Computer Engineering. Academical year 2007-2008

Universidad Carlos III de Madrid

Modelling a RTS Planning Domain with Cost Conversion and Rewards

Author: Vidal Alcázar Saiz

Advisors: Daniel Borrajo Millán, Carlos Linares López

Abstract: This work analyzes an alternative to classical domain definition in PDDL of real life problems such as Real Time Strategy games, explores the possibilities of general problem solvers and innovates in its approach to gaming using automated planning.

Table of Contents

Figures Index

Index of Tables

1. INTRODUCTION

Almost since Artificial Intelligence was conceived as a field of research, games have been a fruitful testbed for many different approaches. Classical games are in most of the cases motivating problems and easy to represent in an adequate AI context, which has lead to great advancements in this field even getting computer players much stronger than professional-level human players. Some recent examples of how computers are being able to surpass humans are the latest challenges against chess world champions and the checkers problem being solved [Schaeffer et al. 317]. However, classical games are seldom extrapolated to real life problems and thus these advances have not had a significant impact in developing software able to cope with everyday problems that are trivial for humans.

Lately, with the advancement in computing, much more complex games have appeared and become one of the largest parts of the entertainment industry. While many types of games exist, those which consist in confronting two or more players in equal conditions as in classical gaming offer almost endless possibilities for AI research. A particular kind of game, real time strategy games, is particularly appealing as their characteristics make them similar to real life situations in which there is a notorious interest, such as military simulations. This, coupled to the fact that there is already a high level of expertise in RTS games (since they are very popular, up to generate a profit of millions of euros and create contests which emulate sport competitions at professional level), makes them a very attractive option.

So far the main problem is that RTS games have been created by private companies that are obviously motivated by economical reasons, so most of the games are closed source projects and lack real AI implementations, making them almost useless to researchers. To overcome this, several projects in the AI community have appeared that recreate RTS games, such as Stratagus [Stratagus]. Most of this projects are still in their earliest stages, however there is a platform created by Michael Buro and Tim Furtak at the University of Alberta which already offers great possibilities for researchers: Open Real Time Strategy (ORTS). This project has been conceived for scientific purposes and not as an attempt to recreate popular commercial RTS games, so there are several characteristics that difference it from these games, offering a favorable environment for AI experimentation.

Apart from being a GPL project, which already has plenty of advantages for academical research, ORTS recreates RTS domains based on a script created by the user, so it is not

limited to a set of games, letting the user design his objects and rules. It is complex enough to simulate commercial games with fidelity and has modules that solve common AI issues in these kind of games such as pathfinding. Besides, it is based on a server-client architecture, which avoids map hacking and allows players to connect their own customized clients, giving the possibility of creating autonomous computer clients, interfaces for humans to play or a hybrid client where the human is in control and the AI performs tasks depending on how the developer designed the client.

One of the main reasons for the popularity of ORTS among these kind of tools is the annual competition which is held by the developers in order to encourage participation and to promote their platform in the AI scene. This competition consists on four different domains which offer different challenges, including pathfinding, resource management,... Out of these four domains, the most complex and the one that is closer to RTS games is the third one, which basically is a fighting contest between two armies with resource management and a simple technology tree. Actually this is the base for almost all the existent RTS games, so it is by far the most interesting domain when trying to solve this kind of problems with AI techniques.

Among the different approaches of AI in games, search has been the most used in classical gaming. Generally the most intuitive way of solving classical games is searching the state space evaluating the different positions looking for the most advantageous move for the player. However, due to the characteristics of RTS games this approach is not valid for solving the aforementioned domain. In this situation, a step forward has to be taken and more complex AI techniques should be used. In this work we will use automated planning as the mental process a human player follows when trying to find an adequate strategy is similar to planning a sequence of actions that lead to a goal. For example, if a human player wants to attack an enemy base he will send the workers to get resources, build soldiers once the resources are gathered and attack with the the base, and if he wants to set another base to increase his resource income he will send a worker close to some resources, build a basic building and produce additional workers. These examples clearly illustrate how actions could be represented as operators that will compose the planning domain of and how using these operators and facts that represent relevant objects in the game goals can be achieved.

The main issue when solving a problem with automated planning is usually the definition of both the domain and the problems, being the domain the set of possible actions that can be taken and the problem the definition of the initial facts and the final goals for a particular

situation. In the third domain of the ORTS competition, specific planning domains have to be designed as its characteristics do not allow to represent every possible action as an operator and the winning condition as the goal of the problem, a computationally too hard problem for current planners. In order to do this abstraction, resource reutilization and other techniques will be used to overcome the main problems of the process of representation.

Planners receive as input a domain and a problem coded by using a specific syntax or language. Traditionally every planner used a self-defined language to represent operators, facts, etc... However as the need of comparing the efficiency of general problem planners and being able to try different planners for the same problem arose, common languages were created to standardize planning problem definition. The most known of this languages is probably STRIPS [Fikes and Nilsson, 1971], named after the original planner that accepted it as input. As time passed and due to the limitations of STRIPS, other languages where developed having it as a base. Nowadays, PDDL, which originally was devised to make the International Planning Competitions possible [McDermott, 98], is the standard for most planners. The version we will use is 2.1, the version used in the 3rd International Planning Competition, although not all its features will be necessary.

As mentioned above, PDDL 2.1 is accepted by many planners, but not all the features are supported by all the planners. In our case, we will use Metric-FF [Hoffmann, 2002] for all the experiments for two reasons: its performance for numeric problems is very good, as seen in the 3rd International Planning Competition; and it supports several key features essential for our work. As an extension other planners could have been tested, but for the purpose of this work a single planner is more than enough, as we are not focusing in performance.

As the last part of this introduction, we will show how the document will be structured:

- Chapter 2, State of the art, will be an overview about the questions that conform the base of the work and the current trends of research about them. It will include several subsections regarding gaming in AI and RTS gaming in particular, ORTS as the platform used for experimentation, automated planning and domain definition in PDDL and the latest achievements in this area by AI researchers.
- Chapter 3, Objectives, will state what are we going to achieve in this work and which results we expect to get. For this, the scope of the work will be clearly specified stating what will be done and what will be left for further works.
- Chapter 4, Definition in PDDL and Client Description, will describe the work done

going from a general point of view to a more detailed analysis. Throughout the description it will have into account the architecture of the system, a description of each module, a list of issues found and how they were solved, etc... It will include as well a brief user's manual for those not familiarized with the system.

- Chapter 5, User's Manual, will give a few guidelines to use ORTS and the planning domain, describing every step from installation to execution.
- Chapter 6, Results, will show the different results of the experimentation analyzing them in each relevant case. Graphics and comparisons between different cases and parameters will be displayed.
- Chapter 7, Conclusions, will consist on a short set of facts derived from all the previous experiments. We will try to conclude whether the results were the ones we were expecting and whether they can be seen as relevant from a scientific point of view.
- Chapter 8, Future Work, will state which are the possible works that may benefit from this project and how further experimentation in this context could be interesting for similar problems.

After the main content, three annexes will be included containing code which is too long to suit in one of the previous chapters:

- Annex A, Game 3 Blueprint, will include the blueprint that defines the third domain of the ORTS competition.
- Annex B, Domain Definition, will be the actual PDDL code that constitutes the domain used as solution.
- Annex C, Example of Problem in PDDL, will be one of the multiple problems used in the experimentations that will represent a real situation in ORTS.

2. STATE OF THE ART

In this section, an overview of how the different aspects of the problem will be given, with an emphasis on references to the latest articles related to the topic. This section will be composed of four subsections, each one describing a part of the work: the first one will deal with AI and gaming, studying classical approaches and analyzing the different challenges that modern RTS games pose to researchers; the second one will describe the platform which will be used to generate problems and to parametrize the domain, ORTS, discussing the differences that exist between it and commercial games and how this particularities allow simulations that were not possible until it was created; the third one will give an overview of the functioning of automated planning and the most extended language for defining planning domains, PDDL. Finally, a last section will focus on the things that have not been studied yet in the field and the relation to our work, so the purpose of the project can be clearly established.

2.1 Real Time Strategy games

This section is centered on real strategy games and the different relevant factors for our work related to them. The first subsection is an overview of this kind of games, introducing concepts and giving information about the common bases of these games. The second subsection comments the current state of AI in RTS games, mostly about commercial RTS games. The third one analyzes the challenges RTS games pose to the study of AI techniques, depicting common problems and the possible related solutions.

2.1.1 Overview of Real Time Strategy games

As per their definition, RTS games are strategy games distinctly not turn based. However, their conception has varied over the years and nowadays a much more specific concept of game is regarded as a standard RTS game. Basically, an RTS game involves resource gathering, base and technological development and tactical combat. The player has control over several units which are managed by issuing orders (or giving them commands) using a cursor to point locations and the keyboard and mouse buttons for the commands themselves, which frees the player from manually controlling

every unit at every single moment. Games involve two or more players who play the game simultaneously without waiting for their adversaries to act, which usually lead to dynamic and fast-paced games, at least when direct confrontations occur. The interface is generally a top-down perspective of the world with frames or bars that either display information or allow interaction with the units.

Although games of similar characteristics exist since the 80's, the foundations of the genre were set in 1992 by Dune II: The Building of a Dynasty developed by Westwood Studios. It was based on the same titled film by David Lynch (which was based itself on the series of novels written by Frank Herbert) and sold as a sequel of a previous game, though it was not directly related to it. This game introduced multiple concepts that would be used later by the trademarks of the genre:

- Resource gathering to fund unit construction and base/technology development: In this game, specific vehicles harvest spice distributed by some areas of the world and carry it back to the base, which adds up to the amount of available spice the player has and that can be spent in certain buildings. This of course varies from game to game generally depending on the universe they are based upon, but it has been kept like that as a staple for most of the successful RTS games.
- A technology tree in which buildings, upgrades and researches are prerequisites for more advanced units or abilities. A typical example is having to upgrade the main building before building a factory, which produces tanks that effectively counter earlier available units such as regular soldiers. Thus, reaching certain technology levels (often denominated as Tiers) before other players usually results in an important advantage.
- A mouse controlled cursor is the main way of playing. This is probably because of its availability, as the use of the mouse in the previous years was not extended, but it is nevertheless an innovative point of the game.
- Fog of war: this feature means that the world map is not initially revealed and it is shown as the player's units, characterized with a line of sight, scout the map. Besides, a part of the map is only updated when a unit is watching it, so enemy units are not displayed unless they are in the line of sight of an allied unit. This splits the map in three kind of areas, areas currently being observed, in which the

information is complete, areas already explored but not currently observed, in which static objects as buildings or terrain features are visible (usually in a grayed fashion) but no dynamical units are displayed, and the unexplored areas, usually completely black. This feature has a great impact on gameplay and greatly encourages scouting. In the figure below, the unit scouted the upper right part, which is now grayed because it is out of the line of sight, while the unexplored areas remain black..

Figure 1: Example of fog of war in Age of Empires II.

- Distinct factions with particularities between them, such as unique units, or game related advantages. In this case, three houses were available: the Atreides, the Harkonnen and the Ordos.
- A typical RTS interface, shown in Figure 2. Almost all the later RTS games have used a similar interface. It is generally composed by menu options regarding the game itself (Quit, Save,...), a minimap showing the world, a description for the selected units and an area with the possible actions and abilities of the unit.

Figure 2: Dune II's interface, a typical interface in the genre.

After the success of Dune II, a series of similar RTS games developed mainly by the aforementioned Westwood Studios and Blizzard Entertainment would hit the market selling literally millions of copies and popularizing the genre in the industry of entertainment. Some of the most known games are the Command & Conquer series, the Warcraft series, Starcraft, the Age of Empires series and Total Annihilation. The popularity of these games is such that international competitions with professional players are frequently held, being the most notorious case the Starcraft competitions that take place in South Korea [Starcraft Competitions]. Some games differentiated themselves by adding other elements from other genres or using features uniques to them, but up to date the standard RTS game is still greatly based on these generation of games.

From a strategical point of view, two contrary aspects have to be taken into account: macromanagement and micromanagement. Macromanagement is taking general decisions like attacking, retreating, setting a new base, researching a new technology,... while micromanagement refers to actually controlling the units in the game. Macromanagement is usually the strategical part of the game, while micromanagement is mainly dependent on the skill of the player at giving orders using the interface. This is a great concern in RTS games, as it is the main difference between human and computer players. While for humans developing a sound strategy is usually a simple thing,

computers players are unable to do so in an easy way, as we will analyze in the following section; on the other hand, micromanagement is clearly favorable to computers, which virtually have a cursor for every unit, while human players have to issue commands one by one. Actually at expert levels the skill in micromanagement is usually the decisive factor among human players, which had led to critics of the genre to say that there is little strategy involved and that RTS games have degenerated into a sort of clicking game where good reflexes are the most desirable skill. To reduce the importance of this factor, developers are including AI scripts and interface options like squad management, unit behavior definition, auto usable skills and queues of actions that allow the player to focus on the more strategical aspects of the game.

Usually most of the basic strategies are easily understood. Building, gathering, attacking, defending are all concepts that even players new to the genre can understand intuitively. However, several strategic concepts particular to the genre are often used which basically describe the main ways of playing. The following is a list of strategies that commonly appear in RTS games:

- **Scouting:** due to the fog of war, the information about the world is incomplete, so in order to get information the players have to scout the map with their units. This is essential for the game, as generally the best strategy is to counter the other player actions, like attacking undefended bases or retreating when an enemy force is heading towards an allied base. Expert players are known to be able to scout constantly while developing their own strategies. Besides many games are rock-paper-scissors games in the way that there are units designed to counter other units but that have a weakness against other units as well, encouraging scouting given the importance of knowing which kind of units the opponents are using. An interesting fact is that to counter the weakness of computer players, they cheat and have complete information (see next section), so scouting is not done by computer players.
- **Harassing:** another key concept at high level playing, harassing consists on making hit and run attacks with a few units against vital elements of an opponent, such as gatherers or key buildings. This diverts the opponent and makes him focus on the defense of this point usually forcing him to change his strategy, which hampers him as he cannot pay attention to other activities. A

classical example is harassing the workers at the main base while the main forces attacks another base. If not careful, the opponent will either fail at defending the attacked base or lose part of his gathering units.

- **Crippling the enemy economy:** a very common tactic in some RTS games is making attacks against the gatherers of an opponent even if it means losing the attacking units. If successful, this can be a winning move: such an attack is a direct hit on his economy as time and resources are necessary to rebuild them, interrupting the flow of resources and thus the amount of military units that can be produced, which highly compensates the units the attacker lost on his attack. This strategy was specially common in Starcraft, as it was a game greatly based on how fast units could be produced. As a side note, many of these attacks are done using transports which carry military units to the enemy base and eventually serve as a way of retreating, being called “drop” in this case.
- **Map control:** Resources in RTS games tend to be scattered around the map, so one of the key factors is claiming and defending the resources so it can be turned into units to beat the opponents. Furthermore, maps are designed with obstacles, different types of terrain, etc... so there are choke points where a clever use of the terrain can mean the difference between winning or losing an encounter. Playing accordingly to this is called map control, and it is one of the main factors that can decide a game. Having map control means that the player will receive more resources than his opponents, letting him win by attrition or simply by sheer power as he will be able to produce more units and research more advanced technologies. This shows the influence of the map in the way of playing the game; a typical example is a map in which there is only a way out of the main bases: if an opponent blocks that exit, it will be able to prevent the opponent from leaving the base and ensuring that he will not be attacked anywhere else on the whole map. Usually important spots on the map are kept by building expansions to gather resources more quickly and to act as a secondary bases or building offensive structures (turrets, cannons or mine fields come to mind as possible examples).
- **Rushing:** This strategy consist on building an offensive force as quickly as possible and attacking the opponent before he is ready to defend himself from enemy forces. This kind of attack is usually done in the earliest stages of the game using fast and offensive-oriented units and is a natural counter to teching. Using this strategy was at a time considered by the community as a cheap tactic

to win the game before the players set their strategies. However, rushing is indeed a risky tactic, as it involves sacrificing the economy and the development of the player looking for a direct win; f it fails, as it usually happens against turtling opponents, the rushing player is at a clear disadvantage and a counter attack usually settles the game for good. Being things like this, nowadays rushing is seen as a perfectly valid strategy, as it takes a certain skill to execute it at the right moment and successfully.

- **Turtling:** Turtling means playing the game defensively, usually building defensive structures at the main base and gathering a substantial force before taking any step. Turtling is the most common tactic among newcomers to the genre, though it is usually disregarded as a helpful strategy as it fails at map control and often means an eventual lose. The only interesting point of it is that it counters rushing naturally.
- **Teching:** Teaching means disregarding both attack and defense to employ the resources on technologies or advancements that will give the upper hand to the teching player. Building a factory to produce tanks earlier instead of creating a few soldiers to defend from a possible attack is a clear example of teching, as once the tanks are out they will theoretically win the battle against plain soldiers. This tactic is of course very dangerous, but in order to win it is something that must be done at several moments in the game so the force of the player does not become obsolete. Again and showing the rock paper scissors mentality of RTS games, rushing counters teching while teching counters turtling.
- **Battle micromanagement:** In most RTS games, units have health points and they are destroyed when these points are reduced to 0. However, a unit with a single health point left functions as well as a unit at full health, so one of the most important tactics when battling is to spread the enemy fire as evenly as possible among allied units while focusing fire on the opponent's weakest units. This is achieved by manually ordering the individual units to retreat when at low health so they can escape (and maybe lure attacking units) and to attack a single unit that is specially vulnerable. It is obvious that with this kind of functioning, 8 units at half health will defeat 4 units of the same kind at full health, thereby the importance of micromanagement. This is subject to plenty of factors, as the complexity of the games, with range, area of effect attacks, complex abilities, different kind of armors and many other features can overshadow this, but it is still kept as a rule of thumb.

All of these strategies are used in which is usually considered the standard RTS match: eliminating the opposing players. However, RTS games are not limited to this kind of setup. To spice up matches, in modern games players are given the option to choose the winning conditions. Some examples are killing a unique unit of an opponent, reaching a given amount of resources, holding a position or keeping an object during a set amount of time,... Besides, for single player game campaigns are created with a plot and a series of scenarios (scripted premade maps), though they often deviate from the basics of RTS games.

As an additional comment on strategies in standard RTS games, it is interesting to note that expertise of human players has allowed the design of the best ways to begin the game, as it happened with openings in chess. These openings are usually referred to as build orders, even though they include actions different from building or producing new units. Often there are variation of this build orders that lead to a particular strategy, such an early rush. For example, in a game in which four workers are initially available, a possible build order would be this one:

1. Worker 1 scouts enemy base.
2. Worker 2 builds barracks.
3. Workers 3 and 4 gather resources.
4. Build 3 additional workers. First two gather resources, the third builds a turret.
5. Once the barracks are built, two soldiers are produced. Both harass the enemy base.

Here the steps are numbered but certain actions, such as issuing orders to the initially available workers, are concurrent, so it is not a strict sequence of actions.

2.1.2 Artificial Intelligence in RTS games

Traditionally, AI in computer gaming has been conceived pragmatically; that is, it is not the research of AI what really matters but rather getting software capable of either beating humans or providing a good gaming experience. That is the reason why AI in

games is generally understood as any technique that leads to an appearance of intelligence. Some of these techniques are drawn upon classical AI, but their origin may be from any other field. Besides most of the games are too complex to be solved using a single specific technique, so game AI is an unspecific term. A classical example is computer chess, which is often roughly based on a minimax algorithm with pruning along with many other techniques like opening and endgame databases and transposition tables [Computer chess]. However, commercial videogames are generally not as AI friendly as classical games and seldom use real AI implementations; for instance, only a handful first person shooter games, which are one of the most popular genres, use real AI techniques for their computer players, as done in FEAR, where limited automated planning is used [Orkin, 2006].

Due to this conception of game AI and the many kind of games that exist, developing computer players for commercial games is usually regarded as a doubtful method of contributing to AI research. In the particular case of RTS games, only pathfinding techniques are extensively used. Actually, RTS computer players are nowadays considerably weaker than human players and an average player can beat them with no difficulty. In most of the cases, computer players consist of a set of rules that represent the transitions of a finite state machine designed by the developer. This leads to a rigid behavior in which the same strategy is used and there is no adaptation at all, which makes up for a very predictable player. Funny enough, it is a proven fact that in order to try to pose a challenge to human players computer players in RTS cheat, as they always have complete information and usually have advantages over the human players in the hardest AI levels, such as getting more resources or having tougher units --this is not told to the player, though.

Fortunately, some research dealing with AI has been done lately in RTS games, mainly from AI research groups instead of game developing companies. With the upcoming of new tools like Wargus and ORTS, researchers have found new ways of working with RTS games. An example of this trend of research is Aha's work [Aha, Molineaux and Ponsen, 05], in which he studies the use of case-based plan selection to design a client that fully plays a standard RTS game simulated with Wargus with a certain degree of success. This client is compared to model computer players with fixed strategies and with a genetic algorithm developed by Ponsen and Spronck, another way of trying to solve the problem through AI techniques. As for automated planning, research is still in

its earliest stages, with few publications focusing on this issue. To show how little has been done up to now on the subject we will comment the most recent publication with impact, which has been the one released by Chan and Fern in the 2007 ICAPS [Chan et al., 07]. In this paper the problem to solve has been relaxed to three resource gathering problems: Gather a certain amount of gold, produce a certain number of workers (which contribute to gathering resources) and produce a certain number of soldiers (which do not contribute to gathering resources). The domain includes a mine, a base, a number of initial workers and the possibility to build additional workers or soldiers by spending gold. The main difficulty is finding the number of workers that must be build to minimize the time, and the solution proposed by the authors is to solve using planning as many problems as possible numbers of workers within a reasonable range in real time, which adds very little to a real RTS computer player. Actually, the most interesting point for this work is that it uses PDDL 2,1, the same language that will be used for our domain. Another work worth of mention is the one published by King and Atkin in 2002 [King, Atkin and Westbrook, 02], in which they presented a general purpose, semi-declarative agent control language known as Tapir that extends and enhances the Hierarchical Agent Control (HAC) architecture with the purpose of using planning in multiagent domains, but apart from that, no other RTS-related researches have meant an advancement in AI.

2.1.3 Challenges of RTS games

To understand why RTS games are so practical for real-time AI research, one must analyze the different problems that arise when designing computer players. Most of these problems come from the existing differences between classical and RTS games and thus remain unsolved given the little advance in AI research in the genre. The main challenges that RTS games offer are the following:

- **Real Time reasoning:** The staple of the genre. Computer players have to take decisions as the situation change, forcing them to be not only fast enough to come with coherent actions in a limited amount of time, something that can happen in fast paced non real time games, but also to adapt and replan following the time flow. This is an extremely complex problem and generally demands

discretization of time and reusing previous computations.

- **Complex worlds:** Worlds in RTS games, though mostly being based on a grid, are generally too large and thus classical interaction with it as done in board games is not possible. Besides the terrain usually has distinct features that heavily influence the game, so computers players not only have to take into account distance and obstacles but also the characteristics of the terrain. Abstraction is compulsory in most cases, using graphs over connected areas and chokepoints with critical points such as resources and bases having a key role.
- **Uncertainty:** RTS games cope in most of the cases with incomplete information because of fog of war or other mechanics and often the game is stochastic as an action may have several possible outcomes depending on how the game is designed. This means that players must act under uncertainty, a very complex task for computer players implying both information gathering and formulating plausible hypothesis.
- **Learning:** RTS games are very dynamic as players have at their disposal different strategies that can be changed throughout the game. Opponent modeling and learning is the main factor that allows a player to adapt and react accordingly to his opponent's actions, something humans are naturally adept at but computer players are not.
- **Resource and technology management:** To win a game there are important problems that must be solved prior to attacking and defending to achieve the winning condition. These problems are usually how to manage the economy and the technology development. These key problems are in fact soft goals that lead once achieved to the conditions that allow the victory, adding complexity when defining the objectives due to the balance that has to be kept between the military issues and these two aspects.
- **Collaboration:** RTS games contain not only multiple units per player, but also are often multiplayer games in which teams of players are formed. This means that there are two issues regarding collaboration: units of the player must act in a coordinated fashion in order to achieve complex goals that require a certain degree of collaboration and different players in the same team must communicate and help each other both defensively and offensively to beat other teams.

The sum of all these problems make RTS games a challenging (if complex) environment in which test new approaches from an AI point of view. This, added to the fact that RTS games encourage competitive playing, is the main reason why they are appealing to the academical community as a testbed besides their practical applications.

2.2 ORTS

ORTS stands for Open Real Time Strategy and is a free RTS game engine which allows users to define RTS games in form of scripts and to connect arbitrary game client software ranging from 3D graphical user interfaces to distributed AI systems. ORTS has been throughly described by its creator, Michael Buro [Buro, 02], so in this section this engine will be described focusing on its features, the way it works and how it is a useful tool for AI research.

2.2.1 Features of ORTS

Conceived in 2001 by Michael Buro and developed by Tim Furtak and several other contributors from the University of Alberta, ORTS stands as an useful tool for AI research by providing an environment in which design intelligent players for RTS games. Its main function is to be able to recreate RTS games with a complexity similar to that of the current commercial RTS games in terms of game design. It implements some features that provide an experience similar to an actual game with the advantage that it has been designed for research, so experiments with AI techniques can be done under realistic conditions and without losing the gaming experience. The main features of ORTS are the following:

- **Real Time environment:** ORTS is completely real time, the interactions between players are asynchronous. Contrary to regular RTS games, in which a lagging player causes the game to halt until it resumes activity or finally disconnects, the game in ORTS is not interrupted and all the non-valid actions sent by the player will just be ignored as they are no longer possible, taking a more realistic approach. No discretization or turns exist, but this can be simulated using game scripting.

- **Terrain:** Terrain is completely customizable. Several levels or layers can be created. Usually three levels coexist, plain ground, elevated ground and air, but games are by no means limited to this. Obstacles, ramps, plateaus, impassable terrain, water portions,... are all possible.
- **Action computation:** Players compute actions for a set of objects at their command. The computational model can range from local to global with respect to the objects in order to reflect given command hierarchies and different levels of physical restrictions imposed by the world.
- **Server-client model:** To play a game, a machine working as a server is needed. Clients connect to this machine receiving views of the world (the state of the game having into account which objects are visible to the player and which objects are not) and sending actions back to the server, where the effects of the actions are solved.
- **Complex interaction of units:** Units are not limited to basic actions such as moving and attacking. Gathering, colliding or recreating complex effects that simulate the so called skills of the units that appear in commercial games are features of ORTS.
- **Flexible game specification:** ORTS is a generic RTS game programming environment. This means that the actual game played when using the ORTS system is not fixed but scripted. A script is used to define the whole game: types of terrains, unit properties such as size, sight range and maximum speed, unit actions (including complex skills), scores, resources, winning conditions, team configuration, etc... This allows both to create specific situations for formal experimentation and appealing games for human players.
- **Client customization:** Clients connect to the server via sockets using an open protocol. Apart from that, clients are not restricted in any way, meaning that the player can use any technology, create his own interfaces, add AI modules,... This means that clients can implement anything from fancy graphics to autonomous or auxiliary clients that help the player to play. For example a player can use a 3D interface with customized buttons for the game, while other uses a 2D interface with AI modules that help him with low level tasks such as gathering resources, while a third can implement an autonomous client that plays on his own with no need for interface as there is no human interaction.
- **Graphical user interface:** Even though players are free to design their clients, a

fairly complete 3D interface is already available, along with an older 2D interface. This interface, while not as fancy as the ones used in commercial RTS games, is appealing enough to give the look and feel of a real game apart from being very well integrated with the game definition system.

Figure 3.: ORTS's 3D Graphic User Interface.

- **Active community:** ORTS has been developed by researchers interested in spreading the use of their tool as much as possible. Therefore, annual competitions are held where the participant explain and share their clients and works about the state of ORTS are regularly published. Thanks to this, part of the community has been interested in it, participating and even collaborating in the development of ORTS.
- **Low Level AI modules:** Thanks to the experience accumulated after the

competitions, several low level AI modules based on real clients have been released and are publicly available for other developers. Some modules include resource gathering, squad formation, pursuing, attacking an area,... This eases the task of developing a fully functional client and allows to focus on other aspects of the game.

2.2.2 ORTS versus commercial RTS games

Initially ORTS was created because of how inappropriate commercial RTS games are for AI research. This is not because the industry opposes to using their games for AI research or because they try to close their projects as much as possible (actually this is not at all true, several RTS games provide map editors and can be modded by the playing communitiy), it is just that the philosophy of the game developing enterprises is not creating the strongest computer player, as this would frustrate the buyers, but to provide computer players that enhance the replayability of the game. That, added to the limitations this kind of projects have (release date, priority of graphics over AI behavior for marketing reasons, budget,...) is what made of these games a not very useful tool for the academic community.

Trying to solve this situation, ORTS came up with a few innovative characteristics that separates it from commercial RTS game and that constitute its foundations. The main differences will be analyzed one by one and then resumed in a comparative table.

- **Cost:** Commercial RTS games are produced for profit therefore having an associate prize to them. On the other hand, ORTS is released under the GNU Public License [GPL], which means that anyone can download the source code at no cost in order to learn how the system works and to contribute to the project by submitting bug fixes and adding new features. It also means that projects that incorporate ORTS code need to release their source code as well. Being free anybody can try it and work on it without economical inversion, which greatly helps to popularize it.
- **Game design:** Each commercial RTS game is a single game and stays like that

with only a few variations. ORTS is not a game, it is an engine which works with scripted games, so the number of games that can be created and played are unlimited. Besides researchers can modify and customize both the tool and the games to their needs, which leads to a broader range of possible experiments.

- **Hack-free environment:** As we mentioned earlier computer players in commercial RTS games cheat in order to pose a challenge to human player, so if a competition was held using AI clients the possibilities of cheating happening are high. However this is not possible in ORTS as the game server maintains the entire world state and sends only visible information to players which connect from remote machines and send back actions that are executed on the server side and never on the client.
- **Custom clients:** Commercial RTS games are closed projects and thus the clients cannot be modified at all, which restricts the players. In ORTS the only thing clients must comply to is the communication with the server. Other than that they can be implemented in any way, creating and modifying interfaces and AI modules at will. This also allows creating remote or distribute clients in which more computational power can be dedicated to AI as opposed to commercial RTS games, which use most of the CPU time rendering the 3D graphics and neglect AI performance.

Feature	Current Commercial RTS Games	ORTS
Cost	~US$ 50	Free
License	Closed Software, no code changes possible	GPL
Topology	Peer-to-Peer: The entire game simulation is run on each peer node	Server-Client: Server simulates world and sends local views to clients
Communication protocol	Closed	Open, specification publicly available
Game Complexity	High for commercial reasons	User defined
Game Specification	Fixed, though some games offer map editors and modding tools	Flexible, games are scripted
GUI	Nice looking but not modifiable by the user	Either the standard one or one created by the user
CPU requirements	High due to the graphics	Depends on the client
Network requirements	Low: Only clicks and keystrokes are sent	Medium: Views and actions are sent back and forth
Unit control	Essentially sequential: clicks and keystrokes + predefined low-level unit behaviour	Parallel: one command per unit per simulation frame and atomic actions + no predefined low-level behaviour
AI	AI code for all players runs on all peer nodes	AI is local to each client
Remote AI	Not supported	Depends on the client

Table 1: Differences between commercial RTS games and ORTS

2.2.3 Functioning of ORTS

Now that the main features that define ORTS have been introduced the only thing left is to give a bit of insight about the inner functioning of ORTS. The three most relevant aspects are vision, game scripting and networking.

- **Vision:** ORTS is a server-client based platform in which clients do not have free access to the game state. Rather every frame the server computes for every player which objects he sees at that moment and sends their information to that player, which will answer with a set of actions associated to those objects. The determination of what is seen is based on which regions of the world are currently visible to a client. This is the union of what can be seen by all objects currently under the client's or an ally's control. The objects can see only what is within their defined range of vision, taking into account obstructions caused by terrain features such as plateaus, but not obstruction by other units.

  The tile–based nature of the ORTS world naturally lends itself to describing an object's visual field in terms of the regions visible from the tiles it is occupying. Regardless of the criteria used to determine visibility, the highly static nature of terrain obstructions allow the results of those visibility computations to be reused for any object looking out from that tile afterward. This ability to avoid re-computation for minor changes in position greatly mitigates any coarseness in the visual description. Moreover, the level of coarseness may be adjusted by changing the number of tiles used to describe the world.
  Unlike an incremental solution, where the change in position of each unit must be taken into account when determining the visible tiles, ORTS generates the entire view independent of the last cycle. As a result, it is unaffected by changes in unit positions or by large portions of the view being quickly hidden and revealed as would be the case when moving troops through highly obstructed terrain. Because the number of interactions (collisions, enemy encounters, etc.) increases with the number of moving objects, computations have been optimized to reduce the resources needed in this worst case. The execution time of the vision computation is linear in the number of objects N, the number of tiles T, and the number of players P, for a time complexity of $O(N+P \cdot T)$.
- **Game Scripting:** The scripting engine performs the interesting game-specific logic and allows for flexible game definitions and client interfaces. High performance tasks common across a large number of possible RTS games such as accurate unit motion and unit vision in the presence of terrain are handled

separately by the server. Everything else, such as weapons and special abilities, is scripted as part of the game definition. The scripting language was designed to provide a convenient way to define unit types and actions. Unit definitions are given in the form of blueprints which list named (usually) integer attributes and actions. The blueprints use a loose multiple inheritance system, allowing them to be combined and nested. New unit types can easily be constructed from functional components. In the client the object creation system is used to create GUI widgets such as buttons and status windows.

When the client receives the game description, which includes unit blueprints, it can locally extend those blueprints by adding extra attributes, sub-objects, or actions. The client can use this functionality to write wrappers for complex actions, add simple background AI, or add event handlers for when an attribute changes. By adding a 3D model sub-object the client specifies how an object will be represented in the world and allows for context sensitive animations. The client extends the scripting language functionality by registering special functions that allow access to OpenGL commands for drawing bitmaps and then simply calling those functions within the script. Mouse and keyboard events received by the client are transferred to the script by calling the actions of a special root GUI object, passing the event information as parameters. This object recursively calls the interface actions of its children until it is handled. Since the scripting language was designed to be able to perform reasonably complicated game logic, eventually errors will occur that cannot be simply debugged by inspection. At this point it becomes invaluable to have some way for the script to write information to the console or to inspect the current state. As a compiler option the interpreter can maintain a stack trace of the current execution with a printout of line numbers and the statement being evaluated at each step. This trace is automatically printed when a trappable error occurs in the script, and can be printed manually from inside a debugger such as gdb.

Because it is relatively trivial to extend the scripting language by adding external C functions it is tempting to do so whenever additional functionality is needed. This can quickly lead to numerous special purpose functions and bloated syntax. Consider the problem of implementing an STL-like vector container. One option is to try to force the language to do something it was never intended to, perhaps by implementing a complicated linked list. Another is to add a C function that returns a pointer to an actual STL vector, with additional functions for adding to it, sorting, etc. To help make the scripting language extensible, objects in the

script are all derived from a common base class, with game objects being only one possible option. To address the previous concern, wrappers have been written for STL vectors and sets, allowing them to be created in the same manner as classes described by blueprints. By modifying the new objects' incremental update functions the container can be used as a sub-object within game units. The graphical client uses derived classes for 3d models and particle systems to attach these things to objects in the game. Script actions take generic script variables as parameters, which may be object pointers, integers, or something else. The next two figures show how the scripting language works, being the first a simple example of a standard unit, in this case a marine, and the second a more complex definition of a homing missile:

```
blueprint marine
    is generic_unit          # include a set of common attributes and default values
    class kevlar armor       # create a sub-object of type "kevlar" named "armor"
    class rifle weapon       # the rifle sub-object has already been defined and
                             # has a "shoot" action defined

    # make zcat constant and assign it the enum ON_LAND
    setf zcat ON_LAND
    setf max_hp 100
    set hp 100
    setf sight 6
    setf radius 5
    set max_speed 3
end
```

Figure 4: Marine blueprint

```
blueprint missile
    has core_attr
    has movement
    setf shape CIRCLE
    setf radius 3
    setf max_speed 20
    set speed 0
    setf zcat IN_AIR
    setf targetable 0
    setf invincible 1
    var hidden det_range 3
    var hidden blast_range 20
    var hidden min_dmg 200
    var hidden max_dmg 350
    var collides 0           # set the collision mask to ignore all other objects
```

```
        # this action takes one object as a parameter, no integer variables, and no hidden variables.
        action track_obj(targ;;) {
                gob e;
                int dmg, damage_type;
                damage_type = this.damage_type;
                if (targ.targetable < 1) break;
I               f (distance(this,targ) <= this.det_range) {
                        # "-1" -> not owned by any player
                        e = create("explosion", -1);
                        e.x = this.x;
                        e.y = this.y;
                        e.zcat = targ.zcat;
                        e.radius = this.blast_range;
                        e.damage_type = EXPLOSIVE;
                        # add the "boom" action to the action queue and
                        # execute it sometime in the current tick
                        e.boom(;this.min_dmg, this.max_dmg, 0;) in 0;
                        # mark the missile as dead - it can still act,
                        # but cannot queue any more actions, and will
                        # be deleted at the end of the current tick
                        kill(this);
                } else {
                        # move events are handled after script actions.
                        # the object isn't teleported, it walks/flies to
                        # the target location at its speed
                        move(this; targ.x, targ.y);
                        # accelerate the missile - applies to above command
                        this.speed += 4;
                        if (this.speed > this.max_speed)
                        this.speed = this.max_speed;
                        # execute this action again in 1 tick
                        # without "in 1" action would be called immediately
                        this.track_obj(targ;;) in 1;
                }
        }

    end
```

Figure 5: Homing missile blueprint

- **Networking:** In each cycle the server sends the state of the world to each client as they perceive it and the client responds by sending a list of actions for the objects it has control of. As the world is explored and portions of the map are revealed, the server sends a list of the newly visible tiles along with a description

of their topography (height, ground type, whether or not the tile slopes in a particular direction). Objects are entirely described by the index of the blueprint used to create the object, and a vector of their current attribute values. Subsequent viewings of the same object (if the object has not been lost from sight) are given in terms of the attribute changes from the last frame.
With the increasing speed of network communication, such data rates are within the limits of current high–speed Internet connections. By applying a moderate amount of compression to the client's view prior to sending it, the necessary throughput is greatly reduced to acceptable levels for even large multi–player games. Experimental results using ORTS suggest that the main communication bottleneck when using server–side simulation and high–speed connections such as cable–modems or DSL is lag rather than data throughput. Lag is induced by data transfer over networks and by associated computational overhead such as message compression/inflation and updating data structures on both communication ends. For the sake of simplicity the first ORTS implementation totally ignored network lag and indeed used blocking TCP I/O on both the server and client side.

2.3 Automated Planning and PDDL

Among all the different branches in AI, planning has proven to be a relevant approach to many different problems in which a certain degree of autonomy must be attained. Computer players RTS games must be completely autonomous intelligent agents which must reach one or more goals through strategies or action sequences, hence the adequateness of this technique for our work. In this section, automated planning and its most popular language for domain and problem definition, PDDL, will be described.

2.3.1 Overview of Automated Planning

Planning in Artificial Intelligence can be defined as decision making about the actions to be taken. This sequence of actions is the strategy that will be followed to achieve a

given goal defined in the problem and is called plan. Impediments for the success of AI in producing genuinely intelligent beings are related to perceiving and representing knowledge concerning the world. The real world is very complicated in all its physical and geometric as well as social aspects, and representing all the knowledge required by an intelligent being may be too inflexible and complicated by the logical and symbolical means almost exclusively used in artificial intelligence and in planning. AI planning (like knowledge representation and learning techniques in AI in general) are currently best applicable in restricted domains in which it is easy to identify what the atomic facts are and to exactly describe how the world behaves. These properties are best fulfilled by systems that are completely man-made, or systems in which planning can view the world at a sufficiently abstract level.

Research that has lead to current AI planning started in the 1960's in the form of programs that tried to simulate problem solving abilities of human beings. One of the first programs of this kind was the General Problem Solver (GPS) by Newell and Simon [Ernst et al., 1969]. GPS performed state space search guided by estimated differences between the current state and the goal states.

At the end of 1960's Green proposed the use of theorem-provers for constructing plans [Green, 1969]. However, because of the immaturity of theorem-proving techniques at that time, this approach was soon mostly abandoned in favor of specialized planning algorithms. There was theoretically oriented work on deductive planning which used different kinds of modal and dynamic logics [Rosenschein, 1981] but these works had little impact on the development of efficient planning algorithms. Deductive and logic-based approaches to planning gained popularity again only at the end of the 1990's as a consequence of the development of more sophisticated programs for the satisfiability problem of the classical propositional logic [Kautz and Selman, 1996].

One of the most well known early planning systems is the STRIPS planner from the beginning of the 1970's [Fikes and Nilsson, 1971]. The states in STRIPS are sets of formula, and the operators change these state descriptions by adding and deleting formula in the sets. Heuristics similar to the ones used in the GPS system were used in guiding the search. The definition of operators, with a precondition as well as add and delete lists, corresponding to the facts that respectively become true and false, and the

associated terminology, is still in common use, although restricted to atomic facts, that is, the add list is simply the set of state variables that the action makes true, and the delete list similarly consists of the state variables that become false.

Starting in the mid 1970's the dominating approach to domain-independent planning was the so-called partial-order [Sacerdoti, 1975], which remained popular until the mid-1990's and the introduction of the Graphplan planner [Blum and Furst, 1997] which started the shift away from partial-order planning to types of algorithms that had earlier been considered infeasible, even the then-notorious total-order planners. The basic idea of partial-order planning is that a plan is incrementally constructed starting from the initial state and the goals, by either adding an action to the plan so that one of the open goals or operator preconditions is fulfilled, or adding an ordering constraint on operators already in the plan in order to resolve a potential conflict between them.

In parallel to partial-order planning, the notion of hierarchical planning emerged [Sacerdoti, 1974], and it has been deployed in many real-world applications. The idea in hierarchical planning is that the problem description imposes a structure on solutions and restricts the number of choices the planning algorithm has to make. A hierarchical plan consists of a main task which is decomposed to smaller tasks which are recursively solved. For each task there is a choice between solution methods. The less choice there is, the more efficiently the problem is solved. Furthermore, many hierarchical planners allow the embedding of problem-specific heuristics and problem-solvers to further speed up planning.

Nowadays and thanks to the success of different trends in automated planning, many different approaches coexist. Generally the modeling and algorithms used by the planner are the differencing characteristics between them, although most of them can be classified not only by their design but also by the kind of problems they are able to solve. This set of problems establishes the division between different branches of automated planning and their characteristics are usually the challenges planners must cope with. Several important aspects in planning problems are the following:

- **Determinism:** In the simplest form of planning the state of the world at any

moment is unambiguously determined by the initial state of the world and the sequence of actions that have been taken, making the world deterministic. The assumption of a deterministic world holds in many simple planning problems. However, when the world is modeled in more detail and more realistically, the assumption does not hold any more: the plans have to take into account events that take place independently of the actions and also the possibility that the effects of an action are not the same every time the action is taken, even when the world appears to be the same. Nondeterminism comes from two different sources: first, any feasible model of the world is very incomplete, and events that are possible as far as our beliefs are concerned can be viewed as nondeterministic; second, many actions themselves are by their nature nondeterministic, either intentionally or unintentionally, like the actions in which the outcome depends on a probability.

- **Observability:** In deterministic planning problems with one initial state there is no need to use observations as the state of the world after taking certain actions can be completely predicted. Hence a plan, if one exists, is simply a sequence of actions. However, when the actions or the environment can be nondeterministic, or when the initial state is not exactly known, it is not in general possible to reach the goals by using one fixed sequence of actions. The actions have to depend on the observations. There are two possibilities: first, planning could be interleaved with plan execution: only one action is chosen at a time, it is executed, and based on the observations the next action is chosen, and so on; second, a complete plan is generated, covering all possible events that can happen, and it is executed, without further planning during execution. These two approaches are computationally very close, but the first approach does not require explicitly representing all the action sequences that might be needed, it only has to find a guarantee that such action sequences exist. The possible observations have a strong impact on how exactly the actual state of the world can be determined: the more facts can be observed, the more precisely the current state of the world can be determined, and the better the most appropriate action can be chosen. If there is a lot of uncertainty concerning the current state of the world it may be impossible to choose an appropriate action. If the current state can always be determined uniquely we have full observability. If the current state cannot be determined uniquely we have partial observability, and planning algorithms are forced to consider sets of possible current states.

- **Time:** Most work on planning uses discrete time and actions of unit duration. This means that all changes caused by an action at time point t are visible at time point t+1. So changes in the world take only one unit of time, and what happens between two time points is not further analyzed. More complicated models of time and change are possible, even though most types of problems can be analyzed in terms of discrete time by making the unit duration sufficiently small.
- **Control information and plan structure:** In the basic planning problem a plan is to be synthesized based on a generic description of how the actions affect the world. There may be, however, further control information that may affect the planning process and the plans that are produced. In hierarchical planning, for example, information on the structure of the possible plans is given in the form of a hierarchical task network, and the plans that are produced must conform to this structure, which may substantially improve the efficiency of planning.
- **Plan quality:** The purpose of a plan is often just to reach one of the predefined goal states, and plans are judged only with respect to the satisfaction of this property. However, actions may have differing costs and durations, and plans could be assessed in terms of their time consumption or cost. The classical measure for quality plan has been plan length; nevertheless, modern planners often support fluents and time flow, so the research is steadily turning into getting better plans based on minimizing or maximizing a given variable.

2.3.2 Bases of Heuristic Planning

The main characteristic that defines planning is the search done through a set of states, called the state space. This state space is the combination of all the possible values of the variables that define the problem. States and the connections between them can be represented in several ways, mostly related to finite automata theory. The most popular ways of representing the state-space is using transition systems, in which graphs whose nodes are the different states and whose arcs are the transitions between them are used (or alternatively incidence matrix-based representations); propositional logic, in which a relation of truth must be proven between the initial state and the goal using propositional formulas as equivalent to state transitions; and succinct transition systems, in which the

representation of the states of a transition system is done using valuations of state variables instead of enumerating them as in classical transition systems.

Searching through the space is done using a given algorithm. The simplest possible planning algorithm generates all states (valuations of the state variables), constructs the transition graph, and then finds a path from the initial state to a goal state. The plan is then simply the sequence of operators corresponding to the edges on the shortest path from the initial state to a goal state. However, this algorithm is not feasible when the number of state variables is high, as the number of states is 2^n where n is the number of variables, and an informed algorithm, this is, an algorithm that uses an heuristic function to evaluate how far from the goal is the evaluated state, must be used. This form of plan search can be easiest viewed as the application of general-purpose search algorithms that can be employed in solving a wide range of search problems. The best known heuristic search algorithms are A*, IDA* and their variants [Hart et al., 1968; Pearl, 1984; Korf, 1985] which can be used in finding shortest plans or plans that are guaranteed to be close to the shortest ones as long as the heuristic function evaluates accurately for the given problem.

There are two main possibilities to find a path from the initial state to a goal state: traverse the transition graph forwards starting from the initial state, or traverse it backwards starting from the goal states. The main difference between these possibilities is that there may be several goal states (and one state may have several predecessor states with respect to one operator) but only one initial state: in forward traversal we repeatedly compute the unique successor state of the current state, whereas with backward traversal we are forced to keep track of a possibly very high number of possible predecessor states of the goal states. Backward search is slightly more complicated to implement but it allows to simultaneously consider several paths leading to a goal state.

The most important heuristics are estimates of distances between states. The distance is the minimum number of operators needed for reaching a state from another state. When search proceeds forwards by progression starting from the initial state, we estimate the distance between the current state and the set of goal states. When search proceeds backwards by regression starting from the goal states, we estimate the distance between

the initial state and the current set of goal states as computed by regression. There are many kind of heuristics, being some of the more popular for general problems admissible max heuristic, inadmissible max heuristic and relaxed plan heuristic.

2.3.3 PDDL 2.1

Every two years since 1998, the International Planning Competition are held to compare state of the art planners and encourage further research in the area. Until the creation of the IPC, planners had their own syntax to define domains and problems (even though many of them accepted STRIPS or languages based on it). To created a base of planning problems that could serve as a benchmark for comparisons between the planners, PDDL was created. PDDL stands for Planning Domain Definition Language and originally contained STRIPS, ADL and some other minor features, and had a great impact in the development of new planning system as many of them adapted themselves to it to avoid being left out of the most popular events in the area. Basically it departed from STRIPS, being a action-centered-language in which all the operators are actions that can have pre- and post-conditions. The syntax is similar to the one used in Lisp, and separates domain specifications (mainly composed by actions) and problems (composed by facts, goals and an initial state), which allows for different problems to be created for the same domain.

A significant change was made in the third IPC held in 2002. Due to the limitations classical planning problems had and in order to orientate the research towards a more practical approach, numeric and time reasoning were introduced. Besides, these considerations were already part of several planners, but until then these particular features could not be tested. Special attention was given to ensure that the language could express physical properties of the world instead of forcing domain designers to adapt to the constraints of the language and to make it backwards compatible with previous versions of the language.

PDDL 2.1 is structured in levels of increasing expressive power. STRIPS is level 1, the

numeric extensions are level 2, discretized durative actions are level 3, continuous durative actions are level 4 and a set of additional features for modeling spontaneous events are level 5. In the competition and based on the state of the art planners capability, only levels 1, 2 and 3 were used, leaving the more complicated levels as guidelines for further development in planning. This gives the opportunity for a planning system to reject attempts to plan with domains that make use of more advanced features of the language than the planner can handle. Syntax checking tools can be used to confirm that the requirements flags are correctly set for a domain and that the types and other features of the language are correctly employed.

The parameterization of actions depends on the use of variables that stand for terms of the problem instance as they are instantiated to objects from a specific problem instance when an action is grounded for application. The pre- and post-conditions of actions are expressed as logical propositions constructed from predicates and argument terms (objects from a problem instance) and logical connectives. PDDL 2.1 also includes the ability to express a type structure for the objects in a domain, typing the parameters that appear in actions and constraining the types of arguments to predicates, actions with negative preconditions and conditional effects and the use of quantification in expressing both pre- and post-conditions. These extensions are essentially those proposed as ADL [Pednault, 1989]. Something PDDL 2.1 lacks though is the use of structures in domain definitions, which prevents using hierarchical planners like SHOP [Nau, Cao, Lotem and Muñoz-Ávila, 1999].

The original PDDL provides support for numbers by allowing numeric quantities to be assigned and updated. However it was neither extensively used nor standardized, so one of the first decisions made in the development of PDDL 2.1 was to propose a definitive syntax for the expression of numeric fluents. Numeric expressions are constructed, using arithmetic operators, from primitive numeric expressions, which are values associated with tuples of domain objects by domain functions. As presented below, the functions “capacity” and “amount” associate the jug objects with numeric values corresponding to their capacity and current contents respectively. A prefix syntax for all arithmetic operators, including comparison predicates, is used in order to simplify parsing. Conditions on numeric expressions are always comparisons between pairs of numeric expressions. Effects can make use of a selection of assignment operations in order to update the values of primitive numeric expressions. These include direct

assignment and relative assignments (such as increase and decrease). Numbers are not distinguished in their possible roles, so values can represent, for example, quantities of resources, accumulating utility, indexes or counters.

```
(define (domain jug-pouring)
   (:requirements :typing :fluents)
   (:types jug)

   (:functions
   (amount ?j - jug)
   (capacity ?j – jug))

   (:action pour
            :parameters      (?jug1 ?jug2 - jug)
            :precondition    (>= (- (capacity ?jug2) (amount ?jug2)) (amount ?jug1))
            :effect          (and (assign (amount ?jug1) 0)
                                  (increase (amount ?jug2) (amount ?jug1)))
)
```

Figure 6: Use of numeric expressions in PDDL 2.1

Numeric expressions are not allowed to appear as terms in the language (that is, as arguments to predicates or values of action parameters). Numbers do not exist as unique and independent objects in the world, but only as values of attributes of objects. Models are object-oriented in the sense that all actions can be seen as methods that apply to the objects given as their parameters. This object-oriented approach is reflected in the way in which numbers are manipulated only through their relationships with the objects that are identified and named in the initial state. Besides, many current planning approaches rely on being able to instantiate action schemas prior to planning, and this is only feasible if there is a finite number of action instances. The branching of the planner's search space, at choice points corresponding to action selection, is therefore always over finite ranges. The use of numeric fluent variables conflicts with this because they could occur as arguments to any predicate and would not define finite ranges.

The adoption of a stable numeric extension to the PDDL core allowed to introduce a further extension into PDDL2.1, namely a new (optional) field within the specification of problems: a plan metric. Plan metrics specify, for the benefit of the planner, the basis

on which a plan will be evaluated for a particular problem. The same initial and goal states might yield entirely different optimal plans given different plan metrics provided the planner supports this feature. The value total-time can be used to refer to the temporal span of the entire plan. Other values must all be built from primitive numeric expressions defined within a domain and manipulated by the actions of the domain. As a consequence, plan metrics can only express non-temporal metrics in PDDL2.1 domains using numeric expressions. Any arithmetic expression can be used in the specification of a metric as there is no requirement on the expression to be linear. It is the domain designer's responsibility to ensure that plan metrics are well-defined (for example, do not involve divisions by zero).

Metrics are described in the problem description, allowing to easily explore the effect of different metrics in the construction of solutions to problems for the same domain. In order to define a metric in terms of a specific quantity it is necessary to instrument that quantity in the domain description. The case in which plans are constrained by finite availability of resources, is an important and interesting form of the planning problem, but the case in which plans of arbitrarily high utility can be constructed, is obviously an ill-defined problem, since an optimal plan does not exist. It is non-trivial to determine whether a planning problem provided with a metric is ill-defined. In fact, as shown in [Helmert, 2002], the introduction of numeric expressions, even in the constrained way adopted in PDDL2.1, makes the planning problem undecidable. The problem of finding a collection of actions which does not consume irreplaceable resources and has an overall beneficial impact on a plan metric is at least as hard as the planning problem. Therefore it is clear that determining whether a planning problem is even well-defined is undecidable. This demonstrates that the modeling problem, as well as the planning problem, becomes even more complex when metrics are introduced.

Below a whole domain with a problem will be shown to illustrate how minimizing metrics can be factored in in PDDL 2.1

```
(define (domain metricVehicle)
   (:requirements :strips :typing :fluents)
   (:types vehicle location)

   (:predicates      (at ?v - vehicle ?p - location)
                     (accessible ?v - vehicle ?p1 ?p2 – location))
```

```
    (:functions         (fuel-level ?v - vehicle)
                        (fuel-used ?v - vehicle)
                        (fuel-required ?p1 ?p2 - location)
                        (total-fuel-used))

    (:action drive
            :parameters (?v - vehicle ?from ?to - location)
            :precondition       (and (at ?v ?from)
                                (accessible ?v ?from ?to)
                                (>= (fuel-level ?v) (fuel-required ?from ?to)))
            :effect     (and (not (at ?v ?from))
                        (at ?v ?to)
                        (decrease (fuel-level ?v) (fuel-required ?from ?to))
                        (increase (total-fuel-used) (fuel-required ?from ?to))
                        (increase (fuel-used ?v) (fuel-required ?from ?to)))
    )
)
```

Figure 7: Domain definition for metrics

```
(define (problem metricVehicle-example)
    (:domain metricVehicle)
    (:objects
            truck car - vehicle
            Paris Berlin Rome Madrid – location)

    (:init
            (at truck Rome)
            (at car Paris)
            (= (fuel-level truck) 100)
            (= (fuel-level car) 100)
            (accessible car Paris Berlin)
            (accessible car Berlin Rome)
            (accessible car Rome Madrid)
            (accessible truck Rome Paris)
            (accessible truck Rome Berlin)
            (accessible truck Berlin Paris)
            (= (fuel-required Paris Berlin) 40)
            (= (fuel-required Berlin Rome) 30)
            (= (fuel-required Rome Madrid) 50)
            (= (fuel-required Rome Paris) 35)
            (= (fuel-required Rome Berlin) 40)
            (= (fuel-required Berlin Paris) 40)
            (= (total-fuel-used) 0)
```

```
            (= (fuel-used car) 0)
            (= (fuel-used truck) 0)
    )

    (:goal (and (at truck Paris)
            (at car Rome))
    )

    (:metric minimize (total-fuel-used))
)
```

Figure 8: Problem definition with metrics

Another of the novelties PDDL 2.1 introduced was the use of durative actions. Durative actions rely on a basic durative action structure consisting of the logical changes caused by application of the action. Logical change is always considered to be instantaneous, therefore the continuous aspects of a continuous durative action refer only to how numeric values change over the interval of the action. The modeling of temporal relationships in a discretized durative action is done by means of temporally annotated conditions and effects. All conditions and effects of durative actions must be temporally annotated. The annotation of a condition makes explicit whether the associated proposition must hold at the start of the interval (the point at which the action is applied), the end of the interval (the point at which the final effects of the action are asserted) or over the interval from the start to the end (invariant over the duration of the action). The annotation of an effect makes explicit whether the effect is immediate (it happens at the start of the interval) or delayed (it happens at the end of the interval). No other time points are accessible, so all discrete activity takes place at the identified start and end points of the actions in the plan. Invariant conditions in a durative action are required to hold over an interval that is open at both ends (starting and ending at the end points of the action). These are expressed using the over all construct. If one wants to specify that a fact p holds in the closed interval over the duration of a durative action, then three conditions are required: (at start p), (over all p) and (at end p).

```
(:durative-action load-truck
    :parameters     (?t - truck)
                    (?l - location)
                    (?o - cargo)
                    (?c - crane)
```

```
    :duration       (= ?duration 5)

    :condition      (and (at start (at ?t ?l))
                    (at start (at ?o ?l))
                    (at start (empty ?c)
                    (over all (at ?t ?l))
                    (at end (holding ?c ?o))

    :effect         (and (at end (in ?o ?t))
                    (at start (holding ?c ?o))
                    (at start (not (at ?o ?l)))
                    (at end (not (holding ?c ?o)))
)
```

Figure 9: An example of durative action

The objective of discrete durative actions is to abstract out continuous change and concentrate on the end points of the period over which change takes place. The syntax allows precise specification of the discrete changes at the end points of durative actions. However, when a plan needs to manage continuously changing values, as well as discretely changing ones, the durative action language and semantics need to be more powerful. General durative actions can have continuous as well as discrete effects. These increase, or decrease, some numeric variable according to a specified rate of change over time for that variable. When determining how to achieve a goal a planner must be able to access the values of these continuous quantities at arbitrary points on the time-line of the plan. The marker "#t" is used to refer to the continuously changing time from the start of a durative action during its execution.

```
(:durative-action heat-water
    :parameters (?p - pan)
    :duration ()

    :condition      (and (at start (full ?p))
                    (at start (onHeatSource ?p))
                    (at start (byPan))
                    (over all (full ?p))
                    (over all (onHeatSource ?p))
                    (over all (heating ?p))
                    (over all (<= (temperature ?p) 100))
                    (at end (byPan)))
```

```
    :effect         (and (at start (heating ?p))
                         (at end (not (heating ?p)))
                         (increase (temperature ?p) (* #t (heat-rate))))
)
```

Figure 10: An example of continuous durative action

Note that in the example above the duration is not known a priori and rather a termination condition expressed by the negation of a condition with an "over all" modifier, temperature > 100, is used. Thus, the temperature will gradually change until it reaches 100 degrees, theoretically being able to be modified by other actions over the resolution of this one, although the implications of these interactions over fluents between different actions is a very complex issue that has to be dealt both by modelers and by planner developers.

3. OBJECTIVES

The objectives of this work could be split into two different approaches: a pragmatical one and a theoretical one. A priori, the main objective is rather straightforward: defining a domain for PDDL so a planner can be used to play RTS games. This is actually the pragmatical issue of the work, as departing from its results a client will be created to participate in the 2008 ORTS competition, apart from serving as a base for the development of similar systems in other RTS games. Implementing computer players for games has been a classical AI work for those interested in researching and so such an approach is a perfectly valid axis for a work of this kind.

However, the uses for playing that the definition of the domain have are only a part of the whole work. To fully understand the scope of this work, one must pay attention to the sum of challenges solving such a problem poses to a researcher. Forgetting about the context of the work, which is automated playing in RTS games, what the objective really is is the design a domain with practical consequences for a real time, stochastic, multi-agent problem with incomplete information using conventional planning. This means that the problem to be solved has a complexity similar to that of a real life problem and thus it is right now far from the reach of most planners. Therefore, the main objective is not completely solving the problem itself but rather creating a system whose output could be regarded as useful in the sense that it could be used for real (though not at all optimal) applications.

More important than the output of the planner are the different techniques that will be tested when defining the domain and the problems. The main intent of the experimentation is not to end up creating a good enough system, so the results will focus on how positive the use of certain techniques is for each case and how this could relate to more general problems. Since little has been done until now in planning with problems of this caliber, comparisons are not available and the numeric results will not be taken into account but in the cases where the values are clearly good or bad for the later implementation of the ORTS client.

The system to be created is not a whole functioning client; ORTS will be used as an auxiliary tool to determine the validity of the experimentations as we lack other ways of testing how good a domain definition is. Thus, ORTS will be used to generate problems automatically and eventually to simulate the behavior of an autonomous computer player, but implementing the player itself is out of the scope of this work, apart from not being particularly useful from a

scientific point of view once conclusions are obtained from the use of the different approaches to the problem in planning.

Other than that, another objective for this work is both to encourage the use of realistic application for planning, showing that is possible to achieve good results with a proper approach in which has been regarded as a tool for only theoretical for a long time, and to give publicity to ORTS as a tool for AI research not restricted at all to its practical applications. Even though ORTS has been around the corner since several years ago, no relevant studies coming from researchers other than the creators of the tool have been published, which leads to think that ORTS is still largely unknown in the academical community.

4. DEFINITION IN PDDL AND CLIENT DESCRIPTION

In this section, the main part of the work will be described. It will be divided in subsections which will focus on different aspects of the work. Neither results nor conclusions will be commented here as they will appear in sections 5 and 6. At the end of this document, the whole domain and an example of problem will be included in annexes B and C respectively.

4.1 Problem description

The problem in which all the study is based is the third game of the annual ORTS competition held by ORTS developers at the University of Alberta, Canada. The rules and characteristics will be the ones for the upcoming 2008 competition, which will be held in August 1st. The game has barely changed from the previous two years, so earlier works would retain most of their value; however, only two teams of researchers (being one the ORTS developers) have participated in the previous years, so there is no significant experience from which departing. Besides, none of the teams used classic AI techniques apart from pathfinding and several attempts of designing squads of military units using K-neighbors, sending instead hand coded entries. The description and rules are publicly available both in the folder trunk/docs/ of ORTS and in the official web page for the 2008 competition [ORTS Competition 2008], but nevertheless we will give an overview of the most relevant characteristics of this work.

Game 3 is literally called by the organizers the "full-blown RTS" game. The main characteristics of RTS games appear here, though overly simplified. Basically it is a match between two players (red and blue) in which the objective is to annihilate the opponent. Fog of war is always active, there is a simple technology tree and the games involves both economy and military management, as mineral must be gathered to build new building and units. Buildings are rectangular shaped objects that are static and whose main role is producing units. Units are circular shaped objects that can move around at a certain speed and interact with other objects. The units and buildings are the following:

- **Worker:** The most basic unit, it gathers resources and builds new buildings. It can

attack as well, but that is not its main purpose. It is built at the Control Center.

- **Soldier:** The most basic military unit, it is built at the barracks.
- **Tank:** The most powerful military unit, it is built at the factory and outperforms soldiers.
- **Control center:** The most basic building, produces workers and serves as the point where the mineral is deployed.
- **Barracks:** The most basic military building, produces soldiers. To be built needs at least one control center.
- **Factory:** The most advanced military building, produces tanks. To be built needs at least one control center and one barracks.
- **Sheep:** This unit is neutral and indestructible. It wanders around and serves as a moving obstacle as opposed to the statical obstacles that minerals and hills represent.

Time in the games is measured in discrete frames of equal duration. In the competition a pace of 8 frames per second is used. Once per frame the game server sends individual game views to all clients which then can specify at most one action per game object under their control and send this vector of actions back to the server. All received actions are then randomly shuffled and executed in the server.

Internally, the terrain is represented by a rectangular array of tiles. Each tile is defined by a corner height value and a terrain type (“ground” or “cliff” in the competition, where ground tiles are traversable but cliff tiles are not). Tiles can also be split into two triangles (|\| or |/|) in which case both triangles can have different types. The height field defined by the tile corner heights is continuous, so corners shared by neighboring tiles have identical heights. Objects in the world have a shape and a position on the terrain. The position and size of objects are represented by integers using a scale of 16 points per tile (“tile points”). Shapes are circles or axis-aligned rectangles. Circles are specified by their center and radius. Rectangles are defined by their upper left and lower right coordinates.

A special form of objects are boundaries. These are line segments that objects cannot pass through which are computed from the tile-based terrain representation by considering

ground-cliff tile transitions. The server uses these boundary line segments for efficient collision detection. Both tile and boundary information is maintained in the client.

All objects controlled by a player have a visibility range, defined in terms of tiles. Visibility is computed from the tile containing the center of the object. Any objects intersecting a visible tile are themselves visible. All game objects are represented by a unique object id (an integer). Object ids are assigned deterministically based on the currently available ids. Any object that goes out of view and subsequently comes back into view is not guaranteed to be assigned the same id.

All units have a maximum speed with which they can move. This means that within each simulation frame, the valid moves for an object are constrained to the integer coordinates that are within a distance of less or equal to the speed of the object. Movement targets are only constrained to be integers; any location on the game field is acceptable. Every move is assumed to go in a straight line from the object's current position to its destination. Objects move simultaneously and their current locations are rounded to tile points before being sent to the clients which can lead to temporary and small object overlap on the client side. In case of collisions with boundaries or other objects, the moving objects are stopped at the collision location and no damage is inflicted.

If the target location for a move command cannot be reached in one simulation cycle the object will continue to move in a straight line until a new command is sent or the object collides with a boundary or game object (or scripted game mechanics cause its motion to change). It is not possible to do several moves in one tick in order to avoid obstacles, even if the total distance of the moves is less than the maximum speed of the unit. It is not possible to move inside another object or building, however if somehow trapped inside one it is possible to move away (or out in the case of buildings).

Units can engage in combat with other units and with buildings. Soldiers and tanks can attack from a distance, while workers need to stand close to the attacked object. It suffices for any part of the attacked object to be in range. Specifically, weapon range is compared against the minimum physical distance between any part of the objects. After attacking, the weapons are subjected to a cooldown period before they can attack again. The cooldown time is specified in simulation frames. By dividing the rate of frames per second by the

cooldown, the number of attacks per second is obtained. This way, the average Damage Per Second (DPS) can be easily calculated, which is interesting for players as it is a more realistic way of representing the damage output of a unit.

Objects have hitpoints (HPs) indicating how much damage they can take before being destroyed. Lost HPs cannot be regained. Units and buildings can also have armor which decreases the damage dealt by a weapon by subtracting a constant from each attack. Damage values are uniformly distributed over certain intervals. Units only die after a simulation frame has been completed when their HPs have dropped below 1, ensuring that the order of executing attack actions is irrelevant.

The statistics of the units (also known as stats) are defined are always the same for all the matches. These statistics are the different values the aforementioned characteristics of the units and buildings have. Stats are the key element when comparing units and often strategies are created upon these values. The next table shows the stats of all the units and buildings in game 3.

OBJECT	Worker	Marine	Tank	Sheep	Control	Barracks	Factory
COST	50	50	200	-	600	400	400
HP	60	40	150	∞	1700	1150	1400
SPEED (TP/s)	4	3	3	2	-	-	-
SIZE	r = 3	r = 4	r = 7	r = 4	62 * 62	62 * 46	62 * 46
BUILD TIME	7s	8s	16s	-	38s	25s	25s
VISION	6	6	7	-	4	4	4
ARMOR	0	0	1	0	2	2	2
RANGE	4	64	112	-	-	-	-
DAMAGE	4-6	5-7	26-34	-	-	-	-
COOLDOWN	8	8	20	-	-	-	-
DPS	4-6	5-7	10.4-13.6	-	-	-	-

Table 2: Statistics of the objects in game 3

In this table we can appreciate the differences between units and how this leads them to fulfill different roles. Workers, while being builders and gatherers, are not without defensive capabilities: their damage is only a bit below the one of the soldier, they are faster and far more resilient while costing the same. The difference that makes a soldier a better option is its range: the worker has to stand practically by its target to attack it but the soldier can shoot it from afar. A worker could close the gap between both ranges and maybe kill the soldier thanks to its higher HPs, but in game 3 (and all the other games in the competition) units can attack while moving even backwards, so if intelligently micromanaged a soldier will always win against a worker in a 1 versus 1 situation with no obstacles. The same is true for tanks and soldiers: tanks are four times more expensive than soldiers apart from the minerals invested in the factory but deal far less damage than four soldiers altogether. Nevertheless, they have the same speed and the tanks twice the range, so in an ideal situation tanks would always outperform enemy soldiers, no matter how many soldiers are used. Of course, ideal situations rarely happen, partly because it is designed to be so. Terrain, allied units, sheep, buildings, minerals,... all contribute to create a more realistic and complex battle situation, so choosing tanks over any other unit is not such as straightforward as it may seem, which is one the most attractive features of RTS games.

For simplicity, there is only one resource called minerals. Minerals are spread out on the map in clusters of “patches”. Mineral patches are circular objects. A worker unit can gather minerals when close to a mineral patch. A maximum of four workers can simultaneously harvest one mineral patch. Minerals must be delivered to any control center before they can be used. A worker may drop off its minerals immediately once it is within range. Each mineral patch contains a finite number of minerals, which are decreased when the object is mined. Mineral objects vanish once they are completely mined. Workers may mine at a rate of 1 mineral every 4 simulation frames. A worker can carry a maximum of 10 minerals at any time.

In game 3, the maps are randomly generated every match, though they always have similar characteristics. The initial state is always the same: both players start with 6 workers, a control center, and 600 minerals each, and a nearby mineral patch cluster on randomized

terrain. Start locations are not symmetric and again, invincible sheep are roaming the map randomly. The next screenshot taken from a game simulation depicts the initial state for a player.

Figure 11: Initial state in game 3

The winning condition is to destroy all enemy buildings independently of the number of other units the opponent may posses, which can lead to interesting suicidal tactics in which a player (or both) focuses on destroying the enemy buildings instead of countering the opposing army and disregarding self defense. However, the game has a limit of 20 minutes of play or 9600 frames (20 * 60 * 8 FPS). In case the time limit was reached, the winner is decided by a score based on the mineral gathered, the units and buildings built and the enemy units and buildings destroyed. The equation of the score is the following: (current mineral count / 2) + construction cost of all standing buildings and units + construction cost of all opponent buildings and units that have been destroyed.

All the data mentioned in this section is included in the .bp files located under trunk/tournament-2008, being the most important ones game3.bp and all the blueprint files located under trunk/tournament-2008/terrans, included in Annex A. They include not only the stats of the units but also all the possible actions of each object with the parameters and the scripted effects. Some of the most important descriptions will be included in the Annex A at the end of this document.

4.2 Client structure

ORTS works as a server-client system. The server is in charge of the simulation and synchronization of the in-game events. On the other hand, the client only receives its visible part of the state of the game (also known as a view) each frame and sends backs actions independently of their validity. As it was mentioned in the section 2.2.1, clients connect to the server using sockets, so as long as they comply to the defined protocol they can be implemented in any way. This gives to the developers a great degree of flexibility when designing their client, as it is not restricted in any way.

In our case, the client is coded in C++ and is basically composed of two independent processes that are constantly running: the main loop and the planning process. The main loop contains all the logic of the client and is based on the clients given by the ORTS developers as examples, while the planning process just executes the planner every time it is called.

As this game is a fast paced one, the game state changes quickly. Therefore, planning for very long ranges, apart from being very costly (often to the limit of rendering the problem unfeasible due to the dimension of the state-space) is not particularly useful because the situation may have changed drastically and the plan would no longer be valid. The solution to this situation is to replan as often as possible as long as the plans are of sufficient quality. Basically every time a plan is calculated, the client tries to follow that plan until a new one is generated, in which case the old plan will be discarded and the client will stick to the newly generated plan until the following arrives. Time is here a critical factor, so the design of the domain and the problems has been done around this conception.

The external libraries available for developers have been called modules. Most ORTS applications are built around the concept of modules. Modules provide various services to the application, most of them receive and handle events, and some of them fire events. Unless you are managing the ORTS network protocol yourself, you will need, at the very least, a GameStateModule. This important module keeps track of the various objects in the game state, sends events when a new server view has arrived, and provides access to delta information, that is, which units are new and which have vanished/died. Another useful service is the GfxModule, which creates a graphical interface for the user to view the game state. It can even be customized by passing an implementation of a specific interface to handle player events in various ways, allowing the user to play a game.

The main loop is the manager of the process. It is in charge of the connection issues, the interface, libraries, etc... The class that contains the main method is game3_main.C at trunk/apps/game3/src. This class has almost no real functionality and has been copied almost line by line from other examples. Basically it gathers the possible arguments it may receive by console, includes the different libraries and handlers needed for managing the messages and initializes the graphic interface. The connection is managed with GameStateModule and requires no additional work.

ORTS clients are usually conceived as a set of handlers in charge of managing events represented by a particular kind of message. This is compulsory for view messages and other control messages the server can send to the client, but it is also a way of implementing behaviors for units as the handler takes care of all the messages of a kind related to the unit and no control structures have to be implemented in the main loop. The modules for pathfinding are a typical example of this kind of functioning, being the call to the module the following one:

```
call_handlers(TerrainModule::PathEvent(EventFactory::new_who(),
TerrainModule::FIND_PATH_MSG,TerrainBase::Obj2LocTask(gob, target)));
```

In this sentence, a new event along with a message (FIND_PATH_MSG) is created, passing as the third argument an object that encapsulates the identifier of the object and the location where the unit has to go.

This system of event handling determines the structure of the client. Since every frame a view is received by calling the method recv_view() in GameStateModule, the handler of the message of type GameStateModule::VIEW_MSG is the handler that fires every frame and upon which all the functionality of the client is based. Other than that as long as the libraries are loaded messages are sent and functionality is distributed, so class diagrams are not worth mentioning in clients. This makes the way of implementing client a rather unintuitive task, but offers the advantage of using events with no control structures with external libraries and forgetting about its inner working.

In the client implemented for this work, the class in charge of handling view messages is PlannerEventHandler.C For every view received the client checks whether there is a new plan or not. When a new plan is received replanning must be scheduled so immediately a new problem is created by calling GameStateModule::get_game() and GameStateModule::get_changes() and extracting all the relevant data about the problem. In a real implementation of the client this should be done calling a new process that executes in parallel so replanning is done while the main process goes on playing executing the plans via modules. However since the scope of this work is just studying the design and quality of the domain and the plans generated, no parallel execution will be done and the plans will be generated just for evaluation.

4.3 Domain definition

The purpose of planning in this work is not to obtain plans that could be as good as those calculated by a human player. As the problem to be solved is far more complex than the average planning problem, in this case planning will focus on obtaining coherent plans in a reasonable time. Coherent means that the plans should represent one of the most typical strategies, trying to use as many units as possible and coordinating them to achieve a certain goal. In this section we will describe how the domain will be designed justifying the modeling decisions prior to analyzing the results obtained.

4.3.1 Behaviors as unit consuming operators

When designing a domain for a game, the easiest way of doing it is correlating the possible actions of the units to operators in the domain. This way the plan will reflect every single action for every single unit. However, this is hardly a correct approach in RTS games, as the number of evaluated nodes grows exponentially with the number of possible actions, and the high number of units and the complexity of the world only worsens the situation; if designed in this simplistic way, the planner is likely to not be able solve the problem or at least to do it in a limited amount of time and using a limited amount of memory.

RTS games are characterized by being divided in two levels in terms of controlling units: strategical and tactical actions, respectively correlated to macromanagement and micromanagement. This differentiation of levels allows for an easy hierarchical implementation of computer players. As integrating the low level actions available in the game into the planning process is most of the time unfeasible, a possible solution is making the planner taking care of only one of these aspects. Given that micromanagement usually requires very fast reactions and that it is somehow easier to control using a set of rules (as in a reactive autonomous agent), macromanagement is probably the most suitable aspect of the game using planning. Not only it adapts better to the conception of planning as a sequence of actions that achieve a goal, but also such a design is often a natural result of abstraction over the multiple factors of the domain rather than a conscious decision.

In this work, the proposed solution is not individually controlling every unit executing plans. Instead, we will assume a certain degree of autonomy for the units and assign them a task related to some goal. Depending on the task, a specific set of actions must be followed by the unit, set of actions that will be implemented as part of the micromanagement component of the client, so in order to differentiate both levels of actions more drastically we will use the term behavior to refer to the actions a unit will do in order to achieve a specific goal. This way the planner will assign behaviors to units, which will act autonomously without further intervention by the higher levels of decision of the client.

These so-called behaviors are general concepts associated with particular units, so only some behaviors will be adequate for some units depending on their role. The operator that assigns a behavior will have that into account, so typing (a feature supported by PDDL 2.1) will be necessary. Besides, in most of the cases these behaviors are associated with some specific object in the game, meaning that they will not be just the definition of what the agent will do but also with which object or area in the game. An example will be attacking: a soldier cannot just be ordered to attack, it needs a target, so the operator will involve the soldier, the behavior and the target.

Another particularity of this design decision is that assigning a behavior uses up that unit, this is, that unit cannot be used in another operator once it has been given a behavior. This is justifiable for two reasons: on one hand, behaviors are not instantaneous actions, they go over some time during which a new plan will probably be calculated, so assigning one behavior per planning is good enough if replanning is done in a reasonable time as the situation probably will not change that much for the behavior to become obsolete (such as a soldier attacking an already destroyed base or a worker gathering mineral from a depleted cluster); on the other hand, this easies the generation of plans as reduces the state-space by limiting the number of units that can be taken as a parameter for an operator and frees the fact that represents the unit in case it is needed for another reason, reducing the number of facts the planner has to deal with.

The following subsections are the descriptions of the operators that relate to this approach along with possible ways of implementing the behaviors at low level (the effects of the actions will not be included as they will be described later together with the design decisions that affect them).

4.3.1.1 Attack

Attacking is feasible only for soldiers, and has an enemy base associated as a parameter. It uses the positions of both the soldier and the enemy base to determine the cost of the action, as it will take time for the soldier to get to the enemy base. A simple implementation of this behavior would be making the unit advance towards the enemy base shooting enemies as it encounters them, prioritizing military units over workers and workers over buildings. The PDDL syntax is the following:

```
(:action ATTACK
        :parameters (?soldier - soldier ?enemybase - enemybase ?src - position ?dest   - position)
        :precondition (and
                        (at ?enemybase ?dest)
                        (at ?soldier ?src) )
        :effects (..)
)
```

4.3.1.2 Defend

Defending is an analogous action to attacking: only soldiers can do it, and has an allied base associated as a parameter. The PDDL syntax is the following:

```
(:action DEFEND
        :parameters     (?soldier - soldier ?cc - cc ?src - position ?dest - position)
        :precondition   (and
                        (at ?cc ?dest)
                        (at ?soldier ?src) )
        :effects (..)
)
```

4.3.1.3 Harass

Harassing is similar to attacking, but its objective is deviating the attention of the enemy and not making a direct attack. Other than that it is similar to attack with the exception that in this case the soldier flees from opposing military units instead of engaging them in combat. The PDDL syntax is the following:

```
(:action HARASS
        :parameters (?soldier - soldier ?enemybase - enemybase ?src - position ?dest   - position)
        :precondition   (and
```

```
                    (at ?enemybase ?dest)
                    (at ?soldier ?src) )
        :effects (..)
)
```

4.3.1.4 Scout

Scout is done by workers as soldiers are slower and more likely to harass. Its target is an area marked as unexplored in the definition of the problem, and a way of implementing this behavior can be making the worker wander around the target location in a spiral while avoiding enemies. The PDDL syntax is the following:

```
(:action SCOUT
        :parameters (?worker - worker ?src - position ?dest - position)
        :precondition  (and
                        (unexplored ?dest)
                        (at ?worker ?src)
        )
                :effects (..)
)
```

4.3.1.5 Mine

Mine is obviously only done by workers. It needs a control center near a mineral cluster as a parameter and not an individual mineral as both control centers and clusters are needed to mine, providing an additional level of abstraction to the PDDL definition. To implement this behavior is not a trivial task (actually game 1 is about optimizing this) so the best solution is probably using the existent low AI modules instead of implementing it manually. The PDDL syntax is the following:

```
(:action MINE
        :parameters (?worker - worker ?cc - cc ?src - position ?dest - position)
        :precondition  (and
                        (at ?worker ?src)
```

```
                    (at ?cc ?dest)
                    (> (max-workers-to-mine ?cc) 0)
        )
            :effects (..)
    )
```

4.3.2 Intermediate operators

All the operators are actions related to units or buildings as in game 3 it is the only way an action can be performed. In the previous section all the actions that meant long processes in which the unit is used up until replanning was done. Nevertheless, not all the actions in game 3 can be limitless over time, some of them can be performed in a known amount of time and so the unit or building performing them can be assigned to do something else after that action, which in terms of the domain definition means that the unit or building is not used up but rather its relative work load for the plan is increased depending of the action. The work load of a unit is directly related with cost and so will be described in Section 4.3.4.

One of the possible intermediate operators, build command center, has been left out of the design because of the complexity of determining a favorable position and the importance of always having at least one control center. Therefore, its construction will be scheduled by the client when certain conditions are met (most likely losing the last control center or having too many workers using it to deploy resources) giving it a higher priority over the planner's sequence of actions. These are the implemented intermediate operators.

4.3.2.1 Build worker

This operator represents the action of building a worker at a control center. It must be used once per worker built because PDDL uses individual facts for object representation instead of a numerical value of available objects. When created, a

worker comes at the same position of the control center, and one of its requisites is having at least 50 minerals, the cost of a worker. Both the control center and the worker can be used in subsequent actions of the same plan, justifying this operator being classified as intermediate. Why a predicate (ready ?worker) is listed as a prerequisite will be explained in the Section 4.3.6. The PDDL syntax is the following:

```
(:action BUILDWORKER
        :parameters     (?cc - cc ?position - position ?worker - worker)
        :precondition    (and
                         (ready ?worker)
                         (> (minerals) 50)
                         (at ?cc ?position) )
        :effects (..)
)
```

4.3.2.2 Build soldier

Completely analogous to the previous operator, building a soldier is done at a barracks. The cost is again 50 minerals, but this is a coincidence because the cost of a worker and the cost of a soldier is incidentally the same according to the game description. The PDDL syntax is the following:

```
(:action BUILDSOLDIER
        :parameters     (?barracks - barracks ?position - position ?soldier - soldier)
        :precondition   (and
                         (at ?barracks ?position)
                         (> (minerals) 50)
                         (ready ?soldier) )
        :effects (..)
)
```

4.3.2.3 Build barracks

Similar to the previous operators, but this time performed by a worker. The barracks is built where the worker is, although the exact in-game position must be chosen by the client, because the presence of obstacles, units and nearly every object in the game will not allow its construction. The PDDL syntax is the following:

```
(:action BUILDBARRACKS
        :parameters (?barracks - barracks ?position - position ?worker - worker)
        :precondition    (and
                         (> (minerals) 400)
                         (at ?worker ?position) )
        :effects (..)
)
```

4.3.3 Goal definition

In automated planning one of the most common ways of defining the goal of a plan is using a set of facts that must exist for the goal to be achieved, not unlike setting a satisfiability condition. With the advent of numeric values in planning, another way of describing the goal condition is using one of more fluents (along with facts if needed) and comparing them with a value. This is a typical case in problems whose goal is getting a certain amount of something, as for example an amount of minerals in game 1 for a domain centered in resource gathering.

However, this approach is not valid in our context. A great part of the complexity of RTS games is knowing which actions should be done at every moment to get closer to the winning condition. A classical approach like making the goal match the winning condition is out of the question since it would be equivalent to solve the complete RST game, which is a problem far too complex for any computational system. Rather, each problem generated by the client must have simple goals so a plan for achieving them can be computed in real game time. These simple goals generally correspond to general

actions at a strategic level, such as “attack enemy base A” or “get 500 minerals”, but the main issue here is that more often than not these goals are opposite goals as achieving one means that the other can be achieved. For example, if resources are limited within the problem “attack enemy base A” means using and losing soldiers which are necessary to achieve “defend allied base B”, so the domain definition must deal with not being able to achieve all the possible goals that would lead to a more advantageous situation to the player. This makes these simple goals be qualified as soft goals, in the sense that they are not compulsory to solve the problem, but are a measure of the quality of the plan.

Because goals in RTS games work like this, defining a condition-based goal that involves achieving as many soft goals as possible is a very difficult task if facts or straightforward numeric values are used, more so as RTS games are a very dynamical environment in which the goals can change in a matter of seconds. To solve this, a different approach has been used: rewarded soft goals. This involves two things: first, the goal condition in the problem definition is getting at least a certain number of rewards by attaining soft goals, and second, rewards are obtained not by solving a certain part of the problem but rather by using units in actions that lead to achieving a certain soft goal.

The first of these factors, getting at least a certain number of rewards, is not a hard condition. The rewards can be set to 0 and since the planner is supposed to give plans of some quality, even though the goal condition is satisfied from the very beginning without having to compute anything the planner is likely to come up with a plan that will give some rewards despite that not being its goal. Consequently, this minimum reward is not the goal of the domain, the goal of the domain is maximizing the rewards, so its value is an indicative of the minimum quality of the plan, forcing the planner to give a sequence of actions sufficiently good to be used by the client in a real situation and avoiding lackluster plans which could be of no use. Of course, attention must be paid so this value is not too high making the planner take too much time to give a solution or even making the problem unsolvable, so the election of this parameter is usually a trade-off between quality and performance. As the soft goals that can be satisfied to obtain rewards vary throughout the match, this value must be calculated every time a problem is generated, which is an added difficulty parameterizing the client.

The second of the issues that are related to this solution is using units in actions that lead to achieving a certain soft goal. As described in section 4.3.1 some operators use up a unit as it was another resource giving it a behavior; apart from the reasons already mentioned, this approach also allows rewarding these behaviors accordingly to their utility leading to a natural approach of assigning behaviors. For instance, a soldier that attacks an undefended base will get more rewards than another attacking a heavily defended base (which would result in him being outnumbered by the defending forces), or a scouting worker when there is none doing it yet than when there are already many workers scouting the map, as the benefits will be lesser. Another reason for using this design is the fact that soft goals are often mutually exclusive because the resources (units in this case) are limited and thus using up units to achieve a given goal reflects the impossibility of using it to achieve a different one in the real problem, apart from simplifying the subsequent computation. The exception to this kind of actions is the MINE operator because mining itself does not represent a benefit and it is used instead as the previous step of producing a unit, which will perform a task that will obtain a reward, solving the problem of determining its reward and making emphasis on its means-ends idea.

As aforementioned, determining the reward for each operator is not a trivial task. Depending on how many rewards a unit can get, it will do one thing or another, so an imbalance in rewarding would generate plans oriented to one specific behavior instead of using flexible plans, apart from being relatively easy to replicate with a scripted behavior, actually what we want to avoid in the first place. Again basing us on the expertise in RTS games is the most natural and probably best overall approach. The rewards must be set depending on the nature of the action and here we can distinguish two kind of actions: combat oriented (attacking and defending) and not combat oriented (scouting and harassing).

In the combat oriented actions the best rule is usually assigning a few more soldiers to attack or defend than the enemy force because fewer can mean defeat and more would be what is commonly called by RTS gamers as overkill, this is, using too many units for something that could be done with less units and therefore not using them in more useful tasks. As actions are related to a single unit, the reward must be calculated per unit with the additional problem that the planner does not know a priori how many units

will be attacking or defending the same point (as those would be actions performed afterwards because of the sequential conception of plans in most efficient techniques in automated planning). Therefore, our solution consists of a typical Gaussian function of the form $ae^{-\frac{(x-b)^c}{2c^2}}$ in which a is the maximum reward a unit can get, b is the position of the maximum reward (which should correspond to the number of enemy soldiers in the opposing force plus a little value) and c the width of the Gaussian bell, which should have a value so $f(0) \sim 0$. Figure 12 is a graphic with the rewards when attacking a base defended by 3 soldiers with the maximum reward being 10 obtained when attacking with 2 more soldiers and using 1.1 as c:

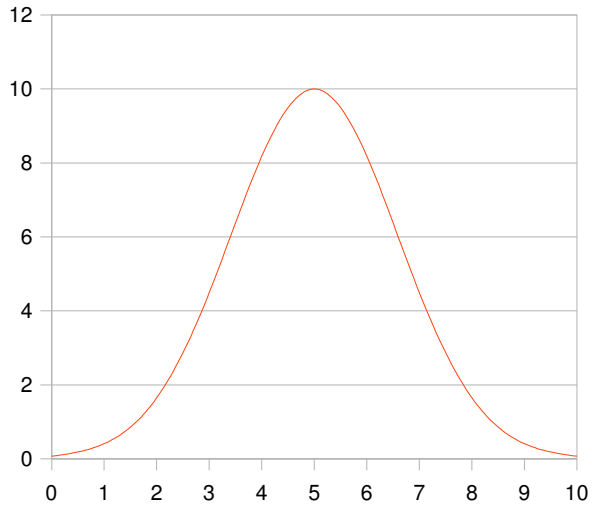

Figure 12: Rewards per soldier

As we can see, the first soldier will contribute with almost no rewards, but the sixth will grant a hefty amount of rewards. Therefore assigning a soldier to attack when too few or too many have been already assigned grants so few rewards that the planner will avoid doing so. Another advantage is that this is a conservative approach to attacking and defending, as it is unlikely that the planner decides to attack or to defend with just a few soldiers if it decides to attack, encouraging an all or nothing behavior that leads to successful attacks. Figure 13 shows how overall rewards are obtained using a cumulative Gaussian function:

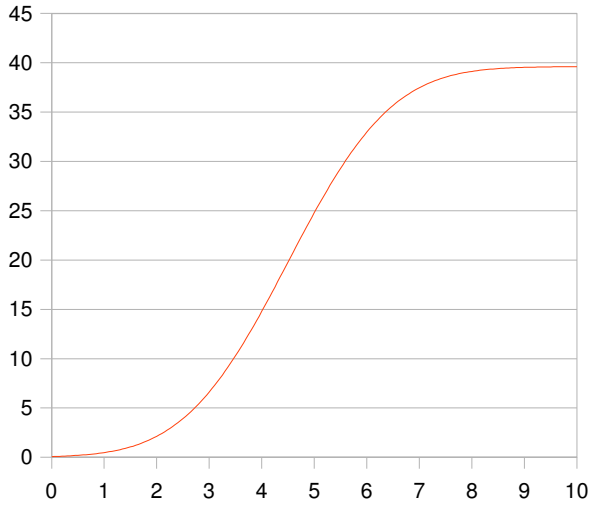

Figure 13: Accumulated rewards

The value of *a* is a parametrized value affected by the existent number of targets. This means that the rewards are divided between the number of enemy bases for the attack operator and the number of allied bases when defending. This is again a decision based on the characteristics of the game: if there are fewer enemy bases than allied ones, it prioritizes attacking as it gets it closer to the winning condition and vice versa when attacking. For instance, if the enemy has two bases and the player three destroying one and losing another is favorable as it leaves the opponent with a single base left, while if the player has only one base left and the enemy more than one, defending will be more rewarding than attacking as it would mean losing the game. This also helps in determining the overall aggressiveness of the plans, as both reward values for attacking and defending are independent and can be changed even throughout the game to encourage one way of playing or another.

A problem that arises when implementing this reward function is that it must be either embedded in the PDDL specification or precalculated before every call to the planner and added in the problem as facts with their associated fluent. Both of them mean a significant overload for the planner because of the calculations it has to do when evaluating a state and the added number of facts and fluents it has to deal with, so a simpler approach when computational resources are scarce is needed. To solve this problem, the best solution is to transform the Gaussian function to a lineal one in which

there are no exponential calculations. The simplest approach to this is using two functions that go from 0 rewards at 0 soldiers assigned to *a* rewards at *x* = enemy-soldiers+3 and from *a* rewards at enemy-soldiers+3 to 0 rewards at *x* = 2*(enemy-soldiers+3). It is a straightforward approach that loses some of the advantages of the Gaussian function but it is very easily calculated and does not have a negative impact on computation time. This is how it is formulated:

$$f(x)=\begin{cases} \dfrac{ax}{enemySoldiers+3} & x \leq enemy-soldiers+3 \\ 2a-\dfrac{ax}{enemySoldiers+3} & x > enemy-soldiers+3 \end{cases}$$

Figure 14: Final rewards function

4.3.4 Costs

In this work, the main concern is the quality of the plan. As it has been analyzed in previous sections, the amount of rewards obtained is one of the ways of judging whether a plan is good enough or not. However, the amount of rewards obtained alone is an indicator of how appropriate the behaviors assigned are and not the overall plan. While attacking a base using a certain number of soldiers may grant a good amount of rewards, choosing which soldiers has not been taken into consideration yet. If only rewards were used, the planner would choose units randomly probably sending them to objectives far away from their position instead of choosing units closer to the target, for example.

Regarding costs, it is mandatory to specify what is considered cost in the domain definition and what is not. When thinking about costs in RTS games, the first thing that comes to mind is the unit's cost, this is, the amount of mineral that is necessary to produce or build the unit. However in this section cost is analyzed from the planner's point of view. Therefore, cost in this context will be understood as a factor that decreases the utility of an action: time spent and concurrency. Actions that produce or that use up resources like mine or build barracks respectively have no effect on the way cost is treated other than when evaluating their utility.

The main issue in cost is time and concurrency then. These problems have always been one of the critical points in automated planning and thus have been faced many times using different approaches. One of the most clear examples of this is PDDL 2.1 itself: it is the first planning definition language that tries to standardize time management in an automated planning setting. However classical approaches in our context are not as useful as they may seem due to the way planners try to optimize plans, so using the time features of PDDL 2.1 has been discarded and a different approach has been attenpted.

Theoretically, state of the art planners are able to use a metric as a guideline for the plan's quality instead of the traditional plan length measurement. Using the declaration that PDDL provides, a metric can be imposed to a planner indicating it that the objective is not only to reach the goal but also either to minimize or maximize a given value. Thanks to the flexibility of PDDL this value can be anything from a single fluent (the most obvious case being time) to the value of any mathematical function that may be expressed in PDDL, so a priori an expression could have been written to be used as an adequate metric for this domain. However this is not true; optimizing in planning is a very hard task and the ways planners are able to use the metrics are very reduced.

A common limitation lies in how many planners use the metrics: they do not try to minimize or maximize a value, instead they look at the operators that modify that value and try to come with a minimum length plan having into account only those operators, often using tweaks to work with operators that decrement the value or when maximizing. In order for this to work, the amount by which those operators modify the value must be before the planning so the metric can be established proportionally. Because of that, most current planners cannot work with state-dependent cost assigments and all the operators must modify the value to minimize homogeneously throughout the plan.

RTS games are very complex and the game state during a game is changing constantly. Due to this dynamic behavior fixed-cost operators are ill suited for their use in RTS games and thus a classical approach would not work as a way to force the planner to give high quality plans. Having into account that in this domain quality depends on several factors, the use of the time features provided by PDDL 2.1 is discarded. More

so, rewards, which have been stated as another way of measuring the quality of the plan, are generated dynamically and would not satisfy the condition that most planners impose to operators when minimizing a value. In this particular case the planner that will be used for the experimentation, Metric-FF, works as specified previously, so an alternative is needed.

As stated before, implementing metric optimization in a planner is a very complex problem that is yet to be solved, so hand-coding a specific planner for this domain is out of the question. With all these limitations, the proposed way of overcoming this problem is using cost-to-operator conversion. The main point in cost to operator conversion is standardizing the assignment of values to the fluent that will serve as a metric so conventional planners can effectively use it to improve the plans. Using this approach does not require big modifications to the domain: the value to minimize is stored in a temporal fluent until the theoretical goal is reached. Once the goal is reached, instead of finishing the execution an operator is used several times to subtract a constant value to the temporal fluent that holds the cost and add that same value to another fluent that will be the actual fluent the planner will try to minimize. This means two things: the metric is modified homogeneously, so planners will not have any problem working with it, and an auxiliary goal, achieved when the auxiliary cost fluent takes a value below 0 once the real goal is reached and usually represented by an unrelated fact, is necessary to stop generating the plan. In our domain this has been implemented like this:

```
(:action TO-END
        :precondition       (and
                            (> (rewards) (rewards-threshold))
                            (> (pre-total-cost) 0) )
                    :effect (and
                            (decrease (pre-total-cost) (cost-increment))
                            (increase (total-cost) (cost-increment)) )
)

(:action END
        :precondition       (and
                            (> (rewards) (rewards-threshold))
```

```
                    (<= (pre-total-cost) 0) )
        :effect    (goal_achieved)
)
```

In this domain, the goal is getting a minimum amount of rewards, represented by the prerequisite "(> (rewards) (rewards-threshold))". Once the goal has been reached, the operator TO-END can be called repeatedly to decrease the fluent that temporally holds the value to minimize and increase the fluent used in the metric. The values are modified by a constant (in this case represented by a fluent that does not change) called cost-increment until pre-total-cost, the temporal fluent, has a value below 0 and the operator END is finally called, adding the fact goal-achieved to the fact database and terminating execution.

Plans generated like this are the same sequence of actions a normal domain would generate but with the distinguishing fact that at the end of the plan there is a series of calls to the operator TO-END, which determines the effective plan length to minimize. An example of a plan would be the following one:

```
step    0: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
        1: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
        2: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
        3: ATTACK SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
        4: HARASS SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
        5: SCOUT WORKER0 POS5-1 POS6-2
        6: TO-END
        7: TO-END
        8: TO-END
        9: TO-END
        10: END
```

As it can be seen, there have been four calls to the TO-END operator, meaning that the cost is between three and four times the constant cost-increment. Other than that, after

discarding these last operators the resulting plan would be a completely conventional one with no further modifications to the domain.

To factor in time and all the concurrency issues the domain has, a cost must be integrated with the rewards system. In this domain, units and buildings can perform different actions, but these actions cannot be performed at the same time. Besides, the order of the actions performed by the same unit or building is a relevant factor so special attention must be made to this. In our design, basically the concept around time is that the latter an action is performed, the least its utility is, which means that the most time has passed, the fewer the rewards are. This has been implemented in two ways: cost due to delays and cost associated to units and buildings.

The cost due to delays in actions is just the loss of utility that occurs when that action cannot be performed immediately. This happens only with actions that require a unit to move to perform it, as the time spent when producing units is factored associating it to the units produced. An example of this would be attacking an enemy base with a nearby soldier instead of with a distant one, which is usually more adequate and thus has a lower cost. As this cost is related with the time the units spend moving toward their target it is based on the position of both the unit and the target and the characteristics of the terrain. The positions in this domain are represented by a fact with two associated fluents which represent their x and y coordinates. Besides, an additional fluent is used to indicate how difficult is for a unit to move across the area designated by the position based on the static obstacles present in the area and the units (both enemy and allied) that move around it. All in all, the cost of moving is on one hand the Manhattan distance, represented by $|x0-x1|+|y0-y1|$ and on the other the difference between the roughness factor of the areas if the value of the target position is higher. This way of calculating the cost of moving is simplistic and probably not accurate for real life problems, but due to the limitations of planners with mathematical operations and the level of abstraction used in the domain it can be a valuable indicator of the cost of moving from the origin to the destination.

The other way of taking into account costs is associating an individual cost to every unit and building that has been involved in an intermediate action. That cost will be added to any other cost a subsequent action may have, so units participating in several actions

will have a higher cost associated. This is most useful for the unit and building producing operators: for instance, if a barracks is used to produce a soldier, a fixed cost will be added to the cost associated to the barracks and the cost associated to the new soldier. This way if that soldier is used to attack an enemy base, its final cost will be higher than the cost of a soldier that already existed when the problem was generated. Besides if that barracks produces another soldier, this second soldier will have the production cost plus the cost the barracks already had associated, meaning that it is counter productive to produce many soldiers at the same barracks. This is reflected in the game in three ways: first, units and buildings do not queue too many actions that could result in a more complex and costly plan, actions that besides could not have been done before replanning, wasting resources and complicating the problem unnecessarily; second, the intermediate actions are distributed so a single unit or building does not perform too many things if other units or buildings can do it, achieving some degree of collaboration at unit level; and third, it deals with the time issues of newly produced units by assigning them an initial cost that otherwise would be zero.

The way costs and rewards are integrated is not a trivial one. Costs are not subtracted from rewards but instead they are added to the temporal cost fluent pre-total-cost while rewards are subtracted to it. This has been done this way because of the way the heuristic function works in most planners. In most of the cases when a planner evaluates a state the heuristic function estimates how good it is depending on the distance that there is to the goal. In this case, being closer to the goal means doing less calls to the TO-END operator, so the lesser the value of pre-total-cost theoretically the better the state. The approach used is to minimize the value instead of maximizing it for two reasons: first, the most natural approach for most planners is minimizing rather than maximizing; second, minimizing means that the best plans will be the shortest ones if rewards and costs are correctly factored in as they will need less calls to the TO-END operator to finish, while maximizing would mean to look for a plan with as many calls as possible, resulting in a bigger state-space to be explored. Minimizing though has an essential disadvantage: pre-total-cost needs a greater than 0 value initially so its value can be reduced. This value must be big enough to allow a margin of improvement (if for example its value is too small it may reach 0 when it could be minimized further and the execution of the plan would end with no calls to TO-END even if there were better plans) but should not be too big as that would mean additional calls to TO-END that will result in a detriment of the performance. This value must be carefully balanced with the rewards, the costs and the constant value used in TO-END to convert the cost to the

final fluent.

Another point worth of mentioning is the way pre-total-cost must be modified. When heuristic planners evaluate the states before expanding them, they usually choose only those that they perceive as favorable for the resolution of the problem. This means that if an operator using costs increases pre-total-cost it has little possibilities to be used. This fact has two consequences, being the first one that it does not allow to use an approach in which rewards reduce the cost instead of the opposite as it has been implemented. A domain in which costs are bigger than rewards and their difference is minimized has the advantage of not having to deal with the initial value of pre-total-cost while keeping the advantages of minimizing, but as cost increases and not decreases with every action, it is unlikely that the planner would search too deep in the space-state. The second consequence is that costs can never be applied in an operator if no rewards are granted at the same time because the planner would seldom use that operator. Therefore no cost is added to pre-total-cost out of the operators that use up units and grant rewards, but rather the cost is carried on associated to the unit as explained before and added only in the last action of the unit. Such a design makes more emphasis in the differentiation between behavior and intermediate actions to the point that intermediate actions are not likely to be used unless the unit participant in that action gets assigned one of the possible behaviors afterwards. In practice, this means that units and resources will not be produced by the sake of it even if just having them is already more advantageous for the player than not doing so, but rather will have the role of a way to get a reward using a new unit to accomplish a more general task.

To sum up everything, the final value that must be minimized at the end of the plan is represented by an initial value minus the summation of the different rewards minus the costs obtained for each action. If a plan contains *n* reward-granting actions, the equation that express the value to minimize is the following:

$$totalCost = initialCost - \sum_{i=1}^{n} (reward_i - cost_i)$$

Figure 15: Metric to minimize

4.3.5 Abstraction

A common trait of modeling in computing independently of the context is abstraction. In many problems there are too many factors to deal with, making inviable to compute them, so a way of reducing the details of the problem without losing key information is needed. In planning domains, in which the computational order of problems is usually very high due to how the space-state is explored, abstraction is one of the main issues that can determine whether a problem is solvable or not. In the domain definition for game 3 several abstractions were used, some of them being the following:

- Only allied units and buildings will be treated individually. Enemy buildings will be represented by a fact “enemybase” independently of the existent buildings. Enemy military units will be associated as a fluent either to enemy bases (when defending them) or to allied control centers (when attacking them). Workers will not be factored in.
- Due to the characteristics of the game, tanks have been taken out. Based on their stats, it is not clear whether tanks are better options to soldiers cost-wise, so to simplify the problem they will not be used in planning. Of course opponents are free to use them, so to take them into account they will be represented as 4 soldiers.
- The map is constituted by a 64 by 64 grid of tiles, each one being a grid of 16 by 16 points (adding up to a world of 1024 * 1024 effective positions). This is too much information to be used in planning, so the map has been split in 64 areas to have a more coarse representation of the world.
- Only the areas relevant to the game state are taken into account. If an area has no objects and has been already explored it will not be included in the problem as there will be no way of interacting with it using planning.
- As the construction of control centers must be scripted, in the domain it is supposed that they are always close enough to mineral clusters and so minerals can be left out of the representation by associating the mine operator to control

centers instead to minerals specifying with a fluent how many workers can mine (as there is a 4 workers limit per mineral and usually not all the minerals in the cluster are accessible).

4.3.6 Object reutilization

Since its publication in 2002, several critics have been done to PDDL 2.1 One of the biggest controversies has been the limited capacity it has to represent sets of facts using fluents instead of enumerating them. For example, in PDDL 2.1 it is impossible to represent "3 soldiers", instead the definition of the problem needs an enumeration of the kind "soldier1 soldier2 soldier3". PDDL is highly based on STRIPS and classical planners have been designed following these lines as well, so it is a logical approach, but it greatly hinders its expressiveness. In RTS games units have several characteristics that differentiates one from another, if only just the position on the world, so such an approach is not a great problem as the units themselves are almost never equivalent. This is true however for the objects already in the world, but not for those that can be built or produced. If a soldier is going to be built, a fact of the type soldier that is not alive is needed to produce it. If it is three soldiers, three different facts are needed. Apparently this is not a great problem as replanning happens often and there is not time to produce a high amount of units each plan, so introducing a reasonable number of facts to be used for this purpose should not be a problem. Despite this, a difficulty lays in that planners treat the facts independently even if using one or another would lead to totally equivalent states (or plans, from a more general point of view). In most other contexts, the computer would simply pick one of the soldiers and would not worry about what could happen if it took another one, as they have the same characteristics. However in automated planning the most common behavior of the planner is calculating independent plans for all the facts that would fulfill the requirements of the operator even if there is no practical difference between using one or another. This means that for every fact in the database a new node is expanded and a subsequent plan may be calculated, multiplicatively increasing the number of states by the number of equivalent facts. Even worse, this happens not only at one point in the plan but rather every time the operator can be evaluated, so in the end the computational increase is of exponential order.

In game 3 soldiers, tanks, workers and their respective production buildings can be built, so the impact of this issue is high. To prevent this from happening object reutilization is used. As shown in the previous sections, operators that grant a reward and thus actively contribute to achieve the goal use up a unit. As in ORTS every unit must have a position, in the domain definition being alive means being at a particular position. When using up a unit, the fact is not taken out of problem or labeled as dead, instead the predicate that relates it to the position it was is deleted and thus the fact that represents the unit can be used to represent a newly produced one, having into account that the plans must be correctly interpreted as the output will show the same fact used several times when it really represents different units throughout the plan. As costs are assigned to units upon being produced, there is no problem with costs either.

This implementation does not solve the problem by itself: if several units are used up, the same situation occurs but this time with facts that represented old units. To overcome this, a predicate “ready” is used as a requirement in the unit production operators along with a logic that prevents having more that one unit of a kind ready at the same time: the “ready” predicate is assigned to a fact when a unit is used up only if there was no other unit at that time with that predicate already. This can be done in PDDL 2.1 using types and existential quantifiers, as done in the attack operator:

```
(not (at ?soldier ?src))
(when (not(exists (?x - soldier)(ready ?x))) (ready ?soldier))
```

This ensures that there is only one fact that can be used to represent a new unit, solving the aforementioned problem. As at least a behavior operator is needed to reach the goal, there will always be a ready unit of each type to be used, with the only case in which this predicate must appear in the problem definition being when there are no units of that kind alive at the moment.

4.3.7 Problem definition

In PDDL the problem definition is completely dependent on the domain definition, but

there are some issues that depend on how the problem is formulated. As the parsing is done differently by every planner, how restricted is the problem definition depends on the planner as long as the problem is well formed; for example, some parsers allow using not initialized fluents while others do not. Basically the problem definition is the initial state and the goal condition, but PDDL 2.1 also includes the possibility of adding a metric. Since the problems are automatically generated by the client in real time, they conform to the same structure:

- The objects are the enemy bases, the allied units and buildings and the relevant positions. If there is not at least a unit of a given kind, an object will be created (and latter assigned the predicate “ready”) so units of that type can be built and used. The relevant positions are those that will later have the predicates “at” or “unexplored” assigned, being all the other positions ignored.
- The fluents assigned to positions are “posX” and “posY”, which hold their Cartesian coordinates that are used for the calculation of the Manhattan distance, and “positionCost”, which is a measure of how costly is to go through the area. None of them change during the generation of the plans.
- All the general fluents are initialized. These include the rewards (attack-reward, defend-reward, scout-reward and harass-reward), the resources (minerals and minerals-gathered), the value of pre-total-cost, the cost increment for the operator TO-END and the reward threshold to reach the goal.
- The unexplored areas are listed using the predicate “unexplored”.
- The position of all the objects is listed using the predicate “at”.
- The fluents that represent the enemy forces are initialized.
- All the accumulated costs and assigned soldiers are initialized to 0.
- The goal (goal_achieved) is set.
- The metric to minimize (total-cost) is set.

5. USER'S MANUAL

This section describes the steps that must be followed to test the planning domain and to install ORTS and use the problem generating client. A Linux compatible operating system must be used, though Metric-FF works in any Unix system and ORTS can be successfully compiled to be used in Windows and Mac/OS.

5.1 Using the planning domain

Since the domain has been defined using PDDL, any planner supporting the features of PDDL 2.1 (the time features are not needed) can be used to test it. As the planner used for the experimentation has been Metric-FF, the steps to execute the domain with the problems used in testing will be the ones needed for it.

Metric-FF can be obtained from its site web [Metric-FF]. It can be used in any Unix system, and needs a Lex and Yacc compatible library. To compile it, a C compiler such as gcc and the makefile application are necessary. To generate an executable binary, just type “make” and a file named “ff” will be created in the same directory from where “make” was issued. The binary file is complete standalone and does not requires any of the files that are included in the source code package. Metric-FF has a simple set of options that allow parameterization of the input and output files, the search algorithm, the weight of the heuristics and the metric option. If Metric-FF is executed without options or with invalid options, the usage is displayed:

```
usage of ff:

OPTIONS   DESCRIPTIONS

-p <str>    path for operator and fact file
-o <str>    operator file name
-f <str>    fact file name
```

```
-E          don't do enforced hill-climbing try before bestfirst

-g <num>    set weight w_g in w_g*g(s) + w_h*h(s) [preset: 0]
-h <num>    set weight w_h in w_g*g(s) + w_h*h(s) [preset: 0]

-O          switch on optimization expression (default is plan length)
```

The domain and the problem can be in a single file using -p or in separated files, using -o for the domain definition and -p for the problem. As the domain will not change except when comparing using cost-to-operator conversion against not using it, the domain and the problems will be in separated files so changes in the domain apply to every problem. Metric-FF uses two search algorithms, first Enforced Hill-Climbing and second Best-First [Hoffmann, 2002]; the first one is unique to FF while the second is a widely accepted search algorithm. The -E option forces the planner to use just Best-First in case this is wanted. In this case, as Enforced Hill-Climbing has proven to be a rather performant algorithm, Best-First will not be used [IPC 2002]. The -g and -h options are used to change the weights of the path-cost function (the cost from the starting node to the current node) and the heuristic function (the estimated distance to the goal) respectively. By default their values are 1 for g(x) and 5 for h(x), and they allow to encourage exploration over exploitation by increasing g(x) and decreasing h(x) and exploitation over exploration when doing the opposite. Finally, the -O option tells the planner that there is a metric to optimize. If the -O option is not used but there is a metric specified in the problem definition, it will be ignored. The default optimization is plan length. All in all, the calls for the experiments will be like the following one:

```
./ff -o game3.PDDL -f game3-problem.PDDL -O
```

5.2 Installing and using ORTS

ORTS is released under the GPL license [GPL] and therefore is free for everybody. There are two ways of getting ORTS. The first one is downloading the daily development snapshot [ORTS snapshot] found in the ORTS web page. The file is a compressed tarball

that must be uncompressed before using. The second one is obtaining it using the subversion server available for development: it is accessible not only to registered users but also to anonymous users by providing a public user name and password. A subversion client is needed to do this. Using the svn application widely available in Unix platforms, the way of getting ORTS would be this one using “guest” as password when prompted:

```
svn co svn://anonymous@bodo1.cs.ualberta.ca:/all/pubsoft/orts
svn co svn://anonymous@bodo1.cs.ualberta.ca:/all/pubsoft/orts_data
```

In either case ORTS_DATA can be obtained or not, as it contains the graphical part of the 3D user interface and thus it is not necessary if it is not going to be used.

Once downloaded, ORTS must be compiled. ORTS has been developed under Linux and works best in that operating system, but can be compiled in Windows using VC++ and in Mac/OS. Several external libraries are needed before compilation: bjam, boost (libboost-dev), glew (with the development package) and sdl-net-dev for the standard ORTS build and OpenGL, glu and glut (or mesa, both of them with the development package too) for the graphical part. Makefile and a C++ compiler (as G++) are also needed. Prior to compilation, the environment variable OSTYPE must be one of the accepted by ORTS. It is changed using different commands depending on the OS, being in a bash shell in Linux "export OSTYPE=LINUX", for example. Some of the accepted values are LINUX for linux-gnu, DARWIN for Mac, CYGWIN for cygwin and MSYS for MinGW. Once created, from the folder orts3/trunk “make init” must be called to create the directory structure, then simply “make” to compile ORTS and both the server and a client with the 3D graphical user interface implemented (named orts and ortsg respectively). All the binaries will be deployed in the orts/trunk/bin folder. ORTS can be tested by opening two terminal and executing “"bin/orts" in the first one and "bin/ortsg" in the second one once the server is waiting for a player; this will show a single-player recreation of the commercial RTS game Starcraft using the 3D graphical interface to show the features of ORTS.

For the graphics client under Linux to reach frame rates over 20 fps currently a mid-range NVIDIA graphics card is needed because the Linux ATI drivers do not support it. Under

Windows there is no such restriction. Having at least 128 MB of video RAM is mandatory and 512 MB of system RAM is minimum when using the graphical user interface developed by the creators of ORTS.

Although clients can be designed as the user prefers, it is recommended to use the available libraries that come with ORTS and part from an already functioning client such as sampleai. ORTS has a version of makefile which is designed to ease the compilation of clients using ORTS libraries. The client must be in a folder or subnested folder in the orts/trunk/apps folder (which determines the name of the client: for example, apps/orts produces the application orts while apps/rtscomp07/game3/uofa produces the application rtscomp07.game3.uofa) containing a src folder where the file containing the main method and other files to be compiled are placed. An "app.mk" file is needed, which is a makefile include file that can use standard Makefile syntax. The first non-comment/whitespace line should be "APP_DIR := " followed by the name of the client as specified above, for example "APP_DIR := rtscomp07.game3.uofa". The second line should be "APP_LIBS := " followed by a space-delimited list of ORTS libraries that must be included. These represent folders in the "libs" subdirectory of the primary ORTS directory whose contents will be made available for include and compilation when building the client. For instance, "kernel" provides access to the .C and .H files in "libs/kernel/src", while "ai/movement" provides access to the files in "libs/ai/movement/src". Almost every client will need at least "kernel", "network", and "serverclient", and if graphical support is needed, "gfxclient".

After this is done, the client can be built by typing "make" followed by the application name, for example "make rtscomp07.game3.uofa". The resulting binaries will be placed in orts/trunk/bin folder as happened whith orts and ortsg when compiling the whole platform, and will have the same name as the application that was compiled. Note that there are a few different ways of compiling an application, with the most often used being debug mode and optimised mode. For instance, "make MODE=opt ortsg" will compile ortsg with optimisation flags, while "make MODE=dbg ortsg" will compile ortsg with debug flags. Debug mode is the default, so specifying it is redundant.

There are plenty of options when running ORTS; besides every client has the option to add additional ones for their own parameters. Thus, here only those related to running the third game will be described. To execute the server for game3, the script

orts/trunk/tournament-2008/game3_orts (which actually is just executing "./bin/orts -game3 -bp tournament-2008/game3.bp -x 64 -y 64 -fplat 0.08 -nplayers 2") can be run, and to test it (or play it) with the 3D interface the script orts/trunk/tournament-2008/game3_ortsg (which again is just executing "bin/ortsg -p tournament-2008/ortsg -refresh 0") can be used.

6. RESULTS

This section reports the experimentations made with the domain and a set of problems produced to test the functionality of the design. As there are many features and parameters that have a significant impact on the results, every subsection will try to be centered on a single aspect so conclusions can be brought exclusively upon it without other issues factoring in. All the experimentations will be done using a 3,5 Ghz Dual Core Intel Pentium Processor with 512 MB RAM under Ubuntu Linux.

6.1 Cost-to-operator conversion and metrics

In state-of-the-art planners, plan quality is becoming increasingly important in contrast to other aspects traditionally regarded as more important as the performance or the number of problems solved. However, in many cases plan quality is measured by minimizing a value that is modified by non-constant effects in the operators; that is, it is modified depending on the state, and this is something state-of-the-art planners do not support. To solve this, and as it was mentioned in section 4.3.4, cost-to-operator conversion was used. The experiments in this section will be using a regular domain in which the goal is simply getting a given number of rewards, using cost-to-operator conversion without using the metric (minimizing the converted fluent that contains the final cost) and using cost-to-operator conversion using the metric, hoping to get better plans for the last option. In this case the problem to solve is a simple problem in which there are four soldiers, three workers, a control center (not being attacked by any enemy soldier), a barracks, an unexplored area and an enemy base with 3 defending soldiers.

This is the plan obtained without cost-to-operator conversion:

```
step    0: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
        1: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
        2: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
        3: END
```

```
        0.00 seconds searching, evaluating 62 states, to a max depth of 0
        0.00 seconds total time

total cost: -352.00
```

As it can be seen, it is a straightforward plan that uses the most rewarding action (attack) and ends once the goal condition is satisfied. The next experiment will implement cost-to-operator conversion using a standard plan length metric instead of minimizing the final cost.

```
step    0: SCOUT WORKER3 POS5-1 POS6-2
     1: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
     2: HARASS SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
     3: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
     4: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
     5: TO-END
     6: TO-END
     7: TO-END
     8: TO-END
     9: END

        0.00 seconds searching, evaluating 16 states, to a max depth of 2
        0.00 seconds total time

total cost:   -1.00
```

In this case, more rewarding actions than in the case before are used, so this is in terms of quality a better plan. As the (cost-increment) fluent used in the TO-END operator has a value of 100, the cost is between 300 and 400 and most likely below 352, which was the final cost in the previous case, so though better the current plan is not significantly better. In the next case the metric (total-cost) will be used:

```
step    0: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
        1: ATTACK SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
        2: MINE WORKER3 CC0 POS5-1 POS4-2
        3: MINE WORKER0 CC0 POS5-1 POS4-2
        4: MINE WORKER1 CC0 POS5-1 POS4-2
        5: BUILDSOLDIER BARRACKS0 POS5-1 SOLDIER4
        6: SCOUT WORKER2 POS2-5 POS6-2
        7: ATTACK SOLDIER4 ENEMYBASE0 POS5-1 POS1-1
        8: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
        9: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
       10: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
       11: TO-END
       12: END

            0.40 seconds searching, evaluating 1102 states, to a max depth of 0
            0.44 seconds total time

total cost:  100.00
```

Now the plan has changed much from the previous one. First of all, it must be noted that the cost is below 100 (it displays 100 because TO-END adds the value of (cost-increment) in the last call independently of the cost left in (pre-total-cost)) which is a much better result. Second, here the planner has used non-rewarding operators (MINE and BUILDSOLDIER) to build an additional soldier and get more rewards, which shows the real improvement over the previous cases. Why BUILDSOLDIER uses SOLDIER4 is because of how object reutilization is implemented as described in section 4.3.6. The drawback of using the metric along with cost-to-operator conversion is the increased number of evaluated nodes: in this case a total of 1102 states have been evaluated as opposed to 62 with the standard design and 12 when using cost-to-operator conversion but not the metric. This is a crucial fact as it took 0.44 seconds to solve a problem that was solved in the previous cases in milliseconds, which creates the usual trade-off between solution cost and time to compute. Another important fact that may be an explanation of why the search space is so much bigger is that while plan length allows using Enforced Hill-Climbing as the search algorithm, when using the final cost as metric Metric-FF uses a

best-first algorithm, which for Metric-FF usually yields worse results.

In this comparison, the metric used is favorable to the third case; that is, minimizing the final cost, because when using plan length there is no improvement unless an operator grants a reward greater than (cost-increase), or a sequence of operators than (cost-increase) multiplied by the number of actions. On the other hand, in the third case it does not matter how many actions are required to decrease the cost even if by a single one (cost-increase). This explains why using the metric leads to such a huge : while when using plan length all the nodes that do not grant a good enough reward are not expanded, when using the metric almost all the nodes are expanded as long as they may contribute to improve the quality of the plan, obtaining therefore such a big search space compared to the other cases. A simple way of proving this is using a lower value for (cost-increase), which should lead to getting results in both size and quality to the case in which the total cost is used as metric. The next plan is the one obtained using a (cost-increase) of 25 and plan length as metric:

```
step    0: SCOUT WORKER3 POS5-1 POS6-2
        1: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
        2: ATTACK SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
        3: HARASS SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
        4: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
        5: TO-END
        6: TO-END
        7: TO-END
        8: TO-END
        9: TO-END
       10: TO-END
       11: TO-END
       12: TO-END
       13: TO-END
       14: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
       15: TO-END
       16: END
```

```
        0.01 seconds searching, evaluating 74 states, to a max depth of 8
        0.02 seconds total time

total cost:   -1.00
```

Not surprisingly, this plan is better as it uses a reward granting operator more than in the previous case but in return it evaluates 74 states instead of only 12, which proves the aforementioned point. As a side note, here the cost is negative because its value is displayed as -1 by default when using plan length as metric. Another thing worth mentioning is that operators different than TO-END are used even when it has already been used, which means that the planner tries to improve the plan even when the real goal has already been achieved. To better illustrate this, an even lower value will be used, a (cost-increase) of only 1.

```
step    0: HARASS SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
        1: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
        2: SCOUT WORKER3 POS5-1 POS6-2
        3: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
        4: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
        5: MINE WORKER2 CC0 POS2-5 POS4-2
        6: MINE WORKER1 CC0 POS5-1 POS4-2
        7: MINE WORKER0 CC0 POS5-1 POS4-2
        8: BUILDSOLDIER BARRACKS0 POS5-1 SOLDIER3
        9: TO-END
       ...
      108: TO-END
      109: ATTACK SOLDIER3 ENEMYBASE0 POS5-1 POS1-1
      110: TO-END
      ...
      118: TO-END
      119: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
      120: TO-END
      121: END
```

```
1.09 seconds searching, evaluating 1778 states, to a max depth of 60
1.09 seconds total time

total cost:   -1.00
```

In the most extreme of the cases, the behavior of the planner is very similar to the case in which the total cost is used. As it can be seen, the plan is different from the other case (a different worker is used, here a soldier harasses while in the other case all the soldiers attack) and the total cost, calculated counting how many times TO-END has been used, is 109, greater than when using final cost, which was below 100. This added to the fact that the values of the space-state and time are bigger (1102 versus 1778 and 0.44 versus 1.09 respectively) shows that using the metric seems to be the better option, but more experimentation is needed. As the most extreme of the cases have been used, there is no information in how both quality and performance evolve when decreasing the value of (cost-increase). Thus, the next chart shows a series of experiments for values of (cost-increase) ranging from 25 to 1.

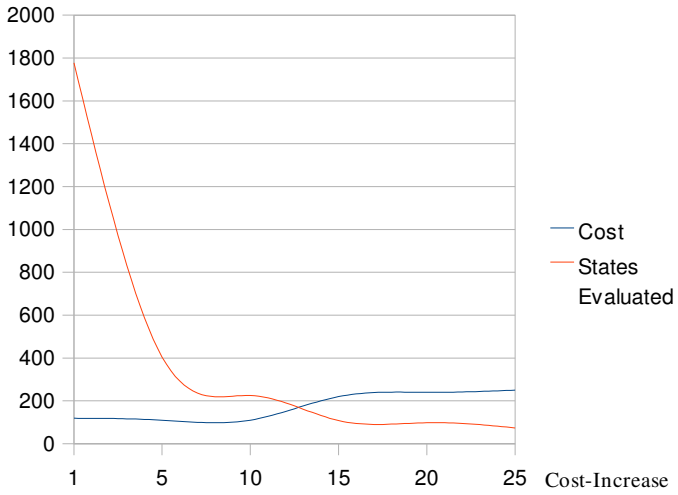

Figure 16: Cost-Increase impact

Figure 16 shows how from a value of 15 for (cost-increment) there is no gain in cost optimization while the number of evaluated nodes grows exponentially. For a value of 15 it

takes 0.04 seconds to solve the problem evaluating 118 states, which is preferable to the case in which the final cost was used as a metric.

As the optimization method used by the planner is essentially the same for both cases, it is tempting to try the same thing when using the metric. However the case is different here, as the plan that we had obtained before was already very good and the number of spaces that were evaluated is already high: as (cost-increment) decreases there is no change in the number of states until it is 64, in which case the number of states go up from 1102 to 1856 while coming up with the same plan. Besides, when it reaches 59 the evaluated states are 148612 (needing 107.21 seconds to finish) again without any improvement but this time with two calls to TO-END, which means that the real cost is 60. This means that the plan obtained the first time is probably the best one the planner can get in any condition, so it makes no sense to keep on decreasing (cost-increment) to try to achieve good results. Instead, the opposite will be tried; that is, increasing its value to see how the plans deteriorate in quality. In this case it is not until (cost-increment) is 251 that the plan worsens slightly after having evaluated 1102 states, just one fewer than the best case and curiously the same that the ones evaluated when using a (cost-increment) of 25 using just plan length instead of the cost as a metric.

The former results may mean that both metrics are probably equivalent for every problem given the correct parameters. To test this, the problem will be made harder by eliminating the barracks and initially giving to the planner the amount needed to build it. In this case, when using the plan length with a cost increment of 15 (which was the optimal amount before) it solves the problem with an equivalent plan to the best plan obtained so far (but this time building the barracks, obviously) evaluating 1395 states, noting that when the cost increment is 19 (8941 states evaluated) or bigger, the plan is worse as the barracks is not built and no new soldiers are produced. On the other hand, the results when using the metric are much worse: if cost increment is 125 or less, the planner is not able to finish in 600 seconds, limit time for the experiments, if only because the purpose of the design is to be valid in real time environments. With a cost increment of 126, it generates the following plan:

```
step    0: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
        1: MINE WORKER0 CC0 POS5-1 POS4-2
```

```
 2: BUILDWORKER CC0 POS4-2 WORKER0
 3: MINE WORKER1 CC0 POS5-1 POS4-2
 4: BUILDWORKER CC0 POS4-2 WORKER1
 5: MINE WORKER2 CC0 POS2-5 POS4-2
 6: BUILDWORKER CC0 POS4-2 WORKER2
 7: MINE WORKER3 CC0 POS5-1 POS4-2
 8: BUILDWORKER CC0 POS4-2 WORKER3
 9: MINE WORKER1 CC0 POS4-2 POS4-2
10: BUILDWORKER CC0 POS4-2 WORKER1
11: SCOUT WORKER0 POS4-2 POS6-2
12: BUILDWORKER CC0 POS4-2 WORKER0
13: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
14: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
15: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
16: ATTACK SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
17: TO-END
18: END

    0.24 seconds searching, evaluating 799 states, to a max depth of 0
    0.28 seconds total time

total cost:  126.00
```

In this plan several workers are created and sent to mine even if that grants no rewards as opposed to building soldiers from a newly created barracks and sending them to attack. This is probably indicative that, when it disregards plan length, the planner tends to consider in depth plans which may not be useful at all for both achieving the goal and optimizing the metric. In this case for example the planner expands nodes that involve creating workers because it has the resources to do so, this is, the prerequisites for the operators are satisfied, and not because it uses them to improve the plan. Being this so, using the metric to try to get above average quality plans is not a good solution as it is not restrictive enough when exploring the search space.

To try to make the problem harder in different way, instead of eliminating the barracks some enemy soldiers attacking the allied control center will be added. In this case even if the problem is not much harder (just a single soldier attacking the enemy base) the situation is the same as before: using plan length the problem is solved with a very good plan evaluating just 92 states using an increase-cost of 15, while using the final cost means that the planner stagnates and does not solve the problem. To shed a bit of light on this, an analysis of how the planner works when searching through the search space must be done. As explained in section 2.3.2, heuristic planners use two functions to determine how good a node is: *g(x)*, which is the weight from the initial state to the current node, and *h(x)*, which is a heuristic function that guesses how far from the goal the current node is. When using plan length, both functions are used just like in every other planning problem, considering shorter plans as better and more easily computable; on the contrary, when using the final cost as metric, *g(x)* is virtually unused until the firsts calls to TO-END are made, as it is only relevant for that operator. The problem with this is that if a node is far from the goal and there is no possibility of using TO-END because the rewards-threshold has not been reached, the evaluation of the node is very inaccurate as there is almost no information of how good it is. Besides as *g(x)* is not used, plan length does not matter and the planner deepens expanding nodes that may not be useful for the plan but that are not considered as worse as the value of *g(x)* is still 0. A possible solution is allowing the planner to use TO-END from the very beginning so both *g(x)* and *h(x)* are meaningful throughout the whole plan. To do so, the hard goal, the minimum amount of rewards specified by (rewards-threshold) will be set to 0. This may allow the generation of very low quality plans, but at least the heuristic function should be more accurate. This time, the same problem which was not solved before (basic problem plus one attacking enemy soldier) will be tested with a value of 0 in (rewards-threshold):

```
step    0: MINE WORKER3 CC0 POS5-1 POS4-2
        1: MINE WORKER2 CC0 POS2-5 POS4-2
        2: SCOUT WORKER0 POS5-1 POS6-2
        3: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
        4: MINE WORKER1 CC0 POS5-1 POS4-2
        5: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
        6: ATTACK SOLDIER0 ENEMYBASE0 POS4-2 POS1-1
        7: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
        8: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
```

```
 9: ATTACK SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
10: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
11: TO-END
12: END


    0.19 seconds searching, evaluating 1915 states, to a max depth of 0
    0.24 seconds total time


total cost:  100.00
```

Now the result is surprising: not only it was able to solve the problem but it has also given quite a good plan (if it had used a soldier to defend the plan would be better, but since it would have meant a call to TO-END in either case, for the planner this plan is as good as the optimal one). The search space, composed of 1915 evaluated states, is reasonable, although still not as good as when not using the final cost as metric. Curiously enough, when decreasing (cost-increase) hoping to get a finner plan, at 50 the best plan is obtained evaluating in this case just 1865 states, so both the plan quality and number of evaluated nodes are better. This result questions whether the use of a hard goal has any positive impact in either plan quality or performance. If this value is increased, once its value is above the rewards the first non-intermediate operator grants (for this particular problem and parameters SCOUT, granting 25 rewards and used always in third place after MINE being used twice) again it is unable to solve the problem in less than 600 seconds. It seems then that initially the planner tends to prefer non-rewarding operators (this can be seen as well in a previous plan obtained when testing the different values of increase cost) and unless the goal is reached with a single call to an operator that grants a reward it expands nodes evaluating intermediate operators while possible. To check if this happens as well when using plan length as metric we will compare the number of states evaluated and the quality of the plans for different values of reward threshold:

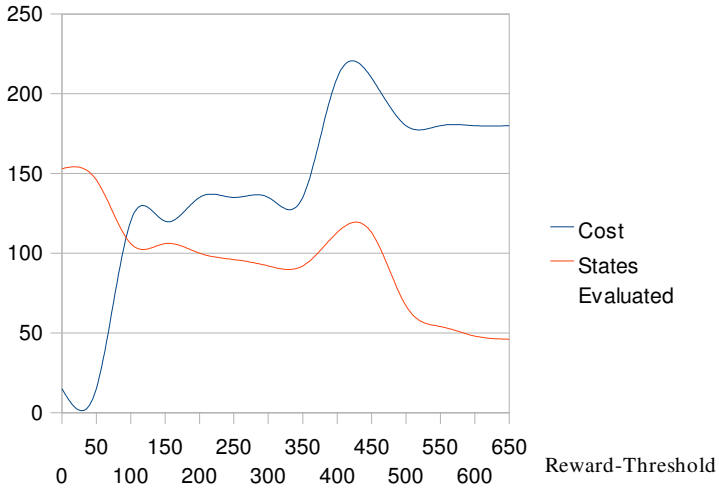

Figure 17: Reward-Threshold impact

In this chart it can be seen that as reward-threshold increases, the cost increases and the number of evaluated states decreases slightly, which was the opposite than expected, as setting a minimum quality should mean getting better overall plans. Besides, when rewards-threshold gets closer to the maximum rewards that can be obtained with the optimal plan, the number of evaluated states rises up, having 32531 evaluated nodes when reward-threshold is 680. All in all, the hard goal not only does not contribute to achieve a minimum quality in the plans but also may complicate the problem resolution using both metrics.

Now that it was cleared up that reward-threshold was hindering the final cost metric, the wisest decision would be taking it out of the domain definition. However if that is done the situation reverts to when rewards-threshold was being used, this is, intermediate operators are preferred over reward-granting ones. To make sure that this is not due to the way Metric-FF relaxes tasks in its heuristics functions when dealing with numeric effects [Metric-FF], a dummy predicate was added substituting the rewards-threshold condition and in the initial state so it is satisfied from the beginning. In this case the results are even worse, so the main point here is that rewards-threshold is relevant for the domain. Something that may be not clear and that should be noted is that if the initial amount of obtained rewards is 0 and rewards-threshold is 0 as well, *the hard goal is not satisfied* because the comparison is strictly greater than rewards-threshold. Actually the optimal value for rewards-threshold is any value in the range from 0 to the estimated amount that

the first reward-granting operator that grants the smallest reward gives (as the experiments showed before), *so the condition can be fulfilled early in the plan but not from the beginning*. If we give rewards-threshold a value of -1, so the goal is satisfied from the beginning, again the result is much worse, confirming this fact. When instead of using the final cost metric, plan length is preferred, there is no significant change in using or not such a value, though plans are slightly better and the number of evaluated states is slightly smaller for a value of 0 compared to a value of -1, so the domain and the rewards-threshold will be the same for both metrics in further experimentation. The cause of this phenomenon is unknown and further experimentation should be done after analyzing in depth the inner functioning of Metric-FF; however, the objective of this work is to implement a domain as general as possible so it can be used by any planner and thus particular issues to a given planner, although interesting for research, are out of the scope of the work.

Now that we know that the optimal value for rewards-threshold is 0 in probably most of the cases we will go back to the three problems used for experimentation and solve them again using this value. A fourth problem combining the lack of barracks and the attacking enemy soldier will be used as well for a more difficult situation. As a reminder, the basic problem consists on four soldiers, three workers, a control center (not being attacked by any enemy soldier), a barracks, an unexplored area and an enemy base with 3 defending soldiers. The optimal value of increase-cost will be used for every case, so the displayed results will be the best case out of a series of experiments with different values of increase-cost.

Problem	Plan length	Final cost
Basic	Best plan, 192 evaluated states, cost increment of 10.	Best plan, 359 evaluated states, cost increment of 170.
Basic with attacking enemy	Best plan, 153 evaluated states, cost increment of 20.	Best plan, 1865 evaluated states, cost increment of 50.
Basic without barracks	Best plan, 207 evaluated states, cost increment of 15.	Best plan, 547 evaluated states, cost increment of 175.
Basic with attacking enemy and without barracks	Suboptimal plan, 249 evaluated states, cost increment of 15.	Bad plan, 52934 evaluated states, cost increment of 200.

Table 3: Comparison of metrics in simple problems

This table clearly shows that the conventional plan length metric outperforms the final cost metric in all the cases and with more constant values of cost-increase (which is a great advantage if they have to be generated dynamically while replanning). Particularly important is the most complex problem, in which the output of the planner for the final cost metric deteriorates to the point of returning a bad plan in which a behavior similar to that observed with high values of rewards-threshold is obtained apart from evaluating a much higher number of states:

```
step    0: MINE WORKER3 CC0 POS5-1 POS4-2
        1: MINE WORKER2 CC0 POS2-5 POS4-2
        2: BUILDWORKER CC0 POS4-2 WORKER3
        3: SCOUT WORKER3 POS4-2 POS6-2
        4: BUILDWORKER CC0 POS4-2 WORKER3
        5: MINE WORKER0 CC0 POS5-1 POS4-2
        6: BUILDWORKER CC0 POS4-2 WORKER0
        7: MINE WORKER1 CC0 POS5-1 POS4-2
        8: BUILDWORKER CC0 POS4-2 WORKER1
        9: MINE WORKER0 CC0 POS4-2 POS4-2
       10: BUILDWORKER CC0 POS4-2 WORKER0
```

```
11: ATTACK SOLDIER0 ENEMYBASE0 POS5-1 POS1-1
12: ATTACK SOLDIER1 ENEMYBASE0 POS5-1 POS1-1
13: ATTACK SOLDIER2 ENEMYBASE0 POS2-5 POS1-1
14: ATTACK SOLDIER3 ENEMYBASE0 POS4-2 POS1-1
15: ATTACK SOLDIER4 ENEMYBASE0 POS3-3 POS1-1
16: TO-END
17: END

    16.22 seconds searching, evaluating 52934 states, to a max depth of 0
    16.27 seconds total time

total cost:  200.00
```

If we take a look at the first part of the plan we can see that intermediate operators are extensively used for no good reason, an analogous situation to that obtained with restrictive hard goals. If increase-cost is decreased to get a finner plan, the planner is not able to solve the problem in less than 600 seconds. This means that the final cost metric, although initially promising, is less adequate in all the problems and useless for more complicated ones. This might be not because of the metric itself but because of not being to use Enforced Hill-Climbing when minimizing a numeric value, but best-first has demonstrated in several occasions to be a rather consistent search algorithm and the difference is just too big for this to explain the results.

A last thing that must be taken into account is that the two fluents used as parameters in the experiments, rewards-threshold and cost-increment, are not absolute values, but depend on the rewards and the costs used by the operators. If the rewards and operators were to change, so should do rewards-threshold and cost-increment to achieve the same results, this is, they are relative parameters and not general to all the possible problems.

6.2 Problem complexity

In the earlier experiments a very simple problem was used to compare metrics. However this domain is meant to be real life applications and so it will face harder problems. One of the main points in computing when solving problems independently of the field is seeing how time and memory needed to solve them scale as the difficulty of problems rise up. The difficulty of problems depends on multiple factors and analyzing how they affect the complexity is usually very insightful as it allows to redesign the solving method to optimize the performance of the used technique. In this case, each potential factor will be analyzed independently so the results are more accurate.

6.2.1 Initial value of pre-total-cost

As seen in the domain definition in section 4.3.4, the value to minimize if the number of reward-granting actions in the plan is *n* is

$$totalCost = initialCost - \sum_{i=1}^{n} (reward_i - cost_i)$$

All the values in the equation are parameterizable, but rewards and costs are associated to operators and units and their values are not constant. Initial cost, on the other hand, is assigned once to the fluent pre-total-cost at the beginning of the problem. As it was mentioned when described, initial-cost should leave a margin of improvement so at least in the end a TO-END call is done but should be limited so no unnecessary uses of TO-END happen too. Ideally the value should be between the rewards minus the costs of the best plan and that value plus the value of cost-increase. Obviously, the best plan is unknown, so the value must be guessed instead. Since in the problem definition the maximum rewards are calculated for each of the operators and they are exclusive meaning that if a soldier is used to attack it can be used to harass, for example, a possible upper limit to the value can be calculated by multiplying the number of units of each type that exist (plus those that could be produced) by the highest of the maximum rewards of the actions they can perform. Directly using this upper limit is probably a bad idea since if the planner is able to produce a very good plan it could be too high or if the planner does not use all the available units it could be too low, but it is a good guideline in which to base our assumptions.

Theoretically the main reason why this value could alter the performance of the planner is that the planner does not restrict itself once the hard goal has been achieved and tries to improve the plan using operators even between calls to TO-END. Since plans are sequence of actions, these operators can be used at different points, leading to equivalent states but generating additional branches in the search space that may be explored by the planner increasing the time needed to return a suboptimal plan. Generally, the shorter the plan the less likely is this to happen as there are fewer points in which the planner may decide to further explore the state-space instead of keeping using TO-END, so if additional calls to TO-END are needed a downtime in performance may occur. To test this the most complex of the problems used in the previous section will be solved using different initial values for the cost, showing them in a chart to illustrate the evolution of the size of the number of evaluated states. For this problem an almost optimal value is 800, so the experiments will depart from that value using the plan length as metric and a cost-increment of 15 (meaning that every 15 units an additional call to TO-END is needed):

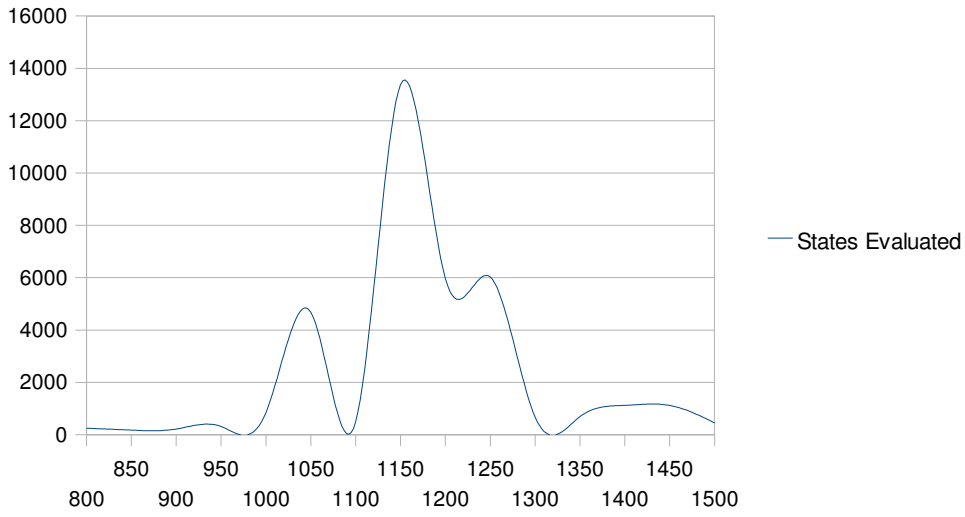

Figure 18: Initial-cost impact

As shown in the chart, the results do not seem to follow a particular distribution. Early conclusions may lead to think that the distribution is semi-Gaussian, but they vary greatly depending on the problem without a clear link between the results. The plans

were all either the best one or the immediate next one, so the quality did not decrease. Something that must be noted is that for the last case, an initial cost of 1500, a total of 59 calls to the TO-END operator were done as opposed to 14 calls to "real" operators, an already high proportion, so it is not clear whether additional calls can negatively affect the result with a high probability. Just to check the behavior of the final cost metric, an easier problem (basic problem without barracks with a cost increment of 175) will be solved in a similar way. When done, a curious fact is observed: even if no calls to TO-END are needed, the space-state size greatly increases, and when a single additional call is needed the planner is not able to solve the problem in a reasonable time. Again this seems to confirm the conclusions of the previous section, that when using the total cost metric the planner tends to wander around the search space due to the uselessness of the $g(x)$ function in the heuristic.

6.2.2 Number of units and other in-game parameters

Usually in RTS games a great number of units exist at the same time. This is considered one of the main computational problems RTS games have, as not only new actions must be calculated for each unit but also as they can interact among them more targets for every action exist. Theoretically if a unit of a type can perform only one action having as a target a unit of the same type, n actions with $(n-1)$ possible targets can be possible, which would lead to a complexity of $O(n^2)$ in the worst case. If there is more than a single action it affects polynomially the situation, the same case as if there are several targets for each action. Then, if there are a actions with t targets and n units, the final complexity in the worst case would be $O(an^{t+1})$, a potentially high order, which is why the number of units and actions matters so much to the game designers.

In the current domain, the number of actions and targets are limited, 2 or 3 actions per type of unit which have as target either an enemy base (ATTACK and HARASS), a control center (MINE), an unexplored location (SCOUT) or nothing at all (BUILDBARRACKS). The number of enemy bases and control centers will be more likely below 2 or 3 during the whole game, so they are not important factors. Consequently, the experiments will be done changing the number of soldiers, workers and unexplored areas. The number of minerals (both at the initial state and those

gathered throughout the plan) can also affect the state-space size as they can be used to produce additional units, so two measures will be taken: first, when experimenting with other factors the number of initial minerals and those obtained per MINE action will always be 0 so they do not alter the results; second, a series of experimentations will be done using exclusively these two values as it will be done with the other factors as well.

The basic problem is very limited because it does not grant enough rewards for the planner to try to use all the available soldiers if their number is too high, so the number of possible targets for the actions that soldiers can perform will be increased. In total there will be two enemy bases and three allied control centers, being defended by 3 and 4 soldiers and attacked by 2, 1 and 4 soldiers respectively. Other than that workers will be taken out and there will not be a barracks to force the planner to focus solely on military actions. The initial cost for pre-total-cost may change because the more soldiers are, the more rewards can be obtained and leaving it with a constant value would mean that the results for the problems with few soldiers are worse than what they really are. The results will be displayed in the following chart:

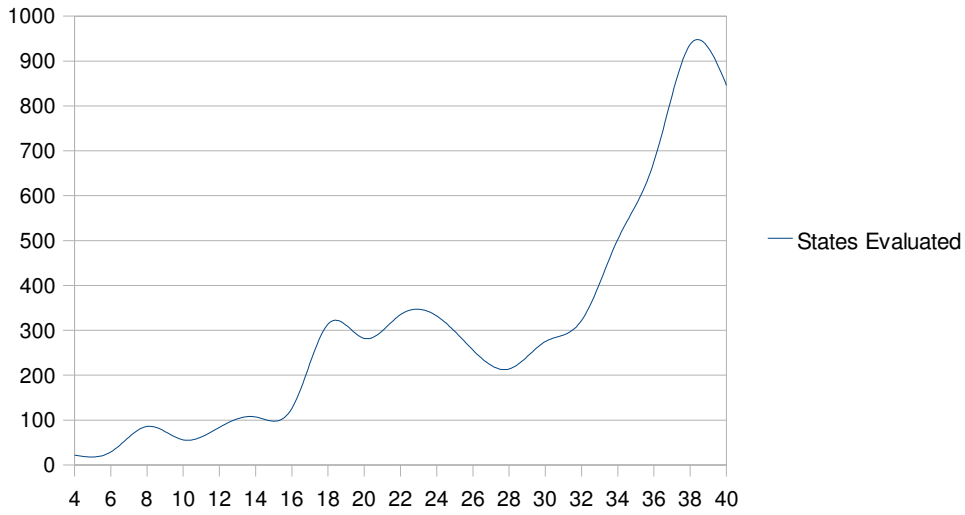

Figure 19: Number of soldiers impact

As expected, the size of the number of evaluated states increases as the number of soldiers goes up, but its impact is not important. The last of the experiments, with 40 soldiers, took less than 5 seconds to be solved, which is a fairly good result taking into account that 40

soldiers is already a rather high number for a RTS game. Besides the number of evaluated nodes seems to scale linearly instead of exponentially, probably the most important point when extrapolating this design to more complex problems. This can probably be explained by the way plans are elaborated: in most of the cases, the soldiers were used sequentially, meaning that they were usually selected depending on their order in the parsing of the problem by the planner. The following plan is a very clear example, using almost the 40 soldiers in inverse order:

```
step    0: HARASS SOLDIER39 ENEMYBASE0 POS4-2 POS1-1
        1: TO-END
        2: DEFEND SOLDIER38 CC2 POS2-5 POS7-5
        3: DEFEND SOLDIER37 CC2 POS5-1 POS7-5
        4: DEFEND SOLDIER36 CC2 POS5-1 POS7-5
        5: DEFEND SOLDIER35 CC2 POS7-3 POS7-5
        6: DEFEND SOLDIER34 CC2 POS3-3 POS7-5
        7: DEFEND SOLDIER33 CC2 POS4-2 POS7-5
        8: DEFEND SOLDIER32 CC2 POS2-5 POS7-5
        9: DEFEND SOLDIER31 CC1 POS5-1 POS3-3
       10: DEFEND SOLDIER30 CC0 POS5-1 POS4-2
       11: DEFEND SOLDIER29 CC0 POS4-2 POS4-2
       12: DEFEND SOLDIER28 CC0 POS2-5 POS4-2
       13: DEFEND SOLDIER27 CC0 POS5-1 POS4-2
       14: DEFEND SOLDIER26 CC0 POS5-1 POS4-2
       15: DEFEND SOLDIER25 CC1 POS7-3 POS3-3
       16: ATTACK SOLDIER24 ENEMYBASE0 POS3-3 POS1-1
       17: ATTACK SOLDIER23 ENEMYBASE0 POS4-2 POS1-1
       18: ATTACK SOLDIER22 ENEMYBASE0 POS2-5 POS1-1
       19: ATTACK SOLDIER21 ENEMYBASE0 POS5-1 POS1-1
       20: ATTACK SOLDIER20 ENEMYBASE0 POS5-1 POS1-1
       21: ATTACK SOLDIER19 ENEMYBASE0 POS4-2 POS1-1
       22: ATTACK SOLDIER18 ENEMYBASE1 POS2-5 POS6-0
       23: ATTACK SOLDIER17 ENEMYBASE0 POS5-1 POS1-1
       24: DEFEND SOLDIER15 CC2 POS7-3 POS7-5
```

```
25: ATTACK SOLDIER16 ENEMYBASE0 POS5-1 POS1-1
26: ATTACK SOLDIER14 ENEMYBASE0 POS3-3 POS1-1
27: ATTACK SOLDIER13 ENEMYBASE0 POS4-2 POS1-1
28: ATTACK SOLDIER12 ENEMYBASE1 POS2-5 POS6-0
29: DEFEND SOLDIER11 CC0 POS5-1 POS4-2
30: DEFEND SOLDIER9 CC0 POS4-2 POS4-2
31: TO-END
32: ATTACK SOLDIER8 ENEMYBASE1 POS2-5 POS6-0
33: DEFEND SOLDIER5 CC2 POS7-3 POS7-5
34: ATTACK SOLDIER2 ENEMYBASE1 POS2-5 POS6-0
35: DEFEND SOLDIER10 CC0 POS5-1 POS4-2
36: TO-END
37: ATTACK SOLDIER7 ENEMYBASE0 POS5-1 POS1-1
38: TO-END
39: END

    2.54 seconds searching, evaluating 846 states, to a max depth of 2
    3.46 seconds total time

total cost:   -1.00
```

This may mean that costs that depend on pathfinding do not have an important impact and that the planner practically ignores them. Actually the plans are coherent and tend to use the units appropriately, which leads to think that it is not the case. Nevertheless, to check if this is true, costs were increased by multiplying by 10 the cost obtained with the Manhattan distance. When this was done the behavior was practically the same but the number of evaluated states was bigger, probably because of the higher level of constraint that greater costs impose to the selection of operators, so this is probably an inner issue of Metric-FF instead of a consequence of the way costs are implemented.

Despite the previous positive results, there is a very important issue that the chart does not reflect: every experiment has been done using a suboptimal value of increase-cost, so good results are to be expected. Furthermore, as the number of soldiers increases, the problem

becomes more easily influenced by the variation in the initial cost to the point that for the experiments with more than 30 soldiers a too big value could rend the problem unsolvable because the planner used too much memory. Besides this happened with differences in the value of the initial cost as small as 200 when the optimal value was around 5000, which implies that the selection of the value of the initial cost is not only critical for the resolution of the problem but also a very hard task if generated dynamically because of the small margin where it becomes useful. Of course, the initial cost depends largely on the other related parameters as the rewards, the costs and increment-cost, so the problem it is not associable exclusively to this variable, but this shows how often the parametrization and not the design of the solution is the hardest part of solving the problem.

Now the same problem will be solved for workers. It will not be tested as thoroughly as the number of soldiers because analogous results are expected, but nevertheless some experiments will be done because of how much more difficult is to obtain rewards by using workers, as they have either to scout (which grants strictly diminishing rewards) or mine to produce soldiers that will obtain the rewards. Something that must be noted is that a similar case to the restriction that rewards-threshold imposed to the planner (in the sense that if it was greater than the smallest reward it made the planner search the state-space in a breadth fashion) happens with the amount of mineral that a workers gathers with MINE: if that amount is less than the cost of producing a soldier, instead of chaining several calls to MINE with different workers and building a soldier to get rewards, it seems to evaluate that plan as a bad one and explorers the other possibilities which leads to searching over a huge space. Setting the amount of minerals gathered to the cost of a soldier is a rather unrealistic approach but this problem can be overcome using abstraction (using squads of workers instead of individual ones) and is probably the only option to avoid stagnation. Apart from that another factor that complicates the resolution of the problem is the accumulated cost of the barracks: with each additional soldier produced at a barracks, its accumulated costs increases so using soldiers produced from the same barracks is not as useful as producing them in parallel. This is a constraint on the problem and forces the planner to look further in the search space in order to improve the quality of the plan. Since in this experiment the main goal is to isolate the impact of the number of workers in the problem, the additional cost of producing a soldier will be set to 0. For further experimentation it can be interesting to see how the complexity of the problem evolves as this variable changes, but for now it will not be taken into account. The next chart shows the series of tests done for the aforementioned purpose:

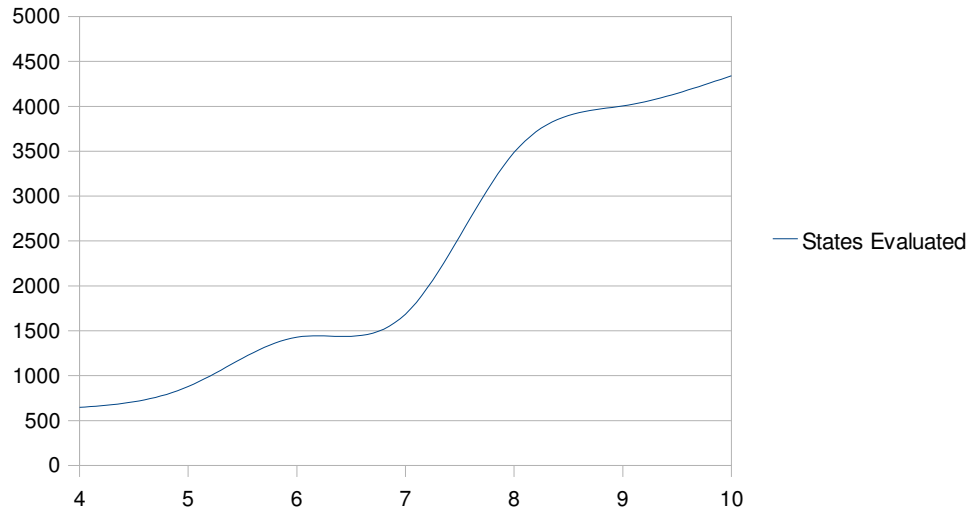

Figure 20: Number of workers impact

In this case the range of values for the number of workers is much shorter because of two reasons: first, there are many more influential factors that scale as the difficulty of the problem rises and measurements may be not as accurate as with the soldiers, and second, even with relatively small numbers of workers the number of evaluated states is already quite big. Actually, when using 11 or more workers the problem is so hard that the planner cannot solve it without using up too much memory. This is expected as getting rewards in this case is using miniplans (consisting on mining, producing a soldier and either attacking, harassing or defending) for every available worker, but the values are too high for this design to be effective used in real time environments. However and as it happened with the minerals gathered per MINE operator, using squads to further abstract the problem may be a possible solution. The main problem is how to separate the managing of the SCOUT action, which is performed by a single worker and gets no benefit if done by a squad, and the MINE action, which works more or less the same performed either by squads or by single workers, but an accurate interpretation of the plan by the player may solve this issue.

Once the impact of the number of workers has been measured and to ensure the completeness of the analysis, the parameter that was left out in the previous experiment, the accumulated cost that barracks have when building a soldier, will be used as the variable for a new series of tests. In this case, the problem will be the same as before but using a fixed number of workers (seven workers because the size of the search space is still manageable by the planner and leaves margin for generating a diverse range of plans of

different qualities) and with three barracks instead of only one to see if the planner generate plans with concurrent actions. These are the results:

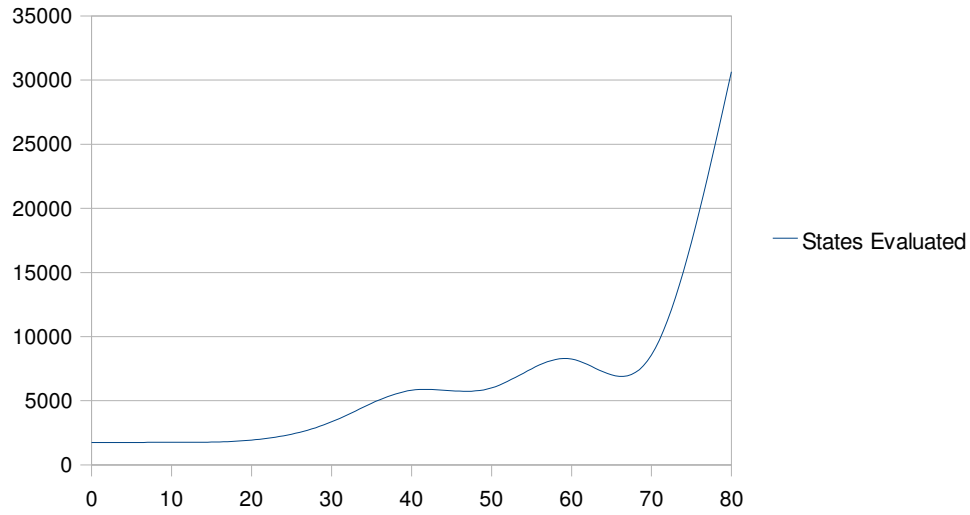

Figure 21: Accumulated-cost impact

Accumulated-cost is another constraint whose effect was expected to be detrimental to the performance of the planner, but it was not expected to have such an impact. In this case the progression of the size of search space is exponential even though the initial value of pre-total-cost was modified accordingly so it would not influence the results. What is worse, its effect on the plan is almost insignificant as in most of the cases the planner chooses the same barracks instead of preferring other with lesser costs, leading to no concurrency at all when producing new soldiers.

As the last experiment of this section, the impact of the initial number of minerals will be measured. Basically, apart from scouting workers gather mineral which is used to produce soldiers which in turn reap the rewards, so having an initial amount of mineral greater than zero can lead to the same plan without the MINE action. Since these partial plans are shorter than the process of gathering and producing, they will probably affect the size of the number of evaluated states less than the number of workers but still more than the number of soldiers. The parameters for this experiment will be the same as those for the workers, though obviously without workers and varying the value of the fluent mineral. When given enough mineral to produce 40 soldiers, the maximum number of soldiers that

was used in an earlier experiments dealing with soldiers, the planner evaluates 6947 states, more or less 7 times the number of states evaluated when using soldiers directly. Besides, the number of states scales linearly with the units of minerals, so the results are quite good having into account that the reward granting operators cannot be used unless a soldier is produced, which may had mislead the heuristic function.

6.2.3 Improvements of object reutilization

In section 4.3.6 a solution was proposed to overcome the limitations of PDDL 2.1 when working with sets of identical objects. The main issue was that since in PDDL 2.1 it is impossible to express a number of objects of a given type, individual objects must be declared in the problem. These objects are treated by planners as different possible arguments for the operators and lead to evaluating identical states in practice as different states in the state-space, increasing the complexity of the search. Therefore, a predicate was introduced as a prerequisite for operators that created units to force the planner to choose the arguments only from one object and not from all the possible options. To test the effectiveness of this technique, the problem to solve will be analogous to the problem solved when measuring the impact of the number of initial minerals in the previous subsection, but here several objects of the type soldier will be added in the initial state along with the ready predicate to replicate a common situation in which to allow building several soldiers additional objects of that type are added without reusing them. The normal problem for 500 minerals, which allows the creation of 10 soldiers, was solved evaluating 278 states; however, if there are 10 objects of the type soldier with the ready predicate, the resolution implies evaluating a total of 4178 states, a much worse situation. Not only that, when doing so and due to the increased complexity of the problem, the planner comes up with a worse plan than in the case in which object reutilization was used. This confirms that using object reutilization is a good solution to some of the limitations of PDDL 2.1 to the point that a medium difficulty problem for a heuristic planner becomes a trivial one.

6.2.4 Parameters of the planner

For the experimentation Metric-FF was used as heuristic planner. As described in section 5.1, Metric-FF can be executed with different options, two of which are relevant for the planning domain. First, the modifiers of the *g(x)* and the *h(x)* functions: This is a feature common to most of the heuristic planners and determines the balance between exploration and exploitation by assigning different weights to the cost from the initial state to the current state and the cost of the heuristic function respectively. This is of vital importance for the proposed planning domain as many of the problems arose when the planner gave special preference to exploration over exploitation due to how rewards were granted. Consequently, several experiments will take place using these values. The proposed problem will be similar to one of the problems used in the previous experiments trying to include all the relevant factors of the design: it has 10 soldiers, 5 workers, 100 minerals, 3 allied bases attacked by 4, 3 and 0 enemy soldiers and 2 enemy bases defended by 3 and 6 enemy soldiers. As a side note it is worth mentioning that the modification of these parameters is possible only when using best-first as the search algorithm and not when using enforced hill-climbing, which is the Metric-FF's algorithm by default. Since the final value of the function used to evaluate the utility of expanding a node is the addition of *g(x)* and *h(x)* (a completely linear function), the value that will be used will be the proportion between the weight of *g(x)* and the weight of *h(x)*, being the actual value the weight of *h(x)* divided by the weight of *g(x)* as the value of the first is usually much higher than the value of the latter. However when doing the experiments a curious fact is observed: the size of the search space and the quality of the plans do not scale as the proportion changes, but instead there are only two cases determined by a threshold which consists on the comparison of the two values: independently of the proportion, when the weight of *h(x)* is greater than the weight of *g(x)* a plan with a number of evaluated states is obtained and when it is the opposite case, another plan and number of evaluated states are obtained. Particularly, the regular case, a greater value for the weight of *h(x)*, yields better results both in plan quality and search space size, being the difference in many cases that the problem is only solvable with this configuration, as otherwise if exploration is favored the planner expands too many nodes and does not finish in a reasonable time. This is an expected result because of how badly the planner behaved when it tended to search in width depending on the values of other parameters, but why it is a threshold is something that is not clear. This is probably due to the heuristic function of Metric-FF, so further

experimentation with other planners would be advisable. Nevertheless, again this is out of the scope of this work, so it will be done only in future works.

The second parameter of the planner that can be changed is the search algorithm: Metric-FF implements enforced hill-climbing, an algorithm particular to this planner, but also supports best-first search, so an option is given to choose from the two of them. This parameter is not relevant for our work, as the design tries to be as independent from the planner as possible, but it may be insightful to explain the results of former experiments. In this case, the same problem that was used before will be solved using both methods. When done, best-first obtains a much better plan (it minimizes the cost to 45 compared to 270, both from an initial cost of 2300) but evaluates more states (1605 states compared to 971). Similar experiments give comparable results, though the differences are not great. Explaining this would mean deeply analyzing both algorithms, so we will not enter into detail, but nevertheless it is interesting to note that actually the result expected was this one, as enforced hill-climbing tends to outperform best-first in time at the expense of quality [Hoffmann, IPC 2002].

6.2.5 Practical utility of the plans

In the previous subsections, the main technical aspects of the domain have been analyzed in regard to adequateness, efficiency, etc... However, it is very important not to forget which was initially the main goal of the work: developing an AI system that could play in a such complicated game like a RTS game. Until now the quality of the plans has always been measured in function of the final cost (or the number of calls to the TO-END operator), but it is important to note that that cost comes from an equation specifically designed for the domain definition and that may not be directly translated into terms of usefulness. As judging how appropriate is a plan is a task that is impossible to accomplish automatically, plans should be evaluated by humans with a certain degree of experience in RTS games. Apart from that, there is no other way of telling if a plan is well suited for the problem or not. Therefore, in this section a plan for what could be a standard problem will be analyzed in order to check whether it could be a valid plan for a computer player to base its behavior on.

The problem to solve is roughly the same used in the previous subsection to test the difference between the search algorithms: 10 soldiers, 5 workers, 100 minerals, 3 allied bases attacked by 4, 2 and 0 enemy soldiers and 2 enemy bases defended by 3 and 7 enemy soldiers. The problem is actually the problem that appears in the Annex C as an example of input for the planner, but to better understand the problem the plan generated will be thoroughly explained. Here is the plan obtained when using best-first search:

```
step    0: DEFEND SOLDIER0 CC1 POS4-4 POS3-3
        1: DEFEND SOLDIER1 CC0 POS7-2 POS4-2
        2: SCOUT WORKER2 POS7-2 POS7-2
        3: DEFEND SOLDIER2 CC0 POS6-5 POS4-2
        4: ATTACK SOLDIER3 ENEMYBASE0 POS1-3 POS1-1
        5: ATTACK SOLDIER4 ENEMYBASE0 POS2-3 POS1-1
        6: DEFEND SOLDIER5 CC0 POS5-3 POS4-2
        7: ATTACK SOLDIER6 ENEMYBASE0 POS1-7 POS1-1
        8: ATTACK SOLDIER8 ENEMYBASE0 POS4-1 POS1-1
        9: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
       10: ATTACK SOLDIER0 ENEMYBASE0 POS4-2 POS1-1
       11: ATTACK SOLDIER7 ENEMYBASE0 POS3-2 POS1-1
       12: SCOUT WORKER1 POS5-4 POS6-3
       13: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
       14: ATTACK SOLDIER9 ENEMYBASE0 POS4-2 POS1-1
       15: MINE WORKER4 CC2 POS1-1 POS7-5
       16: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER9
       17: ATTACK SOLDIER9 ENEMYBASE0 POS4-2 POS1-1
       18: MINE WORKER3 CC2 POS4-6 POS7-5
       19: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER9
       20: DEFEND SOLDIER0 CC0 POS4-2 POS4-2
       21: MINE WORKER0 CC0 POS4-2 POS4-2
       22: BUILDSOLDIER BARRACKS0 POS4-2 SOLDIER0
       23: ATTACK SOLDIER0 ENEMYBASE0 POS4-2 POS1-1
       24: DEFEND SOLDIER9 CC0 POS4-2 POS4-2
```

```
25: TO-END
26: TO-END
27: END

    11.14 seconds searching, evaluating 13288 states, to a max depth of 0
    11.45 seconds total time

total cost:    0.00
```

Objects are used in the problem definition to represent concepts: CC0 is the allied base attacked by 2 enemy soldiers, CC1 is not attacked, CC2 is attacked by 4 enemy soldiers, ENEMYBASE0 is defended by 3 enemy soldiers and ENEMYBASE1 is defended by 7 enemy soldiers. In the plan, 5 soldiers defend CC0, another one defends CC1, 9 soldiers attack ENEMYBASE0, 3 workers scout, 3 gather mineral and 5 new soldiers are produced. The results are quite promising: the plan is attacking with the main force the least defended base while defending the two bases that are attacked by fewer soldiers and abandoning the base that is attacked by 4 soldiers. This is because of the way the number of bases determine the rewards: the more there are, the less rewarding is attacking or defending them. In this case, the enemy has two bases while the player has three, so if both sides lose one, the new situation is favorable to the player, expressed by dividing the rewards for attacking between two and the rewards for defending between three. The plan implies defending a base which is not attacked with a soldier, but this is due to the fact that the number of soldiers used in the problem definition is not the real number of enemy soldiers but the number of soldiers plus a number that ensures that the battle is won, so even if there are no soldiers attacking an allied base, in the problem definition there always be at least that number of additional enemy soldiers. Workers are also effectively used: the rewards for scouting decrease as individual workers are sent to explore undiscovered areas, so in the first half of the plan two workers are commanded to scout; however, as the plan advances the planner decides that it is more profitable to use them for mining than scouting so new soldiers can be produced and used for attacking and defending, which is actually a very coherent decision. Besides, this way of using the soldiers leads to what probably is the most important fact in the plan: the planner uses all the available resources (100 minerals from the initial state and 150 additional minerals from mining) to produce new soldiers, which means that it really plans ahead and tries to get the most out of the resources, be it units or minerals, it has at its disposal. This is exactly the behavior that was

expected to get when the design was done, so the plans generated using the appropriate parameters are arguably quite good.

Apart from the quality of the plan, the other factor that determines its usability is the time needed to compute it. In this case a total of 13288 states were evaluated, taking 11.45 seconds. This result is not as good as the quality as in 12 seconds the state of the game can change very much depending on how fast-paced the RTS game is. In particular, the pace of the game 3 in ORTS is not very fast, but more than 10 seconds may be on the verge of being unusable. It should be noted that this particular problem was parametrized looking for quality over performance (by using best first instead of enforced hill climbing, for example), so this result could be improved to make the problem more manageable at the cost of losing quality. Besides, the machine used to the experimentations was not exclusively used by the planner and it is not at all among the most potent computers that are found nowadays in the market, so having at the computer player's disposal a more capable machine would greatly reduce the time needed.

7. CONCLUSIONS

After describing all the design and the experimentation some comments are done herein. The first important thing that must be noted is the complexity of the problem. In sections 2.1.2 and 2.1.3 a notion of how difficult the problem was given by studying the research done in the field and the challenges RTS games pose to AI researchers. Indeed, the definition of the domain was far from being an easy task. First of all, the abstraction of the problem went through a number of phases in which an equilibrium between performance and representativity had to be kept, something that was very difficult due to the many different types of objects participating in the problem and the need of keeping the domain simple enough for problems to be solved quickly and accurately. There were some implementations that proved to be very useful: for instance, taking advantage of the fact that the effectiveness of gathering resources depended on the distance between control centers and minerals, minerals were taken away by assigning the operator MINE to control centers instead of minerals (which is the most natural approach) avoiding to represent the large number clusters of minerals that can be present in the domain. However, many of the problems came when, after deciding on a way of simplifying the domain, the experimentation proved that there were limitations either in usability or in complexity of the problem. A notable example can be the limitations of PDDL 2.1 in representing sets of equivalent objects, which is traditionally solved by adding individual objects instead of using some kind of numerical value with the drawback of increasing the possible combinations of operators and parameters when using those objects. In this case, after realizing that one of the causes of the low performance in the first specifications was this, object reutilization was implemented and tested along with other possible solutions, proving that it was a valid alternative to using sets of objects and therefore confirming its utility not only for this particular problem but for any other of similar characteristics, as this is a known and commonly encountered issue when using PDDL 2.1 Nevertheless, the challenges that the definition of the problem posed led to having many parameters which must be appropriately initialized for the problem to be solved respecting the time constraints that RTS games impose to computer players.

A significant fact that marked the process of design of the planning domain was the definition of goals. Classical problems usually have a well-defined goal which can be easily defined in the domain specification. However, RTS games are too complex for their winning condition to be used as a goal and so require using more cunning alternatives. In the approach of this work, continuous replanning was a staple of the design, so the goals were not long-term goals

which try to fulfill the winning condition but rather goals that could force the planner to come up with strategies involving several steps in time that could give an advantage to the player. Therefore, the problem was not a satisfiability problem but an optimization one, in which the utility of the plan should be maximized. The usefulness of the plan depended on how the units were used to complete certain tasks necessary in every RTS game, and to represent this a set of behaviors that lead to fulfilling a given role in the accomplishment of the plan was implemented. There are two key factors in this approach: first, the fact that the different tasks are often mutually exclusive as they need resources that are limited in number, and second, the way of establishing a degree of usefulness for employing units to perform the different tasks. To include in the implementation these two facts tasks were represented as soft goals that granted rewards when achieved. Here we had to deal with the particularities of automated planning, as plans are sequences of actions whose steps are evaluated and chosen as the nodes are expanded, something that opposes to the fact that the usefulness of achieving certain goals depended on the resources dedicated to them, both in number and in characteristics. Again, the solution was using relatively complex numerical functions in which rewards were dynamically assigned to units performing the tasks and costs were calculated and subtracted from the utility of the action depending on several characteristics of the unit being used, which ended up defining the value to maximize as rewards minus costs, or rather to minimize as an initial value minus the rewards plus the costs. This was a positive choice as high quality plans were obtained in the results but not only in terms of rewards, as they were coherent plans a human player could have used to try to win the game.

Another problem related to automated planning was how poorly state of the art planners manage non-monotonic metrics. As RTS games are complex environments, the values to optimize in the domain definition are completely dynamical, meaning in terms of planning that they are state dependent and change throughout the plan. This is a situation that modern planners cannot deal with, yet it is probably the most important factor when determining plan quality in the domain definition due to the system of dynamically assigned rewards and goals. Again and to overcome this the approach was converting the metric to calls to a dummy operator so the planner optimizes the plan length instead of a numerical value. This solution, though simple, worked very well in many cases and can be easily used in a domain definition or directly implemented in a planner, setting one of the few possible approaches to optimizing heterogeneously modified values.

All in all, a broad range of innovative techniques were used which succeeded to solve a problem of high complexity as a RTS game and which had never been faced using automated

planning. Besides, the objectives of keeping the solution independent from the planner and applicable to other problems were satisfied, which means that the conclusions of this work will be useful not only in gaming but also in many other environments of similar characteristics.

8. FUTURE WORK

Initially, this work originated from the idea of implementing a client to play in the third domain of the ORTS competition using automated planning. However, due to the multiple challenges the definition of the domain posed, the scope was changed to a more theoretical one dealing with all the issues that solving a complex problem like this one using automated planning had. The goals of this thesis have been achieved; hence, the next step would be taking all this work to a more practical plane and finally creating the client to participate in the competition implementing the architecture of the planner, the behaviors of the units, etc... so the real efficiency of the techniques used in the domain could be tested.

Apart from that, our intent of using standard languages and techniques opens two important paths for research: first, all the techniques used in the definition can be tested with any other planner that supports PDDL 2.1 or even implemented in the planner in order to increase their effectiveness and to avoid having to define them in the domain. This could have as consequence both using problems similar to the ones characteristic from RTS games as benchmarks or testbeds for state of the art planners and opening new trends in research intending to solve issues like the impossibility of optimizing non-monotonic values in planning problems; second, because the techniques are not problem dependent, they could be used as well in similar situations like military simulations or complex multi-agent environments, proving that their usefulness goes beyond the field of gaming.

BIBLIOGRAPHY

- [Aha, Molineaux and Ponsen, 05] Learning to win: Case-based plan selection in a real-time strategy game. Proceedings of the Sixth International Conference on Case-Based Reasoning (pp. 5-20). Chicago, IL: Springer
- [Blum and Furst, 1997] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. Artificial Intelligence, 90(1-2):281–300, 1997.
- [Buro, 02] M. Buro, ORTS: A Hack-Free RTS Game Environment, Proceedings of the International Computers and Games Conference 2002, Edmonton, Canada
- [Chan et al., 07] Extending Online Planning for Resource Production in Real-Time Strategy Games with Search, workshop “Planning in Games” - ICAPS 2007, Rhode Island, September 2007
- [Computer chess] Article in Wikipedia: http://en.wikipedia.org/wiki/Computer_chess
- [Ernst et al., 1969] G. Ernst, A. Newell, and H. Simon. GPS: A Case Study in Generality and Problem Solving. Academic Press, 1969.
- [Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2(2-3):189–208, 1971.
- [GPL] General Public License, http://www.gnu.org/licenses/gpl.html
- [Green, 1969] C. Green. Application of theorem-proving to problem solving. In D. E. Walker and L. M. Norton, editors, Proceedings of the 1st International Joint Conference on Artificial Intelligence, pages 219–239. William Kaufmann, 1969.
- [Hart et al., 1968] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum-cost paths. IEEE Transactions on System Sciences and Cybernetics, SSC-4(2):100–107, 1968.
- [Helemrt, 2002] Helmert, M. (2002). Decidability and undecidability results for planning with numerical state variables. In Proceedings of AIPS-02.
- [Hoffmann, 2002] J. Hoffmann, The Metric-FF Planning System: Translating ``Ignoring Delete Lists'' to Numeric State Variables, Journal of Artificial Intelligence Research, special issue on the 3rd International Planning Competition
- [IPC 2002] Results on the 2002 International Planning Competition, http://planning.cis.strath.ac.uk/competition/results.html
- [Kautz and Selman, 1992] H. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, Proceedings of the 10th European Conference on Artificial Intelligence, pages 359–363. John Wiley & Sons, 1992.

- [King, Atkin and Westbrook, 02] Tapir: the Evolution of an Agent Control Language, In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy
- [Korf, 1985] R. E. Korf. Depth-first iterative deepening: an optimal admissible tree search. Artificial Intelligence, 27(1):97–109, 1985.
- [McDermott, 98] McDermott, D. the AIPS-98 Planning Competition Committee 1998: PDDL-the planning domain definition language http://www.cs.yale.edu/homes/dvm
- [Metric-FF] Metric-FF's official web page http://members.deri.at/~joergh/metric-ff.html
- [Nau et al., 1999] Nau, D., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. In Proceedings of IJCAI'99.
- [Orkin, 2006] Three States and a Plan: The A.I. of F.E.A.R.. 4. Game Developers Conference 2006
- [ORTS Competition 2008] ORTS competition 2008 http://www.cs.ualberta.ca/~mburo/orts/AIIDE08/
- [ORTS snapshot] ORTS daily development snapshot http://www.cs.ualberta.ca/~mburo/orts/src_snapshot/snap.html
- [Pearl, 1984] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.
- [Pednault, 1989] Pednault, E. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. In Proceedings of KR-89, pp. 324–332.
- [Rosenschein, 1981] S. J. Rosenschein. Plan synthesis: A logical perspective. In P. J. Hayes, editor, Proceedings of the 7th International Joint Conference on Artificial Intelligence, pages 331–337. William Kaufmann, August 1981.
- [Sacerdoti, 1974] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. Artificial Intelligence, 5:115–135, 1974.
- [Sacerdoti, 1975] E. D. Sacerdoti. The nonlinear nature of plans. In Proceedings of the 4th International Joint Conference on Artificial Intelligence, pages 206–214, 1975.
- [Schaeffer et al. 317] Checkers Is Solved :Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, Steve Sutphen, Science 14 September 2007, Vol. 317. no. 5844, pp. 1518 – 1522, DOI: 10.1126/science.1144079
- [Starcraft competitions] Bloomberg News, last updated in January 15, 2006 http://www.bloomberg.com/apps/news?pid=email_us&refer=asia&sid=a2JvzciDnpB4
- [Stratagus] Stratagus web page http://www.stratagus.org

ANNEX A: Game 3 Blueprint

```
# $Id: game3.bp 2597 2007-05-25 23:54:20Z orts_furtak $

#=====================================================================

<DEFINE>

# fixed constants

TP              16   # TILE_POINTS; see GameConst.H
FPS              8   # simulation frames per second

# game constants

GAME_TIME    (20*60*FPS)  # in simulation frames

NUM_WORKERS      6   # initially

CURRENT_SUPPLY 300
MAX_SUPPLY     300  # currently not enforced
MINERALS       600  # at start

# Damage is calculated as follows: Damage = Rolled Attack Value - Target
Armor
BUILDING_ARMOR   2

# control center stats

CONTROL_COST           600
CONTROL_HP            1700
CONTROL_SIGHT            4 # in tiles
CONTROL_WIDTH     (4*TP-2) # in TP
CONTROL_HEIGHT    (4*TP-2) # in TP
CONTROL_BUILDTIME (38*FPS) # seconds

# barrack stats

BARRACKS_COST          400
BARRACKS_HP           1150
BARRACKS_SIGHT           4
BARRACKS_WIDTH     (4*TP-2)
BARRACKS_HEIGHT    (3*TP-2)
```

```
BARRACKS_BUILDTIME (25*FPS)

# factory stats

FACTORY_COST           400
FACTORY_HP            1400
FACTORY_SIGHT            4
FACTORY_WIDTH     (4*TP-2)
FACTORY_HEIGHT    (3*TP-2)
FACTORY_BUILDTIME (25*FPS) # seconds

# worker stats

WORKER_DMG1              4 # attack value uniformly
WORKER_DMG2              6 # distributed in [DMG1,DMG2]
WORKER_RANGE        (TP/4) # in TP
WORKER_COOL        (1*FPS) # seconds
WORKER_COST             50
WORKER_HP               60
WORKER_SIGHT             7
WORKER_RADIUS            3 # in TP
WORKER_SPEED             4 # TP/frame
WORKER_BUILDTIME   (7*FPS) # was 160
WORKER_SUPPLY            1
WORKER_BONUS             0 # not used

# marine stats
# marine hp lowered slightly
# damage set to constant for easy testing(may be randomized again later)
# cooldown lengthened to be more visibly obvious
# build time doubled as it seemed too fast before
# marine cost lowered

MARINE_DMG1             5
MARINE_DMG2             7
MARINE_ARMOR            0
MARINE_RANGE       (4*TP)
MARINE_COOL       (1*FPS)
MARINE_COST            50
MARINE_HP              40
MARINE_SIGHT            7
MARINE_RADIUS           4
MARINE_SPEED            3
MARINE_BUILDTIME  (8*FPS) # was 192
MARINE_SUPPLY           1
```

```
MARINE_BONUS            0 # not used

# tank stats
# tank hp lowered significantly for balance with respect to marines
# damage set to constant for easy testing(may be randomized again later)
# cooldown scaled with respect to marine cooldown
# build time scaled with respect to marine build time
# tank cost lowered to reflect changes in stats
# tank armor meant to balance marine cost efficiency

TANK_DMG1             26
TANK_DMG2             34
TANK_DMG3              0  # not used
TANK_DMG4              0  # not used
TANK_ARMOR             1
TANK_COOL   ((10*FPS)/4)  # 2.5 sec
TANK_COOL2             0  # not used
TANK_COST            200
TANK_HP              150
TANK_RANGE        (7*TP)
TANK_RANGE1       (2*TP)  # minimum when anchored
TANK_RANGE2       (7*TP)  # not used
TANK_SPLASH            5  # not used
TANK_SPLASH2          15  # not used
TANK_SIGHT            10
TANK_RADIUS            7
TANK_SPEED             3
TANK_BUILDTIME  (16*FPS) # was 400
TANK_SUPPLY            2
TANK_BONUS             0 # not used
TANK_SWITCH_TIME  (3*FPS)

</DEFINE>

#====================================================================

<INCLUDE>
"main.bp"
"common/scorekeeper.bp"
"common/common.bp"
"common/units.bp"
"common/player.bp"
"common/shepherd.bp"
</INCLUDE>
```

```
#=====================================================================

<BLUEPRINTS>

<INCLUDE>
"terrans/weapons.bp"
"terrans/armor.bp"
"terrans/tools.bp"
"terrans/units.bp"
</INCLUDE>

#=====================================================================

blueprint start_loc

  var x     0
  var y     0
  var owner 0

  # The constructor is called at object creation -- since the start_loc
  # has x,y coordinates defined in the world string, they won't be
  # set when the constructor is called.

  # Therefore, 'place' is called to initialize the start location at
  # time 0.

  constructor init(;;) {
    this.place() in 0;
  }

  action place(;;) {
    gob unit;
    gob building;
    int i;

    building = create("controlCenter", this.owner);
    building.x = this.x;
    building.y = this.y;
    # update the location of the object -- this takes care of updating
    # motion sectors so that collision testing is accurate when adding
    # the workers.
    position(building);

    for (i=0; i<NUM_WORKERS; i+=1) {
      unit = create("worker", this.owner);
```

```
      close_to(building, unit; TP/4);
      position(unit);
    }
  }

end

#=====================================================================

blueprint timer

  var timeout GAME_TIME # maximum number of simulation cycles
  var owner   0

  constructor init(;;) {
    this.check() in 1;
  }

  action check(;;) {
    int i,j;
    int score;
    int best_score, winner;

    if (this.owner) break;

    i = tick();
    if (i >= this.timeout) {
      shared.sk.print(); # print out game stats
      winner = -1;

      best_score = -1000000;

      j = shared.sk.units_trained.size() - 1; # number of players
      for (i=0; i<j; i+=1) {

        score = (shared.sk.mineral_count[i]/2) +
                (shared.sk.standing_cost[i])   +
                (shared.sk.destruct_cost[i]);

        if (score > best_score) {
          best_score = score;
          winner = i;
        }

        #------------------- TIE ------------------#
```

```
      #specific for this 2 player setting
      if (score == best_score) {
        winner = 99;
      }

#        if (shared.sk.buildings_lost[i] <= best_bnum) {
#          if (shared.sk.buildings_lost[i] < best_bnum) {
#            best_bnum = shared.sk.buildings_lost[i];
#            best_unum = shared.sk.units_lost[i];
#            best_time = shared.sk.time_destroyed[i];
#            winner = i;
#          } else {
#            if (shared.sk.units_lost[i] <= best_unum) {
#              if (shared.sk.units_lost[i] < best_unum) {
#                best_bnum = shared.sk.buildings_lost[i];
#                best_unum = shared.sk.units_lost[i];
#                best_time = shared.sk.time_destroyed[i];
#                winner = i;
#              } else {
#                if (shared.sk.time_destroyed[i] < best_time) {
#                  best_bnum = shared.sk.buildings_lost[i];
#                  best_unum = shared.sk.units_lost[i];
#                  best_time = shared.sk.time_destroyed[i];
#                  winner = i;
#                }
#              }
#            }
#          }
#        }
      }

      cout("\n@ TIME %d ", tick());
      shared.sk.print(); # print out game stats
      cout("\n");

#      for (i=0; i<j; i+=1) {
#        cout("p= %d : min= %d sta= %d des= %d tot= %d | ",
#              i,
#              shared.sk.mineral_count[i],
#              shared.sk.standing_cost[i],
#              shared.sk.destruct_cost[i],
#              (shared.sk.mineral_count[i]/2) +
#              (shared.sk.standing_cost[i]) +
#              (shared.sk.destruct_cost[i]));
#      }
```

```
#      cout("\n");

      end_game(winner);
    }

    # early victory condition
    # specific to a 1v1 game

    for(i=0; i<2; i+=1){
      if (shared.sk.buildings_alive[i] <= 0 && shared.sk.buildings_lost[i]
> 0) {

          winner = (i+1) % 2;

          cout("\n@ EARLY %d ", tick());
          shared.sk.print(); # print out game stats
          cout("\n");

          end_game(winner);
      }
    }
    this.check() in 1;
  }

end

#============================================================================

# global objects

global timer score_timer;
global shepherd_bp shepherd;

#============================================================================

</BLUEPRINTS>

#============================================================================
```

ANNEX B: Domain Definition

```
(define (domain game3)

(:requirements :adl)

(:types     position
            object
            ally - object
            cc - ally
            barracks - ally
            unit - ally
            worker - unit
            soldier - unit
            enemybase - object
)

(:predicates
            (at ?object - object ?pos - position)
            (goal_achieved)
            (unexplored ?pos - position)
            (ready ?ally - ally)
)

(:functions
            (positionCost ?pos - position)
            (posX ?pos - position)
            (posY ?pos - position)
            (accumulated-cost ?ally - ally)
            (enemy-soldiers ?base)
            (soldiers-assigned ?base)
            (max-workers-to-mine ?cc - cc)
            (pre-total-cost)
            (total-cost)
            (minerals)
            (minerals-gathered)
            (rewards)
            (attack-reward ?enemybase - object)
            (attack-reward-increase ?enemybase - object)
            (attack-reward-decrease ?enemybase - object)
            (defend-reward ?cc - cc)
            (defend-reward-increase ?cc - cc)
            (defend-reward-decrease ?cc - cc)
            (rewards-threshold)
            (cost-increment)
            (scout-reward)
            (scout-reward-decrease)
            (harass-reward ?enemybase - object)
            (harass-reward-decrease)
            (worker-build-cost)
            (soldier-build-cost)
```

```
)

(:action TO-END
            :precondition  (and
                              (> (rewards) (rewards-threshold))
                              (> (pre-total-cost) 0)
            )
            :effect     (and
                              (decrease (pre-total-cost) (cost-increment))
                              (increase (total-cost) (cost-increment))
            )
)

(:action END
            :precondition  (and
                              (> (rewards) (rewards-threshold))
                              (<= (pre-total-cost) 0)
            )
            :effect     (goal_achieved)
)

(:action ATTACK
            :parameters (?soldier - soldier ?enemybase - enemybase ?src -
position ?dest - position)
            :precondition  (and
                        (at ?enemybase ?dest)
                        (at ?soldier ?src)
            )
            :effect     (and

                        (increase   (pre-total-cost)   (accumulated-cost   ?
soldier))
                        (when (< (posX ?src) (posX ?dest)) (and (increase
(pre-total-cost) (- (posX ?dest) (posX ?src)))(increase (accumulated-cost ?
soldier) (- (posX ?dest) (posX ?src)))))
                        (when (> (posX ?src) (posX ?dest)) (and (increase
(pre-total-cost) (- (posX ?src) (posX ?dest)))(increase (accumulated-cost ?
soldier) (- (posX ?src) (posX ?dest)))))
                        (when (< (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?dest) (posY ?src)))(increase (accumulated-cost ?
soldier) (- (posY ?dest) (posY ?src)))))
                        (when (> (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?src) (posY ?dest)))(increase (accumulated-cost ?
soldier) (- (posY ?src) (posY ?dest)))))
                        (when (< (positionCost ?src) (positionCost ?dest))
(and (increase (pre-total-cost) (- (positionCost ?dest) (positionCost ?
src))) (increase (accumulated-cost ?soldier) (- (positionCost ?dest)
(positionCost ?src)))))

                        (not (at ?soldier ?src))

                        (when (not(exists (?x - soldier)(ready ?x))) (ready
?soldier))
```

```
                        (when  (>  (enemy-soldiers  ?enemybase)  (soldiers-
assigned ?enemybase))
                              (and  (increase  (rewards)  (attack-reward  ?
enemybase) )
                                    (decrease   (pre-total-cost)   (attack-
reward ?enemybase) )
                                    (increase   (attack-reward   ?enemybase)
(attack-reward-increase ?enemybase))
                              )
                        )

                        (when  (<=  (enemy-soldiers  ?enemybase)  (soldiers-
assigned ?enemybase))
                              (and  (increase  (rewards)  (attack-reward  ?
enemybase) )
                                    (decrease   (pre-total-cost)   (attack-
reward ?enemybase) )
                                    (decrease   (attack-reward   ?enemybase)
(attack-reward-decrease ?enemybase))
                              )
                        )

                        (decrease   (harass-reward   ?enemybase)   (harass-
reward-decrease))
                        (increase (soldiers-assigned ?enemybase) 1)

            )
)

(:action DEFEND
            :parameters (?soldier - soldier ?cc - cc ?src - position ?dest
- position)
            :precondition  (and
                        (at ?cc ?dest)
                        (at ?soldier ?src)
                        (> (enemy-soldiers ?cc) 0)
            )
            :effect     (and

                        (increase   (pre-total-cost)   (accumulated-cost   ?
soldier))
                        (when (< (posX ?src) (posX ?dest)) (and (increase
(pre-total-cost) (- (posX ?dest) (posX ?src)))(increase (accumulated-cost ?
soldier) (- (posX ?dest) (posX ?src)))))
                        (when (> (posX ?src) (posX ?dest)) (and (increase
(pre-total-cost) (- (posX ?src) (posX ?dest)))(increase (accumulated-cost ?
soldier) (- (posX ?src) (posX ?dest)))))
                        (when (< (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?dest) (posY ?src)))(increase (accumulated-cost ?
soldier) (- (posY ?dest) (posY ?src)))))
                        (when (> (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?src) (posY ?dest)))(increase (accumulated-cost ?
soldier) (- (posY ?src) (posY ?dest)))))
                        (when (< (positionCost ?src) (positionCost ?dest))
```

```
(and (increase (pre-total-cost) (- (positionCost ?dest) (positionCost ?
src))) (increase (accumulated-cost ?soldier) (- (positionCost ?dest) 
(positionCost ?src)))))

                        (not (at ?soldier ?src))

                        (when (not(exists (?x - soldier)(ready ?x))) (ready 
?soldier))


                        (when (> (enemy-soldiers ?cc) (soldiers-assigned ?
cc))
                              (and (increase (rewards) (defend-reward ?
cc) )
                                    (decrease (pre-total-cost) (defend-
reward ?cc) )
                                    (increase (defend-reward ?cc) (defend-
reward-increase ?cc))
                              )
                        )

                        (when (<= (enemy-soldiers ?cc) (soldiers-assigned ?
cc))
                              (and (increase (rewards) (defend-reward ?
cc) )
                                    (decrease (pre-total-cost) (defend-
reward ?cc) )
                                    (decrease (defend-reward ?cc) (defend-
reward-decrease ?cc))
                              )
                        )

                        (increase (soldiers-assigned ?cc) 1)

            )
)


(:action HARASS
            :parameters (?soldier - soldier ?enemybase - enemybase ?src - 
position ?dest - position)
            :precondition  (and
                        (at ?enemybase ?dest)
                        (at ?soldier ?src)
            )
            :effect     (and

                        (increase (pre-total-cost) (accumulated-cost ?
soldier))
                        (when (< (posX ?src) (posX ?dest)) (and (increase 
(pre-total-cost) (- (posX ?dest) (posX ?src)))(increase (accumulated-cost ?
soldier) (- (posX ?dest) (posX ?src)))))
                        (when (> (posX ?src) (posX ?dest)) (and (increase 
(pre-total-cost) (- (posX ?src) (posX ?dest)))(increase (accumulated-cost ?
```

```
soldier) (- (posX ?src) (posX ?dest)))))
                        (when (< (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?dest) (posY ?src)))(increase (accumulated-cost ?
soldier) (- (posY ?dest) (posY ?src)))))
                        (when (> (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?src) (posY ?dest)))(increase (accumulated-cost ?
soldier) (- (posY ?src) (posY ?dest)))))
                        (when (< (positionCost ?src) (positionCost ?dest))
(and (increase (pre-total-cost) (- (positionCost ?dest) (positionCost ?
src))) (increase (accumulated-cost ?soldier) (- (positionCost ?dest)
(positionCost ?src)))))

                        (not (at ?soldier ?src))

                        (when (not(exists (?x - soldier)(ready ?x))) (ready
?soldier))

                        (increase (rewards) (harass-reward ?enemybase))
                        (decrease    (pre-total-cost)    (harass-reward    ?
enemybase))
                        (decrease   (harass-reward   ?enemybase)   (harass-
reward-decrease))

            )
)

(:action SCOUT
            :parameters (?worker - worker ?src - position ?dest - position)
            :precondition  (and
                        (unexplored ?dest)
                        (at ?worker ?src)
            )
            :effect     (and

                        (not (unexplored ?dest))

                        (increase   (pre-total-cost)   (accumulated-cost   ?
worker))
                        (when (< (posX ?src) (posX ?dest)) (and (increase
(pre-total-cost) (- (posX ?dest) (posX ?src)))(increase (accumulated-cost ?
worker) (- (posX ?dest) (posX ?src)))))
                        (when (> (posX ?src) (posX ?dest)) (and (increase
(pre-total-cost) (- (posX ?src) (posX ?dest)))(increase (accumulated-cost ?
worker) (- (posX ?src) (posX ?dest)))))
                        (when (< (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?dest) (posY ?src)))(increase (accumulated-cost ?
worker) (- (posY ?dest) (posY ?src)))))
                        (when (> (posY ?src) (posY ?dest)) (and (increase
(pre-total-cost) (- (posY ?src) (posY ?dest)))(increase (accumulated-cost ?
worker) (- (posY ?src) (posY ?dest)))))
                        (when (< (positionCost ?src) (positionCost ?dest))
(and (increase (pre-total-cost) (- (positionCost ?dest) (positionCost ?
src))) (increase (accumulated-cost ?worker) (- (positionCost ?dest)
(positionCost ?src)))))
```

```
                        (not (at ?worker ?src))

                        (when   (not(exists   (?x   -   worker)(ready   ?x)))
(ready ?worker))

                        (increase (rewards) (scout-reward))
                        (decrease (pre-total-cost) (scout-reward))
                        (decrease (scout-reward) (scout-reward-decrease))

            )
)

(:action MINE
            :parameters (?worker - worker ?cc - cc ?src - position ?dest -
position)
            :precondition  (and
                        (at ?worker ?src)
                        (at ?cc ?dest)
                        (> (max-workers-to-mine ?cc) 0)
            )
            :effect     (and
                        (not (at ?worker ?src))

                        (when   (not(exists   (?x   -   worker)(ready   ?x)))
(ready ?worker))

                        (decrease (max-workers-to-mine ?cc) 1)
                        (increase    (minerals)    (-    (minerals-gathered)
(accumulated-cost ?worker)) )
            )
)

(:action BUILDWORKER
            :parameters (?cc - cc ?position - position ?worker - worker)
            :precondition  (and
                        (ready ?worker)
                        (>= (minerals) 50)
                        (at ?cc ?position)
            )
            :effect     (and
                        (increase   (accumulated-cost   ?cc)   (worker-build-
cost))
                        (assign   (accumulated-cost   ?worker)   (accumulated-
cost ?cc))
                        (at ?worker ?position)
                        (decrease (minerals) 50)
                        (not (ready ?worker))
            )
)

(:action BUILDSOLDIER
            :parameters (?barracks - barracks ?position - position ?soldier
- soldier)
```

```
            :precondition  (and
                        (at ?barracks ?position)
                        (>= (minerals) 50)
                        (ready ?soldier)
            )
            :effect     (and
                        (increase  (accumulated-cost  ?barracks)  (soldier-
build-cost))
                        (assign  (accumulated-cost  ?soldier)  (accumulated-
cost ?barracks))
                        (at ?soldier ?position)
                        (decrease (minerals) 50)
                        (not (ready ?soldier))
            )
)

(:action BUILDBARRACKS
            :parameters (?barracks - barracks ?position - position ?worker
- worker)
            :precondition  (and
                        (>= (minerals) 400)
                        (at ?worker ?position)
                        (ready ?barracks)
            )
            :effect     (and
                        (at ?barracks ?position)
                        (decrease (minerals) 400)
                        (not (ready ?barracks))
            )
)

)
```

ANNEX C: Problem Definition

```
(define (problem game3-problem)

(:domain game3)

(:objects
pos0-0 - position
pos0-1 - position
pos0-2 - position
pos0-3 - position
pos0-4 - position
pos0-5 - position
pos0-6 - position
pos0-7 - position
pos1-0 - position
pos1-1 - position
pos1-2 - position
pos1-3 - position
pos1-4 - position
pos1-5 - position
pos1-6 - position
pos1-7 - position
pos2-0 - position
pos2-1 - position
pos2-2 - position
pos2-3 - position
pos2-4 - position
pos2-5 - position
pos2-6 - position
pos2-7 - position
pos3-0 - position
pos3-1 - position
pos3-2 - position
pos3-3 - position
pos3-4 - position
pos3-5 - position
pos3-6 - position
pos3-7 - position
pos4-0 - position
pos4-1 - position
pos4-2 - position
pos4-3 - position
pos4-4 - position
pos4-5 - position
pos4-6 - position
pos4-7 - position
pos5-0 - position
pos5-1 - position
pos5-2 - position
pos5-3 - position
```

```
pos5-4 - position
pos5-5 - position
pos5-6 - position
pos5-7 - position
pos6-0 - position
pos6-1 - position
pos6-2 - position
pos6-3 - position
pos6-4 - position
pos6-5 - position
pos6-6 - position
pos6-7 - position
pos7-0 - position
pos7-1 - position
pos7-2 - position
pos7-3 - position
pos7-4 - position
pos7-5 - position
pos7-6 - position
pos7-7 - position
enemybase0 - enemybase
enemybase1 - enemybase
soldier0 - soldier
soldier1 - soldier
soldier2 - soldier
soldier3 - soldier
soldier4 - soldier
soldier5 - soldier
soldier6 - soldier
soldier7 - soldier
soldier8 - soldier
soldier9 - soldier
worker0 - worker
worker1 - worker
worker2 - worker
worker3 - worker
worker4 - worker
barracks0 - barracks
barracks1 - barracks
barracks2 - barracks
cc0 - cc
cc1 - cc
cc2 - cc
) (:init
(= (posX pos0-0) 0)
(= (posY pos0-0) 0)
(= (positionCost pos0-0) 32)
(= (posX pos0-1) 0)
(= (posY pos0-1) 1)
(= (positionCost pos0-1) 30)
(= (posX pos0-2) 0)
(= (posY pos0-2) 2)
(= (positionCost pos0-2) 386)
(= (posX pos0-3) 0)
```

```
(= (posY pos0-3) 3)
(= (positionCost pos0-3) 736)
(= (posX pos0-4) 0)
(= (posY pos0-4) 4)
(= (positionCost pos0-4) 160)
(= (posX pos0-5) 0)
(= (posY pos0-5) 5)
(= (positionCost pos0-5) 124)
(= (posX pos0-6) 0)
(= (posY pos0-6) 6)
(= (positionCost pos0-6) 124)
(= (posX pos0-7) 0)
(= (posY pos0-7) 7)
(= (positionCost pos0-7) 287)
(= (posX pos1-0) 1)
(= (posY pos1-0) 0)
(= (positionCost pos1-0) 32)
(= (posX pos1-1) 1)
(= (posY pos1-1) 1)
(= (positionCost pos1-1) 30)
(= (posX pos1-2) 1)
(= (posY pos1-2) 2)
(= (positionCost pos1-2) 136)
(= (posX pos1-3) 1)
(= (posY pos1-3) 3)
(= (positionCost pos1-3) 148)
(= (posX pos1-4) 1)
(= (posY pos1-4) 4)
(= (positionCost pos1-4) 119)
(= (posX pos1-5) 1)
(= (posY pos1-5) 5)
(= (positionCost pos1-5) 282)
(= (posX pos1-6) 1)
(= (posY pos1-6) 6)
(= (positionCost pos1-6) 412)
(= (posX pos1-7) 1)
(= (posY pos1-7) 7)
(= (positionCost pos1-7) 412)
(= (posX pos2-0) 2)
(= (posY pos2-0) 0)
(= (positionCost pos2-0) 25)
(= (posX pos2-1) 2)
(= (posY pos2-1) 1)
(= (positionCost pos2-1) 27)
(= (posX pos2-2) 2)
(= (posY pos2-2) 2)
(= (positionCost pos2-2) 24)
(= (posX pos2-3) 2)
(= (posY pos2-3) 3)
(= (positionCost pos2-3) 61)
(= (posX pos2-4) 2)
(= (posY pos2-4) 4)
(= (positionCost pos2-4) 132)
(= (posX pos2-5) 2)
```

```
(= (posY pos2-5) 5)
(= (positionCost pos2-5) 586)
(= (posX pos2-6) 2)
(= (posY pos2-6) 6)
(= (positionCost pos2-6) 711)
(= (posX pos2-7) 2)
(= (posY pos2-7) 7)
(= (positionCost pos2-7) 611)
(= (posX pos3-0) 3)
(= (posY pos3-0) 0)
(= (positionCost pos3-0) 20)
(= (posX pos3-1) 3)
(= (posY pos3-1) 1)
(= (positionCost pos3-1) 15)
(= (posX pos3-2) 3)
(= (posY pos3-2) 2)
(= (positionCost pos3-2) 12)
(= (posX pos3-3) 3)
(= (posY pos3-3) 3)
(= (positionCost pos3-3) 49)
(= (posX pos3-4) 3)
(= (posY pos3-4) 4)
(= (positionCost pos3-4) 86)
(= (posX pos3-5) 3)
(= (posY pos3-5) 5)
(= (positionCost pos3-5) 411)
(= (posX pos3-6) 3)
(= (posY pos3-6) 6)
(= (positionCost pos3-6) 611)
(= (posX pos3-7) 3)
(= (posY pos3-7) 7)
(= (positionCost pos3-7) 611)
(= (posX pos4-0) 4)
(= (posY pos4-0) 0)
(= (positionCost pos4-0) 45)
(= (posX pos4-1) 4)
(= (posY pos4-1) 1)
(= (positionCost pos4-1) 25)
(= (posX pos4-2) 4)
(= (posY pos4-2) 2)
(= (positionCost pos4-2) 12)
(= (posX pos4-3) 4)
(= (posY pos4-3) 3)
(= (positionCost pos4-3) 49)
(= (posX pos4-4) 4)
(= (posY pos4-4) 4)
(= (positionCost pos4-4) 91)
(= (posX pos4-5) 4)
(= (posY pos4-5) 5)
(= (positionCost pos4-5) 411)
(= (posX pos4-6) 4)
(= (posY pos4-6) 6)
(= (positionCost pos4-6) 611)
(= (posX pos4-7) 4)
```

```
(= (posY pos4-7) 7)
(= (positionCost pos4-7) 611)
(= (posX pos5-0) 5)
(= (posY pos5-0) 0)
(= (positionCost pos5-0) 200)
(= (posX pos5-1) 5)
(= (posY pos5-1) 1)
(= (positionCost pos5-1) 25)
(= (posX pos5-2) 5)
(= (posY pos5-2) 2)
(= (positionCost pos5-2) 12)
(= (posX pos5-3) 5)
(= (posY pos5-3) 3)
(= (positionCost pos5-3) 17)
(= (posX pos5-4) 5)
(= (posY pos5-4) 4)
(= (positionCost pos5-4) 54)
(= (posX pos5-5) 5)
(= (posY pos5-5) 5)
(= (positionCost pos5-5) 59)
(= (posX pos5-6) 5)
(= (posY pos5-6) 6)
(= (positionCost pos5-6) 116)
(= (posX pos5-7) 5)
(= (posY pos5-7) 7)
(= (positionCost pos5-7) 191)
(= (posX pos6-0) 6)
(= (posY pos6-0) 0)
(= (positionCost pos6-0) 200)
(= (posX pos6-1) 6)
(= (posY pos6-1) 1)
(= (positionCost pos6-1) 25)
(= (posX pos6-2) 6)
(= (posY pos6-2) 2)
(= (positionCost pos6-2) 12)
(= (posX pos6-3) 6)
(= (posY pos6-3) 3)
(= (positionCost pos6-3) 25)
(= (posX pos6-4) 6)
(= (posY pos6-4) 4)
(= (positionCost pos6-4) 76)
(= (posX pos6-5) 6)
(= (posY pos6-5) 5)
(= (positionCost pos6-5) 10)
(= (posX pos6-6) 6)
(= (posY pos6-6) 6)
(= (positionCost pos6-6) 85)
(= (posX pos6-7) 6)
(= (posY pos6-7) 7)
(= (positionCost pos6-7) 520)
(= (posX pos7-0) 7)
(= (posY pos7-0) 0)
(= (positionCost pos7-0) 35)
(= (posX pos7-1) 7)
```

```
(= (posY pos7-1) 1)
(= (positionCost pos7-1) 75)
(= (posX pos7-2) 7)
(= (posY pos7-2) 2)
(= (positionCost pos7-2) 200)
(= (posX pos7-3) 7)
(= (posY pos7-3) 3)
(= (positionCost pos7-3) 345)
(= (posX pos7-4) 7)
(= (posY pos7-4) 4)
(= (positionCost pos7-4) 178)
(= (posX pos7-5) 7)
(= (posY pos7-5) 5)
(= (positionCost pos7-5) 214)
(= (posX pos7-6) 7)
(= (posY pos7-6) 6)
(= (positionCost pos7-6) 54)
(= (posX pos7-7) 7)
(= (posY pos7-7) 7)
(= (positionCost pos7-7) 56)

(= (pre-total-cost) 2700)
(= (total-cost) 0)
(= (minerals) 100)
(= (minerals-gathered) 50)
(= (rewards) 0)
(= (scout-reward) 90)
(= (scout-reward-decrease) 30)
(= (harass-reward-decrease) 25)
(= (cost-increment) 15)
(= (rewards-threshold) 0)
(= (worker-build-cost) 10)
(= (soldier-build-cost) 10)

(= (defend-reward cc0) 100)
(= (defend-reward-increase cc0) 20)
(= (defend-reward-decrease cc0) 40)
(= (enemy-soldiers cc0) 5)
(= (soldiers-assigned cc0) 0)
(= (max-workers-to-mine cc0) 50)

(= (defend-reward cc1) 100)
(= (defend-reward-increase cc1) 100)
(= (defend-reward-decrease cc1) 200)
(= (enemy-soldiers cc1) 1)
(= (soldiers-assigned cc1) 0)
(= (max-workers-to-mine cc1) 5)

(= (defend-reward cc2) 100)
(= (defend-reward-increase cc2) 15)
(= (defend-reward-decrease cc2) 30)
(= (enemy-soldiers cc2) 7)
(= (soldiers-assigned cc2) 0)
(= (max-workers-to-mine cc2) 5)
```

```
(= (attack-reward enemybase0) 150)
(= (harass-reward enemybase0) 200)
(= (attack-reward-increase enemybase0) 25)
(= (attack-reward-decrease enemybase0) 50)
(= (enemy-soldiers enemybase0) 6)
(= (soldiers-assigned enemybase0) 0)

(= (attack-reward enemybase1) 150)
(= (harass-reward enemybase1) 200)
(= (attack-reward-increase enemybase1) 15)
(= (attack-reward-decrease enemybase1) 30)
(= (enemy-soldiers enemybase1) 10)
(= (soldiers-assigned enemybase1) 0)

(at enemybase0 pos1-1)
(at enemybase1 pos6-0)
(at cc0 pos4-2)
(at cc1 pos3-3)
(at cc2 pos7-5)
(at barracks0 pos4-2)
(at worker0 pos4-2)
(at worker1 pos5-4)
(at worker2 pos7-2)
(at worker3 pos4-6)
(at worker4 pos1-1)
(at soldier0 pos4-4)
(at soldier1 pos7-2)
(at soldier2 pos6-5)
(at soldier3 pos1-3)
(at soldier4 pos2-3)
(at soldier5 pos5-3)
(at soldier6 pos1-7)
(at soldier7 pos3-2)
(at soldier8 pos4-1)
(at soldier9 pos4-2)

(= (accumulated-cost worker0) 0)
(= (accumulated-cost worker1) 0)
(= (accumulated-cost worker2) 0)
(= (accumulated-cost worker3) 0)
(= (accumulated-cost worker4) 0)
(= (accumulated-cost soldier0) 0)
(= (accumulated-cost soldier1) 0)
(= (accumulated-cost soldier2) 0)
(= (accumulated-cost soldier3) 0)
(= (accumulated-cost soldier4) 0)
(= (accumulated-cost soldier5) 0)
(= (accumulated-cost soldier6) 0)
(= (accumulated-cost soldier7) 0)
(= (accumulated-cost soldier8) 0)
(= (accumulated-cost soldier9) 0)
(= (accumulated-cost barracks0) 10)
(= (accumulated-cost cc0) 10)
```

```
(= (accumulated-cost cc1) 10)
(= (accumulated-cost cc2) 10)

(unexplored pos6-2)
(unexplored pos1-2)
(unexplored pos4-3)
(unexplored pos6-3)
(unexplored pos7-2)
(unexplored pos3-2)
(unexplored pos2-7)



) (:goal (goal_achieved) )
(:metric minimize (total-cost) ) )
```