Ilkka Hartala

Goal Oriented Action Planning for Agent Simulations

Bachelor of Business Administration

Business Information Technology

Spring 2016

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä(t): Hartala Ilkka

Työn nimi: Tavoitteellinen tehtäväsuunnittelu agenttisimulaatiokäyttöön

Tutkintonimike: Tradenomi, Tietojenkäsittely

Asiasanat: tekoäly, tavoitteellinen tehtäväsuunnittelija, optimointi, helppokäyttöisyys, simulaatio, älykkäät agentit, C#, ANTRL, Unity

Opinnäytetyössä tehtiin helppokäyttöinen ja tehokas tavoitteellinen tehtäväsuunnittelija reaaliaikaisiin ja vuoropohjaisiin agenttisimulaatioihin. Järjestelmää tullaan käyttämään henkilökohtaisissa projekteissa.

Tavoitteellinen tehtäväsuunnittelu on tekoälyarkkitehtuuri, joka erottaa toisistaan agentin toiminnot ja toimintajärjestyksen. Tämä mahdollistaa optimaalisten ratkaisujen löytymisen moniin haasteisiin ja lisää tekoälysuunnittelun joustavuutta.

Tehtäväsuunnittelijan kehitys koostuu järjestelmäarkkitehtuurista, ohjelmoinnista ja optimoinnista. Työn toteutuksessa käytettiin monenlaisia ohjelmointimalleja, teknologioita ja tekniikoita: A*-polunetsintäalgoritmia tehtäväsuunnitteluun, ANTLR-syntaksijäsentäjää tiedostomuotoiluun, data-orientoitunutta ohjelmointia optimointiin ja Unity-kehitysympäristöä esimerkkiprojektin tuotantoon.

Kaiken kaikkiaan projekti onnistui. Tehtäväsuunnittelija valmistui tavoitteiden mukaisesti ja esimerkkiprojekti esittelee suunnittelijan toimintaa simulaatioympäristössä. Järjestelmän kehitystyö jatkuu tulevien projektien tarpeiden mukaisesti.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author(s): Hartala Ilkka

Title of the Publication: Goal Oriented Action Planning for Agent Simulations

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: Goal oriented action planning, Artificial intelligence, Data oriented programming, Usability, Optimization, Simulation, C#, ANTLR, Unity

The objective of this thesis is to create an easy to use and performant goal oriented action planner for interactive real-time and turn-based agent simulations. The system will be used in personal projects.

Goal oriented action planning is an artificial intelligence architecture which decouples what the agent can do from what the agent will do. This allows the system to come up with optimal solutions for manifold challenges and increases the flexibility of the AI design process.

The development of the planner consists of system architecture design, programming and optimization. A variety of programming paradigms, technologies and techniques were used in the process: A* pathfinding algorithm for planning, ANTLR syntax parser for creating custom file formats, data oriented programming for optimization and Unity for the example project production.

Overall the project was a success. The planner works as expected and the example project demonstrates planning functionality in a simulated environment. Nonetheless the development of the system is far from over as there is always room for improvement and refinement based on the needs of future projects.

Table of Contents

LIST OF SYMBOLS

GOAP	Goal Oriented Action Planning.
AI	Artificial Intelligence. AI in interactive media refers to algorithms which mimic intelligent behaviour.
ANTRL	ANother Tool for Language Recognition.
DLL	Dynamically Linked Library.
OOP	Object Oriented Programming. A widespread programming paradigm which joins the program design and code together by encapsulating data and functionality in objects.
DOD	Data Oriented Design. A programming paradigm which focuses on data and memory management. It can be used in conjugation with other paradigms.
AoS	Array of Structures. A way of arranging data in memory by storing objects composed of variables in arrays.
SoA	Structure of Arrays. A way of arranging data in memory by storing individual variables in their own arrays.
Heap	A memory space used for dynamic data allocation.
Stack	A memory space used for static data allocation and function scope management.

1 INTRODUCTION

Artificial intelligence has two basic definitions: academic study of general intelligence, software that can think and learn; and algorithms used in interactive media; software that seems intelligent, but is really faking it. [1, page 13]

Goal oriented action planning falls soundly in the second group with a foot in the first group. While the designer does specify all the actions the planner can do, i.e. it does not learn, there is a hint of emergent thinking. A pool of actions big enough will yield clever solutions to problems the author did not foresee.

Learning is also not completely out of the picture. The planner prefers actions of low cost. Manipulating these values based on past plans and the current state of the world would allow for a basic level of learning. Nonetheless this avenue will not be explored in this thesis.

The objective of this project is to create a functional goal oriented action planner and an example project using said planner.

2 THEORY

2.1 GOAP

GOAP is a simplified STRIPS-like planning architecture specifically designed for real-time autonomous character behaviour. It was originally developed by Jeff Orkinn at Monolith Productions to run the AI in F.E.A.R. and has since been utilized in many digital games including Condemned, S.T.A.L.K.E.R, Shadow of Mordor and Tomb Raider. [2]

Unlike state machines and behaviour trees, GOAP decouples actions from plans meaning the system can piece together novel solutions to problems in a changing environment. At best complex and robust plans emerge from a small set of well-thought-out actions.

Despite lofty premises the program design is surprisingly simple. The system is given an initial world state and a goal state to match. Then it searches through a pool of actions which affect the world state. The optimal list of actions is found using a pathfinding algorithm.

GOAP is at its best when used as a high-level decision making system, i.e. the brain of the AI agent. Lower level actions, like movement and sensing, should be handled by other systems which feed their output back into the next world state.

2.2 ANTLR

ANTLR is a powerful parser generator for reading, processing, executing, or translating structured text and binary files. In other words it can be used to read custom file syntaxes [3]. It is widely used in the industry by Twitter, Google and Oracle to name a few. [4]

ANTLR allows creating sleek and simple file formats with little effort. Learning how the parser works takes more time than using standard file formats, like XML and JSON, but the outcome is worth the initial investment.

2.3 DOD

DOD is a programming paradigm which emphasizes understanding how data and logic operations function on hardware level. Its major purpose, for this project, is to make the program runtime execution fast.

DOD can be applied selectively on critical parts of the program, where it is most useful, or as a general design principle [5]. Other potential benefits of DOD include painless parallel processing, high tolerance for refactoring and code detached from overly specific designs.

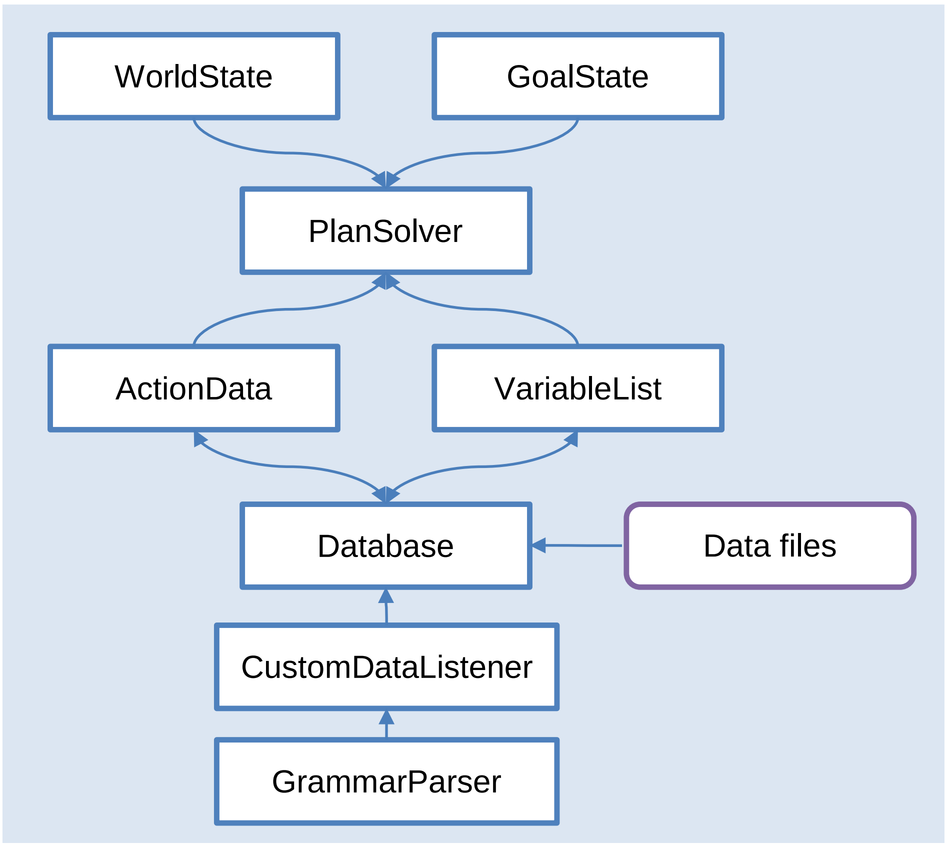
3 AI SYSTEM DEVELOPMENT

The goal of the development is to create an easy to use and fast system. As such all technical decisions have to strike a balance between usability and execution speed. The plan solver, as the most performance critical part of the system, was optimized as much as possible without sacrificing a minimum level of user interface niceties. Nonetheless some compromises were made to squeeze out a few extra milliseconds here and there. Whether that was the right calls is revealed later when working with the example project.

3.1 System Architecture

The system architecture consists of multiple key classes and data files (See Figure 1). These components will be explained in the following chapters.

Figure 1: System architecture diagram



3.2 GOAP implementation

3.2.1 World state

The world state is simply an array of variables. The most performant option of representing these variables would be a single 64 bit number where each bit stands for a specific variable. This would limit the amount of variables per world state to 64, which is more than enough for most use cased. Unfortunately it would also limit the variable type to boolean: true or false, 1 or 0. This is not adequate design as most projects use integer values to store all sorts of data about their world state.

To accomplish a user friendly system an array of integers was used instead. This gives greater flexibility for the user because boolean and even string variables can be represented with integers. As a trade-off comparing and copying world states will be slower than with a single 64-bit number, which is a bit of a bummer.

3.2.2 Goal state

The goal state could be defined as yet another world state with specific target values for user selected variables. The problem is that specific values are both error prone and arduous to work with as the plan solver can easily get stuck in a loop if the action effects are not defined carefully.

To solve this an array of conditions is used as the goal state. A condition is defined by a variable, an operator and a target value, for example the condition "Money > 100" returns true if the "Money" variable in the current world state is bigger than 100.

Under the hood a condition is constructed with two integer values "min" and "max" and a conditional expression. The "min" and "max" values are set differently for each operator. For instance with the equals operator both values are set to be the target value, while with the “bigger than” operator the "min" value is set to the target value and the "max" value is set to int.MaxValue (the maximal integer value).

How the condition works is revealed by the conditional expression:

```
return min <= value && value <= max;
```

In the "Money > 100" example a value of 150 will be bigger than the "min" value of 100 and smaller than the "max" value of int.MaxValue returning true. A value of 50, on the other hand, is smaller than the "min" value of 100 and will immediately return false.

3.2.3 Action data

Action is the unit of change in the plan solving process. All actions together define the possibility space for all AI plans.

An action is composed of an array of requirements, an array of effects and a cost. The plan solver can only use an action if all the requirements are met in the current world state. The requirements use the same conditions as the goal state.

The effects are applied to the world state with arithmetic operations. For example the effect “Money += 10” increments the “Money” variable in the current world state by 10.

The arithmetic operations are implemented as a switch/case structure which selects the proper arithmetic method based on the operation type. At first a delegate was used to do the matching directly, but that was revealed to be much slower than a simple switch/case.

3.2.4 VariableList

VariableList is an object composed of an array of variable names used by the solver to match a variable index with a variable value from the world state.

The system would function without such a construct, but it turned out to be rather suboptimal for the solver to deal with arrays of arbitrary length. Having a fixed length array removed this issue.

3.2.5 Database

The Action and VariableList objects are stored in a database for ease of access. In practice there are two databases ActionDatabase and SharedDatabase. ActionDatabase is used for storing and parsing Actions only, while SharedDatabase holds VariableList objects and potentially other data in the future.

3.2.6 PlanSolver

A plan is simply a sequence of actions performed one after another. To find a plan from a pool of actions one needs a plan solver.

A simple solver iterates through all available actions and constructs a search tree where every node consists of a new world state that an action has modified. When a world state matching the goal state is found it is recorded. When the search has run to a specified tree depth it is terminated and the shortest matching plan is selected from all the found matches.

This initial, naïve, implementation was easy to setup and sufficed for primary testing, but a pathfinding algorithm is required for the solver to be of any use in a real project. [6, page 4]

3.2.7 Pathfinding

At first a fringe search algorithm was incorporated into the plan solver which I had developed a few years ago and used in prior projects. This did not turn out well.

While the fringe search algorithm is potentially faster on grid based maps than A* [7], an industry standard path finding algorithm, in this task it failed to perform adequately. It is possible that my own implantation was not well optimized to begin with, but rather than spending any more time on that the good old A* algorithm was implemented instead.

An open-source A* implementation from a C++ GOAP project on GitHub was used as the basis for the solver [8]. In the end the A* solver performed twice as fast as the naïve solver within the testing setup. Proper performance numbers will be presented in the optimization chapter.

3.3 ANTRL implementation

To make working with actions a breeze a simple external data format is required. ANTLR is used to parse this format.

Here is an example action in the custom data format:

```
Action GoToHospital
       Cost
       2

       Requirements
       Sick == true;

       Effects
       Position = Hospital;
```

ANTLR uses a robust grammar to create custom syntax parsers. An in depth explanation of this grammar is beyond the scope of this thesis, but here is a peek at the specific part of grammar which parses the above Action syntax:

```
(
       ACTION name

       COST COLON? cost

       (
       REQUIREMENTS COLON?
       (
       (requirement) (COMMA | SEMICOLON)
       )+
       )*
```

```
        EFFECTS COLON?
        (
        (effect)+ (COMMA | SEMICOLON)
        )+
)+
```

Full documentation of the grammar can be found on the ANTLR GitHub page [9]

3.3.1 Setup

Setting up ANTLR in a C# project is not difficult. Only two external libraries are required on top of the grammar file: antlr-4.5.1-complete.jar and Antlr4.Runtime.dll.

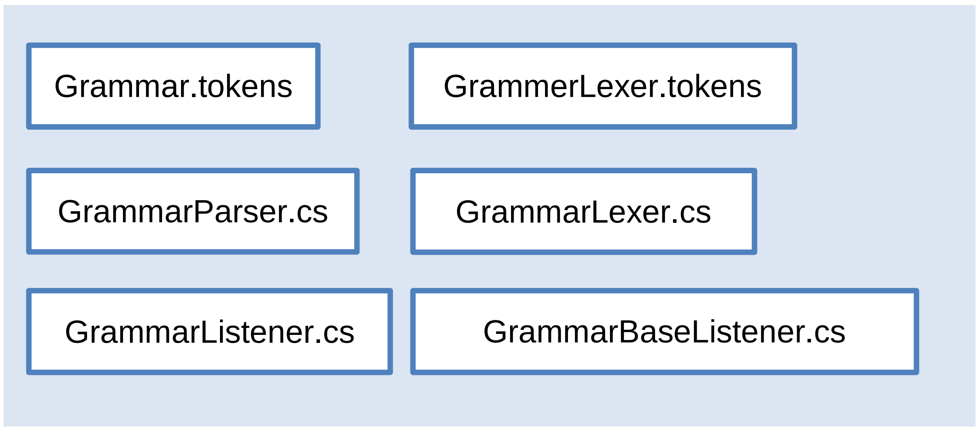
The jar file is used to compile the grammar file into a bunch of C# classes. This is done with the following command line command:

```
java -jar antlr-4.5.1-complete.jar -Werror -Dlanguage=CSharp -o ../GrammarParser/
Grammar.g4
```

Where “../GrammarParser/” is the path to which the runtime C# code is generated and “Grammar.g4” is the path to the grammar file to parse. This command can of course be run from a bat file.

The compiler generates six files from the grammar. (See Figure 2)

Figure 2: ANTLR grammar source files



The GrammarBaseListener serves as a base class for the custom grammar listener. The listener is notified whenever a specific token is processed by the parser. For instance, the name of an action is a token as it needs to be recorded in the ActionDatabase.

3.4 Optimization

The initial version of the system was created using standard object oriented design. OOP is good for quickly turning a design into working code. Objects are also easy to understand and modify if the program architecture is designed carefully.

The number one problem with OOP in resource intensive applications, like AI algorithms, is inefficiencies in memory management. In the worst case scenario object are stored in the heap and multiple methods are called within each object every iteration.

To mend these performance problems data oriented design techniques were utilized to replace object oriented code wherever possible.

The Visual Studio profiler was used extensively to study the performance of the program. In addition numerous tests and multiple versions of the planner algorithm were created to see how different optimizations affect the execution speed.

3.4.1 DOD vs OOP performance example

The following test illustrates the performance difference between OOP and DOD. Iterating an array of 1 million objects and calling functions within said objects wastes a lot of processing time just by fetching object references from the heap and opening new functions in the stack. Accessing arrays directly, on the other hand, yields great results. (See Table 1)

Table 1: Average performance of object oriented data processing versus data oriented data processing with 1000000 method calls and 20 iterations. (See Appendix 1)

OOP	6.41 ms
DOD	3.73 ms

3.4.2 Heap management

The first order of business is to store the SolverNode objects used by the PlanSolver in a fixed size array rather than generating them dynamically at runtime. The downside of this action is having to set a hard limit on the maximum amount of nodes the solver can use, but the upside is a significant performance boost. (See Table 2)

Table 2: Average performance of dynamic allocation versus static allocation with 20000 solver updates and 20 iterations.

Dynamic allocation	204.31 ms
Static allocation	171.15 ms

3.4.3 From AoS to SoA

The next step up from heap management is to go from an array of structures approach where the node objects are stored in an array to a structure of arrays approach where the each variable in the node object is turned into its own array and the solver functions access these variables directly via node indices.

The unfortunately truth is that C# was designed as an object oriented language and doing things in the data oriented way results in more verbose and bulky code.

The fortunate truth is that, in this case, the optimization was done on code which does not need to change much anymore, apart from possible bug fixes and small tweaks, meaning verbosity had little effect in the development proper.

The milliseconds saved speak for themselves. (See Table 3)

Table 3: Average performance of SoA and AoS with 20000 Solver updates and 20 iterations.

SoA	171.15 ms
AoS	157.75 ms

3.4.4 Garbage management

Garbage collection can create frequent performance drops if the amount of garbage generated is high. The easiest way to minimize such problems is to not use any objects at all, apart from permanent storage objects like PlanSolver, WorldState and GoalState. As such no temporary objects are generated by the PlanSolver or other parts of the code-base.

Any Lists and Dictionaries still in the solver were replaced with arrays due to the garbage created when manipulating said collections.

After these steps the amount of garbage created at runtime was reduced to virtually zero. The action data reader still generates some garbage as it works with strings, but it is not a big deal as it runs only once at program start up.

3.4.5 A* Heuristic

A standard node or grid based A* algorithm comes with a heuristic which estimates the minimum cost from the current node to the goal node. The heuristic is used to trim down the amount of nodes examined in the search, thus speeding up the algorithm. [10]

In such situations there is a plethora of heuristic functions to choose from. These heuristics are also fast to calculate meaning their use in most cases is very much a win-win. Only if the amount of nodes is low and the computation cost of the heuristic is high, should one not use a heuristic at all.

The planner fits the bill. It has a relatively low amount of actions to consider and the heuristic is not a simple distance calculation, but rather a costly check which iterates all goal state conditions every time an action is visited. As such running no heuristic produces noticeable benefits.

One additional reason for this speed boost is the requirement fields found on each action. These fields make sure that unnecessary and inappropriate actions are not searched in vain.

It is possible that multiplying the amount of actions and cutting down on requirements will reduce performance dramatically, but so far that has not been an issue.

3.5 DLL creation

Creating the DLL in Visual Studio turned out to be a cinch. All you need to do is to set the project target framework to “Unity3.5.net full Base Class Libraries”, the project output type to “Class Library” and build.

To clean up the public interface of the project any classes the user does not need to use, such as the main class, are made internal. Internal classed do not show up in the namespace outside the library itself.

An XML file with all method and class summaries can be generated to allow viewing said info when using the DLL in external projects. The only downside of this is that, for some reason, Visual Studio does not strip out internal and private summaries, meaning all that data is available for any prying eyes. They can be removed manually of course, but that is a tad annoying.

4 USER WORKFLOW

The system was designed to be powerful as well as easy to use. The user needs to do the following steps to get the AI up and running.

4.1 Import the DLL to the project

In Unity this is as easy as dragging the DLL into a plugins folder. In a Visual Studio C# project the DLL needs to be linked to the user project as a reference. To call any of the functions in the AI system the GOAP_AI namespace needs to be declared in source files with the “using” directive.

```
using GOAP_AI;
```

4.2 Create Action and VariableList data files

The first file includes all the actions used by the solver:

```
Action Work
       Cost
       2

       Requirements
       Sick = false;

       Effects
       Money += 10;
```

The second file includes the main VariableList:

```
VariableList
       Money;
       Sick;
```

Both files are simple txt files. Actions can also be stored in multiple files, if there is a lot of them.

In Unity these files can be read from any arbitrary data folder, the Resource folder, the Streaming Assets folder or even AssetBundles. The file reader implementation is left to the user.

4.3 Initialize the database

The ActionDatabase is initialize by calling the InitDatabase method with the text contents of all the action.txt files, in this case read from an arbitrary folder called TextData:

```
ActionDatabase.InitDatabase(File.ReadAllText("TextData/Actions.txt"));
```

The SharedDatabase holds the variable lists and is initialized similarly:

```
SharedDatabase.InitDatabase(File.ReadAllText("TextData/Variables.txt"));
```

4.4 Create PlanSolver object

PlanSolver is created via the class constructor:

```
planSolver = new PlanSolver(SharedDatabase.MainVariableList, 10000);
```

It requires the main variable list and solver node pool size as arguments. Both are used for optimization behind the scenes.

4.5 Create WorldState and GoalState objects

These states are also created via their constructors. Initial facts and conditions are set immediately afterwards. They can also be changed later in code.

```
CurrentState = new WorldState(SharedDatabase.MainVariableList);
CurrentState.SetFact("Money", 0);
CurrentState.SetFact("Sick", false);

EndState = new GoalState(SharedDatabase.MainVariableList);
EndState.AddCondition("Money >= 100");
EndState.AddCondition("Sick == false");
```

4.6 Run PathSolver

The solver creates a plan from the current world state and a goal state.

```
plan = planSolver.SolvePlan(CurrentState, EndState);
```

In this case the plan will be populated with 10 work actions. If the sick variable in the world state is set to true, the solver will return an empty plan as it cannot find any actions which can be run while sick.

From this starting point more actions and variables could be added to expand the possibility space of the solver. This is what the example project is for.

5 EXAMPLE PROJECT DEVELOPMENT

5.1 Unity development environment

Unity is a modern, multiplatform engine used for interactive 3D and 2D projects by teams of all sizes. It is especially popular amongst small independent teams, but it has also been used for high profile releases like Hearthstone, Endless Legend, Pillars of Eternity and Cities: Skylines. [11]

5.2 Basic framework

The simulation comprises of AI agents who go about their day completing tasks, mainly making money by working. Each agent lives in a house and works in a workplace. The world also includes a hospital, for those who get sick, and a market as the agents need to buy and eat food.

An agent can be given an array of actions to complete. The agent will continue completing actions one at a time until there are no more actions left. At this point the agent will request another set of actions. In this basic framework completing an action simply equates to waiting for a set period of time.

5.3 AI implementation

The GOAP system is used to give the agents their daily plan. The goal state is set up to increase the amount of money owned and to be at home with full energy. With these conditions the plan gets populated with the following basic actions: go to work, work, eat, go home and sleep.

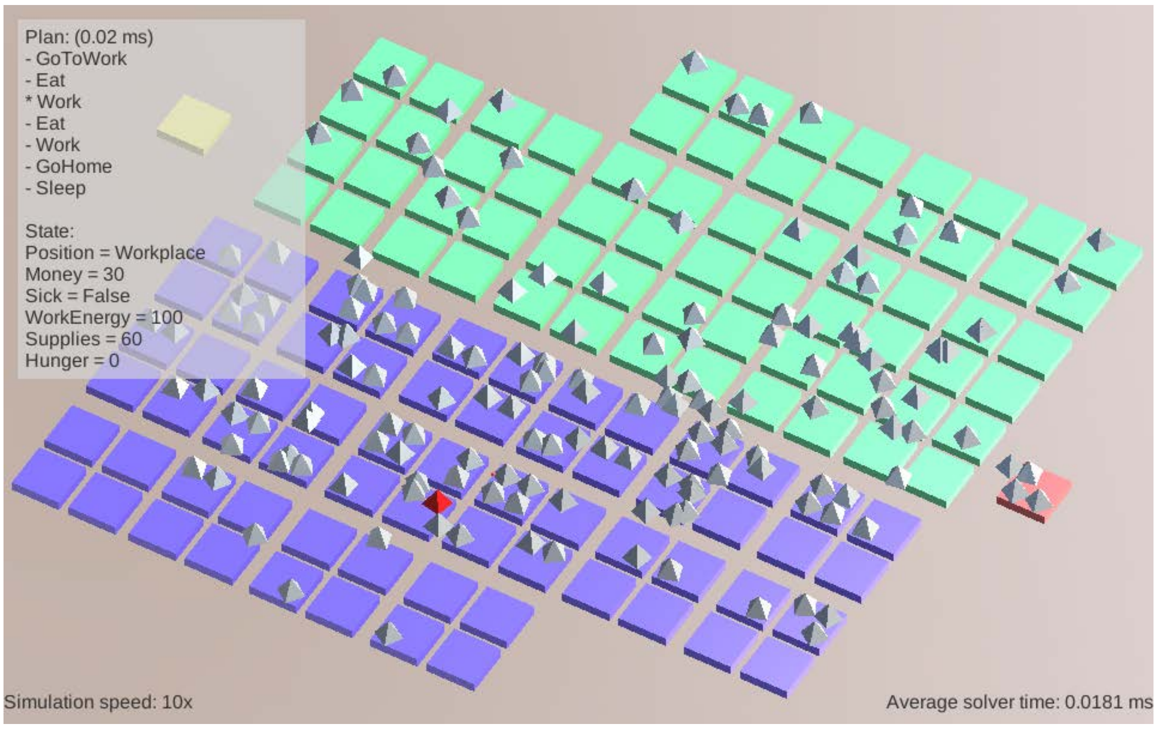
Every time an agent sleeps its energy is regenerated and the goal is adjusted to demand more money. There is a 10% chance for the agent to become sick. A sick agent will recalculate its plan to include a trip to the hospital. Another deviation from the work routine is a trip to the market whenever food supplies run out.

In the end of the day a new plan is calculated yet again. Thus continues the circle of life.

5.4 Showcase

The project runs in the browser thanks to Unity's WebGL HTML5 support and can be tested in the following link [12].

Image 1: Screen capture from the example project



Plan: (0.02 ms)
- GoToWork
- Eat
* Work
- Eat
- Work
- GoHome
- Sleep

State:
Position = Workplace
Money = 30
Sick = False
WorkEnergy = 100
Supplies = 60
Hunger = 0

Simulation speed: 10x

Average solver time: 0.0181 ms

6 SUMMARY

All in all the project was a success. The AI system is functional, easy to work with and performant enough for most use cases. The example project, while not very complicated, does demonstrate planning in changing conditions.

The most technically challenging task was improving performance. In the end I learned quite a few optimization techniques, but there were a setbacks along the way.

In one such case I got so excited by an overwhelmingly positive initial test result of struct performance that I rushed to implement it in the solver. Unfortunately the test only measured reading speed and the solver requires quite a bit of writing as well. Fortunately I managed to leverage that knowledge elsewhere in the code where only reading speed was at stake.

The user workflow seems adequate, to say the least, and it will improve as I continue using the system in the future. There are also some technical questions that were not addressed yet.

One improvement concerns the handling of VariableLists. At the moment only the main list, including all variables, is allowed. Using multiple smaller lists would potentially enhance flexibility and even increase performance.

Another problem is data serialization. How to make sure that action and variable indices in saved plans stay the same when loaded into a new version of the solver with more actions and variables?

There are many ways to solve these issues, and I can already think of a few, but it is better not to implement any theoretical solutions before the problems occur in production code.

REFERENCES

1 Raynor W. The International Dictionary of Artificial Intelligence. The Glenlake Publishing Company, 1999.

2 Alumni media GOAP
http://alumni.media.mit.edu/~jorkin/goap.html
[26.04.2016]

3 antlr.org
http://www.antlr.org/
[26.04.2016]

4 antlr.org testimonials
http://www.antlr.org/testimonials.html
http://www.antlr3.org/testimonial/list.html
[26.04.2016]

5 dataorienteddesign.com
http://www.dataorienteddesign.com/dodmain/
[26.04.2016]

6 Orkin J, Agent Architecture Considerations for Real-Time Planning in Games
http://alumni.media.mit.edu/~jorkin/aiide05OrkinJ.pdf
[26.04.2016]

7 Björnsson Y, Beating A* at Pathfinding on Game Maps
https://webdocs.cs.ualberta.ca/~holte/Publications/fringe.pdf
[26.04.2016]

8 Stolk, GPGOAP / astar.c
https://github.com/stolk/GPGOAP/blob/master/astar.c
[26.04.2016]

9 ANTLR documentation
https://github.com/antlr/antlr4/blob/master/doc/index.md
[26.04.2016]

10 A* Heuristic
http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
[26.04.2016]

11 List of notable Unity releases
https://en.wikipedia.org/wiki/List_of_Unity_games
[26.04.2016]

12 Example project link
http://huvaakoodia.github.io/GOAP_ExampleProject1_WEB/
[26.04.2016]

ATTACHMENTS

```
class Program
{
        private class Test1
        {
            Test2 t1, t2;
            public Test1()
            {
                t1 = new Test2();
                t2 = new Test2();
            }

            public void Process()
            {
                t1.Add(t2);
            }
        }

        private class Test2
        {
            int x = 10, y = 20, z = 30;

            public void Add(Test2 class2)
            {
                x += class2.x;
                y += class2.y;
                z += class2.z;
            }
        }

        static void Main(string[] args)
        {
            //OOP vs DOD test

            Stopwatch stopwatch = new Stopwatch();

            int dataAmount = 1000000;

            Test1[] Test1Array = new Test1[dataAmount];

            int[] x1 = new int[dataAmount];
            int[] y1 = new int[dataAmount];
            int[] z1 = new int[dataAmount];

            int[] x2 = new int[dataAmount];
            int[] y2 = new int[dataAmount];
            int[] z2 = new int[dataAmount];

            //initialize data
            for (int i = 0; i < dataAmount; i++)
            {
                Test1Array[i] = new Test1();

                x1[i] = 10;
                y1[i] = 20;
                z1[i] = 30;

                x2[i] = 10;
                y2[i] = 20;
                z2[i] = 30;
            }

            //run test
            int iterationAmount = 20;
            double OOPTime = 0, DODTime = 0;

            Console.WriteLine("Press any key to start:\n");
            Console.ReadLine();

            Console.WriteLine("Test start:\n");

            for (int i = 0; i < iterationAmount; i++)
            {
                Console.WriteLine("Iteration: " + (i + 1));
                //OOP
                stopwatch.Start();
                for (int a = 0; a < dataAmount; a++)
                {
                    Test1Array[a].Process();
                }
                stopwatch.Stop();
                OOPTime += stopwatch.Elapsed.TotalMilliseconds;
                stopwatch.Reset();

                //DOD
                stopwatch.Start();
                for (int a = 0; a < dataAmount; a++)
                {
                    x1[a] += x2[a];
                    y1[a] += y2[a];
                    z1[a] += z2[a];
                }
                stopwatch.Stop();
                DODTime += stopwatch.Elapsed.TotalMilliseconds;
                stopwatch.Reset();
            }

            Console.WriteLine("\nTest Results:\n");

            Console.WriteLine("Averate OOP time: " + (OOPTime / iterationAmount));
            Console.WriteLine("Averate DOD time: " + (DODTime / iterationAmount));

            Console.ReadLine();
        }
    }
```