Problème du placement géographique de labels

Rapport de travail de diplôme

Décembre 2003 - Révision N°2

Auteur : Grégory Burri
Prof. responsable : Éric Taillard

École d'ingénieurs du canton de Vaud eivd
Route de Cheseaux 1
CH-1400 Yverdon-les-bains, Suisse

Résumé

Le problème du placement de labels consiste à disposer des légendes autour de points en minimisant les chevauchements. Afin de faciliter l'application d'algorithmes, on limite le nombre possible de positions d'un label par rapport à son point. Le problème ainsi posé reste NP-difficile, il est donc obligatoire de recourir à des heuristiques de recherche. La méta-heuristique POPMUSIC [4] permet de décomposer les problèmes en sous-problèmes qui seront eux-mêmes traités à l'aide d'une heuristique de recherche avec Tabous [1].
Les résultats obtenus sont de très bonne qualité en un temps court comparativement à d'autres algorithmes comme CGA [2]. Un petit algorithme est également mis en oeuvre sous la forme d'une deuxième passe après l'optimisation de placement qui permet de masquer les labels qui entrent en collision, un poids est assigné à chacun des labels afin de savoir lesquels masquer pour ne garder que les plus importants. Par exemple pour le placement de noms de villes sur une carte géographique le poids pourrait être le nombre d'habitants.

Mots-clefs

"placement d'étiquettes", "placement de labels", "placement de légendes", "recherche avec tabous", "POPMUSIC", "heuristique", "meta-heuristique"

Table des matières

1 Introduction

Le problème du placement de labels, ou étiquettes, se rencontre très fréquemment dans le domaine de la cartographie. Il est très important de positionner correctement les labels, comme par exemple les noms de villes ou de lieux, sur une carte géographique afin d'en faciliter la lecture. Ce problème est relativement fastidieux à réaliser à la main, c'est pour cela qu'une approche automatisée s'impose.

Dans ce document nous abordons le problème d'étiquetage de points en appliquant une méthode heuristique de recherche avec tabous [1] puis en l'améliorant à l'aide d'une méta-heuristique POPMUSIC [4]. Les points à étiqueter peuvent être des villes, des églises, des sommets de montagnes etc. Il existe deux autres types d'étiquetage qui sont la dénomination de zones comme les lacs ou les forêts et la dénomination de droites ou de courbes telles que les routes ou rivières, ces deux types de problèmes ne seront pas abordés ici mais pourraient être résolus de la même manière.

La recherche avec tabous a été adoptée car elle donne de bons résultats sans avoir recours à des algorithmes trop complexes. L'heuristique POPMUSIC va permettre de découper le problème en sous-problèmes de taille fixe puis d'appliquer une recherche avec tabous à chacun des sous-problèmes. Cette méthode va permettre de faire baisser la complexité tout en augmentant la qualité de la solution finale.

Ce document est organisé comme suit. En section 2 le problème du placement de labels est exposé, la section 3 est consacrée à la recherche avec tabous, le calcul ainsi que la mesure de sa complexité et l'étalonnage des paramètres sont abordés. En section 4 la meta-heuristique POPMUSIC, appliquée comme une couche au dessus de la recherche avec tabous, est présentée. La section 5 permet de se rendre compte des différences de performances entre POPMUSIC et la recherche avec tabous en les comparant puis POPMUSIC est comparé à d'autres algorithmes de la littérature comme CGA [2] en section 6. En section 7 la pondération des labels est introduit. En section 8 l'implémentation des méthodes vu aux sections précédentes est exposée, certaines structures de données employés y sont présentées. La section A liste les outils et bibliothèques utilisées pour l'implémentation. Finalement La section 9 est consacrée à la conclusion. Les codes sources sont fournis sous la forme d'annexes.

2 Placement de labels

Nous considérons deux variables pour placer un label, sa position par rapport au point et le nombre de chevauchements que le label engendre.

La position du label est discrétisée afin d'en facilité l'application d'algorithmes, c'est à dire qu'un label ne peut pas se placer n'importe où autour de son point mais seulement à des endroits prédéterminés. Quatre positions sont utilisées dans le reste de ce document, la valeur de ce nombre n'affecte en rien l'application des méthodes de calcul. Les quatre positions sont en haut à gauche, en haut à droite, en bas à gauche et en bas à droite. À chaque position est attribuée une valeur de préférence, "0" étant la position la plus avantagée alors que "1" la plus désavantagée, cette valeur va entrer dans le calcul du coût d'une solution en la pondérant avec le nombre de chevauchements Ceci à pour but d'homogénéiser au maximum le placement des labels par rapport à leur point afin d'en faciliter la lecture. La figure 1 montre les quatre posibilités de placement avec un exemple de poids pour chaque position.

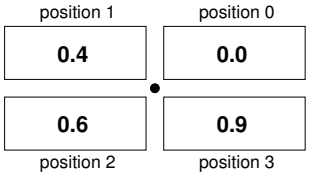


FIG. 1 – *Positions possibles pour un label autour de son point.*

Si nous définissons p comme étant le nombre de positions d'un label et n le nombre total de labels alors le nombre de possibilités s'élève à p^n ce qui est énorme pour des problèmes de taille même modeste. Par exemple pour 200 labels à placer et quatre positions possibles, ce qui représente un problème relativement petit, le nombre de solutions possibles qu'il faudrait analyser avec un algorithme exact serait de $4^{200} = 2.6 \cdot 10^{120}$ ce qui représente un nombre plus élevé que tous les atomes de l'univers.

Il est naturellement impensable d'énumérer toutes les solutions possibles et de choisir la meilleure parmi celles-ci. On qualifie alors ce problème de NP-difficile, c'est à dire qui se rapporte à un comportement polynomial non déterminé. Il faut donc adopter non pas un algorithme déterministe qui va trouver LA solution mais une méthode de recherche qui soit la plus intelligente possible, la recherche avec tabous et POPMUSIC sont appliquées à ce problème en section 3 et respectivement 4, il en existe beaucoup d'autres qui en général s'inspirent de phénomènes naturels comme les algorithmes génétiques [2] ou les colonies de fourmis.

Les figures 2 et 3 montrent une résolution de problème de taille 200 à l'aide de POPMUSIC.

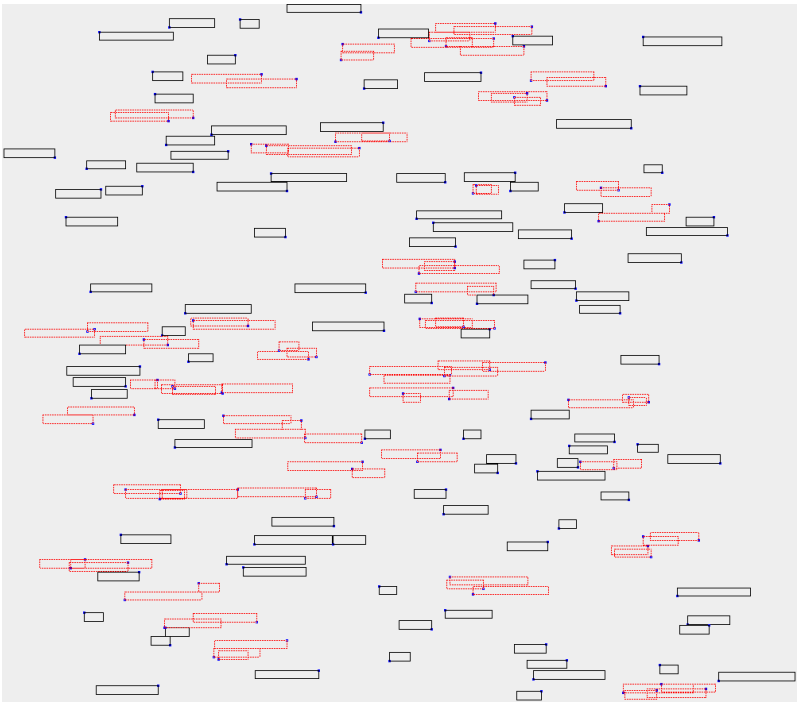


FIG. 2 – *Problème de taille 200, configuration initiale.*

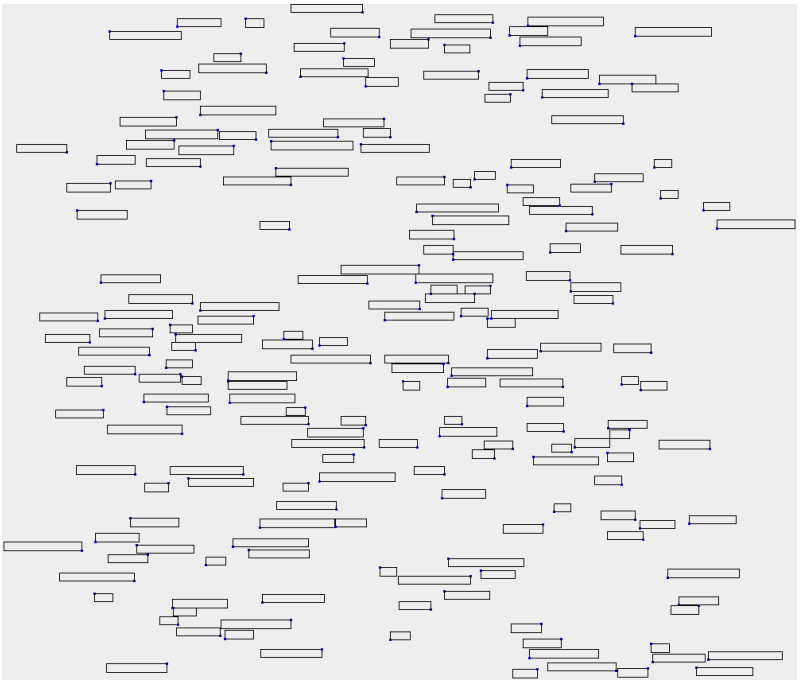


FIG. 3 – *Résolu en moins d'une seconde avec POPMUSIC.*

3 Recherche avec tabous

3.1 Introduction

La recherche avec tabous consiste à empêcher certains changements d'états pendant un certain temps afin de ne pas tourner en rond dans les solutions. Le but n'est pas de directement trouver la solution en ne cessant d'améliorer notre résultat courant, car ceci va forcément aboutir à un état bloqué, on ne pourrait alors plus effectuer de changement sans détériorer notre solution.

Si l'on accepte alors certaines solutions moins bonnes que les précédentes, tout en ayant mémorisé la meilleure, il est fort probable que l'on retombe sur l'optimum local puis que finalement on ne fasse que tourner en rond.

Pour palier à ce problème une liste de changements tabous est mise en place. cette liste consiste à mémoriser les derniers changements effectués sur la solution afin de les interdire par la suite. Cette liste est de taille limitée, au bout d'un certain nombre d'itérations des modifications redeviennent accessibles.

Dans notre cas nous avons choisi une liste de taille variable calculée en fonction du nombre de chevauchements courants, cette taille sera mise à jour toutes les n itérations :

$$taille = min_t + int(p_t \cdot nLCh)$$

$nLCh$ est le nombre de labels chevauchants au moins un autre label.
min_t est la taille minimale de la liste tabou.
p_t est un facteur de pondération par rapport au nombre de chevauchements.

Les constantes min_t et p_t sont ajustées à la section 3.3 à l'aide d'une méthode de recherche.

3.2 Approche

Voici l'algorithme utilisé, les phases 1 et 2 sont considérées comme des phases d'initialisation.

1. Calculer et mémoriser tous les chevauchements potentiels possibles, c'est à dire que pour chaque position de chaque label il faut énumérer quels autres labels pourraient entrer en collision si ils se trouvent dans une certaine position.
2. Placer tous les labels dans la meilleure position dictées par les préférences. On ne tient pas compte des chevauchements. Ceci est la solution initiale.
3. Pour un certain nombre d'itérations, effectuer.
   (a) Créer la liste des candidats, elle contient tous les labels ayant le coût le plus élevé.
   (b) Pour chaque candidat calculer sa meilleure position.
   (c) Prendre le candidat le meilleur, c'est à dire celui qui dégrade le moins ou qui améliore le plus la solution. Si il n'améliore pas la solution et se trouve dans la liste tabou alors on passe à l'itération suivante.
   (d) Ajouter le candidat à la liste tabou si il ne s'y trouve pas déjà.
   (e) Mettre à jour la solution courante en effectuant la modification.
4. FIN.

3.2.1 Calcul du coût

Le but de l'algorithme est de baisser au maximum le coût de la solution, on peut écrire :

$$\sum_{i=0}^{n-1} C\,(i)$$

où
$C\,(i)$ est le coût du label i

n est le nombre de labels

La somme des coûts des labels est la fonction à minimiser.

Le coût d'un label se calcule ainsi :

$$C(i) = \alpha_1 \cdot nbChevauchement(i) + \alpha_2 \cdot préférence(i)$$

où
$nbChevauchement(i)$ est le nombre de chevauchements que produit le label i.
$preference(i)$ est la préférence de la position du label comme vu en section 2.
α_1 et α_2 sont des valeurs de pondération suivant ces règles :

$$0 \leq \alpha_1 \leq 1$$
$$0 \leq \alpha_2 \leq 1$$
$$\alpha_1 = 1 - \alpha_2$$

Par exemple un α_1 égal 0.8 et un α_2 égal 0.2 donnera plus d'importance aux chevauchements plutôt qu'a la position relative du label.

3.2.2 La liste de candidats

La liste de candidats a pour but de mieux cibler les coûts à minimiser et de réduire le temps de calcul pour un volume de labels important.

La liste est de taille variable, elle est recalculée, comme la liste tabou, toutes les m itérations à l'aide de la formule suivante :

$$taille = min_c + int(p_c \cdot nLch)$$

min_c est la taille minimale de la liste.
$nLch$ est le nombre de labels chevauchants au moins un autre label.
p_c est un facteur de pondération par rapport au nombre de chevauchements.

De la même manière que la taille de la liste tabou, ces constantes sont ajustées à l'aide d'une méthode de recherche, voir la section 3.3 pour plus de précisions.

Un problème rencontré avec la liste de candidats se présente lorsque tous les labels de la liste font aussi partie de la liste tabou et tous sont dans la position optimale, il n'est donc pas possible d'en séléctionner un.

Pour pallier à ce problème, lorsque qu'il survient, nous augmentons p_c par un facteur f_a (a pour 'agrandissement') ce qui a pour effet d'augmenter le nombre de candidats. Puis à chaque itération ce facteur est divisé par f_r (r pour 'réduction') jusqu'a revenir au facteur d'origine de p_{base}.

Ces constantes sont également définis en section 3.3.

3.3 Méthode d'étalonnage des constantes

3.3.1 Les constantes

Afin d'obtenir les meilleurs résultats possibles il est necessaire de définir les valeurs les plus appropriées pour un certain nombre de constantes. Cet étalonnage se fait pour un nombre d'iterations ainsi qu'une taille de problème bien précis. Les constantes sont listées ci-après avec à chaque fois une plage de valeurs définie de manière intuitive.

- p_{base} est le facteur minimum de pondération de la taille de la liste des candidats, il est le p_c initial $[0.1, 0.8]$;
- f_r est le facteur de réduction du facteur p_c $[1.1, 1.5]$;

- f_a est le facteur d'agrandissement du facteur p_c $[5, 20]$;
- p_t est le facteur de pondération de la taille de la liste tabou $[0.1, 0.8]$;
- m définit la fréquence de recalcul de la taille des listes tabous et de candidats $[10, 50]$;
- min_c est la taille minimum de la liste des candidats $[2, 20]$;
- min_t est la taille minimum de la liste tabou $[2, 10]$

3.3.2 Approche

Le but est de faire varier les constantes afin de trouver un résultat meilleur, un algorithme de descente est utilisé, il est présenté ci-dessous.

1. Effectuer un certain nombre de fois, suivant la précision voulue. À chaque itération on traite une variable différente ⇒ variable courante.
   (a) Si la plage de valeur de la constante courante est très fine alors on passe à la constante suivante.
   (b) Réalisation de deux séries de calculs pour les deux valeurs extrèmes de la constante courante (série A pour la valeur inférieure et série B pour la valeur supérieure). Les valeurs des autres constantes sont calculées comme étant la moyenne de leurs deux valeurs extrêmes.
   Les deux séries sont identiques, le générateur est initialisé, au début de la série, avec le même germe.
   Le résultat est calculé comme étant le nombre de labels qui en chevauche au moins un autre.
   (c) On compare les deux résultats, si ils sont identiques on ne fait rien, si le résultat de la série A est meilleur alors on rapproche la valeur supérieure d'un certain pourcentage vers la valeur inférieure. Si c'est la série B qui est meilleure alors on rapproche la valeur inférieure d'un certain pourcentage vers la valeur supérieure.
2. FIN.

Pour que le résultat soit pertinent il faut maximiser le nombre de calculs dans une série pour obtenir un maximum de configurations différentes. Cet algorithme n'est pas utilisé pour optimiser la recherche avec tabou simple car on ne peut pas prédire l'ordre de grandeur de la taille du problème, de plus pour des problèmes de tailles moyennes ou supérieures, le temps de calcul assez elevé limiterait la taille des séries. Par contre, cet optimisation est utilisée dans le cas de POPMUSIC (voir section 4.4) où la taille des sous-problèmes est petite et connue.
les sources de cet algorithme sont fournis en C.30.

3.4 Initialisation

L'initialisation de la structure de données représentant tous les chevauchements potentiels entre label, voir le premier point de l'algorithme en section C.30, est réalisée à partir de données sous la forme de coordonnées géographiques.

Voici l'algorithme de construction de la structure de données :

* Pour chaque label ⇒ l_1
  * Pour chaque position p_1 de l_1
    * Pour chaque label ⇒ l_2
      * Pour chaque position p_2 de l_2
        * Si l_1 dans la position p_1 entre en conflit avec l_2 dans la position p_2 alors le mémoriser sans la structure de données.

On se rend rapidement compte que la complexité est en $O(n^2)$ en sachant que le nombre de positions d'un label ne dépend pas du nombre de labels. Comme cette étape peut être mémorisée, sous la forme d'un fichier par exemple, le temps de calcul qu'elle engendre n'est pas pris en compte

dans le calcul et la mesure de la complexité des algorithmes de recherche avec tabous et POPMUSIC.

Le tableau 1 ainsi que la figure 4 montrent des temps d'initialisations en fonction de nombres de labels. Pour eviter l'influence des éventuels processus tournant sur la machine, chaque mesure est effectuée dix fois. La fonction de tendance, en blanc, montre que la complexité est plutot d'ordre $O(n^2)$.

Taille	**Temps [ms]**
50	30.00
100	109.00
200	428.00
400	1669.00
800	6579.00
1600	25994.00
3200	103198.00
6400	422216.00

TAB. 1 – *Mesures de la complexité de la phase d'initialisation.*

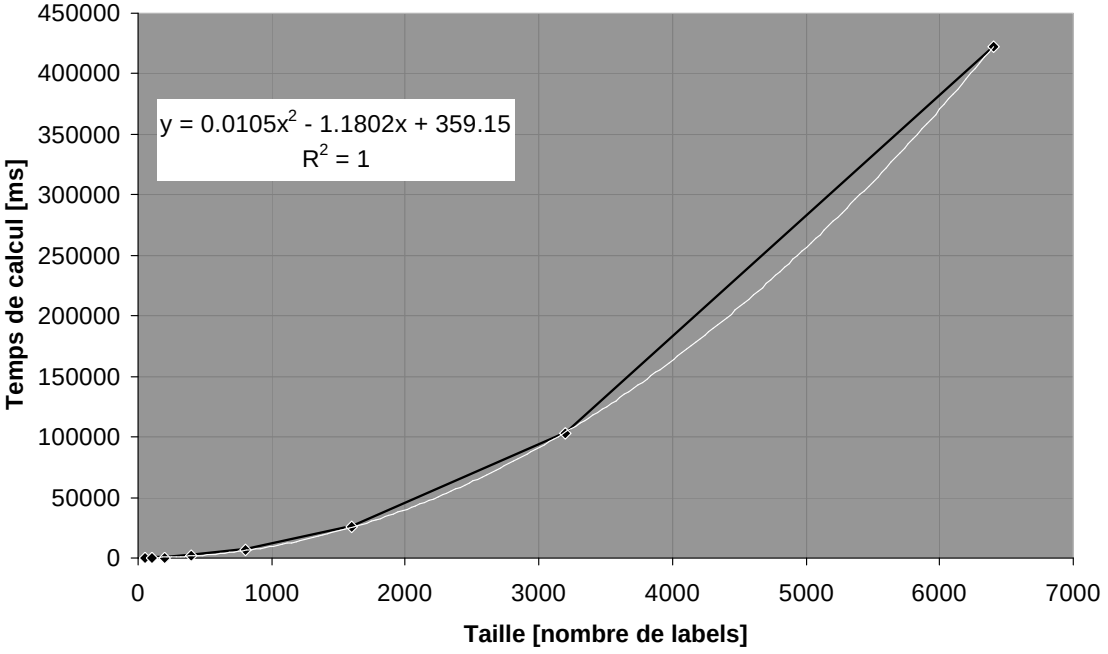


FIG. 4 – *Graphique de la complexité de la phase d'initialisation.*

Les codes sources réalisant ces calculs sont fournis en C.31

3.5 Complexité

La complexité est un point fondamental pour évaluer les performances d'un algorithme. Dans un premier temps elle est calculée de façon théorique (3.5.1) puis des mesures sont effectuées (3.5.2) pour appuyer les calculs. On suppose que la densité de labels ne varie pas en fonction de la taille du problème c'est à dire que la moyenne de collision d'un label avec un autre est constante quel que soit le nombre de labels.

3.5.1 Théorique

Les étapes de l'algorithme de la section 3.2 sont reprises ici afin d'en évaluer leur complexité, les étapes une et deux sont ignorées car elles font partie de l'initialisation.

1. La complexité est constante car le nombre d'itérations est une constante.

a) Constant. La liste de candidats s'appuie sur une liste de labels triée par coûts ce n'est donc qu'une manipulation de référence.

b) Pour calculer le coût d'un label il faut tester chaque collision potentielle, comme on suppose que la densité est constante alors le nombre de labels pouvant entrer en collision est constant. La taille de la liste des candidats dépend du nombre de labels on peut donc dire que le point b) est de complexité n.

c) Cette opération vérifie pour chaque candidat si il ne se trouve pas dans la liste tabou. Cette recherche dépend de la taille de la liste tabou qui elle-même dépend du nombre de chevauchements donc de la taille du problème. Mais le nombre de chevauchements diminue en général rapidement pour tendre vers une valeur constante, la complexité baisse donc rapidement pour atteindre n.

d) Temps constant, c'est un simple ajout à la fin d'une liste.

e) La mise à jour de la solution nécessite de maintenir un liste de labels triés. Cela se fait en $n \cdot log(n)$.

La complexité globale de l'algorithme est en $n \cdot log(n)$.

3.5.2 Mesuré

Les mesures ont été effectuées avec les paramètres suivants :
- Nombre d'itérations : 5000
- Distance entre le label et son point : 0
- Poids des chevauchements par rapport à la position : 1

Les calculs sont réalisés pour des tailles de problèmes allant de 50 à 6400 par pas de facteur deux. Pour que la densité reste constante la taille de la zone est calculée en fonction du nombre d'étiquettes de cette manière : $c = 2 \cdot \sqrt{n}$, c est la longueur d'un côté de la zone qui est de forme carrée et n est le nombre de labels. Chaque mesure est la moyenne de dix calculs ayant chacun une configuration initiale propre, le tableau 2 et la figure 5 montrent l'accroissement du temps de calcul en fonction de la taille du problème. On constate que la croissance du temps de calcul est quasiment linéaire.

Taille	**Temps [ms]**
50	388.00
100	749.00
200	1507.00
400	4726.00
800	15269.00
1600	32222.00
3200	72267.00
6400	151966.00

TAB. 2 – *Mesures de la complexité de la recherche avec tabous.*

Les codes sources réalisant ces calculs sont fournis en C.31

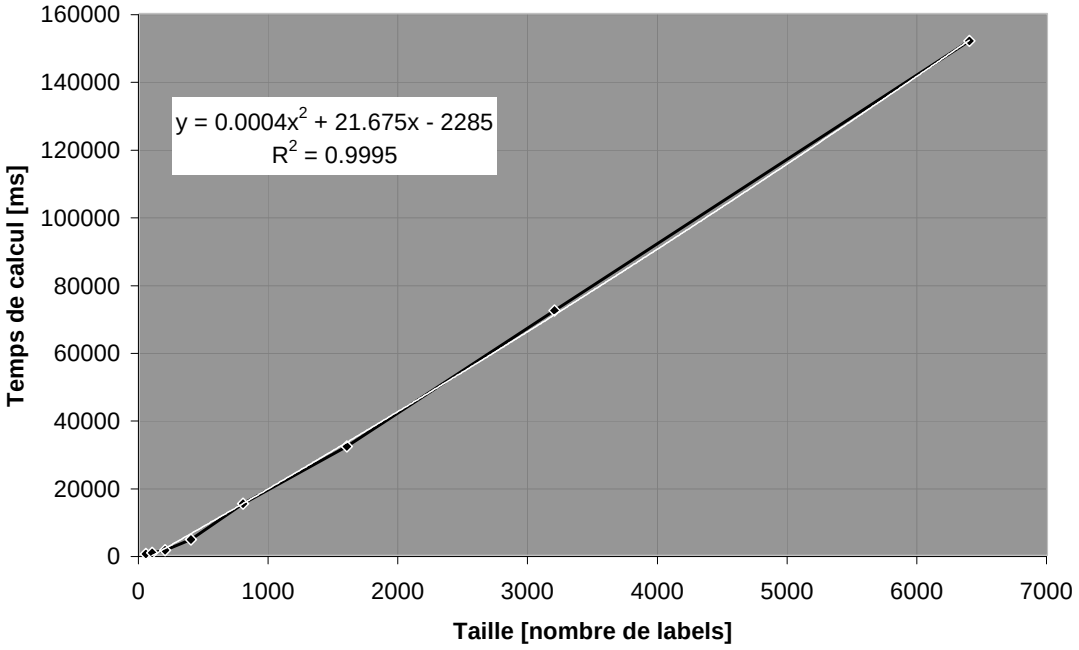


FIG. 5 – *Graphique de la complexité de la recherche avec tabous.*

4 POPMUSIC

4.1 Introduction

Nous avons pu constater que la recherche avec tabous est très rapide mais la qualité de ses solutions décroît fortement avec la croissance de la taille du problème, voir section section 5 page 20 pour la comparaison de la qualité entre la recherche avec Tabous et POPMUSIC[4]. Une idée serait d'effectuer de petites optimisations locales successives. La méta-heuristique POPMUSIC[4] propose justement une méthode d'optimisation pour les problèmes pouvant être découpés et résolus de manière locale, chaque sous-problème est ensuite résolu en utilisant la recherche tabou vu en section 3.

4.2 Approche

1. O est un ensemble de points de la solution, initialement vide.
2. r est la taille maximale d'un sous-problème.
3. Tant que O ne contient pas tous les points, répéter.
   (a) Choisir un point de façon aléatoire qui ne se trouve pas dans O.
   (b) Créer un sous-problème dont la taille n'excède pas r à partir du point choisi (voir 4.2.1).
   (c) Appliquer la recherche avec tabous sur le sous-ensemble.
   (d) Si la solution globale est meilleure alors on la met à jour avec les changements effectués sur le sous-ensemble et on vide O, si elle est moins bonne ou égale alors on ajoute le point choisi à O.
4. FIN.

Remarques :
- La valeur d'une solution est calculée comme étant le nombre de labels entrant en colision.
- Lors de l'optimisation avec tabou d'un sous ensemble, les labels hors de ce sous-ensemble sont pris en compte dans le calcul du coût de la solution.

4.2.1 Création d'un sous ensemble

Le sous-ensemble se construit en partant du point et en mémorisant ses voisins puis les voisins des voisins etc. Un point est considéré comme voisin d'un autre si il peut potentiellement entrer en conflit avec celui cî, la relation est bijective. La figure 6 montre le voisinage en couches d'un point.

Voici l'algorithme utilisé.

1. La pile P est initialisée avec le point choisi.
2. Le nombre de voisins précédent v_p est initialisé à 1.
3. Le nombre de voisin courant v_c est initialisé à 0.
4. Tant que v_p n'est pas égal à zéro, répéter.
   (a) $v_c \leftarrow 0$
   (b) Pour chacun des v_p derniers points de P, répéter.
      i. Empiler le voisin dans P.
      ii. Si la taille de P est égale à r (taille maximal) alors FIN.
      iii. incrémenter v_c de 1
   (c) $v_p \leftarrow v_c$
5. FIN.

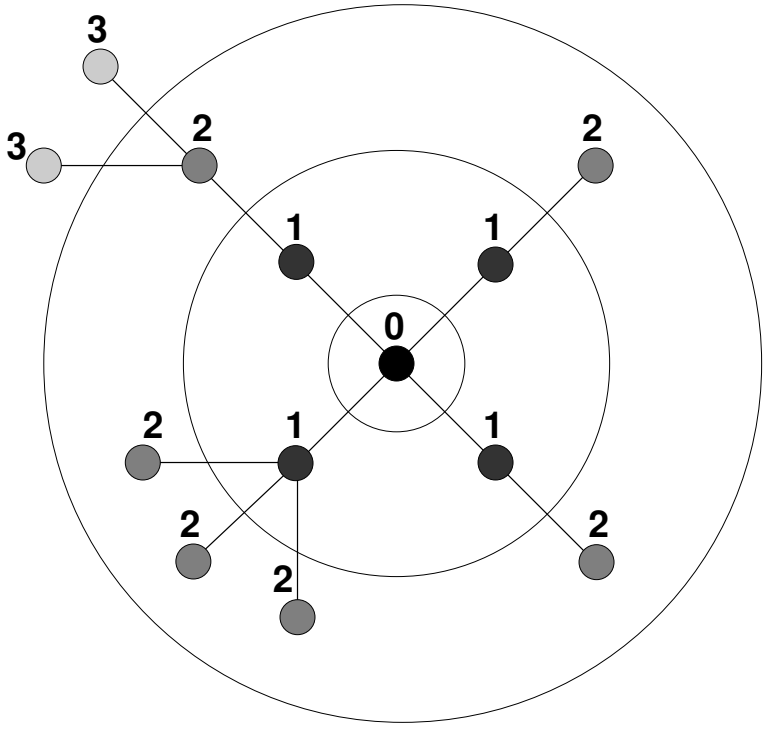


FIG. 6 – *Voisinage en couches d'un point, chaque arête représente un chevauchement potentiel.*

4.3 Modification de POPMUSIC

Une modification est apportée à POPMUSIC, elle consiste à modifier légèrement le point d) de l'algorithme de la section 4.2. Au lieu d'ajouter uniquement le point qui a servi de germe on ajoute tout le sous-ensemble à O. Cette modification, appelée POPMUSIC v2, donne des résultats un peu moins bon que la version initiale mais en des temps nettement meilleurs. Ceci peut se voir à la section 6.

4.4 Étalonnage des constantes

Grâce au découpage en sous-problèmes nous pouvons estimer la taille moyenne des sous-ensembles de points et donc appliquer la méthode d'étalonnage des constantes de la recherche avec Tabous vu à la section 3.3. Les paramètres *nombre itération* et *taille sous problème* ont été définis de manière empirique après plusieurs essais sur des problèmes dont la taille varie entre 1000 et 5'000.

- *nombre itération* = 70
- *taille sous problème* = 75

Le nombre moyen de la taille des sous problèmes est de 40 pour une densité égale aux problèmes de taille 1000 se trouvant ici : `http://www.lac.inpe.br/~lorena/download/missae/d1000/`. L'étalonnage est réalisé avec cette valeur.

Les valeurs initiales des constantes sont celles proposées en section 3.3. Chaque série de calculs porte sur 3 fois 500 configurations différentes avec à chaque fois une densité d égale à 2, 2.5 et 3 (la densité définit la taille des côtés de la zone dans laquelle vont être placés les points : $c = d \cdot \sqrt{n}$).

Les valeurs trouvées après 86 itérations sont listées ci-dessous :

- p_{base} : 0.73 ;
- f_r : 1.3 ;
- f_a : 15 ;
- p_t : 0.77 ;
- m : 47 ;

– min_c : 18 ;
– min_t : 9

4.5 Complexité

4.5.1 Théorique

Que ce soit pour la version une ou deux on ne peut pas prédire combien de fois chaque sous-problème va devoir être optimisé, mais on peut affirmer qu'ils le seront au moins tous une fois. La complexité est donc en $Omega(n \cdot complexité tabou)$ où n est le nombre d'étiquettes. La complexité de la recherche avec tabous étant en $O(n \cdot log(n))$ la complexité de POPMUSIC avec tabous est : $Omega(n^2 \cdot log(n))$

4.5.2 Mesuré

Les mesures ont été effectuées avec les paramètres suivants.
– Taille des sous-problèmes : 70
– Nombre d'itération : 75
– Distance entre le label et son point : 0
– Poids des chevauchements par rapport à la position : 1

Les valeurs des constantes sont les mêmes que celles de la section 4.4.

Les conditions de mesures (densité, nombre de calcul) sont les mêmes que celles de la recherche avec tabous, référez vous à la section 3.5.2. le tableau 3 et la figure 7 montrent l'accroissement du temps de calcul en fonction de la taille du problème pour la version 1 et 2 de POPMUSIC.

On constate que la croissance du temps de calcul est proche des prévisions, l'algorithme numéro 1 possède une complexité plus grande mais donne aussi des solutions de qualité légérement supérieures.

Taille	Temps [ms] - Version 1	Temps [ms] - Version 2
50	260.00	45.00
100	920.00	155.00
200	3390.00	527.00
400	9603.00	1568.00
800	28511.00	4947.00
1600	73538.00	13928.00
3200	194895.00	43031.00
6400	-	140997.00

TAB. 3 – *Mesures de la complexité de POPMUSIC.*

Les codes sources réalisant ces calculs sont fournis en C.31

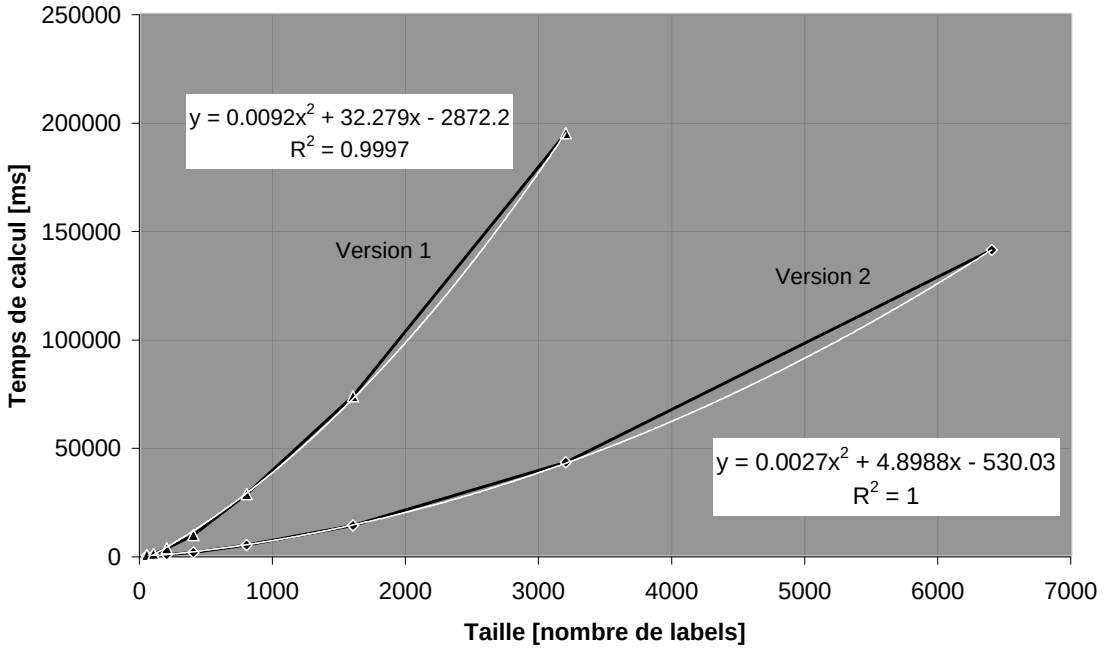


FIG. 7 – *Graphique de la complexité de POPMUSIC.*

5 Comparaison entre Tabou et POPMUSIC v1

5.1 Outil utilisé

Un outil de comparaison, voir [3], est utilisé afin de voir clairement la différence d'efficacité entre les deux heuristiques. Pour cela un fichier par heuristique est généré, il contient le temps de calcul et la qualité de la solution pour un nombre d'itérations allant de 10 à 2'560 par pas de facteur 2. Chaque calcul est reproduit 15 fois avec des configurations de problèmes initiales différentes.

5.2 Paramètres pour les mesures

Pour la recherche avec tabous, les mêmes paramètres que ceux utilisés à la section 3.5.2 sont employés, les paramètres de POPMUSIC sont les même que ceux de la section 4.5.2.

5.3 Résultat

La figure 8 montre le résultat obtenu avec un problème de taille 500, on observe de façon evidente que l'heuristique POPMUSIC trouve de meilleures solutions plus rapidement.

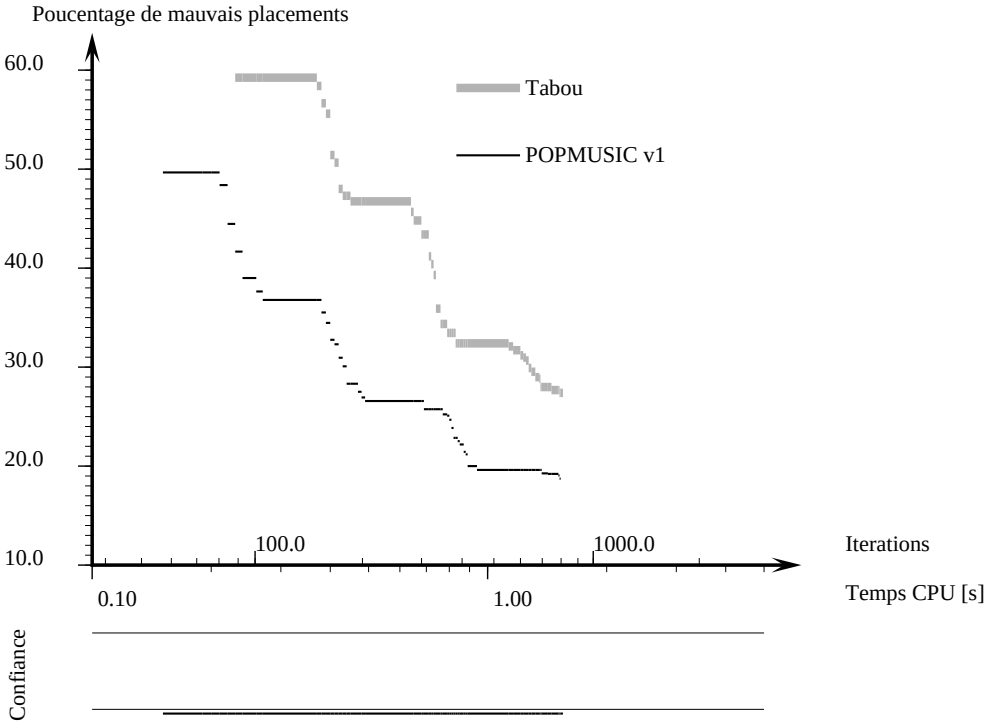


FIG. 8 – *Comparaison entre la recherche avec tabous et POPMUSIC.*

6 Comparaison de POPMUSIC avec d'autres algorithmes

6.1 Conditions de mesure

Pour évaluer les performances en termes de vitesse mais surtout de qualité de solution, une comparaison de POPMUSIC avec d'autres algorithmes est faite ici, les données concernant les autres algorithmes sont reprises de la section 4 de [2]. Les instances de problèmes se trouvent ici `http://www.lac.inpe.br/~lorena/instancias.html`. Voici une énumération des conditions.

1. Le nombre de points n : 100, 250, 500, 750, 1000 ;
2. Pour chaque problème nous calculons la moyenne du pourcentage de labels placés sans conflit sur 25 configurations initiales ;
3. Les préférences de placement, voir section 2 ne sont pas prise en compte ;
4. Les paramètres de l'algorithme sont listés ci-après et **sont utilisés pour toutes les instances et pour les deux versions sauf indication contraire**.

Voici les paramètres utilisés, voir section 3.3 et 4.4 pour plus d'informations.

- nombre d'itérations : 70
- tailles des sous-problemes :
  - version 1 : 40
  - version 2 : 75
- p_{base} : 0.73
- p_r : 1.3
- p_a : 15
- p_t : 0.77
- f : 47
- min_c : 18
- min_t : 9

6.2 Résultats

Le tableau 4 montre le poucentage de labels placés sans collision (les valeurs de POPMUSIC sont arrondies au dixième de décimale). POPMUSIC version 1 et version 2 donne des résultats meilleurs que tous les autres algorithmes et ceci en un temps nettement plus court, le tableau 5 montre les temps de calculs. Les calculs avec POPMUSIC ont été faits sur un Pentium III 747Mhz, l'auteur de [2] ne précise pas la fréquence de la machine utilisée, il est simplement dit qu'ils ont été effectués sur un Pentium III.

les sources du programme se trouvent à l'annexe C.32.

Algorithme	**100**	**250**	**500**	**750**	**1000**
POPMUSIC + TS version 1	100.0	100.0	99.6	97.4	92.3
POPMUSIC + TS version 2	100.0	100.0	99.5	97.2	91.6
CGA_{best}	100.00	100.00	99.6	97.1	90.7
$\text{CGA}_{average}$	100.00	100.00	99.6	96.8	90.4
Tabu search	100.00	100.00	99.2	96.8	90.00
GA with masking	100.00	99.98	98.79	95.99	88.96
GA	100.00	98.40	92.59	82.38	65.70
Simulated Annealing	100.00	99.90	98.30	92.30	82.09
Zoraster	100.00	99.79	96.21	79.78	53.06
Hirsh	100.00	99.58	95.70	82.04	60.24
3-Opt Gradient Descent	100.00	99.76	97.34	89.44	77.83
2-Opt Gradient Descent	100.00	99.36	95.62	85.60	73.37
Gradient Descent	98.64	95.47	86.46	72.40	58.29
Greedy	95.12	88.82	75.15	58.57	43.41

TAB. 4 – *Résultats numériques en terme de qualité.*

Algorithme	**100**	**250**	**500**	**750**	**1000**
POPMUSIC + TS version 1	0.0	0.0	0.3	3.5	20.0
POPMUSIC + TS version 2	0.0	0.0	0.2	1.3	4.4
CGA_{best}	0	0.6	21.5	228.9	1227.2
$\text{CGA}_{average}$	0	0.6	21.5	195.9	981.8
Tabu search	0	0	1.3	76.0	352.9

TAB. 5 – *Temps de calcul pour trouver la meilleure solution.*

7 Pondération des labels

7.1 Introduction

Pour un plan ou une carte géographique cela n'est pas acceptable qui il y ait des chevauchements d'étiquettes car la lisibilité devient très difficile. Dans certains cas, comme par exemple lors d'un zoom à grande échelle, ces chevauchements deviennent inévitables, il faut alors masquer certaines étiquettes. Pour ce faire une notion de pondération est introduite, un poids est attribué à chaque étiquette qui va déterminé si il faut l'afficher ou non dans le cas où il y a chevauchement. Au plus le poids est grand, au plus l'étiquette aura des chances de ne pas être masquée.

7.2 Algorithme

L'algorithme qui va déterminer quel vont être les labels à masquer va être appliqué comme une seconde passe après POPMUSIC+TS v2.

1. Trier les étiquettes par ordre de poids, du plus grand au plus faible.
2. Pour chaque étiquette triée.
   (a) elle n'est pas masquée alors masquer toutes les étiquettes qui la chevauchent.
3. FIN.

La complexité de cet algorithme est en $O(n \cdot log(n))$ de part sa phase de tri préliminaire.

Le tri est introduit à cause de cas particuliers, la figure 9 montre 3 labels de poids différents, si par exemple on les traite dans l'ordre 1, 2 puis 3, le 1 va masquer le 2 puis le 2 ne va pas être traité puisqu'il est masqué et finalement le 3 va masquer le 1, on n'aura donc au final plus que le 3 qui sera visible alors que le 2 pourrait l'être aussi. Il existe encores d'autre cas particuliers qui ne sont pas vus ici. Le fait de trier les étiquettes garantit que celles de plus gros poids seront affichées même si elles en chevauchent beaucoup d'autres de plus faible poids.

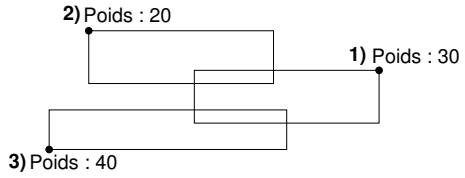


FIG. 9 – *Cas spécial obligeant un tri préliminaire.*

7.3 Application pratique

Dans le but d'illustrer cette technique et de la mettre en pratique sur un exemple un peu plus concret, POPMUSIC+TS et la pondération sont appliqués à la Suisse puis un zoom est effectué sur la zone situé entre Lausanne et Neuchâtel pour avoir une plus grande précision de lecture.

Plusieurs essais réalisés sur la carte de la Suisse, notament en faisant varier la taille des sous-ensembles et en disposant initialement les labels de manière aléatoire (au lieu de la meilleure position), ont montrés que les résultats sont meilleurs avec des sous-ensembles de taille d'environ 70 et en les disposants initialement de manière aléatoire, nous ne tenons alors plus compte de placements relatifs des labels.

Le placement aléatoire a permis de passer de 882 labels affichés à 911 sur un total de 2863. Le temps d'initialisation est d'environ 145 secondes et le temps de calcul de 230 secondes. La structure de données générée par l'initialisation pourrait être stockée dans un fichier ce qui ferait disparaitre le temps de la première phase.

La figure 10 montre la configuration initiale, puis elle est optimisée à l'aide de POPMUSIC+TS v2 et masquée à la figure 11. La figure 12 montre le changement d'échelle.

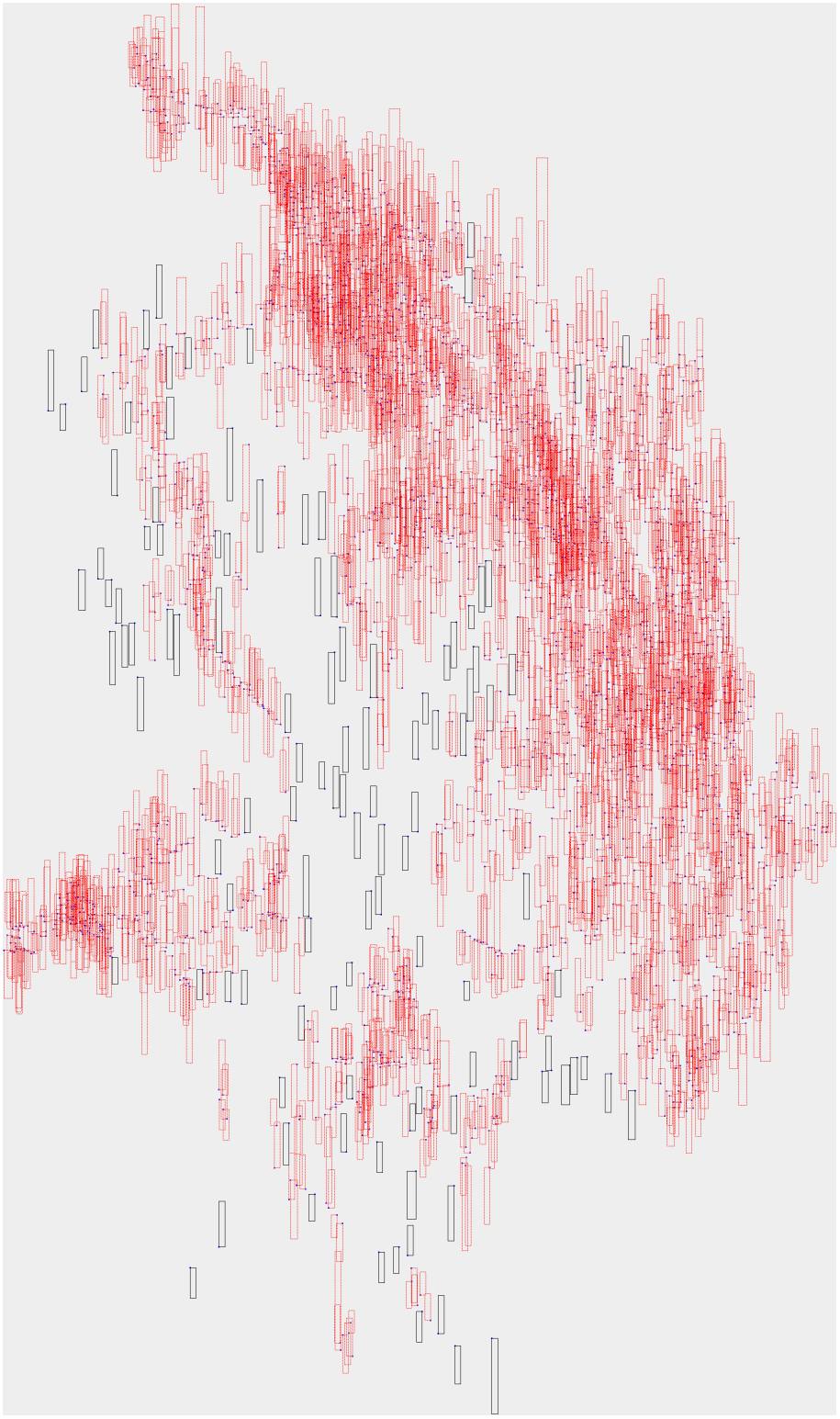


FIG. 10 – *Configuration initiale.*

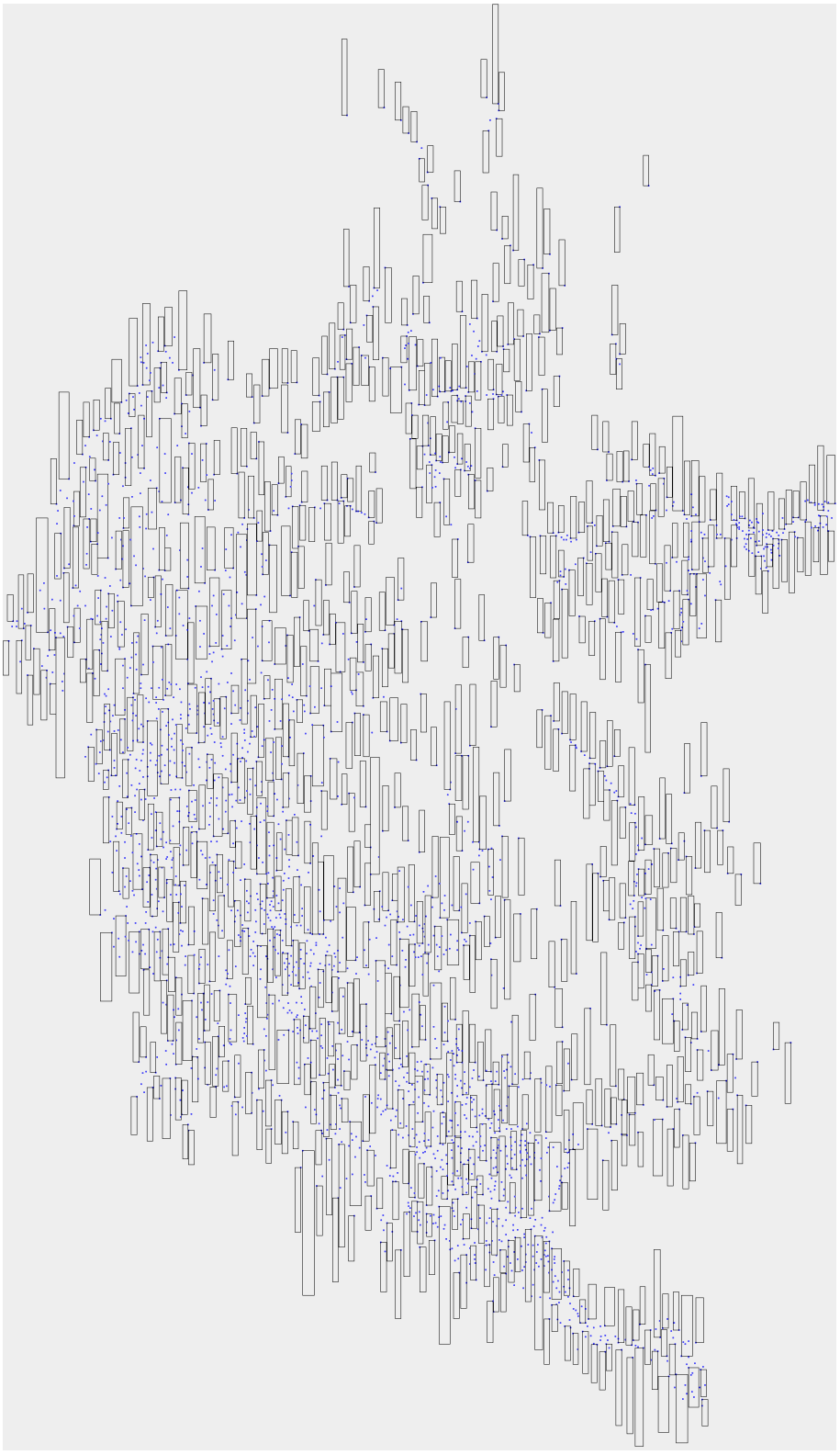


FIG. 11 – *Après optimisation.*

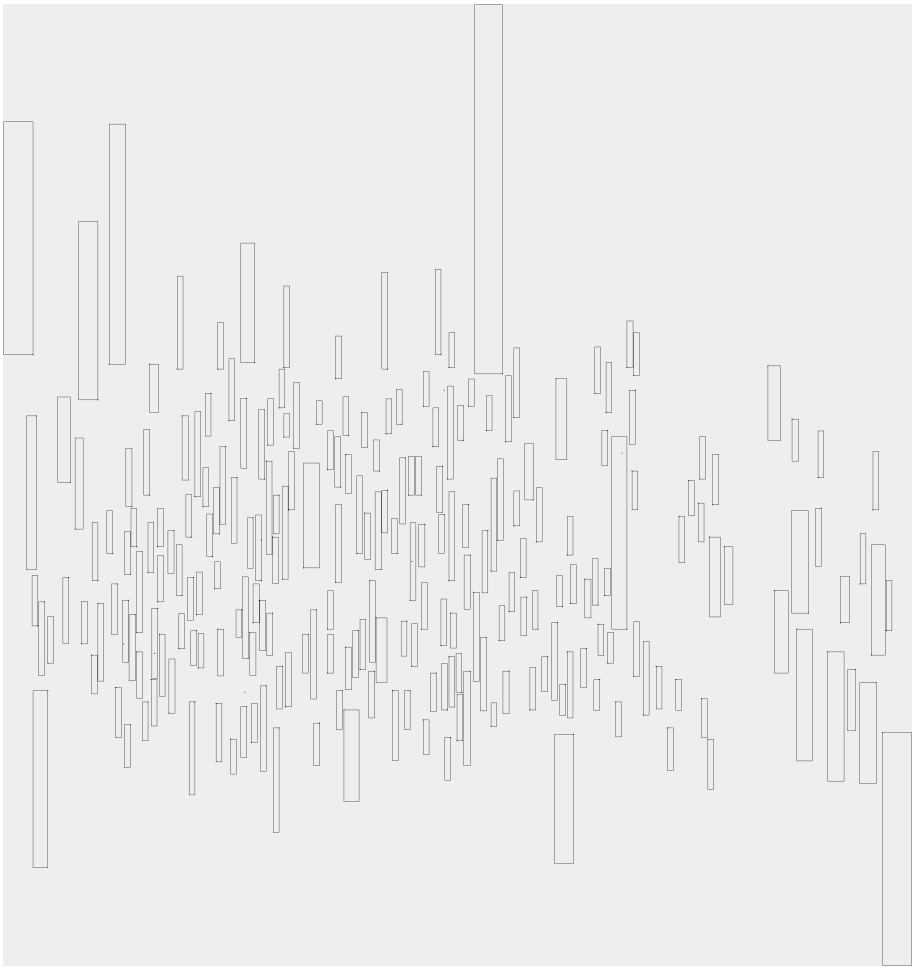


FIG. 12 – *Agrandissement d'une zone.*

8 Implémentation

Afin de comprendre comment ont été programmés les heuristiques, une présentation du programme est faite ici. Le langage de programmation est principalement le C++, les longues séries de calculs sont en général faites par un petit script en Bash ou en Ruby. L'annexe A fournit des informations au sujet de la compilation et des bibliothèques employées. Les diagrammes de classes se trouvent à l'annexe B. Tous fichiers du projet peuvent être téléchargés à cette adresse : `http://www.euphorik.ch/placement_labels`

8.1 Découpage du programme en espaces de noms

Le programme est découpé en plusieurs espaces de noms représentés par des répertoires.

- Base : c'est le programme de base qui comprend le point d'entrée ("main"). Il contient la représentation d'un problème et d'une solution générale, un générateur de problèmes, un lecteur de fichiers contenant des coordonnées géographiques.
- Base/tools : contient un outil de génération d'images.
- Base/Tabu : tout ce qui concerne l'heuristque de placement tabous. Deux classes peuvent être instanciées, une pour résoudre le problème à partir de coordonnées de points et l'autre à partir d'un fichier contenant les chevauchements potentiels.
- Base/Tabu/POPMUSIC : contient les algorithmes propres à POPMUSIC, cet espace est contenu dans "Tabu" car POPMUSIC l'utilise.

8.2 Détails des structures de données

Voici une présentation de la façon dont certaines structures de données sont réalisées.

8.2.1 Gestionnaire de chevauchements

Cette structure représente tous les chevauchements possibles entre étiquettes, elle peut être interrogée pour, par exemple, connaître combien de conflits créent une étiquette, le coût d'une étiquette mise dans une certaine position ou le nombre d'étiquettes en chevauchant au moins une autre. Les possibilités de chevauchements sont représentées en figure 13.

Pour donner un exemple concrêt par rapport à la figure 13 on peut dire que l'étiquette n°4 dans la position 0 entre en conflit avec l'étiquette n°2 dans la position 0 et 2.

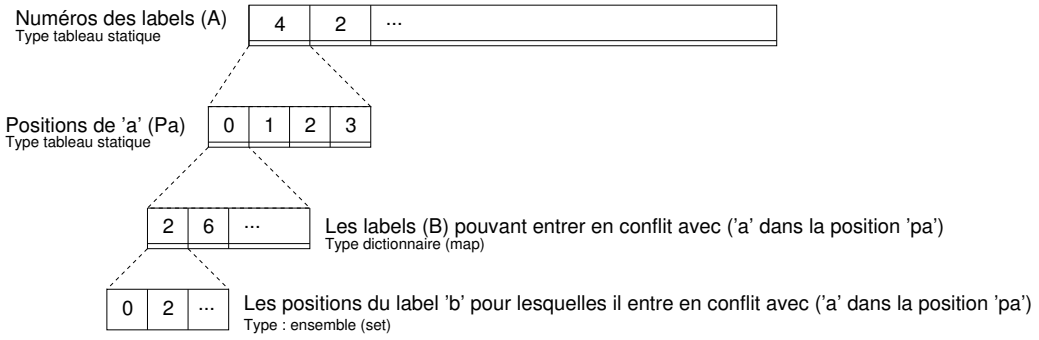


FIG. 13 – *Structure des chevauchements potentiels.*

Cette structure est dérivée pour l'algorithme POPMUSIC afin d'ajouter des fonctions permettant de ne montrer qu'un sous-ensemble. La recherche avec tabous est ensuite appliquée à ce sous-ensemble.

8.2.2 Liste des coûts

Cette structure de données est utilisée dans la recherche avec tabous, c'est la liste triée des coûts de chaque label, la seule opération nécessaire est la mise à jour d'un coût, ceci doit engendrer une réorganisation de la liste afin de la maintenir triée. À l'aide de cette structure triée un simple pointeur permet de définir la liste des candidats comme montré en figure 14.

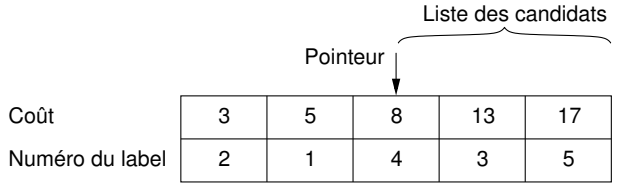


FIG. 14 – *La liste des candidats définie sur la liste des coûts.*

9 Conclusion

Dans un premier temps la recherche avec tabous a permis d'optimiser de petits problèmes très rapidement, mais avec l'accroissement du nombre de labels la qualité des solutions produites baisse très rapidement et l'algorithme montre ses limites en ayant une convergence très lente.

Cette difficulté a pût être résolue à l'aide la méta-heuristique POPMUSIC[4] qui, une fois utilisée comme une couche au dessus de la recherche avec tabous, a donnée de très bons résultats sur des problèmes conséquents. Une légère modification de POPMUSIC a été réalisée dans le but d'augmenter la vitesse d'exécution, la qualité de la solution s'en trouve alors quelque peu affectée mais reste tout à fait acceptable.

Une comparaison avec d'autres algorithmes de la littérature a montré la supériorité de POPMUSIC, ceci en terme de qualité des solutions produites mais aussi en terme de vitesse d'exécution.

Finalement, l'introduction de notion de pondération a autorisé une application de l'algorithme sur un cas réel d'étiquetage.

Références

[1] Missae Yamamoto, Gilberto Camara, and Luiz Antonio Nogueira Lorena. Tabu search heuristic for point-feature cartographic label placement. Technical report, Instituto Nacional de Pasquisas Espaciais, ? ? ? ?

[2] Missae Yamamoto and Luiz Antonio Nogueira Lorena. A Constructive Genetic Approach to Point-Feature Cartographic Label Placement. In *The Fifth Metaheuristics International Conference*, 2003.

[3] Éric D. Taillard. Comparison of non-deterministic iterative methods. Technical report, University of Applied Science of Western Switzerland : eivd campus at Yverdon, 2001.

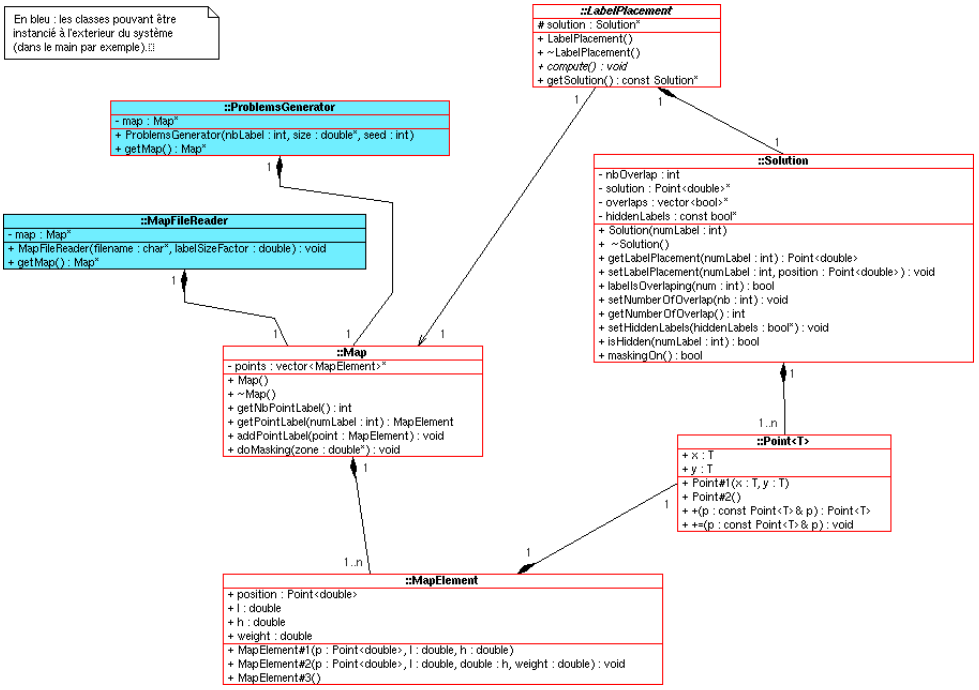
[4] Éric D. Taillard and Stefan Voss. Popmusic Partial Optimisation Metaheuristic Under Special Intensification Conditions. Technical report, University of Applied Science of Western Switzerland : eivd campus at Yverdon and Technische Universität Braunschweig Institut für Wirtschaftswissenschaften, Informationsmanagement, 1999.

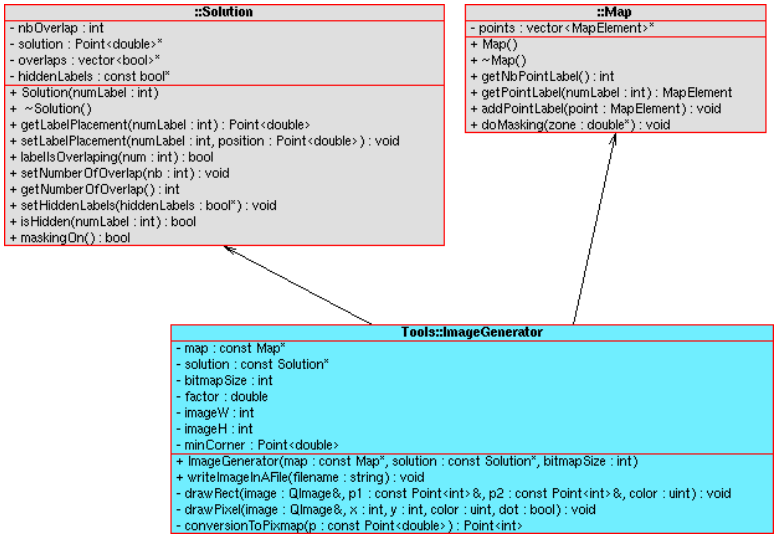
A Outils employés

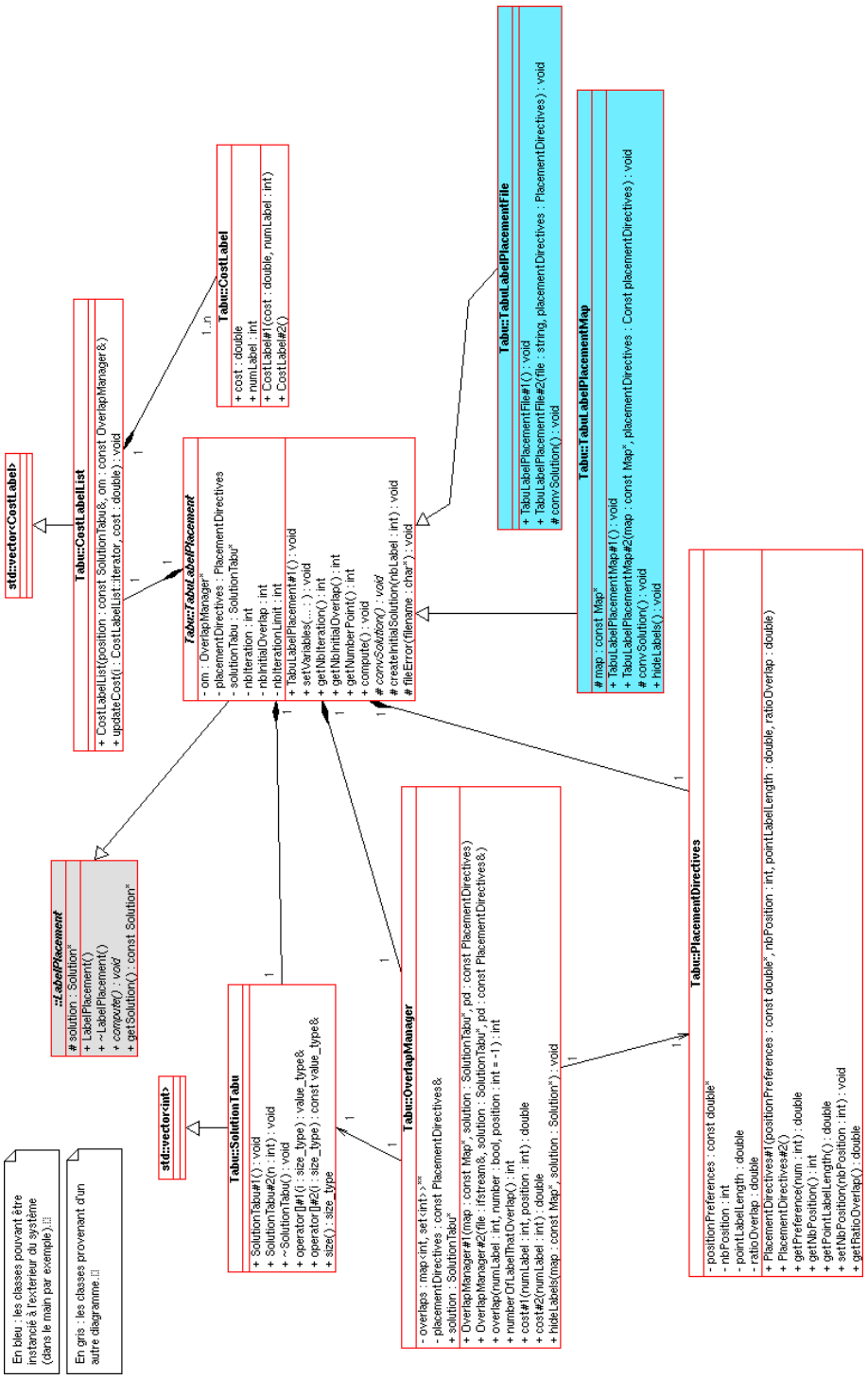
Voici une liste des outils employés, ils sont tous libres.

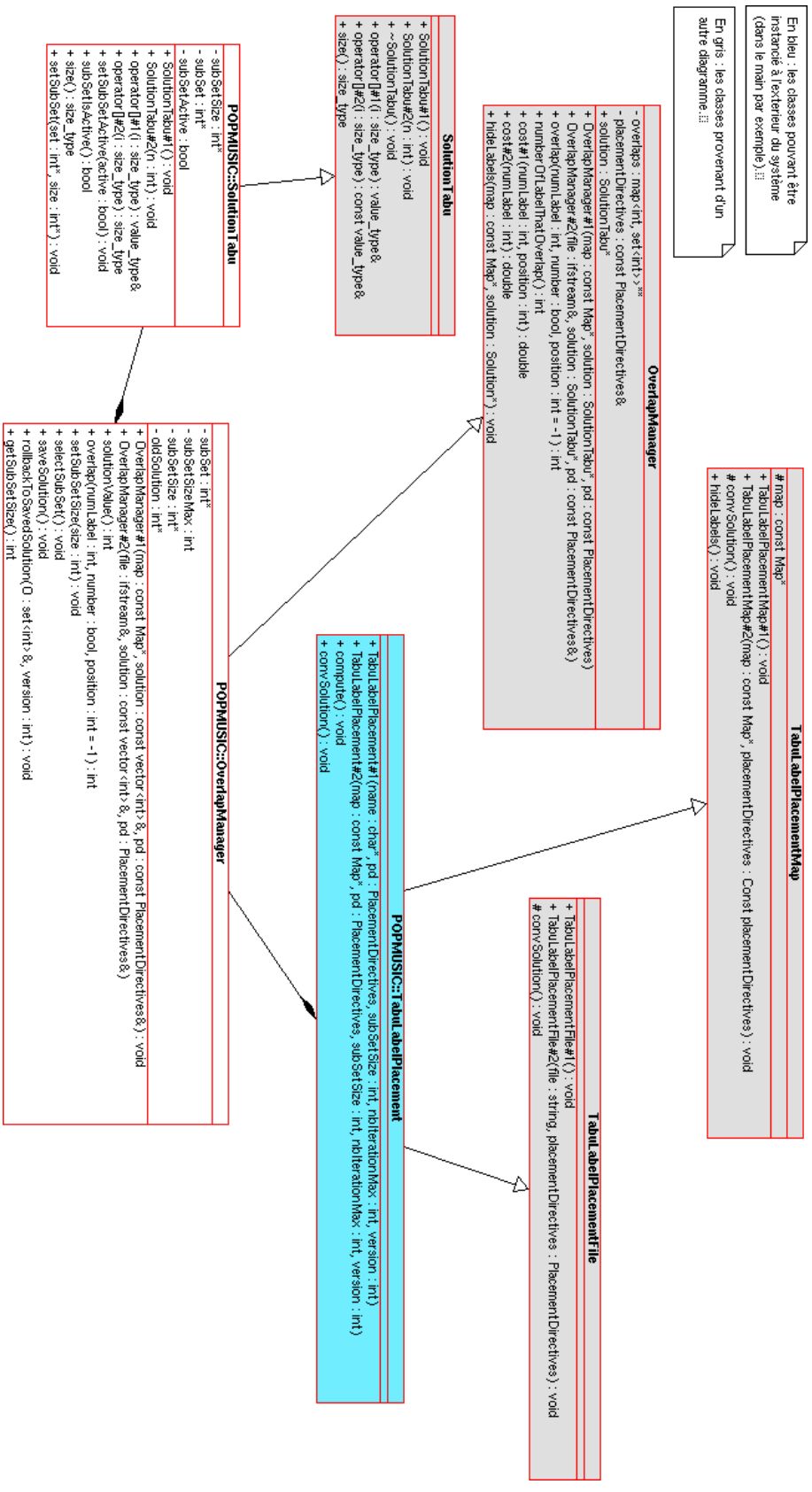
- Linux Mandrake 9.2 (développement) et Debian Woody testing (serveur de calculs) ;
- C/C++ et Ruby pour les langages de programmation ;
- La bibliothèque QT (X11) pour la génération d'image, elle est téléchageable sous licence GPL2 à cette adresse : `http://www.trolltech.com/download/qt/x11.html` ;
- QT designer pour l'environnement de développement ;
- GCC 3.3.1 pour le compilateur, le Makefile est généré automatiquement à partir du fichier de projet de QT designer ;
- Umbrello pour les diagrammes de classes ;
- Xfig pour les figures graphiques ;
- LaTeX pour la documentation.

B Diagrammes de classes






En bleu : les classes pouvant être instancié à l'extérieur du système (dans le main par exemple).
En gris : les classes provenant d'un autre diagramme.
std::vector<CostLabel>
Tabu::CostLabelList
+ CostLabelList(position : const SolutionTabu&, om : const OverlapManager&)
+ updateCost() : CostLabelList::iterator_cost : double) : void
Tabu::CostLabel
+ cost : double
+ numLabel : int
+ CostLabel#1(cost : double, numLabel : int)
+ CostLabel#2()
1..n
::LabelPlacement
solution : Solution*
+ LabelPlacement()
+ ~LabelPlacement()
+ compute() : void
+ getSolution() : const Solution*
Tabu::TabuLabelPlacement
- om : OverlapManager*
- placementDirectives : PlacementDirectives
- solutionTabu : SolutionTabu*
- nbIteration : int
- nbInitialOverlap : int
- nbIterationLimit : int
+ TabuLabelPlacement#1() : void
+ setVariables(...) : void
+ getNbIteration() : int
+ getNbInitialOverlap() : int
+ getNumberPoint() : int
+ compute() : void
# convSolution() : void
# createInitialSolution(nbLabel : int) : void
# fileError(filename : char*) : void
Tabu::TabuLabelPlacementFile
+ TabuLabelPlacementFile#1() : void
+ TabuLabelPlacementFile#2(file : string, placementDirectives : PlacementDirectives) : void
# convSolution() : void
Tabu::TabuLabelPlacementMap
map : const Map*
+ TabuLabelPlacementMap#1() : void
+ TabuLabelPlacementMap#2(map : const Map*, placementDirectives : Const placementDirectives) : void
# convSolution() : void
+ hideLabels() : void
std::vector<int>
Tabu::SolutionTabu
+ SolutionTabu#1() : void
+ SolutionTabu#2(n : int) : void
+ ~SolutionTabu() : void
+ operator[]#1(i : size_type) : value_type&
+ operator[]#2() : size_type) : const value_type&
+ size() : size_type
Tabu::OverlapManager
- overlaps : map<int, set<int> >*
- placementDirectives : const PlacementDirectives&
+ solution : SolutionTabu*
+ OverlapManager#1(map : const Map*, solution : SolutionTabu*, pd : const PlacementDirectives)
+ OverlapManager#2(file : ifstream&, solution : SolutionTabu*, pd : const PlacementDirectives&)
+ overlap(numLabel : int, number : bool, position : int = -1) : int
+ numberOfLabelThatOverlap() : int
+ cost#1(numLabel : int, position : int) : double
+ cost#2(numLabel : int) : double
+ hideLabels(map : const Map*, solution : Solution*) : void
Tabu::PlacementDirectives
- positionPreferences : const double*
- nbPosition : int
- pointLabelLength : double
- ratioOverlap : double
+ PlacementDirectives#1(positionPreferences : const double*, nbPosition : int, pointLabelLength : double, ratioOverlap : double)
+ PlacementDirectives#2()
+ getPreference(num : int) : double
+ getNbPosition() : int
+ getPointLabelLength() : double
+ setNbPosition(nbPosition : int) : void
+ getRatioOverlap() : double


En bleu : les classes pouvant être instanciés à l'extérieur du système (dans le main par exemple).
En gris : les classes provenant d'un autre diagramme.
TabuLabelPlacementMap
map : const Map*
+ TabuLabelPlacementMap#1() : void
+ TabuLabelPlacementMap#2(map : const Map*, placementDirectives : Const placementDirectives) : void
# convSolution() : void
+ hideLabels() : void
OverlapManager
- overlaps : map<int, set<int>>*
- placementDirectives : const PlacementDirectives&
+ solution : SolutionTabu*
+ OverlapManager#1(map : const Map*, solution : SolutionTabu*, pd : const PlacementDirectives)
+ OverlapManager#2(file : ifstream&, solution : SolutionTabu*, pd : const PlacementDirectives&)
+ overlap(numLabel : int, number : bool, position : int = -1) : int
+ numberOfLabelThatOverlap() : int
+ cost#1(numLabel : int, position : int) : double
+ cost#2(numLabel : int) : double
+ hideLabels(map : const Map*, solution : Solution*) : void
SolutionTabu
+ SolutionTabu#1() : void
+ SolutionTabu#2(n : int) : void
+ ~SolutionTabu() : void
+ operator[]#1(i : size_type) : value_type&
+ operator[]#2(i : size_type) : const value_type&
+ size() : size_type
POPMUSIC::SolutionTabu
- subSetSize : int*
- subSet : int*
- subSetActive : bool
+ SolutionTabu#1() : void
+ SolutionTabu#2(n : int) : void
+ operator[]#1(i : size_type) : value_type&
+ operator[]#2(i : size_type) : size_type
+ setSubSetActive(active : bool) : void
+ subSetIsActive() : bool
+ size() : size_type
+ setSubSet(set : int*, size : int*) : void
POPMUSIC::OverlapManager
- subSet : int*
- subSetSizeMax : int
- subSetSize : int*
- oldSolution : int*
+ OverlapManager#1(map : const Map*, solution : const vector<int>&, pd : const PlacementDirectives&) : void
+ OverlapManager#2(file : ifstream&, solution : const vector<int>&, pd : PlacementDirectives&)
+ solutionValue() : int
+ overlap(numLabel : int, number : bool, position : int = -1) : int
+ setSubSetSize(size : int) : void
+ selectSubSet() : void
+ saveSolution() : void
+ rollbackToSavedSolution(O : set<int>&, version : int) : void
+ getSubSetSize() : int
POPMUSIC::TabuLabelPlacement
+ TabuLabelPlacement#1(name : char*, pd : PlacementDirectives, subSetSize : int, nbIterationMax : int, version : int)
+ TabuLabelPlacement#2(map : const Map*, pd : PlacementDirectives, subSetSize : int, nbIterationMax : int, version : int)
+ compute() : void
+ convSolution() : void
TabuLabelPlacementFile
+ TabuLabelPlacementFile#1() : void
+ TabuLabelPlacementFile#2(file : string, placementDirectives : PlacementDirectives) : void
# convSolution() : void

C Code source

C.1 main.cpp

```
/**
  * Travail de diplôme 2003
  * Optimisation de placement de labels sur une carte géographique.
  * Fichier principal.
  *
  * Etudiant : Grégory Burri
  * Professeur : Eric Taillard
  * Date : 26.06.03
  * Compilé avec gcc 3.3.1, le makefile est généré
  * automatiquement à l'aide du programme 'qmake'
  * qui est fournit avec QT Designer.
  * QT designer ainsi que les bibliothèques QT de developpement
  * sont fournit gratuitement à cette adresse :
  * http://www.trolltech.com/download/qt/x11.html
  *
  * L'indentation utilisé et de type tabulaire,
  * une tabulation = 4 caractères espaces.
  * La longueur d'une ligne est de maximum 100 caractères pour
  * garantir une impression sans retour à la ligne intempestif.
  */

#include <iostream>
#include <string>
#include <ctime>
using namespace std;

//pour résoudre un problème avec l'heuristique "Tabou"
#include "tabu/TabuLabelPlacement.h"
#include "tabu/PlacementDirectives.h"
//metaheuristique POPMUSIC
#include "tabu/popmusic/PopTabuLabelPlacement.h"
using namespace Tabu;

//pour générer un résultat de type pixmap
#include "tools/ImageGenerator.h"
using namespace Tools;

//pour générer des données aléatoires
#include "problemsGenerator/ProblemsGenerator.h"

//pour lire des fichiers de données
#include "MapFileReader.h"

/**
  * Renvois un argument de la ligne de commande
  * si il n'existe pas alors lève une exception (pour éviter un "segmentation fault")
  * p1 : le numéro de l'argument à renvoyer
  * p2 : tous les arguments
  * p3 : le nombre d'argument total
  */
char* getArg(int numArg, char** argv, int argc)
{
    if(numArg >= argc) throw exception();
    return argv[numArg];
}

/**
  * Affiche l'utilisation du programme.
  * p1 : le nom de programme
  */
void printHelp(char* progName)
{
    cout << "Cartographics label placement with tabu search"
        " heuristic and POPMUSIC metaheuristic" << endl;
    cout << "Usage : " << progName << " [OPTIONS]..." << endl <<
    "Options are :" << endl <<
    "  -g <1 for enable GUI>" << endl <<
    "  [ -fo <file (type = potential overlap)> | -fp <file (type = liste of points)>]" << endl <<
    "  -c <factor label size> (use only with -fp option)" << endl <<
    "  -z <left> <top> <right> <bottom>" << endl <<
    "  -w <1 for enable hidding label with low weight and overlaping>"
        " (use only with -fp option)" << endl <<
    "  -p <1 for enable POPMUSIC euristic version 1, 2 for version 2>" << endl <<
    "  -u <size of subset (for POPMUSIC only)" << endl <<
    "  -ip <number of max iteration for POPMUSIC>" << endl <<
    "  -d <POPMUSIC seed (-1 for time)>" << endl <<
    "  -s <size X> <size Y>" << endl <<
    "  -i <number of iteration max>" << endl <<
```

```
    "  -ti <set how calcul the total compute time  "
80          " 1: initialisation time + algorithm time  "
            " 0: algorithm time  "
            " 2: initialisation time>" << endl <<
    "  -n <number of label>" << endl <<
    "  -r <weight overlap over position (ratio)>" << endl <<
85  "  -v <initial candidate list factor size><candidate list reduce factor> " <<
    "<candidate list growing factor><tabu list factor size> " <<
    "<number of iteration before recompute size of lists><minimal candidate list size>" <<
    "<minimal tabu list size>" << endl <<
    "  -e <random point placement seed (-1 for time)>" << endl <<
90  "  -l <label to point length>" << endl <<
    "  -m <image size>" << endl <<
    "  -a <image filename>" << endl;
}

95 //#################//
int main(int argc, char** argv)
{
    /*~~~~~~~~~~~DONNÉES (avec valeurs par défaut)~~~~~~~~~~~~*/
    double size[2] = {30, 30}; //la taille de la surface
100 int nbLabel = 100; //le nombre de label à placer
    /* statiques (pour l'instant) */
        int nbPosition = 4; //le nombre de position d'un label par rapport au point
        double preferences[] = { 0.0, 0.4, 0.6, 0.9 };
    /* fin statique */
105 int nbIteration = 1000; //le nombre d'itération
    double ratioOverlapPosition = 1; //l'importance du chevauchement par rapport à la position
    int seed = -1; //la germe pour la génération aléatoire
    double labelToPointLength = 0.0; //la distance entre le label et son point
    int imageSize = 1000; //taille de l'image résultat
110 string filename = "tmp/result"; //le nom du fichier de sortie

    //la taille de la liste de candidat par rapport à au nombre de label se chevauchant
    double candidateSizeBase = 0.73;
    double candidateReduceFactor = 1.3;
115 double candidateGrowingFactor = 15;

    //la taille de la liste tabou par rapport à au nombre de label se chevauchant
    double tabuSize = 0.77;

120 //après combien d'itération il faut recalculer la taille des listes (tabou et candidats)
    int nbIterationListSizeRecompute = 47;

    //taille minimale de la liste de candidat
    int minimalCandidateListSize = 18;
125
    //taille minimale de la liste tabou
    int minimalTabuListSize = 9;

    //si depuis un fichier : le nom du fichier
130 char* dataFilename = 0;

    /* POPMUSIC */
    int sizeOfSubSet = 70; //le nombre de point contenu dans un sous ensemble
    int seedPOPMUSIC = -1;
135 int nbIterationMaxPOP = -1; //le nombre d'itération maximum

    /* fichier de points */
    double factorLabelSize = 3;
    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
140
    //données depuis un fichier de données ? (si oui de quel type)
    enum FileType {NONE=0, OVERLAP, LIST_OF_POINT};
    FileType withDataFilename = NONE;

145 int withPOPMUSIC = 0; //utiliser la metaheuristique POPMUSIC
    bool withImageFile = false; //génère les images png ?
    bool withHiddenLabel = false; //masque les labels de faible poids ?

    //doit-on restreindre le calcul sur un certaine zone ?
150 bool widthMaskingZone = false;
    //la zone à masquer : 0=bord gauche, 1=bord haut, 2=bord droite, 3=bord bas
    double maskingZone[4];

    enum TimeType {COMPUTE=0, ALL, INITIALISATION};
155 //par défaut on ne compte que le temps d'initialisation
    TimeType howComputeTime = COMPUTE;

    try
    {
160     //parcours les arguments fournis au programme
        for(int numArg=1; numArg < argc; numArg+=2)
```

```
{
    string arg = string(argv[numArg]);
    if(arg == "--help" || arg == "help")
    {
        printHelp(argv[0]);
        return 0;
    }
    else
        if(arg == "-f" || arg == "-fo"){
            dataFilename = getArg(numArg+1, argv, argc);
            withDataFilename = OVERLAP;
        }
        if(arg == "-fp"){
            dataFilename = getArg(numArg+1, argv, argc);
            withDataFilename = LIST_OF_POINT;
        }
        else if(arg == "-c"){
            factorLabelSize = atof(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-v"){
            candidateSizeBase = atof(getArg(numArg+1, argv, argc));
            numArg++;
            candidateReduceFactor = atof(getArg(numArg+1, argv, argc));
            numArg++;
            candidateGrowingFactor  = atof(getArg(numArg+1, argv, argc));
            numArg++;
            tabuSize = atof(getArg(numArg+1, argv, argc));
            numArg++;
            nbIterationListSizeRecompute = atoi(getArg(numArg+1, argv, argc));
            numArg++;
            minimalCandidateListSize = atoi(getArg(numArg+1, argv, argc));
            numArg++;
            minimalTabuListSize = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-w"){
            withHiddenLabel = string(getArg(numArg+1, argv, argc)) == "1";
        }
        else if(arg == "-p"){
            withPOPMUSIC = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-u"){
            sizeOfSubSet  = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-ip"){
            nbIterationMaxPOP = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-d"){
            seedPOPMUSIC = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-s"){
            size[0] = atof(getArg(numArg+1, argv, argc));
            numArg++;
            size[1] = atof(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-i"){
            nbIteration = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-ti"){
            switch(atoi(getArg(numArg+1, argv, argc)))
            {
                case 0 : howComputeTime = COMPUTE; break;
                case 1 : howComputeTime =  ALL; break;
                case 2 : howComputeTime =  INITIALISATION; break;
            }
        }
        else if(arg == "-n"){
            nbLabel = atoi(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-r"){
            ratioOverlapPosition = atof(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-z"){
            widthMaskingZone = true;
            maskingZone[0] = atof(getArg(numArg+1, argv, argc));
            numArg++;
            maskingZone[1] = atof(getArg(numArg+1, argv, argc));
            numArg++;
            maskingZone[2] = atof(getArg(numArg+1, argv, argc));
            numArg++;
            maskingZone[3] = atof(getArg(numArg+1, argv, argc));
        }
        else if(arg == "-e"){
            seed = atoi(getArg(numArg+1, argv, argc));
```

```
245                 }
                    else if(arg == "-l"){
                        labelToPointLength  = atof(getArg(numArg+1, argv, argc));
                    }
                    else if(arg == "-m"){
250                     imageSize = atoi(getArg(numArg+1, argv, argc));
                    }
                    else if(arg == "-a"){
                        filename = string(getArg(numArg+1, argv, argc));
                        withImageFile = true;
255                 }
            }
        }
        catch(exception)
        {
260         cerr << "Arguments error" << endl;
            printHelp(argv[0]);
            exit(1);
        }

265     //définit les préférences de placement
        //(voir la classe "PlacementDirectives" pour plus de détail)
        PlacementDirectives placementDirectives(preferences,
            nbPosition, labelToPointLength, ratioOverlapPosition);

270     //la map sous forme de point
        Map* map = 0;

        if(withDataFilename == NONE)
            map = ProblemsGenerator(nbLabel, size, seed).getMap();
275     else if(withDataFilename == LIST_OF_POINT)
            map = MapFileReader(dataFilename, factorLabelSize).getMap();

        //on enlève les point hors de la zone définit
        if(map != 0 && widthMaskingZone)
280         map->doMasking(maskingZone);

        TabuLabelPlacement* tabu;

        //pour mesurer le temps de calcul de l'initialisation
285     clock_t timeInit = clock();

        //données depuis un fichier ou les chevauchement potentiels sont déjà indiqués
        if(withDataFilename == OVERLAP)
        {
290         if(withPOPMUSIC)
                tabu = new POPMUSIC::TabuLabelPlacement(dataFilename, placementDirectives,
                    sizeOfSubSet, seedPOPMUSIC, nbIterationMaxPOP, withPOPMUSIC);
            else
                tabu = new TabuLabelPlacementFile(dataFilename, placementDirectives);
295     }
        else
        {
            if(withPOPMUSIC)
                tabu = new POPMUSIC::TabuLabelPlacement(map, placementDirectives,
300                 sizeOfSubSet, seedPOPMUSIC, nbIterationMaxPOP, withPOPMUSIC);
            else
                tabu = new TabuLabelPlacementMap(map, placementDirectives);
        }

305     //définit les paramètres de l'algorithme
        tabu->setVariables(nbIteration, candidateSizeBase,
                candidateReduceFactor, candidateGrowingFactor,
                tabuSize, nbIterationListSizeRecompute, minimalCandidateListSize,
                minimalTabuListSize);
310
        timeInit = (clock() - timeInit)/1000;

        if(withImageFile && (withDataFilename == NONE || withDataFilename == LIST_OF_POINT))
        {
315         //génère l'image de la solution initiale
            ImageGenerator pixmap_initial(map, tabu->getSolution(), imageSize);
            pixmap_initial.writeImageInAFile(filename + "_initial.png");
        }

320     clock_t timeCompute = clock();

        //réalise la calcule tabou
        tabu->compute();

325     //si on veux masquer les labels de poinds faible
        if(withHiddenLabel && withDataFilename == LIST_OF_POINT)
            dynamic_cast<Tabu::TabuLabelPlacementMap*>(tabu)->hideLabels();
```

```
    timeCompute = (clock() - timeCompute)/1000;

    clock_t time = clock();
    //en fonction du temp demandé
    switch(howComputeTime)
    {
    case ALL :
        time = timeInit + timeCompute;
        break;

    case COMPUTE :
        time = timeCompute;
        break;

    case INITIALISATION :
        time = timeInit;
        break;
    }

    if(withImageFile && (withDataFilename == NONE || withDataFilename == LIST_OF_POINT))
    {
        //génère l'image de la solution finale
        ImageGenerator pixmap_final(map, tabu->getSolution(), imageSize);
        pixmap_final.writeImageInAFile(filename + "_final.png");
    }

    cout << "The variables are : " << endl <<
    "size : " << size[0] << " x " << size[1] << endl <<
    "Number of label : " << tabu->getNumberPoint() << endl <<
    "Number of iteration max : " << nbIteration << endl <<
    "Weight overlap over position : " << ratioOverlapPosition << endl <<
    "Seed for the randomizer : " << seed << endl <<
    "Length label to point : " << labelToPointLength << endl <<
    "Number of iteration : " << tabu->getNbIteration() << endl;

    if(withPOPMUSIC)
        cout << "Size of the subset : " << sizeOfSubSet << endl;

    cout << endl << "Number of initial overlaping : " << tabu->getNbInitialOverlap() << endl <<
    "Number of overlaping : " << tabu->getSolution()->getNumberOfOverlap() << endl;
    //si le "masquage" des labels est enclenché alors affiche le nombre de label visible
    if(withHiddenLabel)
    {
        int nb = 0;
        for(int i=0; i<tabu->getNumberPoint(); i++)
            if(!tabu->getSolution()->isHidden(i)) nb++;

        cout << "Number of unmasking label : " << nb << endl;
    }

    cout << "Time of computation : " << time << " ms" << endl <<
    "----------------------------" << endl;

    return 0;
}
```

C.2 LabelPlacement.h

```
/**
  * Classe abstraite représentant le calcul général de l'optimisation
  * de placement de labels.
  * On lui fournit un tableau de points à labeliser et pour chaque point
  * un rectangle de taille LxH représentant le texte à placer.
  */

#ifndef _LABELPLACEMENT_H_
#define _LABELPLACEMENT_H_

//la structure de données représentant les labels à placer
#include "Map.h"

//la structure de données repésentant la solution du placement
#include "Solution.h"

class LabelPlacement
{
public :

    /**
      * Les constructeurs.
      */
    LabelPlacement();
    virtual ~LabelPlacement();

    /**
      * Commence le calcul.
      */
    virtual void compute()=0;

    /**
      * Renvois la solution trouvée.
      */
    virtual const Solution* getSolution() const;

protected :

    Solution* solution; //la solution
};

#endif
```

C.3 LabelPlacement.cpp

```
#include "LabelPlacement.h"

//#################//
LabelPlacement::LabelPlacement(){}

//#################//
LabelPlacement::~LabelPlacement()
{
    delete this->solution;
}

//#################//
const Solution* LabelPlacement::getSolution() const
{
    return this->solution;
}
```

C.4 Point.h

```
/**
  * Représente un point dans un espace
  * à 2 dimensions.
  */

#ifndef _POINT_H_
#define _POINT_H_

template<class T>
struct Point
{
    /**
      * Le constructeur.
      * p1 : la valeur en x
      * p2 : la valeur en y
      */
    Point(T, T);
    Point();

    /**
      * Addition de 2 points.
      * p1 : le point
      */
    Point<T> operator+(const Point<T>&) const;
    void operator+=(const Point<T>&);

    T x;
    T y;
};

#endif
```

C.5 Point.cpp

```
#ifndef _POINT_CPP_
#define _POINT_CPP_ 1

#include "Point.h"

//#################//
template<class T>
Point<T>::Point(T x, T y)
{
    this->x = x;
    this->y = y;
}

//#################//
template<class T>
Point<T>::Point()
{
    this->x = 0;
    this->y = 0;
}

//#################//
template<class T>
Point<T> Point<T>::operator+(const Point<T>& p) const
{
    return Point<T>(this.x + p2.x, this.y + p1.y);
}

//#################//
template<class T>
void Point<T>::operator+=(const Point<T>& p)
{
    this->x += p.x;
    this->y += p.y;
}

#endif
```

C.6 Solution.h

```
/**
  * Classe représentant la solution après l'optimisation du placement.
  */

#ifndef _SOLUTION_H_
#define _SOLUTION_H_

#include <vector>
#include <iostream>
using namespace std;

#include "Point.cpp"

class Solution
{
public :

    /**
      * Constructeur.
      * p1 : le nombre de label
      */
    Solution(int);

    ~Solution(); //destructeur

    /**
      * Renvois le placement relatif d'un label.
      * p1 : Le numéro du label
      */
    Point<double> getLabelPlacement(int) const;

    /**
      * Assigne l'emplacement d'un label.
      * p1 : le numéro du label
      * p2 : l'emplacement
      * p3 : entre-t-il en collision ?
      */
    void setLabelPlacement(int, Point<double>, bool);

    /**
      * Test si un label entre en collision.
      * p1 : le numéro du label
      */
    bool labelIsOverlaping(int num) const
    {
        return (*this->overlaps)[num];
    }

    /**
      * Définit le nombre de chevauchement
      * p1 : le nombre de chevauchement.
      */
    void setNumberOfOverlap(int nb)
    {
        this->nbOverlap = nb;
    }

    /**
      * Renvois le nombre de chevauchement.
      */
    int getNumberOfOverlap() const
    {
        return this->nbOverlap;
    }

    /**
      * Définit les labels qui sont cachés.
      * p1 : les labels cachés
      */
    void setHiddenLabels(const bool* hiddenLabels)
    {
        this->hiddenLabels = hiddenLabels;
    }

    /**
      * Savoir si un label est masqué.
      * p1 : le numéro du label
      */
    bool isHidden(int numLabel) const
    {
        if(this->hiddenLabels == 0) return false;
```

```
            return this->hiddenLabels[numLabel];
        }

        /**
          * Savoir si le masquage est activé
          */
        bool maskingOn() const
        {
            return this->hiddenLabels != 0;
        }

    private :

        //le nombre de chevauchement
        int nbOverlap;

        //la position relative de chaque label
        Point<double>* solution;

        //les chevauchement des label
        vector<bool>* overlaps;

        //les labels cachés
        const bool* hiddenLabels;
    };

    #endif
```

C.7 Solution.cpp

```
#include "Solution.h"

//#################//
Solution::Solution(int nbLabel)
{
    this->solution = new Point<double>[nbLabel];
    this->overlaps = new vector<bool>(nbLabel);
    this->hiddenLabels = 0;
}

//#################//
Solution::~Solution()
{
    delete[] this->solution;
    delete this->overlaps;
}

//#################//
Point<double> Solution::getLabelPlacement(int numLabel) const
{
    return this->solution[numLabel];
}

//#################//
void Solution::setLabelPlacement(int numLabel, Point<double> position, bool overlap)
{
    this->solution[numLabel] = position;
    (*this->overlaps)[numLabel] = overlap;
}
```

C.8 Map.h

```
/**
  * Classe représentant une ensemble de point ayant chacun
  * un label le définissant.
  * C'est la donnée à optimiser.
  */

#ifndef _MAP_H_
#define _MAP_H_


#include <vector>
using namespace std;

#include "Point.cpp"

/**
  * Un element de la map
  */
struct MapElement
{
    /**
        * Constructeur.
      * p1 : la position du point
          * p2 : la largeur du label
      * p3 : la hauteur du label
          */
    MapElement(Point<double>, double, double);
    // p4 : le poids du label
    MapElement(Point<double>, double, double, double);
    MapElement();

    Point<double> position;
    double l; //largeur
    double h; //hauteur
    double weight; //le poids
};

class Map
{
public :

    /**
      * Constructeur et destructeur.
      */
    Map()
    {
        this->points = new vector<MapElement>();
    }
    ~Map()
    {
        delete this->points;
    }

    /**
      * Renvois le nombre de label
      */
    int getNbPointLabel() const;

    /**
      * Renvois un point.
      * p1 : le numéro du label
      */
    MapElement getPointLabel(int) const;

    /**
      * Assigne un point et son label
      * p1 : le point et son label
      */
    void addPointLabel(MapElement);

    /**
      * Masque une zone de point.
      * p1 : la zone (droit, haut, gauche, bas)
      */
    void doMasking(double*);

private :

    //tous les points ainsi que leur label respectif
    vector<MapElement>* points;
};
```

```
#endif
```

C.9 Map.cpp

```
#include "Map.h"

//#################//
MapElement::MapElement(Point<double> p, double l, double h)
{
    this->position = p;
    this->l = l;
    this->h = h;
    this->weight = 0;
}

//#################//
MapElement::MapElement(Point<double> p, double l, double h, double w)
{
    this->position = p;
    this->l = l;
    this->h = h;
    this->weight = w;
}

//#################//
MapElement::MapElement()
{
    this->position = Point<double>(0, 0);
    this->l = 0;
    this->h = 0;
    this->weight = 0;
}

//#################//
int Map::getNbPointLabel() const
{
    return int(this->points->size());
}

//#################//
MapElement Map::getPointLabel(int numLabel) const
{
    return (*this->points)[numLabel];
}

//#################//
void Map::addPointLabel(MapElement point)
{
    this->points->push_back(point);
}

#include <iostream>
using namespace std;

//#################//
void Map::doMasking(double* z)
{
    //parcours les points et enlève ceux qui ne sont pas dans la zone

    vector<MapElement>* tmp = new vector<MapElement>();

    for(vector<MapElement>::iterator i = this->points->begin(); i != this->points->end(); i++)
        if(i->position.x > z[0] && i->position.x < z[2] &&
            i->position.y < z[1] && i->position.y > z[3])
                tmp->push_back(*i);

    vector<MapElement>* tmp2 = this->points;
    this->points = tmp;
    delete tmp2;
}
```

C.10 MapFileReader.h

```
/**
  * Permet de lire un fichier .map contenant une liste de point.
  * Chaque point est composé de coordonnés (x,y), d'un label et d'un poids.
  */

#ifndef _MAP_FILE_READER_H_
#define _MAP_FILE_READER_H_

#include <cstdlib>
#include <fstream>
using namespace std;

#include "Map.h"

class MapFileReader
{
public :
    /**
      * Constructeur.
      * p1 : le nom du fichier de données
      * p2 : le facteur définissant la taille des étiquettes
      */
    MapFileReader(char*, double);

    /**
      * Renvois le fichier sous la forme d'un Map*
      */
    Map* getMap();

private :
    Map* map;
};

#endif
```

C.11 MapFileReader.cpp

```
#include "MapFileReader.h"

#include <iostream>
#include <string>
#include <cmath>
using namespace std;

//#################//
MapFileReader::MapFileReader(char* name, double labelSizeFactor)
{
    ifstream file(name);
    if(!file)
    {
        cerr << "Error : the file " << name << " doesn't exist" << endl;
        exit(1);
    }

    int nbPoint; //le nombre de point

    file >> nbPoint;

    this->map = new Map();

    for(int i=0; i<nbPoint; i++)
    {
        int x, y;
        double l, h, weight;
        char cityName[255];

        file >> x;
        file >> y;
        file >> weight;

        file.get(cityName, 255);

        //n'ajoute pas les villes qui n'ont pas de poids
        if(weight == 0.0) continue;

        //1.5 represente le ratio entre la hauteur d'un caractère et sa largeur
        h = labelSizeFactor * 1.5;
        l = double(strlen(cityName)) * labelSizeFactor;

        //********************** TEMPORAIRE **********************//
        //facteur d'agrandissement en fonction du poids
        double f = log((weight < 1 ? 1 : weight) / 10000.0) + 3.0;
        if(f > 4.0) f = 4.0;
        if(f < 2) f = 2;
        l *= f;
        h *= f;
        //********************** FIN TEMPORAIRE **********************//

        this->map->addPointLabel(MapElement(Point<double>(x, y), l, h, weight));
    }
}

//#################//
Map* MapFileReader::getMap()
{
    return this->map;
}
```

C.12 ProblemGenerator.h

```
/**
  * Generateur aléatoire de problèmes de placement
  * de labels.
  */

#ifndef _PROBLEMSGENERATOR_H_
#define _PROBLEMSGENERATOR_H_

//le type d'objet représentant le problème à générer
#include "../Map.h"

class ProblemsGenerator
{
public :

    /**
      * Constructeur.
      * p1 : le nombre de label
      * p2 : la taille de la surface de placement (un tableau x-y)
      * p3 : le germe, si égal à -1 alors germe en fonction du temps
      */
    ProblemsGenerator(int, double*, int);

    /**
      * Renvois la map généré.
      */
    Map* getMap() const;

private :

    Map* map; //les données générée
};

#endif
```

C.13 ProblemGenerator.cpp

```
#include "ProblemsGenerator.h"

#include <cstdlib>
#include <ctime>
using namespace std;

#include "../Point.cpp"

//#################//
ProblemsGenerator::ProblemsGenerator(int nbLabel, double* size, int seed)
{
    this->map = new Map();

    srand(seed == -1 ? time(0) : seed);

    //place chaque label à une position aléatoire uniformement réapartie
    //sur la surface
    for(int i=0; i<nbLabel; i++)
    {
        double labelWidth = (double(rand())/RAND_MAX) * 4 + 1;
        double labelHeight = 0.5;

        Point<double> p((double(rand())/RAND_MAX) * size[0], (double(rand())/RAND_MAX) * size[1]);
        this->map->addPointLabel(MapElement(p,labelWidth ,labelHeight));
    }
}

//#################//
Map* ProblemsGenerator::getMap() const
{
    return this->map;
}
```

C.14 Tools : :ImageGenerator.h

```
/**
  * Permet de générer une image pixmap d'une solution
  */

#ifndef _TOOLS_IMAGEGENERATOR_H_
#define _TOOLS_IMAGEGENERATOR_H_

#include <string>
using namespace std;

#include <qimage.h>

//représente un solution complète
#include "../Solution.h"
#include "../Map.h"

namespace Tools
{
    class ImageGenerator
    {
    public :

        /**
          * Constructeur.
          * p1 : l'emplacement des labels
          * p2 : la solution à générer
          * p3 : la taille de l'image de sortie
          */
        ImageGenerator(const Map*, const Solution*, int);

        /**
          * Ecrit l'image dans un fichier.
          * p1 : le nom du fichier
          */
        void writeImageInAFile(string);

    private :

        /**
          * Dessine un rectangle sur un "QImage".
          * p1 : l'image
          * p2 : le coin inférieur gauche du rectangle
          * p3 : le coins supérieur droite du rectangle
          * p4 : la couleur
          * p5 : en trait-tillé (ortho) ? (pour afficher les labels qui entre en collision)
          */
        void drawRect(QImage&, const Point<int>&, const Point<int>&, uint, bool) const;

        /**
          * Dessine une pixel sur un "QImage" en vérifiant sa validité.
          * p1 : l'image
          * p2 : le coin inférieur gauche du rectangle
          * p3 : le coins supérieur droite du rectangle
          * p4 : la couleur
          */
        void drawPixel(QImage&, int, int, uint) const;

        /**
          * Convertit une coordonnée sur le "map" vers une coordonnée sur l'image bitmap.
          * p1 : Le point à convertir
          */
        Point<int> conversionToPixmap(const Point<double>) const;

        const Map* map; //les labels

        const Solution* solution; //la solution

        int bitmapSize; //la taille de l'image (le coté le plus grand)
        int nbPixelBorder; //le nombre de pixel de marge autour des labels
        double factor; //le facteur entre le map et l'image bitmap
        int imageW, imageH; //la taille de l'image
        Point<double> minCorner;//la coodonnée minimum
    };
}

#endif
```

C.15 Tools : :ImageGenerator.cpp

```
#include "ImageGenerator.h"
using namespace Tools;

#include <iostream>
using namespace std;

#include <qcolor.h>
#include <qstring.h>

#include "../Point.cpp"

#define ns ImageGenerator

//#################//
ns::ImageGenerator(const Map* map, const Solution* solution, int bitmapSize)
:map(map), solution(solution)
{
    this->bitmapSize = bitmapSize;
    this->nbPixelBorder = 2;

    //définit la taille de la zone de placement
    Point<double> pMax(-99999, -99999);
    Point<double> pMin(99999, 99999);

    //parcours tous les points afin de cherche les extremes pour défiinir la taille de la zone
    for(int i=0; i<this->map->getNbPointLabel(); i++)
    {
        double xMax = this->map->getPointLabel(i).position.x +
            (this->map->getPointLabel(i).l + this->solution->getLabelPlacement(i).x > 0 ?
            this->map->getPointLabel(i).l + this->solution->getLabelPlacement(i).x : 0);
        if(xMax > pMax.x) pMax.x = xMax;

        double yMax = this->map->getPointLabel(i).position.y +
            (this->map->getPointLabel(i).h + this->solution->getLabelPlacement(i).y > 0 ?
            this->map->getPointLabel(i).h + this->solution->getLabelPlacement(i).y : 0);
        if(yMax > pMax.y) pMax.y = yMax;

        double xMin =  this->map->getPointLabel(i).position.x +
            (this->solution->getLabelPlacement(i).x < 0 ?
            this->solution->getLabelPlacement(i).x : 0);
        if(xMin < pMin.x) pMin.x = xMin;

        double yMin =  this->map->getPointLabel(i).position.y +
            (this->solution->getLabelPlacement(i).y < 0 ?
            this->solution->getLabelPlacement(i).y : 0);
        if(yMin < pMin.y) pMin.y = yMin;
    }

    this->minCorner = pMin;

    //hauteur et largeur de la surface de placement
    double width = (pMax.x - pMin.x);
    double height = (pMax.y - pMin.y);

    //calcul le taille de l'image finale en fonction de la taille demandé
    this->imageW = int(width > height ? this->bitmapSize - (2*this->nbPixelBorder) :
        double(this->bitmapSize - (2*this->nbPixelBorder)) * width  / height);
    this->imageH = int(height > width ? this->bitmapSize - (2*this->nbPixelBorder) :
        double(this->bitmapSize - (2*this->nbPixelBorder)) * height  / width);

    //facteur entre l'image bitmap et la "map"
    this->factor = double(this->imageW) / width;

    this->imageW += (2*this->nbPixelBorder);
    this->imageH += (2*this->nbPixelBorder);
}

//#################//
void ns::writeImageInAFile(string filename)
{
    //crée l'image
    QImage image(this->imageW, this->imageH, 32);

    //les couleurs des composants de l'image
    uint backgroundColor = 0xEEEEEE;
    uint labelColor = 0x000000;
    uint labelColorOverlaping = 0xFF0000;
    uint pointColor = 0x0000FF;

    image.fill(backgroundColor); //remplit le fond
```

```
    //dessine chaque label
    for(int i=0; i<this->map->getNbPointLabel(); i++)
    {
        //le point
        Point<int> pos = this->conversionToPixmap(this->map->getPointLabel(i).position);

        //dessine le point
        this->drawRect(image, Point<int>(pos.x-1, pos.y-1),
            Point<int>(pos.x+1, pos.y+1), pointColor, false);

        if(!this->solution->isHidden(i))
        {
            //dessine le label
            this->drawRect(image,
                this->conversionToPixmap(Point<double>
                    (this->map->getPointLabel(i).position.x +
                    this->solution->getLabelPlacement(i).x,
                    this->map->getPointLabel(i).position.y +
                    this->solution->getLabelPlacement(i).y)),
                this->conversionToPixmap(Point<double>
                    (this->map->getPointLabel(i).position.x +
                    this->solution->getLabelPlacement(i).x +
                    this->map->getPointLabel(i).l,
                    this->map->getPointLabel(i).position.y +
                    this->solution->getLabelPlacement(i).y +
                    this->map->getPointLabel(i).h)),
                this->solution->labelIsOverlaping(i) &&
                    !this->solution->maskingOn() ? labelColorOverlaping : labelColor,
                this->solution->labelIsOverlaping(i) &&
                    !this->solution->maskingOn());
        }
    }

    //écrit l'image
    image.save(QString(filename), "PNG");

    cout << "Image file \"" << filename << "\" have been written" << endl;
}

//#################//
void ns::drawRect(QImage& image, const Point<int>& p1,
    const Point<int>& p2, uint color, bool dot) const
{
    //les bords horizontaux
    for(int i=(p1.x<p2.x?p1.x:p2.x)+1; i<(p1.x<p2.x?p2.x:p1.x); i++)
    {
        if(dot && int(i+p1.x/2) % 3 == 0) continue;
        this->drawPixel(image, i, p2.y, color);
        this->drawPixel(image, i, p1.y, color);
    }

    //les bords verticaux
    for(int i=(p1.y<p2.y?p1.y:p2.y); i<=(p1.y<p2.y?p2.y:p1.y); i++)
    {
        if(dot && int(i+p1.y/2) % 3 == 0) continue;
        this->drawPixel(image, p1.x, i, color);
        this->drawPixel(image, p2.x, i, color);
    }
}

//#################//
void ns::drawPixel(QImage& image, int x, int y, uint color) const
{
    if(x >= 0 && x < image.width() && y >= 0 && y < image.height())
        image.setPixel(x, y, color);
}

//#################//
Point<int> ns::conversionToPixmap(const Point<double> p) const
{
    return Point<int>(int((p.x - this->minCorner.x) * this->factor) + this->nbPixelBorder,
        int(-(p.y - this->minCorner.y) * this->factor + this->imageH-1) - this->nbPixelBorder);
}
```

C.16 Tabu : :TabuLabelPlacement.h

```
/**
  * Classes effectuant l'optimisation de placement de label
  * à l'aide de l'heuristique 'Tabou'.
  */

#ifndef _TABU_TABULABELPLACEMENT_H_
#define _TABU_TABULABELPLACEMENT_H_

//les directives de placements
#include "PlacementDirectives.h"
using namespace Tabu;

#include <vector>
using namespace std;

//la structure de données représentant les labels à placer
#include "../Map.h"

//la structure de données repésentant la solution du placement
#include "../Solution.h"
#include "../LabelPlacement.h"

#include "SolutionTabu.h"
#include "OverlapManager.h"

namespace Tabu
{
    class TabuLabelPlacement : public LabelPlacement
    {
    public :

        /**
          * Constructeur,
          * Initialise certains paramètres par défaut.
          */
        TabuLabelPlacement();

        /**
          * Permet de définir les variables de l'algorithme.
          * p1 : le nombre d'itération maximum
          * p2 : le facteur définissant la taille de la liste de cadidats de base
          * p3 : le facteur de réduction du facteur p3
          * p4 : le facteur d'accroissement du facteur p3
          * p5 : la taille de la liste tabou en fonction du nombre de chevauchement
          * p6 : le nombre d'itération avant de recalculer les tailles des listes
          * (candidat et tabou)
          * p7 : la taille minimale de la liste des candidats
          * p8 : la taille minimal de la liste tabou
          */
        void setVariables(int nbIterationLimit,
            double candidateSizeBase, double candidateReduceFactor,
            double candidateGrowingFactor,
            double tabuSize, int nbIterationListSizeRecompute,
            int minimalCandidateListSize, int minimalTabuListSize)
        {
            this->nbIterationLimit = nbIterationLimit;

            this->candidateSizeBase = candidateSizeBase;
            this->candidateReduceFactor = candidateReduceFactor;
            this->candidateGrowingFactor = candidateGrowingFactor;
            this->tabuSize = tabuSize;
            this->nbIterationListSizeRecompute = nbIterationListSizeRecompute;
            this->minimalCandidateListSize = minimalCandidateListSize;
            this->minimalTabuListSize = minimalTabuListSize;
        }

        /**
          * Renvois le nombre d'itérations.
          */
        int getNbIteration() const
        {
            return this->nbIteration;
        }

        /**
          * Renvois le nombre de chevauchement initial
          */
        int getNbInitialOverlap() const
        {
            return this->nbInitialOverlap;
        }
```

```
        /**
          * Renvois le nombre de label (point).
          */
        int getNumberPoint() const
        {
            return int(this->solutionTabu->size());
        }

        /**
          * Commence le calcul.
          */
        virtual void compute();

    protected :

        /**
          * Convertit une "solution tabou" en une solution générique.
          */
        virtual void convSolution()=0;

        /**
          * Crée la solution de départ qui consiste à placer les labels dans la position préférée.
          * (même position pour tous les label)
          * p1 : le nombre de label
          * p2 : l'importance du chevauchement par rapport à la position
          */
        virtual void creatInitialSolution(int, double);

        /**
          * Est appelé lorsqu'un fichier de données n'est pas trouvé.
          * p1 : le nom du fichier
          */
        void fileError(char*);

        //le gestionnaire de placement
        OverlapManager* om;

        //les directives de placement (préférences de la position des labels)
        PlacementDirectives placementDirectives;

        //la solution : un tableau d'entier ou chaque nombre indique le placement du label
        SolutionTabu* solutionTabu;

        int nbIteration; //le nombre d'itération

        //le nombre de chevauchement initial
        int nbInitialOverlap;

        //le nombre d'itération maximum
        int nbIterationLimit;

        //la taille de la liste de candidat par rapport à au nombre de label se chevauchant
        double candidateSizeBase;
        double candidateReduceFactor;
        double candidateGrowingFactor;

        //la taille de la liste tabou par rapport à au nombre de label se chevauchant
        double tabuSize;

        //après combien d'itération il faut recalculer la taille des listes (tabou et candidats)
        int nbIterationListSizeRecompute;

        //taille minimale de la liste de candidat
        int minimalCandidateListSize;

        //taille minimale de la liste tabou
        int minimalTabuListSize;
    };

    class TabuLabelPlacementMap : virtual public TabuLabelPlacement
    {
        public :

        /**
          * Constructeur.
          * p1 : les données : les points ou il faut mettre un label
          * p2 : les directive de placement, par exemple les préférences de placement
          * d'un label autour d'un point.
          */
        TabuLabelPlacementMap(const Map*, PlacementDirectives);
        TabuLabelPlacementMap(){}
```

```
165         /**
              * Définit les labels à masquer en fonction de leur poinds
              */
            void hideLabels();

170         protected :

            /**
              * Convertit une "solution tabou" en une solution générique.
              */
175         void convSolution();

            const Map* map;
        };

180     class TabuLabelPlacementFile : virtual public TabuLabelPlacement
        {
            public :

            /**
185           * Constructeur.
              * p1 : le fichier contenant les données
              * p2 : les directive de placement, par exemple les préférences de placement
              * d'un label autour d'un point.
              */
190         TabuLabelPlacementFile(char*, PlacementDirectives);
            TabuLabelPlacementFile(){}

            protected :

195         /**
              * Convertit une "solution tabou" en une solution générique.
              */
            void convSolution();
        };
200 }

    #endif
```

C.17 Tabu : :TabuLabelPlacement.cpp

```
#include "TabuLabelPlacement.h"
using namespace Tabu;

#include <list>
#include <algorithm>
#include <fstream>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "CostLabelList.h"

#define TLP TabuLabelPlacement
#define TLPM TabuLabelPlacementMap
#define TLPF TabuLabelPlacementFile

//##################//
TLP::TabuLabelPlacement()
{
    //le nombre d'itération maximum
    this->nbIterationLimit = 1000;

    //la taille de la liste de candidat par rapport à au nombre de label se chevauchant
    this->candidateSizeBase = 0.2;
    this->candidateReduceFactor = 1.1;
    this->candidateGrowingFactor = 15; //1.5

    //la taille de la liste tabou par rapport à au nombre de label se chevauchant
    this->tabuSize = 0.35; //0.25

    //après combien d'itération il faut recalculer la taille des listes (tabou et candidats)
    this->nbIterationListSizeRecompute = 50;

    //taille minimale de la liste de candidat
    this->minimalCandidateListSize = 5;

    //taille minimale de la liste tabou
    this->minimalTabuListSize = 7;
}

//##################//
TLPM::TabuLabelPlacementMap(const Map* map, PlacementDirectives placementDirectives)
{
    this->placementDirectives = placementDirectives;

    this->map = map;

    this->solution = new Solution(this->map->getNbPointLabel());

    //la futur solution, par défaut on les places à la meilleure position
    this->solutionTabu = new SolutionTabu(this->map->getNbPointLabel());

    //crée la solution initiale
    this->creatInitialSolution(this->map->getNbPointLabel(),
        this->placementDirectives.getRatioOverlap());

    //le gestionnaire de chevauchement
    this->om = new OverlapManager(this->map, this->solutionTabu,
        this->placementDirectives);

    //convertit la solution tabu en solution générique
    this->convSolution();
}

//##################//
TLPF::TabuLabelPlacementFile(char* filename, PlacementDirectives placementDirectives)
{
    this->placementDirectives = placementDirectives;

    ifstream file(filename);
    if(!file)
        this->fileError(filename);

    int nbLabel, nbPosition;

    file >> nbLabel;
    file >> nbPosition;

    //solution qui ne servira à rien puisque aucune coordonné
```

```
        //n'est fournit dans les fichiers de données
        this->solution = new Solution(nbLabel);

85      //la futur solution, par défaut on les places à la meilleure position
        this->solutionTabu = new SolutionTabu(nbLabel);

        //crée la solution initiale
        this->creatInitialSolution(nbLabel, this->placementDirectives.getRatioOverlap());
90
        this->placementDirectives.setNbPosition(nbPosition);

        //le gestionnaire de chevauchement
        this->om = new OverlapManager(file, this->solutionTabu,
95          this->placementDirectives);

        //convertit la solution tabu en solution générique
        this->convSolution();
    }
100
    //#################//
    void TLP::compute()
    {
        double candidateSize = this->candidateSizeBase;
105
        const int INFINITE = 1000000000;

        //la valeur de la meilleure solution
        int bestSolutionValue = INFINITE;
110
        //la liste des coûts
        CostLabelList cll(*this->solutionTabu, *this->om);

        //les candidats
115     CostLabelList::iterator candidates;
        int candidateListLength; //la taille de la liste des candidats

        this->nbIteration = 0; //compte le nombre d'itération

120     //la liste des label tabous
        list<int> tabuList;
        int tabuListLength = 0; //la taille de la liste tabou

        //le nombre initial de chevauchement (pour information)
125     nbInitialOverlap = this->om->numberOfLabelThatOverlap();

        //le nombre de chevauchement
        int numberOfLabelThatOverlap = nbInitialOverlap;

130     //tant qu'il y a encore des chevauchement et que le nombre d'itération n'est pas atteint
        while(this->nbIteration != nbIterationLimit && numberOfLabelThatOverlap != 0)
        {
            //le facteur de la taille de la liste de candidat diminue
            //si elle est plus élevé que le facteur de base
135         if(candidateSize > candidateSizeBase) candidateSize /= candidateReduceFactor;

            //toute les n itérations : recalul la taille de la liste de candidat et tabou
            if(this->nbIteration % nbIterationListSizeRecompute == 0)
            {
140             tabuListLength = minimalTabuListSize + int(tabuSize * numberOfLabelThatOverlap);

                candidateListLength = minimalCandidateListSize +
                    int(candidateSize * numberOfLabelThatOverlap);

145             //si la liste des candidats excède le nombre de label
                if(candidateListLength > int(this->solutionTabu->size()))
                    candidateListLength = int(this->solutionTabu->size());
            }

150         candidates = cll.end() - candidateListLength;

            //cherche à modifier l'emplacement d'un label de la liste
            //des candidats afin de trouver le coût le plus faible
            double bestCostDif = -INFINITE;
155         double bestCost = 0;
            CostLabelList::iterator labelListCost;
            int bestPosition = -1;

            for(CostLabelList::iterator i = candidates; i != cll.end(); i++)
160         {
                //pour chaque position possible
                for(int pos = 0; pos<this->placementDirectives.getNbPosition(); pos++)
                {
                    if(pos == (*this->solutionTabu)[i->numLabel]) continue;
```

```
165
                double cost = this->om->cost(i->numLabel, pos);

                //si le coût est meilleur que le meilleur trouvé et qu'il ne se trouve pas
                //dans la liste tabou ou si il améliore la solution actuelle
170             //alors le mémorise comme meilleur coût courant
                if((i->cost - cost > bestCostDif) &&
                    (i->cost - cost > 0 ||
                    find(tabuList.begin(), tabuList.end(), i->numLabel) == tabuList.end()))
                {
175                 bestCostDif = i->cost - cost;
                    bestCost = cost;
                    labelListCost = i;
                    bestPosition = pos;
                }
180         }
        }

        this->nbIteration++;

185     //si parmis les candidats aucun n'a pû être choisit
        //(ils font tous partit de la liste tabou) alors on augmente temporairement
        //la taille de la liste de candidats
        if(bestPosition == -1)
        {
190         //on limite la taille pour éviter la dégénération
            if(candidateSize < 10e4) candidateSize *= candidateGrowingFactor;

            candidateListLength = minimalCandidateListSize +
                int(candidateSize * numberOfLabelThatOverlap);
195
            if(candidateListLength > int(this->solutionTabu->size()))
                candidateListLength = int(this->solutionTabu->size());
            continue;
        };
200
        //ajoute le label à la liste tabou
        if(find(tabuList.begin(), tabuList.end(), labelListCost->numLabel) == tabuList.end())
            tabuList.push_front(labelListCost->numLabel);
        if(int(tabuList.size()) > tabuListLength) tabuList.resize(tabuListLength);
205
        //met à jour la position
        (*this->solutionTabu)[labelListCost->numLabel] = bestPosition;

        //met à jour le cout du label
210     cll.updateCost(labelListCost, bestCost);

        numberOfLabelThatOverlap = this->om->numberOfLabelThatOverlap();
        if(numberOfLabelThatOverlap < bestSolutionValue)
            bestSolutionValue = numberOfLabelThatOverlap;
215 }

    this->solution->setNumberOfOverlap(bestSolutionValue);

    //convertit la solution tabu en solution générique
220 this->convSolution();

}

//#################//
225 void TLPM::hideLabels()
{
    this->om->hideLabels(this->map, this->solution);
}

230 //#################//
void TLPM::convSolution()
{
    double length = this->placementDirectives.getPointLabelLength();

235 //parcours tous les labels
    for(int i=0; i<this->map->getNbPointLabel(); i++)
        this->solution->setLabelPlacement(i,
            (*this->solutionTabu)[i] == 0 ?
                Point<double>(length, length) :
240             (*this->solutionTabu)[i] == 1 ?
                Point<double>(-length - this->map->getPointLabel(i).l, length) :
            (*this->solutionTabu)[i] == 2 ?
                Point<double>(-length - this->map->getPointLabel(i).l,
                    -length - this->map->getPointLabel(i).h) :
245             Point<double>(length,
                    -length  - this->map->getPointLabel(i).h),
            this->om->overlap(i, false, -1) > 0
```

```
        );
}

//#################//
void TLPF::convSolution()
{
    //parcours tous les labels
    for(unsigned int i=0; i<this->solutionTabu->size(); i++)
        this->solution->setLabelPlacement(i, Point<double>(0,0), false);
}

//#################//
void TLP::creatInitialSolution(int nbLabel, double ratioOverlap)
{
    int bestPosition = 0;

    //si la position des labels n'a pas d'importance alors
    //on va les placer initialement de manière aléatoire
    if(ratioOverlap != 1.0)
    {
        //cherche la meilleurs position pour le placement initial des labels
        double bestPositionValue = 99;
        for(int i=0; i<this->placementDirectives.getNbPosition(); i++)
            if(this->placementDirectives.getPreference(i) < bestPositionValue)
            {
                bestPositionValue =this->placementDirectives.getPreference(i);
                bestPosition = i;
            }
    }

    for(int i=0; i<nbLabel; i++)
        (*this->solutionTabu)[i] = ratioOverlap != 1.0 ? bestPosition :
            int((double(rand())/RAND_MAX)*4);
}

//#################//
void TLP::fileError(char* name)
{
    cerr << "Error : the file " << name << " doesn't exist" << endl;
    exit(1);
}

#undef TLP
#undef TLPM
#undef TLPF
```

C.18 Tabu : :SolutionTabu.h

```
/**
  * Représente une solution c-à-d la position de chaque label.
  */
#ifndef _TABU_SOLUTIONTABU_H_
#define _TABU_SOLUTIONTABU_H_

#include <vector>
using namespace std;

namespace Tabu
{

    class SolutionTabu : public vector<int>
    {
    public :

        /**
          * Constructeur, initialise le nombre de label à 0.
          */
        SolutionTabu() : vector<int>(0){}

        /**
          * Constructeur.
          * p1 : le nombre de label de la solution
          */
        SolutionTabu(int n) : vector<int>(n){}

        /**
          * Destructeur.
          */
        virtual ~SolutionTabu(){}

        /**
          * Permet d'acceder à la position d'un label.
          * p1 : le numéro du du label
          */
        virtual value_type& operator[](size_type i)
        {
            return *(begin() + i);
        }

        /**
          * Permet d'acceder à la position d'un label, renvois un const.
          * p1 : le numéro du du label
          */
        virtual const value_type& operator[](size_type i) const
        {
            return *(begin() + i);
        }

        /**
          * Renvois la taille de la solution, renvois la taille du sous-ensemble si il est actif.
          */
        virtual size_type size() const
        {
            return vector<int>::size();
        }
    };
}

#endif
```

C.19 Tabu : :PlacementDirectives.h

```
/**
  * Représente les préférences de placement des
  * labels.
  * Un label peut-être placé, relativement à son point,
  * de quatre manières différentes :
  * - en haut à droite [0]
  * - en haut à gauche [1]
  * - en bas à gauche [2]
  * - en bas à droite [3]
  * Pour chaque position une préfétence est donnée,
  * elle part de 0 (meilleur placement) jusqu'a 1
  * (moins bon placement).
  */

#ifndef _TABU_PLACEMENTDIRECTIVES_H_
#define _TABU_PLACEMENTDIRECTIVES_H_

namespace Tabu
{
    class PlacementDirectives
    {
    public :

        /**
          * Constructeur.
          * p1: les préferences de chacune des position
          * p2 : le nombre de position
          * p3 : la distance entre les lablel et leur point
          * p4 : l'importance du chevauchement par rapport à la position
          *  entre 1 et 0, 1 étant le plus fort
          */
        PlacementDirectives(const double*, int, double, double);
        PlacementDirectives(){}

        /**
          * Renvois une préférence.
          * p1 : le numéro préférence
          */
        double getPreference(int) const;

        /**
          * Renvois le nombre de position.
          */
        int getNbPosition() const;

        /**
          * Renvois la distance entre le label et son point.
          */
        double getPointLabelLength() const;

        /**
          * Définit le nombre de position.
          * p1 : le nombre de position
          */
        void setNbPosition(int);

        /**
          * Renvois le poids des chevauchements par rapport à la position.
          */
        double getRatioOverlap() const;

    private :

        //les préférences
        const double* positionPreferences;

        //le nombre de position du label
        int nbPosition;

        //la distance entre le label et son point
        double pointLabelLength;

        //l'importance du chevauchement par rapport à la position
        double ratioOverlap;
    };

    class ErrorDirectives {};
}

#endif
```

C.20 Tabu : :PlacementDirectives.cpp

```
#include "PlacementDirectives.h"
using namespace Tabu;

//#################//
PlacementDirectives::PlacementDirectives(const double* p, int nbPosition,
    double pointLabelLength, double ratioOverlap)
{
    this->ratioOverlap = ratioOverlap;

    if(pointLabelLength >= 0)
        this->pointLabelLength = pointLabelLength;
    else
        throw new ErrorDirectives();

    this->nbPosition = nbPosition;
    this->positionPreferences = p;

    //vérifie la validité des préférences
    for(int i = 0; i < this->nbPosition; i++)
        if(this->positionPreferences[i] > 1 || this->positionPreferences[i] < 0)
            throw new ErrorDirectives();
}

//#################//
double PlacementDirectives::getPreference(int num) const
{
    //vérifie si la préférence existe
    if(num <= this->nbPosition && num >= 0)
        return  this->positionPreferences[num];
    throw new ErrorDirectives();
}

//#################//
int PlacementDirectives::getNbPosition() const
{
    return this->nbPosition;
}

//#################//
double PlacementDirectives::getPointLabelLength() const
{
    return this->pointLabelLength;
}

//#################//
void PlacementDirectives::setNbPosition(int nb)
{
    this->nbPosition = nb;
}

//#################//
double PlacementDirectives::getRatioOverlap() const
{
    return this->ratioOverlap;
}
```

C.21 Tabu : :OverlapManager.h

```
/**
  * Gère les interactions entre les labels, on peut par exemple
  * demander combien de chevauchement crée un label.
  * Définit également le coût de placement d'un label.
  */

#ifndef _TABU_OVERLAPMANAGER_H_
#define _TABU_OVERLAPMANAGER_H_

#include <map>
#include <set>
#include <vector>
#include <fstream>
#include <algorithm>
using namespace std;

#include "../Solution.h"
#include "../Map.h"

#include "SolutionTabu.h"
#include "PlacementDirectives.h"

#include "../Map.h"

namespace Tabu
{
    class OverlapManager
    {
    public :
        /**
          * Constructeur.
          * p1 : les label et leur point
          * p2 : la solution
          * p3 : les préférences de placement
          * TODO : rendre le nombre de position dynamique
          * (il faut faire un générateur de placement autour du point).
          */
        OverlapManager(const Map*, SolutionTabu*,const PlacementDirectives&);

        /**
          * Constructeur.
          * p1 : le fichier de données
          * p2 : la solution
          * p3 : les préférences de placement
          */
        OverlapManager(ifstream&, SolutionTabu*,const PlacementDirectives&);

        /**
          * Renvois soit le nombre d'autre label que touche 'label' (p2 = true),
          * ou soit si la label touche au moins un autre label (p2 = false).
          * p1 : le numéro du label
          * p2 : ^
          * p3 : la position du label, par défaut prends sa position courante (facultatif)
          */
        virtual int overlap(int, bool, int) const;

        /**
          * Renvois le nombre de label en touchant au moins un autre.
          */
        int numberOfLabelThatOverlap() const;

        /**
          * Retourne le coût que le label n° p1 engendre, calculé selon la formule :
          * alpha1 * nb chevauchement + alpha2 * préférence de placement.
          * p1 : le numéro du label
          * p2 : la position du label (facultatif)
          */
        double cost(int, int) const;
        double cost(int) const;

        /**
          * Définit les labels à masquer (en fonction de leur poids).
          * p1 : les labels, contient les poids
          * p2 : la solution dans laquelle le masquage va être indiqué
          */
        void hideLabels(const Map*, Solution*);

    protected :

        typedef map<int, set<int> > element;
```

```
        //l'ensemble des chevauchements possibles
        element** overlaps;

        //les directives de placement
        const PlacementDirectives& placementDirectives;

        //la solution, comment se positionne les labels,
        //rappel : il y a 4 positions posible : 0, 1, 2, 3
        SolutionTabu* solution;
    };
}

#endif
```

C.22 Tabu : :OverlapManager.cpp

```
#include "OverlapManager.h"
using namespace Tabu;

#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;a

#include "../Point.cpp"

#define ns OverlapManager

//#################//
ns::OverlapManager(const Map* map, SolutionTabu* solution,
    const PlacementDirectives& pd)
: placementDirectives(pd), solution(solution)
{
    double length = placementDirectives.getPointLabelLength();

    this->overlaps = new element*[map->getNbPointLabel()];

    //pour chaque label on définit 4 positions possibles
    for(int i=0; i<map->getNbPointLabel(); i++)
        this->overlaps[i] = new element[4];

    //à chaque itération on calcul le coin inférieur gauche des 2 labels dont on veut
    //tester si ils se chevauchent
    Point<double> bottomLeft1;
    Point<double> bottomLeft2;

    //les hauteurs et largeurs des labels à tester
    double h1, h2, l1, l2;

    //pour chaque label
    for(int i=0; i<map->getNbPointLabel(); i++)
    {
        //pour chaque position du label
        for(int j=0; j<4; j++)
        {
            //pour chaque label2
            for(int i2=0; i2<map->getNbPointLabel(); i2++)
            {
                if(i==i2) continue;

                //pour chaque position du label2
                for(int j2=0; j2<4; j2++)
                {
                    l1 = map->getPointLabel(i).l;
                    l2 = map->getPointLabel(i2).l;
                    h1 = map->getPointLabel(i).h;
                    h2 = map->getPointLabel(i2).h;

                    //calcul le coin inférieur gauche du premier label
                    switch(j)
                    {
                    case 0 :
                        bottomLeft1 = Point<double>(length, length); break;
                    case 1 :
                        bottomLeft1 = Point<double>(-length - l1, length); break;
                    case 2 :
                        bottomLeft1 = Point<double>(-length - l1, -length - h1); break;
                    default :
                        bottomLeft1 = Point<double>(length, -length - h1); break;
                    }
                    bottomLeft1 += map->getPointLabel(i).position;

                    //calcul le coin inférieur gauche du deuxième label
                    switch(j2)
                    {
                    case 0 :
                        bottomLeft2 = Point<double>(length, length); break;
                    case 1 :
                        bottomLeft2 = Point<double>(-length - l2, length); break;
                    case 2 :
                        bottomLeft2 = Point<double>(-length - l2, -length - h2); break;
                    default :
                        bottomLeft2 = Point<double>(length, -length - h2); break;
                    }
                    bottomLeft2 += map->getPointLabel(i2).position;

                    //test si le label 1 entre en conflit avec le label 2,
```

```
                    //si oui alors l'indique dans la structure de données
                    if(
                        //si le point 1 se trouve dans le label 2 => chevauchement
                        (map->getPointLabel(i).position.x > bottomLeft2.x &&
                            map->getPointLabel(i).position.x < bottomLeft2.x + l2
                        &&
                        map->getPointLabel(i).position.y > bottomLeft2.y &&
                            map->getPointLabel(i).position.y < bottomLeft2.y + h2)
                        ||
                        //si le point 2 se trouve dans le label 1 => chevauchement
                        (map->getPointLabel(i2).position.x > bottomLeft1.x &&
                            map->getPointLabel(i2).position.x < bottomLeft1.x + l1
                        &&
                        map->getPointLabel(i2).position.y > bottomLeft1.y &&
                            map->getPointLabel(i2).position.y < bottomLeft1.y + h1)
                        ||
                        //si le label 1 chevauche le label 2 => chevauchement
                        (((bottomLeft1.x > bottomLeft2.x &&
                            bottomLeft1.x <= bottomLeft2.x + l2) ||
                        (bottomLeft1.x + l1 > bottomLeft2.x &&
                            bottomLeft1.x + l1 <= bottomLeft2.x + l2) ||
                        (bottomLeft1.x < bottomLeft2.x &&
                            bottomLeft1.x + l1 >= bottomLeft2.x + l2))
                        &&
                        ((bottomLeft1.y > bottomLeft2.y &&
                            bottomLeft1.y <= bottomLeft2.y + h2) ||
                        (bottomLeft1.y + h1 > bottomLeft2.y &&
                            bottomLeft1.y + h1 <= bottomLeft2.y + h2) ||
                        (bottomLeft1.y < bottomLeft2.y &&
                            bottomLeft1.y + h1 >= bottomLeft2.y + h2)))
                    )
                        this->overlaps[i][j][i2].insert(this->overlaps[i][j][i2].end(), j2);
                }
            }
        }
    }
}

//#################//
ns::OverlapManager(ifstream& file, SolutionTabu* solution,
    const PlacementDirectives& pd)
: placementDirectives(pd), solution(solution)
{
    this->overlaps = new element*[this->solution->size()];

    //pour chaque label on définit les positions possibles
    for(int i=0; i<int(this->solution->size()); i++)
        this->overlaps[i] = new element[this->placementDirectives.getNbPosition()];

    int nbOverlap; //le nombre de label que chevauche le label i

    //pour chaque point
    for(int i=0; i<int(this->solution->size()); i++)
    {
        //pour chaque position du label
        for(int j=0; j<this->placementDirectives.getNbPosition(); j++)
        {
            file >> nbOverlap;
            for(int k=0; k<nbOverlap; k++)
            {
                int tmp;
                file >> tmp;
                tmp--; //on commence à 0 et non à 1

                int numPoint = int(tmp / this->placementDirectives.getNbPosition());
                int numPosition = tmp % this->placementDirectives.getNbPosition();

                //on ne tient pas compte du chevauchement avec soi même
                if(numPoint == i) continue;

                this->overlaps[i][j][numPoint].insert(this->overlaps[i][j][numPoint].end(),
                    numPosition);
            }
        }
    }
}

//#################//
int ns::overlap(int numLabel, bool number, int position = -1) const
{
    int nb = 0; //nombre de label en collision
    int pos = position;
```

```
165
        if(position == -1) pos = (*this->solution)[numLabel];

        //pour chaque potentiel label pouvant entrer en conflit
        for(element::iterator i = this->overlaps[numLabel][pos].begin();
170         i!=this->overlaps[numLabel][pos].end(); i++)
            if(find(i->second.begin(), i->second.end(), (*this->solution)[i->first]) !=
                i->second.end())
            {
                nb++;
175             if(!number) break;
            }

        return nb;
    }
180
    //#################//
    int ns::numberOfLabelThatOverlap() const
    {
        int nb = 0;
185
        for(int i = 0; i < int(this->solution->size()); i++)
            if(this->overlap(i, false) == 1) nb++;

        return nb;
190 }

    //#################//
    double ns::cost(int numLabel, int position) const
    {
195     int nbOverlap = this->overlap(numLabel, true, position);

        //alpha1 * nCh + alpha2 * position
        return this->placementDirectives.getRatioOverlap() * nbOverlap +
            (1 - this->placementDirectives.getRatioOverlap()) *
200         this->placementDirectives.getPreference((*this->solution)[numLabel]);
    }

    //#################//
    double ns::cost(int numLabel) const
205 {
        return this->cost(numLabel, -1);
    }

    //#################//
210 //classe définissant un poids pour un label
    struct Weight
    {
        Weight(int num, double weight) : num(num), weight(weight){}
        int num;
215     double weight;
    };

    //classe de comparaison entre 2 'Weight'
    struct Comp
220 {
        int operator()(const Weight& w1, const Weight& w2)
        {
            return w1.weight > w2.weight;
        }
225 };

    void ns::hideLabels(const Map* map, Solution* solution)
    {
        //tableau permettant de savoir quel label est masqué
230     bool* hiddenLabel = new bool[map->getNbPointLabel()];

        //un tableau de 'Weight', c'est à dire quel label a quel poids
        vector<Weight> weightSorted;

235     //pour chaque label
        for(int i=0; i<map->getNbPointLabel(); i++)
        {
            //on introduit le poids et le numéro du label dans le tableau
            weightSorted.push_back(Weight(i, map->getPointLabel(i).weight));
240         hiddenLabel[i] = false; //on en profite pour mettre à faux les labels cachés
        }

        //trie le tableau des poids de manière à les traiter dans l'ordre
        //en commencant pas le plus lourd par la suite
245     sort< vector<Weight>::iterator, Comp >(weightSorted.begin(), weightSorted.end(), Comp());

        //parcours chaque labels (du plus lourd au moins lourd)
```

```
    for(vector<Weight>::iterator i=weightSorted.begin(); i!=weightSorted.end(); i++)
    {
        //si il est caché alors se label ne peux plus en cacher un autre
        if(hiddenLabel[i->num]) continue;

        //pour chaque label avec lequel un conflit peut avoir lieu
        for(element::iterator j = this->overlaps[i->num][(*this->solution)[i->num]].begin();
            j!=this->overlaps[i->num][(*this->solution)[i->num]].end(); j++)

            //on regarde si la position de ce label existe dans l'ensemble
            //des positions dans lequelles il entre en conflit
            if(find(j->second.begin(), j->second.end(), (*this->solution)[j->first]) !=
                j->second.end())
                    hiddenLabel[j->first] = true; //si il entre en conflit alors on le masque
    }

    //on donne à la solution le tableau des labels masqués
    solution->setHiddenLabels(hiddenLabel);
}

#undef ns
```

C.23 Tabu : :CostLabelList.h

```
/**
  * Mémorise les coûts des labels.
  * La liste est en permanance triée par coût ceci afin de pouvoir
  * facilement définir un ensemble de label ayant un coût supérieur.
  */

#ifndef _TABU_COSTLABELLIST_H_
#define _TABU_COSTLABELLIST_H_

#include <vector>
using namespace std;

#include "SolutionTabu.h"
#include "OverlapManager.h"

namespace Tabu
{
    /**
      * Permet de stocker le coût des labels, élément de la liste
      */
    struct CostLabel
    {
        /**
          * Constructeurs.
          * p1 : le coût
          * p2 : le numéro du label
          * p3 : vrai si le label chevauche au moins un autre label
          */
        CostLabel(double cost, int numLabel)
        : cost(cost), numLabel(numLabel){}

        CostLabel(){}

        double cost; //le coût de placement
        int numLabel; //le numéro du label
    };

    class CostLabelList : public vector<CostLabel>
    {
    public :

        /**
          * Constructeur.
          * p1 : le placement des éléments
          * p2 : le gestionnaire des chevauchements
          */
        CostLabelList(const SolutionTabu&, const OverlapManager&);

        /**
          * Change le coût d'un label.
          * p1 : un ponteur sur le label de la liste
          * p2 : le nouveau cout
          */
        void updateCost(CostLabelList::iterator, double);
    };
}

#endif
```

C.24 Tabu : :CostLabelList.cpp

```
#include "CostLabelList.h"
using namespace Tabu;

#include <algorithm>
using namespace std;

#define ns CostLabelList

//permet de trier les labels par leur coût
struct Cmp
{
     bool operator()(const CostLabel& c1, const CostLabel& c2)
     {
          return c1.cost < c2.cost;
     }
};

//#################//
ns::CostLabelList(const SolutionTabu& position, const OverlapManager& om)
: vector<CostLabel>(position.size())
{
     //calcul le coût de tous les labels
     for(int i = 0; i<int(position.size()); i++)
          (*this)[i] = CostLabel(om.cost(i), i);

     sort<CostLabelList::iterator, Cmp>(this->begin(), this->end(), Cmp());
}

//#################//
void ns::updateCost(CostLabelList::iterator i, double cost)
{
     i->cost = cost;
     sort<CostLabelList::iterator, Cmp>(this->begin(), this->end(), Cmp());
}

#undef ns
```

C.25 Tabu : :POPMUSIC : :PopTabuLabelPlacement.h

```
/**
  * Classe implementant la placement des labels à l'aide
  * de la metaheuristique 'POPMUSIC'.
  * Cette classe se base sur la recherche tabu.
  */

#ifndef _POPTABULABELPLACEMENT_H_
#define _POPTABULABELPLACEMENT_H_

#include "../TabuLabelPlacement.h"
#include "PopOverlapManager.h"
#include "PopSolutionTabu.h"

namespace Tabu
{
    namespace POPMUSIC
    {
        class TabuLabelPlacement : public TabuLabelPlacementMap, public TabuLabelPlacementFile
        {
        public :
            /**
              * Constructeur à partir d'un fichier.
              * p1 : le nom du fichier
              * p2 : les préférences de placement
              * p3 : la taille du sous ensemble
              * p4 : la graine qui va déterminer l'ordre de séléction des points
              * p5 : le nombre maximum d'iteration (si -1 alors va jusqu'au bout)
              * p6 : la version de popmusic : 1=version original 2=version modifié
              */
            TabuLabelPlacement(char*, PlacementDirectives, int, int, int, int);

            /**
              * p1 : les label et leur point
              * p2 : les préférences de placement
              * p3 : la taille du sous ensemble
              * p4 : la graine qui va déterminer l'ordre de séléction des points
              * p5 : le nombre maximum d'iteration (si -1 alors va jusqu'au bout)
              * p6 : la version de popmusic : 1=version original 2=version modifié
              */
            TabuLabelPlacement(const Map*, PlacementDirectives, int, int, int, int);

            /**
              * Commence le calcul.
              */
            void compute();

        protected :

            /**
              * Convertit une "solution tabou" en une solution générique.
              */
            void convSolution();

            //la taille des sous ensemble
            int subSetSize;

            //le nombre maximal de sous-ensemble
            int nbIterationMax;

            //la vesrion de popmusic
            int version;
        };
    }
}

#endif
```

C.26 Tabu : :POPMUSIC : :PopTabuLabelPlacement.cpp

```
#include "PopTabuLabelPlacement.h"
using namespace Tabu;

#include <cstdlib>
#include <fstream>
using namespace std;

#define ns Tabu::POPMUSIC::TabuLabelPlacement

//#################//
ns::TabuLabelPlacement(char* name, PlacementDirectives pd,
    int subSetSize, int seed, int nbIterationMax, int version)
: TabuLabelPlacementMap(), TabuLabelPlacementFile()
{
    srand(seed == -1 ? time(0) : seed);

    this->version = version;

    this->nbIterationMax = nbIterationMax;

    this->placementDirectives = pd;

    ifstream file(name);
    if(!file)
        this->fileError(name);

    int nbLabel, nbPosition;

    file >> nbLabel;
    file >> nbPosition;

    //solution qui ne servira à rien puisque aucune coordonné
    //n'est fournit dans les fichiers de données
    this->solution = new Solution(nbLabel);

    //la futur solution, par défaut on les places à la meilleure position
    this->solutionTabu = new POPMUSIC::SolutionTabu(nbLabel);

    //crée la solution initiale
    this->creatInitialSolution(nbLabel, this->placementDirectives.getRatioOverlap());

    this->placementDirectives.setNbPosition(nbPosition);

    //le gestionnaire de chevauchement
    this->om = new POPMUSIC::OverlapManager(file, this->solutionTabu,
        this->placementDirectives);


    //convertit la solution tabu en solution générique
    Tabu::TabuLabelPlacementFile::convSolution();

    this->subSetSize = subSetSize;
}

//#################//
ns::TabuLabelPlacement(const Map* map, PlacementDirectives pd,
    int subSetSize, int seed, int nbIterationMax, int version)
: TabuLabelPlacementMap(), TabuLabelPlacementFile()
{
    srand(seed == -1 ? time(0) : seed);

    this->version = version;

    this->nbIterationMax = nbIterationMax;

    this->placementDirectives = pd;
    this->map = map;
    this->solution = new Solution(this->map->getNbPointLabel());

    //la futur solution, par défaut on les places à la meilleure position
    this->solutionTabu = new POPMUSIC::SolutionTabu(this->map->getNbPointLabel());

    //crée la solution initiale
    this->creatInitialSolution(this->map->getNbPointLabel(),
        this->placementDirectives.getRatioOverlap());

    //le gestionnaire de chevauchement
    this->om = new POPMUSIC::OverlapManager(this->map, this->solutionTabu,
        this->placementDirectives);

```

```
    //convertit la solution tabu en solution générique (pas joli joli comme méthode)
    Tabu::TabuLabelPlacementMap::convSolution();

    this->subSetSize = subSetSize;
}

//#################//
void ns::compute()
{
    //mute l'overlap manager
    POPMUSIC::OverlapManager* omTmp = dynamic_cast<POPMUSIC::OverlapManager*>(this->om);

    //définit la taille maximale des sous-ensemble
    omTmp->setSubSetSize(this->subSetSize);

    //l'ensemble des points dont on ne peut plus améliorer le sous-problème
    set<int> O;

    //désactive la vue du sous ensemble
    dynamic_cast<POPMUSIC::SolutionTabu*>(this->solutionTabu)->setSubSetActive(false);

    int solutionValueAvant; //la valeur de la solution actuelle
    int solutionValueApres = omTmp->solutionValue();
    int solutionValueDebut = solutionValueApres; //mémorise le nombre de chevauchement initial


    //compte le nombre d'itération
    int nbIterationPOP = 0;

    //tant que tout les points ne se trouve pas dans O
    while(O.size() != this->solutionTabu->size() &&
        (this->nbIterationMax == -1 || nbIterationPOP < this->nbIterationMax))
    {
        nbIterationPOP++;

        solutionValueAvant = solutionValueApres;

        //on tire un point aléatoirement qui ne se trouve pas dans O
        int point;
        do
        {
            point = int((double(rand())/RAND_MAX)*this->solutionTabu->size());
        }while(O.find(point)!=O.end());

        //définit le sous ensemble
        omTmp->selectSubSet(point);

        //active la vue du sous ensemble
        dynamic_cast<POPMUSIC::SolutionTabu*>(this->solutionTabu)->setSubSetActive(true);

        //mémorise la solution avant l'optimisation
        omTmp->saveSolution();

        //applique l'optimisation
        Tabu::TabuLabelPlacement::compute();

        //désactive la vue du sous ensemble
        dynamic_cast<POPMUSIC::SolutionTabu*>(this->solutionTabu)->setSubSetActive(false);

        solutionValueApres = omTmp->solutionValue();
        if(solutionValueApres < solutionValueAvant)
        {
            O.clear();
        }
        else
        {
            //revient à l'ancienne solution
            omTmp->rollbackToSavedSolution(O, this->version);

            //si c'est la version original alors on insert uniquement le point,
            //sinon on insert tout le sous problème
            if(this->version == 1) O.insert(point);

            solutionValueApres = solutionValueAvant;
        }
    }

    //définit les variables de résultats
    this->nbInitialOverlap = solutionValueDebut;
    this->solution->setNumberOfOverlap(solutionValueApres);
    this->nbIteration = nbIterationPOP;

    //convertit la solution tabu en solution générique (pas joli joli comme méthode)
```

```
165     if(this->map)
            Tabu::TabuLabelPlacementMap::convSolution();
        else
            Tabu::TabuLabelPlacementFile::convSolution();
    }
170
    //#################//
    void ns::convSolution()
    {}
```

C.27 Tabu : :POPMUSIC : :PopSolutionTabu.h

```
/**
 * Représente la solution mais permet de restreindre la vue à un sous ensemble
 * Ceci permet d'appliquer Tabou seulement à un sous-ensemble de point ce qui est
 * nécessaire dans le cas de POPMUSIC.
 */

#ifndef _POP_TABU_SOLUTIONTABU_H_
#define _POP_TABU_SOLUTIONTABU_H_

#include "../SolutionTabu.h"

namespace Tabu
{
    namespace   POPMUSIC
    {

        class SolutionTabu : public Tabu::SolutionTabu
        {
        public :

            /**
             * Constructeur, par défaut la solution est de taille nulle.
             */
            SolutionTabu() : Tabu::SolutionTabu(0)
            {
                this->subSetActive = false;
            }

            /**
             * Constructeur.
             * p1 : la taille de la solution
             */
            SolutionTabu(int n) : Tabu::SolutionTabu(n)
            {
                this->subSetActive = false;
            }

            /**
             * Permet d'acceder à la position d'un label.
             * p1 : le numéro du du label
             */
            value_type& operator[](size_type i)
            {
                return this->subSetActive ? this->at(subSet[i]) : this->at(i);
            }

            /**
             * Permet d'acceder à la position d'un label, renvois un const.
             * p1 : le numéro du du label
             */
            const value_type& operator[](size_type i) const
            {
                return this->subSetActive ? this->at(subSet[i]) : this->at(i);
            }

            /**
             * Active ou désactive le sous ensemble de point,
             * tout les autres points ne peuvent pas être accedés.
             * p1 : activation / désactivation
             */
            void setSubSetActive(bool active)
            {
                this->subSetActive = active;
            }

            /**
             * Savoir si le sous-ensemble est actif.
             */
            bool subSetIsActive() const
            {
                return this->subSetActive;
            }

            /**
             * Renvois la taille de la solution,
             * renvois la taille du sous-ensemble si il est actif.
             */
            size_type size() const
            {
                if(this->subSetActive)
                    return *this->subSetSize;
```

```
                else
                    return Tabu::SolutionTabu::size();
            }

            /**
              * Définit le sous ensemble.
              * p1 : le sous-ensemble sous la forme d'un tableau de numéro de point
              * (ceux qui appartiennent au sous-ensemble).
              * p2 : la taille du sous-ensemble
              */
            void setSubSet(int* set, int* size)
            {
                this->subSet = set;
                this->subSetSize = size;
                this->subSetActive = true;
            }

        private :

            //la taille du sous ensemble
            int* subSetSize;

            //le sous ensemble
            int* subSet;

            //est-ce que la vue est limité au sous ensemble courant ?
            bool subSetActive;
        };
    }
}

#endif
```

C.28 Tabu : :POPMUSIC : :PopOverlapManager.h

```
/**
  * Cette classe redéfinit légérement la sementique de sa mère :
  *  Les fonctions 'numberOfLabelThatOverlap', 'overlap' et 'cost'
  * s'appliquent à un sous ensemble des points.
  */

#ifndef _TABU_POPMUSIC_POPOVERLAPMANAGER_H_
#define _TABU_POPMUSIC_POPOVERLAPMANAGER_H_

#include <map>
#include <set>
#include <vector>
#include <fstream>
using namespace std;

#include "../../Map.h"
#include "../OverlapManager.h"
#include "PopSolutionTabu.h"

namespace Tabu
{
    namespace POPMUSIC
    {
        class OverlapManager : public Tabu::OverlapManager
        {
        public :
            /**
              * Constructeur #1
              * p1 : les label et leur point
              * p2 : la solution
              * p3 : les préférences de placement
              */
            OverlapManager(const Map*, Tabu::SolutionTabu*, const PlacementDirectives&);

            /**
              * Constructeur #2
              * p1 : le fichier de données
              * p2 : la solution
              * p3 : les préférences de placement
              */
            OverlapManager(ifstream&, Tabu::SolutionTabu*,const PlacementDirectives&);

            /**
              * Renvois la valeur de la solution (le nombre de chevauchement)
              */
            int solutionValue() const;

            /**
              * Renvois soit le nombre d'autre label que touche 'label' (p2 = true),
              * ou soit si la label touche au moins un autre label (p2 = false).
              * p1 : le numéro du label
              * p2 : ^
              * p3 : la position du label, par défaut prends sa position courante (facultatif)
              */
            int overlap(int, bool, int) const;

            /**
              * Définit la taille (maximale) des sous-ensembles.
              * p1 : la taille
              */
            void setSubSetSize(int);

            /**
              * Définit les éléments du sous-ensemble en fonction d'un point de départ.
              * p1 : le point de départ
              */
            void selectSubSet(int);

            /**
              * Mémorise la solution définit par le sous ensemble courant.
              */
            void saveSolution();

            /**
              * Restore la solution courante par l'ancienne
              * (dans le cas ou la nouvelle et moins bonne)
              * p1 : l'ensemble O
              * p2 : la version de popmusic à appliquer
              */
            void rollbackToSavedSolution(set<int>&, int);
```

```
            /**
              * Renvois la taille du sous-ensemble courant.
              */
            int getSubSetSize() const;

        private :

            //le sous ensemble
            int* subSet;

            //la taille maximale des sous-ensembles
            int subSetSizeMax;

            //la taille du sous-ensemble courant
            int* subSetSize;

            //pour stocker temporairement la solution (afin de pouvoir revenir en arrière)
            int* oldSolution;
        };
    }
}

#endif
```

C.29 Tabu : :POPMUSIC : :PopSolutionTabu.cpp

```
#include "PopOverlapManager.h"
using namespace Tabu;

#include <list>
#include <map>
using namespace std;

#include "../../Point.cpp"

#define ns Tabu::POPMUSIC::OverlapManager

//#################//
ns::OverlapManager(const Map* map, Tabu::SolutionTabu* solution, const PlacementDirectives& pd)
: Tabu::OverlapManager::OverlapManager(map, solution, pd)
{}

//#################//
ns::OverlapManager(ifstream& file, Tabu::SolutionTabu* solution, const PlacementDirectives& pd)
: Tabu::OverlapManager::OverlapManager(file, solution, pd)
{}

//#################//
int ns::solutionValue() const
{
    int nb = 0;
     //désactive la vue du sous ensemble
    dynamic_cast<POPMUSIC::SolutionTabu*>(this->solution)->setSubSetActive(false);
    for(int i = 0; i < int(this->solution->size()); i++)
        if(Tabu::OverlapManager::overlap(i, false, -1) == 1) nb++;

    return nb;
}

//#################//
int ns::overlap(int numLabel, bool number, int position = -1) const
{
    bool subSetActive = dynamic_cast<POPMUSIC::SolutionTabu*>(this->solution)->subSetIsActive();

     //désactive la vue du sous ensemble
    dynamic_cast<POPMUSIC::SolutionTabu*>(this->solution)->setSubSetActive(false);

    int nb = Tabu::OverlapManager::overlap(subSetActive ?
        this->subSet[numLabel] : numLabel, number, position);

    //active la vue du sous ensemble
    dynamic_cast<POPMUSIC::SolutionTabu*>(this->solution)->setSubSetActive(subSetActive);

    return nb;
}

//#################//
void ns::setSubSetSize(int size)
{
    //if(this->subSetSize != 0) delete(this->subSet);,/m

    this->subSetSizeMax = size;
    this->subSet = new int[this->subSetSizeMax];
    this->subSetSize = new int;
    this->oldSolution = new int[this->solution->size()];

    static_cast<POPMUSIC::SolutionTabu*>
        (this->solution)->setSubSet(this->subSet, this->subSetSize);
}

//#################//
void ns::selectSubSet(int firstPoint)
{
    //permet de mémoriser les points qui ont déjà été pris dans le sous ensemble
    bool pointTaken[this->solution->size()];
    for(int i = 0; i<int(this->solution->size()); i++)
        pointTaken[i] = false;
    pointTaken[firstPoint] = true;

    //compteurs
    int c1 = 1; //le nombre de point ajouté dans le niveau courant
    *this->subSetSize = 1; //le nombre de point total du sous-ensemble

    //le premier point est initialement introduit dans le sous-ensemble
    this->subSet[0] = firstPoint;

    //pour savoir quand faut il arreter la construction du sous-ensemble
```

```
    bool goOut = false;

    //parcours l'arbre des points en largeur
    while(!goOut)
    {
        //parcours les c1 derniers points ajoutés à la liste
        int tmp = c1;
        c1 = 0;
        for(int i = *this->subSetSize-1; i > *this->subSetSize-tmp-1 && !goOut; i--)
        {
            //on parcours les positions du point courant
            for(int pos = 0; pos < this->placementDirectives.getNbPosition()  && !goOut; pos++)
            {
                //on parcours les points pouvant entrer en conflit
                for(element::const_iterator potentialConflictPoint =
                    this->overlaps[this->subSet[i]][pos].begin();
                    potentialConflictPoint != this->overlaps[this->subSet[i]][pos].end() && !goOut;
                    potentialConflictPoint++)
                {
                    //si le point n'a pas déjà été prit alors on l'ajoute à la liste
                    if(!pointTaken[potentialConflictPoint->first])
                    {
                        pointTaken[potentialConflictPoint->first] = true;
                        this->subSet[*this->subSetSize] = potentialConflictPoint->first;
                        c1++;
                        (*this->subSetSize)++;

                        //si la taille du sous-ensemble maximale
                        //à été atteinte alors arrete sort des boucles
                        if(*this->subSetSize == this->subSetSizeMax) goOut = true;
                    }
                }
            }
        }
        //si aucun voisin n'a pu être trouvé alors arrête l'algorithme
        if(c1 == 0) goOut = true;
    }
}

//#################//
void ns::saveSolution()
{
    for(int i=0; i<*this->subSetSize; i++)
        this->oldSolution[this->subSet[i]] = (*this->solution)[i];
}

//#################//
void ns::rollbackToSavedSolution(set<int>& O, int version)
{
    for(int i=0; i<*this->subSetSize; i++)
    {
        (*this->solution)[this->subSet[i]] = this->oldSolution[this->subSet[i]];

        //si la version est égal à 2 alors on insert tout le sous-ensemble dans O
        if(version == 2) O.insert(this->subSet[i]);
    }
}

//#################//
int ns::getSubSetSize() const
{
    return *this->subSetSize;
}

#undef ns
```

C.30 batch/tabu_recherche_dichotomique_des_parametres/batch.rb

Recherche des meilleurs paramètres pour "Tabou".

```
#!/usr/bin/ruby

=begin
Effectue une recherche dichotomique des èparamtres
pour éamliorer l'éefficacit de la recherche avec Tabou
=end

##########DONNEES##########

#le nombre d'éitration
$nbIteration = 70

#le nombre dé'tiquette
$nbLabel = 40

params = [
#la taille de la la liste de candidats initiale
[0.1, 0.8],

#le facteur de érduction de la taille de la liste des candidats
[1.1, 1.5],

#le facteur d'agrandissement de la taille de la liste des candidats
[5.0, 20.0],

#la taille de la liste tabou par rapport au nombre de chevauchement
[0.1, 0.8],

#la nombre d'iteration avant de recalculer la taille des listes
[10.0, 50.0],

#la taille minimale de la liste de candidats
[2.0, 20.0],

#la taille minimal de la liste tabou
[2.0, 10.0]]


#Le nombre de configuration de testà  chaque éitration
$nbConfiguration = 500

#Le taux de édensit édfinissant la taille de la surface, écalcul comme ceci :
# size = sqrt(nombre_de_label)*édensit
$densityArray = [2, 2.5, 3]

#le nombre d'éitration de recherche
nbIterationSearch = 200

##########FIN DONNEES##########

#pourcentage de édplacement de la valeur
$diplacePercent = 0.2 #20%

=begin
effectue une ésrie de calculs
=end
def runAComputation(candidateSizeBase,
candidateReduceFactor, candidateGrowingFactor, tabuSize,
nbIterationListSizeRecompute, minimalCandidateListSize, minimalTabuListSize, seedBase)

    nbOverlapInitial = 0.0
    nbOverlapFinal = 0.0
    time = 0.0

    seed = seedBase
    $nbConfiguration.times{

        $densityArray.each{|density|
            size = (Math::sqrt($nbLabel)*density).to_i

            #permet d'avoir des configuration édiffrentesà  chaque essais
            #mais des ensembles de configuration identiques
            seed += 1

            resultats = `../../cartographic_label_placement -i #{$nbIteration}\
            -n #{$nbLabel} -s #{size} #{size} -e #{seed} -v #{candidateSizeBase}\
            #{candidateReduceFactor} #{candidateGrowingFactor} #{tabuSize}\
            #{nbIterationListSizeRecompute.to_i} #{minimalCandidateListSize.to_i}\
            #{minimalTabuListSize.to_i}`
```

```
80
          /Number of initial overlaping : (\d+).*Number of overlaping\
   : (\d+).*Time of computation : (\d+)/m =~ resultats

          nbOverlapInitial += $1.to_f
85        nbOverlapFinal += $2.to_f
          time += $3.to_f
       }
    }
    nbOverlapInitial /= $densityArray.size + $nbConfiguration
90  nbOverlapFinal /= $densityArray.size + $nbConfiguration
    time /= $densityArray.size + $nbConfiguration

    [nbOverlapFinal, time]
end
95

#renvois la moyenne entre 2 valeurs se trouvant dans un tableau
def average(values)
    (values[0]+values[1])/2.0
100 end


nbIterationSearch.times{|i|

105   #si la édiffrence entre les 2 valeurs est trop faible alors n'essais pas de les éamliorer :
      next if params[i % 7][0] + 0.02 > params[i % 7][1]

      results = Array::new

110   #un nombre entre 0 et 100000000
      seed = rand(100000000)

      #effectue 2 ésries, utilisant respectivement la èpremire
      #valeur du èparamtre courant et la èdeuxime valeur
115   2.times{|j|

         results[j] = runAComputation(
            ((i + 7) % 7 == 0 ? params[0][j] : average(params[0])),
            ((i + 6) % 7 == 0 ? params[1][j] : average(params[1])),
120         ((i + 5) % 7 == 0 ? params[2][j] : average(params[2])),
            ((i + 4) % 7 == 0 ? params[3][j] : average(params[3])),
            ((i + 3) % 7 == 0 ? params[4][j] : average(params[4])),
            ((i + 2) % 7 == 0 ? params[5][j] : average(params[5])),
            ((i + 1) % 7 == 0 ? params[6][j] : average(params[6])), seed)
125   }

      #compare les deux érsultats (en terme de chevauchement)
      if results[0][0] < results[1][0]
         #si la èpremire valeurà  donner un meilleur érsultat alors
130      #remplace la èdeuxime par une pond?ration de n% de la èpremire et de 100-n% de la èdeuxime
         params[i % 7][1] = params[i % 7][0]*$diplacePercent+params[i % 7][1]*(1.0-$diplacePercent)
      elsif results[0][0] > results[1][0]
         #si la èdeuxime valeurà  donner un meilleur érsultat alors
         #remplace la èdeuxime par une épondration de 100-n% de la èpremire et de n% de la deuxi?me
135      params[i % 7][0] = params[i % 7][1]*$diplacePercent+params[i % 7][0]*(1.0-$diplacePercent)
      end

      #affiche le érsultat
      puts "------#{i%7}------" #le énumro du èparamtre étest
140   puts "iteration : #{i}" #le énumro de l'éitration
      puts "#{format("%.4f", results[0][0])} <=> #{format("%.4f", results[1][0])}"
      puts 'Parms : '
      params.each{|p| #les èparamtres
         puts "#{format("%.4f", p[0])} #{format("%.4f", p[1])}"
145   }
}
```

C.31 batch/mesure_complexite_tabu_popmusic/batch.rb

Mesure de la complexité de la recherche avec tabous, de POPMUSIC ainsi que de la phase d'initialisation.

```
#!/usr/bin/ruby

=begin
Effectue plusieurs érsolution de èproblme en faisant varier la taille du èproblme deà
50  6400 par facteur de 2 sur les 2 algorithmes : tabou et popmusic.
un fichier par algo est écre, il se éprsente sous la forme de 2 colonnes :
<taille du èproblme> <temps de calcul>.

Effectueé galement une mesure du temps de l'initialisation.
=end

#les èparamtres pour tabou et popmusic
#on ne tient pas compte du temps d'initialisation
algosParams =
[["tabu.txt",      "-ti 0 -p 0       -i 5000 -r 1 -l 0 "],
 ["popmusic1.txt", "-ti 0 -p 1 -u 70 -i   75 -r 1 -l 0 -v 0.73 1.3 15 0.77 47 18 9 "],
 ["popmusic2.txt", "-ti 0 -p 2 -u 70 -i   75 -r 1 -l 0 -v 0.73 1.3 15 0.77 47 18 9 "],
 #pour mesurer le temps d'initialisation (-ti 2)
 ["ini.txt",       "-ti 2 -p 0       -i 1 -r 1 -l 0 "]]

#le nombre de èproblme (en fait la moyenne)
nbSeries = 10

#pour chaque algorithme
algosParams.each{|params|

   #écre le fichier
   file = File::new(params[0], 'w')

   #pour plusieurs tailles de èproblme
   size = 50
   while(size <= 6400)

      time = 0; #comptabilise le temps

      nbSeries.times{|serie|

         #affiche le calcul courant :
         puts "> #{params[0]} - #{size}"

         #la taille de la zone (écalcul pour garder la êmme édensit en fonction du nombre de label)
         zoneSize = (Math::sqrt(size)*2).to_i

         puts "../../cartographic_label_placement #{params[1]}" +
            " -s #{zoneSize} #{zoneSize} -n #{size} -e #{serie}"

         resultats = `../../cartographic_label_placement #{params[1]}\
 -s #{zoneSize} #{zoneSize} -n #{size} -e #{serie}`

         /Time of computation : (\d+)/ =~ resultats

         time += $1.to_f
      }


      file.puts(format("%15i%15.2f", size, time/nbSeries.to_f))

      size *= 2
   end
}
```

C.32 batch/pop_mesure_de_qualite_sur_des_problemes_de_la_litterature/batch.sh

```
#!/bin/bash

#applique POPMUSIC+TABUà  tous les èproblmes
#de la taille qui est édonne en
#éparamtre (les èproblmes se trouve dans
#le érpertoire 'data').

#évrifie qu'il y ait bien 3 èparamtres
if (( $# != 3 )); then
    echo "Usage : batch.sh <POPMUSIC version : 1|2> <size of the subsets><file num>"
    exit
fi

#écre le fichier ùo seront éstock les érsultats
echo -n '' > batch.txt

#le édbut du nom du fichier de édonne
fichier="../../data/$3/d$3_"

#compteur de boucle
nb=1;

#pour chaque fichier
while (( $nb <=  25 )); do

    #ècomplte le numero du fichier pour qu'il soit toujours sur 2 digit
    if (($nb < 10)); then
        zero='0'
    else
        zero=''
    fi

   echo "fichier numero : $zero$nb"

   echo "fichier :  $fichier$zero$nb.dat" >> batch.txt


    #####lancement du calcul
    #(-p 1):activation de POPMUSIC
    #(-f ):le fichier de édonnes
    #(-i 70):le nombre d'iteration
    #(-u 75):la taille des sous-èproblmes
    #(-d 1):le germe (pour choisir les sous-ensembles)
    #(-v ):les èparamtres
    #(-r 1):pour ne pas tenir compte des positions de labels
    ../../cartographic_label_placement\
    -p $1\
    -f $fichier$zero$nb.dat\
    -i 70\
    -u $2\
    -d 1\
    -v 0.73 1.3 15 0.77 47 18 9\
    -r 1 >> batch.txt || exit
    #####

    #éincrment le compteur
    nb=$(($nb + 1))
done

#calcul la moyenne du nombre de chevauchement ainsi
#que du temps de calcul de la meilleure solution
#et enregistre le érsultats dans 'averages.txt'
awk '
BEGIN {
nbOverlap = 0;
nbPoint = 0;

bestSolution = 99999;
bestOk = 0;
computeTime = 0;
}
(NR-13) % 15 == 0{
    nbOverlap += $5;
    nbPoint += 1;

    if(nbOverlap < bestSolution)
    {
        bestSolution = nbOverlap;
        bestOk = 1;
    }
}
```

```
(NR-14) % 15 == 0{
    if(bestOk)
    {
        computeTime = $5;
        bestOk = 0;
    }
}
END {
print "Nombre de chevauchement moyen       : ", nbOverlap/nbPoint;
print "Temps de la meilleure solution [ms] : ", computeTime
}
' batch.txt >> averages.txt
```

D Cahier des charges

Dans un projet de semestre, l'étudiant a programmé une première version d'une recherche avec tabous pour le placement, à l'horizontale, d'étiquettes rectangulaires dont les poids étaient identiques. Ce programme devra dans un premier temps être amélioré, de façon à diminuer sa complexité, à le rendre plus robuste et à pouvoir traiter des placements plus généraux.
La recherche ainsi définie devra être incorporée dans une technique de type "popmusic" afin de pouvoir placer avec un temps de calcul limité des étiquettes sur des plans comprenant un grand nombre d'objets. L'efficacité du programme devra être comparée avec d'autres méthodes de la littérature. Des généralisations du problème (étiquette de poids différenciés, non rectangulair, etc.) seront abordées si le temps le permet.[1]

[1] Repris tel quel du document officiel