Walling in Strategy Games via Constraint Optimization

Florian Richoux, Alberto Uriarte, Santiago Ontañón

Walling in Strategy Games via Constraint Optimization

Florian Richoux
JFLI, CNRS / LINA
University of Tokyo / Université de Nantes
Tokyo, Japan / Nantes, France
florian.richoux@univ-nantes.fr

Alberto Uriarte and **Santiago Ontañón**
Computer Science Department
Drexel University
Philadelphia, PA, USA 19104
{albertouri,santi}@cs.drexel.edu

Abstract

This paper presents a constraint optimization approach to *walling* in real-time strategy (RTS) games. Walling is a specific type of spatial reasoning, typically employed by human expert players and not currently fully exploited in RTS game AI, consisting on finding configurations of buildings to completely or partially block paths. Our approach is based on local search, and is specifically designed for the real-time nature of RTS games. We present experiments in the context of the RTS game StarCraft showing promising results.

Introduction

This paper presents a constraint optimization approach to *walling* in real-time strategy (RTS) games. Walling is a specific type of spatial reasoning, typically employed in RTS games by human expert players, consisting on finding configurations of buildings to completely or partially block paths in order to block or slow-down enemy attacks.

Spatial reasoning is one of the open challenges in RTS game AI, and, except for a few approaches (such as the work of Perkins (Perkins 2010)) has not received enough attention in the literature (Ontanón et al. 2013). Spatial reasoning however, is key in RTS and other games where the action occurs in maps that are not predefined in advance, and where players need to exploit the spatial features of the different possible map configurations in which the game can occur. This paper presents one step towards that direction by addressing a specific spatial reasoning problem, and showing how a specific form of constraint optimization is particularly well suited to the real-time nature of RTS games.

Specifically, the approach to walling presented in this paper is based on local search, which does not ensure finding optimal solutions (not even actually finding a solution at all), but in practice produces high-quality solutions under real-time constraints. Our approach consists of three basic steps: 1) identifying a location where to perform walling, 2) specifying the parameters of the wall, and 3) using local search to actually determine the exact composition of the wall.

The remainder of this paper is organized as follows. First we provide some background on spatial reasoning. Then we present our local search approach. After that, we present an empirical evaluation of our method using maps from StarCraft, a RTS game that has emerged in the recent years as the standard testbed for RTS Game AI.

Spatial Reasoning in RTS Games

This section introduces real-time strategy (RTS) games and the problem of *walling*.

Real-Time Strategy Games

Real-time Strategy (RTS) is a sub-genre of strategy games where players need to build an economy (gathering resources and building a base) and military power (training units and researching technologies) in order to defeat their opponents (destroying their army and base). From a theoretical point of view, the main differences between RTS games and traditional board games such as Chess are that 1) they are *simultaneous move* games, 2) actions are *durative*, 3) RTS games are "real-time" (which actually means is that each player has a very small amount of time to decide the next move), 4) most RTS games are partially observable, and 5) the complexity of these games, both in terms of state space size and in terms of number of actions available at each decision cycle is very large (for example, the state space of StarCraft is at least 10^{1685} compared to 10^{50} for a game like Chess) (Ontanón et al. 2013). For those reasons, standard techniques used for playing classic board games, such as game tree search, cannot be directly applied to solve RTS games without the definition of some level of abstraction, or some other simplification. Interestingly enough, humans seem to be able to deal with the complexity of RTS games, and are still vastly superior to computers in these types of games (Buro and Churchill 2012).

The experiments presented in this paper are carried out in the RTS game *StarCraft: Brood War*, which is an immensely popular RTS game released in 1998 by Blizzard Entertainment, and which has emerged in the past few years as the standard testbed for RTS game AI.

Spatial Reasoning

The map used for RTS games is not fixed, and might change from game to game. Thus players need to spatially analyze the map to adequately maneuver military units, to determine where to place new buildings, among many other spatial decisions. This problem of analyzing the geometry of the map

in order to extract strategic information that can be used in subsequent decision making processes is called *spatial reasoning*, and has been identified as an open problem in RTS game AI (Buro 2003; Ontanón et al. 2013).

From a theoretical point of view, spatial reasoning has been studied in many fields, such as AI or cognitive science, where approaches such as spatial logics or qualitative spatial representations have been studied. The most influential approach to spatial logics is the RCC8 (*Region Connection Calculus with 8 relationships*) spatial logic (Randell, Cui, and Cohn 1992), which allows to reason about spatial regions and their relations by concepts analogous to those in Allen interval logic (Allen 1984), although basic calculus is NP-complete (Gerevini and Nebel 2002). Qualitative spatial representations typically use spatial logics for representation purposes, but do not use there axiomatic inference mechanisms, but qualitative inference, such as analogical or case-based reasoning (Forbus 2008).

In the specific area of RTS game AI, work on spatial reasoning in RTS games has mainly focused on three different problems: 1) map analysis: how to divide the map into a set of disjoint regions with certain properties of interest (mainly to detect *chokepoints* that are easy to defend), 2) unit maneuvering: how to spatially maneuver groups of units in combat in order to gain an advantage over the opponent, and 3) map generation. The best known work on map analysis is that of Perkins (Perkins 2010), who presented an algorithm to decompose a map into regions and chokepoints, implemented in the popular library BWTA, that is used in most StarCraft playing bots. An on-line terrain analysis gathering terrain knowledge through exploration is also proposed by (Si, Pisan, and Tien Tan 2014). Work on unit maneuvering is typically performed using potential fields or influence maps (Hagelback 2012; Uriarte and Ontañón 2012). Finally, different forms of spatial reasoning are performed by automatic map generating frameworks (Raúl Lara-Cabrera 2014; Togelius, Preuss, and Yannakakis 2010) in order to ensure some gameplay or aesthetic properties of the maps.

In this paper, we focus on a different spatial reasoning problem: *walling*, consisting on finding building configurations that completely or partially block a path. Walling is commonly employed by expert human players, and with the exception of (Certicky 2013), has not received much attention in the literature.

Problem Statement

A classical tactic in RTS to defend a base is to make a wall, that is, to construct buildings side by side in order to close or to narrow the base entrance. Closing a base gives the player extra-time to prepare a defense, or helps him to hide some pieces of information about his current strategy. Narrowing an entrance creates a bottleneck that is easier to defend in case of invasion. In this paper we will focus only in walls constructed by buildings, discarding small narrow passages that can be closed by small units like workers.

In StarCraft, the map space is defined by two grids: The *walk grid*, where each cell is an 8×8 pixels square, and the *build grid*, where each cell is a 4×4 walk tile square (*i.e.*, of 32×32 square of pixels). Each cell in the build grid is called a *build tile*. Moreover some build tiles are *buildable* and some are *not-buildable*. The approach presented in this paper is applicable for any RTS game for which such a *build grid* exists.

We will define two properties of buildings: their *build size*, and their *real size*. The build size is a pair (w, h) of build tiles. In order to create such a building, we need a rectangle of buildable tiles in the map (w build tiles in width and h build tiles in height). The real size is a pair (w_p, h_p), such that $w_p \leq 32 \times w$ and $h_p \leq 32 \times h$, representing the actual size of the building in pixels once it's constructed in the game, where 32 is the size in pixels of a build tile in StarCraft (but might be different for other RTS games). The real size of a building can then be smaller than its build size. This is actually always the case in StarCraft.

This means that two buildings constructed side by side are still separated by a gap which may be big enough to let small units enter, like Zerglings, Marines or Zealots in StarCraft.

The walling problem is an optimization problem described as follows: Given a chokepoint and two buildable tiles s and t (start and target tiles), choose a set of buildings B and place them on the map such that:

- All buildings of the set are part of a wall (i.e. they are contiguous), and the tiles s and t are covered by buildings.
- A target optimization function f about the wall may be minimized. In this paper, we will focus on three different optimization functions:
  - The number of buildings in the wall.
  - The number of gaps between buildings which are big enough to let small units enter.
  - The required technology level in the game (in games like StarCraft, some buildings require technologies that require resources to acquire, so it is interesting to build walls with buildings of low technology).

Walling via Constraint Optimization

The approach to walling presented in this paper consists of three main stages: *chokepoint identification*, *wall specification*, and *wall creation*, which correspond, respectively to: identifying a place in the map where the wall can be created, determining the exact coordinates where we want to generate the wall, and finally determining which exact buildings do we need and in which coordinates. The following subsections describe each of these stages in turn.

Chokepoint Identification

For the chokepoint identification we rely on Perkins algorithm for map decomposition (Perkins 2010) implemented in the BWTA library. This algorithm decomposes a map into regions and chokepoints that connect exactly two regions. It first computes *obstacle polygons* (representing each of the non-walkable regions). Then, using the edges of these polygons, a Voronoi diagram of the line segments is generated. Then after pruning some vertices of the resultant Voronoi diagram, the algorithm looks for the vertices with degree two and with a small distance to the nearest obstacle polygon.

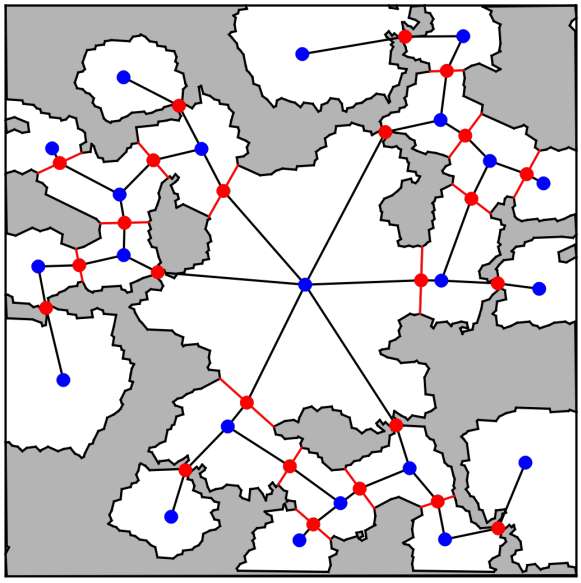

Figure 1: Result of decomposing a map in regions (blue dots) and chokepoints (red dots) using BWTA

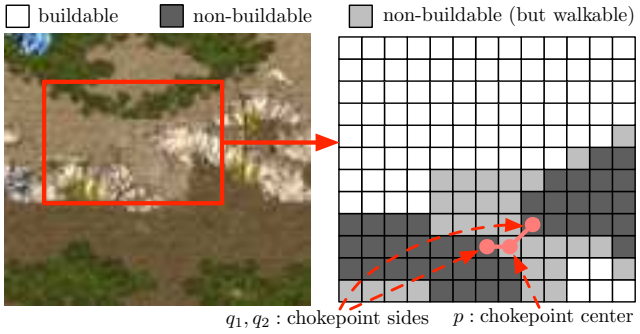

Figure 2: Mapping of a portion of a StarCraft map to a grid of build tiles.

Those vertices are marked as chokepoints. The result of decomposing a map in regions and chokepoints using BWTA's implementation of the algorithm is shown in Figure 1.

Although the algorithm used in BWTA is the standard algorithm used in most work using StarCraft, the algorithm has some false positive and false negative results (i.e. it marks as chokepoints some areas that would not be considered as chokepoints by a human, and vice versa). In order to mitigate these errors we filtered the chokepoints that are too small (smaller than 3 build tiles) or too big (larger than 12 build tiles) to build a wall or the ones in which building a wall does not make sense (where there are no buildable cells around the chokepoint)[1].

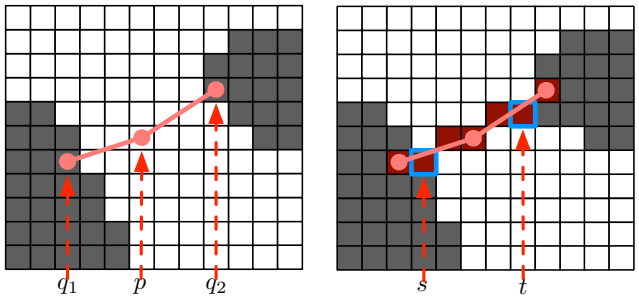

Figure 3: Determining the start (s) and target (t) coordinates of a wall when the chokepoint is buildable.

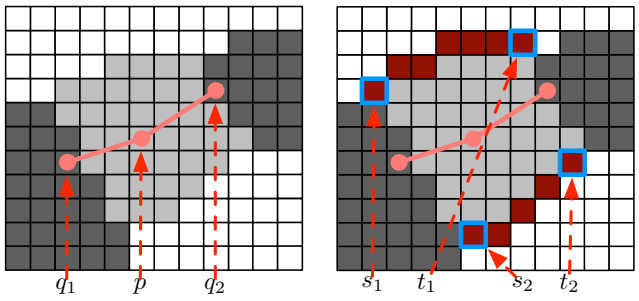

Figure 4: Determining the start and target coordinates of a wall when the chokepoint is non-buildable. There are two possible walls in this case s_1 to t_1 and s_2 to t_2.

Wall Specification

In the second stage, given a chokepoint $\langle p, q_1, q_2 \rangle$, where p are the coordinates of the chokepoint, and q_1 and q_2 are the coordinates of the sides of the chokepoint (i.e., the coordinates of the two walls in each side of the chokepoint, as illustrated in Figure 2), we want to determine the specific points s and t where the wall should start and end. We distinguish three main situations:

- **Buildable Chokepoint:** When all the cells in the grid in the segments that connect q_1 to p and p to q_2 (except for q_1 and q_2) are buildable. This is the simplest scenario, and s and t can be determined using the following procedure (illustrated in Figure 3):
  1. Using Bresenham's line drawing algorithm (Bresenham 1965), we trace a line from q_1 to p. The first point in this line that is buildable is labeled as s.
  2. Using Bresenham's line drawing algorithm (Bresenham 1965), we trace a line from q_2 to p. The first point in this line that is buildable is labeled as t.
- **Non-Buildable Chokepoint:** When all the cells in the grid in the segments that connect q_1 to p and p to q_2 (except for q_1 and q_2) are not-buildable. In this case, the choke point is walkable, but not-buildable. Therefore, the

[1]Our modified version of BWTA, that analyze maps offline, can be downloaded from: `https://bitbucket.org/auriarte/bwta2`

wall cannot be built in the chokepoint itself. s and t can be determined as follows (Figure 4):

1. We run a *flood-fill* algorithm starting from point p to determine the set of connected (using 4-connectivity) cells that are not-buildable, but walkable (e.g. the set of grey coordinates in Figure 4). Let us call this set R.
2. We find the set of buildable cells in the map that have at least one neighbor (4-connectivity) in R and at least one neighbor outside of R (the set of dark red cells in Figure 4). Let us call this set B (border).
3. We split the set B into disjoint sets $B_1, ..., B_n$, such that all the cells in B_i are neighbors (8-connectivity) and no cell in B_i is a neighbor of any cell in B_j if $i \neq j$ (there are two such sets of cells in Figure 4).
4. For each set B_i, determine the two extremes s_i and t_i, such that s_i and t_i are the pair of points from B_i that are further from each other that satisfy that they both have at least one neighbor (8-connectivity) that is not-buildable and not-walkable. Each of these pairs s_i, t_i determines the start and target points of a possible wall (two such walls are found in Figure 4).

Notice that it is theoretically possible to have $n > 2$, but in this case more than one wall would be potentially needed to block the chokepoint. In this situation, we just consider the chokepoint as not a candidate for blocking with a wall.

- **Mixed Chokepoint:** Although not common in StarCraft maps, chokepoints with a mix of buildable and non-buildable cells in the segments that connect q_1 to p and p to q_2 sometimes occur. However, we do not consider this scenario in our approach.

Given the start and target coordinates of a wall: s, t, we determine whether the wall is feasible by computing the distance between s and t. If s and t are too far apart, constructing the wall might not bring any significant advantage, and thus it should not be attempted. On the other hand, if s and t are too close, there might not be space for placing buildings, and the wall might not be possible. For the experiments presented in this paper, we only considered those situations where the following conditions where satisfied: $4 < |t_x - s_x| \leq 12$ and $3 < |t_y - s_y| \leq 9$.

Wall Creation

We express walling as a Constraint Optimization Problem (COP). A COP is a tuple (V, D, C, f) where V is a set of k variables, D the set of domains of each variable, *i.e.*, sets of values that variables can take, C is the set of constraints upon V and $f\colon V^k \rightarrow \mathbb{R}$ is a k-ary objective function to minimize or maximize. A *configuration* is a mapping of each variable in V to a value in their domain in D. A configuration may then satisfy none, some or all constraints of the COP instance. In this latter case, the configuration is called a *solution*.

In (Certicky 2013), a COP is presented to model the walling problem, where V is the set of buildings that may compose the wall, D the set of possible positions for each building of V. Two objective functions f_v and f_h are present to respectively minimize the vertical size and the horizontal size of the largest gaps between buildings. This leads to a complex multi-objective optimization problem, where goals do not always prevent foes to go through the wall, but where an imperfect wall is considered to be better if its largest gap can let pass only one zergling rather than two side by side.

In this paper, we propose a different approach: V and D remain respectively the set of buildings we can use to build the wall and the set of possible positions for these variables, but we propose a different set of constraints C:

1 *Overlap*: buildings should not overlap each others.

2 *Buildable*: buildings can be built on their positions.

3 *NoHoles*: there should not be any holes of the size of a build tile (or greater) in the wall.

4 *StartingTargetTile*: there should have exactly one building constructed on the starting tile s, and one building (could be the same) on the target tile t.

Let's denote by t_{ij} the tile at position (i, j) on the map, with $1 \leq i \leq length(map)$ and $1 \leq j \leq height(map)$. The special position $(0, 0)$ in D indicates that buildings assigned to this position have not been selected to build the wall. Let *isBuildable(t_{ij})* be the predicate returning true if and only if the tile t_{ij} is buildable. We denote also by t_s and t_t the start and target tile, respectively. Thus, we formally define our constraints as follows:

Overlap:

$$\begin{array}{l}\forall b_1, b_2 \in V, \\ \forall 1 \leq i \leq length(b_1), \\ \forall 1 \leq j \leq height(b_1), \\ \forall 1 \leq i' \leq length(b_2), \\ \forall 1 \leq j' \leq height(b_2), \\ \exists t^1, t^2 \in D \; s.t. \\ \big(b_1 \neq (0,0) \wedge b_2 \neq (0,0)\big) \Leftrightarrow \big(t^1_{11} = b_1 \wedge t^2_{11} = b_2 \wedge t^1_{ij} \neq t^2_{i'j'}\big)\end{array}$$

Buildable:

$$\begin{array}{l}\forall b \in V, \\ \forall 1 \leq i \leq length(b), \\ \forall 1 \leq j \leq height(b), \\ \exists t \in D \; s.t. \; (t_{11} = b) \Leftrightarrow isBuildable(t_{ij})\end{array}$$

For the following constraint, let's denote by $b.x$ and $b.y$ respectively the x and y coordinates of a building b, and by l and h respectively the functions $length$ and $height$.

NoHoles:

$$\exists b_s, b_t \in V, \Big[b_s = b_t \Leftrightarrow$$
$$\Big(\exists b'_s, b'_t \in V,\ b_s \neq b'_s,\ b_t \neq b'_t,\ s.t.$$
$$\big(b_s.x + h(b_s) + 1 = b'_s.x \vee b_s.x = b'_s.x + h(b'_s) + 1 \vee$$
$$b_s.y + l(b_s) + 1 = b'_s.y \vee b_s.y = b'_s.y + l(b'_s) + 1\big)$$
$$\wedge$$
$$\big(b_t.x + h(b_t) + 1 = b'_t.x \vee b_t.x = b'_t.x + h(b'_t) + 1 \vee$$
$$b_t.y + l(b_t) + 1 = b'_t.y \vee b_t.y = b'_t.y + l(b'_t) + 1\big)\Big)\Big]$$
$$\wedge$$
$$\Big[\forall b \in V,\ b_s \neq b \neq b_t,\ b \neq (0,0),$$
$$\exists b_a, b_b \in V, b_a \neq b \neq b_b,\ s.t.$$
$$\big((b.y = b_a.y + l(b_a) + 1) \wedge (b.x = b_b.x + h(b_b) + 1)\big) \vee$$
$$\big((b.x = b_a.x + h(b_a) + 1) \wedge (b.x + h(b) + 1 = b_b.x)\big) \vee$$
$$\big((b.x = b_a.x + h(b_a) + 1) \wedge (b.y + l(b) + 1 = b_b.y)\big) \vee$$
$$\big((b.y = b_a.y + l(b_a) + 1) \wedge (b.x + h(b) + 1 = b_b.x)\big) \vee$$
$$\big((b.y = b_a.y + l(b_a) + 1) \wedge (b.y + l(b) + 1 = b_b.y)\big) \vee$$
$$\big((b.x + h(b) + 1 = b_a.x) \wedge (b.y + l(b) + 1 = b_b.y)\big)\Big]$$

StartingTargetTile:

$$\exists b_1, b_2 \in V,$$
$$\exists 1 \leq i \leq length(b_1),$$
$$\exists 1 \leq j \leq height(b_1),$$
$$\exists 1 \leq i' \leq length(b_2),$$
$$\exists 1 \leq j' \leq height(b_2),$$
$$\exists t^1, t^2 \in D\ s.t.$$
$$\big(t^1_{11} = b_1 \wedge t^2_{11} = b_2\big) \Leftrightarrow \big(t^1_{ij} = t_s \wedge t^2_{i'j'} = t_t\big)$$

Some RTS games might require additional constraints. For example, if using the Protoss race in StarCraft, one should add a fifth constraint to consider *Pylons*.

We propose to use a single function f to minimize chosen among the three presented above. The most complex and interesting one is the objective function trying to reduce the number of gaps in the wall that are big enough to let small units to pass. This function can be adapted according to the opponent race. Indeed, the smallest units in the game are Zerg's Zergling (16 pixels wide $\times$16 pixels high), Terran's Ghost (15×22) and Terran's Marine and Medic (17×20), and Protoss' Zealot (23×19), if we don't consider Protoss' Scarab (Reaver's projectiles, 5×5 pixels). In this paper, experiments are reported assuming a Zerg opponent.

There are few works in RTS game AI using constraint programming techniques, and even fewer using metaheuristics. Among others, branch and bound algorithms (not a metaheuristic) have been used to optimize build order (Churchill and Buro 2011). Genetic algorithms have been used off-line to optimize build order, but with multiple objectives, analyzed in (Kuchem, Preuss, and Rudolph 2013) and a population-based algorithm has been used for multi-objective procedural aesthetic map generation (Raúl Lara-Cabrera 2014). To solve our COP instance, again we have chosen a different technique than in (Certicky 2013). In that previous work, an ASP logic program has been written and is solved by the ASP solver Clingo. Even if it allows finding an optimal solution, time computations may not fit real-time games like RTS, since their formulation requires up to 200ms per ASP solver call. In contrast, in the rules of the annual AIIDE StarCraft competition, it is clearly specified that a bot where the time computation during a frame exceeding 55ms more than 200 times loses automatically the game. In order to have faster optimization, we opted for a local search method based on the Adaptive Search algorithm (Codognet and Diaz 2001; Caniou et al. 2014), which is up to our knowledge one of the fastest metaheuristic to solve constraint-based problems. Even if walling might only be used a few times during a game, aiming at fast computations is important since our approach can be used for many other tasks in RTS games, such as base layout. Thus, one can run our solver to make a wall at a base entrance, but also to manage building placements into the base.

The main idea of the Adaptive Search algorithm is the following one: a cost function is declared for each kind of constraint in the COP telling how much a constraint is far to be solved within the current configuration. The output of such a cost function is a *constraint cost*. If the cost of a constraint c is zero, it means c is currently satisfied. One can then give a global cost to a configuration, usually by adding the cost of each constraint in the COP instance. The originality of Adaptive Search is that it projects constraints cost on variables, *i.e.*, it sums the cost of all constraints where a given variable occurs. For instance, if a variable x appears in constraints c_1 and c_2 only, then the projected cost on x will be the cost of c_1 plus the cost of c_2. This allow the algorithm to know what variables are the most responsible for the violation of constraints, and then permit to apply a sharper variable selection heuristic.

The main problem with metaheuristics is that these methods can be trapped into a local minimum, *i.e.*, a non-optimal configuration where there are any local moves leading to a better configuration. To escape from these situations, one classical and efficient method is to simply restart the algorithm from a random-selected configuration. Thus, our COP solver has two different behaviors:

Satisfaction: The user only asks for a wall, without requiring optimization on any of the three objective functions presented above. In that case the solver tries to find a wall satisfying our four constraints within one run limited by a timeout (for instance 20 ms).

Optimization: The user only asks for an optimized wall, for instance trying to have as few large gaps as possible. Then, the solver will launch a series of runs limited by the given timeout (like 20ms), and saves what is the best solution found so far, according to the objective function. It stops when: 1) It finds a perfect solution *i.e.*, an optimization cost equal to zero (which is not possible for some objective functions like minimizing the number of buildings); or 2) It reaches a global timeout, for example 150ms. Launching a series of small runs is important because it means we can slice a 150ms optimization run into several 20ms independent pieces that can be executed in different frames; so that an 150ms optimization run do not exceed the 55ms limit per frame during competitions.

Table 1: Overall Experimental Results over 48 different problems, extracted from 7 different maps from the StarCraft AI competition. Results are the average of 100 runs in each problem (total of 4800 runs per configuration).

#Attempts	1	2	3	4	5	10	20	50
Average Cost	1.59	0.58	0.33	0.18	0.13	0.03	0.01	0.0006
Solved %	45.83%	71.10%	80.70%	88.45%	91.58%	97.23%	99.18%	99.95%

Table 2: Optimization results over 48 different problems, extracted form 7 different maps from the StarCraft AI competition. Results are the average of 100 runs in each problem (total of 4800 runs per configuration).

	Satisfaction run	*Optimization run*	*Optimization run solved*
Optimizing number of buildings (number of buildings)	3.12	2.65	96.83%
Optimizing gaps (number of gaps)	1.19	0.05	96.79%
Optimizing tech-level (average tech level of buildings)	1.95	1.56	95.87%

Experimental Evaluation

This section presents an empirical evaluation of our approach in the RTS game StarCraft. In this paper, we only considered Terran buildings without their extensions. The main goals of our evaluation are: 1) to determine the quality of solutions that can be achieved under the tight timing constraints in RTS games, 2) to determine how much can optimization improve the base solutions provided by a satisfaction run.

In order to evaluate our approach we used seven maps from the StarCraft AI competition: *Benzene*, *Aztec*, *Circuit Breaker*, *Python*, *Heartbreak Ridge*, *Andromeda*, and *Fortress*. From each of those maps, we extracted a collection of chokepoints and employed the methods described in this paper to determine the start and target coordinates of the walls. In total this resulted in 48 wall specifications. All the experiments were executed on a PC with a 2.7GHz Intel Core i7 and 4GB of RAM, running Ubuntu 12.04 64-bit[2].

Table 1 shows the results obtained using satisfaction runs in the 48 chokepoints using a timeout of 20ms. Specifically, we show both the average solution cost (0 means that a wall was found) and the percentage of times that a wall was found while building walls for the 48 chokepoints. The *Average Cost* indicates how far we are from a solution. Thus, the evolution of this value in Table 1 is more meaningful than the values themselves. The presented results are the average of 100 runs. Each column in Table 1 shows the results when we allow our system to restart a different number of times. The first column shows results when the solver only has one attempt (with a timeout of 20ms), solving 45.83% of the problems. The second column shows results when our solver has two attempts (each time with a timeout of 20ms), and showing that 71.1% of the times at least one of the attempts resulted in a solution. As Table 1 shows, when giving our solver at least 5 attempts, more than 90% of the problems were solved. This is remarkable, since, due to the small timeout used, it means that our solver can run once per game frame. Thus, after 5 game frames (*i.e.* 0.21 seconds of gameplay) the probability of having found a solution to a walling problem is very high. With a higher number of attempts the probability of success stabilizes at 99.95%.

Table 2 shows the results for optimization runs using the three different optimization objectives considered in this paper. During these optimization runs, the solver launched a series of successive runs of 20ms, with a global timeout of 150ms. For each configuration we compare the walls generated using a satisfaction run versus the walls obtained using an optimization run. Satisfaction runs have been launched allowing 8 attempts, so the total computation time is about 160ms, a bit more than for optimization runs. Considering optimizing the number of buildings, we can see that our solver was able to reduce the average number of buildings from 3.12 to 2.65. This means that using optimization we can reduce the number of buildings to construct, and thus save a significant amount of resources (gas and minerals). Similarly, gap optimization can reduce the number of gaps big enough to let pass a zergling from 1.19 per wall, to 0.05 per wall, thus significantly making the walls more difficult to penetrate by the enemy. Concretely, satisfaction runs found 929 perfect walls (without gaps letting small units pass) among 4609 walls (over 4800 runs), and optimization runs found 4440 perfect walls among 4646 walls (again over 4800 runs). For tech-level optimization, we classified buildings along 4 different tech-levels (command centers and supply depots have tech-level 0, and all the other buildings have a higher tech-level depending on their requirements). Optimization managed to reduce the average tech level of the buildings in the wall from 1.95 to 1.56, meaning that the wall has less restrictions, and thus, can be built earlier.

Conclusions

This paper has presented a constraint optimization approach to *walling* in real-time strategy (RTS) games. Walling constitutes a specific case of the general problem of spatial reasoning in RTS games, which has not received enough attention in the literature. Our solution is based on *Adaptive Search*, and can be configured to generate walls optimizing different criteria, such as the number of buildings, number of gaps, or tech-level of the buildings used.

As shown in our empirical evaluation, our approach is well suited for the real-time nature of RTS games, since it can generate solutions under tight real-time constraints, and computation can be easily spread across several game

[2]Source code can be downloaded from: `https://github.com/richoux/Wall-in`

frames, to improve the probability of finding a solution.

As part of our future work, we would like to improve our approach to allow for more flexible wall definition. For example, instead of specifying concrete starting and target positions for the wall, we would like to provide a start and target broad areas, giving the solver more freedom for finding solutions in tight space configurations. Additionally, we would like to include additional optimization constraints, such as minimizing the wall cost, or minimizing the amount of time that would require to build the wall. We would also like to optimize the solver allowing parallelism following the parallel scheme proposed in (Caniou et al. 2014), and incorporate reinforcement learning approaches to tune the solver's parameters. We would also like to package the current solver into a reusable module for use in StarCraft AI competition bots. Finally, we are currently exploring other spatial reasoning problems that could be addressed using extensions of our current approach.

References

Allen, J. F. 1984. Towards a general theory of action and time. *Artificial intelligence* 23(2):123–154.

Bresenham, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems journal* 4(1):25–30.

Buro, M., and Churchill, D. 2012. Real-time strategy game competitions. *AI Magazine* 33(3):106–108.

Buro, M. 2003. Real-time strategy games: a new ai research challenge. In *Proceedings of IJCAI 2003*, 1534–1535. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Caniou, Y.; Codognet, P.; Richoux, F.; Diaz, D.; and Abreu, S. 2014. Large-scale parallelism for constraint-based local search: The costas array case study. *Constraints* 19(4):1–27.

Certicky, M. 2013. Implementing a wall-in building placement in starcraft with declarative programming. *CoRR* abs/1306.4460.

Churchill, D., and Buro, M. 2011. Build order optimization in starcraft. *Proceedings of AIIDE* 14–19.

Codognet, P., and Diaz, D. 2001. Yet another local search method for constraint solving. In *proceedings of SAGA'01*, 73–90. Springer Verlag.

Forbus, K. 2008. Qualitative modeling. In Van Harmelen, F.; Lifschitz, V.; and Porter, B., eds., *Handbook of knowledge representation*. Elsevier.

Gerevini, A., and Nebel, B. 2002. Qualitative spatio-temporal reasoning with rcc-8 and allen's interval calculus: Computational complexity. In *ECAI*, volume 2, 312–316.

Hagelback, J. 2012. Potential-field based navigation in starcraft. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 388–393. IEEE.

Kuchem, M.; Preuss, M.; and Rudolph, G. 2013. Multi-objective assessment of pre-optimized build orders exemplified for starcraft 2. In *Computational Intelligence and Games (CIG)*.

Ontanón, 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. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)* 5(4):1–19.

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *AIIDE*, 168–173.

Randell, D. A.; Cui, Z.; and Cohn, A. G. 1992. A spatial logic based on regions and connection. *KR* 92:165–176.

Raúl Lara-Cabrera, Carlos Cotta, A. J. F.-L. 2014. A self-adaptive evolutionary approach to the evolution of aesthetic maps for a RTS game. In *IEEE World Congress on Computational Intelligence (WCCI)*.

Si, C.; Pisan, Y.; and Tien Tan, C. 2014. Automated terrain analysis in real-time strategy games. In *Proceedings of the 9th International Conference on the Foundations of Digital Games (FGD 2014)*.

Togelius, J.; Preuss, M.; and Yannakakis, G. N. 2010. Towards multiobjective procedural map generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 3. ACM.

Uriarte, A., and Ontañón, S. 2012. Kiting in RTS games using influence maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.