

Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Csaba Tóth

Simulace spolupracujících dělníků s optimalizací
Simulation of cooperating workers with optimization

Department of Theoretical Computer
Science and Mathematical Logic

Supervisor: RNDr. David Kronus, Ph.D.
Department of Theoretical Computer Science and
Mathematical Logic

Study program: Computer Science,
General Computer Science (IOI)

2009

I would like to give thanks to my supervisor RNDr. David Kronus, Ph.D., for his time and advices, to Timo Bingmann, whose compiler tool integration example I used as starting point of the script language, to my family and friends who helped my work with great ideas.

I hereby certify that I wrote the thesis by myself, using only the referenced sources. I agree with lending the thesis.

In Prague 2009-05-28

Csaba Tóth

Contents

Chapter 1.	Introduction.....	5
1.01	Motivation	5
1.02	The Aim of the work	5
Chapter 2.	Multiagent systems.....	7
2.01	Basic definition	7
2.02	The properties of the environment.....	7
2.03	Agents and intelligence	8
2.04	Formal definition of agents.....	9
2.05	Types of agents	11
2.06	Agents interacting	12
2.07	Communication.....	12
2.08	Cooperation.....	13
Chapter 3.	Selected approach.....	14
3.01	The environment	14
3.02	The workers	15
3.03	The goals of the simulation	16
Chapter 4.	The whouse program	17
4.01	Introduction.....	17
4.02	Elements of the simulation	17
4.03	The work cycles	19
4.04	The graphic interface.....	20
4.05	The dialog windows	22
4.06	The input.....	24
4.07	The output.....	24
4.08	The built-in script language.....	25
4.09	The built-in functions	26
4.10	Main script specific functions.....	29
4.11	The worker specific functions.....	30
4.12	Setting up the program	34
4.13	Possible improvements in the future	35
Chapter 5.	Conclusion	37
	Bibliography.....	39
	Appendix.....	40

Název práce: Simulace spolupracujících dělníků
s optimalizací
Autor: Csaba Tóth
Katedra (ústav): Katedra teoretické informatiky a matematické
logiky
Vedoucí bakalářské práce: RNDr. David Kronus, Ph.D.
e-mail vedoucího: david.kronus@mff.cuni.cz

Abstrakt: V předložené práci studujeme problém spolupráce dělníků při ukládání a vyzvedávání krabic ve skladu. Dělníci spolu komunikují, ale mají pouze omezenou paměť na informace o mapě skladu a na komunikaci s ostatními. Jejich úkolem je snažit se optimalizovat některý zvolený parametr jejich práce, např. časovou odezvu při vyzvedávání krabice ze skladu. Prostředkem optimalizace je uložení a reorganizace krabic ve skladu a vhodná komunikace mezi dělníky.

Klíčová slova: simulace, optimalizační nástroj, multiagentní systémy, spolupráce, komunikace

Title: Simulation of cooperating workers with
optimization
Author: Csaba Tóth
Department: Department of Theoretical Computer Science
and Mathematical Logic
Supervisor: RNDr. David Kronus, Ph.D.
Supervisor's e-mail address: david.kronus@mff.cuni.cz

Abstract: In the present work we study the problem of cooperating workers placing and finding boxes in a storehouse. Workers communicate but have limited memory for information about the map of storehouse and for communication with other workers. Their task is to optimize with respect to some criteria, e.g. response time of finding a box in a storehouse, by means of placing and reorganizing boxes and appropriate communication among workers.

Keywords: simulation, optimization tool, multiagent system, cooperation, communication

Chapter 1. Introduction

1.01 *Motivation*

Nowadays quick and efficient temporal storage is an important aspect for many major companies from high tech parking houses to delivery firms. The key qualities of such storage facilities are short insert and withdrawal times. Further improvements such as space optimization (by eliminating ramps and conventional elevators), modular design, and energy efficiency (by the reduction of the need of lighting, air conditioning) can be reached by replacing human work force with artificial agents.

The general approach to these automated systems is through centralization. A main organizing entity controls all the knowledge about the warehouse structure, its content and its workers.

- The system has to know not only the whole usable storage capacity, but also its structure so it could assign the correct shelf, to each new item.
- The system has to know the actual content of the warehouse (and its position in the structure). In many cases the system also has to remember the expiration time of some types of goods and serve out the older items first, than periodically throw out the already expired goods.
- The system has to drive its working elements.

The weak points of the above described system are its rigidity and the dependency on its central database. To at least partially eliminate these problems it would be possible to divide the system into several, simpler, hierarchic subsystems.

A further improvement of a multiagent implementation is its robustness and reliability. There are no irreplaceable elements, the system can without effort undergo a “graceful degradation” when some of the working elements fail. A similar central system always has a bottleneck which can be painful when an error or necessary update of the main entity occurs.

1.02 *The Aim of the work*

As my bachelor thesis I chose to create an observation tool to help research in the field of storage facilities with distributed commanding system where the “intelligence” is moved from the central control entity to the end nodes – in our case to the working agents.

In these storage facilities the main database could be divided in a way that the central entity has to know only what is stored in the warehouse, but the exact positions of the items and the structure of the available storage place can be completely left for the working agents or other hierarchic sub-elements.

These working agents should be able to communicate with each other; what enables them to exchange information and also organize themselves into the mentioned hierarchic workgroups.

Chapter 2. Multiagent systems

2.01 *Basic definition*

From the definition of *Michael Wooldridge* [1]: *multiagent system* is one that consists of a number of agents which interact with one another, typically by exchanging messages and/or orders through some sort of infrastructure in their *environment*.

Based on *Michael Georgeff's* definition an *agent* of this system is also a computer system that is capable of autonomous actions on behalf of its or owner. In other words an agent is an independent element of the environment that can figure out for itself what it needs to do in order to satisfy its design objectives, rather than having to be told explicitly what to do in any given moment.

2.02 *The properties of the environment*

The agents act in an *environment* defined by the following characteristics:

- *Accessible / Inaccessible*
- *Deterministic / Non-deterministic*
- *Static / Dynamic*
- *Discrete / Continuous*

Accessible / Inaccessible:

If the selected environment is accessible, that means the agent can obtain complete and accurate information of the environment at any given time, all the relevant information are at it's disposal. In most real-world situations however this would be impossible, so the commonly used environments are inaccessible.

Deterministic / Non-deterministic:

In a deterministic environment all operations can be taken as atomic operations. All actions the agent executes have a guaranteed effect. This reduces the need of control and the required complexity of the agents; however the deterministic environments are just as rare in the real world as completely accessible ones.

An interesting remark made by Russel and Norwig in 1995 points it out that after an environment reaches sufficient complexity it doesn't matter if it's actually still deterministic, in practice it should be regarded as non-

deterministic. Their consequence is that real world environments should always be regarded as non-deterministic in the agent's perspective.

Static / Dynamic:

A static environment can be assumed to remain unchanged except by the performance of actions by the agent (a single selected one). A dynamic environment may have all sorts of entities operating in it while also changing it in ways beyond the agent's control. The physical world is a highly dynamic environment.

Discrete / Continuous:

In this work I use the definition of discrete environment as an environment where there are fixed finite number of possible perceptions, actions and consequences of those actions.

Non-deterministic, non accessible environments can be taken as if there would be a sphere of influence around every agent where they have at least partial control over their surrounding. Everything out of this sphere remains unknown and unreachable for the agent.

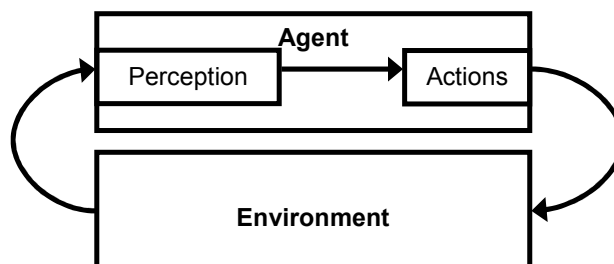
2.03 Agents and intelligence

Agents are required to be able to make “the right” decisions on their own. This ability is generally called intelligence, and can be characterized by several separate capabilities. The following list of capabilities was suggested by *Wooldridge and Jennings* in 1995:

- *Reactivity*
- *Proactiveness*
- *Social ability*

Reactivity means agents are able to perceive their environment and respond to its changes.

Proactiveness means the agent's behavior is directed



2-1: The scheme of a reactive agent

by its goals. In addition to reactions; it initiates own actions in order to satisfy its design objectives.

Social ability means agents are capable of interacting with other agents (and possibly other elements of the environment) in order to satisfy their design objectives.

2.04 Formal definition of agents

Abstract view of agents taken nearly by the word from the book of Michael Wooldridge [1]:

Let us assume that the environment may be in any of a finite set E of discrete, instantaneous states:

$$E = \{e, e^1, \dots\}$$

Whether the environment is discrete or continuous is of little importance, because any continuous environment can be modeled by a discrete environment to any desired degree of accuracy.

Agents are assumed to have a repertoire of possible actions available to them, which transform the state of the environment. Let

$$Ac = \{\alpha, \alpha^1, \dots\}$$

Be finite set of actions. Then the basic model of agents interacting with their environment is as follows:

The environment starts in some state, and the agent begins by choosing an action to perform on that state. As a result of this action, the environment can respond with a number of possible states. However, only one state will actually result - though of course, the agent does not know in advance which it will be. On the basis of his second state, the agent again chooses an action to perform. The environment responds with one of a set of possible states, the agent then chooses another action, and so on.

A run, r , of an agent in an environment is thus a sequence of interleaved environment states and actions:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \dots \xrightarrow{\alpha_{u-1}} e_u$$

Let

- \mathcal{R} be the set of all such possible finite sequences (over E and Ac);

- \mathcal{R}^{Ac} be the subset of these that end with an action; and
- \mathcal{R}^E be the subset of these that end with an environment state.

We will use r, r^1, \dots to stand for members of \mathcal{R} .

In order to represent the effect that an agent's actions have on an environment, we introduce a state transformer function (from Fagin, 1995):

$$\tau : \mathcal{R}^{Ac} \rightarrow \wp(E)$$

Thus a state transformer function maps a run to a set of possible environment states – those that could result from performing the action.

There are two important points to note about this definition. First, environments are assumed to be history dependent. In other words, the next state of an environment is not solely determined by the action performed by the agent and the current state of the environment. The actions made earlier by the agent also play a part in determining the current state. Second, note that this definition allows for non-determinism in the environment. There is thus uncertainty about the result of performing an action in some state.

If $\tau(r) = \emptyset$ (where r is assumed to end with an action), then there are no possible successor states to r . In this case, we say that the system has ended its run. We will also assume that all runs eventually terminate.

Formally, we say an environment Env is a triple $Env = \langle E, e_0, \tau \rangle$ where E is a set of environment states, $e_0 \in E$ is an initial state, and T is a state transformer function.

We now need to introduce a model of the agents that inhabit systems. We model agents as functions which map runs (assumed to end with an environment state) to actions (from Russell and Subramanian):

$$Ag : \mathcal{R}^E \rightarrow Ac$$

Thus an agent makes a decision about what action to perform based on the history of the system that it has witnessed to date.

While environments are implicitly non-deterministic, agents are assumed to be deterministic.

Let Ag be the set of all agents. We say a system is a pair containing an agent and an environment. Any system will have associated with it a set of possible runs; we denote the set of runs of agent Ag in environment Env by $\mathcal{R}(Ag, Env)$. For simplicity, we will assume that $\mathcal{R}(Ag, Env)$ contains only terminated runs, i.e. runs r such that r has no possible successor states: $\tau(r) = \emptyset$. (We will thus not consider infinite runs.) Formally, a sequence

$$(e_0, \alpha_0, e_1, \alpha_1, e_2, \dots)$$

represents a run of an agent Ag in environment Env if

1. e_0 is the initial state of Env ;
2. $\alpha_0 = Ag(e_0)$; and
3. for $u > 0$,

$$e_u \in \tau((e_0, \alpha_0, \dots, \alpha_{u-1})), \text{ where } \alpha_u \in Ag((e_0, \alpha_0, \dots, \alpha_{u-1})).$$

Two agents Ag_1 and Ag_2 are said to be behaviorally equivalent with respect to environment Env if and only if

$$\mathfrak{R}(Ag_1, Env) = \mathfrak{R}(Ag_2, Env),$$

And simply behaviorally equivalent with respect to all environments.

2.05 Types of agents

Purely reactive agents:

One of the most basic agent designs is the *purely reactive agent* – These agents make decisions purely on the basis of the actual state of the environment. They make no reference at all to the past. This way of acting is called *tropism*; it was formulated by Genesereth and Nilsson in 1987.

This kind of entities are a used in the most basic roles of our world such as thermostats. Their main and probably only advantage is they require no internal memory whatsoever.

Formally they can be simply described by the function:

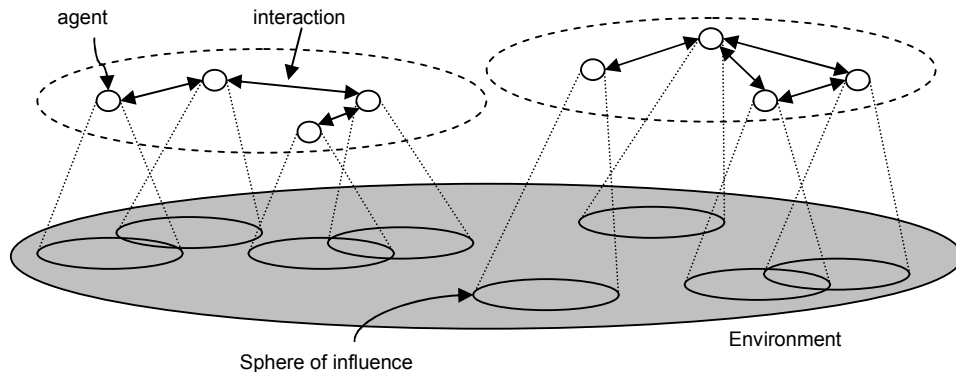
$$Ag : E \rightarrow Ac$$

Agents with state:

The more sophisticated agents are called *agents with state*. These agents have some kind of internal data structure holding information about the environment's state and history. The simplest construction of such agents is basically *finite state machines*.

2.06 Agents interacting

A typical multiagent system, as one would expect contains a number of agents which interact with each other through communication. Different agents have different spheres of influence which may coincide in some cases. These relations may give rise to dependency relationships between the agents.



2-1: Typical structure of a multiagent system based on an image from [1]

When agents get into contact with each other their behavior based on their dependency on each other can be described by one of these categories (defined by Sichman and Demazeau, 1995):

- **Independence:** There is no dependency between the agents.
- **Unilateral** dependence: One agent depends on the other agent, but not vice versa.
- **Mutual** dependence: Both agents depend on each other with respect to the same goal.
- **Reciprocal** dependence: The first agent depends on the other for some goal, while the second also depends on the first for some other goal (the goals can be the same, but that's not necessary, which means that mutual dependence is a subcategory of reciprocal dependence).

These categories can be further refined by considering the fact that these relations may be **locally believed**, or **mutually believed**.

2.07 Communication

Communication basically means exchange of information. But in agent related context it can be even more powerful. Agents can request others to perform some task for them. An autonomous agent has control over both its state and its behavior. It can not be taken granted any of the nearby entities will execute a given action just because another agent wants it to but if their objectives allow it they can be more successful in cooperation.

In most environments agents can neither force other agents to perform actions, nor directly modify data on the internal state of other agents. What they can do is perform actions (communicate) in an attempt to influence other agents appropriately. This fact is further explained in the *speech act theory*.

2.08 Cooperation

From the view of cooperation we can categorize the agents based on their interests:

- *Self interested agents*
- *agents working for a common goal*

The **self interested** agents inhabit the same environment, but have their very unique goals what in many cases makes them competitors to each other. To description of their relations there are used dominance strategies and the Nash Equilibriums.

But in my present work there are more interesting the **agents working for a common goal**. Historically most work on cooperative problem solving has made this benevolence assumption. This assumes that the agents can rely on the information and requests got from each other. They will help each other even if that means that one or more agents must “suffer” in order to do so. This design also brings the fact that agents are simpler in design than the self interested ones.

Chapter 3. Selected approach

3.01 *The environment*



3-1: Medicine distribution warehouse, Bratislava

The program tries to be a simulation of a real warehouse environment. This means, I had to go through the properties of a real warehouse and select the ones that I taught to be relevant, build them into the simulation, while ignore, or throw away other issues I didn't find relevant enough.

In general the real world systems are *dynamic*, *non-deterministic*, *inaccessible*, and *continuous*.

Dynamic and **non-deterministic**, because there are most probably multiple agents working with the same set of objects near each other. Their design objectives may differ and in their work they are likely to interfere with each other. There are also unforeseen errors, broke-downs that make the real world so complex.

The **inaccessibility** of the environment is one of the main points of my work. As I mentioned before the traditional approach of storage facilities is through a central database system. This, if built and maintained properly, provides a completely accessible environment for the decision making entity. The whole point of this program is to decentralize the system and whit that done remove the need of accessibility from the storage facility.

The real world is clearly a continuous environment, however in the world of computers we are usually provided to work with a discrete environment.

Because the above facts I chose to put my simulation in a *dynamic*, *non-deterministic*, *inaccessible*, but *discrete environment*.

The warehouse can be extended in the runtime through the script language inserting new container positions and waypoints to be explored by the agents – their environment is changing beyond their power.

There are also multiple workers in the simulation, they can't be sure what their operations will result when interacting with the same set of containers.

The inaccessibility is represented in a way that the agents see only the single object where they are actually standing and they know what waypoints can be reached from their current position. All other position information they can only remember.

3.02 *The workers*

The agents of my simulation on default are expected to behave as *agents working for a common goal*. They should cooperate to keep up the work in the storage facility. This fact can be easily changed by the user with overwriting the script of one or more agents thus with that simulating the actions of a faulty or misbehaving agent. It can be easily done by the tools of the program, but it's not part of the program's original goals.

The type of agent dependency I used in my working agents is reciprocal. They have (most probably) different sub-goals – to position or retrieve different packets – but they share information with each other about the packet's whereabouts.

The agent communication I realized through a messaging system. Originally there were two types of messages in the program – a specified message holding information about the locations of the known packets, and a more general text message type. Later I used only the general messages which gives more freedom to the agents in their communication however increases the load on their text processing abilities.

An interesting concept of the multiagent systems is the multi-layered organization of the agents. In my simulations I expected all agents to be equal. This means that no agent takes orders from others, neither gives any. They simply send informational messages to each other. However the *whouse* program is fully able to operate multi layered worker infrastructure – it can be done simply through the message system.

I choose to leave out this approach from my experiments because it partially reintroduces one of the main issues to the warehouse world I wanted to eliminate in the first place – the existence of a bottleneck in the system. If there are privileged elements between the workers who are organizing the others, their loss, or their misbehavior can be nearly as crucial as the failure of

the single central entity. We could solve the above issue by deciding that the agents at their birth all will be equal than later by some way of selection do they occupy their positions. This all could be done, and much ore but I think this would take us quite far from the original goal of my project.

Agent learning is another aspect I choose to leave out from my simulation. I created a program to be a simulation tool for human programmers who are interested in testing their own ideas rather than letting the computer to think out a suitable solution. The user has the tools to alter or replace the driving scripts of each worker, or even the driving script of the simulation itself at any point of his/her experiment. I think this gives the enough flexibility to the program, in fact I think it possibly even compromises the results of the simulation giving the possibility of human intervention.

3.03 *The goals of the simulation*

The task of the workers is to successfully receive packets from the exchange points and position them into the free containers of the warehouse. When a return request arrives the workers should as quickly as possible return with the packet to an exchange points. The lengths of these complex operations are counted in work cycles.

The simulation is guided by a main script (or directly by the user through the graphic interface) which has control over all the important data of the simulation:

- How many workers should be in the warehouse
- How many packets, and when should be inserted to the warehouse
- When should be the Packets returned from the warehouse
- How and when should be the warehouse extended
- How should be the control output created.

Using these options the user has the ability to simulate a wide variety of warehouse situations, nearly everything I could think of.

The program itself in general does not support batch execution – multiple runs of different simulations, however on a selected warehouse map the main script of the simulation can be set in a way that it would replay the same scenario (or any different scenarios) multiple times. The shortcomings of this “batch execution” are that the main script has no control over the memories and in general over the inner states and position of the workers so there is no way to reset them to the same position and state between the different runs of a given scenario.

Chapter 4. The whouse program

4.01 *Introduction*

The program is a multiplatform simulation tool representing the work of independent agents in their work environment. The selected type of environment is a warehouse where the workers' collective goal is to make the warehouse functional – store packets and on request retrieve them from the warehouse.

Each agent is driven by its own driving script written in the program's script language that can be modified by the user. The scripts are intended to serve one or more of the following sub-goals:

- receive outer requests
- receive packets
- find place for packets
- reorganize packets
- retrieve packets

The user sets the environment through a properly formed input file, than sets the precise details using the windowed environment and the scripts of the working elements.

4.02 *Elements of the simulation*

The warehouse structure is represented as a graph. Two types of elements build up the graph – the *corners* and *corridors*.



4-1 Distribution
warehouse Fenix

The *corridors* also represent the storage area of the warehouse. Each section of a corridor can be taken as one storage place. In the program these objects are named as **ContainerLine** (Cline).

The *corners* connect two or more corridors, or terminate a single corridor. The agents use these positions as *waypoints* to identify their position. Their name in the program is **Waypoint** and all have their unique identification number.

Specialized *waypoints* are the *exchange points* (**ExchangePoint**). These positions, besides serving as ordinary waypoints, also double as connection to

the outer world. Workers, packets, and packet requests enter the simulation through these objects. At least one *ExchangePoint* is needed in each warehouse. If there are more of them they are all expected to be equal, there exists no main *ExchangePoint*.

The goods stored in the simulation are represented by **packets**. Packets have their own unique *identifier* number, a *type*, *weight* and string *content* (the content is irrelevant from the perspective of the simulation).

The *weight* determines which *worker* is able to carry it, and the *type* is defined to help the agents organizing the *packets*.

The working agents of the program are called the **Workers**. The workers in the actual version are driven by their *script* that is freely modifiable by the user. Another important characteristic of the agents are their *carrying ability* which determines the weight they can move. They have a *backpack* which holds the packets the worker is actually carrying. Each of these packages is put into one of three possible pockets of the backpack or with other words is labeled with one of three labels. These labels (packets) are meant to be dedicated to one of the main actions the worker can have connected to the packets:

- “*in*”, meaning the agent brought it in to the warehouse,
- “*out*” meaning the agent is returning with the packet to an exchange point,
- “*move*” meaning the agent is only relocating the position of the packet in the warehouse.

The memory of the workers is represented in two entities. The first, simpler one is the *request memory*. This is a simple collection of packet identifiers and packet types representing the packet return requests that were received by the worker and are under completion. These requests can be duplicated, created and deleted by the agents themselves making it possible to exchange them through the message system.

The other memory entity of the working agents is called the *WorkerMemory*. This object is able to hold three types of information:

- The *identifiers of exchange points* seen so far by the worker.
- *Packet information* - containing the exact *location*, *type* and *identifier* of a selected packet.
- *Waypoint pairs* representing the corridors of the warehouse.

In the *WorkerMemory* the old information is gradually overwritten by the new details the worker is picking up.

The *WorkerMemory* is generally used to answer queries like how does the agent get from its current position

- to a selected packet,

- to a selected type of packets,
- or to a selected exchange point.

Both memories have a limited capacity that is changeable by the user.

If the agent is commanded through the script language (what is the case in the current version of the program) there are also two built in stacks that are expected to be used as a memory however these are not part of the agent itself. For more details about the stacks see sections “**The built-in script language**” and “**The built-in functions**”.

The agents communicate with each other through **messages**. These messages are pieces of information – in the actual version strings – that is sent to a selected *distance*. All agents closer than the defined distance will receive the *message*. The length of the message is not limited in any way in the program.

4.03 *The work cycles*

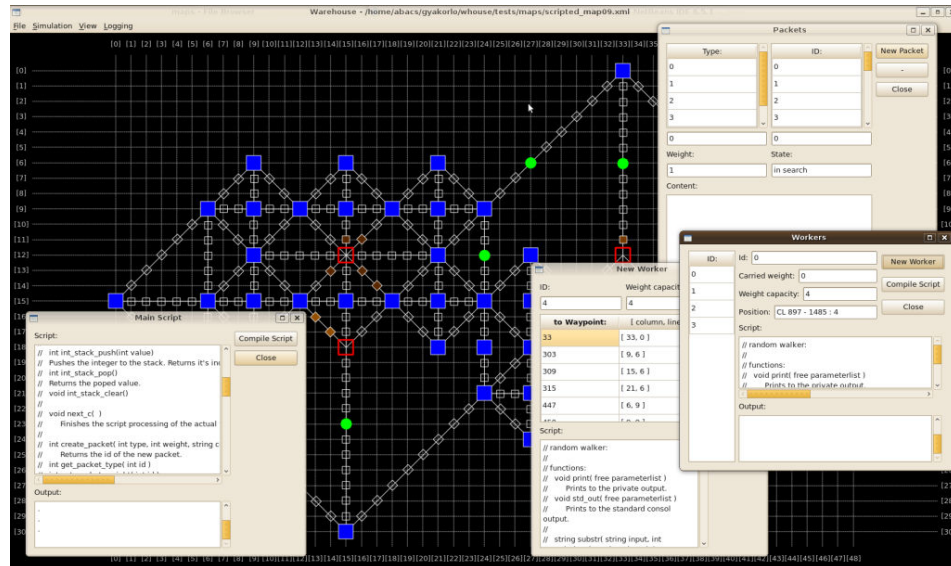
The time in the simulation is counted in work cycles.

In a work cycle (or simulation step) the main script and each of the worker scripts are executed in a predetermined order until they reach their last instruction, or a function call that is defined to temporarily terminate the script execution. Such functions are the movement commands and the `next_c()` function. This behavior assures that the agent’s work is uninterrupted in each cycle. All the information obtained about the environment stays valid until the next function call that takes the initiation from the worker and gives it to another one. At the next work cycle the agent is advised to refresh its knowledge of the environment. If in the last cycle the script of the worker has reached its last instruction, it is reset to the first one.

The user should also note that at the beginning of each work cycle the worker is provided with the messages it received from the last simulation step. If the agent does not process these messages before finishing the actual work cycle they will be lost forever and replaced by new messages.

4.04 The graphic interface

The Program consists of a main window and several dialog-windows. The main window's central part is the map screen where the actual simulation is represented. The size of the representation depends on the size of the main window.



4-2: Image of the running simulation

The visible elements of the simulation are:

- *Waypoints*
- *Exchange points (ExchangePoint)*
- *Corridors (ContainerLine)*
- *Workers*
- *Packets*

The **waypoints** are represented as blue rectangles.

The **exchange points** are shown as see-through red squares, similar to the *waypoints*.

The **corridors** connecting the *waypoints* and the *exchange points* are white lines with white see-through rectangles. The number of rectangles shows the length and storing capacity of the corridors.

The **workers** are represented as green dots. These objects can be placed only on top of the *waypoints*, *exchange points*, or *corridor* containers.

The last visible elements are the **packets**. Those are visible only when placed into a *corridor's* container; in that case the color of the container changes according to the *packet's* type.

The **main window's menu** has the following menu options:

- „File“
- „Simulation“
- „View“
- „Log“

The „**File**“ menu's menu options are:

- „**Open Map**“ – opens a prepared simulation file, if another simulation was already selected, closes the old one.
- „**Exit**“ – terminates the program.

The „**Simulation**“ menu:

- „**Step**“ – gives order to the actual simulation to provide a single simulation step (work cycle).
- „**Run**“ – By checking this option the simulation executes simulation steps until unchecked.
- „**Speed**“ submenu – Changes the time delays between the simulation steps shorter or longer by a linear constant value. Affects the execution only when *Run* menu option is activated.
- „**Workers**“ submenu – The “**Workers**“ menu option opens the “**Workers**“ dialog window, while the “**Create worker**” shows the “**Create worker**” dialog.
- „**Packets**“ submenu – The “**Packets**” menu option opens the “**Packets**” dialog window, while the “*Create packet*” shows the **Create packet** dialog.
- “**Main script**” menu option – Opens the “**Main script**” dialog window.
- “**Guided simulation**” menu option – Enables or disables the execution of the main script at the work cycles. It makes easier to switch between a script guided simulation and a real-time user guided one (unchecked checkbox).

The “**View**” menu:

- “**Visualize**” menu option – Enables or disables the visual representation of the simulation. Disabling the visualization makes the program run faster, however, to feel any difference; the simulation speed has to be set to minimum value.

- “**Grid**” menu option – Turns on or off the background grid that helps identifying the *waypoints*.

The “**Log**” menu:

- “**Open Logfile**” menu option – Opens a file dialog to select an xml logfile where the simulation should write. The old content is overwritten.
- “**Write Log**” menu option – Enables or disables logging.

4.05 The dialog windows

Packets dialog:

This window contains the details of a single packet. These details are its *identifier*, *type*, *weight*, actual *status*, and its text *content*.

The *identifier* is unique for all packets in the simulation; their type is represented by an integer value. The text content is a standard string.

The *status* of the packet can be

- “*outside*” – not in the simulation.
- “*in the warehouse*” – the packet has been sent to the warehouse, and is in a container, in a worker’s backpack, or waiting for a worker at an exchange point.
- “*in search*” – a packet request has been generated and sent to the exchange points.

The screenshot shows a window titled "Packets" with a standard OS window control bar. Inside, there are two vertical lists at the top: "Type:" and "ID:". The "Type:" list has four items: 0, 1, 2, and 3, with item 0 selected. The "ID:" list also has four items: 0, 1, 2, and 3, with item 0 selected. To the right of these lists are two buttons: "New Packet" and "Close". Below the lists, there are two input fields: "Weight:" with the value "1" and "State:" with the value "in search". At the bottom, there is a large text area labeled "Content:" which is currently empty.

4-3: The details of the Packet "0"

To select a packet to be shown in the dialog, the user has to select the correct type from the types list, than the required identifier from the ID list.

The “**Create Packet**” button opens the *Create Packet* dialog.

The third button in the window changes according to the actual state of the packet. It can be a “**Send in**” button, that puts the packet into the simulation, or “**Ask back**” that generates a return request for the packet.

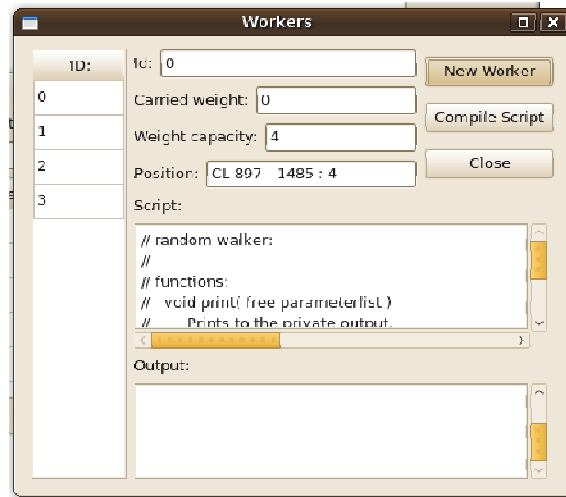
Create packet dialog:

In this window the user can set the type, weight and content of a new packet; than the program creates the packet with the smallest unused identifier and adds it to the simulation.

Workers dialog:

This window contains the details of a single agent. These details are its *identifier*, *maximal weight*, the worker is able to carry, the *actual weight load*, actual *positron in the warehouse*, the *driving script*, and the *output*.

The *identifier* is unique for all workers in the simulation.

The screenshot shows a window titled "Workers" with a list of worker IDs (0, 1, 2, 3) on the left. The details for worker 0 are displayed on the right. The fields include: "Id:" with value 0, "Carried weight:" with value 0, "Weight capacity:" with value 4, and "Position:" with value "CL 897 1485 : 4". There is a "Script:" text area containing code:

```
// random walker:
//
// functions:
// void printU (free parameterlist)
// Prints to the private output.
```

 Below the script is an "Output:" text area. Buttons for "New Worker", "Compile Script", and "Close" are visible.

4-4: The details of the worker "0"

The driving script can be rewritten by the user. The button “**Compile script**” recompiles and reinitializes the actual agent’s driving script and the compilation’s result is placed into the *output* field.

To select a worker to be shown in the dialog, the user has to select the identifier of the worker type from the types list, than the required identifier from the ID list.

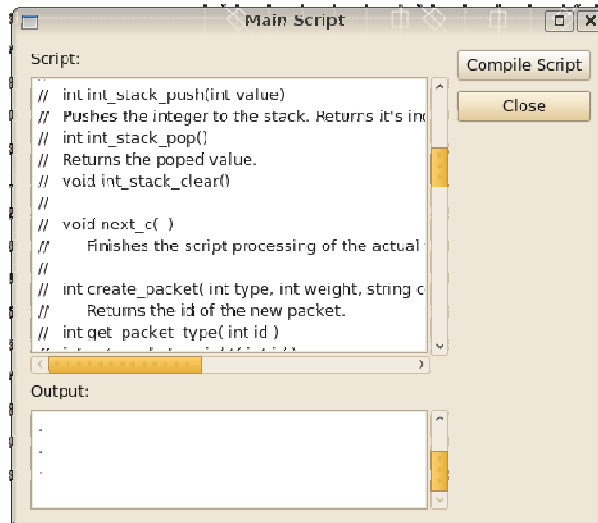
The “**Create Worker**” button opens the *Create Worker* dialog.

Create worker dialog:

In this window the user can set the carrying capacity, the starting waypoint and the driving script of the new agent; than the program creates the worker with the smallest unused identifier and adds it to the simulation.

Main script dialog:

This dialog contains two fields, the *main script*, and its *output*. If the user chooses to modify, or rewrite the main script, he should recompile it with the “**Compile script**” button.



4-5: A running script in the Main script dialog

4.06 The input

The program's main input is an xml file.

If the program is started from the command line, the following options are available:

- "-l" LOGFILE Opens a logfile for overwriting after the program has started.
- "-m" MAPFILE Opens the selected map file after the program has started.
- "-p" Opens the *Packets dialog* after the program has started.
- "-s" Opens the *Main script dialog* after the program has started.
- "-w" Opens the *Workers dialog* after the program has started.

The input file is expected to be an xml file defined in the **map.dtd** file. There can be set the number and driving script of workers, the types of the packets, and the main script.

The layout of the warehouse is given by a character map, for further details see the file **input_file_format.pdf**.

4.07 The output

The program may produce three types of outputs.

- xml log file
- windowed output

- Text output to the standard console output.

The xml log is generated automatically and can be directed to a selected file. Its format is defined by the **log.dtd** file.

The Main script – and Workers dialogs are both containing an output field where the running scripts are writing their output.

The script language contains a function that writes directly to the standard output for logging purposes.

4.08 *The built-in script language*

The syntax of the language is based on the C language with some important differences such as:

- There are no arrays, or complex types.
- There are no pointer operations.
- Variable initialization is compulsory at declaration.
- Variable names when already defined should be preceded by a „\$“ character.

```
// example script
{
    print("hello world!\n");

    for(int i = 0; $i < 2; $i++){
        print($i, ". hello again!\n");
    }

    print("good bye!\n");
}
```

4-6: A simple example script

The scripts can be of two types – single step scripts or long running scripts.

A single step script is used in most agent scenarios. This means the script describes a single decision tree that the worker evaluates in each and every work step.

At the end of the work step the evaluation reaches one of the end nodes and the script terminates. In the next step the worker starts the script from the beginning with reinitialized variables.

To somewhat extend the describing strength of the language in these single step scenarios, there are two additional function groups operating with a stack. The two stacks are an integer and a string stack. These are not overwritten between different script runs so they are expected to be used mostly as status information storages.

The other type of script supported by the language is a long running script. This script is completed through multiple simulation steps, in some cases through the entire simulation. This behavior is expected from the main script. The language supports two types of variables: signed integer (int) and string (string). Both types are defined by the most common operators used in similar languages.

Additional properties of the script language:

- Expressions with no effect are cut from the evaluation tree. Typically such operations are expressions without variable assignment, or without functions that in some way changes the state of the simulation.
- Stacks provide virtually unlimited storage place for the convenience of the user, but it is not expected to be “abused”.
- Worker scripts run until they reach a movement function, the main script has to be stopped explicitly using the *next_c()* function.
- The main script and each of the worker owned scripts is running as a separate entity. There is no way to change information between them except of using the message sending systems.

For the syntax of the script language see **script_diagrams.pdf**.

4.09 *The built-in functions*

The script language provides a list of preregistered functions in the fields of string operations, output and variable storing.

In the following context „free parameters“ means that the function accepts any number or type of parameters.

The indexing of the following functions is always expected in the interval {0, 1, ...}.

Output functions:

Print the selected values to the program’s output.

```
void print( free parameters )
```

- The parameters can be of both string and integer type. They are concatenated to a single string and sent to the built in output (in the whouse program to the local worker or main script outputs).

```
void std_out( free parameters )
```

- The parameters can be of both string and integer type. They are concatenated to a single string and sent to the standard output.

String related functions:

They implement the most basic string operations. In addition to these functions string concatenation is implemented using the “+” operator.

```
string substr( string input
              , int start_index
              , int substr_length )
```

- It returns a substring of the input string starting at index position to the “index + substr_length” position. If the substr_length value is too big, it returns the largest possible substring. If there is no such substring at all, or the index is invalid (smaller than 0, or larger or equal to the size of the original string) returns an empty string.

```
int strlen( string input )
```

- It returns the length of the input string.

```
string int_to_str( int value )
```

- Translates an integer value to string representation.

```
int str_to_int( string value )
```

- Translates a string number to integer representation if there is any. If the string is not the representation of a valid number, it returns 0.

Pseudorandom number generator:

Uses the standard C language pseudorandom generator functions)

```
int rand( int minimal_value, int maximal_value )
```

- It returns a random integer value from the selected interval.

```
void srand( int seed )
```

-It seeds the random generator with the seed input value.

```
void srand( )
```

- It seeds the random generator with the current system time.

String stack:

A stack that remains intact between distinct script runs.

`string str_stack(int index)`

- It returns the value found on the index position from the string stack.

`string str_stack_set(int index, string value)`

- It sets the value in the index position on the string stack. It returns the old, replaced value.

`int str_stack_size()`

- It returns the number of values on the string stack.

`int str_stack_push(string value)`

- It pushes a new value to the top of the string stack. It returns the index of the new value.

`string str_stack_pop()`

- It removes the value from the top of the string stack. It returns the removed value. If the stack is empty, returns an empty string.

`void str_stack_clear()`

- It removes all elements from the string stack.

Integer stack:

A stack that remains intact between distinct script runs.

`int int_stack(int index)`

- It returns the value found on the index position from the integer stack.

`int int_stack_set(int index, int value)`

- Sets the value in the index position on the integer stack. It returns the old replaced value.

`int int_stack_get_size()`

- It returns the number of values on the integer stack.

`int int_stack_push(int value)`

- It pushes a new value to the top of the integer stack. It returns the index of the new value.

`int int_stack_pop()`

- It removes the value from the top of the integer stack. It returns the removed value. If the stack is empty, returns an empty string.

`void int_stack_clear()`

- It removes all elements from the integer stack.

4.10 *Main script specific functions*

The following functions are only available in the main script.

`void next_c()`

- Breaks the script execution until the next working cycle.

Packet related:

`int create_packet(int type, int weight, string content)`

- Creates a new Packet outside of the warehouse with the selected type, weight, and content. It returns the identifier of the new packet.

`int get_packet_type(int id)`

- It returns the type of the packet with the selected identifier. If no such packet exists, returns -1.

`int get_packet_weight(int id)`

- It returns the weight of the packet with the selected identifier. If no such packet exists, returns 0.

`string get_packet_content(int id)`

- It returns the content of the packet with the selected identifier. If no such packet exists, returns an empty string.

`string get_packet_status(int id)`

- It returns the status of the packet with the selected identifier. If no such packet exists, returns an empty string. Possible status values: "in" means in storage, "outside" means not in the warehouse, "in search" means a search request has been sent to the workers.

`int get_number_of_packets()`

- It returns the number of existing packets in the simulation.

`int get_number_of_packets_outside()`

- It returns the number of packets that are not in the warehouse or in search.

`int get_number_of_packets_inside()`

- It returns the number of packets that are in the warehouse and are not in search.

`int get_number_of_packets_in_search()`

- It returns the number of packets that are in the warehouse and are in search.

```
int send_packet_to_warehouse( int packet_id )
```

- Puts the packet to one of the ExchangePoints (to the first one) which gives it to a worker to be placed into the warehouse. Returns 1 if succeeds, or 0 if no such packet is found outside the warehouse.

```
int send_packet_to_warehouse( int packet_id, int epoint_id )
```

- Puts the packet to one of the ExchangePoints (with the selected identifier) which gives it to a worker to be placed into the warehouse. Returns 1 if succeeds, or 0 if no such packet is found outside the warehouse or no ExchangePoint exists in the warehouse.

```
int ask_packet_from_warehouse( int packet_id )
```

- Puts a packet request to one of the ExchangePoints which gives it to a worker so the selected packet could be returned. Returns 1 if succeeds, or 0 if no such packet is found in the warehouse.

ExchangePoint related:

```
int get_number_of_epoints( )
```

- Returns the number of existing exchange points in the warehouse.

```
int get_epoint_id( int index )
```

- Returns the identifier of the index-th exchange point in the warehouse. If no such point exists there, returns 0.

Worker related:

```
int create_worker( int weight_capacity, string script )
```

- Creates a worker and places it to the first exchange point with the selected carrying capacity and script. It returns the id of the new worker.

```
int get_number_of_workers( )
```

- Returns the number of workers (agents) in the warehouse.

4.11 *The worker specific functions*

The following functions are only available in the worker scripts.

Packet related:

Picking up and putting down the packets both at container positions both at exchange points.

`int get_packet_type(int id)`

- Returns the type of the packet if it's at the worker's backpack, or on the actual position.

`int get_packet_weight(int id)`

- Returns the weight of the packet if it's at the worker's backpack, or on the actual position.

`string get_packet_status(int id)`

- Returns the status of the packet if it's at the worker's backpack, or on the actual position.

`int get_weight_capacity()`

- Returns the weight limit the worker can carry.

`int get_carried_weight()`

- Returns the currently carried weight.

`int get_number_of_packets_in_container(int backpack_flag)`

- Returns the number of packets in the selected container (in, out, move). {in ... 1, out ... 2, move ... 3}.

`int script_get_packet_id(int backpack_flag, int index)`

- Returns the id of the index-th packet in the worker backpack in the selected container. {in ... 1, out ... 2, move ... 3}.

`int is_shelf_empty()`

- Returns true, if the shelf is empty.

`int peek_packet_id()`

- Returns the identifier of the packet found on the current shelf, or -1, if there is no packet.

`int peek_packet_type()`

- Returns the type of the packet found on the current shelf, or -1, if there is no packet.

`int peek_packet_weight()`

- Returns the weight of the packet found on the current shelf, or 0, if there is no packet.

`int pick_up_packet(int backpack_flag)`

- Picks up a packet from the current position if possible. Returns the identifier of the picked up packet if successes.

`int put_down_packet(int packet_id)`

- Puts the packet with the selected identifier to the current container position (or to the current exchange point).

`int put_down_packet_type(int backpack_flag, int packet_type)`

- Puts the packet with the selected type and worker backpack container to the current container position (or to the current exchange point).

Position related:

`int get_position_type()`

- Returns the type of object where the worker is standing. Possible values { 0 means unset value, 101 means corridor, 201 means waypoint, 202 means exchange point }

`int get_position_id()`

- Returns the identifier of the actual waypoint, or exchange point where the worker is standing. If the worker is standing on a corridor, returns -1.

`int get_subposition()`

- Returns the sub-element's index where the worker is standing.

`int get_number_of_directions()`

- Returns the number of possible directions to go from the current position.

`int get_direction(int index)`

- Returns the identifier of the waypoint where the selected direction is leading.

`int get_distance(int wp_id)`

- Returns the distance of the selected waypoint.

`int move_towards(int wp_id)`

- The worker makes a step towards the selected waypoint. Returns the remaining distance.

`int exists_wpoint(int wpoint_id)`

- Returns the existence of the waypoint with the selected identifier.

`int exists_epoint(int epoint_id)`

- Returns the existence of the exchange point with the selected identifier.

`int exists_cline(int cline_starts_id, int cline_ends_id)`
- Returns the existence of the ContainerLine with the selected identifier pair.

`int create_cline(int cline_starts_id, int cline_ends_id)`
- Creates the required ContainerLine if it does not interfere with the graph's structure. Returns an indicator of success.

Message related:

`void send_message(int distance, string text)`
- Sends the message to the selected distance with the selected text.

`int get_number_of_messages()`
- Returns the number of received messages.

`string get_message()`
- Returns the last of the received messages, or empty string.

Request related:

The requests are held by the worker in a list identified by their packet identifier and packet type.

`int get_request_limit()`
- Returns the maximal number of requests.

`int set_request_limit(int new_limit)`
- Sets the maximal number of requests. If there are more requests in the container than the new limit, returns with failure.

`int get_number_of_requests()`
- Returns the number of packets requests.

`int add_packet_request(int packet_id, int packet_type)`
- Adds a new request with the selected id and type. If the id and type pair is in the list, does nothing. If there is no free place, returns 0.

`int get_request_from_epoint()`
- Asks a request from the current position (exchange point only). Returns success indicator.

`int remove_packet_request_by_id(int packet_id)`
- Removes a packet request. Finds it by its packet identifier.

```
int remove_packet_request_by_type( int packet_type )
```

- Removes a packet request. Finds it by its packet type.

```
int is_packet_requested( int packet_id )
```

- Returns true if the packet identifier is between the requests.

```
int is_type_requested( int packet_type )
```

- Returns true if the packet type is between the requests.

```
int get_requested_packet_type( int packet_id )
```

- Returns the type of the selected packet.

```
int get_requested_packet_id( int request_index )
```

- Returns the identifier of the index-th request.

4.12 **Setting up the program**

The program's current version is written and tested in the operating system of 64 bit *Linux*. The binary of the program on the CD was also compiled for this system; however the project was written using only tools and extensions that are available on a wide range of operating systems.

It uses standardized *C++* functions where it is possible. The outer shell of the program - handling both the windowed environment and its graphics - is written in the *QT environment* (the previous version used *WinAPI* functions with *OpenGL* graphics).

It is possible to compile the program on any system that has the utilities *g++*, *make*, and the mentioned *QT environment*.

If *make* is not available on the selected platform, but *QT* is, the project can be easily set up from only the sources using the QT automated project creation tools (*qmake* in Linux).

To **compile** the program on the 64 bit Linux the user should enter the directory *whouse/linux_64* and use the command *make*. (There are also supported the other usual make options.)

To **run** the program on the 64 bit Linux the user should enter *whouse/linux_64/bin* directory and execute the *whouse* binary file. For further details about the program options see section “**The input**”.

To set up the program on **other QT supported systems** the user should create a new dedicated directory in the *whouse* directory, copy the *whouse.pro* file from the original Linux directory to the new one and create the project from the copied file using the local QT tools.

The project also uses the tools *bison* and *flex* to create its script language interpreter, but the recompilation of the language files is not necessary, the generated files are already put into the source directory. If the user wanted to change and recompile them he should use the *Parser.yac* and *Scanner.lex* files. For the exact commands creating the currently used generated source files see the *compiler_construction_commands.txt* file in the compiler sources subdirectory.

To **update** the programmer's **documentation** the user should enter the directory *whouse/doc* and run the *doxygen* utility from there or running it with the file *whosue/doc/Doxyfile*.

4.13 *Possible improvements in the future*

Conceptual improvements

There is a parameter named *weight* introduced into the simulation that determines how much can a worker carry. If a packet is too heavy, it is possible that some workers can't pick it up at all. There is no way of multiple agents carrying a single packet that would be too heavy to a single worker, but it's an interesting concept and I would like to realize it in the future.

In the current version of *whouse* batch execution is not part of the program. The user has to open each input file by hand or terminate the program completely and start it from the command line with the new input file. In the future this capability should be added to the program.

Technical improvements

The program in its present state does not recognize infinite loops in the scripts. If such situation occurs, the program has to be terminated by the system or by the user. There should be a control entity in the script language implementation that stops the suspiciously behaving scripts.

The language should be extended with the possibility to define own functions in the script files. This would make the worker scripts much shorter and easier to read.

In the present state of the program the built-in path memory is not accessible from the script language. It would greatly reduce the complexity and possibly

lower the time requirements of the scripts if the path finding algorithms would be definitely moved into the native C++ code.

In the QT environment (the current version) the program has no graphic representation for its so called “active log” which follows the amount of items such as workers, packets, messages and their actual position. The previous version of the program was able to create a chart from these details. This feature should be also put into the new handler structure in the future.

In the present state the warehouse extending functions (adding new Waypoints, ExchangePoints, and ContainerLines to the warehouse) are connected only to the script language, later these should be also connected to the graphic user interface.

The implementation of worker communication consists of text messages. It would improve their message processing abilities if they communicated through an already “parsed” system that would not append together the message string at sending to be taken apart by the receiver but rather forward it in pieces.

Chapter 5. Conclusion

The distributed database driven model of the storage facility works but as it could be expected, it has not only benefits but also shortcomings. In this chapter I would like to point out some of the issues I experienced.

The most severe of the shortcomings I experienced is the length of the *retrieval times*. The workers have no problem quickly placing the packets to a free place than later reorganizing it; but it needs reasonably more working agents to provide at least similar retrieval times as the single central entity system would require.

As I observed there are two key differences that provide the speed difference.

The production systems with the central entity sends the retrieval request (or a command sequence equivalent of that) to the closest worker, while in my distributed system it needs to wait until an agent without other tasks to do returns for further instructions. The results of this issue could be at least reduced by introducing priorities, connecting them to the possible worker actions and giving high priority to returning to the exchange points.

The other, more interesting point is, in the centralized systems the main entity has the power to decide which task is the most important at any point and can alter the behavior of the workers accordingly, while in my system each agent decides about its behavior independently based on its limited amount of information. This inconvenience could be solved by increasing the number of agents and differentiating them by assigning them to different types of tasks so in each type of action there would be enough workless agents ready to complete tasks according to the actual needs.

These type groups I realized in my program (built into the agents AutoWorker) as agent states that would change based on the perceptions and recent actions of the agent. These states cover only the most important task categories:

- to search for a selected packet,
- to find the closest exchange point (used when returning with a packet),
- to find a convenient container position to drop the packet,
- To reorganize the storage place.

In small warehouses this solution is sufficient to provide the required quick retrievals, but as the warehouse gets larger, the retrieval times does not rise proportionally.

Another, but different issue with the autonomous agents is their tendency of *venturing away* from their known territory. After they leave their charted up

surrounding according to their settings they either forget the old territory in favor of the new one, or simply get lost for a while.

As a solution of this issue I tried to limit their movement to known paths after their memory has filled up with the first several explored waypoints and with that bound them to their known area, where they can work efficiently. This however disables one of the great characteristics of the autonomous agent design; that it is able by itself exploring and putting to use parts of the environment that to the point were unknown, or possibly didn't even exist (In our case newly added waypoints and corridors). To keep the good characteristics of both behavior I think there should be introduced a system that would from time to time allow the agents to venture away from their known territories but keep them returning to their initial surroundings.

In my point of view the distributed agent system's potential can be used mainly in situations where their work environment is regularly and/or unexpectedly changing to the point that the central entity can't keep up with the updates seamlessly. In these situations the slower retrieval times of the distributed system are not an issue and their adaptability and robustness can be a great advantage.

Bibliography

- [1] Michael Wooldridge: An Introduction to Multiagent Systems, John Wiley & Sons (2002), ISBN-10:0-471-49691-X
- [2] Yoav Shoham, Kevin Leyton-Brown: Multiagent Systems: algorithmic, game-theoretic, and logical foundations, Cambridge University Press (2009)
- [3] Stanley P. Franklin: Artificial Mind, The MIT Press (1995, fifth printing 2001)
- [4] José M. Vidal: Fundamentals of Multiagent Systems (2007)
- [5] J. Tweedale, N. Ichalkaranje, C. Sioutis, B. Jarvis, A. Consoli, G. Philips-Wren: Innovations in multi-agent systems, Journal of Network and Computer Applications (2006)
- [6] Nikos Vlassis: Multiagent Systems and Distributed AI, University of Amsterdam (2003)

Appendix

- `whouse/doc/input_file_format.pdf`
guide to create an input file.
- `whouse/doc/map.dtd`
Format definition of the input xml file.
- `whouse/doc/log.dtd`
Format definition of the output xml file.
- `whouse/doc/script_diagrams.pdf`
The syntactic and lexical diagrams of the script language.
- `whouse/doc/doxy/index.html`
The programmer's documentation created using Doxygen.
- `whouse/linux_64/bin/whouse`
The program itself (compiled for 64bit Linux systems).
- `whouse/linux_64/Makefile`
Automatically generated makefile of the program.
- `whouse/linux_64/whouse.pro`
The project file of the program.
- Source files in the `whouse/src` directory
- Test files in the `whouse/tests/maps` directory