ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

SOLVING BUILDING PLACEMENT PROBLEM IN A REAL-TIME STRATEGY VIDEO GAME

Graduation Project

Volkan Ilbeyli
040100118

Department : Computer Engineering
Division : Computer Engineering

Advisor : Asst. Prof. Dr. Sanem SARIEL

July 2014

ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

SOLVING BUILDING PLACEMENT PROBLEM IN A REAL-TIME STRATEGY VIDEO GAME

Graduation Project

Volkan Ilbeyli
040100118

Department : Computer Engineering
Division : Computer Engineering

Advisor : Asst. Prof. Dr. Sanem SARIEL

July 2014

Declaration of Authenticity

I would personally like to acknowledge you that;

1. I have specified all the references of the quotations by other sources that are used in my thesis, by stating the original sources,
2. Except of the quotations from other sources; all of the theoretic studies, software and hardware that determines the main topic of the project and are used in the thesis are done by me.

Istanbul, 2014

Volkan İlbeyli

Acknowledgements

First and foremost, I would like to thank Asst. Prof. Dr. Sanem Sarıel for accepting me and helping me complete this project and to my dear family who always supported me helped me become a successful individual.

I would like to thank Fevzi ‘Arcane’ Altuncu and Ali Sinanoğlu for encouraging me to continue and finish this project as StarCraft is the three of ours’ favorite game of all times.

Finally, special thanks to Gökhan Çoban and Mert Salık for all the projects we have done together, their criticisms and for all the ideas we shared together, as they have helped me to become this person who is always aiming for the best and chasing his childhood dream as of now.

SOLVING BUILDING PLACEMENT PROBLEM IN A REAL-TIME STRATEGY GAME
(SUMMARY)

StarCraft [4] is a real-time strategy video game, created by Blizzard Entertainment in 1998 and is the recent platform for real-time artificial intelligence. With the development of BWAPI, a hack tool used to access in game data, StarCraft player- AI researchers began to research in the area, starting in 2009. Recent publications in the area such as *A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft, Fast Heuristic Search for RTS Game Combat Scenarios,* and more, proved the tool to be quite useful. Among the many other sub-problems, blocking the entrance of a base, namely walling in, is another sub-problem that is covered in this study.

A simple game playing agent is implemented that simply follows a certain build order (the order by which the agent produces units or structures) and executes the logic program that is obtained and modified from Michal Certicky's *Implementing a Wall-In Building Placement in StarCraft with Declarative Programming*. The logic program takes the data that is gathered by the game playing agent by the help of BWAPI's [5] terrain analyzer tool and solves the constraint satisfaction problem (CSP) using answer set programming (ASP) paradigm.

The solution process involves two stages: acquiring the answer set and optimizing the results. Two optimization methods are used and analyzed one of which minimizes the gap values between the structures and the other minimizes the cost of the wall using buildings' resource cost values. Each of the methods has its advantages and disadvantages as one method might be preferred over the other method in certain situations with respect to the choke point's width.

GERÇEK ZAMANLI STRATEJİ OYUNUNUNDA BİNA YERLEŞTİRME PROBLEMİNİN ÇÖZÜLMESİ

(ÖZET)

StarCraft, Blizzard Entertainment tarafından üretilen ve 1998 yılında piyasaya sürülen gerçek zamanlı bir strateji oyunudur ve günümüzde gerçek zamanlı yapay zeka sistemleri için kullanılan platformdur. Oyun motorunun detaylarına takılmadan oyun içi veriye erişimi sağlayan ve bir hack-tool olan BWAPI'nin 2009 yılında geliştirilmesiyle gerçek zamanlı yapay zeka sistemlerine dair araştırmalar hız kazandı. Son zamanlarda yayınlanan A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft, Fast Heuristic Search for RTS Game Combat Scenarios, vb. makaleler BWAPI'nin ne kadar kullanışlı olduğunun kanıtıdır. Diğer birçok problemin arasında bulunan üs girişini kapatma problemi, diğer adıyla wall-in, bu çalışmada ele alınmaktadır.

Proje kapsamında öncelikle belli bir birim sırasını takip eden ve Michal Certicky'nin Implementing a Wall-In Building Placement in StarCraft with Declarative Programming makalesinden esinlenerek gerçeklennen ve geliştirilen mantıksal programı çalıştıran otonom bır yapay zeka sistemi gerçeklenmiştir. Mantıksal program, otonom yapay zeka sistemi tarafından BWAPI'nin yardımıyla elde edilen arazi bilgisi ile bir constraint satisfaction problemı olarak modellenen üs girişini kapama problemini answer set programming (ASP) paradigmasını kullanarak çözmektedir.

Oyunda binalar yan yana yerleştirildiğinde binaların yerleşimine göre aralarında boşluklar kalmaktadır. Çözüm yöntemi bu boşlukları dikkate alarak bir çözüm üretmektedir. Bir diğer dikkate alınan ölçüt ise binaları inşa etmek için gerekli olan kaynak miktarıdır.

Çözüm süreci cevap kümesini elde etmek ve optimum çözümü bulmak olarak iki aşamadan oluşur. Kullanılan ve analiz edlien optimizasyon metotları binalar arasındaki boşluğu ve duvarın oluşumunda kaynak harcamasını minimize eden iki metottan oluşur. Her yöntemin kendine göre avantajları ve dezavantajları olmasına karşın boğum noktasının genişliğine bağlı olarak bazı durumlarda bir metodun diğerine göre tercih edilebilir olduğu gözlemlenmiştir.

Boğum noktası uzunluğu, uygun olan yer karosu (tile) sayısı, boğum noktasının bulunduğu bölgenin şekli, harita dekorları (ağaçlar, göçükler, engeller, vb.) gibi değişkenlerin bulunması nedeniyle tek başına programın çalışma süresini tam olarak belirleyebilmek için yeterli bir parametre değildir. Buna rağmen bu parametrenin incelenmesi yeteri kadar ve uygulanabilir sonuçlar elde etmek için yeterlidir.

Çalışma sonunda elde edilen sonuçlara göre mantıksal programa gerçek senaryolara uygun olan dört saniyelik bir koşma üst sınırı verildiğinde her iki optimizasyon yöntemi de 280 piksel genişlik ve sonrası için dört saniyeden uzun sürede program sonlanmaktadır. Bu sonuca göre duvar örülebilecek en geniş boğum noktası 280 piksel uzunluğu geçmemelidir. 135 piksel - 280 piksel aralığındaki sonuçlar incelendiğinde boşluk optimizasyonu yönteminin dört saniyeden uzun sürdüğü ancak kaynak optimizasyonunun ise diğer yönteme göre oldukça çabuk sonlandığı görülmüştür. Bu nedenle 135 piksel - 280 piksel

uzunluk aralığındaki boğum noktaları için kaynak harcaması optimizasyonu tercih edilmelidir. Bundan daha dar doğum uzunluğuna sahip bölgelerde ise boşluk optimizasyonunu kullanan mantıksal program dört saniyenin altında sonlandığı için boşluk optimizasyonu 135 piksel ve altındaki boğum noktası genişliğine sahip bölgeler için tercih edilen yöntem olmalıdır.

Mantıksal programın ürettiği sonuçlar oyun içinde gözlemlendiği zaman harita dekoru bulundurmayan ve görece olarak daha basit bir şekle sahip boğum noktalarında Şekil 15 ve Şekil 16'da görüldüğü gibi iki binanın yerlerinin değiştiği; harita dekoru bulunduğu zaman boşluk optimizasyonu yapıldığında Şekil 17 ve Şekil 18'de görüldüğü gibi bu dekorun duvar örerken kullanıldığı gözlemlenmiştir. Bunlardan farklı olarak ise bazı durumlarda, mantıksal programda ortam modellenirken bazı basitleştirmelerin kullanılması nedeniyle programın Şekil 19'da görüldüğü gibi optimal olmayan sonuç üretebildiği ya da çözüm olduğu halde programın çözüme ulaşamadığı gözlemlenmiştir.

Basitleştirmelerden doğan sorunları ortadan kaldırmak için yürüme karolarının (walk tile) birbirinden uzaklığını kontrol eden ek bir kısıtlama getirilebilir; yapay zeka sisteminin profosyonel StarCraft oyuncularına daha yakın bir performans gösterebilmesi için boğum noktalarından avantaj elde edebilmek adına kısmi duvar örme algoritması geliştirilebilir.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Project Scope and Purpose

The algorithms used in strategy games such as Go and Chess has come a long way since the beginning of the AI researches and managed to beat the top human players in late 90s and early 2000s. However, things are way more complicated in real-time environments such as Real-Time Strategy (RTS) games. StarCraft is an RTS video game created in 1998 by Blizzard Entertainment and is the game the AI algorithms are tested on in this project. In environments such as RTS games the space and states are much larger than the ones in regular strategy games while the constraints are way stricter, thus making RTS AI way more complicated than the regular AI. To give an idea about the vastness of state space in RTS games, below is the quote of Churchill from his paper [1]:

"From a theoretical point of view, the state space of a StarCraft game for a given map is enormous. For example, consider a 128_128 map. At any given moment there might be between 50 to 400 units in the map, each of which might have a complex internal state (remaining energy and hitpoints, action being executed, etc.). This quickly leads to an immense number of possible states (way beyond the size of smaller games, such as Chess or Go). For example, just considering the location of each unit (with 128_128 possible positions per unit), and 400 units, gives us an initial number of $16384^{400} \approx\approx 10^{1685}$*. If we add the other factors playing a role in the game, we obtain even larger numbers."*|

StarCraft is a popular RTS game where players compete with each other by collecting resources, making armies, applying various tactics in combat and try to destroy the opponent player [2]. A sample screenshot from the game is given in Figure 1.

Figure 1: A Protoss (a race in game) army destroying its opponent.

RTS games have been an interesting area of research domain for Artificial Intelligence due to the representation of well-defined complex adversarial systems and their divisibility into numerous interesting sub-problems [1]. One of the sub-problems is "The Walling-in Problem", as called by the StarCraft community, Liquipedia, which is defined as *"Walling is the act of intentionally narrowing the passageway between strategical points by placing buildings in a certain setup."* [3].

Walling-in is an important feature of a professional player which has a crucial outcome: In the early game, professional players place their buildings around the choke-points (narrow passages) so that an early aggression could be suppressed. If players do not conduct this strategy, the attacking units of the opposing player might get inside the base, kill the worker units and damage the economy severely, if not destroy all of the units and win the game. A serious damage to economy most probably leads to a defeat. Therefore walling-in is considered very important.

Current StarCraft game playing agents focused on other primary sub-problems such as build-order planning, micro-management, combat tactics, etc. and did not pay much attention to this problem. This project will deal with this problem by formulating the walling-in as a CSP problem, implementing algorithms to solve the problem in declarative programming (ASP, clasp) paradigm. The project is aiming to analyze the current solution and improve it by modifying it.

1.2. Area of Usage

The project is directly related to the ever growing industry of video game entertainment and the research area of real-time artificial intelligence. Improving the gameplay of AI agents, getting them to a higher level near their human rivals will make the gameplay more enjoyable, more realistic and more challenging. The future of RTS AI offers more platforms for human-computer interaction and what the industry needs: more challenge, more fun, more entertainment. In addition, StarCraft leads the real-time AI research area thanks to the BWAPI and fairly complicated game play. Most important papers are published in 2012/2013 and the ever growing research continues while more and more researchers are interested in the area.

1.3. Resource, Time Management and Expectations

There is an API called BWAPI which lets the users communicate with the game engine directly, thus helping the programmer focus on the AI aspects rather than dealing with the irrelevant components of the game engine. A documentation of BWAPI can be obtained from its website which is provided in the references section. Other resources are the game itself*: StarCraft: Brood War* and the game playing agent: *UAlbertaBot* and the example BWAPI AI module.
As for the time management, the project can be divided into subsections:

- Survey of AI techniques used in RTS games (1 – 1.5 months)
- Analysis of UAlbertaBot & Example AI Module (1 month)
- Formulation of the problem (2.5 months)
    - Sensory input
    - Available actions
- Implementation of algorithms (4 months)

- Analysis of the implemented search methods (1 month)
- Test & Iterations (1.5 month)
- Report & Presentation (1.5 month)

At the end of the project, among a successful implementation of the walling-in algorithm, the gameplay of an agent is improved by the implemented algorithm helping them survive early aggressions. This is one of the sub-problems in the RTS game AI that will be attempted to be solved.

1.4. Risk Analysis and Gantt Chart

There are some foreseen risks in the project:

- **Inaccurate time estimation**

  Time estimation made in the beginning of the project might be inaccurate and may result in late submission, less number of implemented algorithms, poor analysis and comparison report. The mitigation of this risk is to monitor the project every week and track the progress carefully.

- **The lectures attended during the terms might be overwhelming**

  Density of the lectures might interfere with the project and decrease the progress velocity, resulting in delays and same negative outcomes described in previous risk.

- **Lack of AI Knowledge**

  Prior to the project, there will be a lack of AI knowledge since no AI courses have been taken thus far. Learning problem formulations, algorithms used for different problems might take some time, which, again, will result in delays and negative outcomes as described above.

- **Poor algorithm performance**

  The implemented algorithms might perform poorly when applied in the real-time environment of the video game. The mitigation of this risk is to do as many tests as possible to measure the performance of the agent and apply necessary improvements.

The time management of the project is explained above and the Gantt Chart of the project workflow is given in Table 1.

Table 1 - Gannt chart of the project

	FALL TERM	SEMESTER	SPRING TERM
Survey of AI techniques used in RTS games			
Analysis of UAlbertaBot			
Formulation of the problem			
Learning how to use Clasp, Implementation of Algorithms			
Analysis of implemented algorithms			
Test & Iterations			
Report & Presentation			

2. THEORETIC RESEARCH

2.1. Sub-Problems, Recent Challenges, Techniques

There are lots of sub-problems in which AI techniques are used in StarCraft. The following are the recent challenges in the field [6]:

- **Planning**
  Since the state space is vast, game-tree search is not applicable, and therefore multiple level of abstraction is needed: long-term planning for a balance in good economy – large army, short-term planning for combat decisions.

- **Learning**
  Three types of learning: *prior learning*, *in-game learning*, *inter-game learning*. Prior learning includes extracting information from replays, or map terrain to gain advantage before the game. In-game learning includes opponent modeling. Inter-game learning concerns about improving the agent from game to game.

- **Uncertainty**
  Map is partially observable, therefore scouting is needed for gathering information about the opponent. Games are adversarial, impossible to predict what opponent will do. Human players consider actions that opponents are likely-to-do.

- **Spatial and temporal reasoning**
  Spatial reasoning: Terrain exploitation: higher ground has advantage over lower ground since units on lower ground has no sight of the higher ground. Deciding where to engage combat to use bottlenecks for advantage. Base expansion, finding and deciding best location according to opponent base's location.

  Temporal reasoning: timing attacks, retreats. Long-term planning of actions that has a higher impact on economy such as upgrades, strategy switching etc. A diagram illustrating spatial and temporal reasoning used in StarCraft is shown in Figure 2.

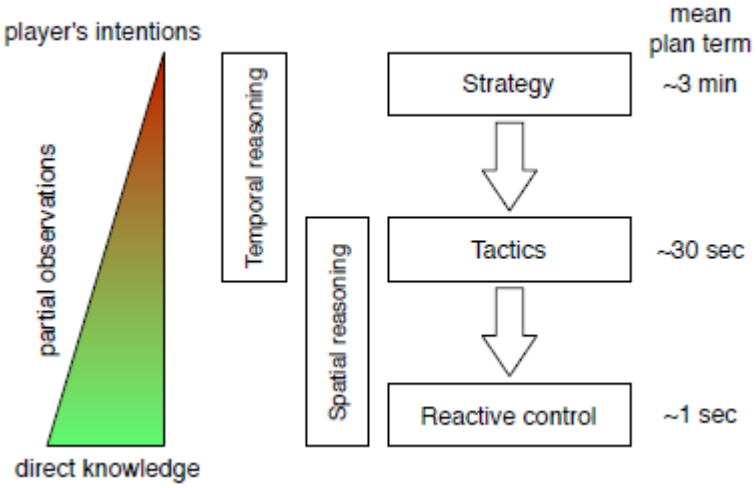

Figure 2 - Spatial and temporal reasoning [6]

- **Domain knowledge exploitation**
  Traditional strategy games (chess, go) exploited large amounts of the domain knowledge and developed good evaluation functions. On the other hand StarCraft has a much larger domain and approaches are in two ways: Hard-coding the strategies into agents and running an algorithm to decide instead of an adaptive approach; large set of replays are created and agents try to learn from replays. Both are still open problems.

- **Task decomposition**
  Strategy: high level decision making: finding an efficient or counter strategy.
  Tactics: Implementation of the current strategy: army composition, building positioning, army movements, timing etc. Tactics concern a group of units.
  Reactive Control: Micro-management: Firing, retreating, kiting (hit-and-run).
  Terrain Analysis: Choke-points, mineral and gas (resource) locations, high grounds.
  Intelligence Gathering: Scouting the opponent.

For the human side of decision making, the StarCraft community mentions two tasks [7]:

- Micro management: micro-management roughly corresponds to the reactive control which involves the individual or a small group of unit control and positioning and etc.
- Macro management: macro covers all of the above except reactive control. Unit producing, expanding the base for extra resource income at the right time, choosing the appropriate unit combination to counter the opponent etc.

Current studies in RTS game AI divided these tasks into three parts as shown in figure 2: strategy, tactics and reactive control.

2.2. Bot Architecture & Some Algorithms Used In the Area

UAlbertaBot [8] is a StarCraft playing agent developed by David Churchill and his team. The bot has a modular and hierarchical structure as illustrated in Figure 3:

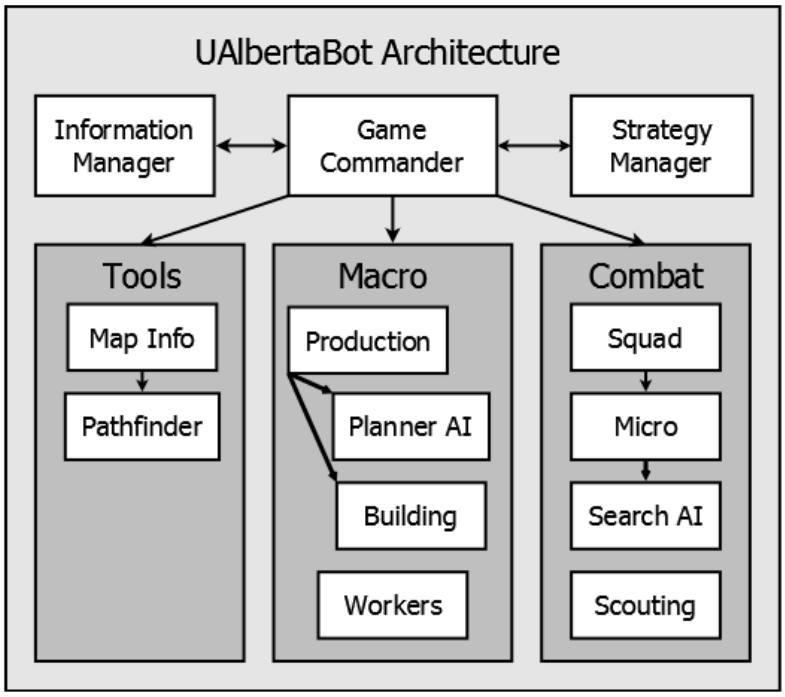

Figure 3: UAlbertaBot architecture [9]

Tasks are partitioned among modules according to their strategic meaning, as an inspiration from the military. High level strategic decisions are made by the game commander by gathering all the information about the current game state. From there, commands are given to sub-commanders and so on which are directly in charge of completing the low level tasks [9].

To solve the problems addressed in the previous section, some algorithms are implemented, given the specifications of the StarCraft task environment as shown in table 2:

Table 2 - StarCraft world model

Task Environment	**Observable**	**Agents**	**Deterministic**	**Episodic**	**Static**	**Discrete**
StarCraft	Partially	Multiple	Stochastic	Sequential	Dynamic	Discrete

Below are some of them used on the UAlbertaBot:

- ABCD (Alpha-Beta search Considering Duration) algorithm is used for micro-management (deciding what each unit will perform in a combat situation) which can be classified as a zero-sum situation [9] as shown in Figure 4. This property together with the fully observable state variables and simultaneous moves places combat games in the class of "stacked matrix games" [1]. Those games can be solved by backward induction starting with terminal states via Nash equilibrium computations.

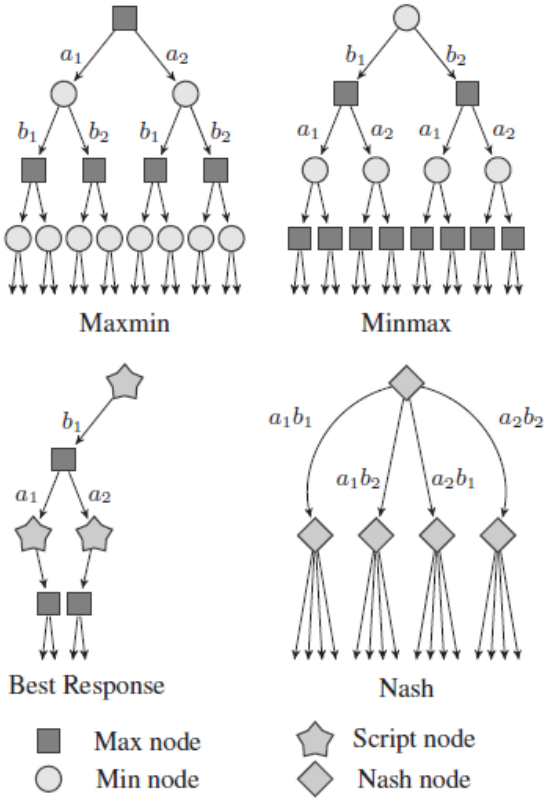

Figure 4: Search algorithms used in UAlbertaBot [1]

- An any-time depth first branch and bounding algorithm is used for build order planning. An ordered sequence of actions which produces a given set of goal units is called a *build order* [1].
- There are a few approaches addressing strategic decision making: hard-coded, planning based and machine learning based approaches. Therefore, various machine learning algorithms are used as well [1].

3. ANALYSIS AND MODELING

3.1. Environment Variables

Liquipedia, the Wikipedia of the StarCraft universe, defines the term “walling” or “walling-in” as follows: “*Walling is the act of intentionally narrowing the passageway between strategical points by placing buildings in a certain setup.*” [3].

Walling in most commonly includes, but is not limited to, closing the entrance of the player’s base, aiming to block the enemy units that are trying to get through as shown in Figure 5.

Figure 5 - A Terran player walling in [11]

The idea behind the walling is shrinking the free space when a combat scenario occurs, therefore providing advantage to the defending player’s units by reducing the surface area of the units when a composition of melee attackers strike.

3.1.1. Tiles

In StarCraft Broodwar, the unit length is measured by pixels. A single square area that is visible to a player when she is building structures is a 32x32 pixel square, named a build tile.

Figure 6 - A Terran player building a Barracks (3x4)

Figure 6 is a screenshot from the game in which a Terran player is going to build a structure named Barracks, which occupies 3x4 build tiles, thus having a width of 128 pixels and a height of 96 pixels. It can be seen from the figure that the lower left corner of the building collides with another building in construction.

The second type of tiles is the walk tile, which has a dimension of 8x8 pixels on which units are able to walk.

3.1.2. Gaps

When two buildings are placed next to each other as to occupy adjacent build tiles, the tiles are not fully occupied due to the fact that the buildings have gaps associated with their sides. Therefore, two neighbor tiles have a gap as much as the summation of each of the structure's associated side. The values of gaps for each building can be found in Liquipedia. The Terran buildings' gap list is shown in Figure 7.

Following this, when a Supply Depot (in the 1st row and the 3rd building in Figure 7) is placed above a Barracks (the 1st row and 2nd building) the gap value will be 13, which is the lowest combination of a gap value when these two buildings are built next to each other. Thus, Supply-Depot-above-Barracks is usually a preferred building placement strategy. When the gaps are wide enough for smaller units, even though all build tiles are occupied by the buildings, these units can get through. The main idea is not blocking the way only by buildings, but it is to narrow the passage so that the opponent's melee units will have a smaller surface area of their targets and thus provide advantage over the combat to the defending player.

3.2. ITUBot and Terrain Analysis

BWAPI provides a built in terrain analysis tool called BWTA (BroodWar Terrain Analysis) which reads and analyzes the map data in a different parallel-to-game thread. At the end of the analysis, all the tile information is able to be obtained by the BWAPI

interface and can be drawn in game area as shown in Figure 8. The map is divided into regions by the choke points, which are defined in Liquipedia [3] as:
"A choke is a narrow pathway or area that creates a funneling effect when moving through it. Similar to high ground, a choke massively favors the defender over the attacker. Examples of a choke include ramps and narrow passageways."

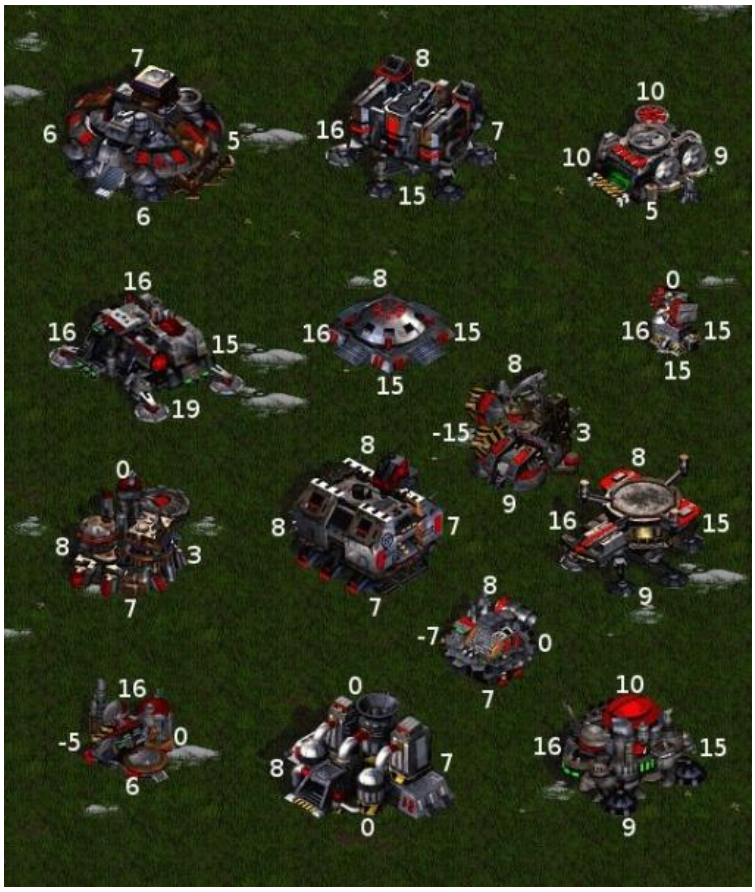

Figure 7 – Enumeration of Terran buildings' gap sizes [10]

Figure 8 - After the terrain analysis, the borders of the base are drawn green and the choke point is drawn red. Orange squares are the building placements as the result of the solver, which is not a part of the terrain analysis and will be explained in the following section.

ITUBot has its build order planned beforehand, meaning that the units and structures the bot is going to make is defined at the initialization stage, i.e., hardcoded into the bot. Since build order planning is another area of the RTS AI, it is currently left for further study (see [12]). The structures are built after the map analysis is finished.

Upon the completion of map analysis, the build and walk tile data are made ready by the tile analysis made by the bot. BWAPI provides the properties of the tiles such as a tile is buildable and walkable or there exists a path between given two tiles or positions (pixels). In order to simplify the logic program, walk tiles (8x8) are extrapolated to build tiles (32x32) by checking the four center walk tiles in a build tile, and then passed to the logic program as walkable tiles.

In the build tile given in Figure 9 there exists 16 walk tiles. The green tiles are walkable while the red tiles are not walkable tiles. The program checks the 4 center walk tiles and according to the number of walkable tiles, this tile is passed as a walkable tile or ignored. Due to this simplification, some complications are foreseen to occur.

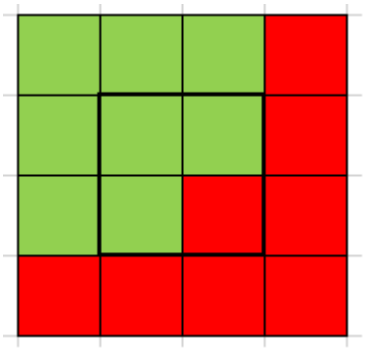

Figure 9 – The abstraction representation (8x8) of a walk tile (32x32) for the logic program. This tile would be considered as a walk tile and appear as a purple tile in the representation.

The tile analysis after the map analysis completes is shown in Figure 10. The purple tiles are the walk tiles represented in the logic program. The logic program produces a result by checking a path availability from a base point to an outside-base point by checking whether adjacent tiles are blocked or not, which will be explained later. The green tiles are inside-base buildable tiles. The presence of red squares in green circles represents that this is a suitable tile to build a supply depot (a structure is placed according to top left tile of the building layout meaning that a 2x3 building built on 44, 70 will occupy the following tiles: 44,70 , 45,70 , 46,70 , 44,71 , 45,71 , 46,71). Likewise, blue squares represent tiles that are suitable for a barracks construction. Finally, the cyan tiles represent the outside base tiles. The chokepoint is in between the area where green and cyan tiles are adjacent to each other. Note that since in BWAPI, the coordinates (0, 0) refer to the upper left corner of the map, increasing X means going right and increasing Y means going down. This means that when buildings are placed on a coordinate (X, Y), the upper left tile of the building is placed on this coordinate.

3.3. Problem Formulation

Given the entities described above (tiles, gaps, structures), it is possible to formulate the problem as a CSP (Constraint Satisfaction Problem). [13] A CSP problem typically has the form of a triple $< X, D, C >$ where X is the variable (in our case, buildings), D is the value (the tile position on which the buildings can be built) assigned to that variable and C are the constraints [13]. Certicky defines three constraints excluding the Protoss' racial building placement constraint [2]:

1. All the buildings should be able to be built on their designated locations depending on the terrain properties (green squares after the analysis, figure 10).
2. Buildings cannot overlap (red square in figure 6).
3. There should not be a path available from inside the base to outside after the buildings are constructed.

A CSP defined in this way often has multiple solutions some of which are more desirable than others. To build a tight wall which prevents smaller units from passing through, a player has to take into consideration the gap values of the buildings. This situation leads to an optimization problem where the minimum gap value is desired in an answer set programming framework.

Certicky solves both the constraint and the minimization problem using Answer Set Programming (ASP) paradigm of logic programming by a tool named *clingo*. Clingo is the combination of the grounder *gringo* and the solver *clasp*, written in C++ and developed at the University of Potsdam. Clingo provides its users the basic ASP constructs such as rules, constraints and facts, as well as a support for generator rules, optimization statements and built-in arithmetic functions and aggregates. For more information, please refer to Certicky's *Wall-in Building Placement (2013) [2]* and the guide provided with the solver bundle by University of Potsdam [14].

Figure 10 - Tile representation of a game environment. The purple tiles are walk tiles, the cyan tiles are the tiles that are outside the base region and the green tiles are buildable inside-base tiles. Red and blue tiles represent the available location for certain buildings, such as Supply Depot or Barracks.

4. DESIGN AND IMPLEMENTATION

4.1. Encoding the Problem in Clasp

An existing logic program originally designed by Michal Certicky solves the walling problem by using ASP [2]. After implementing this solver, it has been observed that it needs modifications for better results. In this project, this solution is extended to improve its efficiency.

The ASP formulation of the problem includes the building variables such as resource cost, width, height and gap which are hardcoded into the logic program as facts. Then, the buildings to be used at the wall construction are specified.

```
% Building / Unit types
buildingType(marineType).
buildingType(barracksType).
buildingType(supplyDepotType).

% Size specifications
width(marineType,1).        height(marineType,1).
width(barracksType,4).      height(barracksType,3).
width(supplyDepotType,3).   height(supplyDepotType,2).

% Cost specifications
costs(supplyDepotType, 100).
costs(barracksType, 150).

% Gaps
leftGap(barracksType,16).       rightGap(barracksType,7).       topGap(barracksType,8).         bottomGap(barracksType,15).
leftGap(supplyDepotType,10).    rightGap(supplyDepotType,9).    topGap(supplyDepotType,10).     bottomGap(supplyDepotType,5).

% Facts
building(barracks1).    type(barracks1, barracksType).
building(barracks2).    type(barracks2, barracksType).
building(supplyDepot1). type(supplyDepot1, supplyDepotType).
building(supplyDepot2). type(supplyDepot2, supplyDepotType).
building(supplyDepot4). type(supplyDepot4, supplyDepotType).
building(supplyDepot3). type(supplyDepot3, supplyDepotType).
```

After presenting the building facts, constraints are specified. The first constraint states that two different buildings cannot occupy the same tile position (cannot overlap) which corresponds to the second constraint specified when formulating the problem. In other words, a tile position (X, Y) occupied by Building1 and Building2 and '*Building1 is not the same as Building2*' cannot all hold true at the same time.

```
% Constraint: two units/buildings cannot occupy the same tile
:- occupiedBy(B1, X, Y), occupiedBy(B2, X, Y), B1 != B2.
```

Next rule states that the occupied tiles by the buildings must be defined. Using the previously defined entities of the buildings, the rule can be specified as follows:

'*If a building is placed on position (X1, Y1) that has a type of BuildingType and that BuildingType has a width of Z and a height of Q, and X2 is in the range of X1 inclusive and X1+Z exclusive, likewise Y2 is in the range of Y1 inclusive and Y1+Q exclusive and the tile (X2, Y2) is walkable; then the position (X2, Y2) is occupied by the building B'*.

```
% Tiles occupied by buildings
occupiedBy(B,X2,Y2) :- place(B, X1, Y1),
                       type(B, BT), width(BT,Z), height(BT, Q),
                       X2 >= X1, X2 < X1+Z, Y2 >= Y1, Y2 < Y1+Q,
                       walkableTile(X2, Y2).
```

The following set of rules simply calculates the vertical and horizontal gaps of the buildings if they have adjacent tiles in a similar fashion to the previous rule. These calculated gap values will later be used in the optimization phase.

```
% Gaps between two adjacent tiles, occupied by buildings.
verticalGap(X1,Y1,X2,Y2,G) :-
    occupiedBy(B1,X1,Y1), occupiedBy(B2,X2,Y2),
    B1 != B2, X1=X2, Y1=Y2-1, G=S1+S2,
    type(B1,T1), type(B2,T2), bottomGap(T1,S1), topGap(T2,S2).

verticalGap(X1,Y1,X2,Y2,G) :-
    occupiedBy(B1,X1,Y1), occupiedBy(B2,X2,Y2),
    B1 != B2, X1=X2, Y1=Y2+1, G=S1+S2,
    type(B1,T1), type(B2,T2), bottomGap(T2,S2), topGap(T1,S1).

horizontalGap(X1,Y1,X2,Y2,G) :-
    occupiedBy(B1,X1,Y1), occupiedBy(B2,X2,Y2),
    B1 != B2, X1=X2-1, Y1=Y2, G=S1+S2,
    type(B1,T1), type(B2,T2), rightGap(T1,S1), leftGap(T2,S2).

horizontalGap(X1,Y1,X2,Y2,G) :-
    occupiedBy(B1,X1,Y1), occupiedBy(B2,X2,Y2),
    B1 != B2, X1=X2+1, Y1=Y2, G=S1+S2,
    type(B1,T1), type(B2,T2), rightGap(T2,S2), leftGap(T1,S1).
```

The following rule is specified to be used as an alternative optimization criterion, which simply calculates the mineral cost of each building that are used in wall construction.

```
cost(B, C) :- place(B, X, Y), type(B, BT), costs(BT, COST), C=COST.
```

Now that the rules and constraints regarding the building placement are specified, the tile information is required. BWAPI's BWTA module thankfully does the analysis to the point where the tiles contain data such as buildable and walkable. The tile information surrounding a choke point is passed to the solver by the bot. A certain range of tiles are passed to the solver in order to reduce the computation time. These facts – buildable tiles in particular – directly satisfy the 1st constraint specified in the problem formulation, i.e., '*All the buildings should be able to be built on their designated locations depending on the terrain properties*'.

```
% Tile information
walkableTile(8, 45).
buildable(barracksType, 32, 60).
buildable(supplyDepotType, 20, 61).
walkableTile(8, 46).
walkableTile(8, 47).
.
.
.
```

In compliance with the 3rd constraint, to check whether there is a path exists from inside the base to outside the base, we must specify the inside and outside base coordinates, as well as the rules for path existence. *insideBase* position is chosen as the closest tile to the

Command Center among the tiles passed to the solver while *outsideBase* position is chosen as the farthest tile to the Command Center which has a path between them. The path existence must be checked, otherwise, if the farthest chosen tile has no connections with the rest of the near-base tiles, any solution will be accepted by the solver even though they do not block the base entrance.

```
insideBase(36, 57). outsideBase(12, 49).

% Constraint: Inside of the base must not be reachable.
:- insideBase(X2,Y2), outsideBase(X1,Y1), canReach(X2,Y2).
```

Next, the reachability rules among tiles must be defined. The first rule states that if a walkable tile is occupied by a building, then that tile is blocked. The second rule simply states that the outside base is reachable by definition, meaning that the solver will start to check the path existence from the outside base. Finally, the set of reachability rules simply states that if any of the adjacent (all 8 directions) walkable tiles are not blocked, that tile is reachable. This is the exact reason that the actual smaller walk tiles are extrapolated to the size of build tiles for simplification purposes. These set of rules take care of the path finding problem associated with the 3rd constraint.

```
% Reachability between tiles.
blocked(X,Y) :- occupiedBy(B,X,Y), building(B), walkableTile(X,Y).

canReach(X,Y) :- outsideBase(X,Y).

canReach(X2,Y) :-
    canReach(X1,Y), X1=X2+1, walkableTile(X1,Y), walkableTile(X2,Y),
    not blocked(X1,Y), not blocked(X2,Y).
canReach(X2,Y) :-
    canReach(X1,Y), X1=X2-1, walkableTile(X1,Y), walkableTile(X2,Y),
    not blocked(X1,Y), not blocked(X2,Y).
canReach(X,Y2) :-
    canReach(X,Y1), Y1=Y2+1, walkableTile(X,Y1), walkableTile(X,Y2),
    not blocked(X,Y1), not blocked(X,Y2).
canReach(X,Y2) :-
    canReach(X,Y1), Y1=Y2-1, walkableTile(X,Y1), walkableTile(X,Y2),
    not blocked(X,Y1), not blocked(X,Y2).
canReach(X2,Y2) :-
    canReach(X1,Y1), X1=X2+1, Y1=Y2+1, walkableTile(X1,Y1), walkableTile(X2,Y2),
    not blocked(X1,Y1), not blocked(X2,Y2).
canReach(X2,Y2) :-
    canReach(X1,Y1), X1=X2-1, Y1=Y2+1, walkableTile(X1,Y1), walkableTile(X2,Y2),
    not blocked(X1,Y1), not blocked(X2,Y2).
canReach(X2,Y2) :-
    canReach(X1,Y1), X1=X2+1, Y1=Y2-1, walkableTile(X1,Y1), walkableTile(X2,Y2),
    not blocked(X1,Y1), not blocked(X2,Y2).
canReach(X2,Y2) :-
    canReach(X1,Y1), X1=X2-1, Y1=Y2-1, walkableTile(X1,Y1), walkableTile(X2,Y2),
    not blocked(X1,Y1), not blocked(X2,Y2).
```

Finally, all possible building placements are generated and then tested if they are in the set of solutions to our CSP. It is desired to check solutions with exactly one *place(barracks1,X,Y)* and *place(supplyDepot1*,X,Y) while the other building placements can be omitted when minimizing the solution. The last line specifies what to minimize, which is the cumulative cost value in this case.

```
% Generate all the potential placements.
1[place(barracks1,X,Y) : buildable(barracksType,X,Y)]1.
0[place(barracks2,X,Y) : buildable(barracksType,X,Y)]1.
1[place(supplyDepot1,X,Y) : buildable(supplyDepotType,X,Y)]1.
0[place(supplyDepot2,X,Y) : buildable(supplyDepotType,X,Y)]1.
0[place(supplyDepot3,X,Y) : buildable(supplyDepotType,X,Y)]1.
0[place(supplyDepot4,X,Y) : buildable(supplyDepotType,X,Y)]1.

% Optimization criterion
#minimize [cost(B, C) = C].
```

In the case of gap minimization, prioritization among minimization literals are used. The highest priority belongs to the vertical and horizontal gap together (notice the @1). In case of no prioritization, which is the case in Certicky's solution, first the horizontal gap would be minimized and then the vertical gap would be minimized. In our case, the total gap value is minimized rather than individual gap types. Since the solver is run once rather than adding buildings iteratively in Certicky's solution, we have to specify the minimization criterion of the building placements. Otherwise, all the buildings will be used in the wall configuration even though the path might be blocked with fever buildings. That is why the minimization for buildings must be specified and must be assigned to the 2nd priority.

```
% Optimization criterion
#minimize [horizontalGap(X1,Y1,X2,Y2,G) = G @1 ].
#minimize [verticalGap(X1,Y1,X2,Y2,G) = G @1 ].
#minimize [place(supplyDepot2,X,Y) @2].
#minimize [place(supplyDepot3,X,Y) @2].
#minimize [place(supplyDepot4,X,Y) @2].
#minimize [place(barracks2,X,Y) @2].
```

As for the comparison of this study's solution and Certicky's solution, this study ignores the enemy units' dimensions when checking reachability since narrowing down the available path to a single unit's size grants enough tactical advantage at this level of gameplay.

4.2. Bot Architecture

ITUBot is built on the example AI module bundled with the BWAPI installer. It inherits all the event functions (`onFrame()`, `onSendText()`, etc.) from the example AI module, as well as the terrain analysis functions, draw functions and data functions as shown in Figure 11. StarCraft AI bots have a similar architecture to that of game engines. `onStart()` function is used for data initializations, `onFrame()` function is called every frame and the primary function for AI calculations. The rest of the module functions are simply event handlers that go through certain procedures after some designated events occur such as a completion of a unit or a discovery of a unit.

Draw functions are used to draw analysis data on the game area after the map analysis is finished. Data functions are used to display information in the game area if certain flags are set. Build order functions return the build order, initialize the build order and execute the build order respectively. Build order execution is a loop that is called at each frame that checks if the conditions are satisfied to build a structure or a unit (population constraint, resource cost, etc.) and then sends the next item in the build order with the construction command to a designated worker.

Wall functions return the vector of pairs of buildings and their tile positions; initialize the logic program's source code and executes the logic program, respectively. Logic program's initialization and execution is done after the map analysis is complete.

ITUBot

```
_buildOrder : queue<UnitType>
_wall : vector< pair<UnitType, TilePosition> >
```

```
// BWAPI AI Module functions
void onStart();
void onEnd(bool isWinner);
void onFrame();
void onSendText(std::string text);
void onReceiveText(BWAPI::Player* player, string text);
void onPlayerLeft(BWAPI::Player* player);
void onNukeDetect(BWAPI::Position target);
void onUnitDiscover(BWAPI::Unit* unit);
void onUnitEvade(BWAPI::Unit* unit);
void onUnitShow(BWAPI::Unit* unit);
void onUnitHide(BWAPI::Unit* unit);
void onUnitCreate(BWAPI::Unit* unit);
void onUnitDestroy(BWAPI::Unit* unit);
void onUnitMorph(BWAPI::Unit* unit);
void onUnitRenegade(BWAPI::Unit* unit);
void onSaveGame(string gameName);
void onUnitComplete(BWAPI::Unit *unit);

// draw functions
Void drawStats();
void drawBullets();
void drawVisibilityData();
void drawTerrainData();
void drawChokeData();

// data functions
void showPlayers();
void showForces();
bool show_bullets;
bool show_visibility_data;

// build order functions
queue<BWAPI::UnitType>& buildOrder();
void populateBuildOrder();
void executeBuildOrder(BWAPI::Unit* unit);

// wall functions
vector<std::pair<BWAPI::UnitType, BWAPI::TilePosition> >& wall();
static void initClingoProgramSource();
static void runASPSolver();
```

Figure 11 - ITUBot UML diagram

Figure 12 illustrates the overview of the execution of AI code by displaying the inner workings of `onStart()` function and `onFrame()` function.

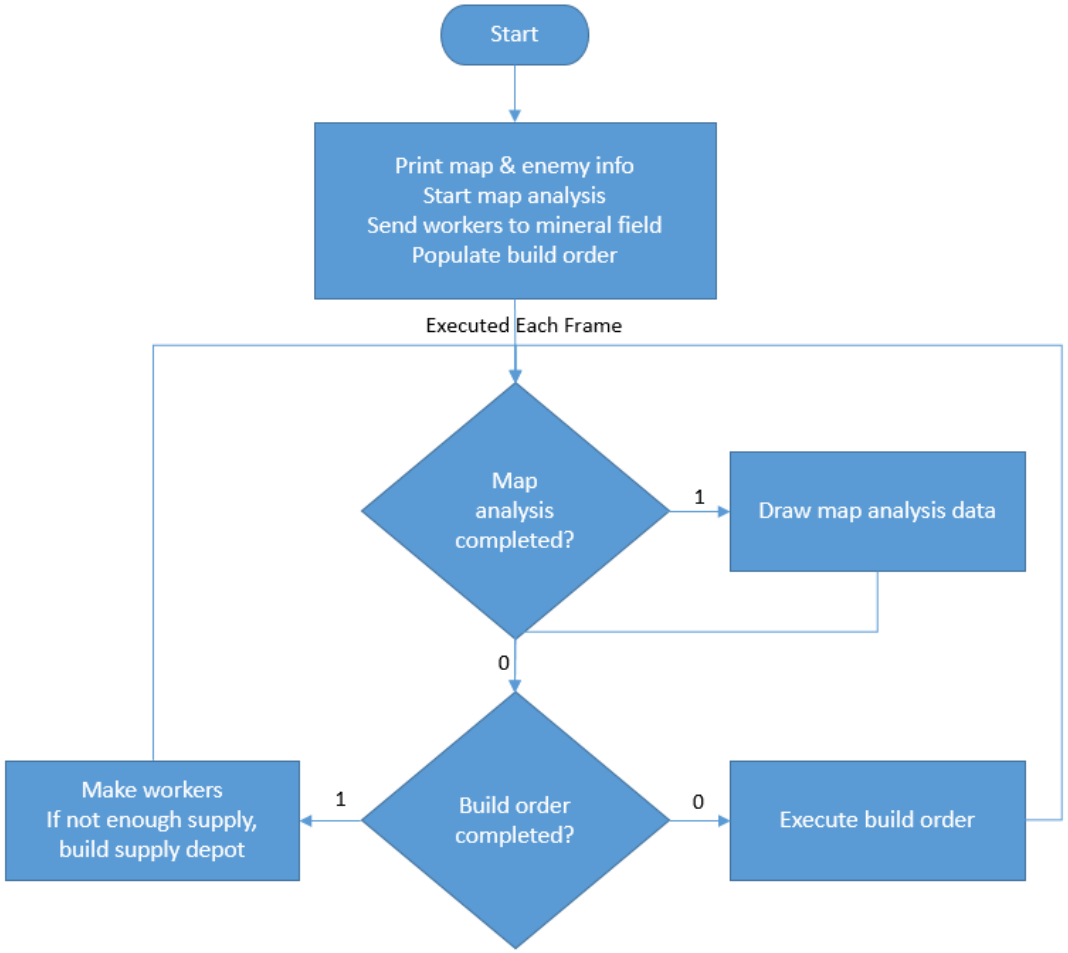

Figure 12 - Flow chart of ITUBot

5. TESTING & RESULTS

Experiments are done for testing the performance of the ASP solver on different settings. The machine (ASUS N550JV) specifications used for testing the solver are as follows:

- CPU: Intel Core i7-4700HQ @ 2.40 GHz
- RAM: 12GB
- HDD: 1TB HGST HTS541010A9E680
- OS: Windows 8.1 Pro

There are 8 different maps used when calculating the solver's running time. The results are presented in table 3. Gap minimization data is given in bold and the time results are in seconds.

Table 3 - Map specific solver results with different optimization criteria

Map	Optimization Criteria	Start Location	Width	Building Count	Time	Time Log10	Time Log2
(4)Boxed In	**Gap**	top right	283,75	**3**	**116,25**	**2,07**	**6,86**
	Cost			3	5,88	0,77	2,55
	Gap	bot right	186,76	**3**	**3,64**	**0,56**	**1,86**
	Cost			3	0,81	-0,09	-0,30
(2)Astral Balance	**Gap**	top	131,06	**3**	**2,30**	**0,36**	**1,20**
	Cost			3	0,77	-0,12	-0,38
	Gap	bot	135,20	**4**	**16,61**	**1,22**	**4,05**
	Cost			4	1,09	0,04	0,13
(4)Lost Temple	**Gap**	top	62,23	**3**	**10,031**	**1,00**	**3,33**
	Cost			3	1,219	0,09	0,29
	Gap	right	58,14	**3**	**79,625**	**1,90**	**6,32**
	Cost			3	2,984	0,47	1,58
	Gap	left	59,82	**2**	**3,109**	**0,49**	**1,64**
	Cost			2	1,000	0,00	0,00
	Gap	bot	62,23	**3**	**4,109**	**0,61**	**2,04**
	Cost			3	1,203	0,08	0,27
(2)Binary Burghs	**Gap**	top	354,18	**5**	**453,109**	**2,66**	**8,82**
	Cost			5	16,375	1,21	4,03
	Gap	bot	273,64	**3**	**16,844**	**1,23**	**4,07**
	Cost			3	1,781	0,25	0,83
(3)Ice Mountain	**Gap**	bot left	72,47	**2**	**0,375**	**-0,43**	**-1,42**
	Cost				0,156	-0,81	-2,68
	Gap	top right	73,54	**2**	**1,91**	**0,28**	**0,93**
	Cost				0,359	-0,44	-1,48

(4)Nightmare Station	**Gap**	top left	193,49	**2**	**1,3**	**0,11**	**0,38**
	Cost			2	0,391	-0,41	-1,35
	Gap	top right	197,59	**2**	**6,578**	**0,82**	**2,72**
	Cost			2	0,797	-0,10	-0,33
	Gap	bot left	187,45	**2**	**8,01**	**0,90**	**3,00**
	Cost			2	0,859	-0,07	-0,22
	Gap	bot right	190,16	**2**	**6,89**	**0,84**	**2,78**
	Cost			2	0,859	-0,07	-0,22
(2)Challenger	**Gap**	top	283,52	**2**	**7,875**	**0,90**	**2,98**
	Cost			3	1,062	0,03	0,09
	Gap	bot	207,69	**2**	**17,625**	**1,25**	**4,14**
	Cost			2	0,922	-0,04	-0,12
(2)Space Madness	**Gap**	top	92,19	**3**	**1,047**	**0,02**	**0,07**
	Cost			3	0,359	-0,44	-1,48
	Gap	bot	96,166	**2**	**0,442**	**-0,35**	**-1,18**
	Cost			2	0,266	-0,58	-1,91

The number in parentheses in the beginning of the map name denotes the map size in player count. (4)Nightmare Station is a 4-player map where (2)Challenger is a 2-player map. Since the running times vary from 0.2 seconds to 177 seconds, to better fit the data into graph for a better readability, the logarithms of base 10 and base 2 are taken of the running times and are used in charts that are shown in Figure 13 and Figure 14.

A side to side comparison for both cost minimization and gap minimization criteria is given in table 4.

Table 4 - Cost minimization results are on the left, gap minimization readings are on the right.

Width	Time	Time Log10	Time Log2	Width	Time	Time Log10	Time Log2
58,14	2,984	0,47	1,58	58,14	**79,625**	**1,90**	**6,32**
59,82	1,000	0,00	0,00	59,82	**3,109**	**0,49**	**1,64**
62,23	1,219	0,09	0,29	62,23	**10,031**	**1,00**	**3,33**
62,23	1,203	0,08	0,27	62,23	**4,109**	**0,61**	**2,04**
72,47	0,156	-0,81	-2,68	72,47	**0,375**	**-0,43**	**-1,42**
73,54	0,359	-0,44	-1,48	73,54	**1,91**	**0,28**	**0,93**
92,19	0,359	-0,44	-1,48	92,19	**1,047**	**0,02**	**0,07**
96,166	0,266	-0,58	-1,91	96,166	**0,442**	**-0,35**	**-1,18**
131,06	0,77	-0,12	-0,38	131,06	**2,30**	**0,36**	**1,20**
135,20	1,09	0,04	0,13	135,20	**16,61**	**1,22**	**4,05**
186,76	0,81	-0,09	-0,30	186,76	**3,64**	**0,56**	**1,86**
187,45	0,859	-0,07	-0,22	187,45	**8,01**	**0,90**	**3,00**
190,16	0,859	-0,07	-0,22	190,16	**6,89**	**0,84**	**2,78**
193,49	0,391	-0,41	-1,35	193,49	**1,3**	**0,11**	**0,38**
197,59	0,797	-0,10	-0,33	197,59	**6,578**	**0,82**	**2,72**

207,69	0,922	-0,04	-0,12		207,69	**17,625**	**1,25**	**4,14**
273,64	1,781	0,25	0,83		273,64	**16,844**	**1,23**	**4,07**
283,52	1,062	0,03	0,09		283,52	**7,875**	**0,90**	**2,98**
283,75	5,88	0,77	2,55		283,75	**116,25**	**2,07**	**6,86**
354,18	16,375	1,21	4,03		354,18	**453,109**	**2,66**	**8,82**
Cost					Gap			

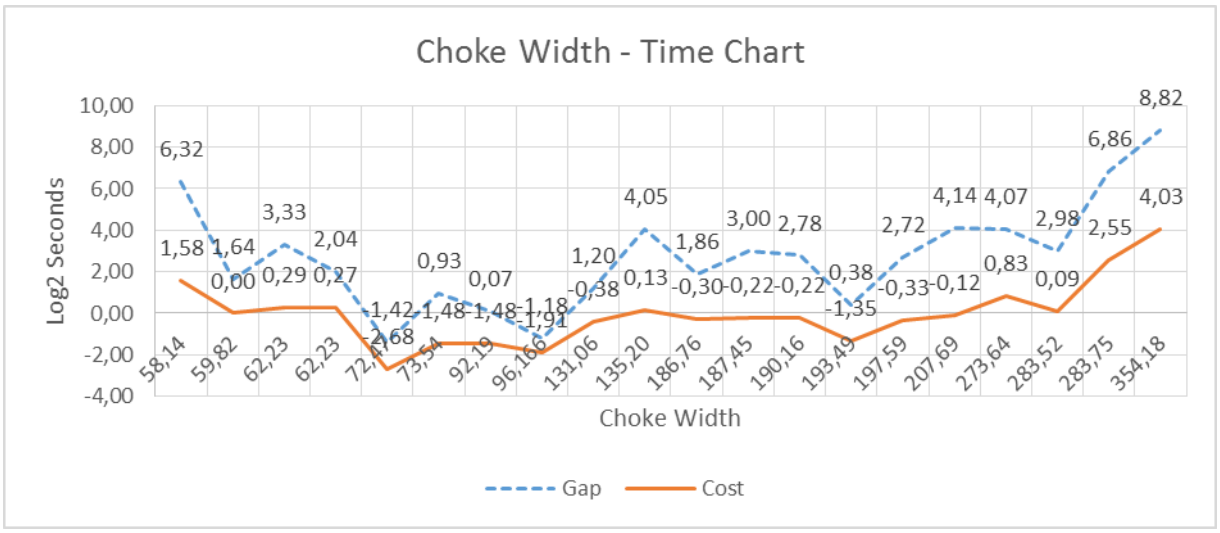

Figure 13 - Choke Width - Time Chart

It is clear from the chart in Figure 13 that choke width alone does not really determine the runtime of the program since there are more variables affecting the building configuration such as the number of available build tiles, the shape of the available build area, doodads (trees, occlusions, obstacles, etc.). Even though the width-time relation is mainly dependent on the map, it is safe to say that after ~135 px width of choke, it is not feasible to use gap minimization and after ~280 px width, it is not feasible to wall in since it will take longer than 4 seconds in each scenario.

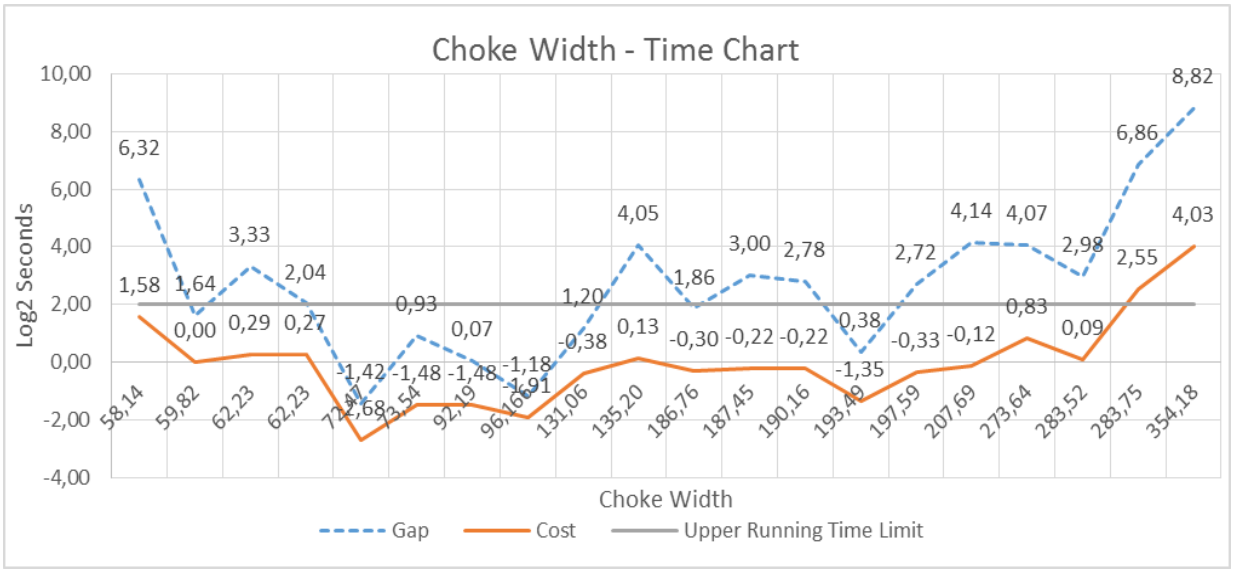

Figure 14 - Choke Width - Time chart with upper limit

Having the upper limit set as shown in Figure 14, it can be concluded that for choke points of width between 70px and 130px, it is feasible to use gap minimization while outside these boundaries up until 280px width it is feasible to use cost minimization.

With the current state of the problem formulation and algorithm implementation, there are some different outputs to the problem. Generally, the cost and gap minimization differ in only two buildings' layout as shown in Figures 15 and 16.

Figure 15 - Map: Astral Balance, location: top, optimization: cost

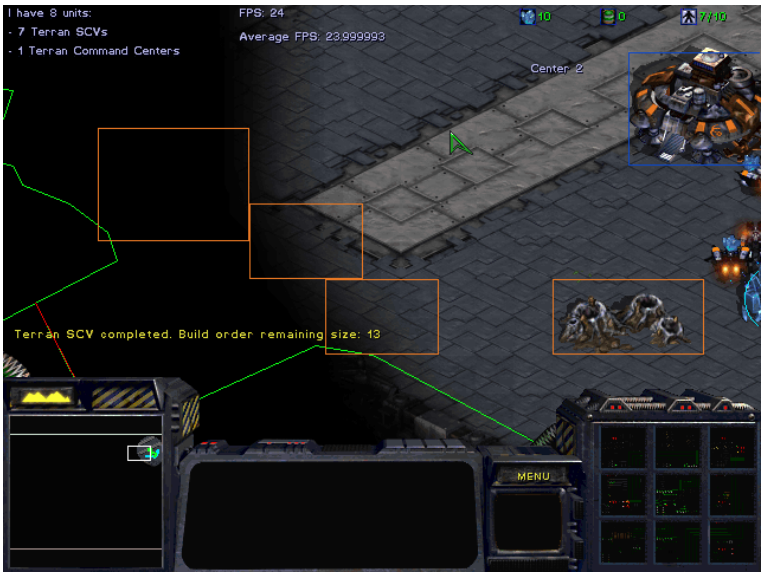

Figure 16 - Map: Astral Balance, location: top, optimization: gap

However, in some scenarios some terrain obstacles are included in the wall when gap is optimized, while in some others layout is preferred when cost is optimized as shown in Figures 17 and 18. Here the tree is used as the part of the wall as placing a building near a tree will not produce a gap value between the tiles.

Figure 17 - Map: Boxed In, location: bottom, optimization: cost. This solution doesn't include terrain obstacles.

Figure 18 - Map: Boxed In, location: bottom, optimization: cost. Gap minimization includes terrain obstacles as they prevent building gaps by having them placed apart.

In some maps where the walkable terrain is close to each other (imagine adjacent walk tiles) but having different height (meaning that a unit cannot walk from one tile to another), the logic program sometimes fails to produce an output or produce non-optimal results due to the walk tiles are considered ‘adjacent’ in the logic program. A non-optimal solution can be seen in Figure 19.

Figure 19 - Map: Binary Burghs, location: top, optimization: gap. The solver manages to find a solution, however the solution is not optimal as there are two adjacent walk tiles near the structure being built that appear to be walkable and adjacent in logic program.

6. CONCLUSION

The problem of building placement in StarCraft AI is solved by using logic programming (ASP) as the AI module calls the logic program when the map analysis is finished.

This study has shown that using two optimization criterion for the problem, either optimization mode has advantages over the other with respect to the width and shape of the choke point. The calculation time raises as the width increases, rendering some situations are non-practical. With the 4 second upper limit of calculation time, choke points with a width larger than 287px is non-practical for walling in for both optimization modes. The gap optimization mode is applicable for choke points with width between 60px and 130px. For the choke point width values larger than this interval takes longer than 4 seconds, thus are not applicable for a real scenario. For choke points with width values smaller than 287px and outside the gap minimization interval are suitable for cost minimization as it takes less than 4 seconds for the logic program to produce the correct optimal solution.

Although the logic program performs well for most of the scenarios, there exists a problem with the current simplified modeling of walk tiles, as seen in figure 19. Even though it is acceptable for the introductory level of gameplay, if one is willing to make an AI system acquire skills that of professional StarCraft players, this problem must be addressed and fixed. For non-optimal solutions as shown in Figure 19, additional constraints can be added. A constraint that checks the length of the path between adjacent tiles is a very good candidate for this situation where the logic program will contain additional data that contains the path information between tiles. By doing this, if two adjacent tiles have an unacceptable path length but they are not occupied by buildings (the scenario in Figure 19), then they will not be considered by the ‘canReach’ predicate that is described in Section 4.1.

The walls produced by ITUBot only include buildings and are applicable for full wall-ins. Wall-ins are not always fully closing the passage. In most of the time, one can see professional players using partial wall-ins to gain advantage in the battlefield when defending an aggression. Therefore, if one is to augment this introductory solution, she must consider units for wall-ins or modify the solution so as to partially wall in a large choke point.

The source code of ITUBot is open and can be found at the online repository on GitHub https://github.com/vilbeyli/. One can always download and modify the source code for contribution.

7. REFERENCES

[1] Churchill D., Saffidine A. and Buro M., (2012). *Fast Heuristic Search for RTS Game Combat Scenarios*. AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Eighth Artificial Intelligence and Interactive Digital Entertainment Conference

[2] Certicky, M., (2013). *Implementing a Wall-In Building Placement in StarCraft with Declarative Programming*

[3] Liquipedia, user-generated StarCraft wiki. Retrieved 2014 from http://wiki.teamliquid.net/starcraft/Walling

[4] StarCraft & StarCraft: Brood War Video Games. Retrieved 2013 from http://us.blizzard.com/en-us/games/sc/

[5] StarCraft: Brood War Application Programming Interface: BWAPI. Retrieved from http://code.google.com/p/bwapi/ and https://github.com/bwapi/bwapi

[6] Ontanon S., Synnaeve G., Uriarte A., Richoux F., Churchill D. and Preuss M., (2013). *A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft*. Computational Intelligence and AI in Games, IEEE Transactions on (Volume: 5, Issue: 4)

[7] Liquipedia, user-generated StarCraft wiki. Retrieved 2014 from http://wiki.teamliquid.net/starcraft/Micro_and_Macro

[8] UAlbertaBot, StarCraft autonomous game-playing agent, developed by David Churchill and his team. Retrieved 2013 from http://code.google.com/p/ualbertabot/

[9] Churchill D., Buro M., (2012). *Incorporating Search Algorithms into RTS Game Agents.* AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment; Eighth Artificial Intelligence and Interactive Digital Entertainment Conference

[10] Liquipedia, user-generated StarCraft wiki. *Terran Buildings Gaps.* Retrieved 2014 from http://wiki.teamliquid.net/starcraft/images/thumb/1/1c/Terran_buildings_gaps.jpg/509px-Terran_buildings_gaps.jpg

[11] Liquipedia, user-generated StarCraft wiki. *Walling as Terran.* Retrieved 2014 from http://wiki.teamliquid.net/starcraft/images/c/c7/Walling_As_Terran.png

[12] Churchill, D.; Buro, M.. *Build Order Optimization in StarCraft*. AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, North America, oct. 2011. Retrieved 2014 from https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4078

[13] Russel, S., Norvig, P. *Artificial Intelligence A Modern Approach Third Edition*. Prentice Hall, 2009.

[14] clasp Answer Set Programming solver developed by University of Potsdam. Retrieved 2013 from http://www.cs.uni-potsdam.de/clasp/