Towards a simulation platform for distributed Multiagent Resource Allocation

Hylke Buisman 0418846
Gijs Kruitbosch 0518832
Nadya Peek 0571709

BSc. Artificial Intelligence
University Of Amsterdam (UvA)
The Netherlands

{hbuisman, gkruitbo, npeek}@science.uva.nl

March 16, 2007

Abstract

Distributed multiagent resource allocation is a field with many interesting and unexplored areas. To explore these systematically and to support theoretical findings there is a need for experimental results. In this report a simulation platform is proposed which hopes to meet these demands. In the framework presented, a user can easily generate a scenario with prespecified amounts of agents and resources, in which the agents have their own preferences and objectives. The agents are able to negotiate amongst themselves to establish trades using money. Using such a scenario, the user is able to run a variety of experiments to see under what circumstances the agents most beneficially manage to reallocate their resources. Finally the platform provides possibilities for visualizing several experiment statistics. Although the platform is not complete up to this point, it does provide a good basis for future work.

Contents

Chapter 1

Introduction

The evolution of computing systems has changed the paradigm of problem solving in many ways. Trends such as their ubiquity, their parallel processing power and their interconnection gave rise to a whole new field of systems. The explosive growth of the internet in the 1990s was especially effective in helping introduce concepts such as software agents (which can act on behalf of a user or other program) and distributed systems (which distribute complex problems) [2, 7]. It became clear that the main way of harnessing the power of many was to combine the behavior of individual agents into an effective composite system, and the term multiagent system was born [10].

The main focus of multiagent system research is the design of an interaction mechanism. This can include both the regulation of interaction between the agents, and occasionally the definition of the intelligent agent architecture itself. Regardless of the power of the designer, the characteristics of the agents and of the problems to be solved need to be known beforehand.

One possibility is that agents are cooperative, meaning they have shared objectives. In this case, the negotiation between agents is mainly focussed on determining which agent has the ability to best solve a certain portion of the problem presented to the system. Two different kinds of systems that address this situation are the *contract net protocol* and the *blackboard architecture*. Contract net protocol has agents submit bids for tasks in which they describe their abilities, after which a central contract manager assigns the tasks. In the case of tasks that are not so modular, the blackboard architecture allows all agents to make use of a global memory to direct coordinated actions and to share intermediate results [15].

If the agents do not share the same objectives, they are said to be noncooperative. In this case, they are assumed to behave according to the principles of *rational decision making*, which means that they act according to their individual preferences and wish to further their personal objectives. Their preferences are generally registered in terms of *utility theory*. Utility, which

is a term transferred from game theory, represents the happiness of each agent after receiving certain goods or tasks. When using utility, it is also possible to use the other mathematical tools provided by game theory such as the concepts of *Pareto optimality*, *Nash equilibrium* and the *minimax* algorithm. This cross-pollination between the fields of economics and computer science has resulted in the heavy scrutiny of the game-theoretic parts of interaction mechanisms, and specifically how to standardize negotiation protocols and agent communication [11].

After determining the behavior of the agents involved, one can focus on the form of negotiation. It is common to distinguish *task-oriented* and *worth-oriented* domains [11]. A task-oriented domain contains a set of all possible tasks, a set of agents and a function which defines the cost of executing any set of tasks. A worth-oriented domain differs in having a set of possible environment states, and instead of having tasks, the agents have joint plans which are evaluated by a function to determine the cost of each joint plan. The agents are no longer only negotiating to minimize their task allocation, but also about what tasks they feel are necessary to reach an ideal state.

A variation of task distribution is resource distribution. Before a multiagent system starts solving a problem, they could be distributing resources. An easy analogy here is how a business manager assigns tools to his employees - without rendering other employees useless, the manager will want to allocate the most effective tools to the most effective employees. This sort of problem can be approached the same way as task distribution, except that instead of taking cost into account, one analyzes the benefits reaped from allocating a certain resource to a certain agent. Like tasks, resources can be divisible or indivisible, and whether they are or not directly affects how they can be distributed over a system of agents.

The actual distribution can be regulated in different ways. The agents may be self-organizing, and may distribute the resources amongst themselves as they see fit. The resources would initially be allocated randomly amongst the agents, and the agents then later set up deals to be able to trade off their initial allocation for one they find more beneficial. Depending on the negotiation tactics and objectives of the agents, this can be an easy grass-roots method for allocating goods. Another option for distribution is *combinatorial auctions*, where a central auctioneer allocates goods according to *package bids* submitted by the agents. Here a package bid is the naming of a price of a set of resources, and the computational burden of figuring out which sets of resources should go to which agent is laid upon the auctioneer [14].

One can allow agents a certain (possibly infinite) amount of money to help them negotiate. Using money in a distributed system, an agent can buy a desirable resource even if he does not have other valuable resources to return. This way, a deal that may otherwise not be rational for an agent

can be made to possibly achieve a more optimal allocation in the future. In a centralized system with finite money, certain richer agents would be able to bid more for certain resource sets, and it would be possible to allow more capable agents to obtain more resources.

As one can see here, many different kinds of problems can be presented to multiagent systems. Each problem presented will have certain attributes. The goods could be atomic or divisible. The goods which are to be distributed could be tasks or resources. Individual agent actions and achievements may or may not need to be shared amongst the agents while solving the problems. Agents may or may not know of other agent's preferences and intentions. The agents may or may not be able to reason about possible worlds. Agents may be strategic or competitive- they may or may not be able to use knowledge of other agents' intentions. All of these attributes affect how a solution is to be obtained, and even merely listing all the possible problems that could be encountered is beyond the scope of this report.

The purpose of our work here was to create a simulation platform we could use in testing what kind of restrictions one could enforce in a society of agents to find how a society could reach different kinds of optimums. There has been a lot of theoretical work on these restrictions, but we believe that testing things with a simulation platform may shed light on new possibilities and help in developing new ideas. Once we find other kinds of restrictions on individual agents which benefit the society as a whole, we can use these to design better interaction mechanisms for multiagent societies.

The report is structured into four main parts. First we provide the reader with the necessary background theory. Then we give a detailed outline of the issues we would like solve in a simulation platform. In the third part, we explain and discuss our implementation. Finally we discuss some of the results we have obtained using the simulation platform.

Chapter 2

Background theory

2.1 What is Distributed Multiagent Resource Allocation?

Resource allocation aims to achieve certain goals in the future by optimally distributing all available resources. A multiagent system is a system in which multiple agents are collectively capable of reaching goals that are difficult to achieve individually. Distributed Multiagent Resource Allocation is the process of distributing a number of resources over a number of agents, where the computational burden can be shared amongst many agents.

There are many different kinds of resources though, which can be distributed differently amongst agents which may have different preferences. In each allocation system, there may be different ways to make deals. This could be done with or without money, and with or without restrictions imposed on the agents for consistent and rational behavior.

There is even an altogether different kind of multiagent resource allocation. In a *combinatorial auction*, the individual preferences of the agents in the system are taken into account to be able to produce the most beneficial allocation, but the computation of the allocation is done by one entity. In this report we are mostly interested in distributed multiagent resource allocation of non-sharable indivisible resources, and we will try to shortly outline the other various possibilities in such a system below.

2.2 Terminology and Symbolism

Here we will give a short outline of some of our commonly used terminology and the symbols we use to represent concepts in this report.

Agents

The set of all agents is denoted as $\mathcal{A} = \{a_1, a_2, ..., a_n\}$.

In this report we mainly work with *individually rational* agents. This means that the agents will not behave in a manner that could be detrimental for their own welfare. For a formal definition, see 3.3.3.

Resources

The set of all resources is denoted as $\mathcal{R} = \{r_1, r_2, ..., r_m\}$.

Resources can be discrete or continuous. Common examples here are cakes (which one can slice) and hard candies (which one consumes individually). In the case of discrete resources, they can be sharable (such as an internet connection) or non-sharable (such as an IP address). In this report we will only consider indivisible non-sharable resources.

Allocations

An allocation A is a division of resources $\mathcal{R}$ amongst the system of agents $\mathcal{A}$.

Every distribution of resources amongst a group of agents is called an allocation. The restrictions that hold upon each allocation in this report stem from the limitation to indivisible non-sharable resources. This means that no resource may be held by two agents at once, and that the sum of each agent's set of resources should become the total set of resources $\mathcal{R}$.

Utility Functions

A utility function is a quantification of an agent's preferences in regards to the resources. These preferences can be expressed in several different ways. A cardinal preference structure assigns numerical values to possible allotments. An ordinal preference structure has agents value certain resources over others. A binary preference structure allows agents to either like or dislike resources. In this report we will use a cardinal preference structure.

In our representation a utility function $u : 2^{\mathcal{R}} \rightarrow \mathbb{R}^+$ maps a set of resources $\{r_1, r_2, ...r_m\}$ of any length from 0 to $|\mathcal{R}|$ to a value in real positive numbers or zero.

Deals

To be able to advance from one allocation to the next, the agents need to make deals. Any deal δ consists of two allocations (A, A') where A and

A' are different allocations of all the resources $\mathcal{R}$. The most simple kind of deal is one where one resource is passed from one agent to another (the 1-resource-deal). You also have cluster deals, in which an agent passes multiple resources to another agent, and bilateral deals, in which agents swap resources. You can extend these types of deals to multiagent deals, in which multiple agents swap single resources, and to combined deals, in which multiple agents swap single or multiple resources. Finally you can also resort to general deals, with any amount of agents and any amount of resources.

Money

To make sure that an agent is willing to give up a certain resource even if its loss will entail a loss in utility, you can introduce money. It is not irrational to expose the agents to unlimited supplies of money to be able to buy resources off each other. Using *payment functions*, one can avoid having to specify who pays who what for each different kind of deal, and instead only show how much each agent pays (to compensate for a gain in utility) and how much each agent receives (to compensate for a loss in utility).

Payment Functions

A payment function p from $\mathcal{A}$ to $\mathbb{R}$ is a function such that the sum of all the payments made during one allocation and by all agents is always equal to 0. A payment from a specific agent a_i is denoted by $p(i)$. When the value of the payment function is negative, the agent receives money, otherwise the agent is paying money.

In this report we only refer to two different payment functions, the locally uniform payment function (LUPF) and the globally uniform payment function (GUPF). In the former, the increase in social welfare is equally distributed amongst the agents participating in the deal. In the latter, the increase in social welfare or 'social surplus' is distributed amongst all agents in the system.

Social Welfare

Based on the utility functions, one can calculate the social welfare $sw(A)$ of the system. There are different kinds of social welfare which you can calculate depending on the objectives of your society. Each definition of social welfare gives a concrete representation of the state of the society at a certain time.

In our report, we mainly use *utilitarian social welfare* (see also 2.3) for which the formal definition is:

$sw(A) = \sum_{i \in \mathcal{A}} u_i(A)$
where u_i are all utility functions.

If the agents manage to more optimally allocate their resources, they can maximize the sum of their utilities, which will increase their social welfare. Depending on which optimal state the agents are striving for, different social welfare definitions may apply.

2.3 Objectives

Some important questions in distributed multi agent resource allocation are: When to stop? Should one stop only after finding an optimal state, or will a feasible state do? What are suitable measures for optimality or feasibility? Should time limits be imposed? What would be the difference in computational complexity of a feasible state versus an optimal state?

An optimal state could be defined as the maximum of the sum of individual utilities. This is also known as *utilitarian social welfare*. In other cases, you could want to maximize the individual utility of the poorest agent, or of the richest. That is known as *egalitarian social welfare* and *elitist social welfare*. You might want to create an *envy-free* system, where no other agent would prefer another agent's bundle over his own.

A feasible state could be defined as any progression towards the optimum. A reason to settle for a feasible state could be a time limitation, where it is necessary to find the best possible allocation in a fixed time span. Other reasons could be attempts to minimize computational complexity. Not having to calculate the actual optimal allocation relieves a great deal of computational burden.

Pareto optimality

Pareto dominance is a relation between two allocations where the dominant allocation is strictly preferred by at least one agent and opposed by none. An allocation is *Pareto optimal* if it is not Pareto-dominated by any other allocation.

Utilitarian Social Welfare

Utilitarian social welfare is the sum of individual utility. The maximum utilitarian social welfare is achieved by maximizing the average agent's welfare. When agents behave according to the criterion of *individual rationality* (see 3.3.3), are allowed an infinite amount of deals and can pay each other to compensate loss in individual utility, they will always achieve optimal

utilitarian social welfare [13]. If we refer to an unspecified kind of social welfare or utility in this report, we are referring to utilitarian social welfare.

Egalitarian Social Welfare

Egalitarian social welfare takes the poorest agent's utility as a measure for the system. Unlike utilitarian social welfare, there is no clear behavior criterion which guarantees optimal egalitarian social welfare. One of the criteria that helps progress egalitarian social welfare is *Pigou-Dalton transfers*. Using Pigou-Dalton transfers, agents do not aim to maximize their own social welfare, but instead attempt to minimize the difference in social welfare between themselves and fellow agents. See also section 3.3.3.

Elitist Social Welfare

Elistist social welfare takes the richest agent's utility as a measure for the system's welfare. In an elitist system, each agent needs to attempt to maximize only its own social welfare.

Envy-Freeness

An agent i is said to *envy* another agent j if agent i would prefer to own agent j's bundle of resources. Another objective could be to attain an *Envy-free* system. In this case, you would need to maximize the average social welfare of every agent in the system and also create other restrictions.

To be able to find more ways that agents could be restricted to converge to a certain social optimum, it would be useful to have a simulation platform in which you can test what happens to a society when these restrictions are imposed. After getting some ideas from a simulation, we might be able to find more theoretical results that didn't seem as obvious before.

Chapter 3

Problem outline

3.1 Introduction

Within the field of Multiagent Systems this report focuses on the area of resource allocation, due to its applicability in many fields such as socio-economics and computing. Indeed, quite a few simulation platforms have been created for multiagent systems (see [5] for a listing). However, most of these are so general, that it may become rather time-consuming to set up and evaluate specific *resource allocation* related problems with these tools. In addition, since some tools have been made for generating scenarios for combinatorial auction problems (e.g. CATS [9]) we will focus on the area of *distributed* multiagent resource allocation, as well as the actual running of experiments using these scenarios.

In this chapter the issues concerning a simulation platform for this field will be discussed. The simulation process can be divided into three parts: *scenario generation*, *negotiation policy* and *experiment support*. The *scenario description* module is responsible for setting up the experiment by defining the settings that are used. The *negotiation policy* module handles the experiment running. Finally the *experiment support* module ensures that all activity is logged and provides facilities for the visualization of experiment statistics. The aspects and difficulties of each module will be put forward in the following sections.

3.2 Scenario generation

The simulation process starts off with creating a scenario. A scenario is defined by the following parameters:

- The number of agents taking part in the negotiation

- The number of resources
- The utility functions used by the agents
- Initial allocation

Changing these settings may have different results per setting. Changing some will mainly influence the computational load (e.g. number of resources) and other can influence whether an optimal state can be reached (e.g. utility functions used). A lot of variation is possible in selecting the utility function, which makes it an interesting setting. Section 4.3.2 will expand on how our platform provides the possibility for random utility function generation.

3.3 Negotiation policy

Once all settings pertaining to the scenario are set, the negotiation module comes in. The major part of the computational load is due to this part of the process. In this section the most important aspects of the negotiation stage are discussed. This may help to clarify why certain functionalities are included in our simulation paltform. First of all the issue of money will be discussed. Then deal selection is introduced, and finally the possiblities regarding agent rationality and deal acceptability are elaborated.

3.3.1 Payments

In order to compensate for deals that are not advantageous for an agent, the use of money is be introduced in the negotiation stage. Here classes of deals, such as 1-resource-deals, are now made possible. Without money, an agent would never donate a resource to another agent (assuming positive utilities). However with side-payments introduced, the agent can donate its resource in exchange for money.

Monetary payments

During the negotiation process many payments can be made and received by one agent. However, instead of storing all payments for each pair of agents, this can also be stored as the total amount of money an agent receives or pays. This can be modelled by a payment function $p : \mathcal{A} \rightarrow \mathbb{R}$, which has to satisfy $\sum_{i \in \mathcal{A}} p(i) = 0$. That is, no money can enter or leave the society of agents. This function is defined as such, that when $p(i) > 0$, agent i pays the amount of $p(i)$ in this negotiation round. When $p(i)$ is negative, however, i receives the amount of $-p(i)$.

As described by Chevaleyre et al. in [3], at least two uses of money can be distinguished:

Infinite money In this situation all agents possess an unlimited amount of money. This means that they can pay any amount necessary. To even more increase the variety of possible scenarios, other additions to this scheme are possible. Examples are products/tasks that have a cost of ∞ (Andersson and Sandholm [1]) and extending utility functions so that situations such as $u_i(R) = \infty$ (Endriss et al. [6]), which implies that an agent is willing to pay *any price* to obtain this resource R.

Limited money It is obvious that having infinite money is not always a realistic assumption. A more reasonable one is where the amount of money is limited. In this case, money can be seen as just another resource.

Necessity

As one can imagine, the possibility of side-payments make the negotiation process more versatile. However, there are even some specific scenarios where side-payments are necessary to guarantee that some maximal state can be reached. An example can be found in [8].

Types of payment functions

When payments are used, several payment functions $p(i)$ can be chosen from. Important to our project are mainly the LUPF and the GUPF payment function, which are the most simple ones available.

LUPF As explained in the terminology, the LUPF payment function divides the social welfare surplus (i.e. the increase of social welfare) over all agents $\mathcal{A}^\delta$ that were involved in the deal. In formula form this is:

$$p(i) = [u_i(A') - u_i(A)] - \frac{[sw(A') - sw(A)]}{|\mathcal{A}^\delta|} \text{ If } i \in A^\delta \text{ else } 0$$

GUPF In contrast, the GUPF divides the social welfare surplus over all agents in the agent society. In formula:

$$p(i) = [u_i(A') - u_i(A)] - \frac{[sw(A') - sw(A)]}{|\mathcal{A}|}$$

Since using different payment functions may render certain social optimal allocations unreachable, it is clear that these parameters should be included in a good simulation platform. For example, using GUPF and IR (see section 3.3.3) deals, will eventually terminate in an efficient and envy free state (see [4]). With LUPF this is not necessarily the case.

3.3.2 Deal selection

Deal selection is concerned with choosing which resources an agent trades with another, if any. In a simulation platform, this amounts to selecting the agents that will make a deal, and choosing which resources they will exchange. It is perhaps the most difficult part (from a theoretical point of view) of negotiation. This is mainly due to the potential combinatorial nature of the problem. The influence of this part is, however, too important for it to be handled superficially. Selecting a different deal selection algorithm can influence the performance of the resource allocation process. It may even influence whether an optimal allocation can be reached. In the section on experiment running (Section 4.5), more information is given about the implemented deal selection algorithms.

3.3.3 Agent Rationality

A final characteristic that has great influence on the negotiation process is the agent's rationality. This characteristic determines whether or not a proposed deal is acceptable for that agent. We will discuss a criterion called *Individual Rationality* and the *Pigou-Dalton* transfers.

IR

Individual Rationality (IR) is a criterion that best fits a utilitarian society. In short, this criterion states that a deal is acceptable for an agent *iff* the corresponding utility increase is greater than the amount of money the agent pays. The reverse also holds: The amount an agent receives should be higher than the decrease of the agent's utility. That is:

$$u_i(A') - u_i(A) > p(i) \text{ for all } i \in \mathcal{A}$$

$$\text{Possibly } p(i) = 0 \text{ if } A(i) = A'(i)$$

As shown in [6] individually rational deals will always lead to an increase in social welfare. This fact shows why this criterion is useful in a utilitarian setting, since this strives for a maximal social welfare.

Pigou-Dalton transfers

This criterion best fits an egalitarian setting. Pigou-Dalton transfers try to reduce the difference in utility between two agents. These transfers have to satisfy the following constraints:

- $\mathcal{A}^\delta = \{i, j\}$
  Only the agents i and j are involved in the deal.

- $u_i(A) + u_j(A) = u_i(A') + u_j(A')$
  The transfer does not change the total amount of utility
- $|u_i(A') - u_j(A')| < |u_i(A) - u_j(A)|$
  The deal reduces inequality

Although this is an interesting criterion, it has not yet been implemented in our platform. It is also not a criterion that ensures convergence to an optimal state, unlike individual rationality. Finding more criteria which do demand convergence would be very beneficial.

Since varying rationality criteria best fit varying social optima, this is a setting that should be changeable when running experiments.

3.4 Experiment support

The greatest distinction between a Multiagent systems implementation and a Multiagent systems simulation platform is the experiment support. For a platform to be useful, it should provide sufficient facilities for calculating and visualizing the statistics of the foregoing experiment. In addition it is desirable to be able to compare different experiments with each other. Here a mixture of practical and theoretical difficulties are encountered. 'What is the best way of storing an experiment?' and 'What types of visualizations are most useful?' are practical issues that need to be addressed.

Chapter 4

Implementation

4.1 Introduction

Given the theoretical ideas and results discussed in the previous sections, we wanted to try to obtain more results by actually testing them in a simulation. We hoped that testing them in such a practical way would perhaps give an indication as to where other interesting theoretical results might be found. In order to do so, we wrote an implementation of Multiagent Resource Allocation ourselves. We decided to use Java because of its platform-independent and Object-Oriented features.

4.2 Architecture

We divided the simulation platform into three main parts: the *scenario generation*, the *experiment running* and the *graphing*. Each of the parts functions autonomously, and can use an input file generated in a previous section. This way the usage of the system becomes more modular, and the user can for instance generate many possible experiments, and later run them all one after another. A user can also run the same experiment many times, and overlay the outcomes into the same graph to be able to extract common trends. To see a schematic overview of the system, see figure 4.1.

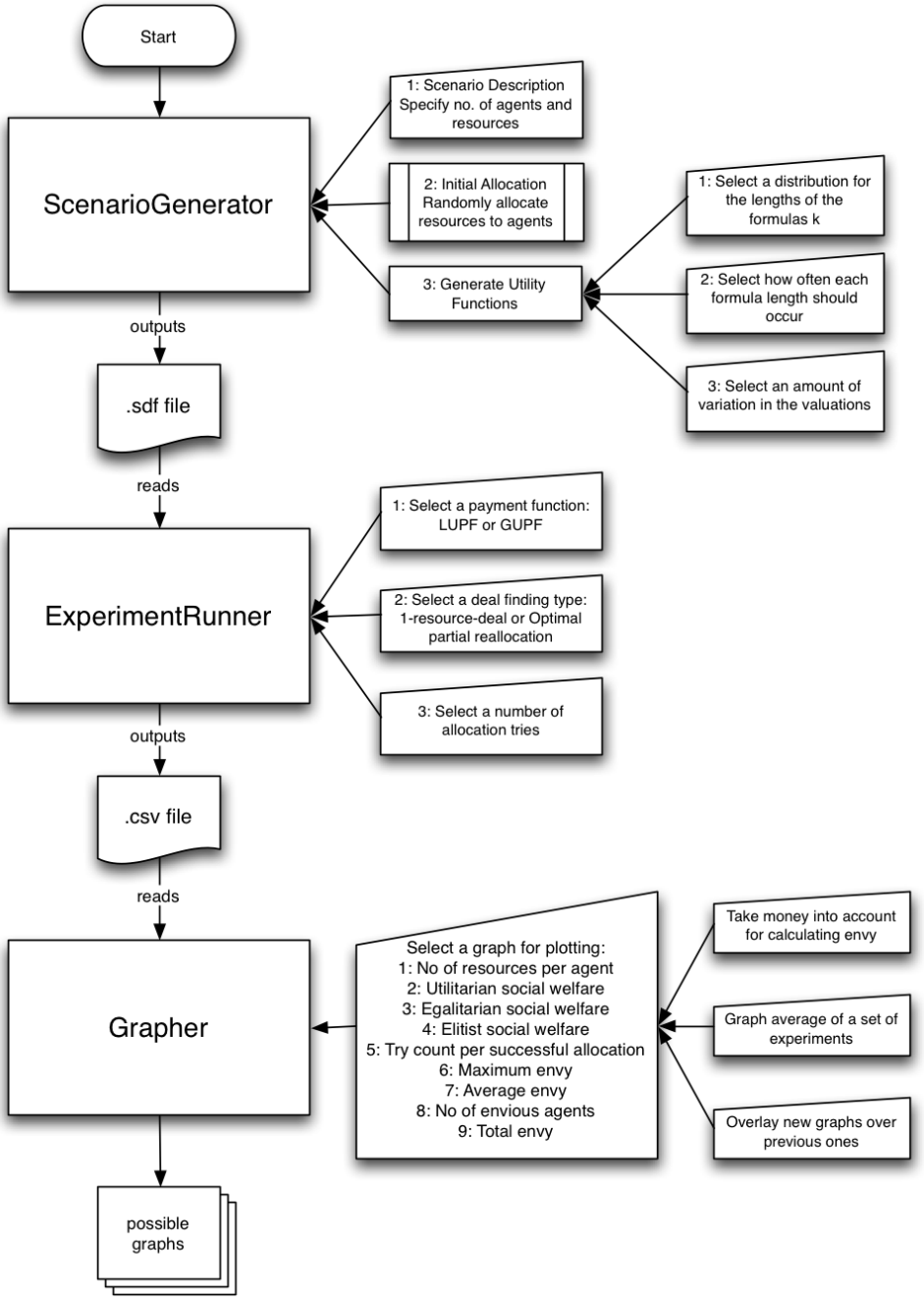


Figure 4.1: An overview of the various packages in the simulation platform and their input parameters.

4.3 Scenario Generation

The first module takes care of the generation of a scenario. In this step, the following items must be defined:

1. the name of the scenario.
2. a description of the scenario.
3. the number of agents to use.
4. the number of resources to use.
5. how the resources are initially divided amongst the agents.
6. how much the agents value the resources (the agents' utility functions).

The name, description, number of agents and number of resources are somewhat self-explanatory. The last two items will be explained in more detail in the following paragraphs.

4.3.1 Initial Resource Allocation

In order to get a working scenario, one must specify an initial allocation of the resources in the scenario amongst the agents involved. To do this, we randomly select an agent for each resource, and then allocate the resource to this agent.

4.3.2 Utility Function Generation

As a next step, it is necessary to supply the agents with utility functions. In the process of simulation, however it is desirable to have control over the amount of randomness in these functions. In some situations randomness is required to differentiate the agents, and at other times it is useful to have predetermined values for the parameters. In this way one can more easily control the experiments.

Weighted Propositional Formulas

In the approach discussed here, Weighted Propositional Formulas are used for the representation of agent preferences. This choice is made, due to its expressive power and its relative simplicity. This approach uses logical formulas to express utility and consists of a *weighted goal base* $GB =$

$\{\langle G_1, \alpha_1\rangle, \ldots, \langle G_n, \alpha_n\rangle\}$. Here G_i is a propositional formula containing resources as atoms and α_i is the weight assigned to the goal. The utility of a certain allocation for agent m is then calculated with:

$$u_{GB}(A(m)) = \sum_{i=1}^{n} \{\alpha_i | A(m) \models G_i\}$$

k-additive form

A more specific form of this approach is called the k-additive form. With this representation, formulas are of the form

$$G = \bigwedge_{r \in T} r \quad \text{With } T \subseteq \mathcal{R} \text{ and } |T| \leq k$$

We will call a goal concerning precisely j resources a j-goal. The weight that is associated to a goal indicates the *additional* utility this subset of resources gives the agent. This form is fully expressive if all (postive and negative) weights are allowed. Note however that in the current implementation the optimal partial reallocation algorithm (see section) assumes that only positive weights are used. Furthermore this form is particularly fit for random generation due to its structure.

Parameters

Although a k-additive UF is much more restricted than when full weighted propositional formulas are used, there are a lot of parameters that can be set. For some of these it is perhaps hard to give a meaningful interpretation of their values, and for others varying the values may not be useful for research or practical ends.

In the remainder of this section a (not exhaustive) number of parameters is given with a few statements about their possible use to give an idea of the variety of parameters. The parameters can be divided in two categories: parameters that cause variation between UFs and parameters that cause variation between the formulas within a UF.

Inter-UF variation

- **The value of k**
  Making k variable will allow the agents to differentiate in the way they value the resources. Do they place a preference on a lot of resources, or only a few?

Inter-Formula variation

- **The amount of goals with a certain length**
  Of all formulas with length j there are at least two reasons one might not want to generate all of these. First of all this is a very computationaly intesive task. Note that the number of possible formulas is equal to the number of possible combinations when choosing j resources out of a total of $|\mathcal{R}|$. This number gets very larger very quickly, for example: if $|\mathcal{R}| = 60$ and $j = 30$ there are 1.18×10^{17} possible formulas. Secondly it can be very useful in order to create realistic scenarios. It might, for instance, seem logical to create a larger percentage of smaller formulas since it is perhaps to complex for an agent to know the additional utility of large combinations of resources.

- **The sum of the weights assigned to all goals**
  That is: $\sum_{i=1}^{n}\{\alpha_i\}$. This parameter indicates how greedy the agent is who owns this utility function. An agent with a higher total weight sum is in some sense more greedy, since it places a larger overall preference on resources. The total weight is implicitly defined by the probability distribution of the assignment of a certain weight.

- **The way this mass is distributed over the j-goals ($1 \leq j \leq n$)**
  In some situations one would want to assign a larger additive utility to singleton sets than to larger sets.

- **The distribution of $\mathcal{R}$ over the different goals**
  Are all resources used in the formulas and which resources are used in which formulas? Some resource might only occur in the shorter formulas or vice versa. Perhaps, however, a random distribution is best for an initial approach.

Parameter instantiation

It is difficult to say what realistic or useful instantiations of these parameters are. Our current implementation thus merely provides several different ways of setting these parameters. Examples are a precise (predetermined) setting or by means of probability distrubtions (such as normal, uniform). It is also possible to manually add modules if these possibilities are not sufficient. What the influence is of each type of instantiation is matter for further research.

Generating formulas of length k

Randomly generating formulas of length k is not as easy as it seems. Its difficulty arises from the fact that an agent should only have distinct formulas.

Thus, if there are 20 possible formulas of length k, and 15 need to be selected, it is likely that the the generation process is time consuming. A more extended analysis and solution to this problem can be found in Appendix A.

4.4 Scenario Representation

In order to conveniently store, modify and share experiment scenarios, we designed a fileformat for these purposes. This section documents that file format, its features and limitations, and it gives a concrete example of what such a scenario file could look like.

We chose to use XML for formatting the data, because XML parsers are ubiquitous by now, so getting the data in and out of the format should be easy, even if one wants to use other tools, or just wants to quickly modify a scenario in order to conduct a slightly different experiment.

4.4.1 Example

The following is an example of a valid XML-formatted description of a scenario involving 5 resources, named r_1 through r_5, and 2 agents, named a_1 and a_2. An agent's utility function is defined by an unordered list of goals, which each have a weight attribute and a value which describes the goal that should be fulfilled in order for the weight to be added in the utility of that agent for his set of resources at that particular time. The initial allocation is semi-random-picked by the authors of this document.

```
<?xml version="1.0"?>
<scenario name="My Scenario for Experiment 2000">
    <description>This scenario was made to test x and y
    and should be run z times and then averaged.</description>
    <resources>
        <resource id="r1"/>
        <resource id="r2"/>
        <resource id="r3"/>
        <resource id="r4"/>
        <resource id="r5"/>
    </resources>
    <agents>
        <agent id="a1">
            <utility>
                <goal weight="3">
                    (and r1 r2)
                </goal>
```

```
                <goal weight="5.6">
                    (or r3 r4)
                </goal>
            </utility>
        </agent>
        <agent id="a2">
            <utility>
                <goal weight="2">
                    (implies r1 r3)
                </goal>
                <goal weight="-4">
                    (biimplies r1 r4)
                </goal>
            </utility>
        </agent>
    </agents>
    <allocation>
        <bundle agent="a1">
            <resource id="r2"/>
            <resource id="r3"/>
            <resource id="r5"/>
        </bundle>
        <bundle agent="a2">
            <resource id="r1"/>
            <resource id="r4"/>
        </bundle>
    </allocation>
</scenario>
```

4.4.2 XML Tags used

In describing a scenario, the following tags and attributes are used:

scenario The `<scenario>` tag contains all the information for the scenario. It must be the root tag. There must be no siblings for this tag. All other listed tags must be descendants of this tag. The `<scenario>` tag should have a `name` attribute giving a (short) name to the scenario.

description The `<description>` tag contains a (longer) textual description for the scenario. It should solely consist of text. It must be a child of the `<scenario>` tag.

agents The `<agents>` tag contains an unsorted list of agents involved in the scenario. It must be a child of the `<scenario>` tag. Its children must solely be `<agent>` tags.

agent The `<agent>` tag contains a description of an agent. It must have an `id` attribute describing the unique identifier associated with this agent. The id shouldn't contain anything other than alphanumeric symbols and the dash and underscore. There must be no other tag within the document with the same id. An `<agent>` tag can contain a `<utility>` tag describing the agent's utility function. An `<agent>` tag must be a child of an `<agents>` tag.

utility The `<utility>` tag contains a description of the utility function of the agent. It must only contain `<goal>` tags. It must be the child of an `<agent>` tag. The formulas representing the agent's preference are Weighted Propositional Formulas (as described in Section 4.3.2).

goal The `<goal>` tag contains a description of a single goal within an agent's utility function. It must have a `weight` attribute that has a numeric value indicating the weight that agent attaches to this specific goal holding in a particular allocation. It should contain text describing a formula that must hold true for the weight to be added to the agent's utility. The formula text should have the Lisp-like syntax described in section 4.4.3 on Goal Representation. A `<goal>` tag must be a child of an `<utility>` tag.

resources The `<resources>` tag contains a description of *all* resources involved in a particular scenario. It must contain only `<resource>` tags.

resource The `<resource>` tag describes a single resource. It must be a child of either a `<resources>` or a `<bundle>` tag. It must have an `id` attribute describing the unique identifier associated with the resource. All resources must appear once, and only once, as a child of the `<resources>` tag and at most once as the child of a `<bundle>` tag. Hence, there should never be more than 2 instances of a single resource id present in any scenario description.

allocation The `<allocation>` tag contains a description of the starting allocation for this scenario. It must be a child of the `<scenario>` tag. It must only have `<bundle>` tags as children.

bundle The `<bundle>` tag describes a bundle of resources allocated to a particular agent in a given allocation. It must be a child of an `<allocation>` tag, and must contain only `<resource>` tags. It must have an `agent` attribute which describes an existing unique identifier of a given agent.

4.4.3 Goal Representation

Goal formulas are represented using a simple Lisp-like syntax. Here's an informal ABNF (Augmented Backus−Naur form) grammar for the syntax used. Note that `<a>*<b>element` means a repetition of `element` with a minimum of `a` and maximum of `b` repetitions. `a` and `b` deault to respectively 0 and infinity.

```
atom =  DIGIT
      / ALPHA
      / "-"
      / "_"
      / 1*atom

formula = atom
        / "(and" formulalist ")"
        / "(or" formulalist ")"
        / "(implies" formula formula ")"
        / "(biimplies" formula formula ")"
        / "(not" formula ")"

formulalist = formulalist formula
            / formula formula
```

In other words, a formula is always one of the following:

- an atom
- a conjunction, i.e. "(and"
- a disjunction, i.e. "(or"
- an implication, i.e. "(implies"
- a bi-implication, i.e. "(biimplies"
- a negation ie "not"

An atom is always at least one character long, and solely consists of alphanumerics and the dash and underscore characters.

A conjunction, disjunction, implication and bi-implication should be followed by at least 2 whitespace-separated formulas, followed by optional whitespace, followed by a closing parenthesis ")".

A negation should be followed by at least 1 whitespace-separated formulas, followed by optional whitespace, followed by a closing parenthesis ")".

4.4.4 Storage

Scenarios should be stored in files with the `.sdf` extension, which is short for Scenario Description File. They should contain at least a collection of agents and resources, but can also provide the utility functions of these agents, and/or the resource allocation of the resources to the agents. To actually run a scenario, all of the above is necessary (but could be assembled from different files).

4.5 Running an Experiment

When running an experiment, essentially what takes place is an iterative search for better allocations. The implementation takes the current allocation and tries to find a better one. When it finds one, it tries to find a new one that is even better, and so on, until a user-specified limit on the number of times it should try to find a better allocation has been reached. The transitions between these allocations are called *deals* . The process of creating a deal is described in the following subsections.

4.5.1 Agent Selection

First, the implementation selects two agents randomly from the set of all agents participating in the scenario. If the second agent is the same as the first agent, it will insist on randomly selecting new agents until this method produces a different agent.

In reality, of course, more than two agents could participate in a deal. This has not been implemented yet. There are two possible ways of finding new deals amongst two agents that we have implemented. Both are discussed in the following subsections.

4.5.2 1-resource deals

A 1-resource deal is, as the name implies, a deal involving just one resource. This is mostly useful if there is money involved, otherwise it would not be possible to transfer most resources (because it would not be rational for an agent to give away a resource he cares about). Note that the current implementation does not support running an experiment without money, so this is not a problem.

This is implemented in a very simple manner. The code loops through all the resources owned by one of the agents, and checks if trading the resource would increase social welfare (which, because we use money, is equivalent with individual rationality). If it does, it stops and trades that resource,

and then the deal is done. If the deal is not rational, it will try the next resource, and if all the resources from this agent don't work out, it tries all those from the other agent. When those fail too, the attempt fails entirely.

4.5.3 Optimal Partial Reallocation

Another possibility of selecting a deal between two agents is optimal partial reallocation. This approach redistributes all the resources owned by two agents in the best way possible. To figure out what an optimal reallocation of resources between two agents is, we will employ A*. This approach was inspired by the proposed algorithms for combinatorial auction problems used by Sandholm for Optimal Winner Determination as described in [14]. When using A* for a certain problem, states, moves and a heuristic need to be defined. To apply A* to this situation, the following definitions are chosen.

A* - States

In combinatorial auction problems, Sandholm shows that a bid-oriented approach tends to give better performance in practice, compared to a good-oriented approach [12]. With this in mind it would be a sensible choice to use a utility-oriented approach. However, finding an admissible heuristic for this approach is rather difficult, and it is not at all evident that the effort of finding such a heuristic is outweighed by the performance improvement. Thus the state space will be defined from a resource perspective. Here a state is defined as:

- For each agent the set of resources which has been allocated to it.
- The set of unallocated resources.

In the initial state no resources are allocated. The final states are characterized by having all resources are allocated.

A* - Moves

Given this state space, a move amounts to making a decision for one resource from the set of unallocated resources. Here we assume that the resource to be allocated is always the 'next in line', or random. This way no actual reasoning has to be done in selecting a resource. Improvements to this approach could consider heuristics for selecting the next resource. The resource in question can be allocated to only one of the agents, two in this case. As a consequence every state has only two successor states.

A* - Heuristic function

In addition, the function f needs to be defined, for the A* algorithm to work correctly. This function is calculated as follows:

$$f(x) = g(x) + h(x)$$

Here $g(x)$ is the total cost of taking this path up to node x, and $h(x)$ the estimation of the costs that can be expected in the remainder of this path. $h(x)$ is better known as the heuristic function. To ensure that the solution path is optimal, the heuristic must be admissible. This means that the chosen heuristic should make an 'optimistic' estimate of the costs. This means that the estimate of the costs should always be lower than the actual costs. However, since this situation deals with utility instead of costs, the heuristic should always overestimate the utility.

Choosing the function $g(x)$ is not difficult: it is the social welfare of the small subcommunity consisting of the two agents. Since it calculates the precise utility of the current allocation, it represents the total utility of the path up to the node x.

For $h(x)$ the following formula will be chosen:

$$\sum_{i \in Agents} \sum_{(G,\alpha) \in Goals_i} \alpha \tag{4.1}$$

Here (G, α) signifies a goal G and its weight α.

Here $Goals_i$ only contains those goals that still have the possibility to yield any extra utility for agent i. This means that $Goals_i$ equals agent i's goal-base minus those goals which contain resources that have been allocated to another agent. Additionally all goals are removed that are already satisfied by the agents currently allocated resources.

Admissibility of the heuristic One can easily show that our heuristic indeed overestimates the actual utility increase. The given function estimates the total utility increase in the remainder of the path by assuming that all goals that can be satisfied, will be satisfied for each agent. It is clear that the actual amount can not be any higher, since all goals are taken into account. The goals that are left out in $Goals$ do not have to be taken into account since they will never yield any additional utility: the agent will never acquire one or more of the resources that are required to satisfy the goal.

However, this heuristic in not the best possible, as the estimate is rather far from the actual utility increase. It is clear that, in most situations, only a few of all goals in $Goals$ will be satisfied since satisfying a goal for one

agent will in many cases invalidate one or more goals for another. As a consequence only a few weights will be added to the utilities of both agents. Regardless of these issues, we will stick to this heuristic for now.

Example search tree

Using these parameters, A* can now be applied. To give more insight into the workings of the search process, a search-tree of a small problem is shown in Figure 4.2.

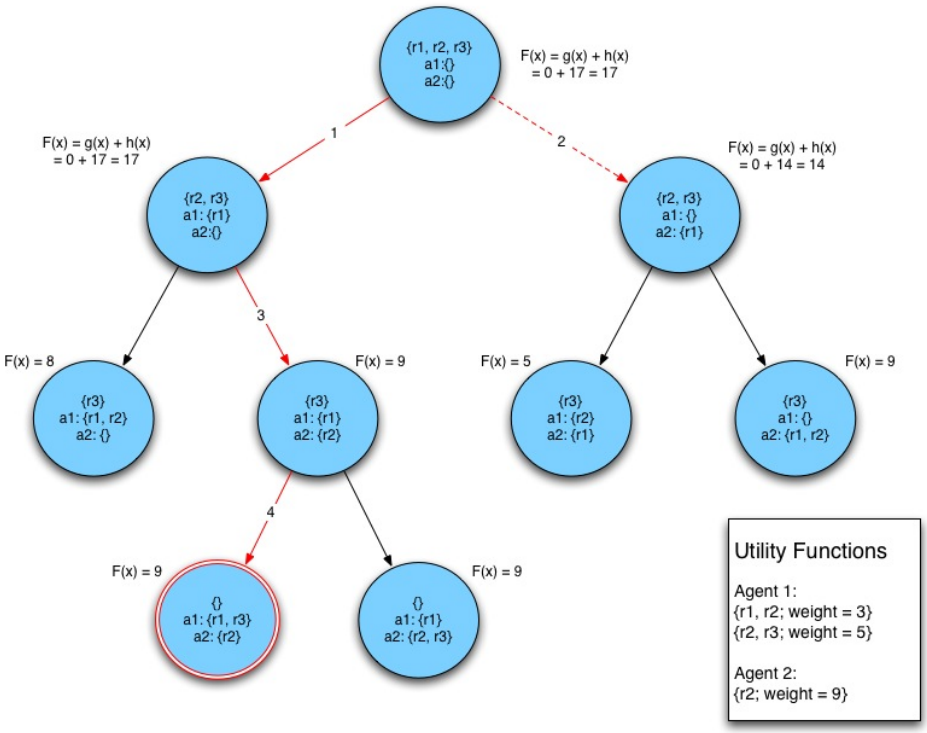


Figure 4.2: Optimal partial reallocation example

Problematic issues - Complexity

Using this approach several problems might occur. The first problem takes place in scenarios with a high resource to agent ratio where the agents only have preferences for a small amount of resources. In these situations there are a lot of resources that no one cares about. This means that, using this heuristic, allocating one of those resources will not give a more precise heuristic value. Since A* receives no additional information beyond the cost, its heuristic is crippled. This makes the algorithm a lot slower, and even

cause it to hang in some situations due to extremely large search trees (e.g. 2^{40} nodes).

One solution to this problem is to add a preprocessing phase in which all resources that are not wanted by anyone can be either randomly allocated or assigned to the agent who already owns them. Resources that are only wanted by one agent can automatically be assigned to this agent. In the current implementation the latter solution was chosen. (See Future work, section 7, for other possibilities). We leave it to the reader to check that this approach saves one step in the above example. After implementing this adaptation, the algorithm indeed turned out to show better performance. Experimental results of the use of this algorithm can be found in section 6.1.

Problematic issues - Global optimality

Although this algorithm does ensure an optimal *partial* reallocation, it can not guarantee a global optimal state after a certain amount of optimal partial reallocation deals. Even though this is a known problem, we illustrate this with an example:

Utility functions

```
Agent 1: {r5; weight=4}
Agent 2: {r1, r4, r7; weight=10}
Agent 3: {r9; weight=3}
```

Initial allocation

```
Agent 1: r1, r2, r3
Agent 2: r4, r5, r6
Agent 3: r7, r8, r9
```

Recall that all goals which contain resources that cannot be allocated to the agent will be removed. This means that if agent 1 and 2 enter optimal partial reallocation, agent 2's goal is removed, since it could never receive r7 (seeing as agent 3 is not participating in the deal). After removing this goal, neither agent 'prefers' r4. Then, if the algorithm is biased to choosing a certain path (say the 'allocate to agent 1' path), agent 2 will never receive r1. And thus the optimal state, where agent 2 receives r1, r4 and r7 is never reached.

Randomly reallocating the non-preferred resources may improve the situation. The chances of finding a better allocation will increase compared to the situation where they are not reallocated or allocated to one agent by default. However, despite this adjustment the algorithm will still be globally non-optimal in some cases.

4.6 Saving the results

Whichever algorithm you use to find deals, the results are always saved in the same, very simple format. A rough example follows:

```
Start experiment data
<?xml version="1.0"?>
<!-- The entire XML scenario file is included here for calculations
that you might want to use on the result. For the sake of readability,
this has been left out. -->

Money: true
Payment: LUPF
End experiment data
Start allocation sequence
Allocation 0
agent0
r0
0.0
0.0
agent1
r1
3.0
0.0
agent2
r2,r3
4.0
0.0

Allocation 5
agent0
r3
1.0
-1.0
agent1
r1
3.0
0.0
agent2
r2,r0
5.0
-1.0

End allocation sequence
```

So in short, first the entire scenario is inserted, so that parsers can calculate things like envyfreeness and such, which require the agents' utility functions. Then there are two lines stating whether money has been used, and what kind of payment function was in use.

After that the 'real' data starts: a long sequence of allocations. The sequence is started by the line "Start allocation sequence". What follows is a sequential list of allocations. Each allocation is specified by a list of data, 4 lines per agent. For each agent, the lines state:

1. the agent's `id`.
2. the agent's resources, separated by commas.
3. the agent's utility at this point.
4. the amount of money the agent has received (negative) or spent (positive).

Each allocation starts with a line saying `Allocation N` where N is the number of tries that it took to get this allocation (from the start of the running of the experiment). Each allocation ends with an empty line (note: other empty lines may occur if an agent owns no resources). The entire sequence of allocations ends with a line stating "End allocation sequence".

4.7 Graphing the Results

While the format of the results is somewhat human-readable, it is really hard to draw comparisons and get an idea of the big picture. For this purpose, we wrote a grapher that can read the output files and produce graphs of some of the more interesting correlations between the different kinds of data contained in the output. The grapher supports the following types of graphs:

- **Number of Resources per agent** This graph gives an overview of the changes in resource possession per allocation. It is meant mostly as a control graph, in which you can check whether an agent might have a very strange utility, or anything else might have gone wrong.
- **Social Welfare per Allocation** This graph shows the increase in the sum of the utility per allocation. It clearly shows the progression towards an optimum, and is very useful for determining the shortest amount of time needed to gain the maximum amount of welfare.
- **Try Count per Successful Allocation** The closer the system gets to an optimal state, the more difficult it becomes to find a possible deal. This graph plots the amount of successful allocations per the amount of deal attempts.

- **Maximum Envy** This graph plots the envy of the most envious agent per allocation.
- **Average Envy** This graph plots the average envy of all the agents the system. As the experiment progresses, you would like to see a decrease in envy as well as an increase in social welfare.
- **Number of Envious Agents per Allocation** Using this graph you can see how envy is distributed throughout the system.
- **Total Envy** A summation of all the envy of all the envious agents per allocation.

Additionally, the graphing system allows you to overlay graphs on top of each other, and graph the output of multiple data files. It also has an option as to whether or not money should be taken into account when calculating envy amongst agents.

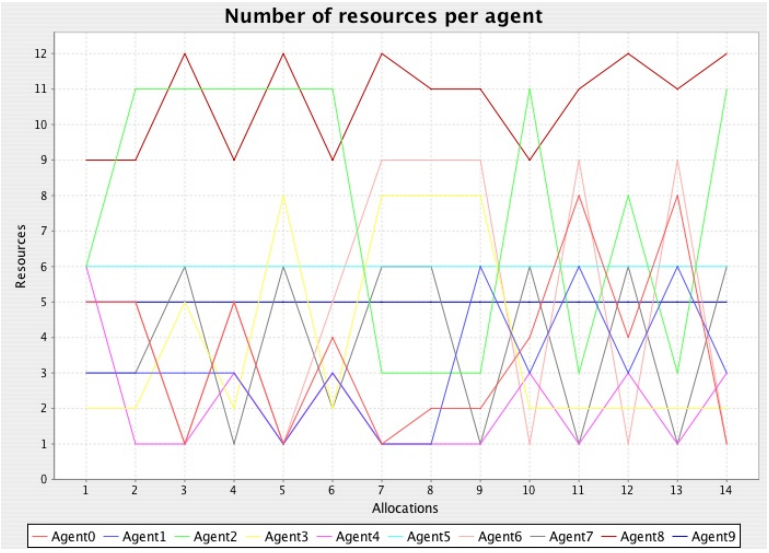


Figure 4.3: An example graph from the Grapher

Chapter 5

Usage

You can see an overview of the simulation platform in figure 4.1.

The simulation platform is provided as either source files, or `jars`. Depending on your preference, you can either compile yourself or use the `jar` files.

5.1 Compilation

You can compile all packages using the build files supplied. If you use a mac or linux, type `./build.sh`. If you use windows, use `build.bat`.

If after you use the source files you want to combine the source into jars again, you can use `deploy.sh` or `deploy.bat`.

If you would like to use jars instead, simply unzip the MARA.zip file, and everything should work with double-clicking.

5.2 The Scenario Generator

In the scenario generator (figure 5.1), you define the number of agents, the number of resources and the utility functions. Note that if you generate many more resources than agents, the complexity will go up exponentially. If each agent receives an average of r resources, and you're looking to redistribute the resources of 2 agents, you will have a complexity which approaches 2^{2r}.

The generator gives you the option to either generate a new random allocation of resources to agents, or use and existing scenario description file (.sdf).

To call the `scenario_generator`, double click the `ScenarioGenerator.jar` package in your favorite file browser.

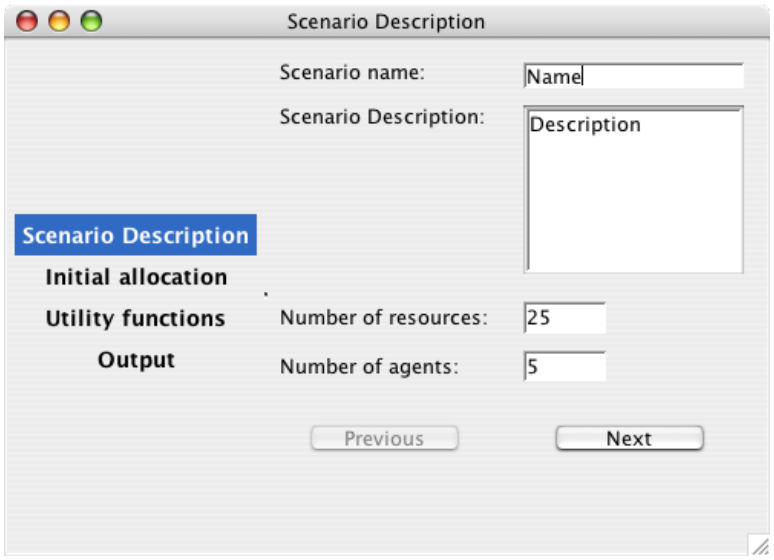


Figure 5.1: Scenario Description Interface

After generating the starting scenario, you can generate utility functions (figure 5.2). The first option is used to specify a distribution for k, where k is the maximum size of the set of resources referenced by one goal in an agent's utility function. A precise distribution of $k = 3$ will result in all agents having utility functions containing between 1 and 3 resources in each goal. An example is `((and ra rb), 5)`, where the set of resources $\{r_a, r_b\}$ receives a value of 5. One can also select a normal distribution for k. Thus, in general this setting defines a distribution, such that every agent is assigned a value k which is drawn from this distribution. In the preview graph you can see what the your distribution will look like using your chosen settings.

The second setting defines a *mapping function* as opposed to a *distribution*. With this parameter the so-called *formula length to count* mapping is specified. This function determines how many formulas will be generated of a certain length. If the selected function returns values that are larger than the maximum amount of possible rules, the maximum is returned instead. When a value below zero is returned, zero is returned. One can choose a linear percentage function, which returns a percentage of the possible amount of formulas of that length. The linear function does not specify the percentage but the exact amount of formulas. Finally a gaussian percentage function can be used as well. It should be noted that the total possible amount of formulas is usually very large, and therefore percentage functions will mostly return very high numbers. Of course this is not a problem if a small k is chosen.

Finally you will need to select the amount of variation in the weights of each formula. Should each set of resources be just as desirable? A common option here is a uniform distribution with both a lower and an upper bound. Do note that you cannot use negative weights, so your lower bound has to be at least 0.

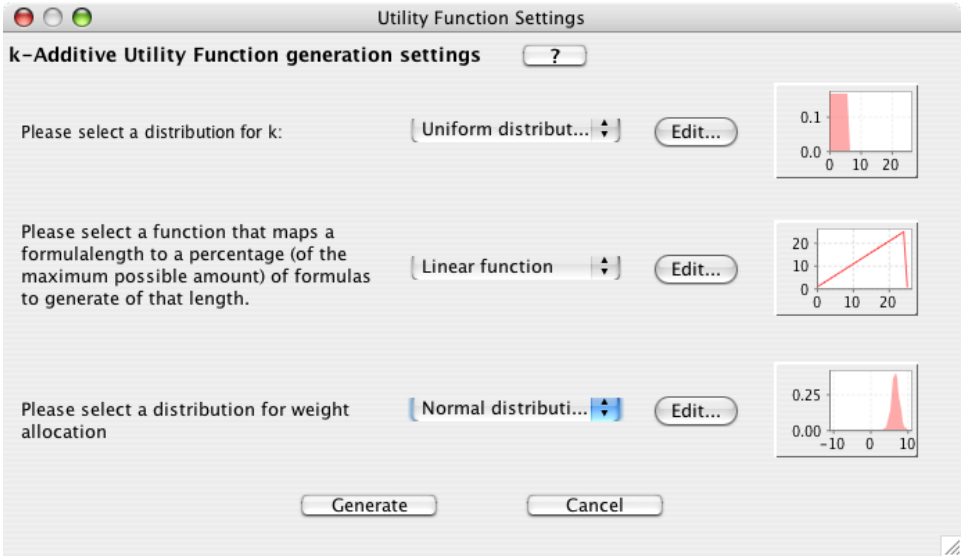


Figure 5.2: Utility Function Generation Interface

5.3 The Experiment Runner

After generating an adequate scenario, you can start to run experiments. In the experiment runner (figure 5.3), you can load your scenario, select the desired payment function and number of allocation tries, and finally run your experiment. You should choose a reasonable amount of allocation tries, for small experiments with less than 20 agents, the 10 000 default is a bit of an overkill. While running our own experiments, we used a rule of thumb that allocation tries was more or less the total amount of agents times the total amount of resources.

To call the experiment runner, double click on the `ExperimentRunning.jar` package.

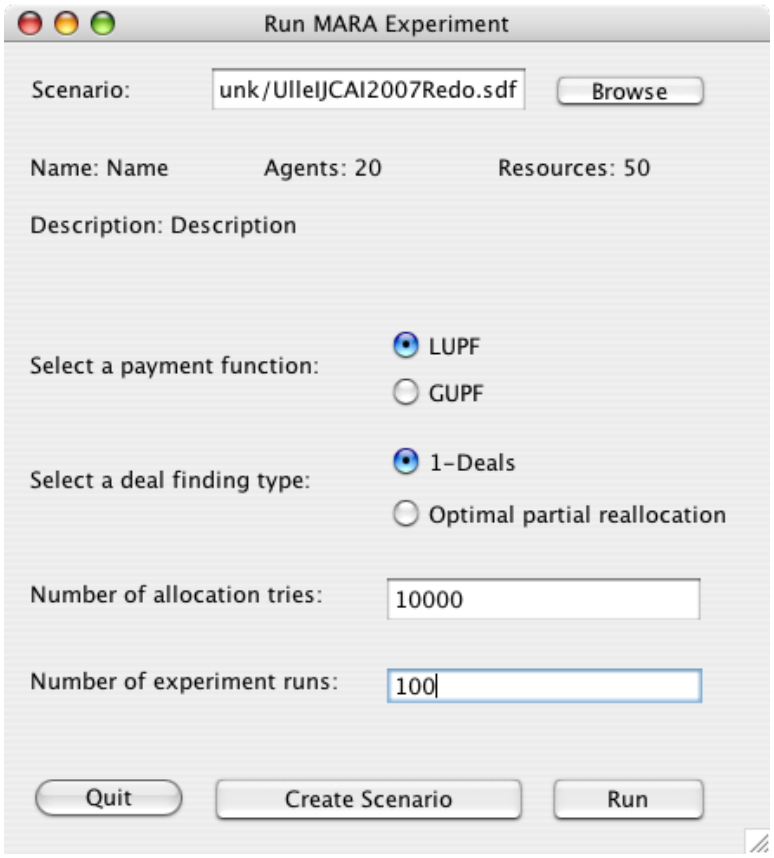


Figure 5.3: Experiment Running Interface

5.4 The Grapher

The experiment running will have generated a .csv file. You can read the output yourself, or you can half the Grapher (figure 5.4) plot it into a graph for you. You can read more about the graph options in 4.7.

To run the Grapher, double click on the `Grapher.jar` package.

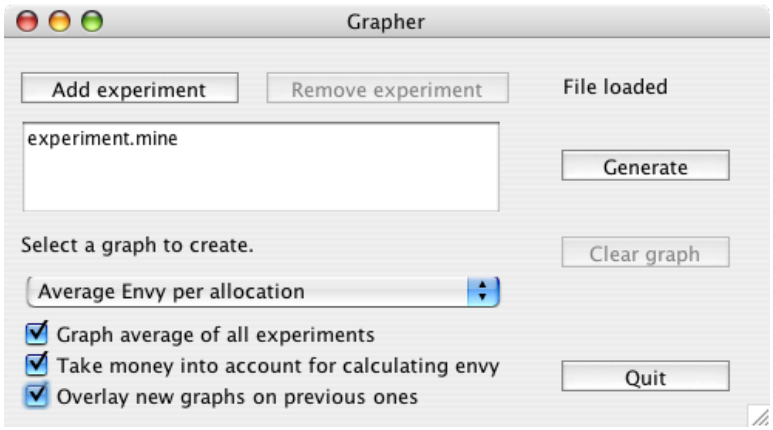


Figure 5.4: Graph Generation Interface

Chapter 6

Results

6.1 Optimal partial reallocation

Using the algorithm and heuristic as described in previous sections, some interesting results were found using the presented simulation platform.

6.1.1 Time-based performance

The quality of the defined heuristic can be assessed by comparing experiment runtimes using this heuristic with those using a trivial heuristic. The trivial heuristic highly overestimates; this is done by adding one million to the original g function. Comparing these to can be a useful indication whether or not the chosen heuristic is any good.

For the experiments a scenario was used with 16 agents and 80 resources. The utility functions had $k = 3$, so for each length up to k, the system generated 3 formulas. The weights were randomly assigned from a uniform distribution between 1 and 100. The initial allocation was generated randomly.

After running the experiment several times, the following averaged results were obtained.

Table 6.1: Experiment run times

Our heuristic	Trivial heuristic	Try count
20.3 s	21.8 s	200
40.5 s	60.9 s	400
82.0 s	686.3 s	800

From these results we can conclude that this heuristic definitely performs better than a trivial heuristic. In the beginning the deals are easy to find and thus even the trivial heuristic performs well. After more tries the deals

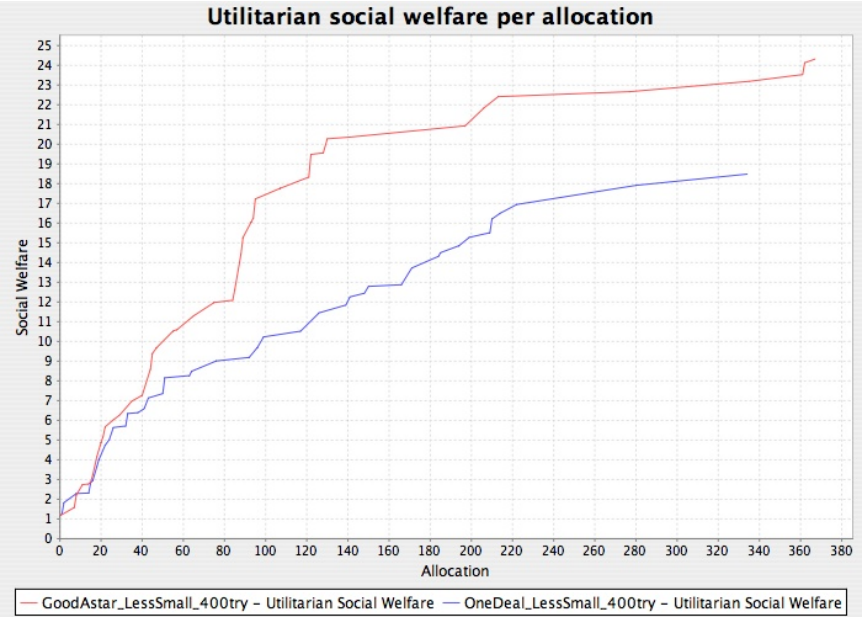
become more complicated. For our heuristic the time grows linear with the amount of tries. In contrast, the trivial heuristic grows much faster.

Of course these results do not show that the chosen heuristic is good. However, they definitely show that it is not the worst.

6.1.2 Optimal partial reallocation vs. 1-resource-deal

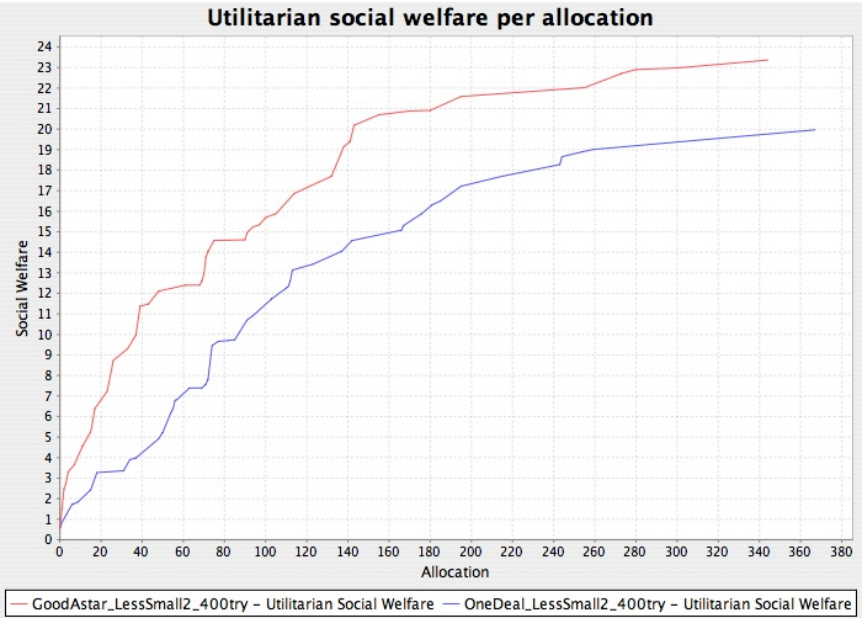
In addition to time performance, it is interesting to investigate the difference between 1-resource-deals and optimal partial reallocation when considering utilitarian social welfare. For this purpose experiments were run using the same scenario as in section 6.1.1. Additionally the same scenario with another intial allocation was used. This produced some mixed but interesting results, these are shown in Figures 6.1 and 6.2.

Figure 6.1: Social welfare increase for initial allocation 1



From these few experiments it seems as if the optimal reallocation increases faster than with the 1-resource-deals. Intuitively this also makes sense since the 1-resource-deal policy randomly selects two agents and a resource while optimal partial reallocation searches the state space far more directed. To support these intuition, more systematic experiments should be run with varying utility functions and ininitial allocations.

Figure 6.2: Social welfare increase for initial allocation 2



Chapter 7

Limitations and Future work

As has become clear in the previous chapters, the possiblities of the simulation platform are vast. Although this is an encouraging prospect, the downside is that we have currently not come to discover many of them. In this section some of the possible improvements are discussed.

7.1 Theoretical work

In this report several theoretic issues have been discussed. Most of these offer possibilities for further improvement. In addition some novel theoretic issues can usefully ameliorate the current implementation.

7.1.1 Optimal partial reallocation

First of all, a lot of work remains to be done regarding the deal finding policy *optimal partial reallocation* (See section 4.5.3). This mainly concerns the heuristic we used.

Negative weights

The chosen heuristic does not support negative weights: using negative weights renders the heuristic inadmissable. As a consequence the k-additive form is no longer fully expressive. It is clear that this is an important limitation, which future work can resolve by finding a better heuristic.

Global optimality

Furthermore, future work should adress the issue that the current heuristic is not always optimal in the global case. This can perhaps be done, by including information about resources owned by other agents that do not take

part in the optimal partial reallocation. Another idea is to value partially completed goals which contain resources that are owned by non partipating agents.

7.1.2 Utility function generation

In section 4.3.2 the issues related to utility function genertation were introduced. Several parameters were described of which a few are implemented in the simulation platform. Future work should examine which (other) parameters are in effect useful in typical distributed multiagent resource allocation situations.

Furthermore it is not at all clear what realistic values are for the current parameters. Questions as 'What distributions and functions are relevant?' and 'Which values are realistic?' should be adressed. Answers to these questions can lead to a more useful set of parameters for utility function generation.

7.1.3 Computing optimal outcomes

Regarding experiment support (Section 3.4) an important and difficult theoretical issue is the question if the optimal outcome of the scenario can be calculated. In, for example, a utilitarian experiment it can be useful to evaluate the used settings by comparing the eventual social welfare with the optimal social welfare. Such a comparison may be able support intuitions about whether using certain settings will ensure that the negotiation approaches the optimum. Future work should thus seek ways to calculate these optima for different types of social optima. It should be noted that it would presumably not be too hard to extend the A* algorithm and its heuristic described in section 4.5.3 to work for the set of all agents and reallocate all resources at once.

7.1.4 Random combination sequence generation

The implementation of the random generation of combination sequences, as described in Appendix A works well. However, it should be further examined if the algorithm performs significantly better than the naive algorithm in *practical* situations.

7.2 Implementation related work

Besides the theoretic work, also a lot of improvements remain to be explored regarding the implementation of the simulation platform.

Complexity issues

An improvement related to the problem described in 'Problematic issues - Complexity' (Section 4.5.3). The preprocessing phase proposed could be replaced by including a resource selection heuristic. If resources that do *not* occur in any agent's utility function are allocated, the function $f(x)$ does not change. Since the heuristic overestimates more in the beginning of the search tree (due to the fact that no goals can be trimmed) the value for $f(x)$ will be unrealistically high. If after a few of these resources a conflicting[1] resource is allocated, the heuristic becomes more realistic and thus smaller. A* will then proceed to expand all nodes in the tree above that allocate non-preferred resources, since the heuristic of those nodes is higher than the just expanded (more realistic) value.

To solve this, a heuristic could be used that selects a resource to allocate next. Intuitively it makes sense to allocate conflicting resources first. It might also be useful to take into account the weights of the goals in which they appear. Although these notions seem intuitively correct, future work should investigate whether this is indeed so.

7.2.1 Memory management

During the use of the current version of the platform, it turned out that generating utility functions can, using certain parameters, cause Java to run out of memory. This occurs when the amount of agents and resources get large in combination with many goals to be generated. It is not clear if the problem lies with the used algorithms or the programming work. This problem should be solved, so that a larger variety of scenarios can be generated.

7.2.2 Rationalities

Currently only *individual rationality* is implemented (See section 3.3.3). To provide support for more types of experiments, it is interesting to implement a wider range of agent rationalities. A good start are the *Pigou-Dalton transfers*.

7.2.3 Experiment automation

Finally it would prove very useful if the platform provides experiment automation. For example, the experimenter should be able to run the same experiment 100 times with different initial allocations, or with certain variations in utility functions. Using this, results could be averaged to provide more reliable outcomes.

[1] A conflicting resource is a resource that appears in a goal of both participating agents.

Chapter 8

Acknowledgements

The contents of this report are the results of an honours program project for BSc AI students on Multiagent Resource Allocation at the University of Amsterdam, as led by Ulle Endriss.

We would very much like to thank Ulle Endriss for all his guidance and support, Bastiaan de Groot for showing us his Python solutions for combinatorial auctions and Marco Wessel for hosting our SVN repository on his server.

Bibliography

[1] M. Andersson and T. Sandholm. Contract types for satisficing task allocation: II experimental results. *AAAI Spring Symposium Series: Satisficing Models*, pages 1–7, 1998.

[2] A.H. Bond and L. Gasser. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Francisco, 1988.

[3] Y. Chevaleyre, U. Endriss, S. Estivie, and N. Maudet. Welfare engineering in practice: On the variety of multiagent resource allocation problems. *Engineering Societies in the Agents World V*, pages 335–347, 2005.

[4] Y. Chevaleyre, U. Endriss, S. Estivie, and N. Maudet. Reaching envy-free states in distributed negotiation settings. *Proceedings of IJCAI-2007*, 2007, 2007.

[5] Y. Chevaleyre et al. *Issues in Multiagent Resource Allocation*. Institute for Logic, Language and Computation (ILLC), University of Amsterdam, 2005.

[6] U. Endriss, N. Maudet, F. Sadri, and F. Toni. On optimal outcomes of negotiations over resources. *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 177–184, 2003.

[7] M. Huhns and M. Singh. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1997.

[8] Geert Jonker, John-Jules Ch. Meyer, and Frank Dignum. Efficiency and fairness in air traffic control. In *BNAIC*, pages 151–157, 2005.

[9] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Conference on Electronic Commerce*, pages 66–76, 2000.

[10] M. Minsky. *The Society of Mind*. Simon and Schuster, New York, 1986.

[11] J.S. Rosenschein and G. Zlotkin. *Rules of Encouter, Designing Conventions for Automated Negotiation Among Computers*. MIT Press, Cambridge MA, 1994.

[12] T. Sandholm and S. Suri. Bob: improved winner determination in combinatorial auctions and generalizations. *Artif. Intell.*, 145(1-2):33–58, 2003.

[13] T. W. Sandholm. Contract types for satisficing task allocation: I theoretical results. *Proc. AAAI Spring Symposium: Satisficing Models*, 1998.

[14] T.W. Sandholm. Optimal winner determination algorithms. In Yoav Shoham Peter Cramton and Richard Steinberg, editors, *Combinatorial Auctions*. MIT Press, 2006.

[15] R. G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. In *Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

Appendix A

Random combination sequence generation

A.1 Introduction

In this appendix the issue of random combination sequence generation will be discussed. This is an issue that needs to be resolved when generating UFs. An algorithm that solves this is useful when, for example, a formula for a k-additive goal is created. These formulas consist of a conjunction of a certain amount of resources. Since conjunction is a commutative and associative operation, the order in which these resources appear in the formula does not affect the semantics of the formula. Thus a *combination* of resources is sought. Moreover, since a simulation should have random aspects, these combinations need to be generated randomly.

A.2 Problem description

As this problem appears in many forms, a more abstract problem description will be given here.

Let $n, k \in \mathbb{N}$ and let O be a set of objects with $|O| = n$
Then $C = \{X \mid X \in 2^O \wedge |X| = k\}$ are all combinations of k objects from O.

Generating a random combination from C is not difficult. The problem of random combination sequence generation, however, amounts to picking more than one distinct combinations randomly from C. If C were completely known this was not a difficult problem. However computing C is computationally intensive, since finding all combinations is a combinatorial

problem. Consequently the core problem amounts to finding an efficient way of generating a sequence S of m distinct combinations *randomly*, while:

$$S \subseteq C \wedge |S| = m$$

A.3 Problem illustration

In a naive approach, one might consider randomly picking a number $x_1 \in \{1, \ldots, n\}$, then choosing $x_2 \in \{1, \ldots, n\}$. In case the newly generated number is equal to a previously generated nubmer, discard it and try again. This is repeated until a complete combination $x_1 \ldots x_k$ is generated. This approach clearly creates a random combination and, when repeated, will generate all possible combinations.

When $k \ll n$, this algorithm runs quite efficiently. However, when this is not the case the algorithm is rather inefficient. In addition, the algorithm might theoretically not terminate at all. More importantly, a problem arises when we want to generate a random *sequence* of distinct combinations. In the naive approach we would sequentially generate several combinations and discard those that were previously generated. Since this applies the same concept as the above algorithm, the problem is only aggravated. To elucidate this point, consider the following example.

In some situation 9 999 out of 10 000 possible combinations need to be randomly generated. When generating the final (9999th) combination, the probability that a combination is generated that was previously generated is $\frac{9998}{10000}$. There is thus a high probability that the generation has to be repeated many times before the final combination is generated.

Taking these problems into account, it is clear that a better algorithm needs to be designed in order to improve efficiency.

A.4 Relevance

As described in the introduction, a solution to this problem can be applied directly in the process of utility function generation. Once the UF generator has determined the amount of formulas to generate of a certain length, this amount of combinations of resources has to be generated randomly. This is exactly the problem as described in the previous section.

A.5 Approach

The problem can be solved by reducing it to a simpler one that already has an efficient solution. This is the random selection of m distinct integers from

an interval [1, l]. The algorithm is as follows:

Algorithm A.5.1: SELECTDISTINCTINTEGERS($upper, m$)

$L \leftarrow \{1, \ldots, upper\}$
for $i \leftarrow 1$ **to** m
do $\begin{cases} pos \leftarrow \text{random number between 0 and } l - i \\ res.add(L_{pos}) \\ L.remove(pos) \end{cases}$
return (res)

This efficiently (in $O(m)$) generates a random sequence of m distinct elements from the list. The problem at hand can be solved using the same approach, once each random combination can be identified with a number. Consider the list of combinations in Table A.1.

ID	x_1	x_2	x_3
1	1	2	3
2	1	2	4
3	1	2	5
4	1	3	4
5	1	3	5
6	1	4	5
7	2	3	4
8	2	3	5
9	2	4	5
10	3	4	5

Table A.1: All combinations of 3 elements from a total of 5 elements

If there is an (efficient) way to compute $x_1x_2x_3$ from a given ID, the problem is solved. Using Algorithm A.5.1, m distinct ID's between 1 and 10 can be generated, and from this the combinations can be computed.

A.6 Algorithm

Obviously, having only the ID is not sufficient for computing a combination with that ID. The algorithm will also require n, the total number of available elements and k, the number of elements we want to choose (5 resp. 3 in the example of Table A.1). Given these three parameters the algorithm should compute the corresponding combination.

A.6.1 Buckets

Notice the interesting structure that the list of combinations exhibits. The first column contains all numbers 1 to $n - k$, the higher the number, the lower the number of repetitions of that number. Using this structure we can divide the list into several *buckets*. A bucket consists of all adjacent numbers within a column that have the same value. Figure A.1 gives a depiction of this division.

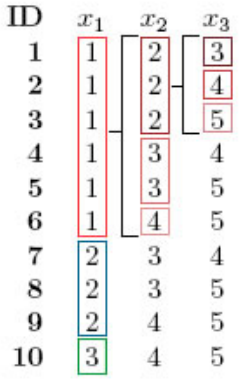


Figure A.1: Conceptual bucket division of combinations

Given the correct bucket of the first column, an equivalent problem arises in the second column. Again a sequence of numbers appears decreasing in the amount of repetitions. To quantify this structure, observe the following.

Define B_L as the bucket at which we arrive by following the sequence of sub-buckets according to the contents of list L. For example:
$B_{[1,2]}$ indicates the 2nd sub-bucket of the 1st bucket. That is, the bucket containing $x_2 = 3$ for IDs 4 and 5 in Figure A.1. To determine in which sub-buckets a certain ID belongs, it is necessary to be able to calculate the size of a bucket:

$$|B_L| \stackrel{def}{=} \begin{pmatrix} n - \sum_{i=1}^{|L|} L_i \\ k - |L| \end{pmatrix} \text{ with } \begin{pmatrix} a \\ b \end{pmatrix} = \frac{a!}{b!(a-b)!}$$

This formula can be understood more easily when justified as follows. When picking k elements out of a total of n, the total number of combinations is $\begin{pmatrix} n \\ k \end{pmatrix}$. The size of a bucket depends on the amount of combinations that can be made by filling the remaining positions from the remaining amount of available elements. When considering a bucket in the i-th column, only

$k - i$ slots remain that need to be filled. Here i corresponds to $|L|$. Furthermore, the amount of elements that remain to choose from depends on the buckets that were chosen in previous steps. Choosing for example the second bucket implies that the element in the first bucket and that in the second bucket can no longer be used the remaing part of the combination. This follows from the fact that the elements in the combination are listed in increasing order.

Using this definition, we can now calculate the size of the example that was given earlier.

$$|B_{[1,2]}| = \begin{pmatrix} 5-3 \\ 3-2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} = 2$$

The final step that remains is computing x_i given L. This can be calculated as follows:

$$x_i = \sum_{j=1}^{i} L_j$$

A.6.2 Pseudo-code

Using this information the complete algorithm can now be constructed. By comparing the given ID with the bucket sizes in the first column, the first bucket can be determined. This process is repeated to compute L. Finally the combination is computed from L. Below the pseudo-code for the algorithm is listed.

Algorithm A.6.1: COMPUTECOMBINATION(ID, n, k)

$bucketEnd \leftarrow 0$
for $slot \leftarrow 0$ **to** $k - 1$

do
- $L.add(1)$
- **while** $ID > bucketEnd + |B_L|$
  - **do**
    - $bucketEnd \leftarrow bucketEnd + |B_L|$
    - $L_{slot} \leftarrow L_{slot} + 1$
- $Comb.add(\sum_{j=1}^{slot} L_j)$

return $(Comb)$