18/12/2013

Compte rendu
TP2 Algorithmique
INFO521

Université de Savoie
Licence 3 STIC Info
Mickaël Koza
Céline de Roland

Introduction

Pour réaliser l'algorithme des fourmis, nous avons commencé par l'interface graphique, car cela nous semblait utile pour visualiser la progression de l'algorithme par la suite.

Ensuite nous avons travaillé sur le modèle : nous nous sommes basés sur les formules données en cours.

Enfin, nous avons créé la fonction meilleur chemin et la représentation graphique associée, paramétré le critère de convergence, créé divers jeux de tests, amélioré la qualité de notre code et corrigé les bugs qui restaient.

Partie 1 : l'interface graphique.

Notre interface est composé d'une fenêtre contenant un panneau principal, lui même découpé en 3 parties :

1) Le panneau supérieur contient simplement un titre

2) Le panneau central (p_figure) contient le graphique représentant les villes et les chemins

3) Le panneau inférieur contient les champs et boutons permettant d'ajouter de nouvelles villes, de choisir le nombre de fourmis, et de lancer des cycles.

Le panneau principal fait office de contrôleur principal : il reçoit les actions de l'utilisateur du panneau inférieur, et les renvoie au panneau central.

Le panneau central gère l'affichage du graphique et fait également office de contrôleur : c'est lui qui communique avec le modèle pour ajouter des villes, des fourmis, et faire des cycles.

Partie 2 : modèle

Nous avons identifié 3 objets intervenant dans cet algorithme : Les villes, les fourmis, et le réseau (le graphe des villes).

Les villes

Une ville possède un nom et deux coordonnées, puis des getters et setters sur chacun de ces attributs.

Le réseau

Les attributs du réseau sont :

```
private double arretes[][];
private Ville villes[];
private int nbVilles = 0;
private int meilleur[];
private int cpt = 0;
private double distance_meilleure;
```

Les arrêtes sont représentées par une matrice symétrique de diagonale nulle. Le nombre situé aux coordonnées (i,j) représente l'épaisseur de phéromone sur l'arrête reliant la ville i et la ville j (c'est pourquoi elle est symétrique).

Le tableau meilleur contient les numéros des villes, dans l'ordre où il faut les parcourir pour le chemin optimal (en commençant toujours par la ville 0 : la ville de départ n'a pas d'importance puisqu'on revient au point de départ à la fin).

Les variables cpt et distance_meilleure constituent le critère de convergence que nous verrons en partie 4.

Les méthodes de réseau sont :

```
public void afficher()
public Reseau()
public double arrete(int i, int j)
public int getNbVilles()
public Ville ville(int i)
public void ajoutVille(int x, int y, String nom)
public double distance(int i, int j)
public double lg_parcours(int[] parcours)
public double probaVille(int i, int j)
public void deposer_pheromones(int[] parcours)
public void evaporer()
public int meilleur_chemin()
public Ville getMeilleur(int i)
```

La fonction afficher sert au débuggage.

Réseau initialise les tableaux arretes et villes à 50 cases (c'est le maximum de villes que nous avons fixé).

Les fonctions ville, getNbVilles et arrete sont des getters.

Distance et lg_parcours nous serons utiles lorsqu'on aura besoin de calculer la distance entre deux villes i et j, et pour calculer la longueur d'un parcours de fourmis (en donnant les numéros des villes visitées dans l'ordre dans un tableau)

Les fonctions vraiment intéressantes pour l'algorithme des fourmis sont probaVille, deposer_pheromones et evaporer.

Evaporer est simple, elle mulltiplie tous les coefficients de la matrice arrete par 0,7.

```
public void evaporer()
{
	for (int i = 0; i < this.nbVilles; i++)
	{
		for (int j = 0; j < this.nbVilles; j++)
		{
			this.arretes[i][j] *= (double)0.7;
		}
	}
}
```

probaVille calcule le numérateur de la formule donnée en cours pour la probabilité d'aller à la ville j lorsqu'on se trouve à la ville i. En effet, la question se pose de savoir si ce calcul de probabilité incombe au réseau ou à la fourmis. Le réseau connaît la distance entre les villes et l'épaisseur de phéromone des arrêtes, tandis que la fourmis sait sur quelle ville elle se trouve et quelles villes elle a déjà visité.

Parmi les solutions possibles, nous avons choisi la suivante : le réseau calcule les « briques » nécessaires au calcul de probabilité (les expressions de la forme (epaisseur_pheromone(i,j) / distance(i,j)), et la fourmis utilise ces « briques » pour calculer les probabilités qui la concernent à chaque étape de son parcours.

```
public double probaVille(int i, int j)
{
	return (this.arretes[i][j]/this.distance(i,j));
}
```

deposer_pheromone est appelée par la fourmis à la fin de son parcours. Il s'agit de calculer l'épaisseur de phéromone (1 / longueur_parcours). Ici aussi la fourmis connaît son parcours tandis que le réseau connaît les distances entre les villes. Puisque nous avons choisi précédemment de laisser la fourmis utiliser les fonctions du réseau (et pas l'inverse), la solution s'imposait.

```
public void deposer_pheromones(int[] parcours)
{
	for (int i = 0; i < this.nbVilles - 1; i++)
	{
		this.arretes[parcours[i]][parcours[i+1]] +=
1/this.lg_parcours(parcours);
		this.arretes[parcours[i+1]][parcours[i]] =
this.arretes[parcours[i]][parcours[i+1]];
	}
}
```

Les fourmis

Les attributs de l'objet fourmis sont :

```
	private double[] villes_visitees = new double[50];
	private int nbVilles_visitees;
	private int nbVilles;
	private int ville_actuelle;
	private int ville_initiale;
	private int[] parcours;
```

Parcours contient les numéros des villes par lesquelles est passée la fourmis, dans l'ordre. C'est lui qui sera passé en paramètre de la fonction deposer_pheromone à la fin d'un cycle fourmis.

Villes_visitees contient 0 aux indices correspondants aux villes déjà visitées, la probabilité cumulée d'aller dans cette ville sinon. Il y a ici un non sens conceptuel puisque la probabilité cumulée d'une ville déjà visitée devrait être égale à la probabilité cumulée de la ville précédente et non à 0. Nous l'avons cependant laissé car cela nous permet de savoir immédiatement qu'une ville a déjà été visitée sans devoir parcourir le tableau parcours ou créé un autre tableau.

Nous avons choisi d'enregistrer les **probabilités cumulées** plutôt que les probabilités simples car cela facilite l'opération de choix d'une ville : cela nous donne une série d'intervalles de [0;1], il suffit ensuite de choisir un nombre aléatoire entre 0 et 1, et de choisir la ville correspondant à l'intervalle dans lequel il se trouve.

Nous enregistrons le nombre de villes et le nombre de villes visitées afin de savoir quand s'arrêter.

Ville_actuelle nous permet de savoir où se trouve la fourmis à un instant t, et ville_initiale de boucler le parcours à la fin d'un cycle fourmis.

Les méthodes de l'objet fourmis sont :

```
	public void afficher()
	public Fourmis(Reseau res)
	public void initialiser(Reseau res)
	public void calculer_proba_ville(Reseau res)
	public boolean choisir_ville(Reseau res)
	public int getVilleActuelle()
```

Nous voyons ici en ce qui concerne les dépendances, qu'une fourmis a besoin d'un réseau en paramètre de la plupart de ces fonctions. Nous avons fait attention à ce que réseau n'est pas besoin des fourmis pour éviter la double dépendance.

Afficher sert pour le debuggage

Fourmis est le constructeur, et initialiser sert à repartir avec un parcours et des probabilités vierges à la fin d'un cycle. GetVilleAcutelle est un getter.

Calculer_proba_ville est le pendant de probaVille dans réseau. Ici on applique la formule donnée en cours en utilisant les briques fournies par le réseau, puis on calcule la probabilité cumulée.

```
public void calculer_proba_ville(Reseau res)
{
	//Calcul du dénominateur//
	double total = 0;
	for (int i = 0; i < this.nbVilles; i++)
	{
		if (this.villes_visitees[i] != 0)
		{
			total += res.probaVille(this.ville_actuelle, i);
		}
	}

	//calcul du numérateur et de la probabilité cumulée
	double cumul = 0;
	for (int i = 0; i < this.nbVilles; i++)
	{
		if (this.villes_visitees[i] != 0)
		{
			double proba = res.probaVille(this.ville_actuelle,
i)/total;
			if (proba < 0.001) { proba = (double)0.001; }
//magouille pour éviter que (float)proba donne 0 si proba est trop petit
			cumul += res.probaVille(this.ville_actuelle, i)/total;
			this.villes_visitees[i] = cumul;
		}
	}
}
```

Choisir_ville a été expliquée plus haut lorsque nous avons parlé de l'attribut villes_visitees.

Remarque : le booléen retourné par la fonction est renvoyé au panneau central, afin de lui indiquer lorsque le cycle fourmis est terminé.

```
public boolean choisir_ville(Reseau res)
{
	if (this.nbVilles_visitees < this.nbVilles)
	{
		double nombre_alea = (double)Math.random();
		int i = 0;
		while (nombre_alea > this.villes_visitees[i] ) {i++;}

		this.ville_actuelle = i;
		this.villes_visitees[this.ville_actuelle] = 0;
		this.parcours[this.nbVilles_visitees++] =
this.ville_actuelle;
		this.calculer_proba_ville(res);
		return true;
	}
	else
	{
		res.deposer_pheromones(this.parcours);
		this.ville_actuelle = ville_initiale;
		this.initialiser(res);
		return false;
	}
}
```

Partie 3 : finalisation

Critère de convergence

Nous avons décidé que l'on s'arrêtait lorsque les 100 derniers meilleurs chemins étaient identiques.

Le calcul d'un meilleur chemin se fait dans le réseau : on commence à la ville 0, puis on emprunte l'arrête contenant le plus de phéromone, et on continue le parcours en choisissant toujours l'arrête contenant le plus de phéromone parmi les arrêtes menant à une ville non encore visitée, jusqu'à avoir visité toutes les villes, on revient à la fin à la ville 0.

On calcule la longueur de ce meilleur parcours. Si c'est la même qu'au cycle colonie précédent, on incrémente le compteur, sinon on initialise le compteur à 0. Lorsque le compteur atteint 100 on arrête les cycles.

Le meilleur chemin s'affiche alors en vert sur le dessin.

```
public int meilleur_chemin()
{
	this.meilleur = new int[this.nbVilles];
	this.meilleur[0] = 0;
	double[] nonvisitees = new double[this.nbVilles];
	for (int i = 0; i < this.nbVilles; i++)
	{
		nonvisitees[i] = this.arretes[0][i];
	}
	for (int i = 1; i < this.nbVilles; i++)
	{
		int indiceduplusgrand = 1;
		for (int j = 1; j < this.nbVilles; j++)
		{
			if (nonvisitees[j] > nonvisitees[indiceduplusgrand])
{ indiceduplusgrand = j; }
		}
		this.meilleur[i] = indiceduplusgrand;
		nonvisitees[indiceduplusgrand] = 0;
		for (int j = 0; j < this.nbVilles; j++)
		{
			if (nonvisitees[j] != 0) { nonvisitees[j] =
this.arretes[indiceduplusgrand][j]; }
		}
	}

	if (this.distance_meilleure == lg_parcours(this.meilleur)) {
this.cpt++; }
	else { this.cpt = 0; }
	this.distance_meilleure = lg_parcours(this.meilleur);

	return this.cpt;
}
```

Améliorations

Lorsque tout fonctionnait, nous avons cherché à améliorer l'aspect démonstratif de notre programme :

1) Nous avons modifié l'appel à cycle : à présent on appelle repaint, qui appelle cycle (au lieu de l'inverse), ainsi nous pouvons voir l'aspect du réseau se modifier au fur à mesure de l'exécution de l'algorithme.

2) Nous avons enlevé la possibilité de lancer un seul cycle, qui n'est plus utile. A la place nous avons donné la possibilité à l'utilisateur de choisir un niveau de commentaire :

-1 : aucun commentaire

0 : on affiche seulement le numéro du cycle et le meilleur chemin

1 : on affiche à chaque cycle l'état du réseau

2 : on affiche à chaque cycle l'état d'une des fourmis (la première)

3) Le meilleur chemin ne s'affiche plus en permanence mais seulement à la fin, cela permet de mieux voir les arêtes grossir ou diminuer au cours du temps.

Remarque : Le prix à payer pour une jolie démonstration est le temps d'exécution : l'affichage prend du temps, et on est en plus obligé de faire des pauses pour que l'oeil humain aie le temps de voir l'évolution confortablement.

Ensuite, nous avons joué sur les paramètres de l'algorithme pour qu'il converge au mieux. En augmentant le nombre de fourmis, il converge plus vite, mais donne une solution moins optimale. En diminuant le nombre de fourmis, il converge moins vite, mais donne généralement une meilleure solution.

Nous avons donc décidé qu'un nombre de 20 fourmis pour 15 villes était correct, mais nous avons changé le critère de convergence et l'évaluation du meilleur chemin :

Nous avons constaté que le chemin de stabilisation n'est pas toujours le meilleur, donc nous avons gardé en mémoire le chemin optimal pour l'afficher à la fin.

Puisque la stabilisation est plus lente avec 20 fourmis et que le meilleur chemin est généralement atteint au moins une fois bien avant la stabilisation nous avons modifié le critère de convergence pour : « avoir 10 fois de suite la même longueur ou avoir atteint 2000 cycles ».

En effet, nous avons remarqué que si en 2000 cycles le système n'est pas stabilisé, c'est qu'il est parti dans une mauvaise direction et s'éloigne du chemin optimal, inutile donc de continuer.

Conclusion

Remarque sur l'intérêt de l'algorithme des fourmis

Supposons qu'on veuille résoudre le problème en étudiant tous les chemins possibles pour n villes. Nous plaçons un marchand sut la première ville, il a n-1 choix pour la ville suivante, n-2 pour celle d'après, et ainsi de suite jusqu'à ce qu'il ne lui reste plus qu'un choix, ce qui fait n ! possibilités.

Pour 15 villes, cela est de l'ordre de 10^{12} chemins à étudier, pour 20 villes c'est 10^{18} et pour 30 villes 10^{32}. Notre algorithme utilise au maximum 2000 cycles de 30 fourmis, c'est de l'ordre de 10^{4}. Nous voyons donc que l'algorithme des fourmis, même s'il se contente de donner « une bonne solution » et non « la meilleure », est bien plus efficace en temps de calcul que l'étude de tous les cas.

Améliorations possibles

Nous pensons qu'il était encore possible d'améliorer le critère de convergence, voir même de le paramétrer en fonction du nombre de ville et de fourmis, ou en offrant un choix à l'utilisateur. En effet, nous avons remarqué que la plupart du temps le meilleur chemin donné à la fin est trouvé et la figure bouge peu à partir de 200 cycles, tandis que l'algorithme utilise plutôt entre 800 et 1000 cycles pour se stabiliser.

Jeux de test

Pour nos jeux de test nous avons créé des cartes représentant des réseaux pour lesquels la solution n'est pas trop triviale, que nous pourrons utiliser pour la démonstration. En particulier, nous avons créé une carte réaliste du canton d'origine de Mickaël.

Débuggage et amélioration du code

Cette partie nous a permis de voir ce qu'on avait bien compris et ce qui nous posait encore des difficultés, et de remédier aux difficultés. Les résultats nous semblent réalistes mais comme il s'agit de trouver une solution convenable à un problème complexe, il est difficile de trouver des indicateurs objectifs permettant de démontrer que le contrat est rempli. Nous avons quand même essayé notre algorithme sur des cartes triviales pour vérifier qu'il donnait la bonne solution.

Bilan de notre travail

Ce travail nous a permis de vraiment comprendre en profondeur l'algorithme des fourmis. Nous sommes assez satisfait de l'ordre dans lequel nous avons fait les choses car nous avons pu avancer à tous les stades. Par contre, nous aurions du faire les jeux de tests au début pour perdre moins de temps à entrer les villes manuellement tout au long des tests en cours de développement.