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

André Sales Fonteles, Sylvain Bouveret, Jérôme Gensel

LIG, Université Grenoble Alpes,
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 quelqu'un 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 par des personnes 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 qui veut se déplacer d'accomplir des tâches spatio-temporelles pour lesquelles elle montre une grande affinité et/ou aptitude, sans compromettre son arrivée dans les temps à destination. Nous démontrons que le* PRTU *est NP-complet (dans sa version décisionnelle) et lui proposons un algorithme exact. En outre, nous proposons une architecture de référence pour aider dans l'emploi de la recommandation de ces trajectoires dans un CMS réel. Enfin, nos expérimentations montrent que l'algorithme proposé est une solution faisable pour les instances de* PRTU *ayant quelques centaines de tâches ou moins.*

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. In this paper, 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 show that* PRTU *is NP-complet (in its decision version) and propose a exact algorithm 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 is a feasible solution for instances of* PRTU *with a few hundreds tasks or less.*

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

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

1. Introduction

Au cours des dernières années, de nombreux systèmes de *crowdsourcing* (CMS, pour *Crowdsourcing Market Systems*) sont apparus, dans lesquels 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, où les commanditaires peuvent publier des tâches spatio-temporelles qui requièrent d'être dans un lieu et à un moment spécifique. Sereale [3], TaskRabbit [4] et Medusa (Ra *et al.*, 2012) sont des exemples de tels systèmes. 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 filmer 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.

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 voyager 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 trouver une telle trajectoire, un Problème de Recommandation de Trajectoires Utiles, ou bien 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 Recommandation de Trajectoires Utiles et étudions sa complexité. Dans la section 4,

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 représentés par une fonction d'utilité $u(s)$

nous présentons une méthode naïve pour solutionner le PRTU, avant de proposer un algorithme exact plus performant. 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, quelques 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 leurs utilités, 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 prise 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 tâches pour estimer l'utilité d'une trajectoire, en faisant l'hypothèse que la première est déjà estimée (par exemple en utilisant des méthodes similaires à celles proposées dans les références mentionnées ci-dessus) et donnée comme entrée au problème PRTU.

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 de 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 recomman-

der 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 par une personne en un lieu et une fenêtre de temps. Nous caractérisons une tâche comme 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, $\tau_1^s \in \mathbb{N}$ 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 δ^s est la durée de la tâche. 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 *;*

– $u : \mathcal{S} \to \mathbb{R}^+$ *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 peux quitter le lieu* l_o *;*

– τ_d *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 de arêtes avec* $v \neq v'$ *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 tout 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 doit 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 toute $(s_i^T, t_i) \in T$, $t_i \in [\tau_1^{s_i^T}, \tau_2^{s_i^T}]$ (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, dans la suite de cet article, nous assimilons une trajectoire à une séquence de tâches sans les temps d'arrivée. Par exemple, soient A, B et C des tâches et soit $T = \langle (A, t_A), (C, t_C), (B, t_B) \rangle$ un exemple de trajectoire comprenant ces tâches. La trajectoire T peut être également représentée par $\langle A \to C \to B \rangle$.

Étant donné une trajectoire T, l'utilité de T correspond alors simplement à la somme de toutes les utilités associées à chaque tâche présente dans la trajectoire, à savoir :

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

Maintenant, nous pouvons assembler les concepts précédents et définir le Problème de Recommandation de Trajectoires Utiles de la manière suivante :

PRTU	
Entrée :	Un tuple $(\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G})$.
Sortie :	Une trajectoire T valide d'utilité maximale parmi l'ensemble des trajectoires valides.

3.1. Complexité du problème

La complexité du problème PTRU 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

du document), 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 donné un tuple $(\mathcal{S}, u, l_o, l_d, \tau_o, \tau_d, \mathcal{G})$ 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 donnés 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 $C = c_1, \ldots, c_m$, 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 donnée 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 $((\mathcal{C}, d), k)$ du problème TSP initialement présenté.

Inversement, supposons qu'il existe une solution σ pour l'instance initiale de 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

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 ($k = 2$): $\langle s_1 \rightarrow s_2 \rangle$ and $\langle s_2 \rightarrow s_1 \rangle$; et 2 avec seulement une tâche ($k = 1$): $\langle s_1 \rangle$ and $\langle s_2 \rangle$.

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 élevé 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.

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

```
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
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;
```

Algorithme 1 : Algorithme proposé

Dans la fonction principale, le première action effectuée 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 trajectoire conservée dans l'ensemble, la fonction la plus importante de l'algorithme est exécutée : *expand*. La fonction *expand* 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 détails.

Tout d'abord, la fonction *expand* vérifie s'il est possible pour la personne (l'exécutant) de se déplacer à partir de la localisation, appelée *position*, de 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. Cette décision est prise parce que 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 $[\tau_1^{s'}, \tau_2^{s'}]$. Si oui, la nouvelle tâche est

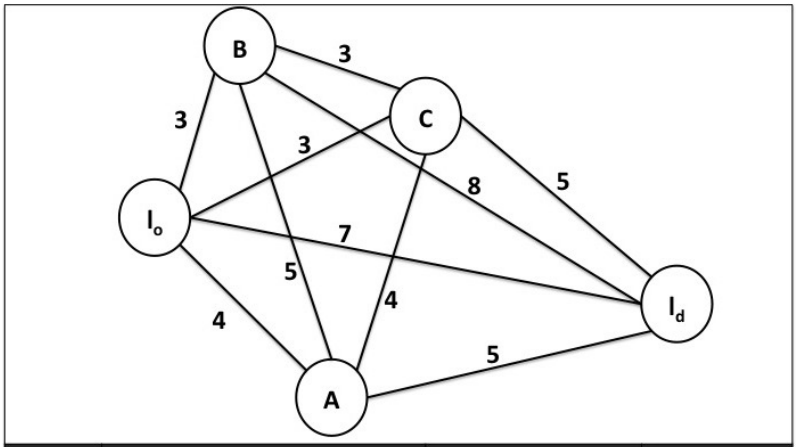
```
1  Fonction expand(T, s, tâchesCand, u, position, τ, l_d, τ_d, *meilleureT, G)
2      τ = max((τ + G.distance(position, l^s)), τ_1^s);
3      dTâcheDestination = G.distance(l^s, l_d);
4      si τ + δ^s + dTâcheDestination ≤ τ_d alors
5          si τ ≤ τ_2^s alors
6              position = l^s;
7              T.ajouter(s, τ);
8              τ = τ + δ^s;
9              tâchesCand = tâchesCand.cloner();
10             tâchesCand = tâchesCand −{s};
11             peutÉtendre = faux;
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 NOT 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
```

Algorithme 2 : Fonction expand

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, l'algorithme compare son utilité avec l'utilité de la meilleure trajectoire (*meilleureT*) trouvée jusqu'alors. 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 quelle ordre). Lorsqu'une telle trajectoire est trouvée, l'algorithme peut s'arrêter.

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



Tâche	Fenêtre de temps ($[\tau_1,\tau_2]$)	Duration (δ)	Utilité (u)
A	[1:00 PM, 1:50 PM]	5 minutes	0.5
B	[1:00 PM, 2:10 PM]	15 minutes	0.5
C	[1:30 PM, 2:40 PM]	10 minutes	0.6

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

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 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.

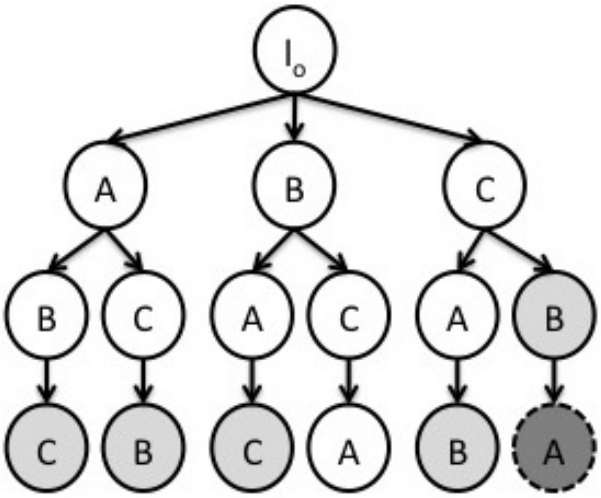


Figure 2. Espace de recherche de trajectoires.

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 consiste en les principaux entités et composants qui jouent un rôle important dans la recommandation de trajectoires utiles : le Gestionnaire de Contexte, le Gestionnaire de RTU, la Base de Données et l'exécutant.

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 regardent 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 nombreuses techniques différentes. Le rôle du Gestionnaire de Contexte est de faire face à ces complexités en observant l'utilisateur en permanence, détecter toute nouvelle intention de voyage (c.-à-d.: un changement de contexte) et notifier le Gestionnaire de RTU en passant l_o, τ_o, l_d, τ_d comme paramètres.

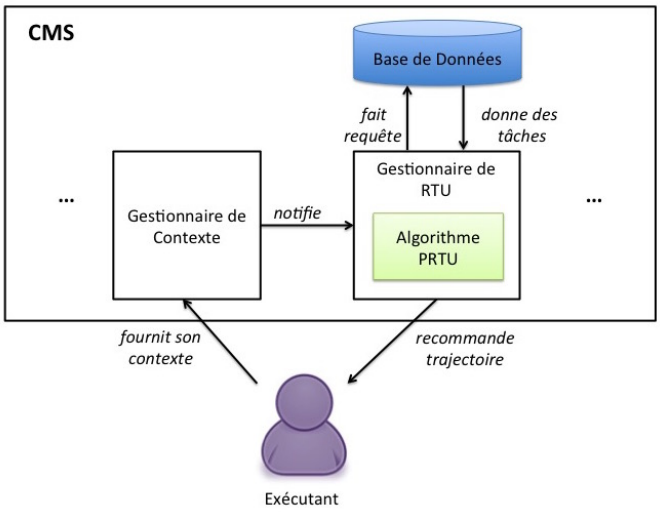


Figure 3. Architecture de référence pour l'emploi de la recommandation.

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 la faisabilité de l'utilisation de notre algorithme pour solutionner le PRTU, nous avons réalisé deux 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, nous avons fixé la fenêtre de temps entre l'origine et la destination $[\tau_o, \tau_d]$ à 1 heure. En outre, nous nous sommes assurés que toutes les fenêtres de temps de toutes les tâches générées avaient une intersection non vide avec la fenêtre de temps de l'exécutant $[\tau_o, \tau_d]$. Nos expérimentations ont été effectuées sur un Mac Book Pro Retina avec un processeur Intel Core i7 3GHz et une mémoire vive de 8Go 1600MHz DD3. Le langage de programmation utilisée a été Java.

Dans notre première expérimentation, nous avons comparé le temps d'exécution de l'approche de type force brute à celui de l'algorithme. Pour cela, nous avons exécuté chacun des deux algorithmes sur 5000 instances aléatoires de PRTU. Plus spécifique-

ment, nous avons exécuté chaque algorithme 1000 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. 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 solutionner 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 performante dans la réduction de l'espace de recherche, ce qui explique la différence entre les deux temps d'exécution.

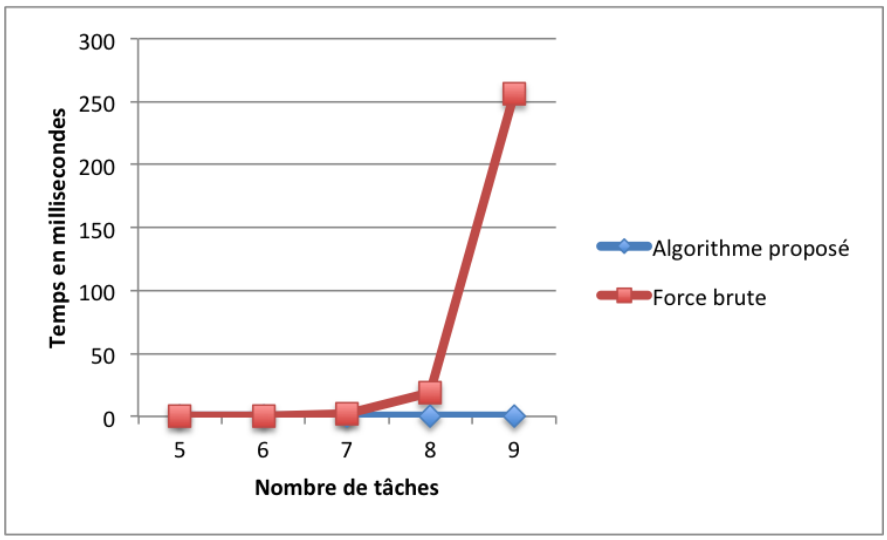


Figure 4. Comparaison entre notre approche et la force brute.

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 entrés de 100 tâches. Au total, l'algorithme a été exécuté 5000 fois en faisant varier la taille des entrées. La figure 5 présente le résultat de l'expérimentation en millisecondes. Même avec 100 tâches, notre algorithme s'est montré 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 élevé à 2,5 secondes.

Nos expérimentations montrent que l'algorithme proposé est une solution faisable pour le PRTU offrant une réponse exacte pour des entrées comptant jusqu'à 100 tâches dans un temps moyen de moins de 80 millisecondes. De plus, si un CMS n'a pas besoin d'une réponse immédiate pour la recommandation de la trajectoire, même des entrées avec quelques centaines de tâches peuvent être traitées dans quelques secondes. Comme dit précédemment, bien que dans le pire de cas l'algorithme génère

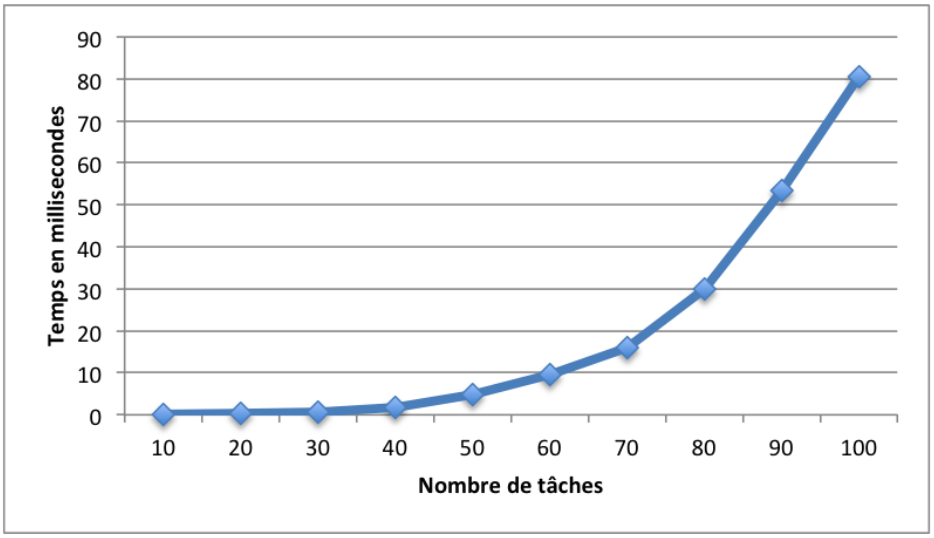


Figure 5. Temps d'exécution de notre algorithme.

autant de trajectoires que la force brute, l'expérimentation a montré que sa stratégie d'élagage réduit sensiblement l'espace de recherche et le temps d'exécution.

7. 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 pour lesquelles elle possèdent un intérêt et/ou des aptitudes, sans compromettre l'arrivée à sa destination avant un temps limite. Par ailleurs, nous avons prouvé que PRTU est NP-complet (dans sa version décisionnelle) et nous avons proposé un algorithme exact pour le résoudre. Enfin, nous avons aussi proposé une architecture de référence pour l'emploi de la recommandation de trajectoires utiles dans un CMS réel. Nos expérimentations ont montré que l'utilisation de notre algorithme est performant dans un scénario comportant jusqu'à cent tâches en donnant une réponse en quelques millisecondes, et quelques secondes lorsque le nombre de tâches s'élève à quelques centaines.

Comme travaux futurs, nous proposerons des algorithmes d'approximation qui donnent une réponse immédiate au PRTU lorsque plusieurs tâches sont impliquées. Nous prévoyons aussi d'expérimenter le taux d'acceptation des utilisateur pour les trajectoires recommandées (c'est-à-dire, à quelle fréquence l'utilisateur accepte de suivre les trajectoires qui lui sont recommandées). Enfin, nous proposons d'étudier si la similarité entre une trajectoire recommandée et la trajectoire que l'exécutant emprunte normalement peut influencer ce taux d'acceptation.

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.*

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.

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