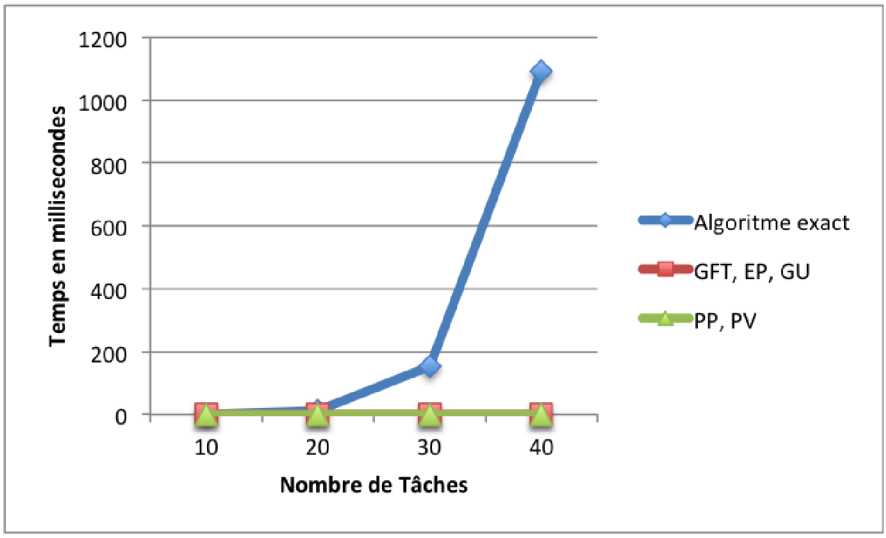


Figure 6. Comparaison des temps d'exécution entre l'algorithme exact et les heuristiques (en utilisant $[\tau_o, \tau_d] = 2$ heures)

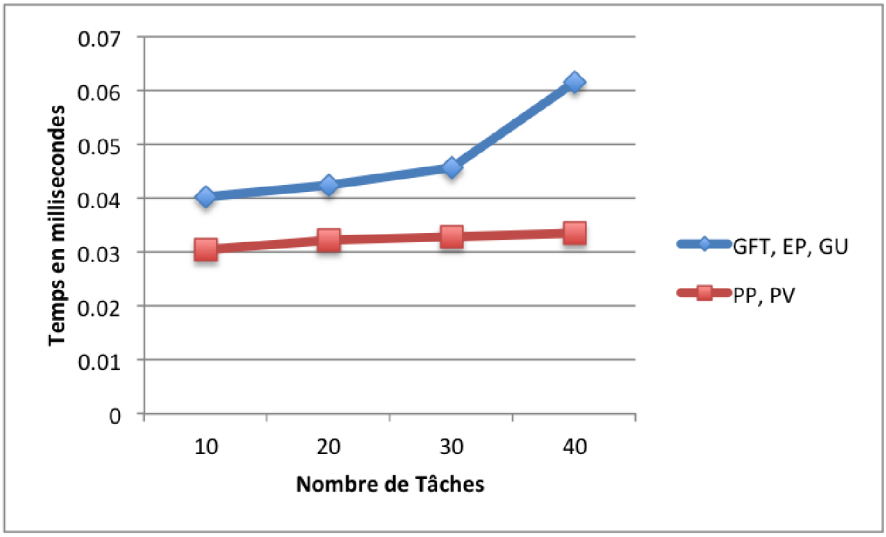


Figure 7. Temps d'exécution des heuristiques (en utilisant $[\tau_o, \tau_d] = 2$ heures)

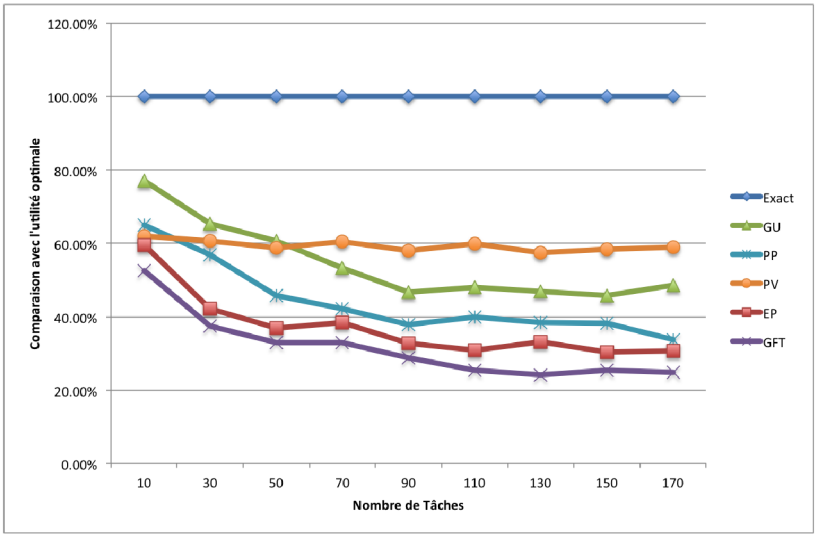


Figure 8. Comparaison entre l'utilité moyenne fournie par les heuristiques et l'utilité optimale

son efficacité diminue de façon constante jusqu'à se stabiliser autour de 47 %, comme la deuxième meilleure option. Pour les instances qui ont un plus grand nombre de tâches, l'heuristique la plus performante est l'heuristique PV. Pendant toute l'expérience, pour des instances de toute taille, elle a fourni en moyenne 59 % de l'utilité optimale, variant légèrement entre 60 % et 59 %.

7. Discussion

L'algorithme exact proposé peut être une solution raisonnable pour le PRTU sur certaines circonstances. Toutefois, plus il existe de trajectoires valides, moins la solution exacte est performante par rapport au temps d'exécution. En règle générale, trois facteurs peuvent augmenter le nombre de trajectoires valides et rendre l'algorithme moins performant ou inefficace :

1. un grand nombre de tâches ($|\mathcal{S}|$) ;
2. une petite durée moyenne de tâches (δ^s) ; et
3. une grande fenêtre de temps de disponibilité de l'exécutant ($[\tau_o, \tau_d]$).

Dans de tels scénarios, on peut utiliser une des heuristiques proposées. L'heuristique gloutonne avec utilité et l'heuristique au plus vite, en particulier, se sont montrées être celles les plus performantes par rapport à l'approximation de l'utilité maximale. Plus précisément, l'heuristique GU est la plus performante lorsque le problème implique un petit nombre de tâches, et l'heuristique PV surpasse celle-ci lorsque le nombre de tâches augmente (à partir de 50 dans nos expérimentations). La différence

entre les temps d'exécution des deux heuristiques est négligeable. Dans un CMS dont la qualité des trajectoires est primordiale, il faut être prudent dans l'utilisation des heuristiques. Les ratios d'approximation montrés ici sont basés sur des résultats moyens empiriques et aucune garantie de qualité minimale n'est fournie.

Bien que nous ne l'ayons pas mentionné, dans un scénario réel l'ensemble de tâches $\mathcal{S}$ peut toujours changer avec le temps. Des nouvelles tâches peuvent être publiées sur le CMS et des tâches anciennes peuvent être accomplies par une autre personne avant que l'exécutant arrive à sa localisation. Toutefois, l'algorithme exact, du moins tel que proposé, est capable de garantir une solution optimale pour le PRTU seulement pour un ensemble de tâches $\mathcal{S}$ immuable. Une solution possible pour ce problème d'algorithme exact est d'exécuter l'algorithme exact à chaque fois que l'ensemble est étendu par l'ajout d'une nouvelle tâche ou lorsque une tâche de l'ensemble est accomplie par quelqu'un d'autre. Cependant, cette approche est très exigeante par rapport à l'utilisation du processeur d'un ordinateur et parfois même infaisable en raison du temps d'exécution nécessaire pour fournir une réponse à certaines instances du PRTU. Comme les heuristiques sont beaucoup plus performantes que l'algorithme exact, ce problème ne se pose pas lorsqu'elles sont utilisées.

8. Conclusion

Dans ce travail, nous avons introduit et formalisé le Problème de Recommandation de Trajectoires Utiles (PRTU). Le PRTU permet à une personne d'accomplir des tâches spatio-temporelles pour lesquelles elle possède un intérêt et/ou des aptitudes, sans compromettre l'arrivée à sa destination avant un temps donné. Nous avons prouvé que PRTU est NP-complet (dans sa version décisionnelle) et nous avons proposé un algorithme exact pour le résoudre. Par ailleurs, étant donné que le PRTU est NP-complet, nous avons proposé et étudié l'utilisation de cinq heuristiques pour trouver une solution approximative au PRTU. Enfin, nous avons aussi proposé une architecture de référence pour l'emploi de la recommandation de trajectoires utiles dans un CMS.

Nos expérimentations ont montré que l'utilisation de l'algorithme exact semble être performant dans un scénario comportant jusqu'à cent tâches et lorsque l'exécutant a une fenêtre de temps d'une heure pour accomplir des tâches. Dans ce cas là, la trajectoire optimale est trouvée en quelques millisecondes. Lorsque le nombre de tâches s'élève à quelques centaines, la réponse est donnée en quelques secondes. Nos expériences ont révélé aussi que les heuristiques proposées sont des alternatives faisables pour trouver de bonnes solutions au PRTU lorsque l'approche exacte n'est pas performante. L'heuristique gloutonne avec utilité a fourni en moyenne jusqu'à 77 % de l'utilité maximale pour des instances qui n'ont pas beaucoup de tâches et l'heuristique au plus vite 55 % pour des instances de toute taille.

Comme travaux futurs, nous envisageons d'expérimenter différentes combinaisons d'heuristiques proposées ici pour essayer d'en trouver d'autres qui sont plus performantes par rapport à l'approximation de l'utilité maximale. Par exemple, il sera

intéressant d'examiner une heuristique qui classe les tâches selon le ratio $utilité/(distance + duration)$ d'une tâche. Nous proposons d'étudier également si la similarité entre une trajectoire recommandée et la trajectoire que l'exécutant emprunte normalement peut influencer ce taux d'acceptation. Enfin, nous envisageons d'étudier les aspects dynamiques du problème d'affectation de tâches, tels que la prise en compte des nouvelles tâches qui ont été publiées ou des tâches qui ont été accomplies par d'autres exécutants.

Remerciements

Ce travail est soutenu par le Ministère de l'Enseignement Supérieur et de la Recherche.

Bibliographie

Ambati V., Vogel S., Carbonell J. G. (2011). Towards task recommendation in micro-task markets. In *Human computation.* Consulté sur http://www.aaai.org/ocs/index.php/WS/AAAIW11/paper/view/4005

Deng D., Shahabi C., Demiryurek U. (2013). Maximizing the number of worker's self-selected tasks in spatial crowdsourcing. In *Proceedings of the 21st acm sigspatial international conference on advances in geographic information systems*, p. 324–333. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2525314.2525370

Difallah D. E., Demartini G., Cudré-Mauroux P. (2013). Pick-a-crowd: Tell me what you like, and i'll tell you what to do. In *Proceedings of the 22nd international conference on world wide web*, p. 367–374. Republic and Canton of Geneva, Switzerland, International World Wide Web Conferences Steering Committee. Consulté sur http://dl.acm.org/citation.cfm?id=2488388.2488421

Fonteles A. S., Bouveret S., Gensel J. (2014). Towards matching improvement between spatio-temporal tasks and workers in mobile crowdsourcing market systems. In *Proceedings of the third acm sigspatial international workshop on mobile geographic information systems*, p. 43–50. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2675316.2675319

Kazemi L., Shahabi C. (2012). Geocrowd: Enabling query answering with spatial crowdsourcing. In *Proceedings of the 20th international conference on advances in geographic information systems*, p. 189–198. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2424321.2424346

Kazemi L., Shahabi C., Chen L. (2013). Geotrucrowd: Trustworthy query answering with spatial crowdsourcing. In *Proceedings of the 21st acm sigspatial international conference on advances in geographic information systems*, p. 314–323. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2525314.2525346

Kittur A., Smus B., Khamkar S., Kraut R. E. (2011). Crowdforge: Crowdsourcing complex work. In *Proceedings of the 24th annual acm symposium on user interface software and technology*, p. 43–52. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2047196.2047202

Lin C. H., Kamar E., Horvitz E. (2014). Signals in the silence: Models of implicit feedback in a recommendation system for crowdsourcing. Consulté sur http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8425

Ra M.-R., Liu B., La Porta T. F., Govindan R. (2012). Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the 10th international conference on mobile systems, applications, and services*, p. 337–350. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2307636.2307668

Yuen M.-C., King I., Leung K.-S. (2012). Task recommendation in crowdsourcing systems. In *Proceedings of the first international workshop on crowdsourcing and data mining*, p. 22–26. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/2442657.2442661