GHOST: A Combinatorial Optimization Solver for RTS-related Problems

Florian Richoux[*], Alberto Uriarte[†], Jean-François Baffier[‡]

Abstract

This paper presents GHOST, a combinatorial optimization solver an RTS AI developer can use as a blackbox to solve any problems encoded by a constraint satisfaction/optimization problem. We show a way to model three very different RTS problems by a constraint satisfaction/optimization problem, each problem belonging to a specific level of abstraction, and test our solver on them, using StarCraft as a testbed. For the three problems (the target selection problem, the wall-ion problem and the build order planning problem), GHOST shows very good results computed within some tens of milliseconds.

Game AI, Real-Time Strategy, StarCraft, CSP, COP, Solver, Optimization, Combinatorics.

1 Introduction

One can see games as a simplification of the world: The playground is smaller, rules are easier and less numerous, thus possibilities are more limited. However, games are rich enough to propose complex, dynamic environments where it remains difficult for a computer to make predictions, have a global understanding of the current situation and then take a decision. This is especially true when information is incomplete, forcing the computer to infer the global state of the game from pieces of information. This is the case with Real-Time Strategy games (RTS), where a Clausewitz's fog of war hides the opponent's moves. Thus, RTS games are very suitable to design Artificial Intelligence techniques that could be applied afterward in other domains.

Many AI techniques are used in RTS games. Without making an exhaustive list, we can cite reinforcement learning, case-base reasoning, Bayesian programming, goal-driven autonomy, potential fields, etc. We recommend the reader to look at surveys from Ontañón et al. [1] and Robertson and Watson [2] to have a very complete overview of AI techniques in RTS games.

However, there are few works in RTS game AI using constraint programming techniques, in particular through constraint satisfaction/optimization problem (CSP/COP) models. Among others, branch and bound algorithms have been used to optimize build order in Churchill and Buro [3]. Genetic algorithms have been used off-line to optimize build order, with multiple objectives, and have been analyzed in Kuchem et al. [4] and a population-based algorithm has been used for multi-objective procedural aesthetic map generation in Lara-Cabrera et al. [5]. Still, CSP/COP offers a convenient, homogeneous framework able to model a huge number of combinatorial and optimization problems, and proposes a various set of algorithms to solve them.

CSP/COP is widely used in Artificial Intelligence to solve problems like path-finding, scheduling, and logistics. Unlike mathematical programming, CSP/COP algorithms are usually not designed to solve one specific problem but are general, able to manage any problems modeled in that framework. Besides the generality, it is also easy and intuitive to model a problem with a CSP/COP. Altogether, these bring ideal conditions to design and develop a user-friendly, easy-to-extend, general solver.

[*]LINA, Université de Nantes, France, and JFLI, University of Tokyo, Japan. florian.richoux@univ-nantes.fr
[†]Computer Science Department at Drexel University, Philadelphia, PA, USA. au49@drexel.edu
[‡]Jean-François Baffier Department of Computing Science of the University of Tokyo, Japan. jf_baffier@nii.ac.jp

1.1 RTS problem families

Ontañón et al. propose in [1] to decompose RTS problems into three families, according to their level of abstraction. From the higher to the lower level, these families are (extracted from [1]):

- **Strategy**: corresponds to the high-level decision making process. This is the highest level of abstraction for the game comprehension. Finding an efficient strategy or counter-strategy against a given opponent is key in RTS games. It concerns the whole set of units and buildings a player owns.
- **Tactics**: are the implementation of the current strategy. It implies army and building positioning, movements, timing, and so on. Tactics concerns a group of units.
- **Reactive control**: is the implementation of tactics. This consists in moving, targeting, firing, fleeing, hit-and-run techniques (also knows as "kiting") during battle. Reactive control focuses on a specific unit.

Problems from different families usually involve different algorithms to solve them. In this paper, we will model one problem for each of these three family, and use our solver GHOST to solve them, without any modifications of the solver core. Only problem models will, of course, differ.

1.2 StarCraft: Brood War

StarCraft: Brood War is the extension of the game StarCraft, both released in 1998 by Blizzard Entertainment. It is an RTS game where three very different races (Terran, Protoss and Zerg) can be played, giving an asymmetric but perfectly well balanced strategy game. In this paper, the term "StarCraft" will actually refer to the game plus its extension StarCraft: Brood War.

A player has to gather two kind of resources, mineral and gas, in order to construct buildings for producing units, upgrading characteristics such as damage points or armor, unlocking technologies and special abilities. Producing units has also a cost, each unit type having their own characteristic (hit points, damage points, size, etc) and a price in consequence.

The game is run on a rectangular map, discretized into two kind of tiles: Walk tiles, composed of 8×8 pixels and build tiles composed of 4×4 walk tiles, *i.e.*, 32×32 pixels. The difference between these two kinds of tiles would be explained in Section 4, while introducing the wall-in problem. Like every RTS games, the map is recovered by a fog of war, only dissipated around our own units/buildings. This means that one cannot know at any time the full state of the game. Also, the player has to deal with a supply capacity, limiting the number of units he can have. The player can increase it by constructing the right building or producing the right unit, according to the race.

StarCraft has been a world wide success, and is still the most sold RTS game of the history with 11 millions of copies distributed. It has been and still is particularly popular in South Korea, where a StarCraft league has been specially created for the game, organizing televised tournaments between sponsored players and teams. Korean professional players, or pro-gamers, are reputed to be among the best StarCraft pro-gamers in the world.

Time's flow in StarCraft is tricky: The game has 7 speed modes from "slowest" to "fastest" where "normal" corresponds to the regular time's flow (one second is one second) and the 6 others disturbing it. This is why when we will write about in-game time, all along this paper we will refer to the terms *StarCraft unit time* rather than second, to not make any confusions with proper seconds. In the fastest mode, one game frame takes 42ms. In StarCraft AI competitions, it is necessary for bots to not make computations longer than 55ms per frame, in which case the bot automatically looses the game after 200 frames exceeding 55ms.

1.3 Goals and summary

The research presented in this paper is an extension of Richoux et al. [6], where the problem to make a wall at a given chokepoint in StarCraft has been modeled and solved thanks to an ad-hoc algorithm.

The goal of this paper is double:

- To present to the RTS AI community our solver GHOST, its architecture, how to model different problems with it and the results we obtained.
- To show that constraint programming can be successfully used in RTS AI, at any level of abstraction.

The paper is organized as follows: In Section 2, we give a short introduction or reminder on constraint satisfaction/optimization problems, how they model problems and what kind of algorithms exists to solve them. Then, we present GHOST architecture, what user is aimed, how to use it to solve an already encoded problem and how to write your own problems via GHOST. Sections 3, 4 and 5 give details about three different problem, each from a specific level of abstraction (from the lower to the higher: Reactive Control, Tactic and Strategy), how we modeled them into a CSP/COP and what results we obtained by applying GHOST. Finally, this paper concludes with future works.

2 GHOST: A General meta-Heuristic Optimization Solving Tool

2.1 A brief introduction to CSP/ COP

Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP) are two close formalisms intensively used in Artificial Intelligence to solve combinatorial and optimization problems. Constraint Programming allows an intuitive and uniform way to model problems, as well as different general algorithms able to solve any problems modeled by a CSP or a COP.

The difference between a CSP and a COP is simple:

- A CSP models a satisfaction problem, *i.e.*, a problem where all solutions are equivalent; the goal is then to just find one of them, if any. For instance: finding a solution of a Sudoku grid. Several solutions may exist, but finding one is sufficient, and no solutions seem better than another one.
- A COP models an optimization problem, where some solutions are better than others. For instance: Several paths may exist from home to workplace, but some of them are shorter.

Formally, a **CSP** is defined by a tuple (V, D, C) such that:

- V is a set of variables,
- D is a domain, *i.e.*, a set of values for variables in V,
- C is a set of constraints.

A constraint $c \in C$ can be seen as a k-ary predicate $c : V^k \rightarrow \{\top, \bot\}$ where $\top$ can be semantically interpreted by true and $\bot$ by false. Thus, regarding the value of the vector V^k, we say that c is either *satisfied* (equals to $\top$) or *unsatisfied* (equals to $\bot$).

Notice also that D should formally be the set of the domain for each variable in V, thus a set of set of values. However it is common to define the same set of values for all variables of V, thus one can simplify D to be the set of values each variable in V can take.

A **COP** a defined by a tuple (V, D, C, f) where V, D and C represent the same sets as a CSP, and f is an *objective function* to minimize or maximize.

Upon a CSP or a COP, one can build a CSP formula or a COP formula, respectively. A CSP/COP formula is a conjunction of constraints from C, each constraint taking their variables from V. We say a CSP/COP formula is satisfied if there exists an assignment $q : V \rightarrow D$ such that each constraint of the formula is satisfied. Considering the vector $v \in V^n$ of variables used in a CSP/COP formula, a vector $z \in D^n$ such that $\forall i \in \{1, n\}, z[i] = q(v[i])$ is called a *configuration*. Since CSP/COP models a specific problem $\mathcal{P}$, a CSP/COP formula represents an instance of $\mathcal{P}$.

Roughly speaking, there exist two different kinds of algorithms to solve CSP/COP problems:

- Tree-based search algorithms, also called complete algorithms since they completely explore the search space of solutions, using smart moves (backtracking, forward checking, etc) to localize and avoid dead-ends in the search space.

- Meta-heuristics, also called incomplete algorithms, since they are based upon local moves to find optimums, rather than looking at the problem as a whole. In practice, these algorithms are the only viable methods to manage industrial-size problems within a reasonable runtime. On the other hand, they cannot prove they have found an optimal solution, neither they cannot prove there are no solutions to the given problem.

For GHOST, we have chosen a meta-heuristic algorithm to be the heart of the solver, *Adaptive Search* from Codognet and Diaz [7]. The reason we have chosen a meta-heuristics is simple: To solve combinatorial and optimization problems while playing a RTS game, the solver needs to be very fast to find a solution, within some tens of milliseconds, which is virtually impossible with tree-based search algorithms. The reason we have chosen *Adaptive Search* is that, although it is not a well-known algorithm, it is one of the fastest meta-heuristics at the moment, up to our knowledge (see Caniou et al. [8]).

In the next sections, we will see that looking for the optimal solution is not always an interesting strategy for RTS games. Indeed, it is sufficient to have a solution "optimal enough" in some tens of milliseconds rather than spending seconds to find a global optimum. Also, the reader has to keep in mind that meta-heuristics are stochastic methods, then two runs on the same problem instance leads to two different solutions within the same runtime. This is why results in this paper are the average of 10 to 100 repeated experiments, regarding the problem. These results with GHOST are very good on all tested problems, and even often better than results the reader can find in the current literature.

2.2 GHOST architecture

GHOST is a template C++ solver[1], under the GNU GPL v3 licence, designed to solve any kind of RTS-related problems, as soon as they are modeled into a CSP or a COP. Two different users are targeted:

- The casual user, who only wants to use GHOST to solve an already encoded problem, like the three problems presented in next sections. This user only needs to instantiate variables, the domain, constraints and eventually an objective to describe the instance of his problem, and to call the function `solve` of the solver to run the search. This is done in 5 short C++ lines.

- The developer user, who has a specific problem not written with GHOST yet. GHOST has been designed to let the implementation of new problems easy without changing a single line in the solver and the existing code, and without expertise needed in Constraint Programming. Also, our solver has been designed to have as few parameters as possible, to avoid tedious and time-consuming parameters tuning before obtaining interesting results.

GHOST is implemented around five main C++ classes: `Variable`, `Domain`, `Constraint`, `Objective` and `Solver`[2]. Interactions between these classes are depicted in Figure 1.

The function `solve` defined in `Solver` follows the steps exposed in Figure 2. It is composed of two main loops: the outer loop for optimization, containing the inner loop for satisfaction. The satisfaction part, in red in Figure 2, only tries to find a possible solution among all configurations, *i.e.*, tries to find an assignment of each variable such that all constraints of the CSP are satisfied. It is possible to call `Solver::solve` without defining any objective functions. In that case, a default objective is applied, doing nothing special, and the solver will output a solution that satisfied all constraints, if it finds one.

If an "real" objective function is set, then the optimization part, in blue in Figure 2, is triggered. Even more: it will influence the satisfaction part (finding a valid solution) if the objective implements optional heuristics to select the variable to change and the value to assign, if the current configuration is not a solution.

[1] Source code available at github.com/richoux/GHOST
[2] See GHOST manual pages (richoux.github.io/GHOST/) for more details

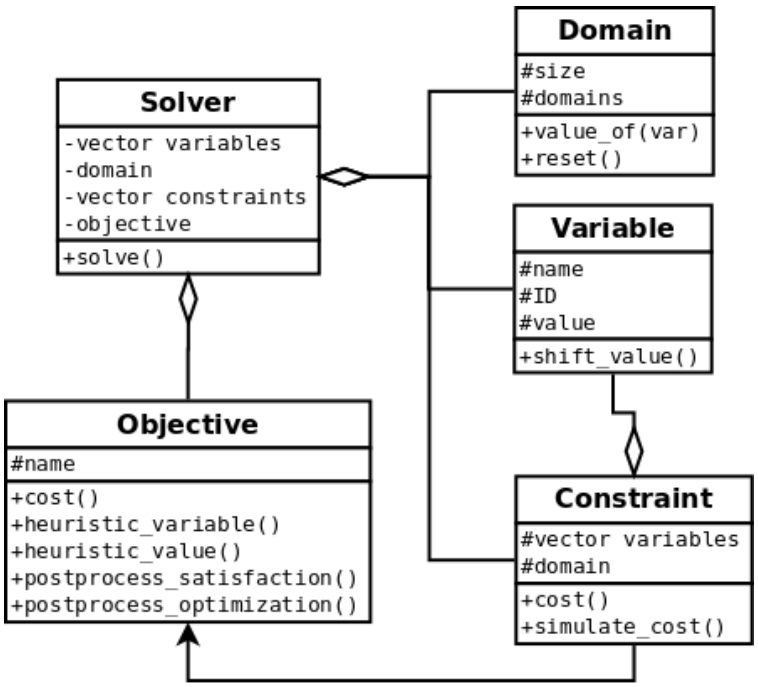

Figure 1: A simplified class diagram of GHOST

The optimization part also applies two optional post-process optimizations, one on the output of the satisfaction loop, one on the final output, giving the solution returned by the solver. We will detail the purpose of such functions later, in particular in Section 4.

The fundamental thing in the optimization part is the optimization loop itself. To explain how this loop works, we have to introduce the two temporal parameters in GHOST. The function `solve` takes two parameters: The first one, mandatory, is the satisfaction timeout x in milliseconds. It means that, if after x ms, we leave the satisfaction loop, certainly without a valid solution since the loop stops as soon as it finds a solution. The second parameter, optional this time, is the optimization timeout y, always in milliseconds. It corresponds also to the total runtime of GHOST, modulo the post-process after the optimization loop (which is negligible in practice and should be about 100 times smaller than y). If the y parameter is not given, then it is set to 10 times x.

Thus, the optimization loop repeats n times the satisfaction loop, and receives in total $m \leq n$ valid solutions. After receiving a new solution from the satisfaction loop, and applying an eventual satisfaction post-process, it calls the objective function to compute the optimization cost, compares it with its saved solution (if any) and keeps the solution with the lower cost. Then it repeats the satisfaction loop to obtain a new solution, and so on, until y ms are reached.

In some way, GHOST is applying a sort of Monte Carlo sampling. Notice that this is not MCTS. It would certainly possible to implement a MCTS though GHOST, by twisting derived classes from `Variable`, `Domain`, `Constraint` and `Objective`, however the authors do not recommend to use GHOST this way, since it should be more efficient to develop a proper, ad-hoc MCTS program.

There exists a third and last parameters in the solver: The length of the tabu list. Indeed, *Adaptive Search* contains a tabu list of explored variables in order to not revisit the same variable too soon during the search. The tabu parameter is then a number expressing how long the solver has to wait before being allowed to revisit an explored variable. Actually, we also have implement escape mechanisms to not make this tabu list strict, by allowing the solver to draw a tabu variable if there are really no other interesting variables. GHOST's tabu list is thus more like a priority list.

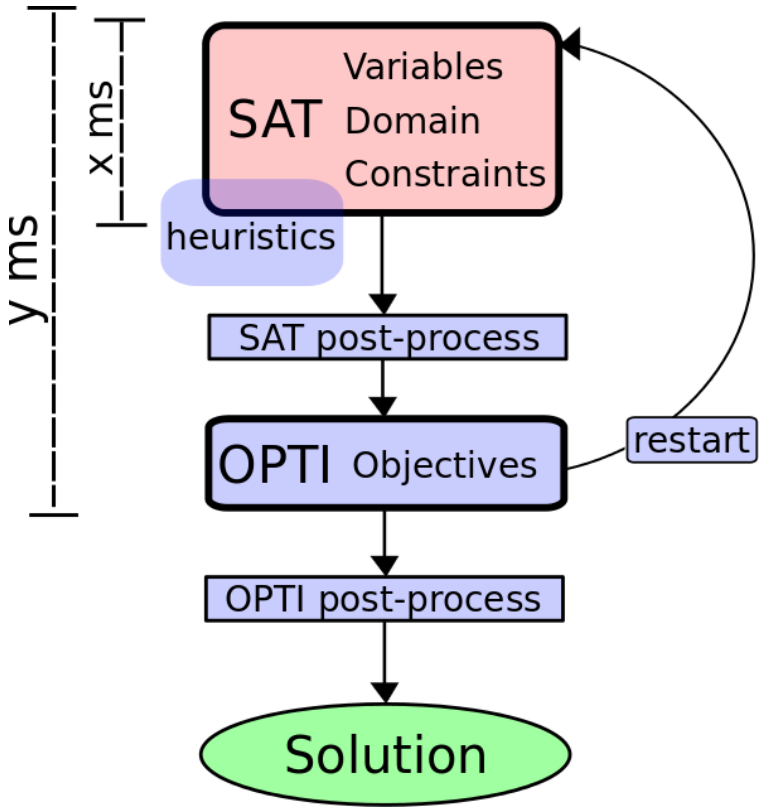

Figure 2: GHOST architecture: In red, the satisfaction inner loop, running for x milliseconds top. In blue, optimization mechanisms. The whole process, excluding optimization post-process, runs under y milliseconds.

During our experiments, we realized that the same value for this parameter was leading to optimal performances, when set to $|V| - 1$. It is then a priori not necessary to tune this parameter, however this remains of course possible to the developer user.

A developer user has to build his own classes upon `Variable`, `Domain`, `Constraint` and `Objective`, by inheriting from them. For instance, to create a class of variables representing units, this is done by `class Unit : public Variable`. Classes composed of other classes, like `Domain` which needs to know on what variables it will work with, must instantiate the right templates. Thus, declaring the domain for the target selection problem is done by `class TargetDomain : public Domain<Unit>`.

Concerning objective functions, GHOST has been designed to minimize their value. If a developer user needs to maximize an objective function f, this can be simply adapted to GHOST by defining the objective function $1/f$. Our solver is designed to deal with mono-objective optimization problem only, thus, one can only choose at most one objective function at the time before running the `solve` function, however the objective function can be dynamically changed between two calls of `solve`. The choice of designing a mono-objective solver is pragmatic: Multi-objective solvers are in general significantly slower, since dealing with more complex problems. Multi-objective solvers are also more difficult to implement, which makes harder one goal of GHOST: To propose a solver both easy to use and easy to extend by implementing new problems.

In the next three sections, we explain how we modeled a reactive control problem, a tactic problem and a strategy problem with a respective CSP/COP, and what results we obtained by applying GHOST. We would like to stress that no modifications, no optimizations of the solver has been done to manage these different problems: The core of the solver, *i.e.*, everything in the `Solver` class, remains unchanged. Even if

post-processes are defined differently in their respective `Objective` classes, the way they are included and called into the solver is rigorously the same.

3 Reactive control problem: The target selection

Reactive control problems have been fairly well studied for StarCraft, with many different techniques applied: We can cite Synnaeve and Bessière proposing in [9] a Bayesian model allowing units to move in group without being in each other's way, finding the right distance to attack, etc; Uriarte and Ontañón in [10] using influence maps for kiting, *i.e.*, moving backward while recharging then attacking forward when one is ready to shoot, and Churchill et al. in [11, 12] showing a heuristic search method making combats outcome predictions. These two last papers are also among the rare ones dealing with the target selection problem in StarCraft.

3.1 Problem statement and model

The target selection problem is a classical reactive control problem: A player has to cope with it several times in each fight. Its satisfaction version is very easy: It consists of assigning to each unit of a fighting group a reachable target, *i.e.*, an enemy unit within our unit range.

In addition, one has also to take into account a cooldown, that is to say a fixed number of StarCraft unit time within a unit recharges and has to wait before shooting again. The cooldown is the same for all units of the same type. We can now describe the target selection problem with the following CSP:

CSP model for the Target Selection problem:
Variables: A group of our units.
Domain: A group of enemy units.
Constraints: Each living, ready-to-shoot unit must aim a living enemy within its range, if any.

The target selection problem is a frame-by-frame problem: We only consider the question "what enemy should I shoot this frame?", without looking at micro-management moves like kiting.

Usually, RTS games define specific characteristics to each unit type making the target selection more subtle. Thus in StarCraft, each unit type has a size (small, medium or large) and a damage type (concussive, normal or explosive). Table 1 shows the damage efficiency according to the aimed unit size and the shooter damage type. For instance, a normal-damage unit will always afflict a target with full damage, whatever the target size. But a concussive-damage unit, like a Terran Vulture, will only make 5 damage points (without counting upgrade neither armor) against a large unit like a Terran Tank, instead of 20 damage points as usual. Moreover, some units have a splash attack, afflicting damage to the target's neighbors. In StarCraft,

Table 1: Damage efficiency matrix in StarCraft.

		Damage type		
		Concussive	Normal	Explosive
size	*Small*	100%	100%	50%
	Medium	50%	100%	75%
	Large	25%	100%	100%

two splash attack types exist: The linear splash, where the unit can hit a line of enemies, such as the Zerg Lurker, and the radial splash where an attack triggers a circled shockwave hitting units around the target within one the three splash radiuses. The first radius afflicts 100% of damage, the second one 50% and the last one 25%. Terran Firebat are special cases mixing both linear and radial splashes.

Splash damages combined with damage efficiency lead to interesting optimization opportunities. In this work, we investigated two different objective functions:

- **Max damage**, where our group tries to deal as much damage as possible within the current frame.

Figure 3: Mirror Terran unit composition and placement used for experiments. Line 1: 5 marines. Line 2: 2 Goliaths and 2 Vultures (in the middle). Line 3: 2 Siege Tanks in tank mode and 2 Ghosts (in the middle). Line 4: 1 Siege Tank in siege mode, doing splash damage.

- **Max kill**, where our group tries to kill as much enemy units as possible within the current frame.

Notice that these two objective are more complex that looking for the maximal damage or the maximal dead enemies each unit can do independently: Imagine a scenario where we have two units U1 and U2 and two enemies E1 and E2, such that U1 can afflict 10 damage points to E1 and 9 to E2, U2 can afflict 8 damage points to E1, but E2 is out of range, and E1 has 5 hit points (HP) left. The best global assignment is U1 to E2 and U2 to E1, even if U1 deals more damage to E1.

3.2 GHOST implementation and results

Figure 3 shows the mirror setup we used for experiments. We chose Terran units since many of them are long-range attack units and this makes the target selection problem more interesting. Also, we packed units, places them close each others, to make significant the Terran Siege Tank's splash damage. Terran Firebat has not been taking into account in the small simulator we wrote for testing GHOST on the target selection problem. We first planned to use SparCraft [3, 11, 12], but we were lacking of time to learn how to use it

Table 2: Average results over 100 simulations for both objective functions, with the setup shown in Figure 3. The first table shows experiments where calls to GHOST lasts for 3ms, and the second table calls lasting for 5ms

				GHOST victory		Opponent victory	
	Objective	Win %	# Draws	# GHOST living units	GHOST HP	# Opponent living units	Opponent HP
3ms	*Max Damage*	86.0	3	2.8	153.4	1.0	37.1
	Max Kill	80.0	2	3.0	163.2	1.1	62.8
5ms	*Max Damage*	91.0	1	2.8	149.6	1.1	58.7
	Max Kill	82.0	3	3.0	169.5	1.0	37.2

and combined it with GHOST. It was then faster to develop an quick ad-hoc simulator (about 200 lines of C++ code).

The simulator is here to emulate a combat of two unit groups and thus needs to trigger cooldowns, manage each unit HP, apply damages, compute Euclidian distances between units. Ranges, damage efficiency and splash damages are directly managed by GHOST. It is important to stress that, in this simulator, enemy units are randomly shooting at a reachable target.

Our simulator does not implement:

- Healing, repair, HP or shield regeneration,
- Terrain level (high/low ground),
- Air shots (in StarCraft, there is always a small chance to miss the target),
- Friendly fire,
- Firebat's unique splash damage.

We ran 100 simulations for both objective functions until one group is completely annihilated, calling GHOST at each StarCraft unit time. Eventually, the two groups can kill one another, leading to a draw. We can then compute GHOST's win rate, as well as the average number of GHOST's living units at the end of the simulation and the average of remaining HP for GHOST living units when GHOST wins, and the same averages when the opponent wins. For these experiments, we first fixed the x satisfaction timeout and y optimization timeout parameters respectively to 1ms and 3ms, and we ran a new series of experiments with parameters $x = 2$ and $y = 5$. In [11, 12], Churchill et al. propose an Alpha-Beta search and run it for 5ms. This is why we have chosen to set the optimization timeout to 3ms and 5ms to compare our results with the same timeout and with a shorter one. For this problem, no particular post-processing optimizations have been necessary.

Although Churchill et al. method predict a 100% victory in average in the different scenarios they have tested, the real win rate in-game was 84%. It is however difficult to compare their results to ours from Table 2 for many reasons: their 84% of victory comes from in-game experiments against the built-in AI, where units are moving. Our simulator, although realistic for damage, do not emulate movements since it is outside the scope of the target selection problem (*i.e.*, answering the question "what enemy units should I aim this frame?"). Plus, the simulator is considering an enemy shooting randomly, and the built-in AI is applying a smarter heuristic.

Table 2[3] shows that the Max Damage objective wins 91% of the time against the mirror unit group shooting randomly if we let 5ms to GHOST to do computations (and 86% within 3ms). The Max Kill objective seems less efficient with a win rate of 82% within 5ms, (80% within 3ms). One explanation is that Max Kill may not be a heuristic as good as Max Damage when enemy units have their full HP. It would be interesting to cross these two objectives to see if it leads to better results.

[3]All target experiment results and the simulator can be found at github.com/richoux/GHOST_paper/tree/master/xp/target

For both objectives, we see that GHOST victories are undeniable, with in average 2.9 remaining units after the simulation, whereas losses are tight, with just one living enemy unit at the end of the combat. The average of total remaining HP is also clearly in favor of GHOST.

3.3 Future works

To go further, we could implement additional objective functions for the target selection problem, like minimizing damage waste, *i.e.*, to try to be as close as 0 HP while killing an enemy unit instead of shooting a 5-HP target with 20 damage points for instance. Even if GHOST is a mono-objective solver, we could craft new objectives by mixing already encoded ones, by simply applying a priority heuristics, like "maximize killings first, and consider maximizing damage as a tiebreaker".

The current implementation only deal with Terran ground units for the target selection problem. Extending it to all StarCraft units would be easy. Finally, improving the current simulator or using SparCraft would make our experiments sharper.

4 Tactic problem: Wall-in

Up to our knowledge, tactic problems has not been intensively investigated for StarCraft. We can cite however the broadly used BWTA library implementing a Voronoi decomposition to cut out and find a delimitation between regions and chokepoints presented in [13].

However, the only works dealing with the wall-in problem is the pioneer paper from Certicky [14] and the wall-in solver from Richoux et al. in [6]. As already written, the latter provided the basis for the present work.

4.1 Problem statement and model

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 easier to defend in case of invasion. In this section we will focus only on walls constructed by buildings, discarding small narrow passages that can be closed by small units like workers.

We 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. 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 this paper, we will call *significant gap* a gap allowing Zerglings (16×16 pixels) to enter into the base.

The wall-in problem has been first modeled in CSP by Certicky in [14]. Then, Richoux et al. proposed a different model, always in CSP, in [6]. The work published in the latter has been GHOST foundation. One can see GHOST has a deep extension and generalization of the solver used in [6].

For GHOST, the model of the wall-in problem is identical to the one in [6]:

CSP model for the Wall-in problem:
Variables: Buildings of the player race.
Domain: Possible positions around the chokepoint.
Constraints: Overlap, Buildable, NoHoles and StartingTargetTile.

Constraints of our models are illustrated in Figure 4. They are defined as follows:

- **Overlap**: buildings do not overlap each others.

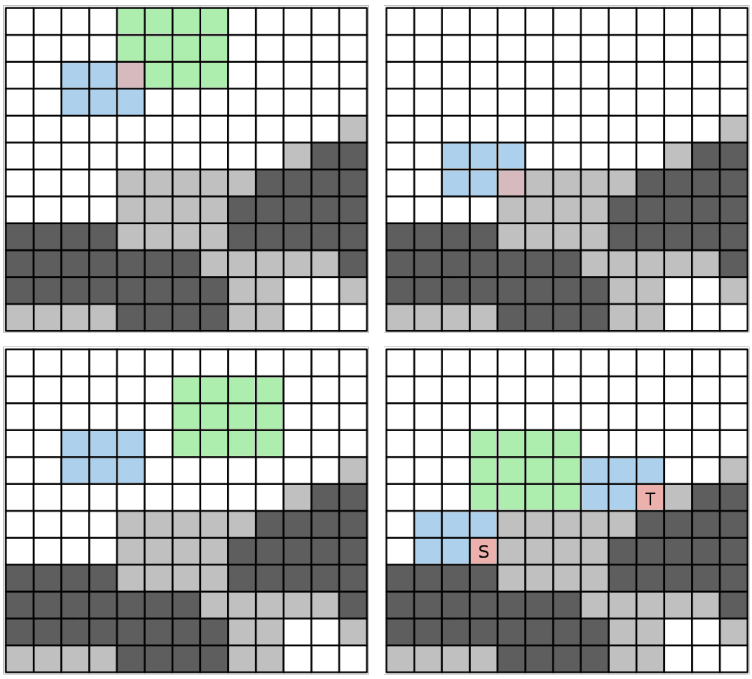

Figure 4: Constraint are: Overlap (upper-left), Buildable (upper-right), NoHoles (bottom-left) and StartingTargetTile (bottom-right). Dark grey tiles represent unwalkable and unbuildable tiles. Light grey tiles are walkable but unbuildable tiles.

- **Buildable**: buildings do not overlap unbuildable tiles.
- **NoHoles**: no holes of the size of a build tile (or greater) in the wall.
- **StartingTargetTile**: there are exactly one building constructed on a given starting tile s, and one building (it can be the same one) on a given target tile t.

Actually, Overlap, Buildable and NoHoles are sufficient to make a wall. We added the constraint StartingTargetTile to help the solver to find how to surround the chokepoint.

4.2 GHOST implementation and results

Like for the target selection problem, we focused on the Terran race for experiments. The wall-in problem offers many interesting optimization opportunities. Therefore, we have implemented three different objective functions, aiming to minimize:

- **Building**: the number of buildings in the wall,
- **Gap**: the number of significant gaps in the wall,
- **TreeTech**: 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).

To evaluate the technology level of a wall, we simply take the depth in the technology tree of the most technological building composing the wall. Thus, a Command Center has a depth 0, a Barracks a depth 1, a Factory a depth 2, and so on.

This time, the satisfaction post-processing is important for these three objective functions, and the optimization post-processing really helps to improve the Gap and the TreeTech objectives.

The role of the satisfaction post-processing is to "clean" the proposed wall, *i.e.*, to remove all unnecessary buildings in the valid solution, such that the resulting wall still satisfies the four constraints of the model.

The optimization post-processing used with the Gap and TreeTech objectives tries to swap each building of the proposed wall with another building from the set of variables V, of the same size and not already used in the wall. This simple permutation can drastically decrease the number of significant gaps between buildings and the technology level required.

Table 3: Results over 48 chokepoints extracted from 7 StarCraft maps. Results are the average of 100 runs for each chokepoint. Each calls of GHOST lasts for 150ms

	Satisfaction run	*Optimization run*	*% solved (opti)*
Building	4.05	2.56	98.04%
Gap	1.32	0.03	97.50%
TreeTech	1.99	1.35	97.54%

Satisfaction runs in Table 3 are GHOST runs without any objective functions and with a satisfaction timeout of 160ms, like in [6] with the difference that in Richoux et al.'s paper, they compiled 8 satisfaction runs of 20ms each to have great chances to find a valid solution. We measure their average number of buildings, significant gaps and technology level in order to match them with optimization runs. Always following the experiment methodology of [6], optimization runs are slightly defavored since there global timeout is 10ms shorter than satisfaction runs, with 150ms. For optimization runs, x and y parameters where fixed to 20 and 150, respectively. We can see in Table 3[4] that optimization runs leads to real improvements compared to satisfaction runs. This is particularly true with the Gap objective, where significant gaps are almost completely eliminated: Over 4,800 runs in total, 4,680 walls have been found (97.50%), and 4527 of them are perfect walls (*i.e.*, without any significant gaps). In other words, 96.73% of walls found by GHOST are perfect.

Since GHOST code has been improved compared to the solver in [6], we obtained slightly better results: the percentage of problems solved goes from 95-96% to 97-98%, walls decided with the Building objective were composed of 2.65 buildings against 2.56 now, the number of significant gaps goes from 0.05 to 0.03 and the technology level from 1.56 to 1.35 currently.

Table 4 shows the percentage of walls found in each of the seven maps from where chokepoints were extracted. These numbers correspond to pure satisfaction runs, since they do not defer significantly for optimization runs. We can see that GHOST has more difficulties to find a wall for Fortress chokepoints. Actually, it is failing from time to time on the same chokepoint, where a valid solution can only be achieved by using two 3×2-sized buildings.

Table 4: Percentage of solutions found for each map.

Map name	% solved
Python	100
Heartbreak Ridge	100
Circuit Breaker	99
Benzene	99
Aztec	97
Andromeda	96
Fortress	90

[4] All wall-in data and experiment results can be found at github.com/richoux/GHOST_paper/tree/master/xp/wallin

4.3 Future works

For the wall-in problem, we could add new objective functions to widen possibilities. For example, trying to make a wall by minimizing the cost, or the makespan. This last objective is trickier and related to the next section (it is somehow the wall-in + the build order problems).

Most importantly, since the required runtime to correctly optimize a wall is longer than a StarCraft frame duration, we could implement GHOST in order to support computation pauses and resumes. Actually, GHOST architecture has been designed by keeping in mind this feature: The satisfaction part is executed in 20ms, *i.e.*, it can be executed within one StarCraft frame in the fastest mode. Marking a pause after each satisfaction loop and resume GHOST at the next frame until the computation ends would not be difficult to do. This is more discussed in detail in Section 6.

In addition to extend the current code to manage all StarCraft races, results in Table 4 give us the feeling that our wall-in model can be refined again to reach a higher percentage of found solutions.

5 Strategy problem: The build order

The reader can find an extensive literature about build order planner and/or prediction for StarCraft: Churchill and Buro propose in [3] a build order planner using a branch and bound technique, Kuchem et al. analyze build order tools for StarCraft II in [4], Cho et al. present in [15] a strategy prediction and build order adaptation system learning from replays and in [16] Synnaeve and Bessière show a Bayesian model to predict the opponent build order in-game.

5.1 Problem statement and model

A build order planning is a series of actions following a specific timing, in order to achieve a goal. Such a goal is a combination of buildings, units, upgrades and researches produced. Usually, the objective for a player is to reach a fixed goal the fastest possible way. However, alternatives can also be considered, like reaching the goal without sacrificing the economy, or focussing first on units in order to have an army quickly available.

A build order planning can be intuitively modeled by a CSP as a permutation problem, where a bijection maps the set of variables to the domain. Changing the value of one variable is then actually swapping its value with another variable.

All actions have a (potentially empty) dependency list, *i.e.*, an action α has a list of actions that are required before starting α. For instance, to start the *Air Weapons Upgrade level 2*, it is required to have finished the *Air Weapons Upgrade level 1*. Notice that we can dive recursively into these dependency lists. Thus if someone aims to do *Air Weapons Upgrade level 2* for Protoss, it required *Air Weapons Upgrade level 1*, which requires itself a *Cybernetics Core*, which requires a *Gateway*.

The CSP model we propose for the build order planning problem is the following one:

CSP model for the Build Order Planning problem:
Variables: All actions we need to reach our goal.
Domain: Order of actions.
Constraints: Each dependency of an action α must occurs before α.

5.2 GHOST implementation and results

We have chosen to focus on Protoss to test GHOST on the build order problem. The current implementation can deal with any Protoss buildings, units, researches and upgrades.

For this problem, we have implemented one objective function only: Minimizing the build order makespan. With this objective function, we did not have implemented a special satisfaction post-processing, but we did for the optimization post-processing. Imagine the case where the user asked, among others, to produce n units of type U. Thus, GHOST will automatically add, recursively, all dependencies of U into the variable set. Suppose the unit type U is produced by the building of type B, and the user eventually asked for $m < n$

buildings of type B. After having computed an optimized build order, the optimization post-processing will retake this solution and try to see if it can shorten the makespan even more by constructing more buildings of type B, to speed up the production of units of type U. This post-processing optimization is significantly efficient in cases the user ask for a high number n of the same unit U, and none or a low number m of B.

Unlike the target selection problem, we needed this time to integrate a simulator inside GHOST to emulate a game (without combats) and be able to compute the makespan of build orders. Thus, this simulator must emulate resources gathering, units producing (including workers), supply capacity, constructions, etc. Our simulator will always try to produce workers until reaching saturation (24 workers per base), as well as maintaining supply in order to never be "supply blocked", that is, unable to produce a unit because we reached the supply capacity limit.

In [3], Churchill and Buro give details about the simulator they developed for their build order planner. We first used the same settings, but after matching GHOST results against build order from Korean pro-gamers, we realized that these settings were too advantageous for the simulator. We then closely analyzed some replays from Korean pro-gamers to refine our simulator settings, listed below (in StarCraft unit time t):

- Time to go build something: 5t (4t in [3])
- Time to go back gathering minerals after building something: 4t (0 in [3])
- Time to go from the base to mineral patches to start mining: 5t (0 in [3])
- Time for a worker to switch from mineral to gas: 5t (0 in [3])
- Mineral gathering rate: 0.68 mineral per worker per t (1.07 in [3])
- Gas gathering rate: 1.15 gas per worker per t (1.66 in [3])

To be sure these parameters make our simulator realist, we matched its execution against the first 2 minutes of games played by Korean pro-gamers. Table 5 give an example of our simulator matched against the Protoss Korean pro-gamer Kim "Bisu" Taek Yong. Gas is not revealed since it remained at 0 for both Bisu and the simulator during the first 2 minutes. Each time the simulator started to produce a probe, *i.e.*, a worker, we write down the simulator and Bisu mineral stock and supply situation (used supply over supply capacity). One can see that these two early game are very similar, with a slight advantage to the simulator since its probe production is nearly perfect.

Table 5: Our simulator execution matched against the pro-gamer Bisu during the first two minutes.

		Simulator in GHOST		Bisu	
Action (by S(im) or B(isu))	Time t	Mineral	supply (used / capacity)	Mineral	supply (used / capacity)
S starts a probe	20	40.8	5/9	40	5/9
S starts a probe	44	19.7	7/9	20	7/9
S starts a probe	64	46.5	8/9	50	8/9
S starts a pylon	75	-	-	-	-
B starts a pylon	78	-	-	-	-
S starts a probe	89	1.2	9/9	12	9/9
S starts a probe	109	60.0	10/17	132	9/17
B starts a gateway	116	-	-	-	-
S starts a gateway	125	-	-	-	-
S starts a probe	125	1.8	11/17	78	9/17

For Table 6[5], GHOST has been run 10 times on each build order. We run experiments on two sets of

[5]All build order data, experiment results and the simulator can be found at github.com/richoux/GHOST_paper/tree/master/xp/build_order

Table 6: Average makespan of Humans and GHOST build orders in StarCraft unit time over 3647 games. Each calls of GHOST lasts for 30ms

Games till 10.000 frames				
Match-up	Humans	GHOST	% solved	Gain
All	656.02	619.62	94.4	36.40
PvP	651.51	608.06	95.0	43.45
PvT	660.50	628.17	93.9	32.33
PvZ	649.20	609.34	95.0	39.86
All pro	643.38	597.25	96.3	46.13
Games till 7.800 frames				
Match-up	Humans	GHOST	% solved	Gain
All	522.50	491.11	98.8	31.39
PvP	516.08	485.56	99.3	30.52
PvT	527.66	506.65	98.3	21.01
PvZ	515.81	458.23	99.7	57.58
All pro	506.38	480.88	100	25.50

build orders: Those extracted from replays by Gabriel Synnaeve[6] and refined by Glen Robertson[7], and those extracted from top Korean pro-gamers.

In total, the first set is composed of 3647 build orders: 768 Protoss versus Protoss, 2043 Protoss versus Terran and 836 Protoss versus Zerg. Replays where these build orders occur are coming from TeamLiquid, GosuGamers and ICCup web sites. Table6 shows that GHOST outperforms by far human build orders, with a mean of 36.4 StarCraft unit time of gain considering 10,000 frames build orders and 31.39 StarCraft unit time of gain considering 7,800 frames build orders, all match-ups taken together.

We have also analyzed some few replays of games played by top Korean pro-gamers, *i.e.*, Kim "Bisu" Taek Yong, Doh "BeSt" Jae Wook, Woo "Violet" Jung Ho, Yang "Cure" Jung Hyun, all Protoss players, against Lee "Flash" Young Ho (Terran), Lee "Jaedong" Jae Dong (Zerg), Ma "sAviOr" Jae Yoon (Zerg) and we match their build orders with GHOST. Since we had to manually select those replays, we have only downloaded 6 of them (2 for each match-up including a Protoss player), giving us a set of 8 build orders to give to GHOST.

Results against these pro-gamers are shown at the very last line of Table 6. One can see that GHOST clearly obtains better build order than Protoss Korean pro-gamers listed above, with a gain of 45.38 StarCraft unit time for 10,000 frames build orders and 24.88 StarCraft unit time for 7,800 frames build orders. We have to lower the first result by stressing that pro-gamers are often already engaged in a fight before 10,000 frames and/or are applying ouside-the-book strategies, and thus minimizing the build order makespan may not be their first priority anymore. To a lesser extent, this is also true with 7,800 frames build orders.

Computation time is only 20ms for satisfaction runs and 30ms for the (global) optimization run. This means that GHOST can compute a highly optimized build order within only one StarCraft frame at the fastest speed. In [3], Churchill and Buro's branch and bound method is computing 90% of the time build orders with the same makespan as pro-gamers in about 3.735 seconds (for build orders with a makespan up to 249s), giving thus a CPU time / makespan ratio of 1.5%. Considering GHOST is computing in average build orders with a makespan of 619.62 StarCraft unit time within 30ms. In the fastest mode, 619.62 StarCraft unit time corresponds to about 423s; leading to a CPU time / makespan ratio of 0.007%.

Why planning a build order is easier than wall-in? A huge part of the answer is that planning a build order can be modeled in CSP by a permutation problem, which drastically decrease the combinatorial complexity of the problem. Also, the satisfaction part for the target selection and the build order problems are not difficult to compute, since constraints modeling these problems are trivial. This is not the case for

[6] emotion.inrialpes.fr/people/synnaeve/TLGGICCUP_gosu_reps.7z
[7] scidrive.uoa.auckland.ac.nz/gameai/scdata/files.txt

the wall-in problem, where even getting a non-optimized valid solution is hard.

5.3 Future works

As for all other problems, we could implement another objective function for the build order problem. For instance, it would be natural to propose an objective function trying to first minimizing the makespan of all army units asked by the user, and then to minimize the makespan of remaining actions (buildings, researches, upgrades). This would allow the user to secure first his base with an army.

6 Discussion and conclusion

In this paper, we introduced GHOST, a combinatorial optimization solver to solve any problems encoded by a constraint satisfaction/optimization problem. We presented three different RTS problems belonging to a specific level of abstraction, and proposed a CSP/COP model for each. Experiments applying GHOST on these models shown very good results computed within some tens of milliseconds, without any modification or optimization of the solver source code. Results obtained are often better than the ones we can find in the current literature.

One claim written in Section 2 is now clear: Looking for the absolute optimal solution may not be the best strategy for RTS games; runtimes to find such solutions can last far beyond what one can allow oneself. Fast meta-heuristics can output an "optimal enough" solution in some tens of milliseconds, and if no good enough solutions has been found, the user can always re-run the solver on the next frame. One has to seriously consider such a way to manage and solve optimization problems in RTS games; results shown in this paper proves it is absolutely viable.

The discerning reader had noted that GHOST is actually a general CSP/COP solver, and can not only solve any RTS-related problems modeled with this formalisms but also any kind of such problems from other fields. One of our starting point is to make GHOST a C++ library to be very easily included into StarCraft bots. This should be done in the next few months by using the BWAPI 4 library, giving us (almost) all informations we need on units, buildings, events, etc, in StarCraft for each race. We could also use other libraries related to other RTS games to make GHOST broadly available.

Some improvements we have in mind concern the implementation of a pause/resume system, allowing GHOST to start a long computation and hash it into small pieces fitting within one frame (say, less than or equals to 30ms to let some time for other computations). GHOST architecture has been designed to make such an implementation easy to do, in particular thanks to the decoupling satisfaction loop - optimization loop. Another improvement would be to let the solver check how many cores are available in the machine running it, and use all of the to speed up the search. This can also be done easily since *Adaptive Search* is known to be very efficient with a straightforward parallel scheme (see Caniou et al. [8]). Indeed, this algorithm has been parallelized on super-computer and shows linear speed-ups over 8,192 cores on some problems (and fairly good speed-ups on others), which are impressive parallel performances.

GHOST has also been designed following the famous Object Oriented programming "open-close principle", to let the door open for extensions without needing to modify the already written classes. It is easy to implement and include new problems in GHOST, and the authors highly encourage contributors to propose implementations of new problems to integrate into the library.

Finally, this work leads us to take a global view on CSP/COP, and to consider the following: Even if a huge number of combinatorial and optimization problems can be modeled with CSP/COP, this framework is not well adapted to deal with uncertainty or incomplete information. This is penalizing for many RTS-related problems, where most of interesting challenges come from the fact that information is incomplete. Some variations of Constraint Programming propose to take into account uncertainty through formalisms like soft constraints or fuzzy constraints, however none of these formalisms favor the design of efficient solvers, up to our knowledge. Thus, far beyond the scope if the present work, we would like to investigate on a new CSP formalism that could manage uncertainty efficiently.

References

[1] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *Transactions on Computational Intelligence and AI in Games (TCIAIG)*, vol. 5, no. 4, pp. 293–311, 2013.

[2] G. Robertson and I. Watson, "A review of real-time strategy game AI," *AI Magazine*, 2014.

[3] D. Churchill and M. Buro, "Build order optimization in starcraft," in *AIIDE*. AAAI Press, 2011, pp. 14–19.

[4] M. Kuchem, M. Preuss, and G. Rudolph, "Multi-objective assessment of pre-optimized build orders exemplified for starcraft 2," in *CIG*. IEEE, 2013.

[5] R. Lara-Cabrera, C. Cotta, and A. J. Fernández-Leiva, "A self-adaptive evolutionary approach to the evolution of aesthetic maps for a RTS game," in *World Congress on Computational Intelligence (WCCI)*. IEEE, 2014.

[6] F. Richoux, A. Uriarte, and S. Ontañón, "Walling in strategy games via constraint optimization," in *AIIDE*. AAAI Press, 2014.

[7] P. Codognet and D. Diaz, "Yet another local search method for constraint solving," in *SAGA*. Springer Verlag, 2001, pp. 73–90.

[8] Y. Caniou, P. Codognet, F. Richoux, D. Diaz, and S. Abreu, "Large-scale parallelism for constraint-based local search: The costas array case study," *Constraints*, vol. 19, no. 4, pp. 1–27, 2014.

[9] G. Synnaeve and P. Bessière, "A bayesian model for RTS units control applied to starcraft," in *CIG*. IEEE, 2011.

[10] A. Uriarte and S. Ontañón, "Kiting in RTS games using influence maps," in *AIIDE*. AAAI Press, 2012.

[11] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *AIIDE*. AAAI Press, 2012.

[12] D. Churchill and M. Buro, "Incorporating search algorithms into RTS game agents," in *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*. AAAI Press, 2012.

[13] L. Perkins, "Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition," in *AIIDE*. AAAI Press, 2010.

[14] M. Čertický, "Implementing a wall-in building placement in starcraft with declarative programming," *arXiv*, 2013.

[15] H.-C. Cho, K.-J. Kim, and S.-B. Cho, "Replay-based strategy prediction and build order adaptation for starcraft AI bots," in *CIG*. IEEE, 2013.

[16] G. Synnaeve and P. Bessière, "A bayesian model for opening prediction in rts games with application to starcraft," in *CIG*. IEEE, 2011.