AN INVESTIGATION ABOUT PROCESS MATCHMAKING PERFORMANCES OF UNSTRUCTURED AND DECENTRALIZED DIGITAL ENVIRONMENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY

BY

BUĞRA ÇAKIR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING
JANUARY 2007

Approval of the Graduate School Of Natural And Applied Sciences

Prof. Dr. Selçuk Soyupak
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. İbrahim Akman
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Instructor Kasım Öztoprak
Co-Supervisor

Asst. Prof. Dr. Hürevren Kılıç
Supervisor

Examining Committee Members

Asst. Prof. Dr. Hürevren Kılıç

Assoc. Prof. Dr. Veysel Gazi

Prof. Dr. Faruk Polat

ABSTRACT

AN INVESTIGATION ABOUT PROCESS MATCHMAKING PERFORMANCES OF UNSTRUCTURED AND DECENTRALIZED DIGITAL ENVIRONMENTS

ÇAKIR, BUĞRA
M.S. Department of Computer Engineering
Supervisor: Asst. Prof. Dr. Hürevren Kılıç
Co-Supervisor: Instructor Kasım Öztoprak

JANUARY 2007, 67 pages

Efficient matchmaking is an important problem in unstructured and decentralized digital environments. We want to investigate the performances of those environments which will good fitting the nature of matchmaking problem. For this purpose in this thesis, different environment models including P2P (Hypergrid, Gnutella with/without ultrapeer); small-world (Watts-Strogatz); heavy-tailed and random versions of Autonomous System Waxman model showing power-law distribution property; random and 2D-Grid are considered. The flooding mechanism enabling process encounters for match purpose is uninformed Breadth-First-Search. In order to test the environments we made performance simulations with the matchmaking problem. Simulations show that the matchmaking performance of random environment outperforms the others for almost all different problems and time-to-live settings. On the other hand, the total cost of small world environment model is the highest for allmost all setups.

Keywords: Unstructured and Decentralized Environments, Process Matchmaking, Performance, Simulation.

Öz

YAPISAL VE MERKEZİ OLMAYAN SAYISAL ORTAMLARIN SÜREÇ EŞLEME PERFORMANSLARI HAKKINDA BİR İNCELEME

ÇAKIR, BUĞRA
M.S. Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Hürevren Kılıç
Ortak Tez Yöneticisi: Okutman Kasım Öztoprak

Ocak 2007, 67 sayfa

Verimli eşleme, yapısal ve merkezi olmayan sayısal ortamlarda problem teşkil etmektedir. Bu tezde sözkonusu ortamlara odaklanılarak, ortamların eşleme performanslarını ölçme amaçlı benzetimler gerçekleştirilmiştir. Çalışmada, P2P(HyperGrid, başuç'lu (ultra-peer) ve başuç'suz Gnutella), Küçük-dünya (Small-World, Watts-Strogatz), ve üs kanunu (power-law) dağılımını gösteren Özerk Waxman sisteminin ağır-kuyruklu (heavy-tailed) uygulaması, rasgele (random), iki boyutlu-ızgara (2D-Grid) gibi farklı ortam modelleri dikkate alınmıştır. Eşleme amacıyla, süreç karşılaşmalarına olanak tanıyan akış mekanizmasının *Bilgilendirilmemiş Önce-Enlemesine-Arama* olduğu kabullenilmiştir. Benzetimlerde, rasgele ortamın eşleme performansının neredeyse diğer tüm problemlere ve eşzamanlı yapılandırmaya oranla performans üstünlüğü gözlenmektedir. Ancak, küçük-dünya ortamı modelinin toplam maliyeti neredeyse tüm kurulumlar içinde en yüksek olanıdır.

Sözlük Terimler: Yapısal ve merkezi olmayan ortamlar, Süreç eşleme, Performans, Benzetim

To Melek and My Family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor...

I deeply thank the members of the Department of Computer Engineering and all members of Atılım University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Designing an efficient digital environment is a major problem, on the contrary it has many factors which must be taken into consideration while investigating a solution. Firstly, [7] evaluated environments by looking at their properties. They enumarated several digital environments which are categorized by their top level properties (i.e unstructured P2P, structured P2P) and ran simulations on generated environments. *Random Graph* environment outperforms in their experiments. Furthermore there are some other available digital environments which could be under consideration in the evaluation of digital environments.

Thus it is a good practice to enumerate different types of environments during the evaluation of digital environments. For example in [18], *Information Networks*, *Social Networks* and *Biological Networks* have been considered and enumerated.

Although it is well seen that some environment is superior to the others by looking at environmental properties, however there are other important factors while evaluating different digital environments. Within the context of a digital environment we have to find the location of entities. Furthermore several mechanisms are proposed. In [2], different *agent location* mechanisms are evaluated. They use the term open MAS(Multi-Agent-Systems) for defining their digital envrironment. Also the term *agent* is used for each entity living in a digital environment. So they look at this problem from the MAS perspective while the mechanisms do not differ.

Furthermore *problem solving* is a major problem for a digital environment. Related to this, it is necessary to define a good representation of problem description. In [11], the problem description used in a digital environment is expressed by a-DFA(Annotated Deterministic Finite Automata). They propose a solution which fullfills the shortcomings

of *WSDL* approaches stated in [9]. They implement a process matchmaking engine [12]. However, this engine matchmake process descriptions in a central way which is not efficient and vulnerable to single point of failure. In [5], the problem is approached as a producer/consumer problem and they proposed a P2P solution where each agent relays its problem description to its neighbors only. Such a P2P mechanism seems efficient, on the other hand communication overhead should be considered.

In this thesis, we are investigating the matchmaking potentials of unstructured and decentralized digital environments. Firstly, unstructured and decentralized digital environments will be generated based on their generation models implemented as topology generator. Secondly, a process matchmaking model proposed in [25] is implemented. Thirdly, the experimental setup is introduced. Finally, simulation results are interpreted.

The practical importance of the outcome of this thesis is to provide an idea about suitable digital environment topology for process matchmaking purposes. By this way, an engineer may decide on overlay level connectivity dynamics of the digital environment.

CHAPTER 2

DIGITAL ENVIRONMENTS

2.1 Digital Environment

The term *Environment* is used when defining the playground of the living things and its creatures. There are several sorts of science disciplines which involve in studying environments from different aspects. For instance, in sociology there exist roles attached to people and they name those as groups of people interacting with each other. On the other hand ecological science defines living organisms and their habitat and study the interaction between these organisms and the environment itself. Whether or not the environment is suitable for the given group of roles, we are studying the best of all conditions which give good accomodation quality for the citizens of these environments.

Analogous to biological environments, we can think of business units as organisms and also imagine that there are relationships and interactions between these business units. We have seen that there have been a lot of similar perspectives while we are metaphoring from biological environment to our business environment case.

We can say that businesses work together through digital collaboration. Therefore we can model business case and its living places (i.e digital world) as an ecosystem of entities as in the biological one.

In [1], it is pointed out that

"*Digital Ecosystem (DE)* is an emerging paradigm for economic and technological innovation. It consists of a self-organizing digital infrastructure, aimed at creating a digital environment for networked organizations (or agents) thus supporting the cooperation,

knowledge sharing and development of open and adaptive technologies and evolutionary domain knowledge rich environments. DE captures the essence of the ecological community in nature, where biological organisms form a dynamic and interrelated complex ecosystem, in analogy with economic organisms (such as business entities) or digital organisms (such as software applications). Digital species, just like biological ones, create and conserve resources that humans find valuable."

Formally one can define a Digital Environment(DE) as a directed graph:

$$DE = (V, E);$$

V : vertices(nodes, agents, entities)

E : ordered pairs of vertices(whose semantics describe communication links, one-way relation)

There have been some issues exposed when creating DE's in the real world domain. Next we explore the foundations which will help while creating DE like architectures.

2.2 Structured and Unstructured Environments

Suppose you are the father of a family. We know that families form the basic unit of environment or community. Then suppose you have a new child. So where does this child belong to? The answer is simple. If you have a child and you are a father of this child then it belongs to your family tree. Therefore we know the structure and estimate the location to which the new entity belongs.

Also we can see that in this environments there are some pattern like structures, for example tree structures and only the size of trees are different and other properties of trees are the same. One could recognize the whole picture of these structures in minutes.

On the other hand, for the unstructured environment the first thing to note is, in the environment there is no one way of adding an entity thus there are no fixed ways of addition.

In the family example, this corresponds to the case at which the child does not know where it belongs to, in this environment.

2.3 Peer-to-peer (P2P)

An example state-of-art digital ecosystem showing unstructured and decentralized characteristics is peer-to-peer (P2P) systems.

We can define P2P as follows; “A P2P system is a network of nodes called *peers* where all nodes collaborate in a manner which are decentralized.”

In Internet, millions of people use P2P applications. Especially P2P is used for file sharing purposes. For instance, one of the most popular file sharing application used in Internet is *Gnutella* [15]. Gnutella propose an environment of its own, in other words its environment. We call the environment in this case *Topology*. Within this context a *Topology* defines the logical connections between peers; on the contrary, properties of these connections clearly categorize different P2P topologies.

The first categorization is *the degree of centralization*. These are *Pure* and *Hybrid* approaches:
Pure P2P [20]:

- Peers act as equals, merging the roles of clients and server
- There is no central server managing the network
- There is no central router

Hybrid P2P [20]:

- Has a central server that keeps information on peers and responds to requests for that information.
- Peers are responsible for hosting available resources(as the central server does not have them), for letting the central server know what resources they want to share, and for making its shareable resources available to peers that request it.
- Route terminals are used addresses, which are referenced by a set of indices to obtain an absolute address.

The second classification is the structure of links created between nodes. This one also affects performance from the perspective of *find the others*. We can construct P2P systems

based on *Unstructured* and *Structured* overlays. In structured P2P networks, a user find the way of its own with the help of hash table like structures thus it is very efficient in structured networks but there are bottlenecks. Firstly Internet is very dynamic, secondly managing such a network is quite pricely. On the other hand unstructured network is not efficient but very comformant to the dynamic structure of Internet and also managing such a network does not expose extra effort. But in this case search efficiency is poor.

We use the term *overlay network* as a representation of P2P network as a logical network, not a physical structure but instead built on to the foundation of physical network nodes thus location of others is transparent to us while we are using P2P architectures. For instance a user of this network could be in somewhere in the world regardless of the location he/she is in. From the user perspective, this user seems to be a neighbor. But we do not know either exactly the location of information or the user.
Structured P2P [20]:

- A new node is added into the overlay P2P network as a consequence of applying overlay rule predefined.

Unstructured P2P [20]:

- A new node is connected to a node which it knows before or dynamically gathered the location by another node.

2.4 Matchmaking on Digital Environments

Matchmaking is a process of forming at least two parties which have goals and which want to establish these goals with the help of each other. For instance, we can think of these parties as two different people. One of them is which we name 'A', is very rich and also want to invest his money on business ideas but dramatically he has no idea. Second one, 'B', is very poor but has great business ideas which can lead him to earn a lot of money so he needs a lot of money in order to establish these ideas. If this two parties come together and collaborate on accomplishing same goals together, then we call this process of finding partnership as *matchmaking*. Note that, matchmaking may not always result successfully.

Matchmaking is also modeled in real life scenarios. Software developers use web services with full distribution of services without centralization. For example if you want to

use services of some other you will staticaly define the service in your program. This is not realistic because if the service definition changes you have to change your program. To overcome the problem, dynamic service discovery and usage is encouraged through the software community. In this case matchmaking solution is the best. If we do not staticaly bind services and use dynamic matchmaking approach it is easy to adapt sofware systems without sacrificing runtime quality of running software. A great deal of research has been made on this topic [11].

The basic discussion around matchmaking is the nature of the environment problem. There are two environment perspectives for the matchmaking problem.

The first one is the *centralized* approach. In this approach we put each party, which want to involve in the matchmaking process, in a pool of matchmaking descriptions and some other party applies to this matchmaking service. After that, a centralized controller, which we call matchmaker, will matchmake and decide whether or not the client party meets the specs of at least one party in the pool available. The matchmaking performance of this approach is the best. Because all the parties are compared to each other one by one. We do not need to locate parties, they are already in the pool. Pool is the central and unique location of encounters. But on the other side it has several bottlenecks. The most important one is the *single point of failure* problem at matchmaker. If the matchmaker is out of service, none of the above will happen. Also the real life is not resilient to this kind of problems. So there must be a better solution.

The second one is the *distributed* approach. In this approach, pool of matchmaking information is distributed and also matchmaking logic is semi centralized or fully decentralized. One of the basic advantages of such an approach is the partial provision of information hiding. We investigate and finally decide on whether there is an architecture which gives good matchmaking performance.

CHAPTER 3

ENVIRONMENT MODELS

3.1 Model Structures

In this section, we mainly introduce the environment models which are *Unstructured* and *Decentralized*. These models are used in our environment study and the foundations of our simulations.

A graph-based representation is assumed to be the framework for model comparison. The computational nodes (or peers) are taken as the nodes of the graph and peer neighborhood structure is described by the connectivity of the graph edges. Each node J of the graph is assigned a *match-seeking* process $C_J = (J, P_J)$, randomly. The reason behind working on *match-seeking* processes is to prevent unnecessary match investigations for already *self-sufficient* processes. Note that our overlay-level representation does not consider weighted edges that would otherwise reflect physical interpretations like bandwidth or physical node distance.

There have been several kinds of environments which are fitting into DE world. But in our study we work on unstructured environments striving for complete decentralization of decision-making and computation: HyperGrid [14], Gnutella [15] and environments showing generic power-law [16], random [17] [18] and small world [19] characteristics.

HyperGrid (HG): It is proposed by Saffre and GhaneaHercock [14] to model P2P networks. The main characteristic of the model is the generation of a graph topology with low graph diameter value and limited node degree. The HG graph grows as a k-ary tree. The connection policy is to connect the leaf nodes having k-1 not connected edges to those nodes at the same level and having degree of less than k. It tries to achieve complete

decentralization of network growth. (Fig. 3.7)

Gnutella (GNU): It is a model for unstructured P2P networks. The main disadvantage of the pure (Gnutella 0.4) model what we call **GNU-NUP** (Fig. 3.2), is significant peer signalling overhead and comparable high network overload. In hybrid Gnutella 0.6 what we call **GNU** (Fig. 3.6), the overhead is reduced by establishing a second routing hierarchy (ultrapeer layer) while keeping the network's complete self-organization ability [20].

Waxman Power-Law (WPL): Two Autonomous System (AS) Waxman model environments whose nodes are placed according to random (**ASW-RND**) (Fig. 3.4) and heavy-tailed (**ASW-HT**) (Fig. 3.3) distributions are examined. The topologies are generated in random manner using Waxman's probability model for interconnecting the nodes of the topology given by:

$P(u, v) = \alpha e^{\frac{-d}{\beta L}}$ where $0 < \alpha$, $\beta \leq 1$;

d is the Euclidean distance from node u to node v, and L is the maximum distance between any two nodes.

Random (RND): *Erdös-Renyi* random graph model is utilized as a baseline model for comparison. In the model, the connection probability *p* is the only parameter used for topology building. The graph is constructed by adding an edge between any two nodes with probability *p*.

Watts-Strogatz small world (WS-SW): In small world network, there is a short path between any two nodes. The main characteristics of small world topologies are known to be low diameter and high clustering coefficient when we compare with random graph of equivalent size. (Fig. 3.5)

2D-Grid: 2D-Grid is a general purpose, *Synthetic*, graph based on NxM where N and M are dimensions of this grid structure. We compare it as the baseline of synthetic graphs. (Fig. 3.1)

3.2 Process Encountering Mechanism in Environments

In the last section we have explored several environments which have different characteristics. We are sure that matchmaking process is going on those environments. Furthermore we begin to explain process encountering mechanism next.

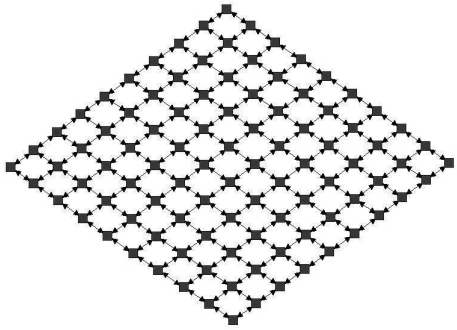

Figure 3.1: 2D-Grid

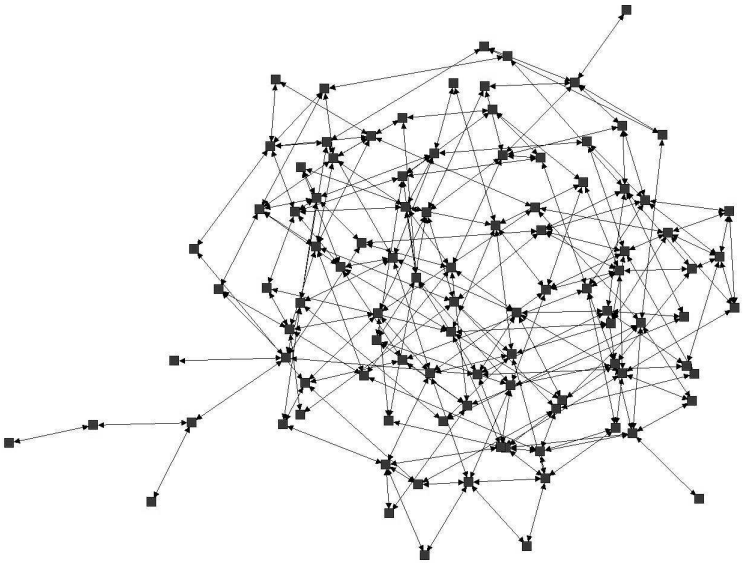

Figure 3.2: GNU-NUP

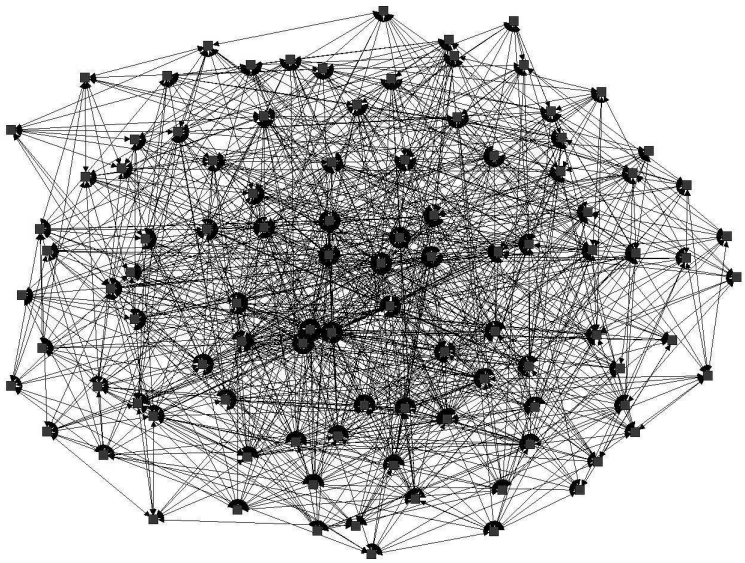

Figure 3.3: ASW-HT

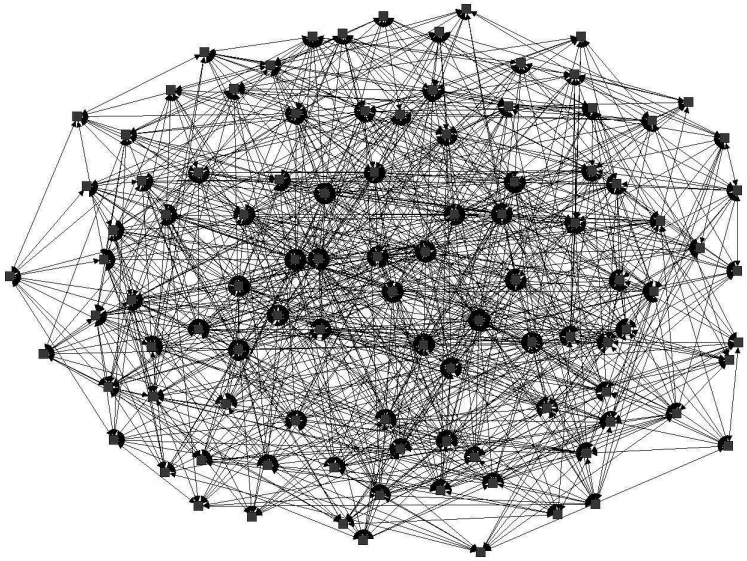

Figure 3.4: ASW-RND

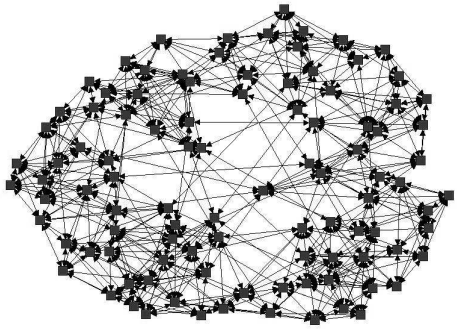

Figure 3.5: WS-SW

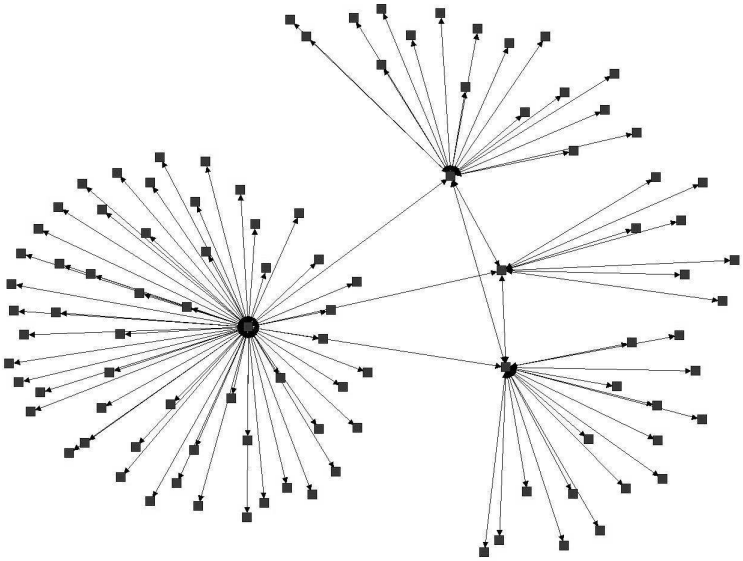

Figure 3.6: GNU

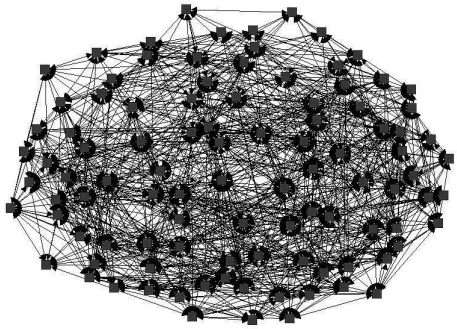

Figure 3.7: HG

In graph theory, Breadth-First-Search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

In our study, we use Breadth-First-Search (BFS). In other words, process exploration takes place by flooding queries across the peers. The used BFS technique is *uninformed* thus it utilizes only the local connectivity knowledge of the graph. During its operation, a *match-seeking* process C_j assigned to a node J is sent to the node's neighbor say node K, and checked whether *matches-with* C_K or not. If match occurs, its cost is calculated as the distance (i.e. the number edges traversed) between two peers. Otherwise, the operation continues with other descendant neighbors until the established Time-To-Live (TTL) value describing the scope (or depth) of the search has been reached. By the end, if match does not occur the cost is set to the initial TTL value.

3.3 Environment Model Generation

The work demonstrates a snapshot approach while evaluating process matchmaking performances of environments. Within this approach, generated environments have different node sizes in order to reflect the population of different sizes. This helps experiments to reflect realistic results for different sized environments.

The environment generator component generates different network topologies on which matchmaking potentials are investigated. There are two basic input parameters of the generator. The name of the environment and the number of nodes to be generated. The other parameters are environment specific and some of them are not related with topology generation. The topology related parameter values are chosen to build graphs of approximately equivalent (edge count) / (node count) ratio - which is assumed to be 10, across all environment models. An exception is Gnutella networks **GNU-NUP** and **GNU** for which attaining the ratio require too long simulation time as one may expect. The obtained ratios for Gnutella environments were around 1. The other exception is the structured **2D-Grid** environment for which the ratio is nearly 4. The sample topologies for **HG**, **RND**, **WS-SW** environments were generated by using Information Visualization Cyberinfrastructure - *IVC* tool [21]; for environments **ASW-RND** and **ASW-HT** we have used the *BRITE* tool [16]; and for **GNU-NUP** and **GNU**, *GnucNS* tool has been used [22]. The **2D-Grid** environment

topologies have been generated in-house.

Especially these mentioned tools have a common property. They model the Internet with different algorithms and their approximation of network representation is similar to the Internet. For researchers of network topic they are very important in order to make experiments. Because construction of such a big Internet like network structures is infeasible. At last researchers make experiment with generated topologies within simulations and it is easy to experiment different setups without constructing physical architectures. In our work we follow the same principle, we have used different tools and have generated lots of topologies. On these topologies we have ran simulations.

3.4 Properties of Generated Environment Models

The information about environment specific topology related parameters for an example network of 100-node are given:

HG: Maximum degree of each node is 26.

GNU-NUP and GNU: For both with and without ultrapeer simulations, the number of starting nodes and the maximum number of nodes that network can handle are both taken as 100. The number of connections that node tries to keep and the maximum number of connections that node will handle are 4 and 6, respectively. For the simulated ultrapeer system the minimum and maximum leaf capacities a node is created with, are 10 and 60.

ASW-RND and ASW-HT: For both environments, network growth type is incremental; the number of neighboring node that each new node connects to is 10 and Waxman parameter *alpha* is 0.15 and *beta* is 0.20. The only difference is due to node placement which is assumed to be random and heavy-tailed, respectively.

RND: Wiring probability is 0.2.

WS-SW: Degree of nodes is 10 and re-wiring probability is 0.0

Making judgment about matchmaking performance of a given environment with a fixed number of nodes, while using only one topology for each, would not be sufficient. Therefore, we try to handle this situation by generating 10 different topologies per environment per fixed number of nodes. Consequently, the overall number of topologies generated was 8x3x10=240. The topological properties of simulated environment models for node counts

of 100, 500 and 1000 are given in Table 3.1. Note that the values on the table reflect a typical topology for each environment among considered 10 different topologies used during matchmaking simulations. The values are measured by using *Ucinet* Social Network Analysis package [23]. Here, the average distance for a graph is the length of the shortest path between two nodes averaged over all node-pairs in the graph.

The diameter of a graph is the length of the longest direct path in the graph between any two nodes. The clustering coefficient of a graph is the proportion (averaged over all nodes) of nodes adjacent to a particular node that are also adjacent to each other [19]. Defining characteristics of small-world networks are low diameter and high clustering coefficient. The values for **WS-SW** in Table 1 reflects such characteristics, however, especially **GNU**, **RND** and **2D-Grid** do not grow as small-world topologies.

Table 3.1: Properties of several topologies

Env.	Nodes	Edges	Avg.Dist.	Avg.Deg.	Dia.	Clust.
GNU-NUP	100	450	3.42	4.5-(1,6)	8	0.131
	500	2526	4.12	5.052-(1,6)	8	0
	1000	5146	4.58	5.146-(1,6)	8	0
2D-Grid	100	360	6.67	3.6-(2,4)	18	0
	500	1880	16.11	3.76-(2,4)	39	0
	1000	3780	28.02	3.78-(2,4)	64	0
HG	100	954	2.53	9.54-(2,11)	4	0.232
	500	4854	3.70	9.708-(2,11)	6	0.106
	1000	9752	3.76	9.752-(2,11)	6	0.051
RND	100	994	2.23	9.94-(3,17)	4	0.094
	500	4994	2.94	9.998-(3,19)	5	0.02
	1000	10050	3.25	10.05-(1,19)	6	0.01
WS-SW	100	1000	2.70	10-(8,13)	5	0.505
	500	5000	3.98	10-(7,12)	7	0.502
	1000	10000	4.47	10-(7,14)	7	0.495
ASW-RND	100	1000	1.84	10-(10,42)	3	0.289
	500	5000	2.426	10-(10,62)	4	0.071
	1000	10000	2.65	10-(10,76)	4	0.038
ASW-HT	100	1000	1.84	10-(10,41)	3	0.252
	500	5000	2.45	10-(10,58)	4	0.075
	1000	10000	2.65	10-(10,71)	4	0.034
GNU	100	210	2.53	2.1-(1,61)	3	0.025
	500	1034	3.45	2.068-(1,64)	5	0.001
	1000	2086	4.25	2.068-(1,65)	6	0

CHAPTER 4

PROCESS MATCHMAKING

4.1 Process Matchmaking in Environments

Suppose we have an environment which is in this case *Unstructured* and *Decentralized*. In this environment suppose entities have special process descriptions. As in the web services approach we can think about those process descriptions as web services descriptions and at last we want to find collaboration possibilites between those web services/process description owners. So a matchmaking approach is needed in order to do this.

In this chapter we begin to explore the matchmaking theory and the foundations of this theory and also define process representation used in our environment study.

4.2 Process Representation and Match Operation

Definition 1: ([25]) Let Z be a finite set of states. *Process* is a tuple (S, P_S) such that S is a one-input transition system $S = (X, V, \delta, I)$ where $X \subseteq Z$ is a finite set of states and V is a finite set representing peer's capabilities. $\delta : X \times V \rightarrow X$ is the state transition function of the process. $I \in X$ is the initial (or starting) state of the process. $P_S \subseteq X$ is the end (or goal) state set .

Due to the introduced input, one-input transition system is an open system. In its graph-like representation, see Fig. 4.1, circle nodes define process states $(x_i \in X)$. The tick circle shows the initial state and dashed circles show the end states. Goal of the peer executing its process is to reach one of its goal states from its starting state.

In Fig. 4.1, Fig. 4.2 and Fig. 4.3; $Z = \{x_1, x_2, x_3, x_4\}, S = (X, V, \delta, I)$

Z : set of all states; S : one-input transition system; X : states; V : transitions; I : goal states; δ : transition function; $S_X = \{x_1, x_2, x_3\}$, $S_V = V_{1S}\}$, $S_\delta = \{((X_1, V_{1S}) \rightarrow X_2)\}$, $S_I = \{X_1\}$ where S_X : set of states of system S; S_V : set of transitions of system S; S_δ : set of transition pairs of system S; S_I : set of starting states of system S; and $P_S = \{X_3\}; P_S$: set of goal states of system S . Similarly in Fig. 4.2, $Q = (X, V, \delta, I)$; where $Q_X = Z, Q_V = \{V_{1Q}, V_{2Q}, V_{3Q}\}, Q_\delta = \{((X_1, V_{1Q}) \rightarrow X_4), ((X_2, V_{2Q}) \rightarrow X_3), ((X_3, V_{3Q}) \rightarrow \mathrm{X}_4)\}, Q_I = \{X_3\}$ and $P_Q = \{X_2\}$.

For both system S and Q, their individual induced behaviours do not satisfy the property of "reaching to" their goal-states P_S and P_Q, respectively. In other words, there exists no successfully executing individual either for process (S, P_S) or process (Q, P_Q). Strictly speaking, it is a must that two processes are both *capability disjoint*. Therefore we eliminate the probability of non-determinisim when two processes are merged together. Here is the definition:

Definition 2: ([25]) Two one-input transition systems S and Q are called *capability disjoint* iff $S_V \bigcap Q_V = \emptyset$.

If both of S and Q have the same goals then they have equivalent set of goals.

Definition 3: ([25]) Processes (S, P_S) and (Q, P_Q) are called *goal-equivalent* iff $P_S = P_Q$.

Systems S and Q in Fig. 4.1 and Fig. 4.2 are *capability-disjoint*. Processes (S, P_S) and (Q, P_Q) are not *goal-equivalent*. A one-input transition system is not *capability-disjoint* with itself however any process is *goal-equivalent* to itself.

If two processes are well formed with the definitions above then two processes are ready for collaboration. Collaboration means capability formation. It is now possible for processes to reach their goals. It is not possible to do that when they are alone. Therefore merging two processs descriptions in the name of capability is a powerful operation.

Definition 4: ([25]) Given two capability-disjoint one-input transition systems S and Q, *merge*(S, Q) operation returns a one-input transition system T such that $T_X = S_X \bigcup Q_X$

, $T_V = S_V \bigcup Q_V, T_\delta = S_\delta \bigcup Q_\delta$ and $T_I = S_I$.

The operation is not commutative but associative. In fact, the merge operation implies a graph union whose nodes (i.e. states) take values from the same domain. Fig. 4.3 shows system T obtained by the merge of systems S and Q.

Definition 5: (Behavior Induced by Input) - ([13]). Given a system $S = (X, V, \delta, I)$ and an input sequence $\Psi \in V^*$, the behavior of S starting from I in the presence of Ψ is a sequence $\xi(\Psi) = \xi[0], \xi[1], \ldots \in X^*$ such that $\xi[0] = I$ and for every i, $\xi[i+1] = \delta(\xi[i], \Psi[i])$.

In the automaton of Fig. 4.3 , an input starting with V_{1S}, V_{2Q} generates a behavior starting with X_1, X_2, X_3 a fact that can be denoted as:

$$X_1 \rightarrow^{V_{1S}} X_2 \rightarrow^{V_{2Q}} X_3$$

Definition 6: (Reachability for Open Systems) - ([13]). Given a system $S = (X, V, \delta, I)$ and a set $P \in X$, is there some input sequence $\Psi \in V^*$ such that the behavior $\xi(\Psi)$ reaches P?

Assume that $\delta(x)$ is the set of all *immediate successors* of x. i.e $\delta(x) = \{x^{'} : \exists v \delta(x, v) = x^{'}\}$ and we can extend this notation to sets of states F by letting $\delta(F) = \{\delta(x) : x \in F\}$. The following algorithm given in [13], computes all reachable states of a one-input transition system.

Algorithm *Reachables*
Input: $S = (X, V, \delta, I)$, a one-input transition system
Output: F, set of all states reachable from I

$F^0 := I$
repeat
  $F^{k+1} := F^k \bigcup \delta(F^k)$
until $F^{k+1} = F^k$
$F^* := F^k$
return F^*

The algorithm is a simple polynomial-time graph search algorithm searching breadth-first manner in which every F^k consists of the states reachable after at most k transitions.

Note that, for the systems of Fig. 4.3, *Reachables*(S) returns X_1, X_2 and *Reachables*(Q) returns X_3, X_4. Similarly, *Reachables*(T) returns X_1, X_2, X_3, X_4. Having defined merge operation and reachables algorithm we can define the match algorithm executed by every peer.

If S and Q come together and if Q lets S to reach at least one goal, then processes S and Q match with each other. Let's define it:

Definition 7: ([25]) Let us given two processes (S, P_S) and (Q, P_Q) of which systems S and Q are *capability-disjoint*. The process (S, P_S) is said to be *matches-with* process (Q, P_Q) iff $(P_S \bigcap$ *Reachables(merge(S, Q)))* $\neq \emptyset$ holds.

For the example given in Fig. 4.3, we can say that (S, P_S) *matches-with* (Q, P_Q) but not vice versa. Because $(P_Q \bigcap$ *Reachables(merge(Q, S)))* $= \emptyset$

Definition 8: ([25]) A process (S, P_S) is said to be *self-sufficient* iff $P_S \bigcap$ *reachables(S)* $\neq \emptyset$, otherwise it is called *match-seeking*.

A *Match-seeking* process always searchs some other process for matchmaking and finally reaching to its goals. Because it can not do this by itself, because it has not this ability, it is not self-sufficient. It has money but do not know how to make profit.

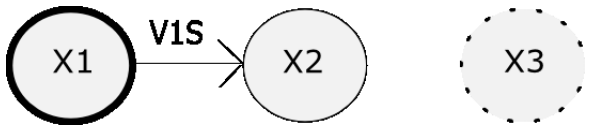

Figure 4.1: Process (S, P_S)

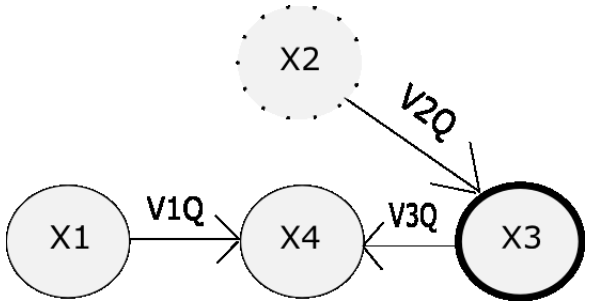

Figure 4.2: Process (Q, P_Q)

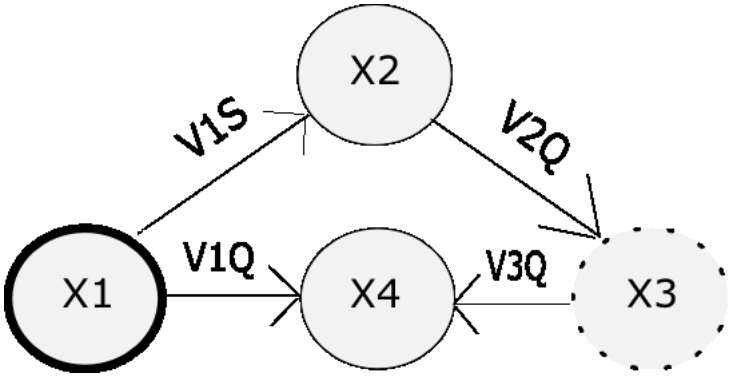

Figure 4.3: Process ($T = \textit{Merge}(S, Q), P_S$)

4.3 Process Generation

The process generator component takes process specifications as an input and produces sample process descriptions according to Definition 1 and Definition 8. The generated processes are guaranteed to be *match-seeking*. Since we assign one process per node, the

size of process population is set to the number of nodes of the current topology under consideration. The following three parameters describing population and individual process characteristics are established. Furthermore r, s, t parameters are choosen for modeling a wide range of problems but r, s, t values are not choosen with a rule. Their combinations constitute 18 problems with different difficulty degrees. We are sure they are enough capable of representing different kinds of problems, from easy to hard.

1. The percentage *r* of processes being identical. For example, if the number of process under consideration is 500 and 10% of them is required to be identical then 450 processes are guaranteed to be different from each other while the rest is identical. The purpose of the parameter is to control the homogeneity of the process society;

2. The number *s* of system states belonging to a process. i.e $\mid S_X \mid$. This parameter defines the process size. The other size control parameter is the number of system capabilities (i.e $\mid S_V \mid$). For all generations, we assumed that

$$\mid S_V \mid = \mid S_X \mid - \mid P_S \mid$$

3. The percentage *t* of system states being goal states. This parameter is used to partially and indirectly control the difficulty of finding a match for the generated process. Higher ratio values are expected to make easier to find a match.

For a given topology, making process assignments per node only once would not be fair to make a judgment about the topology's matchmaking quality. Therefore, for each topology under consideration we have generated 18 different process batches what we call *problems*. The considered parameter values describing the *problems* were r = (10, 20, 50); s = (10, 20, 50) and t = (10, 20).

CHAPTER 5

SIMULATOR AND UTILITIES

5.1 Utility Programs

5.1.1 Producing Problem and Topology Descriptions

We use several tools that generates topology maps (i.e *BRITE, IVC*). But these tools generate different topology formats. Each must be normalized into some form that is usable in our simulator. We used a general format which is suitable for our experiments.

We write utility programs in CPython and use those within the convertion of each topology description from the native file format to the ours. So it is easy to write factory methods in the case where in the near future in the case of any topology generator file format is welcome.

We have proposed the same solution for representing the process descriptions. But in this case we write our process generator program based on some process generation rules. For example we want processes to be *match-seeking*, so we generate descriptions and test whether they are match-seeking or not.

5.1.2 Producing Graph Results

We labeled each topology and presents each as a line pattern for differentiating them on the graphs. There are two graphs we produced. These are *Match* and *Cost*. Before producing these, we have to gather information about the results. For this purpose it is easy to use

SQL. Therefore we put the results as SQL, query back with SQL and then we are able to generate those graphs.

5.2 Simulator

There are two basic functions of matchmaking simulator:

1. To assign generated processes to the nodes of the generated topologies,
2. To execute matchmaking simulations.

For each node of graphs one and only one process is assigned, randomly. In the simulations, the number of total matches and their total costs are taken as two performance criteria. It should be clear that the performance values are not only affected by the connectivity of generated graph topologies but also by the number of hops that match-seeking process lives for (i.e. TTL). During simulations, we considered TTL=1, 3 and 5 values describing the depth of the search. Note that, our overlay level cost calculations do not include communication costs due to message traffic that occurs during simulations.

The tool identifies a typical run encoded as E-N-L-P-T where E is the code of the environment (from 1 to 8), N is the number of graph nodes (100 Nodes, 500 Nodes, 1000 Nodes). L is the TTL value (1, 3, 5), P is the problem id (from 1 to 18) and T is the topology number (from 1 to 10). As a result, the number of executed matchmaking runs is 8x3x3x18x10=12960.

The simulator and process generator were developed using CPython 2.4.3 interpreter. The hardware platform was based on 2.0Ghz CPU machines. As we said in the previous sections we had 12960 runs to be executed. But this number is very dramatic if we run all the sim only on one machine. Thus we think executing runs in parallel on several machines is feasable and gives more promising running times. For instance if we simulate **SW-WS**, in other words 1-100-1-15-1 labeled run last in 33 minutes. Also we have all time values for all run results in result files.

We have to divide the runtime configuration into units which can be independently executed on different machines. Each machine would be responsible for executing runs and write down the simulation results and post this results back to the disk.

We have faced with some issues during the simulations. These are generally related with environmental problems. In fact electricity cutoff and cooling problems are the most important ones.

For the hardware part, we can say that simulator program is running like a benchmark tool. Therefore we have a chance of which hardware is running best or have good hardware parts (i.e. high quality memory unit, defragmented disk).

Along the run, we evolve the simulator program very much in detail. For example automatic execution of runs and automatic posting of run results are the two features newly implemented. This gives us better manageability on the simulation testbed.

CHAPTER 6

EXPERIMENTAL RESULTS

The results given in Fig A.1 to A.18 are calculated for the average of 10 topologies of a given environment, generated problem, TTL value and node count values. Fig A.1 to A.9 are match count results for node sizes 100, 500, 1000, respectively. Fig A.10 to A.18 reflect the match costs for the same node sizes. *Proper-match* case is described as when every process in the population matches with all other processes. Therefore, for an *n* node count, the number of proper-matches is : $nx(n-1)$. *Proper-match* defines a theoretical upper-bound. On the other hand, for a generated match-seeking process population this limit may not be reached even for fully connected topologies. In Fig A.1 to A.9, the lines with legend *full* show the calculated values assuming the n-node graph is fully connected. These values can also be interpreted as fully connected. These values can also be interpreted as problem specific match counts when the matchmaking is executed in centralized way. Therefore, match counts in centralized graph in other words fully connected graph can not be exceeded.

From the match count figures, we can conclude that when the node count is 100 match counts are sensitive to increasing TTL values. However, poor match count results for 500 and 1000 nodes and their insensitivity to increasing TTL values are also observed.

When we compare the match count results of unstructured and decentralized environments among themselves the **RND** environment outperforms the others for almost all different problems and TTL values. The reason behind poor performances of **2D-Grid** and **GNU** environments can be explained by their small average node degree values. The dramatic performance differences between two Gnutella versions **GNU-NUP** and **GNU** observed in all result scales are worth to pay attention. The reason behind poor **GNU** results

is the potential blocking effect of ultrapeers on investigation of further match alternatives. Also, high diameter and low average degree values of **GNU-NUP** and **GNU** made them the poorest environment choice for almost all runs. For higher TTL values, it is observed that P2P model **HG** gives higher match count results than the synthetic environments like **WS-SW**, **ASW-RND** and **ASW-HT**, for almost all problems. Further, it is observed that the node placement in **AS-Waxman** environments does not have a considerable affect on the match count and cost results.

We have noticed an important result for **HyperGrid** topology. If we look at match-count figures with TTL value 3 (Fig A.2, A.5, A.8), **HyperGrid** match-count results are lower than **ASW-RND** and **ASW-HT**. Again, if we look at match-count figures with TTL value 5 (Fig A.3, A.6, A.9), **HyperGrid** outperforms **ASW-RND** and **ASW-HT**. Finally, we can say that match-count performance of **HyperGrid** topology outperforms others with TTL value 5 excluding the winner, **RND** graph.

Apart from those, the resulting graphs are following similar patterns. For all topologies, we have applied several problems that have different difficulties. For all problems, the order of topologies based on match and cost hit did not change. Therefore different problem settings do not affect the order of topologies. This implies an apparent categorization of different environments.

From the problem perspective, Problems 2, 8 and 14 seem the most difficult problems for which match counts for almost all environment models give poor results. In general, we can say that the relative matchmaking performances on different environments do not show dramatic changes for different problems.

The cost results in Fig A.10 to A.18 are calculated in units of hops. If most of the matches are achieved for high TTL values, the cost will be also high. Relatively high cost values obtained for small world WS-SW model implies such situation. In calculations, the cost of a no-match instance is assumed to be the run's upper TTL value. When TTl=1, the cost results in Fig A.10, A.13, A.16 are constant and not problem dependent. In such cases, the cost of unit match and no-match cases are both equal to 1. The problem which has the biggest cost value for all environments is 14.

CHAPTER 7

CONCLUSIONS

In this thesis, process matchmaking performance of different digital environments has been evaluated through extensive number of simulations. The methodology, which is followed in the thesis, is coming from corresponding literature. Existing simulation models and their methodologies are also investigated. Results of corresponding works are cross-checked and we have commented interesting results. To be very precise in the results, there have been an extensive number of simulation runs.

Firstly, a number of digital environments are generated to use in the simulations. To do this, topology generators are used. At the end there have been different kinds of digital environments.

Secondly, process matchmaking model is presented to express how we matchmake process descriptions.

Thirdly, problem descriptions are generated to use in the simulations. Although there are lots of descriptions available, however they are carefully choosen for representing a wide range of problems.

At last, problem descriptions are binded to topology structures and simulation runs are executed. Match and cost performance of each digital environment for each simulation run is recorded. Then records are evaluated from result graphs in order to see how they performed in the simulations.

The work reflects snapshot approach to evaluate process matchmaking performances of digital environments. To reflect realistic results, there have been a great number of snapshots in the runs. However, this is not the only way of evaluating digital environments.

Some dynamicaly changing environments could be used. In those ones, digital environments are evolving and there will be scenarios which will be executed in parallel for each digital environment. But in this case long evolving time of digital environments could be an issue.

Simulation result presents an important conclusion. It has been seen from the results that random(RND) graph has the highest potential for process matchmaking. If this is true the overlay level engineering effort for better process matchmaking setups become obsolute. So as for as our snapshot-based simulations show that one can construct an unstructured and decentralized digital environment for process matchmaking purpose without thinking of overlay level organizations.

REFERENCES

[1] Call for paper announcement of in *Inaugural IEEE Intl. Digital Ecosystems and Technologies Conference - DEST 2007 : http://www.ieee-dest.curtin.edu.au, last accessed: 01-05-2007*.

[2] D.Ben-Ami and O.Shehory, "A comparative evaluation of agent location mechanisms in large scale mas," in *International Conference on Autonomous Agents and Multi-Agent Systems*, pp. 339–346, 2005.

[3] K. Sycara, J. Lu, M. Klusch, and S. Widoff, "K. Sycara, J. Lu, M. Klusch, S. Widoff, Matchmaking among Heterogeneous Agents on the Internet, in Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace, http://citeseer.ist.psu.edu/64891.html, last accessed: 11-04-2006, Stanford University, USA 22-24 March 1999."

[4] I. Constantinescu and B. Faltings, "Efficient matchmaking and directory services," in *WI '03: Proceedings of the IEEE/WIC International Conference on Web Intelligence*, (Washington, DC, USA), pp. 75–81, IEEE Computer Society, 2003.

[5] E. Ogston and S. Vassiliadis, "Matchmaking among minimal agents without a facilitator," in *Proceedings of the Fifth International Conference on Autonomous Agents* (J. P. Müller, E. Andre, S. Sen, and C. Frasson, eds.), (Montreal, Canada), pp. 608–615, ACM Press, 2001.

[6] R. Celebi, H. Ellezer, C. Baylam, I. Cereci, and H. Kilic, "Process matchmaking on a p2p environments," in *CEC '04: Proceedings of the IEEE/WIC/ACM International Workshop on P2P Computing and Autonomous Agents(P2PAA)*, (Hong Kong), pp. 463–466, 2006.

[7] G. H. L. Fletcher, H. A. Sheth, and K. Borner, "Unstructured peer-to-peer networks: Topological properties and search performance," in *Proceedings of the Third International Workshop on Agents and Peer-to-Peer Computing*, (New York, New York, United States), pp. 14–27, July 2004.

[8] I. Ariba, I. Corporation, and M. Corporation, "Universal description, discovery and integration specification, http://www.uddi.org, last-accessed: 09-10-2006" 2000.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language wsdl v1.1 specification, http://www.w3.org/TR/wsdl, last-accessed: 09-12-2006" 2001.

[10] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiant, S. Sharm, , and S. Williams, "Web services conversation language(wscl) v1.0, http://www.w3.org/TR/wscl10/, last-accessed: 09-15-2006" 2002.

[11] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold, "Matchmaking for business processes based on choreographies," in *EEE '04: Proceedings of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04)*, (Washington, DC, USA), pp. 14–32, IEEE Computer Society, 2004.

[12] A. Wombacher, B. Mahleko, and E. Neuhold, "Ipsi-pf: A business process matchmaking engine," in *CEC '04: Proceedings of the IEEE International Conference on E-Commerce Technology (CEC'04)*, (Washington, DC, USA), pp. 127–150, IEEE Computer Society, 2004.

[13] O. Maler, "Control from computer science," Annual Reviews in Control, 26, 2002, pp. 175–187. 2003.

[14] F. Saffre and R. G. Hercock, "Beyond anarchy: self-organized topology for peer to peer networks," *Complexity*, vol. 9, pp. 49–53, November-December 2003.

[15] "Gnutella specification : http://www.gnutella.com, last accessed: 12-10-2006"

[16] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: Universal topology generation from a user's perspective," Technical Report BUCS-TR2001-003, Boston University, 2001. Available at http://citeseer.ist.psu.edu/medina01brite.html, last-accessed: 08-13-2006, 2001.

[17] R. Albert and A. L. Barabasi, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, no. 1, pp. 47–97, 2002.

[18] M. Newman, “The structure and function of complex networks,” SIAM Review, 45(2):167-256, 2003.

[19] D. J. Watts and S. H. Strogatz, “Collective dynamics of ’small-world’ networks.,” *Nature*, vol. 393, pp. 440–442, June 1998.

[20] J. Eberspacher, R. Schollmeier, S. Zls, and G. Kunzmann, “Structured p2p networks in mobile and fixed environments,” in *Proceedings Modelling and Evaluation of Heterogeneous Networks*, Tutorial T4, http://www.comp.brad.ac.uk/het-net/HET-NETs04/CameraPapers/T4.pdf, last-accessed: 10-10-2006, 2004.

[21] “Information visualization cyberinfrastructure: A data-code-compute resource for research and education in information visualization. available online at: http://iv.slis.indiana.edu. last accessed: 11-20-2006,”

[22] “Gnucns network simulator: http://www.gnucleus.com, last accessed: 10-01-2006”

[23] S. P. Borgatti, M. G. Everatt, and L. C. Freeman, “Ucinet for windows: Software for social network analysis,” 6.39 ed. Harvard: Analytic Technologies. 2002.

[24] B. Cakir and H. Kilic, “An Investigation about Process Matchmaking Performances of Unstructured and Decentralized Digital Environments,” in *IEEE DEST ’07: Proceedings of the Inaugural IEEE Intl. Digital Ecosystems and Technologies Conference - DEST 2007 :*, (Cairns Australia), 2007.

[25] H. Kilic, “An Offline Solution to the Matchmaking for Largest Double-Team Set Problem,” in *Proceedings of International Symposium on Innovations in Intelligent Systems and Applications (INISTA 2005)*, pp. 38-41, June 15-18, 2005, Istanbul, Turkey

APPENDIX A

APPENDIX

A.1 Code Listing

A.1.1 Simulator Runner

```
from core import *
from os import *
from datetime import *

'''
    belirtilen algoritma,
    node sayisi,
    ttl degeri,
    algoritmanin node sayisinin topoloji sirasi,
    problem sekli sirasi
    parametrelerine gore

'''

tsdir = ".\\Problems\\"
grphdir = ".\\Topology\\"

def run(ttl, problem, nodenumber, algorithm, sample, tsdir, grphdir):
```

```
    grphdir = grphdir + algorithm + "\\" + str(nodenumber) + "\\"
    tsdir = tsdir + problem + "\\" + str(nodenumber) + "\\"

    rgdir = listdir(grphdir)

    rgsample = ""
    for rg in rgdir:
        if int(rg.split(" ")[1].split(".")[0]) == sample:
            rgsample = rg


    # problem'i graph'a ata.
    g = assign(grphdir+rgsample, tsdir)

    ttlmatchcost, ttlnonmatchcost = 0, 0
    ttlhit = 0

    for i in range(0, nodenumber):
        match, matchcost, nonmatchcost = graphsearch(g, i, ttl, \\
    g[i].ts)
        ttlmatchcost += matchcost
        ttlnonmatchcost += nonmatchcost
        ttlhit += len(match)

    return ttlmatchcost, ttlnonmatchcost, ttlhit

'''
    graphin fully connected cost ve hitini doner.
'''
def runfully(graph):
    totalmatchcost, totalnonmatchcost,totalhit = 0, 0, 0

    for i in range(0, len(graph)):
        hit, matchcost, nonmatchcost = graphsearch(graph,i,1, \\
```

```
graph[i].ts)
        totalmatchcost += matchcost
        totalnonmatchcost += nonmatchcost
        totalhit += len(hit)

    return totalmatchcost, totalnonmatchcost, totalhit


def upload(fileobject):

    from ftplib import FTP
    ftp = FTP('td185.testdrive.hp.com')
    try:
        ftp.login(user='leninam', passwd='Sum1234')
        ftp.set_pasv(True)
        ftp.storlines('STOR ' + fileobject.name, fileobject)
        ftp.close()
        return True
    except:
        return False
def main():


    result = ""
    f = file("problems.txt")
    problems = []
    for line in f:
        problems.append(line[:-1])
    f.close()


    # runlar bir runlist listesinde tutuyoruz.
    runlist = []
    for root, dirs, files in walk("Run\\"):
```

```
        f = file(root+ files[0])
        for line in f:
            runlist.append(line[:-1])

    # her bir run icin calistir
    for eachrun in runlist:

        runparts = eachrun.split("-")
        ainput = runparts[0]

        nodenumber = int(runparts[1])
        alglist=["ConvertedStrogatz", \
                 "ConvertedASWaxman", \
                 "ConvertedASWaxmanHT", \
                 "ConvertedHyperGrid", \
                 "ConvertedGnutella", \
                 "ConvertedGrid", \
                 "ConvertedRandom"]
        algorithm=alglist[int(ainput)-1]

        result = "start " + str(datetime.today()) + "\n"
        problem = problems[int(runparts[2])-1]

        fcg = fullyconnectedgraph(nodenumber, tsdir + problem \\
+ "\\" + str(nodenumber) + "\\")
        fullymatchcost, fullynonmatchcost, fullyhit = runfully(fcg)

        for ttl in range(1, 6, 2):
            for sample in range(1, 11):
                matchcost, nonmatchcost,  hit = run(ttl, problem, \\
nodenumber, algorithm, sample, tsdir, \\
grphdir)
                print hit
                result += "|N: " + str(nodenumber) + " A: " \\
```

```
str(algorithm) + " T: " + str(ttl) + " O: " + str(sample) + \\
   " P: " + str(problems.index(problem) + 1)  + "(" + problem \\
   + ")|MC: " + str(matchcost) + " NMC: " + str(nonmatchcost) \\
   + " MH: " + str(hit) + "|\n"

 result += "|Full Match Cost: " + str(fullymatchcost) \\
+ " Full NonMatch Cost: " + str(fullynonmatchcost) \\
   " Full Hit: " + str(fullyhit) + "|\n"
            print result

        result += "finish " + str(datetime.today())

        f = file(eachrun+algorithm+str(nodenumber)+"result.txt","w")
        f.write(result)
        f.flush()
        f.close()

        '''
        f = file(eachrun + algorithm + str(nodenumber) + "result.txt")
        ftpresult = False
        while ftpresult == False:
            ftpresult = upload(f)
        f.close()
        '''

        print "-------"

main()
```

A.1.2 Core

```
'''
```

```
    Node class
'''
class Node:
    def __init__(self, idnumber):

        # id
        self.id = idnumber

        # neighbourhoods
        self.nb = []

        # resources that node holds
        self.value = []

        # hit
        self.hit = False

        # ts
        self.ts = None

        # CAN properties
        self.xmid = 0.0
        self.ymid = 0.0
        self.zmid = 0.0
        self.visited = False

    def add(self, nb):
        self.nb.append(nb)


'''
    Graph icinde arama algoritmasi
        -  nodelist'de tanimlanan graph icin sourcenode
           baslangic node'u oldugu zaman
```

```
            ttl kadar adim icinde resource’a uygun node’lar
  varsa hitlist icine atilir.
’’’
def graphsearch(nodelist, sourcenode, ttl, resource):
    hitlist = []
    nextlist = nodelist[sourcenode].nb
    matchcost, nonmatchcost = 0, 0
    level = 1

    from reachability import merge, matchmake
    while ttl > 0:
        for n in nextlist:
            ’’’
            if resource in n.value:
                n.hit = True
                hitlist.append(n.id)
            ’’’
            m = merge(resource, n.ts)
            match = matchmake(m, 5)
            if match and n.hit == False:
                n.hit = True
                hitlist.append(n.id)
                matchcost += level
            else:
                nonmatchcost += level


        temp = []
        for n in nextlist:
            if n.hit == False:
                temp.extend(n.nb)
        nextlist = temp
        ttl -= 1
        level += 1
```

```
    for n in nodelist:
        n.hit = False

    return hitlist, matchcost, nonmatchcost


'''
    graph ile belirtilen dosyadan graph'i nodelist icinde
    dondurur.
'''
def loadgraph(graph):

    graphfile = file(graph)
    nodelist = []
    for line in graphfile:
        nodes = line[:-1].split(' ')
        node = Node(int(nodes[0]))
        nodelist.append(node)
    graphfile.close()

    graphfile = file(graph)
    for line in graphfile:
        nlist = line[:-1].split(' ')
        rootnode = nodelist[int(nlist[0])]
        for neighbour in nlist[1:]:
            rootnode.add(nodelist[int(neighbour)])

    return nodelist

'''
    fullyconnected graph uretir, tsdir'la belirtilen
 problemi enjekte eder.
'''
```

```
def fullyconnectedgraph(nodevolume, tsdir):

    # nodevolume buyuklugunde bir node listesi yarat
    nodelist = []
    for i in range(0, nodevolume):
        node = Node(i)
        nodelist.append(node)

    # her bir node'a geri kalan nodelari bagla.
    for i in range(0, nodevolume):
        nodelist[i].nb.extend(nodelist[0:i])
        nodelist[i].nb.extend(nodelist[i+1:nodevolume])

    # automatalari indeks sayisina gore node listesine ata.
    from reachability import loadtsfromdir
    ts = loadtsfromdir(tsdir)
    for i in range(0, len(nodelist)):
        nodelist[i].ts = ts[i]

    return nodelist

'''
    CAN network graph'i dosyadan okur, nodelist olarak dondurur.
'''
def loadcangraph(graph):

    graphfile = file(graph)
    nodelist = []
    for line in graphfile:
        nodes = line[:-1].split(' ')
        node = Node(int(nodes[0]))
        nodelist.append(node)
    graphfile.close()
```

```
    graphfile = file(graph)
    for line in graphfile:
        nlist = line[:-1].split(' ')
        rootnode = nodelist[int(nlist[0])]
        if '.' not in nlist[1]:
            for neighbour in nlist[1:]:
                rootnode.add(nodelist[int(neighbour)])
        else:
            rootnode.xmid = float(nlist[1])
            rootnode.ymid = float(nlist[2])
            rootnode.zmid = float(nlist[3])

    return nodelist[0:(len(nodelist)/2)]


'''
    graph node'larna ts'leri atar.
    bir tane graph dosyasi alir, bir tane
    ts dizini alir. ts dizini altindaki dosyalari
    okur ve graph icindeki her node icin okudugu ts'lerin
    her birini atar.

    geriye graph'i doner (g)
'''
def assign(graphfile, tsdir):
    g = loadgraph(graphfile)
    from reachability import loadtsfromdir
    ts = loadtsfromdir(tsdir)

    for i in range(0, len(g)):
        g[i].ts = ts[i]

    return g
```

A.1.3 Reachability

```
from core import Node
from os import *

class TS:
    def __init__(self, idstr):
        self.id = idstr
        self.nodes = []
        self.actions = []
        self.startnode = None
        self.goalnodes = []
        self.transitions = dict()

'''
    dosyayi dondur
'''
def readfile(filename):
    f = file(filename)
    return f


'''
    Transition system'i dondur.
'''
def transitionsystem(filename, tsname):
    sfile = readfile(filename)
    ts = TS(tsname)
    for line in sfile:
        elts = line[:-1].split(' ')
        node = elts[0]
        if node not in ts.nodes:
            ts.nodes.append(node)
        if elts[1] == 'Start':
```

```
            ts.startnode = elts[0]
        elif elts[1] == 'Final':
             ts.goalnodes.append(elts[0])
        elif elts[1] != '0':
            if elts[1] not in ts.actions:
                ts.actions.append(elts[1])
            if elts[1] not in ts.transitions:
                ts.transitions[elts[1]] = []
            l = ts.transitions[elts[1]]
            l.append([elts[0], elts[2]])

    sfile.close()
    return ts

'''
    Verilen iki gecis sistemin birlestirimini saglar,
    yeni bir gecis sistemi olusturur ve dondurur.
'''
def merge(tsfirst, tssecond):
    '''
        ts1 = transitionsystem(tsfirst, 'first')
        ts2 = transitionsystem(tssecond, 'second')
    '''

    ts1 = tsfirst
    ts2 = tssecond

    # node unification
    nodes = []
    nodes.extend(ts1.nodes)
    nodes.extend(ts2.nodes)
    temp = set(nodes)
    nodes = list(temp)
```

```
# action unification
actions = []
actions.extend(ts1.actions)
actions.extend(ts2.actions)

# start nodes unification
startnodes = ts1.startnode
if startnodes is None:
    startnodes = 0

# goal nodes unification
goalnodes = []
goalnodes.extend(ts1.goalnodes)
#goalnodes.extend(ts2.goalnodes)
temp = set(goalnodes)
goalnodes = list(temp)

# transition unification
transitions = dict()
for key, value in ts1.transitions.iteritems():
    transitions[key] = value
for key, value in ts2.transitions.iteritems():
    transitions[key] = value

# new ts
mergedts = TS('merged')
mergedts.transitions = transitions
mergedts.goalnodes = goalnodes
mergedts.startnode = startnodes
mergedts.actions = actions
mergedts.nodes = nodes

return mergedts
```

```
'''
    State graph icinde goal nodelari belirtilen ttl'e gore
    arar. Ttl degeri bitmeden goal node'larin en az biri bulunursa
    arama basariyla sonlanir. ttl degeri yuksek verilebilir.
'''
def matchmake(ts, ttl):

    startnode = ts.startnode
    finalnodes = ts.goalnodes
    nodelist = ts.nodes

    # dictionary
    d = nextlist(ts.transitions)

    mm = False
    # startnode'a bagli node'lar yoksa match yok.
    if startnode not in d:
        return mm

    adaylar = d[startnode]

    while ttl > 0:
        for fn in finalnodes:
            if fn in adaylar:
                mm = True
                return mm

        temp = []
        for aday in adaylar:
            if aday in d:
                temp.extend(d[aday])

        adaylar = temp
        ttl -= 1
```

```
    return mm


def nextlist(transitions):
    d = dict()
    for key, value in transitions.iteritems():
        for k in value:
            if k[0] not in d:
                d[k[0]] = []

            if k[1] not in d[k[0]]:
                d[k[0]].append(k[1])

    return d

'''
    verilen dizindeki tum ts'leri liste icine yukler.
    liste icinde TS siniflari bulunur veri geri doner.
'''
def loadtsfromdir(directory):
    files = listdir(directory)
    ts = []
    for f in files:
       t=transitionsystem(directory+f, f.split('.')[0].split(' ')[1])
       ts.append(t)

    ts.sort(cmp=lambda x,y: cmp(int(x.id), int(y.id)))

    return ts
```

A.1.4 Generator

```
from random import *
from reachability import *
import copy

'''
    represents a state in the automata
'''
class state:
    def __init__(self, stateid):
        self.start = False
        self.final = False
        self.id = stateid

'''
    represents an automata
'''
class automata:
    def __init__(self, automataid):
        self.id = automataid
        self.states = []
        self.transitions = []

    def serialize(self, directory):
        f = file("Problems" + "\\" + directory + "\\" + "Automata "\\
 + str(self.id) + ".txt", "w")
        for s in self.states:
            if s.start:
                f.write(str(s.id) + " " + "Start")
                f.write("\n")
            elif s.final:
                f.write(str(s.id) + " " + "Final")
                f.write("\n")
```

```
        for t in self.transitions:
            f.write(t)
            f.write("\n")

        f.flush()
        f.close()

def automataconverter(automata):

    ts = TS("a")
    for node in automata.states:

        if node.id not in ts.nodes:
            ts.nodes.append(node.id)
        if node.start:
            ts.startnode = node.id
        elif node.final:
             ts.goalnodes.append(node.id)


    for trans in automata.transitions:

        t = trans.split(" ")
        if t[1] not in ts.actions:
            ts.actions.append(t[1])
        if t[1] not in ts.transitions:
            ts.transitions[t[1]] = []
        l = ts.transitions[t[1]]
        l.append([t[0], t[2]])


    return ts
```

```
def createavoidanceset(states):

    l = [x for x in states if x.final == True]
    return l

def alternativecreation(nos, noss, nogs):

    nodes = range(0, nos)

    # starting states
    ss = sample(nodes, noss)
    while len(ss) == 0:
        ss = sample(nodes, noss)

    # goal states
    gs = sample(nodes, nogs)
    while len(gs) == 0 or set(ss).issubset(gs) == True:
        gs = sample(nodes, nogs)

    # substract goals from the nodes set
    nodeset = set(nodes)
    goalset = set(gs)
    finalset = nodeset - goalset

    # copy final list to the left and right side
    finallist = list(finalset)
    leftside = copy.copy(finallist)
    rightside = copy.copy(finallist)

    # randomize the sides
    shuffle(leftside)
    shuffle(rightside)

    # function numbers
```

```
    functionnumbers = sample(range(0, nos*2), len(finallist))
    while len(functionnumbers) == 0:
        functionnumbers = sample(range(0, nos*2), len(finallist))


    transitions = []
    for n in functionnumbers:
        transitions.append(str(leftside[functionnumbers.index(n)])\\
 + " " + "u" + str(n) + " " \\
 +str(rightside[functionnumbers.index(n)]))

    # automata creation. set states and transitions.
    a = automata("1")
    states = generatestates(nos)
    for s in states:
        if s.id in gs:
            s.final = True
        elif s.id in ss:
            s.start = True
    a.states = states
    a.transitions = transitions

    return a


'''
    generate automata of the specified size
'''
def generateautomata(noa, nosm, nos, noss, nogs):

    # automata
    aut = []

    similaritynumber = float(noa) * (float(nosm) / float(100))
```

```
    a = alternativecreation(nos, noss, nogs)
    for aid in range(0, int(similaritynumber)):
        import copy
        b = copy.copy(a)
        b.id = aid
        aut.append(b)

    for aid in range(int(similaritynumber), noa):
        a = alternativecreation(nos, noss, nogs)
        a.id = aid
        aut.append(a)

    return aut

'''
    generate states of the specified size
'''
def generatestates(nos):

    statelist = []
    for i in range(0, nos):
        statelist.append(state(i))

    return statelist


'''
    set the starting states with the specified number
'''
def setstartstates(noss, numberofstartstate):

    startstates = sample(noss, numberofstartstate)
    while len(startstates) == 0:
```

```
        startstates = sample(noss, numberofstartstate)
    for s in noss:
        if s in startstates:
            s.start = True

    return noss

'''
    set the goal states with the specified number
'''
def setgoalstates(nogs, numberofgoalstate):

    goalstates = sample(nogs, numberofgoalstate)
    while len(goalstates) == 0:
        goalstates = sample(nogs, numberofgoalstate)
    for s in nogs:
        if s in goalstates:
            s.final = True

    return nogs

def isinavoidance(i, av):

    for a in av:
        if i == a.id:
            return True
        else:
            return False

'''
    generate transition functions
'''
def generatetransitions(nos, notf):
```

```
    it = float(notf) / float(nos)
    transitions = []
    for i in range(0, int(it)):

        leftside = range(0, nos)
        rightside = range(0, nos)

        shuffle(leftside)
        shuffle(rightside)

        functions = sample(range(0, nos), nos)

        for n in range(0, len(functions)):
            transitions.append(str(leftside[n]) + " " + "u" \\
     + str(functions[n]) + " " + str(rightside[n]))

    return transitions



# automata 100
totalautomatalist = range(100, 1001, 100)
for eachautomatasize in totalautomatalist:
    '''
    similarity : 10
    state : 10
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 10
    similaritypercentage = 10
    numberofstartstate = 1
    numberofgoalstate = 1
```

```
   a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate)
   for each_a in a:
       each_a.serialize("10-10-1" + "\\" + str(numberofautomata))

   '''
   similarity : 10
   state : 10
   final : 2 (%20)
   '''
   numberofautomata = eachautomatasize
   numberofstates = 10
   similaritypercentage = 10
   numberofstartstate = 1
   numberofgoalstate = 2

   a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate)
   for each_a in a:
       each_a.serialize("10-10-20" + "\\" + str(numberofautomata))

   '''
   similarity : 10
   state : 10
   final : 5 (%50)
   '''
   numberofautomata = eachautomatasize
   numberofstates = 10
   similaritypercentage = 10
   numberofstartstate = 1
   numberofgoalstate = 5

   a = generateautomata(numberofautomata, similaritypercentage,\\
```

```
numberofstates, numberofstartstate, numberofgoalstate )
   for each_a in a:
       each_a.serialize("10-10-50" + "\\" + str(numberofautomata))


   '''
   similarity : 10
   state : 20
   final : 1
   '''
   numberofautomata = eachautomatasize
   numberofstates = 20
   similaritypercentage = 10
   numberofstartstate = 1
   numberofgoalstate = 1

   a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
   for each_a in a:
       each_a.serialize("10-20-1" + "\\" + str(numberofautomata))

   '''
   similarity : 10
   state : 20
   final : 4 (%20)
   '''
   numberofautomata = eachautomatasize
   numberofstates = 20
   similaritypercentage = 10
   numberofstartstate = 1
   numberofgoalstate = 4

   a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
```

```
    for each_a in a:
        each_a.serialize("10-20-20" + "\\" + str(numberofautomata))

    '''
    similarity : 10
    state : 20
    final : 10 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
    similaritypercentage = 10
    numberofstartstate = 1
    numberofgoalstate = 10

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("10-20-50" + "\\" + str(numberofautomata))

    '''
    similarity : 10
    state : 50
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 10
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("10-50-1" + "\\" + str(numberofautomata))
```

```
    '''
    similarity : 10
    state : 50
    final : 10 (%20)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 10
    numberofstartstate = 1
    numberofgoalstate = 10

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("10-50-20" + "\\" + str(numberofautomata))

    '''
    similarity : 10
    state : 50
    final : 25 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 10
    numberofstartstate = 1
    numberofgoalstate = 25

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("10-50-50" + "\\" + str(numberofautomata))
```

```
    '''
    similarity : 20
    state : 10
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 10
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-10-1" + "\\" + str(numberofautomata))


    '''
    similarity : 20
    state : 10
    final : 2 (%20)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 10
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 2

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-10-20" + "\\" + str(numberofautomata))

    '''
```

```
    similarity : 20
    state : 10
    final : 5 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 10
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 5

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-10-50" + "\\" + str(numberofautomata))


    '''
    similarity : 20
    state : 20
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-20-1" + "\\" + str(numberofautomata))

    '''
    similarity : 20
```

```
    state : 20
    final : 4 (%20)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 4

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-20-20" + "\\" + str(numberofautomata))

    '''
    similarity : 20
    state : 20
    final : 10 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 10

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-20-50" + "\\" + str(numberofautomata))

    '''
    similarity : 20
    state : 50
    final : 1
```

```
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-50-1" + "\\" + str(numberofautomata))


    '''
    similarity : 20
    state : 50
    final : 10 (%20)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 10

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-50-20" + "\\" + str(numberofautomata))

    '''
    similarity : 20
    state : 50
    final : 25 (%50)
    '''
```

```
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 20
    numberofstartstate = 1
    numberofgoalstate = 25

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("20-50-50" + "\\" + str(numberofautomata))

    '''
    similarity : 50
    state : 10
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 10
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-10-1" + "\\" + str(numberofautomata))


    '''
    similarity : 50
    state : 10
    final : 2 (%20)
    '''
    numberofautomata = eachautomatasize
```

```
    numberofstates = 10
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 2

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-10-20" + "\\" + str(numberofautomata))

    '''
    similarity : 50
    state : 10
    final : 5 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 10
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 5

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-10-50" + "\\" + str(numberofautomata))


    '''
    similarity : 50
    state : 20
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
```

```
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-20-1" + "\\" + str(numberofautomata))

    '''
    similarity : 50
    state : 20
    final : 4 (%20)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 4

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-20-20" + "\\" + str(numberofautomata))

    '''
    similarity : 50
    state : 20
    final : 10 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 20
    similaritypercentage = 50
    numberofstartstate = 1
```

```
    numberofgoalstate = 10

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-20-50" + "\\" + str(numberofautomata))

    '''
    similarity : 50
    state : 50
    final : 1
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 1

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-50-1" + "\\" + str(numberofautomata))


    '''
    similarity : 50
    state : 50
    final : 10 (%20)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 10
```

```
    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-50-20" + "\\" + str(numberofautomata))

    '''
    similarity : 50
    state : 50
    final : 25 (%50)
    '''
    numberofautomata = eachautomatasize
    numberofstates = 50
    similaritypercentage = 50
    numberofstartstate = 1
    numberofgoalstate = 25

    a = generateautomata(numberofautomata, similaritypercentage,\\
numberofstates, numberofstartstate, numberofgoalstate )
    for each_a in a:
        each_a.serialize("50-50-50" + "\\" + str(numberofautomata))
```

A.2 Simulation Graph Results

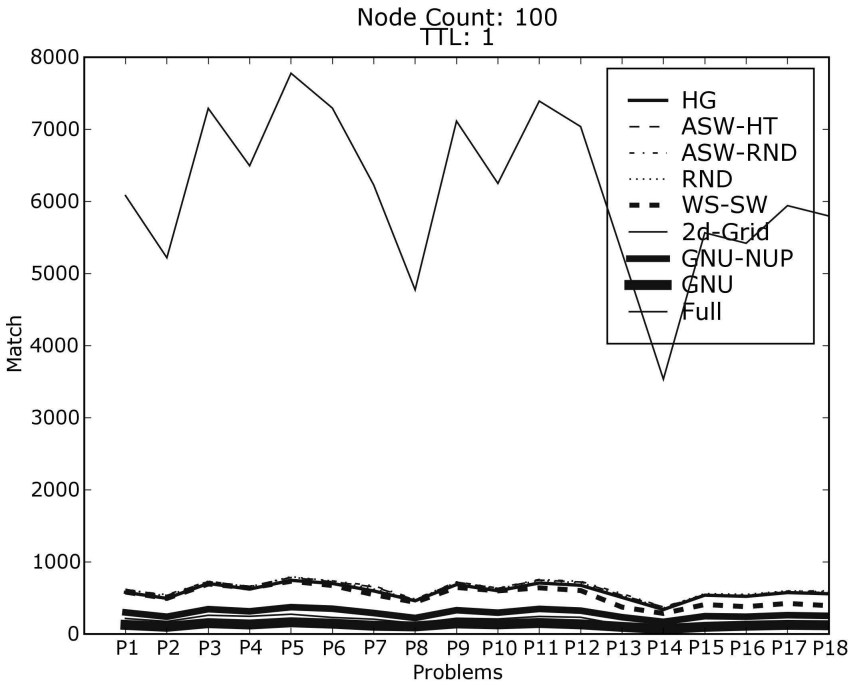

Figure A.1: Match Graph N:100,TTL:1

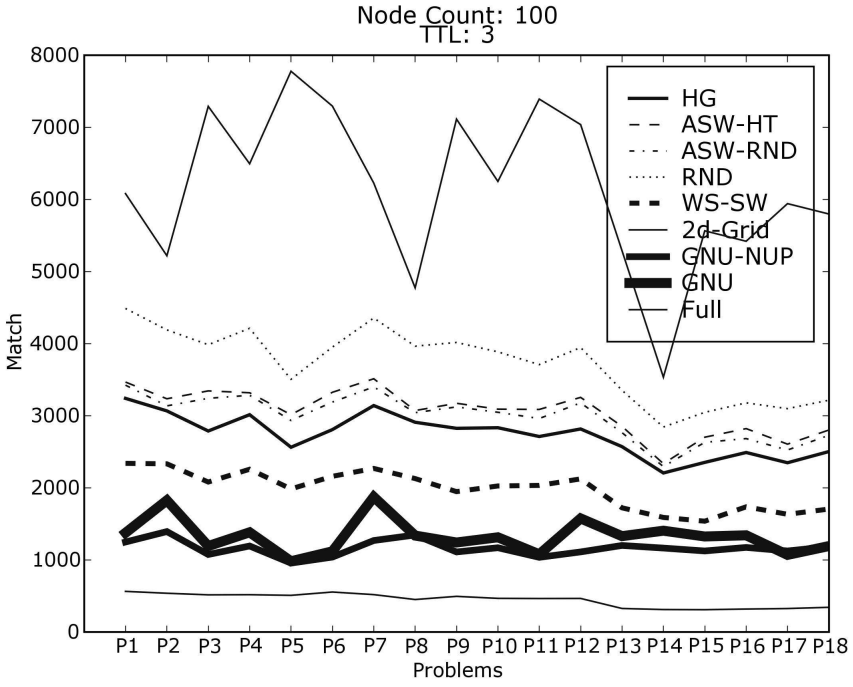

Figure A.2: Match Graph N:100,TTL:3

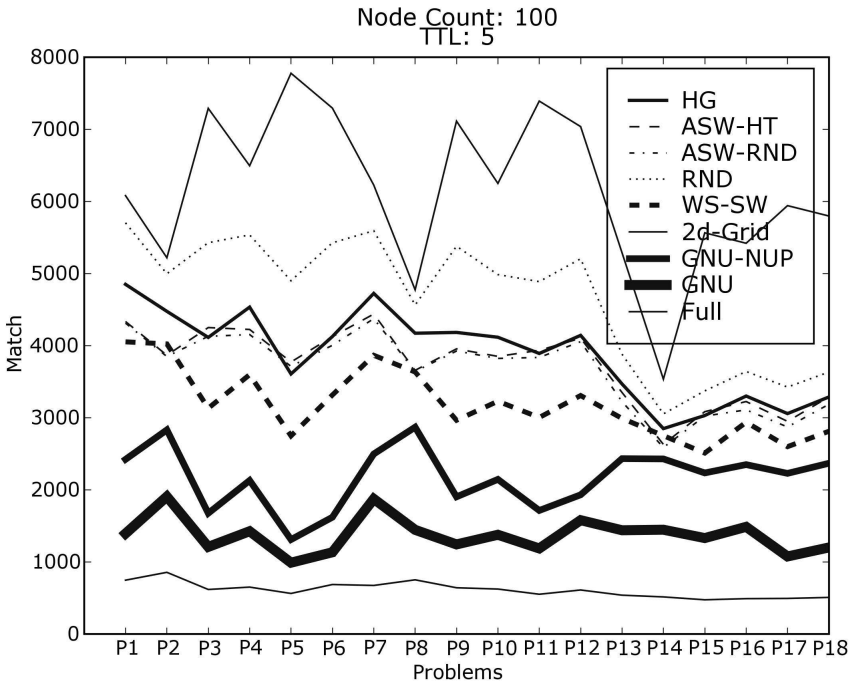

Figure A.3: Match Graph N:100,TTL:5

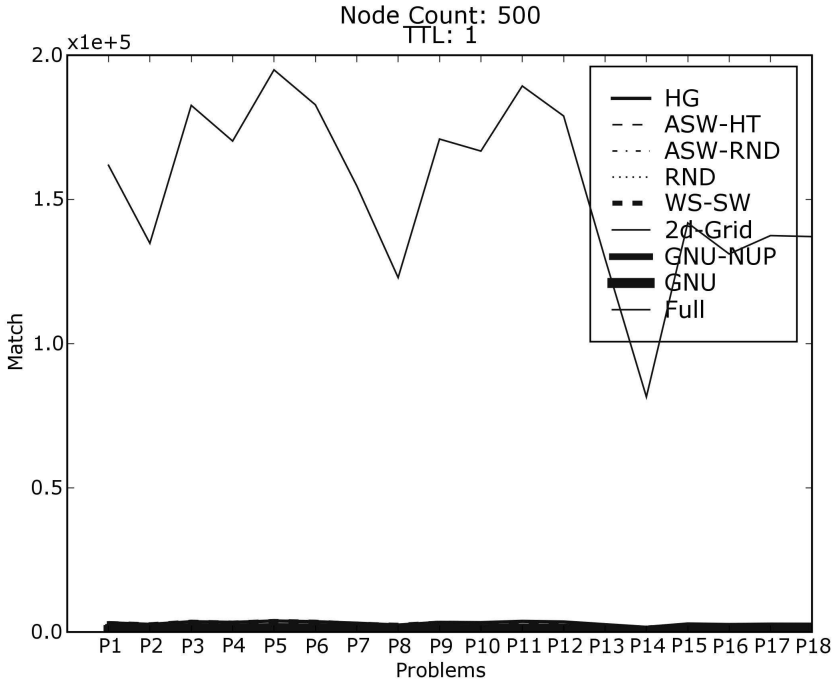

Figure A.4: Match Graph N:500,TTL:1

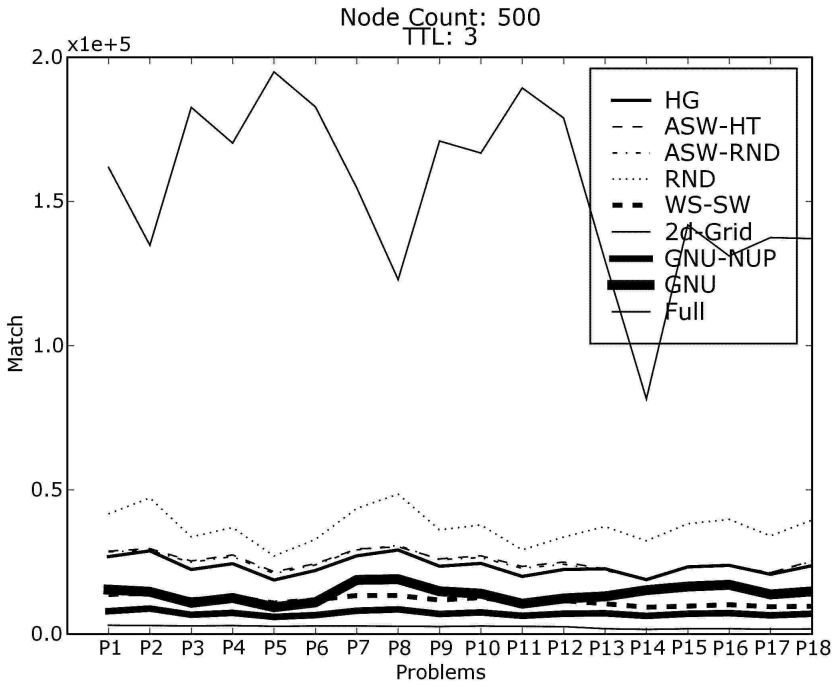

Figure A.5: Match Graph N:500,TTL:3

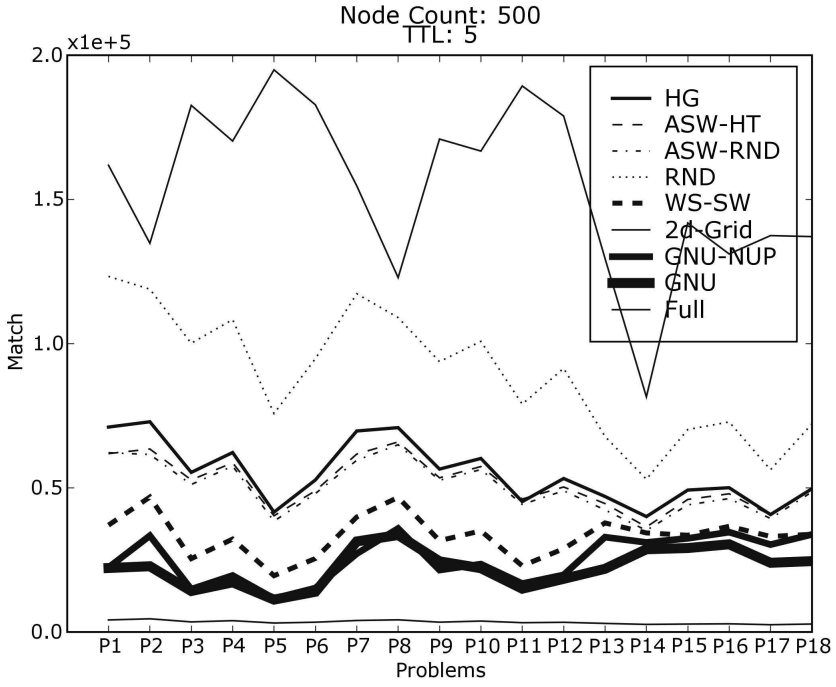

Figure A.6: Match Graph N:500,TTL:5

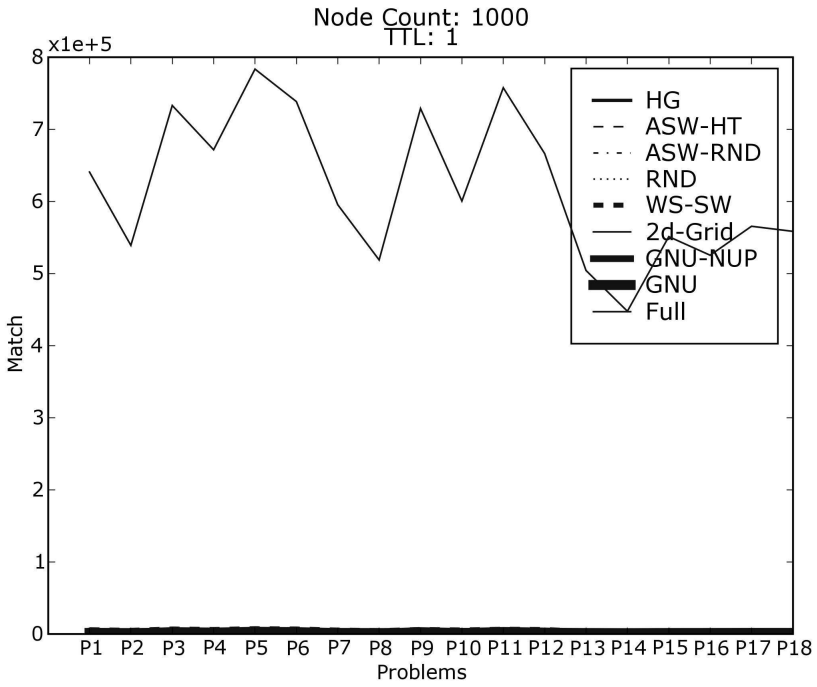

Figure A.7: Match Graph N:1000,TTL:1

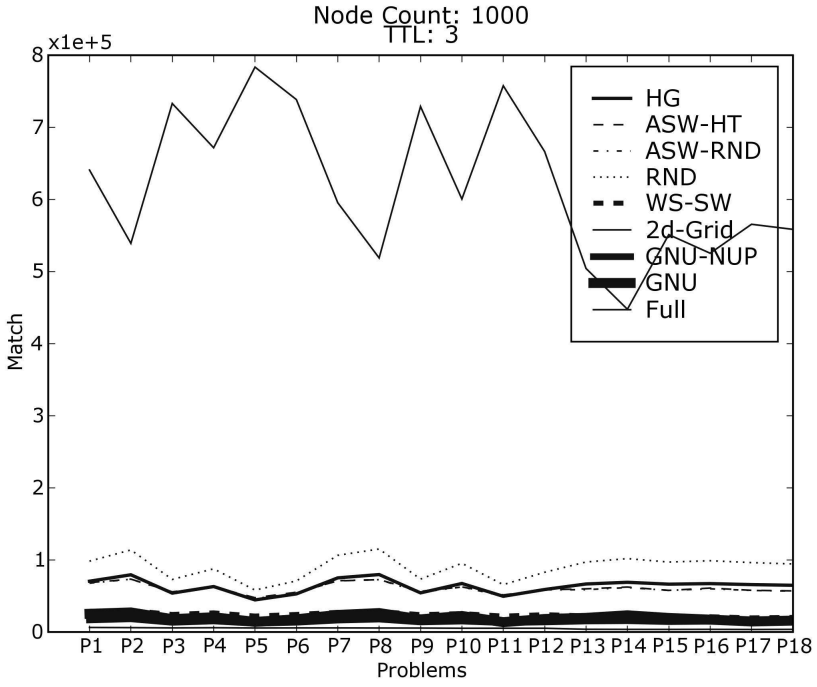

Figure A.8: Match Graph N:1000,TTL:3

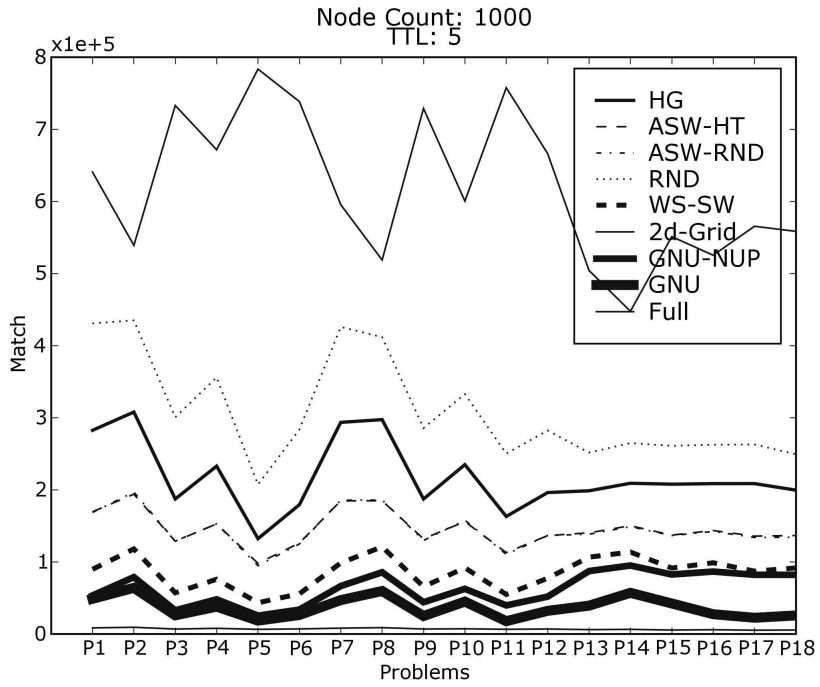

Figure A.9: Match Graph N:1000,TTL:5

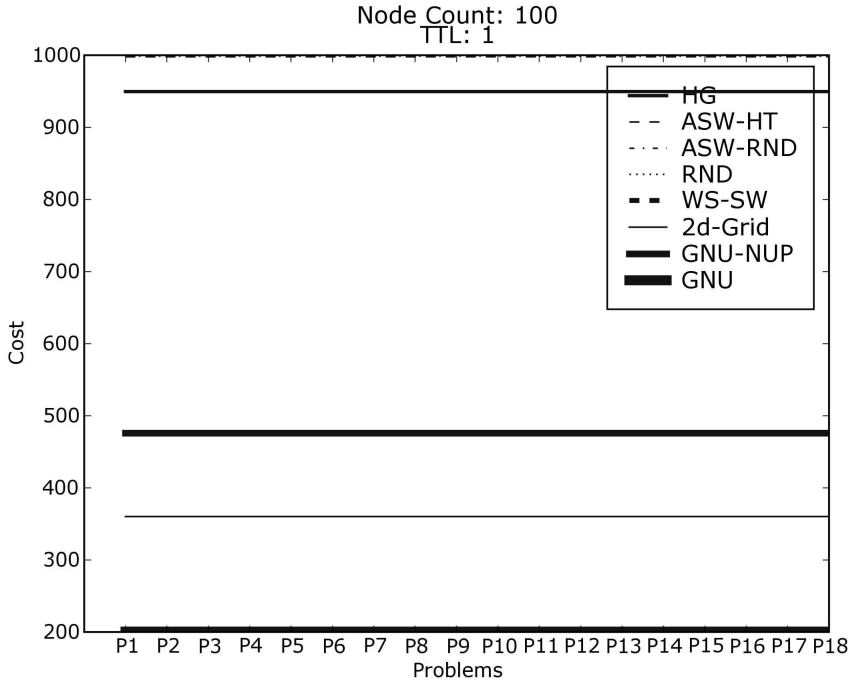

Figure A.10: Cost Graph N:100,TTL:1

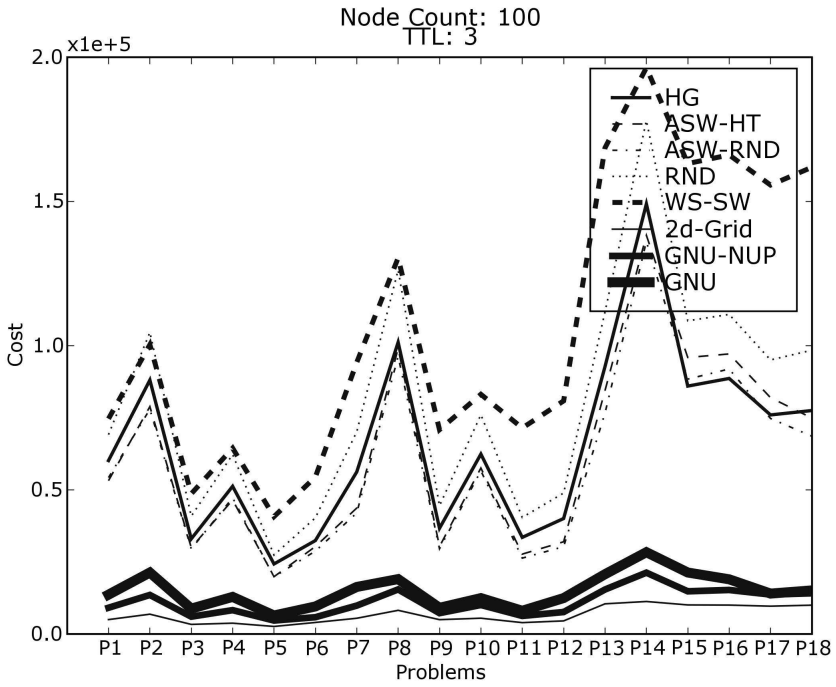

Figure A.11: Cost Graph N:100,TTL:3

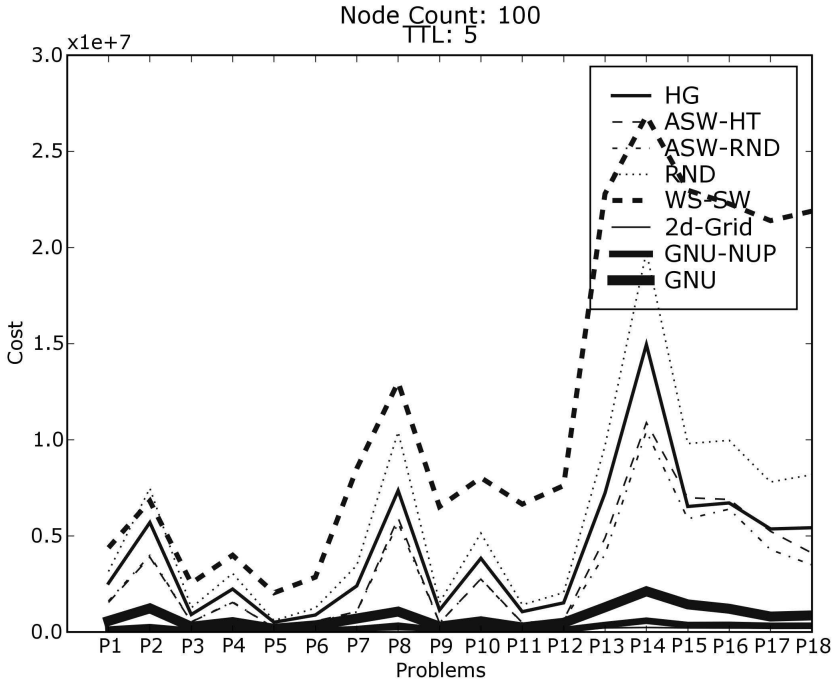

Figure A.12: Cost Graph N:100,TTL:5

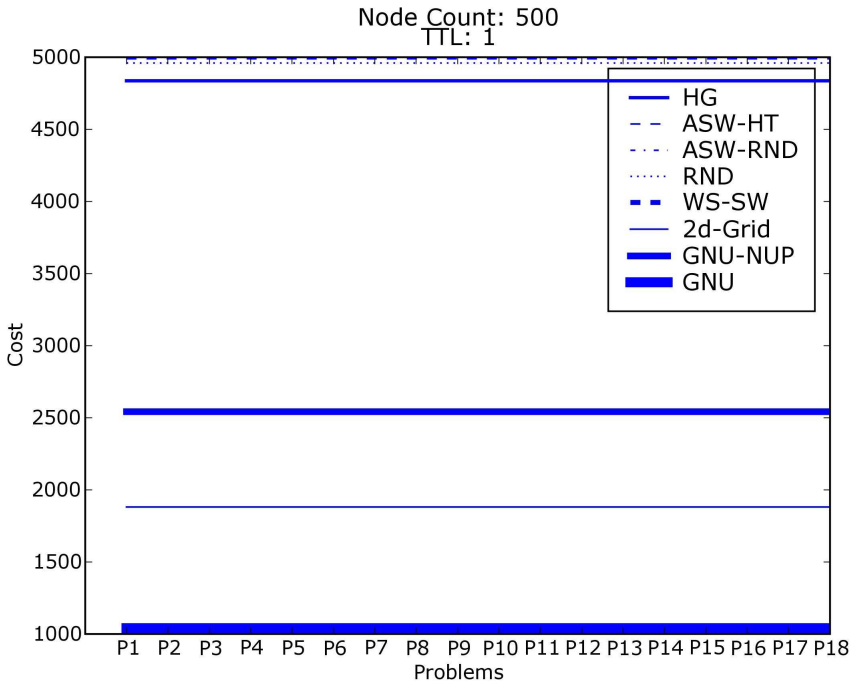

Figure A.13: Cost Graph N:500,TTL:1

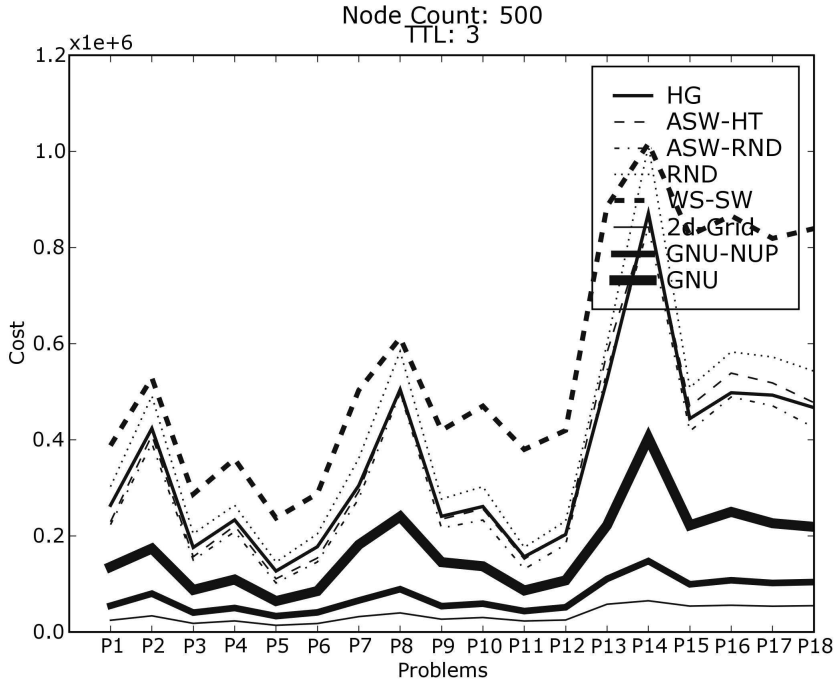

Figure A.14: Cost Graph N:500,TTL:3

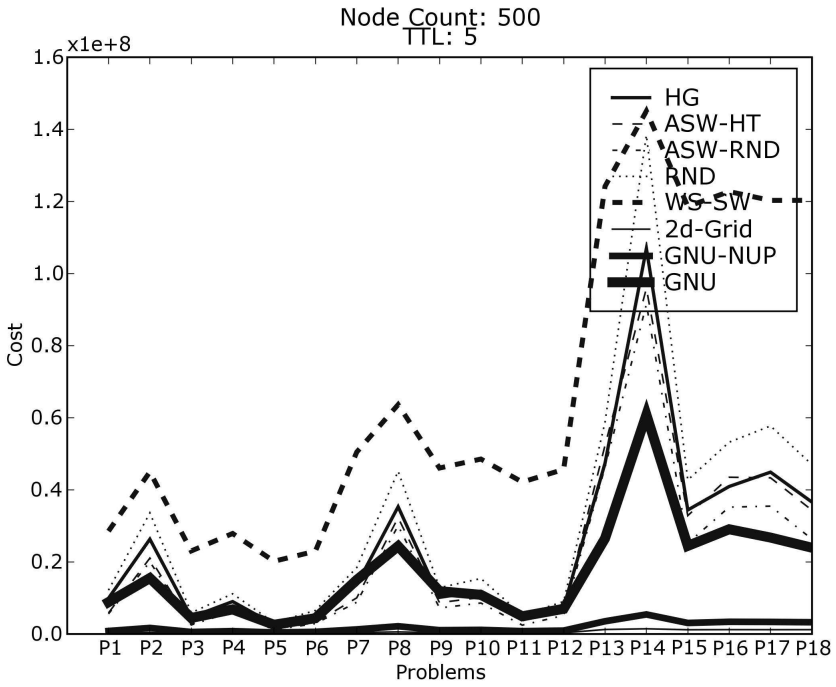

Figure A.15: Cost Graph N:500,TTL:5

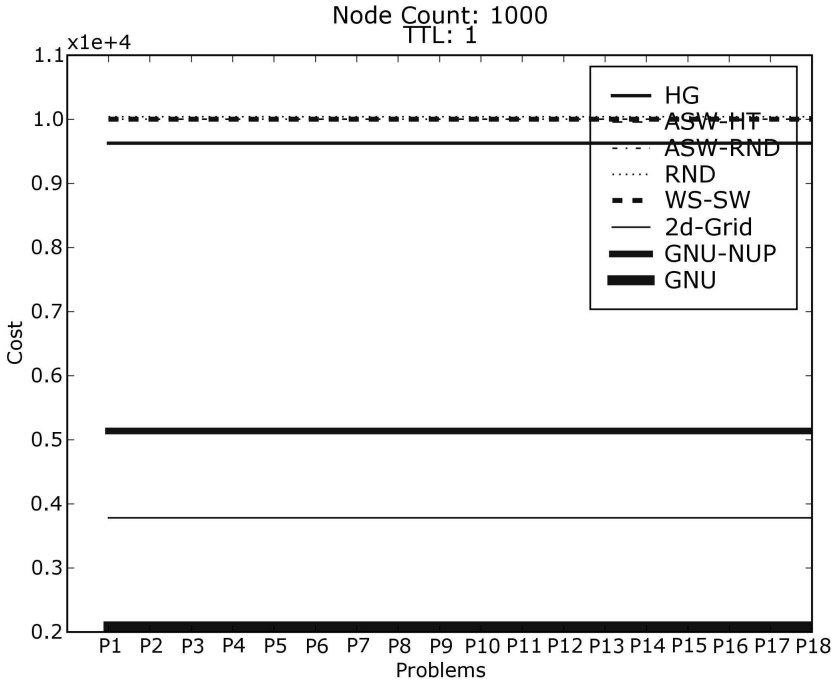

Figure A.16: Cost Graph N:1000,TTL:1

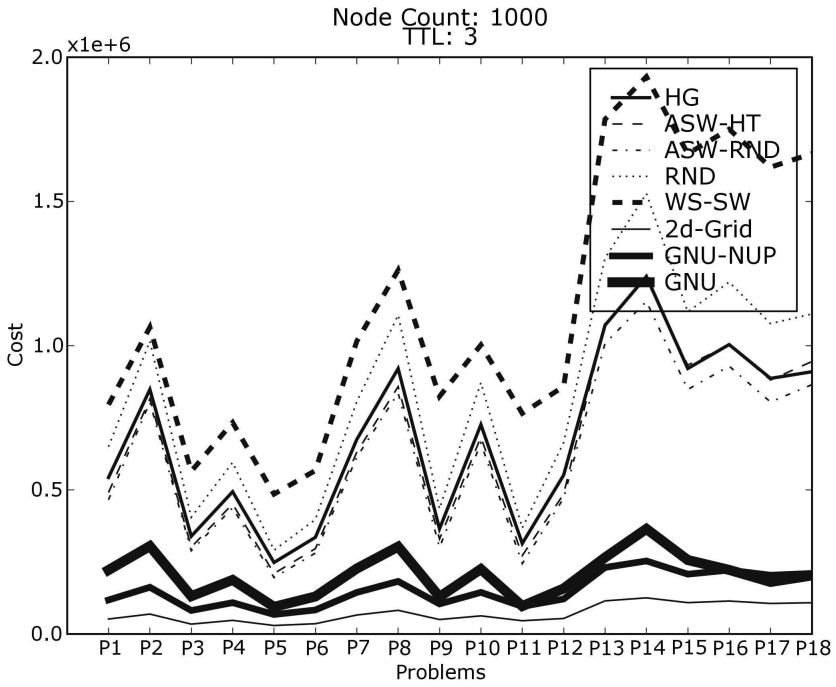

Figure A.17: Cost Graph N:1000,TTL:3

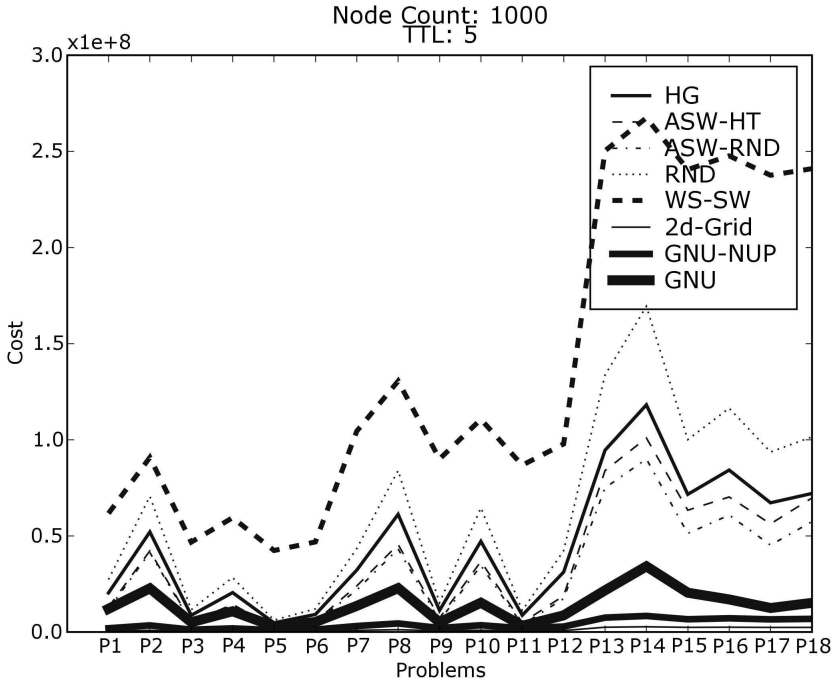

Figure A.18: Cost Graph N:1000,TTL:5