Master Recherche Intelligence Artificielle, Raisonnement, Coopération, Langage

Adéquation et Expérimentation des Systèmes Multi-Agents Adaptatifs au Problème du Voyageur de Commerce

Rapport de Stage
Master Recherche Intelligence Artificielle
Raisonnement, Coopération, Langage

Jesper Stemann Andersen

Encadré par
Pierre Glize & Valérie Camps

Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier
31062 Toulouse CEDEX 9

Juin 2005

Résumé : Depuis quelques années, les systèmes multi-agents ont pris une place de plus en plus importante dans le domaine de l'intelligence artificielle et plus généralement dans celui des systèmes complexes. Ce travail propose une résolution du problème du voyageur de commerce par une approche émergente basée sur la théorie des Systèmes Multi-Agents Adaptatifs (AMAS) développée au sein de l'équipe Systèmes Multi-Agents Coopératifs (SMAC) de l'IRIT. Le système construit a été conçu en utilisant l'Atelier de DEveloppement de Logiciels à Fonctionnalité Emergente (ADELFE) conçu par l'équipe SMAC. L'outil d'adéquation fourni par l'atelier a permis de montrer que la résolution du problème peut être abordée localement par des agents coopératifs et autonomes. ADELFE a également permis de définir une architecture multi-agent composée de deux types d'agents, les agents "sommet" et les agents "cycle" qui tentent en permanence d'entretenir des relations coopératives avec leurs agents voisins. Les premiers résultats obtenus sont encourageants mais ils nécessitent d'être consolidés par un travail approfondi.

Contents

Chapitre 1

Résumé en Français

1.1 Introduction

Le but de mon travail était d'examiner la possibilité de résoudre le problème du voyageur de commerce par une approche émergente basée sur la théorie des systèmes multi-agents adaptatifs (AMAS) développée par l'équipe Systèmes multi-agents coopératifs (SMAC) de l'IRIT.

Le problème du voyageur de commerce est un problème, à première vue simple, mais il s'agit en réalité d'un problème complexe car il appartient à la classe des problèmes "NP-complets". Son objectif est le suivant :

> Un voyageur de commerce doit visiter des villes dont on connaît les coordonnées géographiques. Afin de réduire ses coûts, il cherche une tournée aussi courte que possible en terme de distance parcourue et qui passe exactement une fois par chaque ville.

Ce chapitre présente un résumé de mon travail en français : après avoir brièvement présenté les principes des systèmes multi-agents adaptatifs, il présente en détail les choix de conception effectués, et conclut sur les résultats obtenus.

1.2 Systèmes Multi-Agents Adaptatifs

Les problèmes à résoudre de nos jours sont de plus en plus complexes en raison de leur dynamicité, du grand nombre d'entités qu'ils mettent en oeuvre... Les méthodes algorithmiques classiques ne sont parfois pas utilisables dans un tel contexte et de nouvelles approches deviennent nécessaires. Les systèmes multi-agents, tels qu'ils sont abordés par l'équipe SMAC, permettent d'apporter une solution à ces problèmes.

Les travaux de l'équipe SMAC s'intéressent à la conception de systèmes complexes ouverts, incomplètement spécifiés, plongés dans un environnement dynamique et permettant de résoudre des problèmes n'admettant pas d'algorithme connu pour construire la solution. Ces systèmes sont composés d'agents qui poursuivent un objectif individuel et essaient de maintenir en permanence des interactions locales coopératives avec leurs congénères en évitant les dysfonctionnements tels que l'ambiguïté, l'improductivité, l'inutilité, les conflits ... Grâce à la capacité des agents à s'auto-organiser, le

système est capable de s'adapter par lui-même à toute perturbation et produit alors une fonction collective adéquate dans son environnement.

Dans cette approche le comportement de chaque agent se résume à :

- Percevoir son environnement.
- Décider quelle action il va réaliser.
- Agir dans cet environnement.

Plus précisément, lorsqu'un agent perçoit une situation qu'il juge coopérative dans son environnement, il réalise sa fonction ; lorsqu'il perçoit une situation non-coopérative, il agit pour tenter de revenir à un état coopératif.

Cette théorie a été appliquée à différents problèmes, notamment à la recherche d'information, à la simulation d'une équipe de robots footballeurs, à un système de prévision de crue, à la construction d'emploi de temps, à la bio-informatique... Les résultats obtenus ont incité l'équipe à s'intéresser à la conception d'un Atelier de DÉveloppement de Logiciels à Fonctionnalité Émergente (ADELFE) destiné à des concepteurs informatiques non-spécialistes de ce type de logiciels. Cet atelier se compose d'une notation basée sur UML, d'une méthodologie de conception basée sur la théorie des AMAS ainsi que d'une plateforme comprenant un outil de modélisation graphique.

1.3 Conception

Cette section est un résumé du chapitre 6, conception du système.

Dans la suite de cet exposé, la terminologie adoptée est celle des graphes : un sommet représentera une ville, une arête un chemin.

La méthodologie ADELFE a été appliquée au problème du voyageur de commerce et les résultats obtenus sont présentés dans l'annexe A de ce rapport. Ce travail a permis de vérifier la pertinence de la théorie des AMAS pour résoudre ce problème, puis de définir les agents du système ainsi que leur comportement. Deux types d'agents ont été définis :

- Les agents "sommet" qui réprésentent respectivement un sommet.
- Les agents "cycle" qui réprésentent un cycle de sommets dans le graphe.

Le SMA est constitué d'agents "sommet", chaque agent ayant un voisinage constitué des agents réprésentant les voisins du sommet dans le graphe. Conformément à la théorie des AMAS, le but d'un agent est de rester coopératif avec ses voisins.

La coopération idéale pour un agent "sommet" se définit par :

1. L'agent a une arête en entrée.
2. L'agent a une arête en sortie.
3. L'arête en entrée est la meilleure de son point de vue parmi les arêtes en entrée.
4. L'arête en sortie est la meilleure de son point de vue parmi les arêtes en sortie.

Le but d'un agent est de faire correspondre ses préférences avec celles de ses voisins en choisissant une arête en entrée (puis une arête en sortie) la plus courte possible, qui ne pénalise pas (qui ne dérange pas) davantage les objectifs de ses voisins : elle sera alors la meilleure de son point de vue.

On utilise une mesure de dérangement pour un agent "sommet". Elle est définie comme étant la différence de poids entre l'arête préférée par l'agent et l'arête la plus courte dans une direction donnée tout en prenant en compte les répercussions sur les mesures de dérangement de ses voisins.

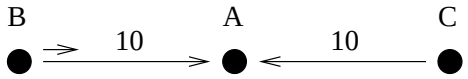


FIG. 1.1 – Décision basé sur les arrêts préférées.

Dans un tel contexte les agents “sommet” se stabilisent en formant plusieurs cycles composés d'agents dans un état coopératif.

Un SMA à deux niveaux s'est avéré nécessaire pour la suite de la résolution. Des agents de macro-niveau (agents “cycle”) ont été ajoutés au-dessus des agents “sommet”. Chaque agent “cycle” est formé d'un certain nombre d'agents “sommet” qui sont dans un état coopératif stable. Les deux types d'agents, agents “sommet” et agents “cycle”, ont des comportements quasi-identiques.

1.3.1 Conception initiale

La mise en oeuvre de la coopération a nécessité la définition de deux notions.

Préférence

Chaque agent mémorise deux préférences – une arête en entrée et une arête en sortie.

Si deux arêtes ont le même niveau de dérangement par rapport à un agent, les préférences de chacun des agents aux extrémités de ces deux arêtes sont utilisées pour déterminer l'arête préférée. Par exemple, dans la figure 1.1, l'agent A préfère l'arête BA à l'arête CA puisque l'arête BA fait partie des préférences de l'agent B.

Dérangement

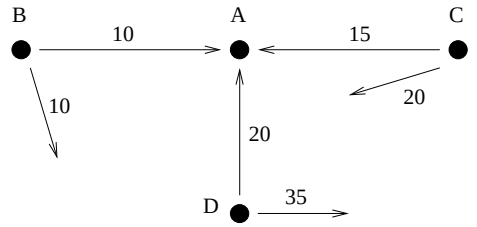
Un agent doit rester le plus coopératif possible avec tous ses voisins. C'est-à-dire qu'en choisissant une arête, un agent doit considérer aussi bien son propre dérangement que le dérangement sur chacun de ses voisins ayant une arête dans la même direction. Soient :

- $x \in \{i, o\}$, la direction de l'arête e (i entrée, o sortie) ;
- $\Delta w_{A_x}(e)$, la différence de poids entre l'arête e et l'arête la plus courte parmi les arêtes que possède l'agent A dans la direction x ;
- $d_{A_x}(e)$, le dérangement de l'agent A ayant choisi l'arête e dans la direction x ;
- $\Lambda_x(A)$, l'ensemble des voisins de A dans la direction x de A ;
- $\lambda \in \Lambda_x(A)$, un des voisins de A dans la direction x

Le dérangement d'un agent A en choisissant l'arête $e = AB$ dans la direction i peut être défini comme la somme de :

- La différence de poids de l'agent A dans la direction i, $\Delta w_{A_i}(e)$.
- Le changement de dérangement pour l'agent B à l'autre extrémité de l'arête, $\Delta d_{B_o}(e)$.
- La somme des changements de dérangement de tous les voisins $\lambda \in \Lambda_i(A)$ excluant leur arête f vers A :

$$\sum_{\lambda \in \Lambda_i(A) \setminus \{B\}} \Delta d_{\lambda_o}(\neg f)$$



Bien que l'arête DA soit l'arête d'entrée ayant le poids le plus élevé pour A, le changement de dérangement sur D en excluant DA est tellement grave qu'elle devient l'arête la meilleure pour A. Voir tableau 1.1 pour les détails.

FIG. 1.2 – Exemple de formule de calcul de dérangement.

Edge, e	$\Delta w_{A_i}(e)$	$\Delta d_{\lambda_o}(e)$	$\Delta d_{\lambda_o}(\neg e)$	$d_{A_i}(e)$
BA	0	0	0	20
CA	5	0	5	20
DA	10	0	15	15

TAB. 1.1 – Exemple de formule de calcul de dérangement.

Nous obtenons la somme :

$$d_{A_i}(e) = \Delta w_{A_i}(e) + \Delta d_{B_o}(e) + \sum_{\lambda \in \Lambda_i(A) \setminus \{B\}} \Delta d_{\lambda_o}(\neg f)$$

Un agent doit considérer son dérangement propre et celui de ses voisins avant de prendre une décision. Par ailleurs il est obligé de considérer le résultat de l'action de ses voisins. Pour cela, les agents ont besoin de conserver les résultats obtenus par l'évaluation de dérangement décrite plus haut. Par conséquent, chaque agent fait fonctionner un tableau comme celui du tableau 1.1. En plus, chaque agent garde en mémoire ses arêtes d'entrée et de sortie préférées.

Le dérangement d'un agent B quand un autre agent lui demande de préférer une arête e est déterminé de la manière suivante :

- Si e n'est pas son arête préférée (il s'agit de f), le dérangement correspondant est la différence de dérangement, $d_{B_x}(e) - d_{B_x}(f)$.
- Si e est son arête préférée la différence de dérangement pour B est nulle.
- Si B n'a pas d'arêtes préférées, la différence de dérangement est $d_{B_x}(e)$.

Le dérangement d'un agent quand un autre agent lui demande d'exclure une arête f est déterminé de la manière suivante :

- Si f est sa meilleure arête, et donc son arête préférée, le dérangement correspondant est la différence de dérangement entre f et la deuxième meilleure arête.
- Si f n'est pas sa meilleure arête ou si aucune arête n'a été préférée, la différence de dérangement est nulle.
- Si f est la seule arête de l'agent dans la direction donnée la différence de dérangement est ∞.

La figure 1.2 et le tableau 1.1 sont des exemples de démonstration de cette formule.

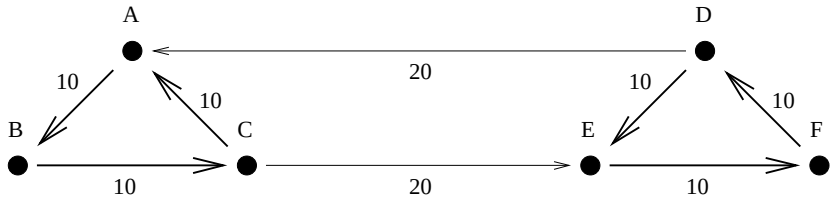


FIG. 1.3 – Un système stable constitué de plusieurs cycles ($ABCA$ et $DEFD$).

Chaque agent utilise toutes ces informations pour répondre aux questions de ses voisins. De cette façon, les connaissances de chaque agent sont propagées d'agent en agent, favorisant l'adaptation des agents aux besoins les uns des autres.

1.3.2 Deux niveaux d'agents

Quand les agents "sommet" sont stabilisés (ils sont dans une situation coopérative), on obtient un certain nombre de cycles. Si ce nombre est égal à un, il s'agit d'un seul cycle Hamiltonien. Sinon on obtient une situation comme décrite dans la figure 1.3.

Chacun de ces cycles peut être représenté par un agent "cycle", ce qui crée un macro-niveau en plus du micro-niveau composé d'agents "sommet". Un agent "cycle" a exactement le même comportement que celui d'un agent "sommet" : Il cherche à coopérer avec ses voisins, comme décrit dans la section 1.3.1. Cependant, ces agents ne sont pas des sommets, mais des cycles.

En cherchant une arête en entrée et une arête en sortie, un agent "cycle" exclut ses propres arêtes internes (arêtes reliant les sommets de ce cycle).

Contrainte sur le micro-niveau

Quand le macro-niveau s'est stabilisé, chaque agent "cycle" sera connecté avec une arête en entrée et une arête en sortie à deux autres agents "cycle".

Le contrôle du système revient alors au micro-niveau, mais le comportement des agents "sommet" est alors modifié : Les deux arêtes préférées de chaque macro-agent deviennent obligatoire pour les agents "sommet" du micro-niveau. Cela signifie que si un agent "sommet" a le choix entre deux arêtes, l'une d'entre elles étant obligatoire, cet agent "sommet" est obligé de préférer l'arête obligatoire. Par ailleurs, si on lui demande d'exclure une arête obligatoire sa réponse doit avoir une valeur de dérangement égale à ∞.

Ainsi ce changement de comportement va résoudre le problème de la figure 1.3. Si le résultat des actions des micro-agents donne encore plus d'un seul cycle, on revient au macro-niveau (constitué d'un nouvel ensemble d'agents "cycle" représentant de nouveaux cycles). Les arêtes obligatoires perdent leur caractère obligatoire dans le macro-niveau suivant.

Un exemple détaillé est présenté dans le chapitre 6.

1.4 Conclusion

Le stage a permis d'obtenir une meilleure connaissance des notions liées à la théorie des AMAS en les mettant en pratique dans le cadre d'un problème concret : Le Problème du Voyageur de Commerce.

Le Problème du Voyageur de Commerce n'est pas un problème aussi simple qu'il parait, puisque il a donné lieu à plusieurs publications et même des livres entiers[Rei94]. Il s'agit d'un des problèmes "NP-complets" les plus connus.

Malgré tous ces travaux, le Problème du Voyageur de Commerce est encore au centre de plusieurs activités de recherche. Des concours sur l'Internet sont encore ouverts, dans le but de trouver des nouvelles façons de résoudre ce problème.

Mon stage consistait à étudier la faisabilité d'une approche de résolution par émergernce pour le problème du voyageur de commerce.

Le système construit a été conçu en utilisant l'Atelier de DEveloppement de Logiciels à Fonctionnalité Emergente (ADELFE) conçu par l'équipe SMAC. L'outil d'adéquation fourni par l'atelier a permis de montrer que la résolution du problème peut être abordée localement par des agents coopératifs et autonomes. ADELFE a également permis de définir une architecture multi-agent composée de deux types d'agents, les agents "sommet" et les agents "cycle" qui tentent en permanence d'entretenir des relations coopératives avec leurs agents voisins.

Les résultats obtenus ne sont que partiels (cf. section 7.2) : Le système implémentant le comportement expliqué dans le chapitre 6, peut résoudre des cas simples du problème, mais il ne donne pas des résultats corrects pour des cas plus complexes – même s'ils s'en approchent. Donc, même si les premiers résultats obtenus sont encourageants, ils nécessitent d'être consolidés par un travail approfondi.

1.4.1 Travail à faire

Mon travail durant les prochains jours, va consister à comprendre pourquoi les résultats obtenus sont incorrects.

Si le problème est résolu, la prochaine étape consistera à comparer les résultats obtenus avec ceux obtenus avec d'autres méthodes de résolution du problème.

1.4.2 Perspectives

Les systèmes multi-agents adaptatifs permettent d'apporter des solutions à des problèmes distribués et dynamiques. Il semble donc naturel de donner la possibilité à l'utilisateur du système d'introduire de la dynamique dans le système durant son fonctionnement (supprimer une arête du graphe...). D'après le théorie des AMAS, le système devrait alors s'adapter automatiquement et prendre en compte cette pertubation.

Cette notion n'a pas été, à ma connaissance, étudiée dans le cadre du problème du voyageur de commerce. Il serait donc très intéressant de pouvoir mettre en oeuvre cette propriété.

Chapter 2

Introduction

The purpose of the project, described in this report, is to examine if it is possible to solve the Travelling Salesman Problem, as a result of an emergent behaviour of a Multi-Agent System constructed according to the Adaptive Multi-Agents Systems Theory of the Systèmes Multi-Agents Coopératifs research group at IRIT.

The Travelling Salesman Problem is a deceptively simple combinatorial problem. It can be stated simply, but solving it is hard:

> A salesman wishes to visit each among a number of cities exactly once and return to the city of departure afterwards. In which order should he visit the cities to minimize the distance travelled?

The report is composed of seven chapters. Chapter 1 consists of a short summary in French of the report. This chapter, chapter 2, is an introduction to the problem and the related terminology. Chapter 3 introduces the Travelling Salesman Problem more formally, while chapter 4 consists of a description of existing methods to solve the Travelling Salesman Problem. Chapter 5 introduces concepts related to Multi-Agent Systems as well as the Adaptive Multi-Agents Systems Theory. Chapter 6 presents the design of the developed system, and is based on the application of an methodology, whose result is presented in appendix A. Chapter 7 presents briefly the implemented Multi-Agent System and the obtained results. Finally chapter 8 summarizes the results obtained from the developed system. A bibliography is present immediately after chapter 8.

2.1 Acknowledgements

I would like to place on record my thanks and appreciation of the helpful assistance of Pierre Glize, Christine Régis, Marie-Pierre Gleizes and especially Valérie Camps who, in spite of bilateral linguistic and cultural barriers, has gone to great lengths to help me produce this report.

2.2 Applications

In addition to the obvious application of finding the shortest tour among a number of cities, the travelling salesman problem can be applied to the problem of pattern

Travel	Networks	Graph theory
Map	Network	Graph
City	Node	Vertex
Road	Link	Edge
Distance	Cost	Weight
Road	Route	Path
Tour	Ring/Route	Cycle

Table 2.1: TSP terminology.

recognition(Cf. [Ski97a] and [Ski97b]).

A pattern can be represented as an ordered list of values. A tour of some cities could thus be interpreted as being a pattern (the values are city names). Let the pattern to be recognized be the shortest tour of some cities. A key observation is that any tour of a number of cities is composed of segments of the optimal tour: These segments can be single city segments, or smaller or longer lists of cities. The longer these segments are, the more optimal the tour is. Equivalently: The longer these segments are, the closer the pattern is to the pattern to be recognized.

The pattern recognition problem can thus be solved by an algorithm solving the travelling salesman problem.

2.3 Terminology

Language British English, will be used in this report, a salesman is thus *travelling* and not traveling, as in American English.

Political correctness Being political correct in a technical report would just be confusing[1], a salesman is thus *a salesman and not a salesperson.*

Taboo Search Taboo Search is often referred to as Tabu Search, the English word for the Polynesian cultural concept[2], tapu (or tabu), however is taboo.

The Travelling Salesman Problem Multiple sets of terms can be used in describing the Travelling Salesman Problem (TSP) – table 2.1 summarises a few central ones of them.

Using travel-terms one is not able to differentiate between single step and multi step roads: A road from Paris through Bordeaux to Toulouse is not conceptually different from a road from Paris directly to Toulouse. Make note of the terms used in the graph theory column of table 2.1 – in this report the TSP is referred to using graph theoretical terms. Cf. section 3.1 for definitions of the terms.

[1] Though it would probably please a few disturbed americans.
[2] Cf. `http://en.wikipedia.org/wiki/Tapu`

Chapter 3

The Travelling Salesman Problem

The TSP comes in many forms and shapes – to differentiate a basic knowledge of graph theory is mandatory. This chapter seeks to equip the reader with the needed knowledge and identify the specific problem to be solved. The chapter mainly consists of five definitions, a discussion of different types of graphs, and finally a choice on the problem to be studied in this report.

3.1 Definitions

The following graph theoretical definitions are based on [Bry92].

Definition 3.1 (Graph) *A graph $G = (V, E)$ consists of a finite non-empty set of vertices V and a set of edges E which is any subset of the pairs:*

$$\{(v, w) \mid v, w \in V, v \neq w\}$$

A particular edge (v, w) is usually written vw. If the graph is directed $vw \neq wv$ otherwise $vw = wv$, $\forall vw, wv \in E$ (and the graph is then called undirected).

E being a set ensures that two vertices can be joined by an edge or not, but it is not possible to have two edges between the same pair of vertices (except if the graph is directed and the two edges are vw and wv).

Definition 3.2 (Path) *In a graph a path is a sequence of vertices:*

$$v_1, v_2, v_3, ..., v_{n-1}, v_n$$

(where repeats are allowed) such that $v_1v_2, v_2v_3, ..., v_{n-1}v_n$ are all different edges of the graph.

A path is thus allowed to cross over itself, but is not allowed to use the same edge twice.

Definition 3.3 (Cycle) *A path of the form*

$$v_1, v_2, ..., v_{n-1}, v_n, v_1$$

where $n > 1$ and the first n vertices are also all different, is called a cycle.

A cycle is thus a path which ends up where it starts out and in addition is not allowed to cross over itself.

Definition 3.4 (Hamiltonian cycle) *Given a connected graph, a Hamiltonian cycle in the graph is a cycle containing all vertices of the graph – and thus contains each vertex exactly once except for the first vertex appearing twice.*

Since a Hamiltonian cycle is a cycle it ends up where it starts out and it can not cross itself. A graph having a Hamiltonian cycle is said to be Hamiltonian.

Definition 3.5 (The Travelling Salesman Problem) *Given a connected graph $G = (V, E)$ and a weighting defined by the weight-function, $w : E \to \mathbb{R}$, the problem is to find a Hamiltonian cycle of minimum weight, i.e. where the weight of the edges that make up the cycle results in the smallest sum possible.*

3.2 Graph diversity

This section will discuss the different types of graphs for which the TSP can be solved.

Hamiltonian or not

It is an open mathematical problem to determine a minimal set of criteria for a graph to have a Hamiltonian cycle. Nevertheless multiple criterias exist that must hold for a graph to be Hamiltonian, some of them being:

- Each vertex must have at least two edges (the degree δv of each vertex $v \in V$ must fulfill $\delta v \geq 2$). An example of the contrary is shown on part (a) of figure 3.1 on the following page.
- The graph can not include a cyclic subgraph as the one shown on part (b) of figure 3.1.

As well as theorems guaranteeing that a graph is Hamiltonian:

- Let $G = (V, E)$ be a graph with $|V| \geq 3$ and such that $\delta u + \delta v \geq |V|$ for each pair of distinct unconnected vertices u and v. Then G is Hamiltonian (Cf. p. 78-79 in [Bry92]).
- Dirac's theorem, a weaker version of the above mentioned theorem[1].
- Chvátal's theorem[2].

Non-euclidean

Though the two graphs on figure 3.1 are visualised in an Euclidean space, the graph can not be assumed to be Euclidean (the triangle inequality might not hold). Otherwise one could profit from geometric optimisations which do not hold in general.

[1] Cf. `http://mathworld.wolfram.com/DiracsTheorem.html`
[2] Cf. `http://mathworld.wolfram.com/ChvatalsTheorem.html`

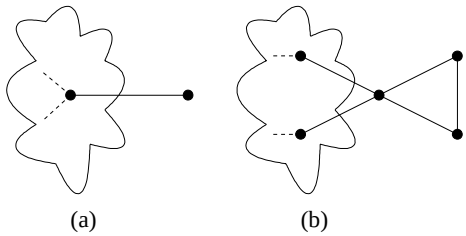


Figure 3.1: (a) A graph with a vertex which can not be reached without visiting both the connected vertex and the edge twice. (b) A graph with two vertices that can not be reached without visiting the central vertex twice.

Connectivity and completeness

It is a natural requirement that the graph is connected, otherwise one could just solve the problem for each separate subgraph.

It is however not a requirement that the graph is complete – thus there is not necessarily an edge between every pair of vertices – and nothing can be said in general on how many and which vertices are connected by edges.

Symmetric or asymmetric TSP

Two types of the TSP can be considered:

- The TSP for graphs with undirected edges called the symmetric TSP.
- The TSP for graphs with directed edges called the asymmetric TSP.

Handling the more general asymmetric TSP is not very different than the symmetric form and thus will the asymmetric TSP be the problem handled in this report. Wherever there might be differences they will be sought clarified and analysed.

3.3 The problem to be studied

To summarise the problem to be studied is defined as follows.

Definition 3.6 (Asymmetric, non-euclidean TSP) *Given a connected, directed, weighted, non-euclidean graph, the problem is to find a Hamiltonian cycle of minimum weight.*

Chapter 4

Existing research

The existing methods for solving the TSP can roughly be divided into two groups[DZ94]:

Cycle construction heuristics Incrementally building a solution to the problem.

Local search methods Searching a solution space for optimal (near-optimal) solutions.

One can say that a cycle construction heuristic tries to get it right the first time – it builds a solution and stops without trying to improve its solution once one is found.

Local search methods on the other hand starts out with a solution (possibly generated by a cycle construction heuristic) and tries to improve on this solution until some stopping criterion is satisfied.

In addition to these two groups a number of metaheuristics exist. Metaheuristics tries to combine ideas from the first two groups and often also ideas from domains not immediately related with the TSP.

4.1 Cycle construction heuristics

Cycle construction heuristics can be further divided into two groups based on what concept from graph theory they use as a basis for constructing the cycle. The first group consists of heuristics based on paths and the second group is based on cycles.

Some of the heuristics mentioned are aimed mainly at solving the TSP for complete graphs. However by applying the conversion mentioned on page 31-32 in [Rei94] an incomplete graph can be translated into a complete graph, by giving the "non-existing" edges a sufficiently large weight. This way, a TSP solution for the complete graph is a valid solution for the incomplete graph if and only if it does not include any of the edges added by the conversion. The heuristics are then in some cases able to solve the problem, however one is not guaranteed to obtain a solution.

4.1.1 Path-based heuristics

For some heuristics a small table will be listed showing the performance measures known for the method. The time complexity is defined with regard to the number of vertices in the graph, $n = |V|$, and the solution quality is defined with regard to the

Performance measure	Lower bound	Upper bound
Time complexity	$\Omega(n^2)$	$O(n^2)$
Solution quality	$rw(C_{opt}), \forall r \in \mathbb{N}$	-

Table 4.1: Performance measures for Nearest Neighbour.

weight of an optimal cycle, $w(C_{opt})$. A N/A in a performance table means that the data was not available. A dash (-) means that no bound exists.

Nearest neighbour

The simplest cycle construction heuristic is the nearest neighbour heuristic (NN)([DZ94] and [Rei94]):

1. Choose a starting vertex, v_1, adding it to the path, p.
2. Considering the end vertex of p, v_i, add the vertex v_j to p, not already in p, connected to v_i by the lightest edge.
3. Stop when all vertices are in p. Add the starting vertex to the end of p making p a Hamiltonian cycle.

A modified version of this heuristic called the double-ended nearest neighbour heuristic (DENN) lets the path grow at both ends of the path.

Although simple the nearest neighbour heuristic has two important drawbacks: It is unable to handle incomplete graphs[1], and the cumulative effect of making greedy choices makes an upper bound on the solution quality impossible, cf. table 4.1[2].

Though a solution produced by NN rarely is optimal, it is often composed of long segments of optimally connected vertices, only interrupted by a few severe mistakes. A NN solution is thus a good starting point for a local search or some other improvement method.

Greedy heuristic

The second cycle construction method based on the concept of a path is the greedy heuristic[JGM+02]:

1. Sort the edge set, E, by weight. Let P be a set of paths, initially consisting of $|V|$ paths of length zero.
2. An eligible edge is an edge that can connect two paths in P without creating a non-Hamiltonian cycle or causing an in-degree or out-degree to exceed one. Choose the lightest eligible edge from the sorted set to connect two paths in P, replacing the two paths in P with the result.
3. Add the final edge making the only path left in P a Hamiltonian cycle.

The greedy heuristic unfortunately suffers from the same problem with incomplete graphs as the NN heuristic.

[1] By always choosing the lightest edge in an incomplete graph one can end up with a cycle not containing all vertices of the graph – to handle this case some way of reversing choices would be needed – this would however simplify to a greedy, brute-force TSP algorithm.

[2] Based on [DZ94] and [Rei94].

Performance measure	Lower bound	Upper bound
Time complexity	N/A	$O(n^2)$
Solution quality	-	$2w(C_{opt})$

Table 4.2: Performance measures for Nearest Insertion.

Performance measure	Lower bound	Upper bound
Time complexity	N/A	$O(n^2 \log n)$
Solution quality	-	$2w(C_{opt})$

Table 4.3: Performance measures for Cheapest Insertion.

4.1.2 Cycle extension heuristics

Instead of constructing a Hamiltonian cycle from the basis of one or more paths, the heuristics in this section starts out with one or more cycles. The following three heuristics are all based on [DZ94] and [Rei94].

The three cycle extension heuristics all share the same structure:

1. Start with a cycle containing just one, arbitrarily chosen vertex v_1.
2. *Search rule* - If the current cycle is not Hamiltonian, search for a vertex v_k not in the cycle (having edges to two consecutive vertices, v_i and v_j, in the cycle), using some kind of search rule.
3. *Insertion rule* - Insert v_k in the cycle using some insertion rule.

The following heuristics are all less greedy than the NN heuristic.

The heuristics have been generalized to complete graphs (generalization to directed graphs have been omitted as this is a minor detail, easy to apply).

Nearest insertion/addition

In an incomplete graph the nearest insertion and the nearest addition heuristic reduces to being the same heuristic.

2. *Search rule* - If the current cycle is not Hamiltonian, search for a vertex v_k not in the cycle, so the weights of the edges connecting v_k to v_i and v_j are minimal.
3. *Insertion rule* - Replace the edge v_iv_j by the edges v_iv_k and v_kv_j.

The performance bounds are shown in table 4.2.

Cheapest insertion

Using the same insertion rule as nearest insertion, the search rule is:

2. *Search rule* - If the current cycle is not Hamiltonian, search for a vertex v_k not in the cycle, so the weight difference by inserting v_k in the cycle is minimal, i.e. minimize $w(v_iv_k) + w(v_kv_j) - w(v_iv_j) \quad \forall i, j, k$.

The performance bounds are shown in table 4.3.

Performance measure	Lower bound	Upper bound
Time complexity	N/A	$O(n^2)$

Table 4.4: Performance measures for Farthest Insertion.

Farthest insertion

The last cycle extension heuristic is based on the, at first extremely unintuitive, concept of inserting the vertices farthest from the cycle first. The idea however is to fix the overall layout of the cycle early in the extension process.

Using the same insertion rule as nearest insertion, the search rule is:

2. *Search rule* - If the current cycle is not Hamiltonian, search for a vertex v_k not in the cycle, so the expression $\min\{w(v_i v_k) + w(v_k v_j)\}$ is *maximal*.

The performance bounds for this last extension heuristic is shown in table 4.4 – a solution quality bound is not shown, since none has been proved. Experiments however have shown that the average performance is far better than for the other extension heuristics; experiments have shown that it is possible to obtain cycles which are approximately $\frac{3}{2}$ times the optimal cycle.

4.1.3 Cycle patching heuristics

Cycle patching differs from cycle extension in that instead of connecting vertices to a cycle, cycles are connected to (or patched with) other cycles.

The general structure is this[JGM+02]:

1. Compute a minimum vertex disjoint cycle cover for the graph (a set of cycles, covering all vertices of the graph, and each vertex is a part of exactly one cycle)[CK04].
2. Apply some patch rule.

The time complexity of cycle patching heuristics is often dominated by the time spent finding a minimum vertex disjoint cycle cover, as this is no easy task[3]. Naturally one can rarely consider the cover of single-vertex cycles to be minimal.

Karp and Steele proposes in [KS85] to apply the patch rule of patching the two largest cycles together, this is reported in [JGM+02] to provide better results than other proposed patch rules such as patching together the two shortest cycles.

4.1.4 Minimum spanning tree heuristics

A spanning tree for a graph, G, is a set of $n-1$ edges joining all vertices of G. A minimum spanning tree is a spanning tree of minimal weight. By removing one edge from a Hamiltonian cycle, in particular C_{opt}, one obtains a spanning tree[THCS01].

Consider a complete graph, G. By doing a depth-first traversal (with the following modification[DZ94]) of a minimum spanning tree of G the edges traversed form a Hamiltonian cycle of G. The modification is to make a shortcut when reaching a leaf-vertex, v. Instead of adding the edges when backtracking to the last vertex, w, having

[3] In particular finding a minimum cycle cover of minimal length has been shown to be NP-complete in [Tho97].

an unvisited subtree, the edge connecting v to the first unvisited child of w is added and the modified depth-first traversal continues.

Though an interesting solution, it unfortunately does not generalize to incomplete graphs (one can not be sure that a shortcut from a given leaf-vertex exists). If the graph is complete and further it satisfies the triangle inequality an upper bound on $2w(C_{opt})$ on the solution quality is assured.[4] Another very popular MST-based heuristic is Christofides.

4.2 Local search methods

Instead of constructing one solution and sticking with it one could try to improve on the solution once it has been found. Local search methods[5] tries to accomplish this by examining transitions from one solution to another in search for better solutions[DZ94].

A number of terms are affiliated with local search:

Solution space The space of solutions, S, e.g. the space of Hamiltonian cycles for a graph.

Evaluation function The measure, $f : S \to \mathbb{R}$, to be optimized, e.g. the weight function for Hamiltonian cycles C, $w : C \to \mathbb{R}$ to be minimized.

Transition set A set of transitions to modify one solution s into another s'.

Neighbourhood A neighbourhood of solutions, $N(s) \subseteq S \quad s \in S$, reachable by transitions from a specific solution s. One has to be able to reach all solutions *through* the neighbourhood. Thus it is not required that $N(s) = S \quad \forall s \in S$, but that there exists an $s' \in N(s)$, so that solutions possibly outside $N(s)$ can be reached ($N(s') \setminus N(s) \neq \emptyset$).

The evaluation function is most often continuous with regard to the neighbourhood of a solution: If the solution is good the solutions in the neighbourhood are largely also good (and vice versa).

The neighbourhood can potentially be very large, causing a huge time consumption if evaluating all solutions in the neighbourhood. Therefore one usually only evaluates a subset of the neighbourhood that should be biased towards the best solutions in the neighbourhood. A few commonly used subsets are:

- A number of randomly selected solutions from the neighbourhood.
- A number of randomly selected solutions from the neighbourhood, $N(s)$, guaranteed to include a solution that is as least as good as s (random selection stops when an improving solution is found).
- A subset of solutions which with high probability are improving.

4.2.1 Transition types

Given a solution, a Hamiltonian cycle, c, a number of methods can be applied to find a new, possibly better, solution. A number of them are listed below, ordered by their level of complexity[Rei94].

Vertex and edge insertion For each vertex v (edge e) examine all alternative positions in the cycle, choosing the one yielding the best new cycle.

[4] More information can be found in [DZ94] and chapter 35.2 of [THCS01].

[5] Also known as neighbourhood search methods or local optimization algorithms.

2-opt exchange For each pair of edges in the cycle, examine the result of removing them, yielding two paths, and reconnecting the two paths in a new way to obtain a new cycle.

n-opt exchange The 2-opt exchange method generalized to removing $n > 2$ edges instead of two (3-opt is usually the most commonly used method, as 4-opt and onwards get unnecessarily complex).

In general the more complex a method is, the better, but more time consuming, it is. For directed, incomplete graphs one must add the requirement of only considering modifications to the cycle that yields a new valid cycle, given the (existing) edges of the graph.

Lin-Kernighan heuristic

A final type of transition is the Lin-Kernighan heuristic. It is motivated on the observations that complex methods normally yield better results, where simpler methods quickly get stuck in local optima[Rei94].

It is based on the fact that sometimes a modification slightly increasing the cycle weight, can lead to better solutions than sticking with strictly improving modifications. The basic principle is to construct complex cycle modifications by combining simpler methods[6], where not necessarily all applications of the methods result in a decrease in cycle weight.

The current cream of the crop when it comes to solving large-scale Travelling Salesman Problems is an implementation[Hel98] of the Lin-Kernighan heuristic which has been able to solve[DACH04] the TSP (to optimality) for all the 24,978 cities in Sweden, and has come within 0,068% of the known best lower bound for a problem consisting of 1,904,711-cities of the world[7]. Of course these problem instances have only been solvable using distributed computing.

4.2.2 Descent methods

The most primitive methods of local search only uses transitions that are improving – these methods are known as descent methods. The problem with descent methods is that they often get stuck in local minima (a solution which is optimal in the neighbourhood, but rarely is globally optimal). One method of avoiding local minima is to run multiple descent searches with different starting points (solutions) and then choose the best local minima.

4.2.3 Simulated annealing

Another way to overcome the problem of local minima is to employ a method allowing non-improving transitions to be used. One such method is simulated annealing. Inspired by the process of steel annealing[8], simulated annealing gradually disallows non-improving transitions. Initially non-improving transitions are allowed with a prespecified probability, and as time passes this probability decreases.

[6] [Rei94] describes a method combining vertex insertion and 2-opt exchange

[7] See `http://www.tsp.gatech.edu/world/index.html`

[8] Hot/melted steel cooling off, particles settling into position, arriving at an at least near-optimal position giving the cooled steel its durable characteristics.

The probability is linearly related to the "temperature". Initially at a high temperature (ideally allowing all transitions), the temperature decreases according to some *cooling scheme* (finally disallowing all non-improving transitions). Though this application of physical terms, simulated annealing is not a direct application of the physical process, the transition to be used is selected at random[DZ94].

Though simulated annealing is good at avoiding local optima, for problems such as the TSP more efficient methods exist, due to the slow downhill movement of simulated annealing (compared to descent methods, there is a probability of simulated annealing going uphill, even when downhill movement would be obvious).

4.2.4 Taboo search

It is obvious that non-improving transitions must be allowed when reaching a local optimum. Taboo search (TS) is a method that, like a descent method, always chooses the best (or least worst) solution in the neighbourhood, but to avoid performing Sisyphean tasks[9] – such as returning to (or staying at) a previously examined solution – it places a taboo on previously visited solutions.

Thus when reaching a local optima it will try to find the least worsening transition, but the solutions visited on the way to the local optima will have become taboo, thus forcing the search to continue elsewhere. TS thus combines the speedy descent of descent methods with the diversification property (ability to explore the solution space extensively) of simulated annealing.

To remember which solutions have been visited a memory is needed. Unfortunately keeping a memory of all visited solutions is very demanding space-wise ($O(n)$ per Hamiltonian cycle). Instead one can remember other characteristics of the search history[DZ94]:

- Maintain a list of transitions. Inverse transitions of the most recent transitions are disallowed. This is very cheap space-wise: For the TSP using 2-opt exchange transitions only the four vertices need to be memorized.
- Memorizing transitions can be too sparse – in some cases revisiting old solutions cannot be avoided with the previous scheme. Instead memorization of transition attribute sets can be applied. For a 2-opt exchange transition an attribute set could again be the four vertices involved, but making these four vertices taboo would not just disallow the specific transition, but all transitions using the four vertices. Other attribute sets could be the edges that are added or removed, the change of predecessors/sucessors of the vertices etc.

The *taboo tenure* is an integer specifying for how long a taboo is valid. If a transition that otherwise is taboo leads to a new best solution, it would be natural to disregard the taboo. Criteria for disregarding the taboo are called *aspiration criteria*. Together with the taboo memory this forms the *short-term memory*, the most fundamental principle of TS.

As a complement to the short-term memory, TS also uses a *long-term memory*. The purpose of this memory is to direct the search in the long run, towards diversification. This memory often assigns a frequency to each solution or attribute set, identifying how many times this solution or attribute set has been visited. The TS then prefers solutions or attribute sets with low visit-frequencies.

[9] Cf. `http://en.wikipedia.org/wiki/Sisyphus`.

"To fill the gap" an *intermediate-term memory* is introduced, whose purpose it is to get the idea of a good solution. The intermediate-term memory records information about previously found solutions, so the search can be intensified around good solutions.

Intensification and diversification are obviously counteracting and thus only one of them is used at a time.

An interesting application of TS on the TSP was given in [ZD96], introducing variable subsets of the neighbourhood as well as an extra transition type. Though applied to the symmetric, euclidean TSP the work generalizes to the asymmetric, non-euclidean TSP.

4.3 Evolutionary algorithms

Evolutionary algorithms (EA) are based on the concept of Darwinian evolution.

The major types of evolutionary algorithms can be differentiated based on their subject of evolution[10]:

Genetic algorithms A set of solutions to a problem evolve.

Genetic programming A set of computer programs, solving a problem, evolve.

Evolution strategy Like genetic algorithms, only solutions are confined to being represented as vectors of real numbers. Self-adaptive mutation rates are added.

Evolutionary programming Like genetic programming, only the structure of the program is fixed, and its numerical parameters are allowed to evolve.

A modification of genetic algorithms, integrating local search, known as memetic algorithms[11] also exist, but its hybrid nature, though producing better results than traditional genetic algorithms, makes it kind of a curiosity.

Genetic algorithms being the most popular, and in addition the most comparable to e.g. simulated annealing and taboo search by the fact that it is based on a set of solutions, will be the only type of evolutionary algorithm studied in this report.

4.3.1 Genetic algorithms

Compared to evolutionary biology, instead of studying populations of creatures, solutions to combinatoric optimization problems are the subjects in the populations of genetic algorithms (GA).

As with local search an evaluation function is used to determine the fitness of the solutions. A standard GA for a combinatoric optimization problem has the form:

1. Initialize the population with individuals being representations of solutions.
2. Let evolution control, until the fitness of the individuals does not improve greatly.

   Reproduction Let the fittest individuals, according to the evaluation function, reproduce. The population is thus enlarged.

   Mutation For each *reproduced* individual, mutation occurs with some (possibly global) probability.

[10] Cf. `http://en.wikipedia.org/wiki/Evolutionary_algorithm`.
[11] Cf. `http://www.densis.fee.unicamp.br/~moscato/memetic_home.html`.

Elimination Evaluate both existing and newly produced solutions. The most fit survives, and evolution continues with the reduced population.

3. Return best solution from the population.

Thus in contrast to local search methods GAs works with a whole set of solutions at any given time, instead of just one solution and its neighbourhood.

For a specific application of a GA one must define the problem dependent parts:

- Representation of solutions.
- Evaluation function.
- Reproduction and mutation functions.

The reproduction function is often referred to as a crossover function, due to the origin in biology where a chromosomal crossover is the exchange of material between two chromosomes.

The following, very frequent used scheme was used in [Gol89]:

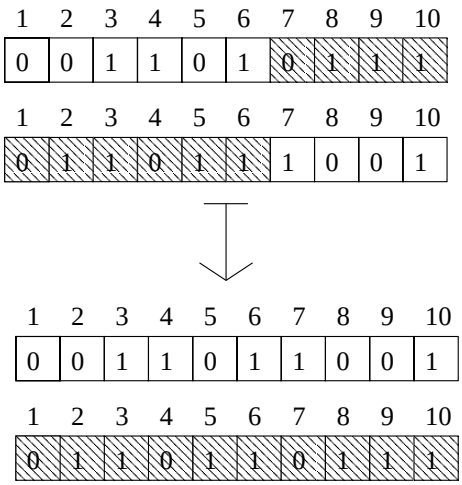
- The representation for a TSP solution is a list of vertices – completely equivalent to the mathematical representation of the cycle.
- The evaluation function is then just a mapping from a cycle to its weight (the sum of the weights of all its edges).
- The genetic funtions works on the list of vertices.
  - Choosing a reproduction function for a genetic algorithm applied to the TSP is a complex choice to make. Basically two lists of vertices are merged by exchanging two parts (of equal length) from each list. One has to make sure that the result of the merge operation results in a valid solution, a valid Hamiltonian cycle of the graph. The fact that not all combinations of the merge results in valid solutions is what makes applying the most simple reproduction function, the 1-point crossover, impossible. For reference the 1-point crossover operation working on binary strings is depicted on figure 4.1.
  - Mutation takes place by replacing a vertex in the vertex list of a solution, by some randomly chosen vertex. An example of this is shown on figure 4.2.

As was the case when considering transition types for local search methods, when considering incomplete graphs, reproduction and mutation operations must ensure that their outcome are valid cycles in the graph.

It seems however unnatural to place the responsibility of checking that a solution is valid on the genetic operators. Instead one can choose to move one step closer to the origin in biology, separating the representation of a solution from the solution itself – the representation corresponding to the chromosome (or genotype), subject to genetic operations, and the solution corresponding to the individual (or phenotype), subject to fitness evaluation.

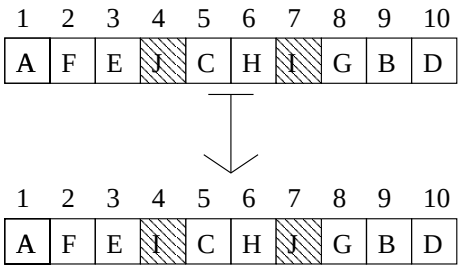
In [Bur03] the mapping from genotype to phenotype is made by a so called *gene expression algorithm*. In analogy with a simplified model of a biological process in a cell

$$\text{DNA} \rightarrow \text{mRNA} \rightarrow \text{protein for phenotype}$$



An example of the 1-point crossover operation working on binary strings. All bits after position 6 in the parents (upper part) are swapped to obtain the children (lower part).

Figure 4.1: The 1-point crossover operation.



An example of the mutation operation working on Hamiltonian cycles. The vertices at position 4 and 7 are swapped.

Figure 4.2: The mutation operation.

the gene expression algorithm[12] does not map directly from genotype to phenotype, but uses an intermediate representation

$$\text{genotype} \rightarrow \text{intermediate representation} \rightarrow \text{phenotype}$$

The intermediate representation may be absent or may appear as a whole sequence of intermediate representations.

The phenotype is then subject to fitness evaluation, while genetic operations work on the genotype – successfully removing the responsibility of checking phenotype-validity to the gene expression algorithm, where the name of the algorithm seems appropriately chosen for its function.

4.4 Neural networks

Applications of neural networks to the TSP all seem to have the same approach[Rei94]:

1. Initialize a cycle of m neurons (often, but not necessarily related to the number of vertices $|V|$) in the space of the vertices defining the problem instance.
2. As long as the stopping criterion is not satisfied, choose a vertex at random, and move the neuron closest to the vertex, and some neurons in its neighbourhood, towards the vertex.
3. Construct a cycle from the final configuration of neurons.

The idea of the neural network approach is to act like a rubber band, extending and morphing towards being a Hamiltonian cycle. Normally though a Hamiltonian cycle is obtained by calculating an approximation (the cycle) of the "the rubber band".

The neurons are as mentioned connected in a ring, interacting with each other. The number of neurons that move towards a given vertex at a given time is the output of the neural network. This number is thus variable.

[Rei94] in 1994 largely dismissed the neural network approach, and it does also seem difficult to get promising results, especially in the case of incomplete graphs. As a consequence the number of publications dealing with neural networks applied to the TSP is also small.

4.5 Swarm intelligence

The term swarm intelligence refers to decentralized, self-organized systems consisting of simple agents[RN02] interacting locally with one another and their environment. Though the decentralization often implies that no global control dictates the behaviour of the system, a global behaviour often emerges as the result of agent interactions.

The as of yet most successful swarm intelligence technique is ant colony optimization.

4.5.1 Ant Colony Optimization

Ants are a biological example of a self-organized system. Travelling between their nest and food sources, they deposit a chemical pheromone, guiding themselves towards optimal routes between nest and food source. It so happens that the chemical pheromone

[12] Cf. page 3 in [Bur03].

is detectable by the ants, and that they prefer to follow paths with large amounts of pheromone deposited. This way the ants will follow the shortest route – more pheromone will be deposited on the shortest path, as more ants will be able to cover this distance in the same time as the time taken to travel any other path.

In mathematical terms they form a minimum spanning tree of the euclidean graph with vertices defined by their nest and food sources. Though complex algorithms exist for computing minimum spanning trees, the ants are only controlled by local interactions[Kun01].

M. Dorigo proposed in his Ph.D thesis to use this ant behaviour as a way to solve the TSP[DG96]:

- A number of artificial ants (not necessarily $|V|$ ants) moves from vertex to vertex in the graph, depositing pheromone on edges travelled. The starting vertex of each ant is chosen at random.
- Ants probabilistically prefer edges with a lot of pheromone deposited and/or light edges, but they are required not to revisit vertices already visited.
- When travelling an edge the ant performs a *local trail update*. A local trail update adds pheromone to the edge, but it also subtracts a certain amount to image the evaporation of chemical pheromone in nature.
- When all ants have completed a cycle of the graph, the ant that followed the lightest cycle of the graph, performs a *global trail update*: It modifies the pheromone amount of each edge in its cycle by adding an amount that is inversely proportional to the weight of the cycle.

This process is iterated until some stopping criterion (usually that no large improvement to the cycle weights occur for some number of consecutive iterations). While the local trail update serves to direct the behavior of the ants towards some level of diversification, by evaporating some of the deposited pheromone, the global trail update serves to intensify the search near the good solutions.

The generic ant colony optimization algorithm for the TSP does not handle incomplete graphs very well. It is a requirement that each ant can traverse a Hamiltonian cycle of the graph, no matter which edges it chooses on its way. This requirement however cannot be satisfied by an incomplete graph.

A way to fix this problem is to let the ants be allowed to undo some of their moves to get back to a state where they are able to complete a cycle of the graph. Another way would be only to consider the pheromone modifications made by ants who were able to complete a cycle.

Since the original publication of the above described, so called Ant System, multiple improvements have been investigated. The best performing among these, so called Ant Colony Optimization metaheuristics, are modifications utilizing local search. These are thus hybrid algorithms. Their general structure is the same as the Ant System, with a local search inserted just before the global trail update, to try to improve on the solutions found[SD99].

4.6 Methods suitable for further study

The cycle construction heuristics are of obvious interest as a source of inspiration for constructing new metaheuristics and will thus be important when designing a new

multi-agent system. In addition the basic versions of local search methods and meta-heuristics, especially Ant Colony Optimization systems, will naturally be candidates for comparisons of solution quality and run time.

Chapter 5

Multi-agent Systems

This chapter will focus on explaining what a Multi-Agent System (MAS) is as well as the so-called Adaptive Multi-Agent Systems (AMAS) Theory – one of the results of research conducted within the Systèmes Multi-Agents Coopératifs (SMAC) research group at Institut de Recherche en Informatique de Toulouse (IRIT).

Multi-agent systems are as the name implies systems of multiple agents. This might seem like a very broad and weak definition and indeed it is. To further define the concept of a MAS a more precise definition of an agent is needed.

Multiple definitions of the concept of an agent has been proposed, however the most commonly used is the minimal definition of [RN02]:

> An agent is an active entity capable of perceiving the world around it, and performing actions in this world. The world in which the agent is situated is referred to as the environment of the agent.

An active entity can be seen as something possessing some degree of autonomy – common examples are humans, self-controlling robots etc.

Having defined what an agent is, a MAS can be further defined[Woo02]:

> A multi-agent system consists of a number of autonomous agents, sharing a common environment, trying to solve a common task.

A number of properties can be related to an agent:

Knowledge The agent can possess knowledge on its environment. When speaking of a single agent as part of a MAS, the environment of the agent includes the other agents. In a MAS each agent can have its own knowledge or the knowledge can be distributed.

Reactive or reasoning The agent can base its actions solely on its perception (a reactive agent) or combine its percepts with its knowledge to decide what to do (a reasoning agent).

Interaction/Communication Often these agents have the ability to interact or communicate with each other. This communication can happen directly or via the environment. Direct communication is simply two agents exchanging messages. An alternative is an agent modifying the environment to communicate something to one or more agents.

Learning An agent can have the ability to learn to obtain new knowledge or adapt to new situations.

Evaluation An agent can use a *utility measure* to evaluate its situation and use this information to decide what to do. Furthermore it can possess long-term goals to reach.

Behavior Using a utility measure or goals an agent can be seen as having rational or irrational behaviour. If the agent obtains knowledge of other agents situation its behavior can be egoistic, altruistic or cooperational. Cooperativity would require the agent to take into account the situation of a number of agents other than itself, but not necessarily all agents of the MAS (this would make it altruistic).

The research in multi-agent systems is motivated by the ever-increasing complexity of the systems needed to be designed. Numerous development tools and methods (UML etc.) have been developed, but they only help to slow down, not stop, the ever more complex systems[JPGG04]. The Internet and Ubiquitous Computing has greatly helped to increase the complexity of systems.

By giving the parts of a system autonomy one hopes to benefit from the interactions among the parts. However the more autonomy a system has, the less easy it is for a designer to control it or even predict what it will do[JPGG04]. To guide the design of systems consisting of autonomous parts researchers have looked to how human and animal societies work and have tried to imitate the benefits obtained from seemingly chaotic interactions between the individuals of societies. Examples are:

Ant Colony Optimization The simple behaviors of ants solve complex problems (cf. section 4.5.1 on page 26).

The free-market economy Compared to the soviet plan economy featuring tight control with production, the free-market economy though seeming "anarchic" in some ways results in better overall results[JPGG04].

Common to these examples are the autonomity of the systems parts while introducing some level of control[JPGG04]. The partial control can result in the emergence of phenomenons not immediately relatable to the control of the systems parts.

5.1 Emergence

In natural systems the concept of emergence in a system can mean:

> The observation of an ostensive phenomenon (it imposes itself to the observer) at the global or macroscopic level. Emergence is defined in terms of the irreducibility of properties associated with higher level theory to properties associated with components in a lower level theory[Kim97].

In an artificial system, in particular a MAS, each agent can be governed by a simple set of rules. If an emergent behaviour of the system emerges, it is the result of interactions among the agents, but the behaviour is not irreducible to the rulesets[1].

When constructing a MAS, with the emergence of a behaviour in mind, it is thus very important not to implement the envisaged global behaviour or even parts of it, as this would result in a system possibly solving the problem, but not due to the

[1] Using the theory of non-linear, dynamic systems one might be able to make the reduction though.

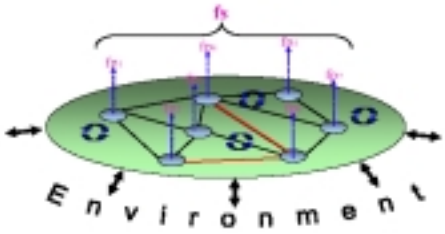


Figure 5.1: The combination of partial functions of agents.

emergence of a behaviour. It is very easy to inadvertently introduce some part of the global behaviour, so great care must be taken when constructing the system.

The organization of a MAS refers to the interaction pattern or interaction configuration of the MAS. At one instant the system might be organized in one way (agent 1 interacting with agent 3, and agent 2 and 4 interacting), while at another instant they may be organized in some other way (agent 1 and 2 interacting, and agent 3 and 4 keeping to themselves).

Self-organization in a MAS refers to the concept of the agents deciding autonomously with whom to interact.

The emergence of a behaviour of a MAS consisting of autonomous agents is thus a result of self-organization.

Due to the fact that the property of emergence cannot be explicitly expressed when constructing a MAS with an emergent property, it is very hard to construct the system to do what one actually wants it to do. One way to handle this problem is to apply a modification of the dreaded and infamous trial and error technique: The primary design process is not sufficient, there needs to be some process of *a posteori* adjustment of the system, should it not solve the given problem[JPGG04].

5.2 Adaptive Multi-Agent Systems

The SMAC research group at IRIT has in the last decade been working with multi-agent systems with emergent properties. They have named their systems Adaptive Multi-Agent Systems (AMAS). The purpose of these systems is to obtain some adequate functionality by emergence due to cooperational behaviour among the agents. A system is functionally adequate when it realizes the right function in its environment according to an external observers point of view. In other words: The adequate functionality is some adequate function which the global function of the system has to approximate.

To understand complex systems which behaviour cannot be understood by examining the systems parts, one must take their interactions into account. The AMAS approach is to consider each agent of the AMAS as an agent achieving a partial function. These partial functions are then combined due to the organization of the system to obtain the global function of the system (cf. figure 5.1). In general the order and manner in which the partial functions are combined will matter[JPGG04].

By changing the organization of the system one is thus changing the global function of the system. This organization is in the AMAS approach managed by cooperativity

– each agent is to maintain cooperative to its neighbourhood of agents.

5.2.1 Cooperativity

The SMAC research group has formulated the theoretical benefits of cooperativity in their AMAS theory.

Ideal cooperativity

Except in virtual applications, all system (even artificial), interacts within our real, physical world. For this reason, its functional adequacy depends strongly on the meaning of cooperation in the real world in order to obtain cooperative interactions. The definition of ideal cooperativity for a system, proposed in the AMAS theory, relies on the three following characteristics:

$C1$ **Understanding** A perceived signal must be interpreted by a cooperative system. Mutual understanding cannot be a postulate but should emerge from mutual adjustment between the system and its environment.

$C2$ **Reasoning** All information (an interpreted signal) must lead to new logical consequences by the system. In other words, informations must bring novelty: A difference compared to the previously available information.

$C3$ **Acting** Conclusions of the reasoning process (results of the system function) must be useful to the environment of the cooperative system.

Thus ideal cooperativity can be summarized by:

$$\text{Ideal cooperativity} = C1 \wedge C2 \wedge C3$$

As these characteristics are also valid for any part of a system, they also apply to an agent.

Any other situation can be judged as uncooperative (a non-cooperative situation (NCS)). From an observer's viewpoint, a system can detect an uncooperative state, when a novelty occurs or when a feedback of a previously erroneous response of the system arises from the environment.

In the AMAS theory, a cooperative agent must fulfill the following algorithm:

- Faced with a cooperative situation from its viewpoint, the agent executes its function.
- Faced with an uncooperative situation, it must try to act in its environment to return to a cooperative state.

The conditions for an uncooperative state can be derived from the definition of ideal cooperativity:

$$\text{Uncooperativity} = \neg C1 \vee \neg C2 \vee \neg C3$$

$\neg C1$ **Lack of understanding** A perceived signal is not understood (incompetence) or has multiple possible interpretations (ambiguity). Faced to a situation of incompetence, a cooperative agent does not ignore the received signal but must try to find other agents which might have interested in it. Faced with a situation of ambiguity, a cooperative agent must try to seek help from others in order to suppress it and in order to find a single interpretation.

$\neg C2$ **Default of reasoning** The received information is already known or has no logical consequence (unproductiveness). Faced with a situation of unproductiveness, a cooperative agent has no possibility to act in the world in such a way that a state change occurs in the world. It must propagate the received signal to agents which might have interest in it.

$\neg C3$ **Inability to act** Results of the reasoning process are useless for the environment (from the agent's viewpoint). Faced with a situation of uselessness, a cooperative agent tries to inform others of its result. This uncooperative situation permits emphasizing two other uncooperative situations called conflict and concurrence respectively.

- Conflict occurs for example, when an agent has just deduced a result which is contradictory with a result deduced by another agent. In such a case, a cooperative agent must try to find another action which will be cooperative with the action of agent l.
- Concurrence occurs when an agent has deduced the same result or has realised the same task as an other agent. In such a case, it must tries to find another action which permits it to coordinate itself with the concurrent agent.

Special case of cooperativity

For a combinatoric optimization problem, the cooperativity can be defined as:

> Given a utility measure to be maximized, an action of an agent is cooperative if and only if the action results in no decrease in the combined utility of the agent and the agents in its neighbourhood.

The concept of the neighbourhood of an agent is specific to each system (and possibly also to each agent). The concrete definition of cooperativity of course depends heavily on the given agent and the utility measure.

Similarly, a special case definition of Non-Cooperative Situations is:

Definition 5.1 (Non-Cooperative Situation) *An agent is in a Non-Cooperative Situation (NCS) if it is possible for it to perform a cooperative action.*

5.2.2 The AMAS theory

Definition 5.2 (Cooperative internal medium system) *A cooperative internal medium system is a MAS where no agent is in a NCS.*

The agent is the internal medium that has to be cooperative.

Theorem 5.1 *For any functionally adequate system situated in some environment, there is at least one cooperative internal medium system, that fulfills an equivalent function in the same environment.*

The proof of this theorem is based on the following axiom and four lemmas, for which the proofs can be found in [Cam98].

Axiom 5.1 *A functionally adequate system has no antinomic activity on its environment.*

An antinomic activity of an agent is an activity decreasing the utility of its neighbours.

Lemma 5.1 *A cooperative system is functionally adequate.*

Lemma 5.2 *For any functionally adequate system S situated in some environment, there exists at least a cooperative system S', which is also functionally adequate in the same environment.*

Lemma 5.3 *Any system having an internal cooperative medium is functionally adequate.*

Lemma 5.4 *For any cooperative system situated in some environment, there exists at least a cooperative internal medium system that is also functionally adequate in the same environment.*

5.2.3 Constructing an AMAS

To construct a MAS according to the AMAS theory one must construct a system where every possible Non-Cooperative Situation is delt with. Thus all situations where an agent has a possibility to improve its cooperativity with regard to its neighbours are handled.

Non-cooperative situations are in many ways analogous to exceptions of traditional programs.

Thus all non-cooperative situations must be detected, and for each the condition for their occurence must be specified as well as how to handle them.

Any agent in an AMAS follows a specific lifecycle which consists of three steps:

- The agent gets perceptions from its environment.
- It autonomously uses them to decide what to do in order to reach its own goal.
- It acts to realize the action it has previously decided on.

More precisely, each agent follows this lifecycle while trying to keep cooperative local interactions with others. Though an agent follows the just defined lifecycle, it has some specific characteristics and is thus composed of five parts that defines its behaviour:

Skills Knowledge about a domain enabling the agent to perform actions.

Aptitudes Abilities an agent possesses to reason on its knowledge (concerning the domain) or on its representation of the world.

Interaction language Enables the agent to interact and communicate with others in a direct or indirect way (possibly, using its environment).

World representation Knowledge used by an agent to represent itself, other agents or its environment.

Social attitude Can be anything from altruistic to egoistic behaviour. Obviously though, the AMAS theory requires agents to be cooperative.

An agent in the AMAS framework must fulfill:

- It is autonomous.
- It is unaware of the global function of the system.
- It has a limited ability to perceive and act in the world.
- It is able to interact with other agents.

- It can detect and handle non-cooperative situations.
- It is cooperative, not altruistic, not egoistic. Its cooperativity is defined via its (limited) knowledge of the world.

This theory has been applied with success to many projects: Foraging ants, knowledge management, e-commerce, flood forecasting, routing in a telephone network, mechanical design, bio-informatics etc. Obtained results have encouraged the SMAC group to promote this kind of programming towards designers not accustomed to developing AMAS. To ease the construction of AMAS agents, a methodology has been developed.

5.2.4 The ADELFE methodology

The ADELFE (Atelier de Développement de Logiciels à Fonctionnalité Emergente) methodology seeks to aid the engineer when constructing systems based on the AMAS approach. The methodology is based on the Rational Unified Process[PG04], uses UML and Agent-UML notations and adds some specific steps to design adaptive systems. The aim is not to add another methodology but to work on some aspects not already considered by existing ones such as complex environments, dynamics or software adaptation. OMG's SPEM (Software Process Engineering Metamodel) has been used to express the ADELFE process. The SPEM vocabulary (WorkDefinitions (WDi), Activities (Aj) and Steps (Sk)) is used to explain the methodology.

Only the requirements, analysis and design WorkDefinitions require modifications in order to be tailored to AMAS and are thus the only presented in the next paragraphs. Other WorkDefinitions appearing in the RUP remain the same.

The idea is to divide the design process into parts, posing multiple simple questions leading to greater knowledge of the domain and applicability of an AMAS.

Two main purposes of the methodology are to guide the engineer to:

- Identify the agents of the system, even when there is more than one level of agents. An agent can itself be composed of agents if its purpose necessites an AMAS.
- Defining the Non-Cooperative Situations of each agent and how to handle them.

The methodology helps choosing agents among entities of the problem domain. The choice of agents should be based on the problem specification – and naturally not on random guesswork.

The ADELFE methodology is separated into four parts as shown on figure 5.2 on the following page. No agent concept occurs before the analysis part, though it is hinted at earlier in the process.

Preliminary requirements First step of any design process is to figure out what the system should do. In particular a set of keywords is defined.

Final requirements Each keyword is categorized as being an active entity or a passive entity, respectively reflecting the keyword having a possibility of acting autonomously or not.

Analysis Identification of agents among active entities. Five properties of each agent are determined:

Skills Knowledge about the domain enabling the agent to perform actions[2].

[2] A specific action of an agent can be seen as a skill as well.

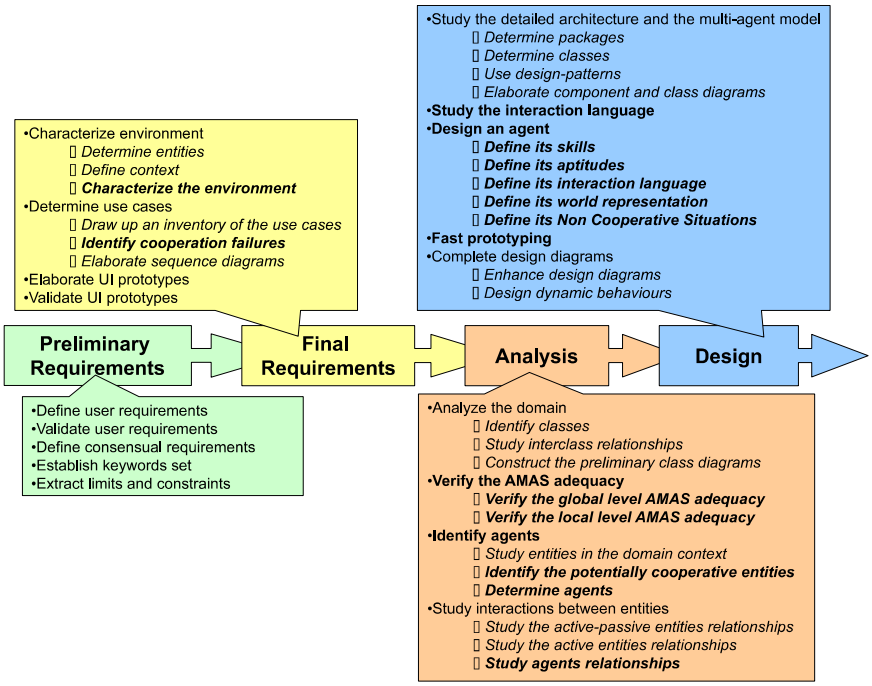


Figure 5.2: ADELFE.

Aptitudes Abilities to reason on the knowledge of the agent or on its world representation.

Interaction language Enabling the agent to communicate with other agents.

World representation A representation of itself, other agents and the surrounding environment.

Non-cooperative situations Conditions to detect and means to handle each NCS.

Design The produced MAS system is tested for adequacy using the AMAS Adequacy tool (a part of the ADELFE toolkit). It is tested on two levels.

- On the local level, each agent is tested for compliance with the AMAS theory.
- On the global level, the MAS is tested for compliance.

If the result of the AMAS adequacy test is not successful - the ADELFE process must be repeated. Possibly one of the found agents can be a MAS itself, thus resulting in a multi-layer AMAS.

Even though a system might contain different types of agents, their behavior should be sought to be as identical as possible as this greatly reduces the number of non-cooperative situations one must face.

Chapter 6

System design

The ADELFE methodology has been applied to the TSP and has resulted in the design shown in appendix A. The adequacy tool of the ADELFE toolkit have shown that the AMAS theory is an applicable approach to solve the TSP – in other words, finding a solution to the TSP can be accomplished by a set of locally cooperative, autonomous agents.

The MAS consists of vertex agents, each agent has a neighbourhood consisting of the agents representing the vertex' neighbours in the graph. The goal of an agent is to remain cooperative with these neighbours.

The ideally cooperative situation of a vertex agent is:

1. It has one incoming edge.
2. It has one outgoing edge.
3. The incoming edge is the lightest edge possible among the incoming edges, while staying cooperative with its neighbours.
4. The outgoing edge is the lightest edge possible among the outgoing edges, while staying cooperative with its neighbours.

To arrive at having exactly one incoming edge and one outgoing edge each agent maintains two *preferences* – one incoming edge and one outgoing edge. The goal of an agent is then to make its preferences match with its neighbours while seeking to prefer the lightest edges possible. When the preferences of all vertices match, the preferred edges hopefully forms a Hamiltonian cycle in the graph.

The utility measure for a vertex has been named the "disturbance measure" – to reflect that being cooperative, in part means not disturbing ones partners. The disturbance measure for an agent is defined to be the weight difference between the preferred edge and the lightest edge in a given direction, while taking the impacts on neighbours into account. Note that the disturbance measure of course should be sought minimized!

Further it was found that a two-level MAS was needed. A macro-level of agents was added to the vertex agents at the micro-level. Each macro-agent is a cycle, formed by a number of micro-agents in a stable cooperative state. The two types of agents, vertex agents and cycle agents have almost identical behaviour.

For now the fact that there exists two levels of agents will be ignored and focus is on the vertex agents – the small differences will be discussed later on.

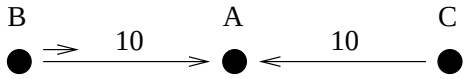


Figure 6.1: Deciding based on preference.

6.1 Preliminary design

This section will seek to describe the preliminary design deduced from the result of the application of the ADELFE methodology.

6.1.1 Edge preferences

As mentioned the preferences of an agent denotes the two choices of edges it has made. If two edges causes the same disturbance to an agent, the agent is faced with a NCS, because according to point 1 and 2 of the above cooperativity list, the agent should only choose *one* edge. In figure 6.1 vertex A has two incoming edges, both with disturbance 10. But since vertex B is preferring the edge to A as its outgoing edge, and C is not preferring its outgoing edge to A, it would be more cooperative for A to choose the incoming edge from B.

6.1.2 The disturbance measure

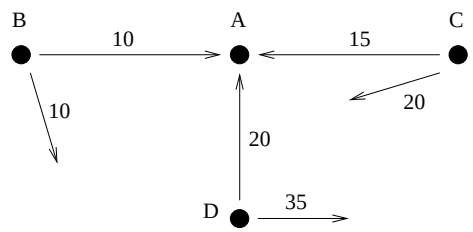
The disturbance measure (or level) denotes how satisfied an agent is – in its current situation or given some change of situation. An agent must try to keep cooperative relations to *all* its neighbours. This means that the disturbance of an agent by choosing an edge should take into account the disturbance level of itself as well as the impact on each neighbour with an edge in the same direction. Let:

- $x \in \{i, o\}$ specify the direction of an edge e.
- $\Delta w_{A_x}(e)$ specify the weight difference between the edge e and the lightest edge of agent A in direction x.
- $d_{A_x}(e)$ specify the disturbance level of agent A by choosing the edge e with the direction x.
- $\Lambda_x(A)$ denote the set of neighbours to A in the direction x of A.
- $\lambda \in \Lambda_x(A)$ denote some otherwise unspecified neighbour of A in the direction x.

The disturbance level of an agent A by choosing the edge $e = BA$ in direction i can now be stated as the sum of:

- The weight difference for the agent A in the given direction i, $\Delta w_{A_i}(e)$.
- The change in disturbance level of the agent B at the other end of the edge, $\Delta d_{B_o}(e)$.
- The sum of the changes in disturbance level of all other neighbours $\lambda \in \Lambda_i(A)$ disregarding their edge f to A:

$$\sum_{\lambda \in \Lambda_i(A) \setminus \{B\}} \Delta d_{\lambda_o}(\neg f)$$



Though the edge DA is the worst incoming edge for A itself, the impact on D by not preferring DA is so severe that it becomes the best edge for A as well. The details are shown in table 6.1.

Figure 6.2: An example visualizing the disturbance level formula.

Resulting in the disturbance level of A, by choosing the edge e in direction i:

$$d_{A_i}(e) = \Delta w_{A_i}(e) + \Delta d_{B_o}(e) + \sum_{\lambda \in \Lambda_i(A) \setminus \{B\}} \Delta d_{\lambda_o}(\neg f)$$

What is the change in disturbance level of an agent B when it is as asked to consider preferring an edge e?

- If e is not its best edge, and thus some other edge f is its preferred edge, the answer must be the difference in disturbance level, $d_{B_x}(e) - d_{B_x}(f)$.
- If e is its best edge the change in disturbance level for B is zero.
- If B has not chosen any edge the change in disturbance level is $d_{B_x}(e)$.

What is the change in disturbance level of an agent when it is asked to disregard an edge f?

- If f is its best edge, and thus its preferred edge, the answer must be the difference in disturbance level between the preferred edge f and the second-best edge.
- If f is not its best edge or no edge has been preferred, the change in disturbance level is zero.
- If f is the only edge of the agent in the given direction the change in disturbance level is ∞, as the agent has no other edge to prefer.

An example of this formula in action is shown on figure 6.2 and the values of the formula are shown in table 6.1.

The formula shows that if an agent would suffer a large penalty in disturbance level by disregarding a given edge (the $\Delta d_{\lambda_o}(\neg e)$ column), this value would add to the disturbance level of all the alternative edges in the same direction. Similarly will an edge with a large penalty by preferring a given edge differentiate itself to the alternative edges (as the alternative edges will not get this possibly large value added to their disturbance level). In combination with the last parameter, the weight difference for the acting agent, this ensures that care is taken for the agent in the neighbourhood with the largest current disturbance level – it is ensured that the preference of an edge, will not prefer the worst edge, or disregard the best edge – whichever causes the largest disturbance increase to the given agent.

Edge, e	$\Delta w_{A_i}(e)$	$\Delta d_{\lambda_o}(e)$	$\Delta d_{\lambda_o}(\neg e)$	Disturbance level
BA	0	0	0	20
CA	5	0	5	20
DA	10	0	15	15

Table 6.1: An example of the formula computing the disturbance level.

An agent should take the disturbance level of itself and its neighbours into account when making decisions. Furthermore it is obliged to consider the results of actions of its neighbours as well as itself. To take each others actions into account, the agents need to store the knowledge obtained when performing the disturbance evaluation described above, so it can inform neighbours of its knowledge. Each agent thus maintains a table as the one shown in table 6.1. In addition to this disturbance level information, each agent keeps track of its preferred incoming and outgoing edge, to resolve situations as the one on figure 6.1.

When an agent is asked about its disturbance level change given some action (preferring or disregarding a given edge) it uses the table of its edges, and its preferences to answer the question. In this way the knowledge of one agent propagates from agent to agent, making the agents adapt to each others needs.

The following is the precedence order of the decision parameters:

Disturbance level The disturbance level by selecting an edge is the primary decision parameter.

Preference If two or more edges have the same disturbance level, the preference of neighbours are used to decide.

Weight difference If two or more edges have both the same disturbance level and the same preference selection with regard to the agent acting, the acting agent simply makes an egoistic choice, selecting the edge with minimal weight (among the edges deemed to be equal).

6.1.3 An example

The following example will seek to visualize that the given behavior of the vertex agents will result in the emergence of a cycle in the graph. In fact a number of cycles will emerge – this will be discussed later on.

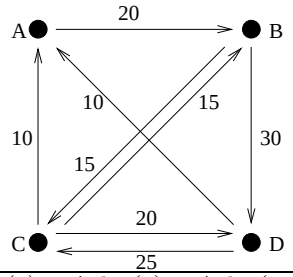
The example to be studied is shown in figure 6.3 along with the disturbance tables of the four vertices A, B, C and D. Each table has been initialized with the weight difference for each edge, zeroes and the resulting disturbance level.

The system lets each agent act and then hands over control to the next. When all agents have acted the system resumes from the start. The order in which the agents act is C, B, A and last D.

Iteration 1, Agent C

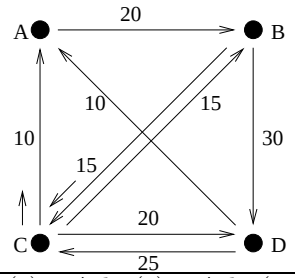
Agent C seeks to prefer an incoming edge and an outgoing edge.

In the first iteration, no agents have acted, and thus no preferences has been made. When C thus asks its neighbours about their change in disturbance level by preferring a given edge they thus just reply their disturbance level for the given edge (as they



Edge of A, e	$\Delta w_{A_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
CA	0	0	0	0
DA	0	0	0	0
AB	0	0	0	0
Edge of B, e	$\Delta w_{B_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
AB	5	0	0	5
CB	0	0	0	0
BC	0	0	0	0
BD	15	0	0	15
Edge of C, e	$\Delta w_{C_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
BC	0	0	0	0
DC	10	0	0	10
CA	0	0	0	0
CB	5	0	0	5
CD	10	0	0	10
Edge of D, e	$\Delta w_{D_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
BD	10	0	0	10
CD	0	0	0	0
DA	0	0	0	0
DC	15	0	0	15

Figure 6.3: Example showing the forming of a cycle. Initial system state.



Edge of C, e	$\Delta w_{C_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
BC	0	0	0	0
DC	10	15	0	25
CA	0	0	0	0
CB	5	0	0	5
CD	10	0	0	10

Figure 6.4: Example showing the forming of a cycle. Iteration 1, system state after agent C has acted. The small black arrows near C show the preferences of agent C

have not yet acted, their disturbance level is equal to the weight difference to their optimal edge). Likewise they all reply zero as disturbance level change when asked to disregard a given edge. The result is shown on figure 6.4.

After all neighbours have been asked about their change in disturbance, given the available options of agent C, the table shows that the best incoming edge is BC and the best outgoing edge is CA (these edges have the smallest disturbance level). Agent C thus prefers these two edges, as marked by the two small arrows on figure 6.4.

Iteration 1, Agent B

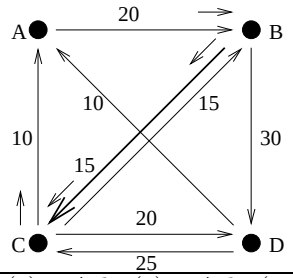
Agent B seeks to prefer an incoming edge and an outgoing edge.

Incoming Considering the edge AB. Agent A has no preferred outgoing edge, so it replies its disturbance level of the edge AB when asked about its change in disturbance level by preferring the edge. When asked to disregard the edge, it has no other outgoing edges, so it replies ∞.

Considering the edge CB, agent C is preferring the other edge CA, so it replies the disturbance level change, $d_{C_o}(CB) - d_{C_o}(CA)$, when asked. If asked to disregard it replies 0 as it does not prefer the edge CB.

Outgoing Considering the edge BC. C prefers BC so it replies zero as disturbance level change when asked what a preferral would do to it. If asked to disregard the edge BC, it replies the disturbance level difference between BC and its only other incoming edge DC.

Considering the edge BD. D has no preference, so it replies the initial values: Its disturbance level if asked to prefer the edge, and zero if asked to disregard the edge.



Edge of B, e	$\Delta w_{B_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
AB	5	0	∞	5
CB	0	5	0	∞
BC	0	0	25	0
BD	15	10	0	50

Figure 6.5: Example showing the forming of a cycle. Iteration 1, system state after agent B has acted.

As the table in figure 6.5 shows, the most cooperative incoming edge is AB and the most cooperative outgoing edge is BC, so B prefers these two edges as shown. The edge BC is made larger, to indicate that the preferences of agent B and agent C match.

Iteration 1, Agent A

Agent A seeks to prefer an incoming and an outgoing edge.

Incoming Considering the edge CA. C prefers CA, so it would cause no change in disturbance level to C if A should prefer CA. If C was not to prefer CA, its second-best edge is the edge CB, thus it replies the disturbance level change between these two edges.

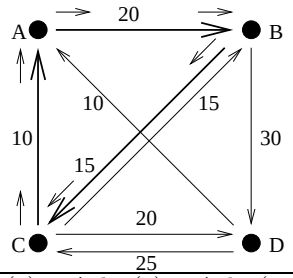
Considering the edge DA. D has no preference, so it replies the initial values: Its disturbance level if asked to prefer the edge, and zero if asked to disregard the edge.

Outgoing Agent A has only one outgoing edge, AB. It however asks B what its change in disturbance level would be should it prefer or not prefer the edge. B has chosen the edge, so it suits B, should A choose AB. If A should for some reason not choose AB, B reports what it knows so far – that it would impose a huge penalty in disturbance on it (the information that agent B got from A when it was acting). Agent A prefers the incoming edge CA and the outgoing edge AB, cf. figure 6.6.

Iteration 1, Agent D

Agent D seeks to find two edges to prefer.

Incoming Considering BD. B prefers the other edge BC, so the change in disturbance level of B if asked to prefer the edge BD is the difference between the two.



Edge of A, e	$\Delta w_{A_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
CA	0	0	5	0
DA	0	0	0	5
AB	0	0	∞	0

Figure 6.6: Example showing the forming of a cycle. Iteration 1, system state after agent A has acted.

As B does not prefer the edge BD it does not care if D should not prefer the edge – it replies a change in disturbance level of zero on the question of disregarding the edge.

Considering CD. C prefers the other edge CA, so the change in disturbance level of C if asked to prefer the edge CD is the difference between the two. As C does not prefer the edge BD it does not care if D should not prefer the edge – it replies a change in disturbance level of zero on the question of disregarding the edge.

Outgoing Considering DA. A prefers the other edge CA, so the change in disturbance level of A if asked to prefer the edge DA is the difference between the two. As A does not prefer the edge DA it does not care if D should not prefer the edge – it replies a change in disturbance level of zero on the question of disregarding the edge.

Considering DC. C prefers the other edge BC, so the change in disturbance level of C if asked to prefer the edge DC is the difference between the two. As C does not prefer the edge DC it does not care if D should not prefer the edge – it replies a change in disturbance level of zero on the question of disregarding the edge.

Agent D prefers the incoming edge CD and the outgoing edge DA, cf. figure 6.7.

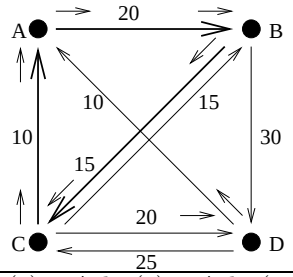
Iteration 2, Agent C

In contrast to when agent C first acted, all agents have now acted, and thus this can lead to a change of preference of C.

The preferred incoming edge stays the same, but the preferred outgoing edge is changed to CD, as a consequence of the information C gets from its neighbours. After agent D has acted it is able to report to C that it would be subject to a large penalty should it not choose the edge CD. Cf. figure 6.8.

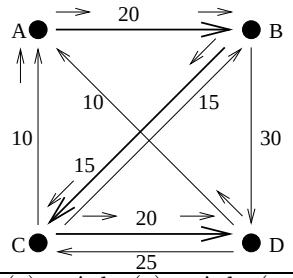
Iteration 2, Agent B

The preferred incoming edge and outgoing edge stays the same. Cf. figure 6.9.



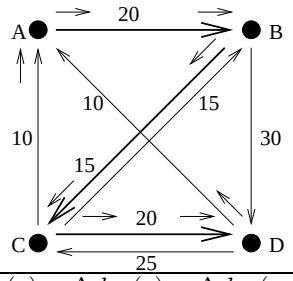
Edge of D, e	$\Delta w_{D_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
BD	10	50	0	60
CD	0	10	0	10
DA	0	5	0	5
DC	15	25	0	40

Figure 6.7: Example showing the forming of a cycle. Iteration 1, system state after agent D has acted.



Edge of C, e	$\Delta w_{C_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
BC	0	0	50	0
DC	10	35	0	95
CA	0	0	5	50
CB	5	∞	0	∞
CD	10	0	50	15

Figure 6.8: Example showing the forming of a cycle. Iteration 2, system state after agent C has acted.



Edge of B, e	$\Delta w_{B_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
AB	5	0	∞	5
CB	0	∞	0	∞
BC	0	0	95	0
BD	15	50	0	160

Figure 6.9: Example showing the forming of a cycle. Iteration 2, system state after agent B has acted.

Iteration 2, Agent A

The preferred incoming edge is changed to DA, as a consequence of the information A gets from its neighbours. After agent D has acted it is able to report to A that it would be subject to a large penalty should it not choose the edge DA. The preferred outgoing edge stays of course the same. Cf. figure 6.10.

Iteration 2, Agent D

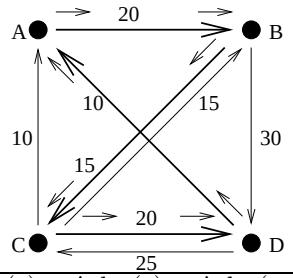
The neighbours of D has adapted to the needs of D, which have no further needs to be covered, so it maintains its preferred edges, cf. figure 6.11.

Note that the system is now stabilized – though the values in the tables will continue to accumulate, their order will remain the same. As a consequence the vertices will not change their preferences and the system is thus stable. It should also be noted that if following a path from vertex A guided by matching preferences (A prefers AB and B prefers AB), one is returned to A after having visited every vertex in the graph – a Hamiltonian cycle is formed. No vertices are isolated or have been left out by its neighbours. It can be verified that the found Hamiltonian cycle is the lightest Hamiltonian cycle of the graph.

6.2 Two levels of agents

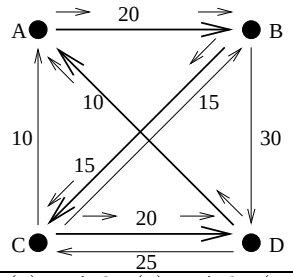
Though the above example gives the impression that the system is able to find the lightest Hamiltonian cycle of a graph, consider figure 6.12. The figure shows the result of the system when operating on a graph consisting of two groups of three vertices. The two groups are further apart (20) than the parts of each group are from each other (10). The result of the system is to create two cycles.

The second level of agents, briefly mentioned earlier, are needed to resolve this problem. When the micro-level of vertex agents has stabilized, a number of cycles are



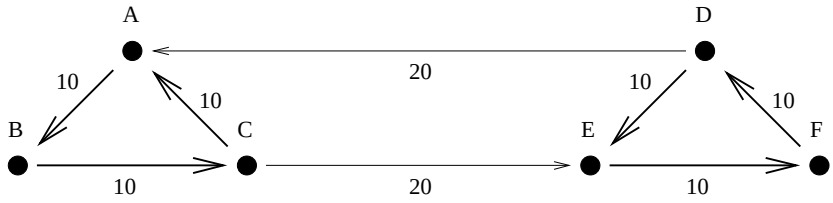
Edge of A, e	$\Delta w_{A_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
CA	0	35	0	70
DA	0	0	35	0
AB	0	0	∞	0

Figure 6.10: Example showing the forming of a cycle. iteration 2, system state after agent A has acted.



Edge of D, e	$\Delta w_{D_x}(e)$	$\Delta d_{\lambda_x}(e)$	$\Delta d_{\lambda_x}(\neg e)$	Disturbance level
BD	10	160	0	205
CD	0	0	35	0
DA	0	0	70	0
DC	15	95	0	180

Figure 6.11: Example showing the forming of a cycle. iteration 2, system state after agent D has acted.



The system has stabilized itself in a situation where two cycles $ABCA$ and $DEFD$ has formed themselves without breaking up and connecting themselves into the obvious Hamiltonian cycle.

Figure 6.12: A stable system consisting of multiple cycles.

present. Each of these cycles are then represented by a macro-level agent – a cycle agent.

It has been found that it is not necessary to reapply ADELFE, due to the fact that a cycle agent has exactly the same behaviour as a vertex agent – it seeks to cooperate with its neighbours as described in section 6.1. Its neighbours however are not vertices, but of course agents from the same level – other cycle-agents. Each cycle-agent thus seeks to obtain the following ideal cooperative situation:

1. It has one incoming edge.
2. It has one outgoing edge.
3. The incoming edge is the lightest edge possible among the incoming edges, while staying cooperative with its neighbours.
4. The outgoing edge is the lightest edge possible among the outgoing edges, while staying cooperative with its neighbours.

All internal edges (edges between vertices that are part of the cycle agent) are disregarded as they do not serve to connect the cycle agent to its neighbours. This of course includes the edges that make up the cycle (the edges on the border of the cycle).

6.2.1 Constraint on the micro level

When the macro level has stabilized, each cycle will be connected to one cycle with an incoming edge and one cycle with an outgoing edge. These two cycles can possibly be just one, single cycle.

The system then returns control to the micro level, but a modification is made. The two preferred edges by each macro agent are made mandatory for the vertex agents at the micro level. This means that if a vertex agent has the choice between two edges, and one of them is marked as mandatory, it is obliged to prefer this edge. Furthermore it replies a difference in disturbance level of ∞ if asked to disregard a mandatory edge.

It should be obvious that this scheme will resolve the problem shown in figure 6.12. If the actions of the micro agents still result in more than one cycle, one revisits the macro level (populated by a new set of cycle agents reflecting the new set of cycles). The mandatory markings of a set a of cycle agents are only valid for the subsequent pass at the micro level.

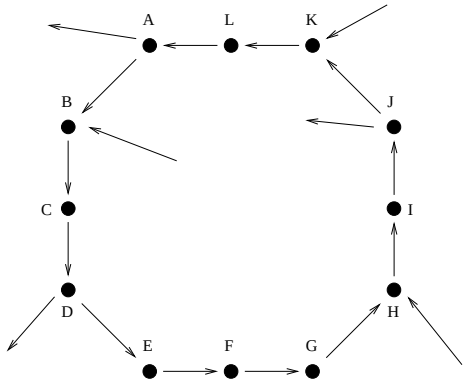


Figure 6.13: Legal and illegal edges for a cycle agent.

6.2.2 Constraint on the macro level

The vertex level imposes a constraint on the cycle level. The constraint is sought visualized on figure 6.13. The incoming edge at vertex K is a legal incoming edge as this does not inhibit the vertex J from choosing an outgoing edge. If J would not have had an extra outgoing edge in addition to the one to K, the incoming edge at K would be illegal. This is the case at the vertex H.

Similarly the outgoing edge at A is a legal edge (B has an extra incoming edge), whereas the outgoing edge at D is illegal (E has no extra incoming edge).

As a consequence, these illegal incoming and outgoing edges are not considered by the cycle agent when deciding on which two edges to prefer.

6.3 Double edges

Consider figure 6.14, nothing in the behaviour control specified in section 6.1 encourages agent A or agent C to prefer the edges to or from agent B. If B is asked what change in disturbance level it would be subject to if an edge was preferred or not, it would reply zero in either case.

A solution is to add the following behaviour requirement:

> If a vertex agent has the option to connect itself to a different vertex agent in each direction it should use it.

What impact does this modification have on the system? Without this modification the vertex agents of a complete, undirected graph, with an even number of vertices would form 2-vertex cycles. Each of these cycle agents would then link themselves to the closest neighbour cycle. This two step behaviour, form cycles and link them, would only be accelerated by the modification. If the graph consisted of an odd number of vertices, the system would in fact not return a result – as is the case on figure 6.14. If it is not possible for an agent to connect itself to two different agents in each direction, it is of course allowed, as this is the special case of the graph consisting of two vertices and two edges.

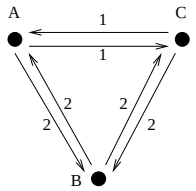


Figure 6.14: Double-edge problem of vertex agents.

Turning to the macro level, it is discovered that the same problem applies to cycle agents. Consider the graph in figure 6.15. Two cycles will obviously form, namely $ABCA$ and $DEFD$. The macro agents will then choose the lightest incoming and outgoing edges while staying cooperative, namely CE and DC. If removing all edges from the graph, that are made impossible to prefer due to CE and DC being mandatory, the graph in figure 6.16 is obtained.

The next run of micro agents and macro agents will result in AC and CB (or BC and CA) being mandatory and the system will transition between these two states indefinitely.

Though another solution could be constructed for this transition-cycle problem, it can be seen that the solution of selecting connections to different agents in each direction can solve this problem as well. An incoming or outgoing edge of a cycle agent can be seen as connecting to another cycle agent as well as to a vertex agent. The requirement analogue to the requirement for vertex agents is: If a cycle agent has the possibility to connect itself to a different vertex agent in each direction it should use it.

In figure 6.15 agent $DEFD$ will choose the edge BE instead of CE, but $ABCA$ will still choose to prefer the edge CE. To make the behaviour consistent on both sides, the requirement is modified to:

> A cycle agent is required to choose an incoming edge and an outgoing edge with no end-vertices in common if it is possible.

The two cycle agents in figure 6.15, $ABCA$ and $DEFD$, will then choose the edge BE instead of CE and a Hamiltonian cycle ($ABEFDCA$) will be the result of the subsequent interactions of the vertex agents.

This is another example of interactions between the micro level and the macro level.

For an agent to make sure that its preferred pair of an incoming and an outgoing edge satisfies the requirement of avoiding double edges while searching for the two edges *minimizing* the disturbance level, an agent needs to consider pairs of edges. Thus after querying each neighbour, an agent compares the pairs of edges starting with the best edges, stopping when a pair is found that satisfies the double edge requirement.

6.4 Final behaviour

To sum it up the decision of an agent (vertex or cycle) is:

1. For each incoming and outgoing edge, query the neighbour for its change in disturbance level, with regard to preferring or disregarding the given edge.

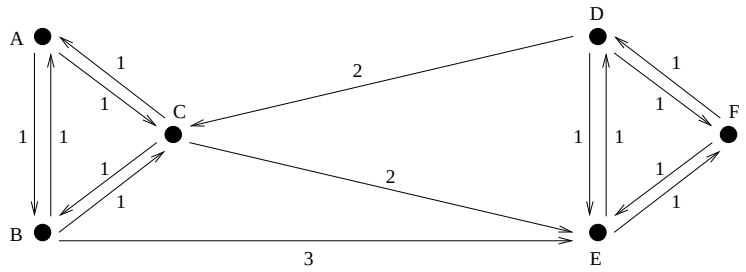


Figure 6.15: Double-edge problem of cycle agents.

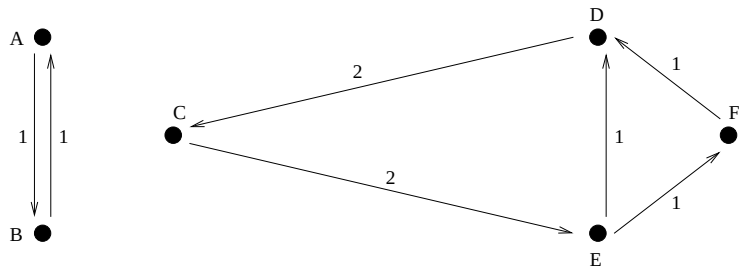


Figure 6.16: Double-edge problem of cycle agents. All edges made impossible by mandatory edges are removed.

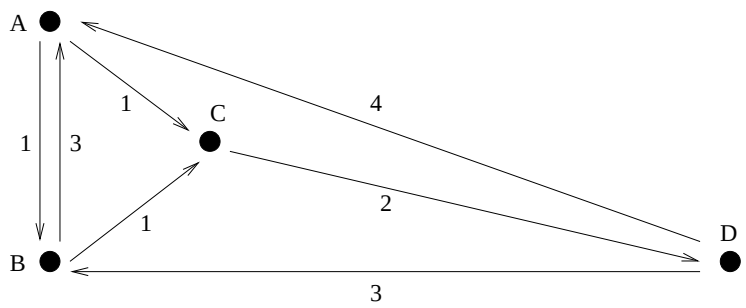


Figure 6.17: Relative weights.

2. Considering all possible combinations of an incoming and an outgoing edge, sort them according to the precedence:
   (a) Disturbance level.
   (b) Preference of neighbours.
   (c) Own weight difference.
3. Starting with the best edge combination, search for the first combination satisfying the double edge requirement.

In addition, for a vertex agent the mandatory markings of macro agents take precedence over disturbance level (and preference and own weight difference).

6.5 Properties of the design

As a sideeffect of the design of the decision control of an agent, the system is capable of handling the situation in figure 6.17. Agent D is situated far away from the three agents A, B and C. It has two edges of different, but similar weight to the cluster of three vertices. Though D individually, would prefer the edge DB over the edge DA, the gain for D is smaller than the penalty A and B has to pay by preferring the edge BA as a result of the choice of D. The propagation of values ensures that this problem of relative weights is handled.

Efficiency

Disregarding the 2-vertex graph special case, the design ensures that at the micro-level, each cycle will consist of minimum 3 vertices – the minimum number of vertices that can connect without creating a double edge, resulting in $\log_3 |V|$ cycles. Each cycle will then connect to at least one other cycle, resulting in $\log_2(\log_3 |V|)$ cycles. Speedwise this seems reasonable.

However for large and especially complete graphs, maintaining the tables with a column for each edge rapidly gets tiresome. Instead one can consider just a subset of the edges in each direction:

- Initially one can consider the subset consisting solely of the lightest edges (the edges with the smallest disturbance level).

- If after querying the subset of neighbours, the disturbance level of one or more edges is larger than the weight of edges not yet considered, the edges not yet considered must be added to the subset of edges.

Similarly one can imagine some smart way of removing edges from the subset to avoid the subset growing to the full set of edges.

Another way to minimize the amount of computation needed is to recognize that only the agents, whose neighbours have changed state (their preferences) in the last iteration, have a need to recompute.

Differences compared to other solutions

Though the behaviour of the agents is somewhat similar to the Nearest Neighbour heuristic – initially they choose the lighest edges, the behaviour of the MAS is based on cooperativity as defined in the AMAS theory. This makes quite a few differences. Instead of just sticking with initial choices of edges, a form of "constraint satisfaction" is applied. In addition the "constraint satisfaction" system is distributed making the system adaptive.

Thus in contrast to cycle construction heuristics the AMAS approach brings adaptivity and flexibility to the problem resolution. In contrast to local search methods, the AMAS approach utilizes a very simple behaviour algorithm to obtain results.

Though an incomplete graph can be converted into a complete graph, the conversion method mentioned in section 4.1 is far from perfect and does by no means guarantee that a solution can be found. The Multi-agent system discussed in this chapter on the other hand *is* able to handle incomplete graphs in contrast to the basic Ant Colony Optimization algorithm and many others.

Chapter 7

Implementation and results

This chapter presents the implementation and the obtained results.

7.1 Implementation

Following the design discussed in chapter 6, the system has been implemented using the Java programming language. The open source JGraphT graph library has been used to perform graph operations, and the also open source JGraph library has been used to visualize the graph, and the results of the system.

The system has been implemented according to the Model-View-Controller design pattern. The model in this case however, is the Multi-Agent System, which has autonomous control, so the system has been split in just two packages:

ControllingModel A package representing the model and the control, i.e. the MAS. Has responsibility of controlling the agents in the model.

View A view package. Has the responsibility of visualizing the ControllingModel.

Figure 7.1 shows the class diagram of the ControllingModel package. A graph consists as usual of a set of vertices and a set of edges. The Graph and Edge classes are part of the JGraphT library and allows the programmer to use any Object as the vertices of a graph. In this case the vertices are VertexAgent objects. The CycleAgent class is, as the VertexAgent class, a generalization of the Agent class. In addition a CycleAgent has information on which vertices it is made up of. Each Agent keeps information on its preferred edges, and the possible edge combinations from which to prefer its two edges. An Agent is able to perceive, decide and act. Each EdgeCombo corresponds to a combination of two rows in the tables in chapter 6. A Disturbance object is thus an object corresponding to one row in the mentioned tables. Disturbance objects and EdgeCombo objects are comparable, to facilitate the decisions of an Agent.

The ComponentManager class is responsible for keeping track of which edges the vertices have agreed on preferring (matches), and for keeping track of which edges have been made mandatory by the CycleAgents (mandatoryEdges). The ControllingModel class is responsible for controlling the computation of the system.

The View package is sufficiently simple that a class diagram is not needed – it simply visualizes the graph, the matching preferences of the vertices or both. All of

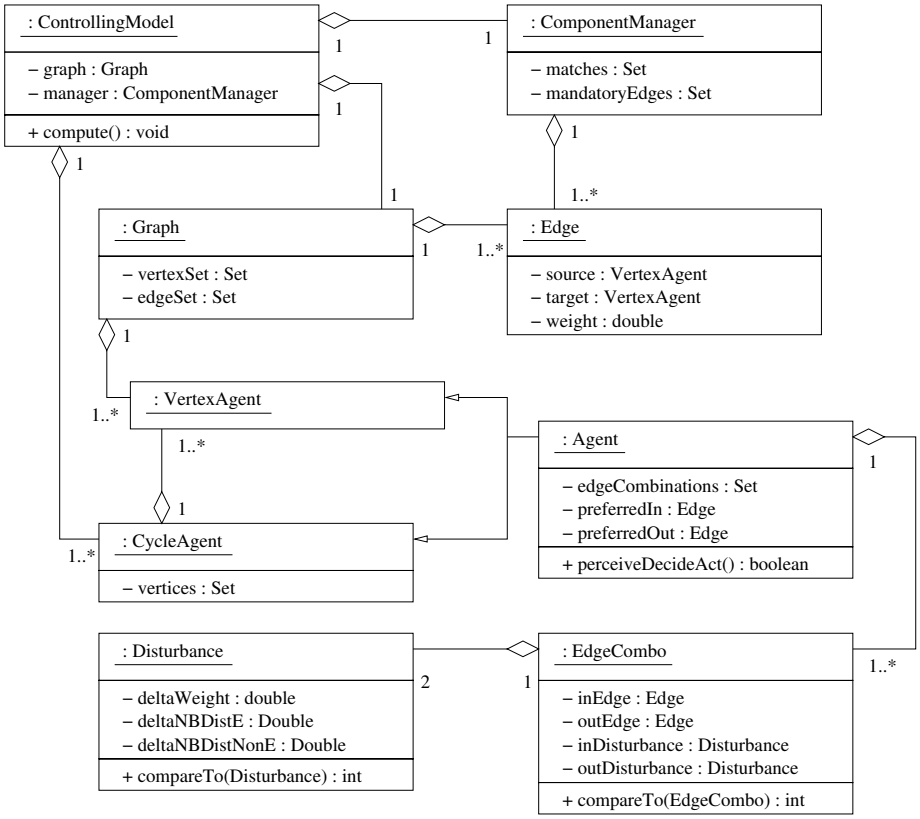


Figure 7.1: Class diagram of the ControllingModel package.

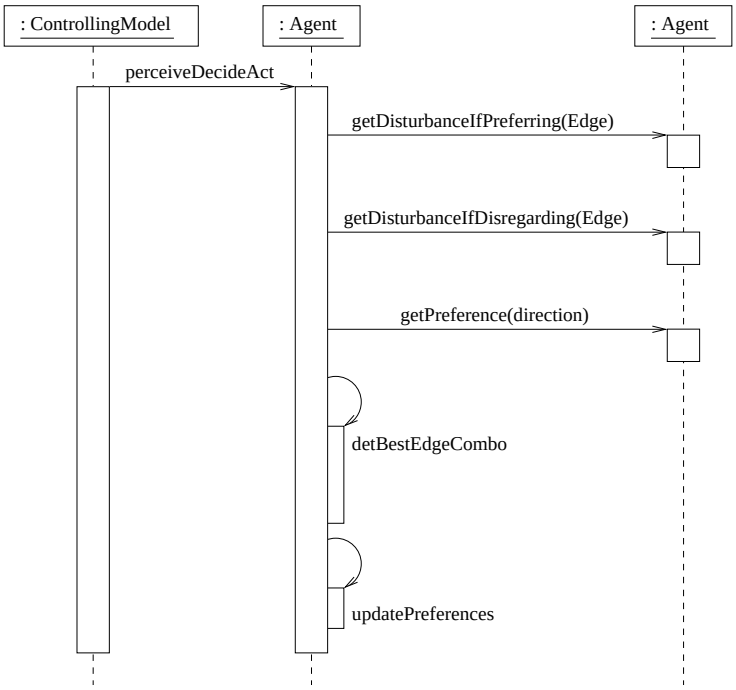


Figure 7.2: Sequence diagram for the perceive, decide, act cycle of an Agent.

its functionality is a result of the application of the JGraphT and JGraph software libraries.

The sequence diagram for the perceive, decide, act cycle of an Agent is shown on figure 7.2. The ControllingModel object starts the process, and the Agent then:

- Updates its knowledges of its environment, by querying its neighbours (getDisturbanceIfPreferring, getDisturbanceIfDisregarding and getPreference).
- Decides on the best edge combination to choose (implicitly using the comparable EdgeCombo and Disturbance objects related to its edges).
- And finally updates its preferences. Though not shown, the ComponentManager is informed of the preference – if there is a match with the neighbour the matches or the mandatory edges are updated reflecting if the Agent is a VertexAgent or a CycleAgent respectively.

The perceiveDecideAct cycle of an Agent returns a boolean value indicating of its preferences changed during the cycle – this value is then used by the ControllingModel to judge if the system has reached a stable state.

7.2 Results

The system, implemented according to chapter 6 and the previous section, is able to find the lightest Hamiltonian cycle for some graphs, while some other graphs result

in Hamiltonian cycles which are good, but not optimal. In addition the system needs an efficiency improvement in order to handle problems with real world size (vertex set cardinality: Tens of vertices and up).

Minor problems were encountered during the final work on this report. The design of the MAS seems right, so the conclusion at the moment is that an undiscovered error is present in the implementation of the system.

The following sections presents the results obtained from tests on complete, undricted, euclidean graphs specified by the cartesian coordinates of vertices.

7.2.1 Problem instances

Figure 7.3 shows the set of cycles obtained by the first pass at the micro level for the `eil51` problem instance of the TSPLIB[Rei95]. It can be seen that though some very bad edges have been chosen, all cycles are made up of (sub)paths from the optimal cycle: (va1, va22, va8, va26, va31, va28, va3, va36, va35, va20, va2, va29, va21, va16, va50, va34, va30, va9, va49, va10, va39, va33, va45, va15, va44, va42, va40, va19, va41, va13, va25, va14, va24, va43, va7, va23, va48, va6, va27, va51, va46, va12, va47, va18, va4, va17, va37, va5, va38, va11, va32, va1).

Because of the inefficiency problem mentioned in section 6.5 no further results have been obtained from this problem instance.

Working instances

This section shows a couple of simple problem instances for which the system has worked as intended.

`example6.tsp` On figure 7.4 the set of cycles after the first pass at the micro level is shown. On figure 7.5 the single cycle obtained from the subsequent pass at the macro level and a following pass at the micro level is shown. It can be seen that the cycle is Hamiltonian and that it is optimal.

`example8.tsp` Similar to the above example, figure 7.6 and figure 7.7 shows the results of the first micro level and the last micro level. The found cycle is Hamiltonian and optimal.

A problematic instance

As mentioned the system is not able to solve the `eil51` problem instance. In addition when reducing the problem instance to a subset of six vertices (va14, va25, va13, va41, va19 and va40) the system though finding a Hamiltonian cycle, it does not find the shortest Hamiltonian cycle, as shown on figure 7.8 and figure 7.9.

Figure 7.3: The first set of cycle agents when operating on the `eil51` instance.

Figure 7.4: The set of two cycle agents when operating on the instance briefly discussed in chapter 6.

Figure 7.5: The Hamiltonian cycle obtained when operating on the instance briefly discussed in chapter 6.

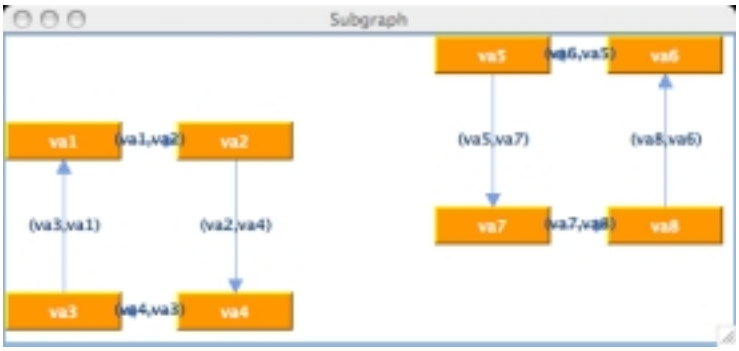


Figure 7.6: The set of two cycle agents after the first pass at the micro level.

Figure 7.7: The obtained Hamiltonian cycle.

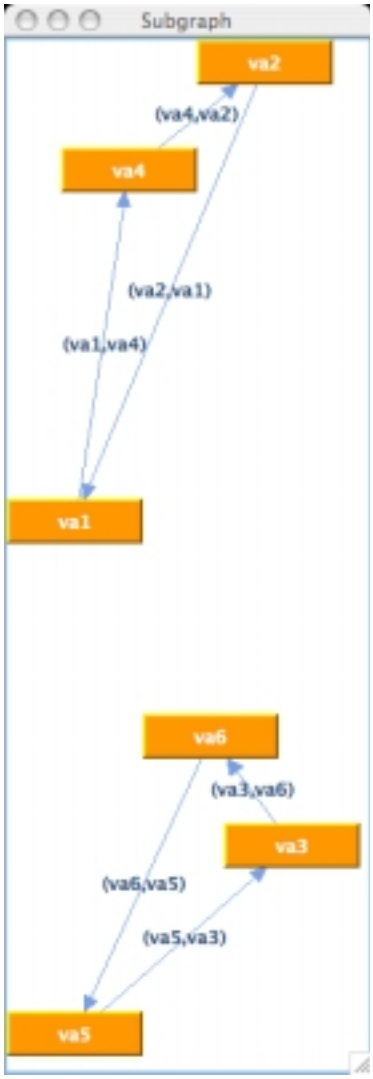


Figure 7.8: The set of two cycle agents after the first pass at the micro level.

Figure 7.9: The obtained Hamiltonian cycle.

Chapter 8

Conclusion

The project has made it possible to obtain a better understanding of the notions of the AMAS theory while applying the theory to a concrete, practical problem: The Travelling Salesman Problem.

The Travelling Salesman Problem is not as simple a problem as it could seem, in particular it has been the basis for numerous publications and even whole books[Rei94]. The Travelling Salesman Problem, in addition to being one of the most widely known NP-complete problems, it is as relevant for further studies as it was when Sir William Rowan Hamilton in 1856 formulated the Hamiltonian Cycle Problem – a simpler version of the Travelling Salesman Problem. In spite of the amount of work done, the Travelling Salesman Problem is still the center of attention for numerous research activities. Competitions on the World Wide Web are held among students and researchers in the attempt to discover novel ways of solving the problem.

The purpose of the project was to examine the feasability of applying the emergent behaviour of a Multi-Agent System to the problem.

The Multi-Agent System have been constructed using the ADELFE (Atelier de DÉveloppement de Logiciels à Fonctionnalité Emergente) methodology conceived by the SMAC research group at IRIT. The adequacy tool of the ADELFE toolkit have shown that the AMAS theory is an applicable approach to solve the TSP – in other words, finding a solution to the TSP can be accomplished by a set of locally cooperative, autonomous agents. ADELFE has, in addition, facilitated the design of a two-level Multi-Agent System architecture – consisting of vertex agents and cycle agents, seeking to maintain cooperative to their neighbouring agents.

Only partial results have been obtained (cf. section 7.2): The system, implementing the behaviour explained in chapter 6, is able to solve small, simple cases of the problem to optimality. But if attempts are made at solving larger, more complex problem instances, the behaviour of the system does not result in *optimal* Hamiltonian cycles – the obtained Hamiltonian cycles are near-optimal though.

8.1 To do

In the time to come, work will be done to discover the source causing the incorrect results.

If the problem is resolved, the next step will be to compare the obtained results with those of existing methods for solving the TSP.

The problem will be sought resolved within the next two weeks, so that results should be available at the presentation of this report.

8.2 Extensions

The distributed, adaptive nature of an Adaptive Multi-Agent System makes it natural to consider the possibility of extending the system, giving the user the ability to modify the graph while the system is online; letting the system dynamically adapt to the modifications made. E.g. if a user removes an edge of the graph, the system should adapt and if the edge was part of the cycle, find a new Hamiltonian cycle not including the removed edge.

According to the AMAS theory the system should be able to adapt to these modifications of the environment.

This aspect of adaptivity to online modifications, has to our knowledge not yet been an integral part or property of methods solving the TSP. In this light it would be very interesting to reach such a result. And not least evaluate the solutions and efficiency of the system.

8.3 Possibilities for future studies

A proof that an AMAS is adequate for solving the TSP or any other combinatoric optimization problem (to optimality) could possibly be constructed using the axiom of the AMAS theory. Keywords are: State space, non-linear dynamics, attractors, fix points, discretization by Poincaré sections. A description of an analogous proof can be found in [Joh03] and in more detail in [FA04].

Bibliography

[Bry92] Victor Bryant. *Aspects of Combinatorics: A wide-ranging introduction*. Cambridge University Press, 1992.

[Bur03] Forbes J. Burkowski. Proximity and priority: Applying a gene expression algorithm to the traveling salesperson problem. In *IPDPS*, page 142. IEEE Computer Society, 2003.

[Cam98] Valérie Camps. *Vers une théorie de l'auto-organisation dans systèmes multi-agents basée sur la coopération : application à la recherche d'information dans une système d'information répartie*. PhD thesis, IRIT, Université Paul Sabatier, 1998.

[CK04] P. Crescenzi and V. Kann. A compendium of np optimization problems. WWW: http://www.nada.kth.se/∼viggo/problemlist/compendium.html, 2004.

[DACH04] V. Chvátal D. Applegate, R. Bixby, W. Cook, and K. Helsgaun. Optimal tour of sweden. Web, May 2004. http://www.tsp.gatech.edu/sweden/index.html.

[DG96] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the traveling salesman problem. Technical Report TR/IRIDIA/1996-3, Université Libre de Bruxelles, Belgium, 1996.

[DZ94] Martin Dam and Martin Zachariasen. Tabu search on the geometric traveling salesman problem. Master's thesis, Department of Computer Science, University of Copenhagen, 1994.

[FA04] Jes Frellsen and Jesper Stemann Andersen. Bachelorprojekt. datamatlæring baseret på informationsteori, May 2004. Department of Computer Science, University of Copenhagen.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[Hel98] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. Technical report, Roskilde University, Denmark, 1998.

[JGM+02] D. Johnson, G. Gutin, L. McGeoch, A. Yeo, W. Zhang, and A. Zverovitch. Experimental analysis of heuristics for the atsp, 2002.

[Joh03] Peter Johansen. A robot control algorithm. lecture note for the cisp phd school may 5-8 2003, April 2003.

[JPGG04] Bruce Edmonds Jean-Pierre Georgé and Pierre Glize. *Methodologies and Software Engineering for Agent Systems, The Agent-Oriented Software Engineering Handbook*, chapter 16, Making Self-Organising Adaptive Multiagent Systems Work – Towards the Engineering of Emergent Multiagent Systems. Kluwer Academic Publishers, 2004. Editors: Federico Bergenti, Marie-Pierre Gleizes and Franco Zambonelli.

[Kim97] Jaegwon Kim. Explanation, prediction, and reduction in emergentism. *Intellectica*, 2(25), 1997.

[KS85] R.M. Karp and J.M. Steele. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, chapter 6, Probabilistic analysis of heuristics. John Wiley and Sons, Chichester, 1985. Editors: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys.

[Kun01] Daniel R. Kunkle. Self-organizing computation and information systems: Ant systems and algorithms. WWW: http://www.redfish.com/dkunkle/mypapers/selfOrgAnts.pdf, 2001.

[PG04] Gauthier Picard and Marie-Pierre Gleizes. *Methodologies and Software Engineering for Agent Systems, The Agent-Oriented Software Engineering Handbook*, chapter 8, The ADELFE Methodology. Kluwer Academic Publishers, 2004. Editors: Federico Bergenti, Marie-Pierre Gleizes and Franco Zambonelli.

[Rei94] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*, volume Lecture Notes in Computer Science, 840. Springer-Verlag, Berlin, 1994. G. Goos and J. Hartmanis.

[Rei95] Gerhard Reinelt. Tsplib. WWW: http://www.iwr.uni-heidelberg.de/groups/comopt/soft/TSPLIB95/TSPLIB.html, 1995.

[RN02] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2002.

[SD99] Thomas Stützle and Marco Dorigo. *Evolutionary Algorithms in Engineering and Computer Science*, chapter ACO Algorithms for the Traveling Salesman Problem. Wiley and Sons, 1999. Editors: K. Miettinen, M. Mäkelä, P. Neittaanmäki and J. Periaux.

[Ski97a] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Ski97b] Steven S. Skiena. The algorithm design manual. WWW, 1997. http://www.cs.sunysb.edu/ algorith/files/hamiltonian-cycle.shtml.

[THCS01] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[Tho97] Carsten Thomassen. On the complexity of finding a minimum cycle cover of a graph. *SIAM J. Comput.*, 26(3):675–677, 1997.

[Woo02] M. Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley and Sons, 2002.

[ZD96] Martin Zachariasen and Martin Dam. *Meta-heuristics: Theory and Applications*, pages 571–587, Tabu Search on the Geometric Traveling Salesman Problem. Kluwer Academic Publishers, Boston, 1996. Editors: I.H. Osman and J.P. Kelly.

Appendix A

Application of ADELFE

In the following the section numbering refers to the activity and step numbers of ADELFE – thus represents section number A.6.1 activity no. 6, step no. 1.

As distribution is a central concept in Multi-Agent Systems it is required to handle a problem in the context of Multi-Agent Systems that the problem be specified in a distributed manner. The TSP is thus for the rest of this appendix specified as follows:

Definition A.1 *Given a connected, directed, weighted, non-euclidean graph, G, the goal is to create a subgraph, G', by eliminating all edges but one incoming and one outgoing edge for each vertex such that the resulting subgraph forms a cycle of the least weight possible.*

This definition distributes the task to be solved onto a set of tasks – each to be solved with regard to each vertex.

A.1 Defining the user requirements

This activity represents the first activity of the "Preliminary requirements" process which extends from Activity 1 to Activity 5.

This first activity concerns the description of the system and of the environment in which the system will be deployed. It consists in defining what to build or what is the most appropriate system for end-users.

End-users, clients, analysts and designers have to list the potential requirements. The context in which the system will be deployed must be understood. The functional and non-functional requirements must be established.

System

The system is a solver for the TSP as defined in definition A.1. It thus requires the definition of a connected, directed, weighted, non-euclidean graph and is then able to solve the TSP and display the solution in some form along with some quantifications on its computations. The purpose of the system is therefore to find the lightest cycle of a given graph visiting each vertex exactly once.

Input A connected, directed, weighted, non-euclidean graph.

Output A cycle of the graph – a list of vertices, edges – along with the weight of the edges and their sum.

The system should be able to create the graph from vertices specified by cartesian coordinates and the edges and their weight specified by a distance matrix.

Environment

The environment for the system is a human user who wants the system to solve the TSP for a graph of his choosing.

Functional requirements

The system should obviously solve the TSP.

The system should be constructed in such a way that it is possible to quantify its efficiency.

It should be sought that the adaptivity of the AMAS be exploited as an auto-optimizing factor in the solving of the problem – the system should adapt its solution towards an optimal solution.

Non-functional requirements

The system is to be implemented in Java and will be composed of a modular solver and interface making up a stand-alone program, which thus have few to none dependencies on other systems.

A.2 Validation of the requirements

In this activity, you must check and approve the requirements established in activity 1. If there is inconsistencies, you must go back to the previous activity in order to improve your requirement specification.

The requirements have been validated in cooperation with the relevant groups.

A.3 Consensual requirements

In this activity, you must update the requirements with consensual requirements. Here again, if there is inconsistencies, a backtrack must be performed to redo the previous activity.

Conditions/capabilities on which the end-users, designers and developers agree the system has to fulfill:

- The system should solve the TSP efficiently in a distributed manner.

A.4 Keywords

You have to list the main concepts used to describe the application and its domain (the system and its environment).

System The complete system solving the TSP.

User The user of the system.

Graph The graph for which a lightest cycle must be found, if possible.

Vertices The vertices of the graph.

Edges The edges of the graph.

Weighting The edge-weighting.

Paths The paths formed by the choosing of edges. An example: The two-edge paths chosen for each vertex.

Cycles The candidate cycles for being the lightest.

A.5 Extraction of limits and constraints

In this activity, you must define the limits and constraints of the system you want to build (your application). They can be found in the expression of non functional requirements and in the definition of the context in which the system will be deployed.

To summarize the requirements (limits and constraints) for the system:

- The graph[1] is connected, directed, weighted and non-euclidean.
- Nothing can be said on the completeness of the graph.
- Thus the graph might or might not be Hamiltonian[2] – and thus it can not be said in advance if a solution can be found.
- The resulting subgraph is to be composed of an ordering of the vertex set of the original graph, and an edge set forming a Hamiltonian cycle in the graph of minimal weight.

A.6 Characterization of the Environment

The characterization of the environment is the first activity of the "Final requirements" process which extends until Activity 10. The main objective of this activity is to define the system environment

A.6.1 Determination of the Entities

In this step you must identify active/passive entities in interaction with the system and constraints on these interactions.

The list below denotes the identified entities of the system and marks them as either found active or passive.

System Active – the system is responsible for starting the calculation and obtaining the results for the user at the end of the calculation.

User Active – obviously.

Graph Passive – no autonomous behavior possible.

Vertices Active – each can be given the choice of autonomously choosing those two among their neighbours they prefer being their connections to other vertices. Cf. definition A.1 on page 67.

[1] Cf. definition 3.1 on page 13.
[2] Cf. definition 3.4 on page 14.

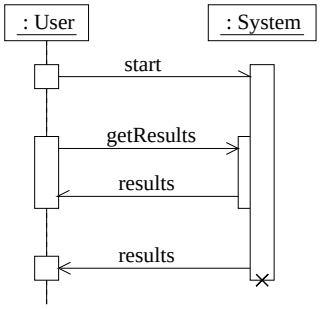


Figure A.1: Sequence diagram for the interactions between the user and the system.

Edges Passive – the possible choice of an edge, including or excluding itself in the problem solution, does not outweigh the fact that the edges form a resource to be used by the vertices.

Weighting Passive – the weight-function is obviously a passive resource to the system.

Paths Passive – though the path between two vertices can have multiple forms, a specific path is something static with no autonomy.

Cycles Passive – a description equivalent to the one for paths can be made for cycles.

A.6.2 Define context

In this step you must characterize data streams and interactions between entities and the system. Data streams between passive entities and the system are expressed using collaboration diagrams. Interactions between active entities and the system are expressed using sequence diagrams.

The only entity exterior to the system is the user. He is obviously an active entity and there is thus no passive entities exterior to the system. His interactions with the system are:

1. Start computation for a graph.
2. Obtain the results.

These interactions are visualised in the sequence diagram on figure A.1.

A.6.3 Characterization of the environment

Characterization of the environment according to the usual characteristics[3] *with the system (still) being the system solving the TSP and the environment being the user.*

Accessible Yes – while the system is functioning the environment is fully observable – everything the system needs to know is accessible[4].

Deterministic No – the system can not foresee what action the user will take in the future based on its actions.

[3] Cf. section 2.3 in [RN02].

[4] Fully observable is the term used by [RN02]

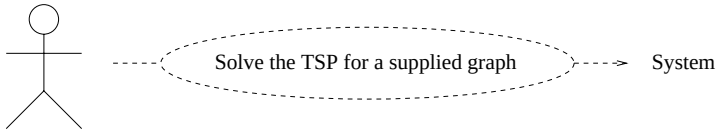


Figure A.2: Diagram of the general use case.

Static Yes – the user does not change parameters relevant to the system during execution, ie. the graph.

Discrete Yes – the user does not change the graph.

A.7 Determination of use cases

The main objective of this activity is to clarify the different functionalities that the studied system must provide. Later, these functionalities will be clustered in one or several use case diagrams between the previously identified active entities and the system.

A use case is detailed using a textual description and specific sequence diagrams. If needed, a use case can have an "exceptions" box in which the conditions under which the service cannot be provided must be mentioned.

A.7.1 An inventory of use cases

Firstly, during this step, you must identify all the different use cases that may exist for the studied system.

The general use case is the following: The user wants the system to solve the TSP for a graph and thus supplies the system with the graph. The system then determines if the graph is Hamiltonian and if so, replies with the lightest Hamiltonian cycle found. The use case is visualised on figure A.2.

A.7.2 Identify Cooperation Failures

During this step, you must think about the events that may lead to situations that are not totally controlled by the system designer and thus can be seen as "harmful".

The aim of this step is to point at "wrong" interactions between entities and the system. The AMAS theory calls these events and situations "cooperation failures". These events can be viewed as a kind of "exceptions".

If the graph is Non-Hamiltonian it is an exception[5] compared to the normal run of the system. In this case the system should reply with an error message saying that it is impossible to find a Hamiltonian cycle for this graph. But as mentioned in section 3.2 on page 14 it is in general not possible to come to such a conclusion in reasonable time. The weak criterias mentioned in section 3.2 on page 14 will thus have to be used.

[5]The term exception is used instead of cooperation failure since it does not seem natural to talk about cooperation between a user and a system.

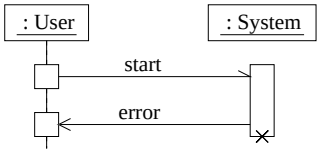


Figure A.3: The sequence diagram in case of the graph not being Hamiltonian.

A.7.3 Elaborate Sequence Diagrams

For each previously established use case, you must now give a corresponding sequence diagram.

The sequence diagram for the first use case is shown on figure A.1 on page 70. The sequence diagram for the case of the graph not being Hamiltonian is shown on figure A.3.

A.8 Development of UI prototypes

In this activity, you must specify, the UIs through which the user(s) will interact with the system and the links between the UIs.

The system should be able to be fed with a graph in a convenient manner – input in the form of a textfile should be a minimum requirement. Textbased output showing the result and timing would also be a minimum requirement for the output of the system.

It could be nice to show the graph and the developing subgraph (the cycle) as the agents operate. For non-euclidean graphs it will be a problem to visualize the graphs. The user should be able to disable advanced output to concentrate on the computations. The quantifications on the calculation time should of course not include input or output.

A.9 Validation of the UI prototypes

The UIs described in the previous section have to be judged from functional or non functional (ergonomic, design...) points of view.

The user interface have been validated in cooperation with the relevant groups.

A.10 Analysis of the Domain

Domain analysis is a static view and an abstraction of the real world established from the requirements and the keywords. Considering separately each use-case by defining scenarii, the designer has to divide the system into entities. These entities may be concrete (e.g. teacher) or abstract (e.g. time slot). The result of this step is a set of entities in preliminary class diagrams

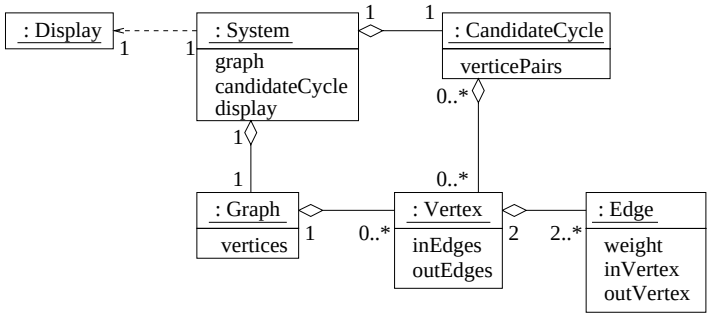


Figure A.4: Preliminary class diagram.

A.10.1 Identify Classes

In this step you have to make use of the previously defined use cases, the corresponding sequence diagrams and the keywords. You can then identify needed classes.

Needed classes identified from the entities, use cases and corresponding sequence diagrams:

System Responsable for treating user input, commencing calculation and returning results.

Display A class for showing results, timings and computation steps. Textual at first.

Graph A class representing the graph.

Vertices The vertices of the graph.

Edges The edges of the graph.

Cycle A class to represent the result.

The weighting of the edges seems more natural to represent as an attribute of an Edge class than a class itself.

A Cycle class is included to represent the result whereas a Path class is not included as this concept does not play a central role in the system.

A.10.2 Study Interclass Relationships

You now have to study the interactions between the different previously identified classes by studying the existing use case and sequence diagrams.

The class relationships are visualized in the class diagram in figure A.4.

A.10.3 Construct the preliminary class diagrams

Once the different classes and their interactions identified, you now have to construct the preliminary class diagram.

The preliminary class diagram is shown on figure A.4.

A.11 Verifying the adequacy of an AMAS

This activity is the first one of the "Analysis" process which extends from Activity 11 to Activity 13.

In this activity, you must verify whether one (or more) Adaptive Multi-Agent System (AMAS) is needed to realize the system you want to build. In some cases an AMAS is not needed. For example, if the algorithm required to solve the task is well-known or if the task is sufficiently simple[6]*, an adaptive system is not needed.*

The verification is done at two levels:

- *At the global level to answer to the question "is an AMAS required to implement the system?"*
- *At the local level to try to determine if some agents needs to be implemented as an AMAS themselves, i.e. if a composite or recursive structure can be identified in the current design of the system.*

A.11.1 Verification at the global level

1. *Is the global task incompletely specified? Is an algorithm a priori unknown?* Yes, the global task is incompletely specified and no effective (polynomial-time) algorithm is known. (Score: 17)
2. *If several entities are required to solve the global task, do they need to act in a certain order?* Nothing can be said on the need for correlated activity for several entities though it may seem likely. (Score: 10)
3. *Is the solution generally obtained by repetitive tests, are different attempts required before finding a solution?* Yes, normally you would incrementally improve the cycle found with regard to its weight until few to none improvements occured. (Score: 20)
4. *Can the system environment evolve? Is it dynamic?* No, the environment is static as specified earlier. (Score: 0)
5. *Is the system process functionally or physically distributed? Are several physically distributed entities needed to solve the global task? Or is a conceptual distribution needed?* Nothing can be said in advance on the physical or functional distribution, though many graphs may be physically partitionable with a minimal cycle consisting of minimal paths/cycles of the graphs partitions. (Score: 10)
6. *Is a great number of entities needed?* Uncertain; though one can say that a large number of entities(vertices and edges) will be used this number is linearly and not exponentially dependent on the size of the graph since nothing can be said the connectivity of the graph. Thus nothing can be said on the number of entities needed. (Score: 10)
7. *Is the studied system non-linear?* Yes, if you imagine the active entities negociating and adapting towards an optimal solution, then the system is definitely non-linear. (Score: 20)
8. *Finally, is the system evolutionary or open? Can new entities appear or disappear dynamically?* No, the system is closed – no vertices or edges are added, removed or modified. (Score: 0)

[6] Or if the system is closed and nothing unexpected can occur.

A.11.2 Verification at the local level

9. *Does the entity need more knowledge than what is available for it to act?* No, each vertex needs only to interact with entities within its own realm – its neighbours. (Score: 0)
10. *Is an entity "big" or not? Is it able to do many actions, to reason a lot? Does it need great abilities to perform its own task?* Uncertain, but likely; On one hand the problem of finding a solution at first looks like it requires human oversight and reasoning, but in this case with vertices confined to their own small realm they do not have too much to "think" about. The first reason is weighted slightly higher than the latter. (Score: 13)
11. *Can the behaviour of an entity evolve? Does it need to adapt to the changes of its environment?* No, the entities will keep the same behavior throughout their "life". (Score: 0)

The result of the adequacy verification by the ADELFE adequacy tool is shown in figure A.5.

A.12 Identifying the Agents

This activity aims at finding what we will consider as agents in the system to be. We are only interested in agents which enable a designer to build an AMAS. These agents are looked for among the entities defined in Activity 6-Step 1 and classes that have been previously defined (Activity 10-Step 1).

A.12.1 Study of the entities in the domain context

The parts of the system will here be evaluated as candidates to an agent role. For each entity defined in the Activity 6-Step 1, you have to decide if it:

- *is autonomous,*
- *has a local goal to pursue,*
- *has to interact with some other entities,*
- *and if it:*
  - *has a partial view of the environment (this means that an entity can only perceive a small part of its environment, that it has only a partial knowledge about it),*
  - *has some abilities of negotiation.*

Entities verifying all the three first criteria may be viewed as agents.

Vertices

Autonomy Yes, each vertex could have the ability to choose one incoming and one outgoing edge to use to connect itself to its neighbours.

Local goal Yes, a local goal of a vertex could be for it to find the combination of an incoming and an outgoing edge resulting in the lightest two-edge path.

Interaction partners Yes, a vertex' interaction partners would thus be its neighbour vertices.

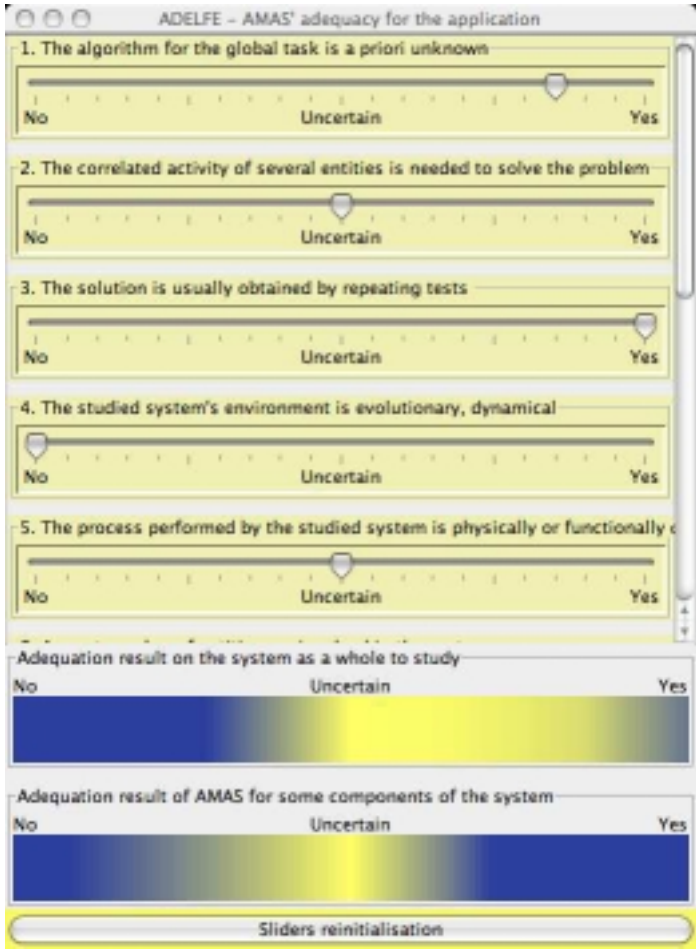


Figure A.5: Adequacy result of the ADELFE adequacy tool.

Partial view Yes, as the vertex is restricting its interactions to a limited set of vertices (its neighbours) it does not need to have a global view of its environment.

Negociation ability Yes, the vertex would be able to negociate with its neighbours on the choosing of incoming/outgoing-edge combinations.

Thus vertices is a candidate for the agent role – as well as being an intuitive choice playing the central role in the problem formulation: "Find a lightest cycle of a graph visiting every *vertex* exactly once" – even though the goal is to find a lightest cycle, every cycle evaluated *must contain every vertex* in the graph.

Edges

Autonomy Yes, edges could have the ability to negociate with its neighbour edges on who should be part of a path from one end-point(vertex) to another.

Local goal No, the edge would not have a persistent local goal to pursue as the negociation with neighbour edges could lead to itself being excluded – thus the elimination of not just itself, but also its goal.

Weighting

Autonomy No, the weight-function can not be allowed to change the weighting at its own will.

Graph, paths and cycles

The graph is too global a part of the problem for it to have a local goal to pursue. Likewise can paths and cycles be excluded on the basis of them being subgraphs.

The vertices are thus at this point the only candidate for being the agents of the system.

A.12.2 Identifying potentially cooperative entities

During its interactions with other entities or with the environment, an entity can encounter failure to respect the protocol or failure in the content of the interaction (misunderstanding...). These failures can be the consequences of a dynamic environment, the openness of the system... They are called Non Cooperative Situations. For each entity coming from the previous step, you have to determine if it:

- *Has to move in a dynamic environment.*
- *Has to face up to cooperation failures,*
- *Has to treat Non Cooperative Situations.*

Only the vertices are left to consider.

Dynamic environment They have to act in a dynamic environment – their neighbours actions affect the environment of the vertex.

Cooperation failures Their communication protocol will be fully adapted to its use, thus they will encounter no communication(cooperation) failures (NCSs of type incomprehension, ambiguity or uselessness).

Non Cooperative Situations A vertex which has yet to accomplish its goal can be seen as being in a NCS.

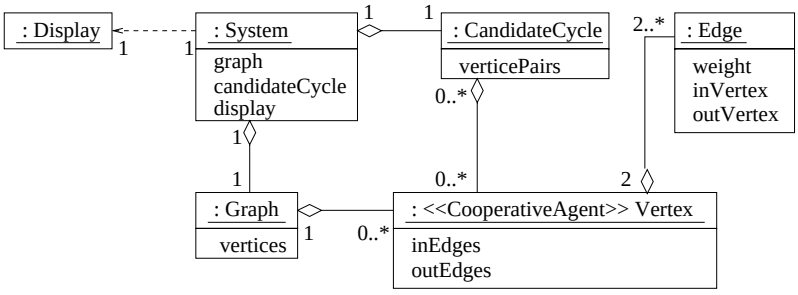


Figure A.6: Class diagram with the Vertex class marked with the CooperativeAgent stereotype.

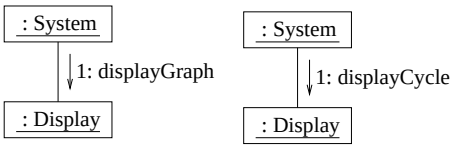


Figure A.7: Collaborations between the system and the display.

A.12.3 Final determination of agents

The entities coming from the previous step can be now considered as agents.

The Vertex class is marked with the CooperativeAgent stereotype on figure A.6.

A.13 Interactions between entities

This activity aims at identifying interactions between the entities previously identified in Activity 6-Step 1.

A.13.1 Relations between active and passive entities

In this step, relations between active and passive entities are expressed using sequence diagrams or collaboration diagrams.

There is only one active entity which is not the user and which is not an agent: The system. The collaboration diagrams between the system and the passive entities are shown on figures A.7 and A.8.

A.13.2 Relations among active entities

In this step, you will express the relations between active entities by sequence diagrams.

There in only one instance of the system, and thus it does not interact with itself.

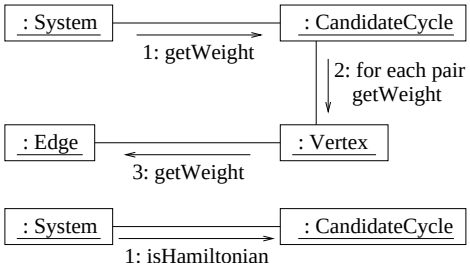


Figure A.8: Collaborations between the system and the candidate cycle.

A.13.3 Agent relations

In this step, protocol diagrams will be used to express relations between agents.

At this point, one is not able to describe, using AUML diagrams, the interactions between the vertex agents.

A.14 The detailed architecture

This is the first activity of the design process which extends from Activity 14 to Activity 18. If the system needs to be built as a recursive multi-agent system and is thus composed of several levels (system one, components one, etc.), this design process must be applied to each of the levels. The main objective of this first activity is to define the detailed architecture of the system in terms of needed packages, classes, objects and agents, to possibly refine this architecture by using some design patterns and/or re-usable components and then to create component and class diagrams.

A.14.1 Determine packages

This step requires to identify different packages.

Graph Representing graphs, vertices, edges and other graph theoretical terms.

Agent A package containing classes for the agents and their interactions – aggregating classes of the Graph package.

View A view package, according to the Model-View-Controller (MVC) design pattern. Has the responsibility of visualizing the model. The model is represented in the graph and agent packages.

Control A control package, according to the MVC pattern. Has responsibility of controlling the agents in the model.

A.14.2 Categorize classes

In this step, you have to categorize the different classes appearing in the preliminary class diagram (Activity 10-Step 3) by selecting which newly defined package they are a part of.

The packages consists of:

Graph Consists of the Graph, Vertex, Edge and Cycle (CandidateCycle) classes.

Agent Consists of a VertexAgent class.

View Consists of the Display class.

Control Consists of the System class.

A.14.3 Use Design Patterns

In this step you have to determine whether the previously defined architecture can be refined by the use of some design patterns and/or re-usable components.

The MVC design pattern has already been chosen, no further application of design patterns seem relevant.

The JGraphT, graph manipulation package can be used as the implementation of the Graph class. It seems very efficient and well designed, so using this package will be better than implementing a custom graph package.

In addition the JGraph package, can be used to display the class – it needs a little bit of customization, so it will be used from within the View package.

A.14.4 Elaborate Component and Class Diagrams

For each created package, you have to elaborate a component diagram and a class diagram.

Omitted – not relevant at this stage.

A.15 Study Interaction Languages

This activity consists in defining the way in which agents are going to interact. If agents interact to communicate, for each scenario (defined in Activities 7 and 13), you have to describe information exchanges between agents. Technically, these protocols will be specified through protocol diagrams using the AUML notation. Note that defining an interaction language is useless if no direct communication is used by agents (for instance, if they only communicate in an indirect manner via the environment).

Omitted – not relevant at this stage.

A.16 Design Agents

In this activity you may refine cooperative agent stereotyped classes defined in the Detailed Architecture (see Activity 12) and the Interaction Languages (see Activity 15). Agents (based on the AMAS theory) have a specific architecture. The designer has to embed this architecture which is composed of different parts. In addition the parts of the agents lifecycle (perceive, decide, act) must be defined during this activity.

A.16.1 Define skills

Skills: Knowledge about the domain enabling the agent to perform actions[7].

A chosen edge is an edge chosen to connect the vertex to other vertices in the cycle under construction. A vertex-agent possesses the following skills:

[7] A specific action of an agent can be seen as a skill as well.

- Choose or unchoose an incoming edge.
- Choose or unchoose an outgoing edge.
- The ability to query the situation of neighbour vertices.

The agent can represent its choice of edges by two preferences – one incoming and one outgoing.

A.16.2 Define aptitudes

Aptitudes: Abilities to reason on the knowledge of the agent or on its world representation.

A preference can be evaluated by how good a choice it is, e.g. if the preferred incoming edge is the lightest incoming edge of the vertex agent, the preference is optimal (if one disregards the neighbours of the vertex). As a consequence, the agent can evaluate its own situation – by evaluating the combination of its two preferences. To be coherent with the cooperativity concept of the AMAS theory, the evaluation measure in this context is named the "disturbance measure" or "disturbance level" – to reflect that being cooperative means *not* disturbing ones partners. Naturally a disturbance measure must be *minimized not maximized*.

Based on the domain knowledge each agent can decide its actions based on the following means of reasoning:

- It has the ability to evaluate its own situation – the disturbance level of its preferred edges.
- Given the disturbance level of itself and its neighbours by changing the preference of a given edge, it is able to deduce the local change in disturbance level.
- Given the preferences of itself and its neighbours, it is able to conclude if its preferences match with those of its neighbours.
- The change in disturbance level and the preference information can be used to compare different actions.

Because the agent is cooperative, the agent can happen to prefer an edge which is not its lightest if its for the greater good of its neighbourhood.

A.16.3 Determine interaction languages

Interaction language: Enabling the agent to communicate with other agents.

The vertex-agent will use message passing as its interaction language.

A.16.4 Determine world representation

World representation: A representation of itself, other agents and the surrounding environment.

The world representation of a vertex, v, is:

- Incoming and outgoing edges (including their weight), denoted by the sets $E_{v_{in}}$ and $E_{v_{out}}$.
- Neighbour vertices.
- Two preferred edges, one incoming p_i and one outgoing p_o.

Name	Inutility
Conditions	Either no incoming or no outgoing edge is preferred.
Actions	Make a preference in the direction lacking a preference.

Table A.1: NCS no. 1

Name	Unproductiveness
Conditions	The two preferred edges are not the least disturbing to the vertex and its neighbours.
Actions	Prefer the least disturbing (preferably the lightest) edges.

Table A.2: NCS no. 2

A.16.5 Define Non Cooperative Situations

You have to give rules which allow the agent to have a cooperative attitude: how to detect and to remove Non Cooperative Situations (NCS) in order to be more cooperative.

Before the non-cooperative situations can be defined we must define what it means for an agent to be in a cooperative situation. The ideal cooperative situation of an agent, v, is:

a. Exactly one incoming edge p_i is preferred.

b. Exactly one outgoing edge p_o is preferred.

c. The preferred incoming edge p_i causes the least disturbance locally.

d. The preferred outgoing edge p_o causes the least disturbance locally.

The set of non-cooperative situations are defined as the disjunction of the negations of the above conjunction:

$$NCS = \neg a \vee \neg b \vee \neg c \vee \neg d$$

This results in the five tables listing the non-cooperative situations – the condition for their occurence and the action needed to handle them.

Name	Inutility
Conditions	An agent A have two edges with equal disturbance value, so it cannot decide which edge to prefer solely from the disturbance value.
Actions	Cooperate with neighbours to prefer the edge that matches the preferences of its neighbours the best.

Table A.3: NCS no. 3

Name	Conflict
Conditions	An agent A have preferred the incoming edge p_i, but the agent B at the other end of p_i has not chosen p_i as its outgoing edge.
Actions	Cooperate with neighbour(s) to reach an agreement.

Table A.4: NCS no. 4

Name	Conflict
Conditions	An agent A have preferred the outgoing edge p_o, but the agent B at the other end of p_o has not chosen p_o as its incoming edge.
Actions	Cooperate with neighbour(s) to reach an agreement.

Table A.5: NCS no. 5