Application d'un algorithme de colonie de fourmis au problème du voyageur de commerce

par Pierre Schwartz (retour aux articles)

Date de publication : 1er juillet 2008

Dernière mise à jour :

Le but de cet article est d'illustrer l'implémentation d'un algorithme de colonie de fourmis sur un exemple concret de recherche opérationnelle : le problème NP complet dit du voyageur de commerce. Prérequis : programmation objet, C++

I - Introduction

Cet article peut être vu comme une suite à celui sur la résolution du voyageur de commerce par un algorithme génétique. Le problème considéré est le même : trouver le plus court chemin hamiltonien sur un graphe donné.

Le problème du voyageur de commerce se prête très bien à l'utilisation de méta-heuristiques comme les algorithmes génétiques ou les colonies de fourmis. Chacune de ces méthodes a ses points forts et ses inconvénients, le but de cet article n'est pas de montrer que telle méthode est meilleure que telle autre, il s'agit juste de montrer les possibilités de chacune d'elles.

II - Principe d'un algorithme de colonies de fourmis

Un algorithme de colonies de fourmis est un algorithme itératif à population où tous les individus partagent un savoir commun qui leur permet de guider leurs futurs choix et d'indiquer aux autres individus des directions à suivre ou au contraire à éviter.

Fortement inspiré du déplacement des groupes de fourmis, cette méthode a pour but de construire les meilleures solutions à partir des éléments qui ont été explorés par d'autres individus. Chaque fois qu'un individu découvre une solution au problème, bonne ou mauvaise, il enrichit la connaissance collective de la colonie. Ainsi, chaque fois qu'un nouvel individu aura à faire des choix, il pourra s'appuyer sur la connaissance collective pour pondérer ses choix.

Pour reprendre la dénomination naturelle, les individus sont des fourmis qui vont se déplacer à la recherche de solutions et qui vont sécréter des phéromones pour indiquer à leurs congénères si un chemin est intéressant ou non. Si un chemin se retrouve fortement phéromoné, cela signifiera que beaucoup de fourmis l'ont jugé comme faisant partie d'une solution intéressante et que les fourmis suivantes devront la considérer avec intérêt.

Un risque apparaît lorsqu'un chemin non optimal est marqué. En effet, les fourmis qui s'en trouveront à proximité seront tentées d'y passer augmentant encore le niveau de phéromone de ce chemin. Pour diminuer le risque d'enfoncer la colonie dans un minimum local du problème, on pourra prendre soin de diminuer automatiquement le niveau de phéromone de tout le système, pour réhausser l'intérêt des autres chemins qui pourraient faire partie de la solution optimale. Ce paramètre, correspondant au taux d'évaporation des phéromone, est l'un des paramètres principaux de l'algorithme.

De la même manière, aucun chemin ne devra être innondé de phéromones et aucun chemin ne devra être totalement invisible, on pourra donc aussi contrôler le niveau de phéromone de chaque chemin pour le maintenir entre des bornes minimum et maximum. Un chemin innondé de phéromones masquerait tous les autres à proximité et un chemin pas du tout phéromoné ne serait jamais choisi par une fourmi, en conséquence nous devons conserver ces chemins avec des valeurs raisonnables. Ces bornes min et max sont aussi des paramètres de l'algorithme.

III - Déroulement global de l'algorithme

III-1 - Itérations et déplacement des fourmis

Nous n'avons pas de notion de générations de population, les itérations de l'algorithme correspondent aux déplacements des fourmis. Pour aller d'un n#ud du graphe à un autre, chaque fourmi aura besoin d'un nombre d'itérations dépendant de la taille de l'arc de graphe à parcourir. Ce mode d'itérations va aussi privilégier les plus courts chemins puisque les fourmis auront besoin de moins d'itérations pour en arriver au bout.

III-2 - Vie d'une fourmi

Chaque fourmi doit connaître la liste des n#uds qu'elle a déjà parcouru et les n#uds encore à parcourir. De plus elle doit mesurer le temps qu'elle passe sur la solution qu'elle explore. À chaque n#ud, la fourmi va étudier les arcs possibles en observant leurs niveaux de phéromone respectifs. Elle n'a ensuite qu'à choisir au hasard, en privilégiant les arcs fortement phéromonés. Une fois arrivée à destination, la fourmi connaît la longueur totale de la solution qu'elle a trouvé, elle peut refaire le chemin en sens inverse pour marquer le chemin avec ses phéromones et enrichir la connaissance collective de la colonie.

```
class ant{
public:
    ant(problem&);    // on donne à chaque fourmi les données du problème et la connaissance collective

    std::vector<int> visitedCities;        // toutes les villes visitées par la fourmi
    std::vector<int> citiesStillToVisit;    // toutes les villes encore à visiter

    long tmpVisitedLength;                // compteur de longueur du chemin parcouru

    enum {
        SEARCHING_PATH,
        RETURNING,
        NOTHING,
    };
    int state;                            // état de la fourmi, en route, en retour ..

    void frame();                    // faire évoluer la fourmi à chaque itération

protected:
    // données de parcours locales
    long currentArcSize;    // longueur de l'arc actuellement parcouru
    long currentArcPos;    // position sur l'arc actuellement parcouru
    int currentOrigin;        // première extrémité de l'arc actuellement parcouru
    int currentDestination;    // seconde extrémité de l'arc actuellement parcouru

    problem& data;
  // référence sur les données du problème et sur la connaisasnce collective

    void findNextSearchDestination();   // détermination du prochain n#ud à atteindre

    int getNearCity(int);        // choix pondéré de n#ud
};
```

Une conception objet nous permet de bien cloisonner chaque fourmi. Chacune décidera elle-même son trajet, en consultant la connaissance collective. À chaque itération, le moteur d'algorithme appelera la fonction frame() de chaque fourmi. Les fourmis situées sur un arc se contenteront d'avancer et celles sur un n#ud choisiront le n#ud suivant.

J'ai défini 3 états pour les fourmis :

- juste créée et à la recherche de son premier n#ud
- en train de chercher une solution, donc déjà engagée dans le graphe
- en train de revenir vers le point de départ et de marquer les chemin avec des phéromones

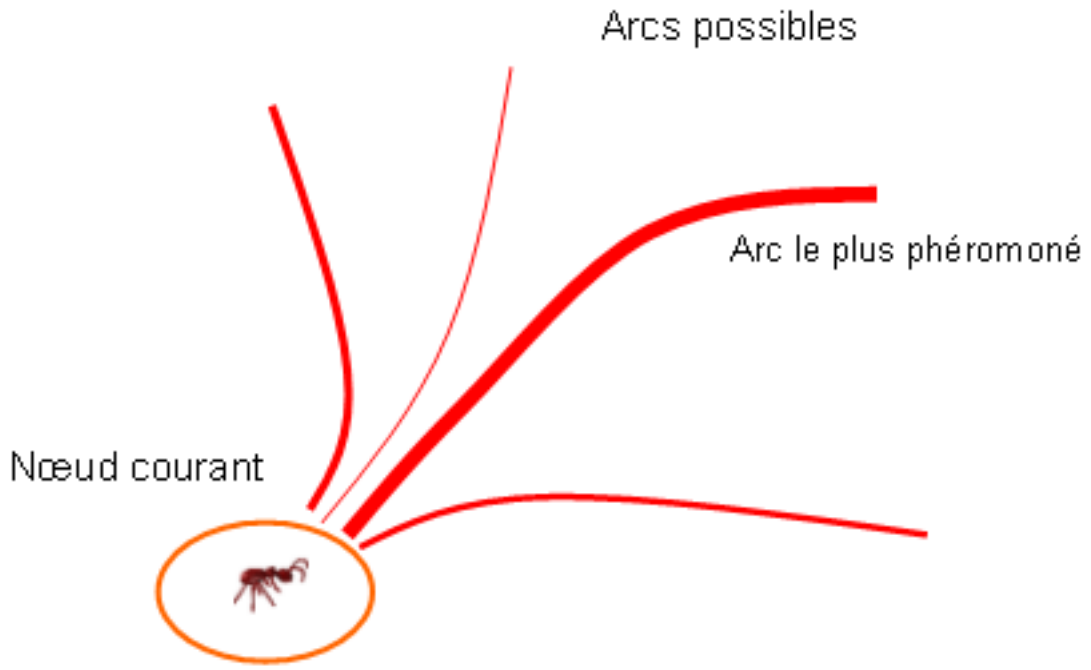
Toutes les fourmis sont créées avec le premier état et elles passent d'un état à l'autre en fonction de leur avancement respectif, spécifié dans la fonction frame.

```
void ant::frame(){
    switch(state){
        case SEARCHING_PATH:
            tmpVisitedLength ++;
        case RETURNING:
            currentArcPos++;
            if (currentArcPos >= currentArcSize)
                findNextSearchDestination();
            break;
        case NOTHING:
            findNextSearchDestination();
            break;
    }
}
```

La fonction qui va contenir le gros du traitement est findNextSearchDestination(). Elle se charge de donner à chaque fourmi une destination à suivre qu'elle soit en recherche ou en train de revenir :

```
void ant::findNextSearchDestination(){
    switch(state){
     // si la fourmi vient d'être créée
        case NOTHING:{
            visitedCities.push_back(0);
            std::vector<int>::iterator tmp = citiesStillToVisit.begin();
            while (tmp != citiesStillToVisit.end()){
                if (*tmp == 0){
                    citiesStillToVisit.erase(tmp);
                    break;
                }
                tmp++;
            }

            int dest = getNearCity(0);
            state = SEARCHING_PATH;
            currentOrigin = 0;
            currentDestination = dest;
            currentArcPos = 0;
            currentArcSize = data.distances[0][currentDestination];

            break;
        }
        // si la fourmi cherche son chemin dans le graphe
        case SEARCHING_PATH:{
            // on a atteint currentDestination
            tmpVisitedLength += data.distances[currentOrigin][currentDestination];
            visitedCities.push_back( currentDestination );

            std::vector<int>::iterator tmp = citiesStillToVisit.begin();
            while (tmp != citiesStillToVisit.end()){
                if (*tmp == currentDestination){
                    citiesStillToVisit.erase(tmp);
                    break;
                }
                tmp++;
            }

            if (citiesStillToVisit.size() == 0){
                // plus rien à visiter, le chemin est complet
```

```
                // on revient vers le nid
                tmpVisitedLength += data.distances[currentDestination][0];

                state = RETURNING;
                currentOrigin =  int(visitedCities.size())-1;
                currentDestination = int(visitedCities.size())-2;
                currentArcSize = data.distances[ visitedCities[currentOrigin] ]
[ visitedCities[currentDestination] ];
                currentArcPos = currentArcSize;
                return;
            }

            int dest = getNearCity(currentDestination);
            currentOrigin = currentDestination;
            currentDestination = dest;
            currentArcSize = data.distances[currentOrigin][currentDestination];
            currentArcPos = 0;
            break;
        }
        // si la fourmi revient au nid
        case RETURNING:{
            if (currentDestination == 0){
                // retourné au nid avec succès
                data.setPheromones(visitedCities[currentOrigin], visitedCities[currentDestination],
 tmpVisitedLength);

                // sauver le résultat, changer de fourmi
                antException e;
                e.a = this;
                e.state = antException::TO_REGISTER;
                throw e;
            }

            // trouver la ville précédemment visitée et la passer en destination
            // mettre des phéromones sur l'arc parcouru
            data.setPheromones(visitedCities[currentOrigin], visitedCities[currentDestination],
 tmpVisitedLength);
            currentOrigin = currentDestination;
            currentDestination = currentOrigin-1;
            currentArcSize = data.distances[  visitedCities[currentOrigin ] ][
  visitedCities[currentDestination ] ];
            currentArcPos = currentArcSize;

            break;
        }
    }
}
```



Pour passer d'une ville à l'autre, une fourmi doit analyser les choix possibles. Le choix des villes suivantes en tenant compte de la connaissance collective est une simple roulette aléatoire :

```
int ant::getNearCity(int from){
    // roulette sur les chemins restants, pondérés par les phéromones
    float pheromoneSize = 0;
    for (int i = 0; i < int(citiesStillToVisit.size()); i++){
        if (citiesStillToVisit[i] == from)
            continue;
        pheromoneSize  += data.pheromones[from][ citiesStillToVisit[i] ];
    }

    float found = float(rand()%int(pheromoneSize*1000))/float(1000)  ;
    float tmpPheromones = 0;
    int ii = 0;
    while (ii < int(citiesStillToVisit.size())){
        if (citiesStillToVisit[ii] == from){
            ii++;
            continue;
        }

        tmpPheromones  += data.pheromones[currentDestination][ citiesStillToVisit[ii] ];

        if (tmpPheromones   > found)
            break;

        ii ++;
    }
    if (ii == citiesStillToVisit.size()){
        // aucune solution acceptable, détruire la fourmi courante
        antException e;
        e.a = this;
        e.state = antException::TO_DELETE;
        throw e;
    }

    return citiesStillToVisit[ii];
}
```

Chaque fois qu'une fourmi est dans une situation particulière (impasse ou solution trouvée), elle en informe le moteur de l'algorithme en levant une exception.

III-3 - Le dépôt de phéromones

L'heuristique de dépôt de phéromones peut grandement modifier le mode de convergence de l'algorithme. D'un point de vue pûrement naïf, on peut tout à fait déposer la même quantité de phéromones sur chaque chemin. Les fourmis engagées dans des chemins longs iront déposer moins de phéromones puisqu'elles pourront essayer moins de chemins. Et au contraire, les fourmis engagées sur les plus courts chemins pourront très vite essayer d'autres chemins. Tout naturellement, les chemins les plus courts vont se retrouver plus phéromonés que les autres.

Cependant on peut aussi utiliser d'autres modes de dépôt de phéromones. Une idée intéressante est de déposer plus de phéromones à mesure que la solution testée est bonne. Une fonction décroissante basée sur la taille du chemin trouvé peut faire l'affaire. Et bien sûr, rien ne vous oblige à prendre une fonction linéaire, histoire de privilégier encore davantage les meilleurs chemins.

IV - Le moteur de l'algorithme

Le moteur est un simple conteneur de fourmis qui déroule les itérations en observant d'éventuelles exceptions venant des fourmis. Il analyse les exceptions levées et lorsqu'il s'agit d'une fourmi coincée dans une impasse, il détruit pûrement et simplement cette fourmi, et quand il s'agit d'une fourmi qui a trouvé un chemin, il note la solution dans un coin et créé une nouvelle fourmi pour remplacer celle qui vient de trouver un chemin.

```
void antSystem::run(int n){
    // pour chaque itération
    for (int i=0; i<n; i++){
        // pour chaque fourmi
        std::list<ant*>::iterator it = ants.begin();
        while (it != ants.end()){
            try{
                 (*it)->frame();
            }catch(antException &e){
                if (e.state == antException::TO_REGISTER)
                    notifySolution(e.a->tmpVisitedLength, e.a->visitedCities);

                if(bestLength <= data.optimalLength)
                     return;

                // on crée une nouvelle fourmi pour remplacer la fourmi courante
                *it = new ant(data);
                delete e.a;
            }
            it++;
        }

        // on évapore les phéromones toutes les 20 itérations
        // juste histoire de ne pas monopoliser toutes les ressources pour ça
        if (i % 20 == 0)
            data.evaporate();
    }
}
```

Le moteur de l'algorithme dépend de 2 paramètres :

- le nombre de fourmis à faire évoluer
- le nombre d'itérations à jouer

V - Les données du problème

Le problème contient simplement le graphe à explorer, ainsi que les phéromones de chaque arc. On peut aussi lui donner des paramètres comme les bornes min et max des phéromones à utiliser ainsi que le taux d'évaporation. C'est lui qui va contenir la connaissance collective de la colonie, sous la forme d'une matrice de phéromones.

```
class problem{
public:
    problem(int, float, float, float);

    void setPheromones(int, int, int);

    void evaporate();

    int nbCities;
    float borneMax, borneMin;
    float evaporation;

    int optimalLength;

    // arcs
    std::vector<std::vector<int> > distances;

    // pheromones
    std::vector<std::vector<float> > pheromones;
};
```

L'évaporation se contente de diminuer le taux de phéromones sur chacun des arcs. Vous pouvez bien sûr prendre le mode de calcul qu'il vous plaira. J'ai choisi un simple calcul de pourcentage :

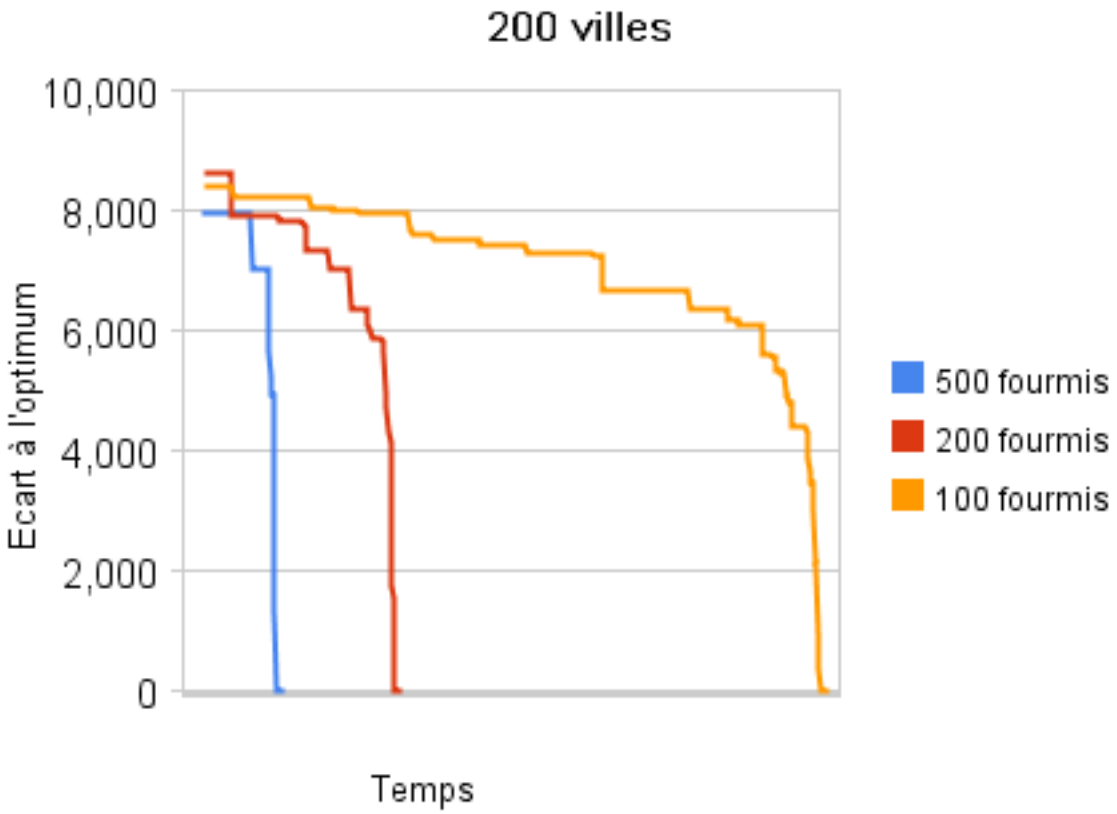
```
void problem::evaporate(){
    for (int i=0; i<nbCities; i++)
        for (int j=0; j<i; j++){
            pheromones[i][j] = float(pheromones[i][j]*(100-evaporation) /100);
            if (int(pheromones[i][j]) < borneMin)
                pheromones[i][j] = borneMin;

            pheromones[j][i] = pheromones[i][j];
        }
}
```

VI - Résultats

Le programmeur a la main sur plusieurs paramètres pour affiner la recherche de solutions :

- Le nombre de fourmis
- Les bornes max et min
- Le taux d'évaporation des phéromones

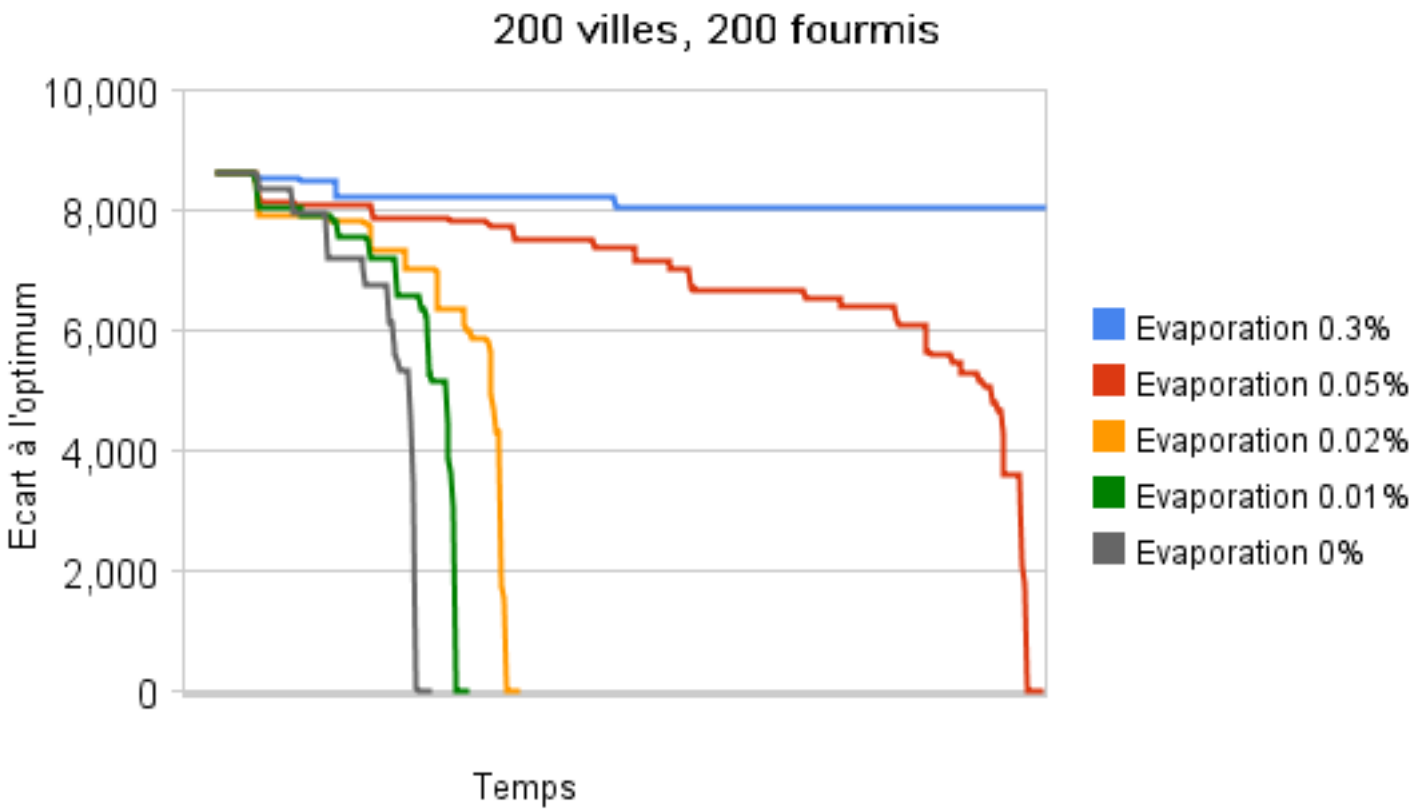
J'ai effectué des mesures pour des résolutions sur 50, 100 et 200 villes. Pour chaque exécution, les premiers résultats ne sont disponibles que lorsque la première fourmi a fini son cycle. De même, les données n'évoluent que lorsqu'une fourmi a fini son trajet et a trouvé une meilleure solution. Toutes les exécutions ont convergé vers la solution optimale plus ou moins vite. Voici les résultats pour la résolution à 200 villes, en jouant sur le nombre de fourmis et sur le coefficient d'évaporation.



Les exécutions avec 100, 200 et 500 fourmis à un taux d'évaporation fixe ont toutes convergé et toujours de la même manière. Les premières valeurs apparaissent dès que la première fourmi a trouvé une solution et les solutions n'évoluent que dès que d'autres fourmis ont trouvé une meilleure solution (d'où la stagnation dès le début des valeurs) et la convergence accélère au fur et à mesure de l'exécution. En effet, chaque fois qu'un arc du graphe est très fortement phéromoné, il sera sélectionné par toutes les fourmis et donc la taille de l'espace de recherche diminuera. La tendance s'accélère à chaque fois qu'un arc est trouvé, jusqu'à arriver à la solution optimale.

De la même manière, avec davantage de fourmis, les phéromones ont moins le temps de s'évaporer, la convergence est plus rapide.

Voici maintenant le diagramme de convergence d'une exécution sur 200 villes, avec 200 fourmis, en jouant sur le coefficient d'évaporation.



Cette fois ci, les résultats sont moins nets. Un taux d'évaporation trop faible ou trop élevé pénalise l'algorithme. Si le taux est trop faible, les phéromones s'accumulent et on prend le risque de s'enfermer dans un minimum local. Avec un taux trop élevé, les fourmis n'ont pas le temps de profiter des informations collectées par leurs cons#urs que déjà les phéromones sont évaporées. Ceci revient exactement à supprimer la connaissance collective, qui est justement la force de la méthode des colonies de fourmis. L'algorithme ne converge pas ou alors très très lentement.

Le nombre de fourmis à prendre et le taux d'évaporation dépendent en fait de la taille du réseau à explorer. Le taux d'évaporation doit être suffisamment faible pour que l'information sur un arc du graphe puisse resservir à une autre fourmi, mais doit être suffisamment important pour éviter les minima locaux. Pour un même réseau, on peut diminuer le nombre de fourmis si on augmente la durée de vie des phéromones (donc si on diminue leur taux d'évaporation). De même, on peut prendre un réseau plus grand avec le même nombre de fourmis si on augmente aussi la durée de vie des phéromones.

La fonction de calcul des phéromones en fonction de la longueur de chaque chemin doit aussi dépendre de la taille du réseau et des bornes min et max à utiliser. Plus le réseau est grand, plus il faudra mettre en avant les arcs intéressants et donc plus la borne max devra être importante et le taux d'évaporation faible. C'est peut-être utopique, mais en creusant le problème, on devrait pouvoir arriver à déterminer automatiquement la valeur de ces paramètres à partir du seul nombre de villes du graphe.

VII - Conclusion

En fournissant des très bons résultats les colonies s'avèrent de bons outils pour résoudre ce genre de problèmes. Le partage des données sur les phéromones est le point fort de cette technique qui peut s'apparenter à de l'intelligence artificielle distribuée où chaque agent vient enrichir la connaissance collective .Le tuning des paramètres reste le point le plus délicat tant il est facile de ne plus converger ou d'exploser le temps de convergence.

Télécharger les sources C++ : **implementation.zip (7 Ko)**

Merci à toute l'équipe de la rubrique Algo et en particulier à **buchs** pour la relecture orthographique.