GHOST: A Combinatorial Optimization Framework for Real-Time Problems

Florian Richoux, Alberto Uriarte, Jean-François Baffier

***Abstract*—This paper presents GHOST, a combinatorial optimization framework that a Real-Time Strategy (RTS) AI developer can use to model and solve any problem encoded as a constraint satisfaction/optimization problem. We show a way to model three different problems as a constraint satisfaction/optimization problem, using instances from the RTS game StarCraft as test beds. Each problem belongs to a specific level of abstraction (the target selection as *reactive control* problem, the wall-in as a *tactics* problem and the build order planning as a *strategy* problem). In our experiments, GHOST shows good results computed within some tens of milliseconds. We also show that GHOST outperforms state-of-the-art constraint solvers, matching them on the resources allocation problem, a common combinatorial optimization problem.**

***Index Terms*—Game AI, Real-Time, Strategy, RTS, StarCraft, CSP, COP, Solver, Optimization, Combinatorics.**

I. Introduction

ONE can see games as a simplification of the world: the domain is smaller and the rules are easier and less numerous, thus possibilities are more limited. However, games are rich enough to propose complex and dynamic environments where it remains difficult for a computer to make predictions, have a global understanding of the current situation, and then make 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 (RTS) games, where a Clausewitz's fog of war hides the opponent's moves. Hence, RTS games are a good domain for testing Artificial Intelligence (AI) techniques that could be applied afterward in other domains.

As compiled in the surveys from Ontañón et al. [1] and Robertson and Watson [2], many AI techniques have been explored 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 offline to optimize build orders with multiple objectives, and have been analyzed in Kuchem et al. [4]. Also, 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 that is able to model a large number of combinatorial optimization problems, and proposes various sets of algorithms to solve them.

CSP/COP is widely used in AI to solve problems such us pathfinding, scheduling, and logistics [6]. Unlike Mathematical Programming, CSP/COP algorithms are not usually designed to solve one specific problem but are general, *i.e.*, they are able to manage any problem 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 and generalized solver.

A. 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:

- **Strategy** corresponds to the high-level decision making process. This is the highest level of abstraction for 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 model one problem for each of these three families. Then we use our framework GHOST to solve them without any modifications of its inner solver.

B. StarCraft: Brood War

StarCraft: Brood War is an RTS game where three different races (Terran, Protoss and Zerg) can be played, giving an asymmetric but well balanced strategy game. In this paper, the term "StarCraft" will actually refer to the game plus its expansion StarCraft: Brood War.

StarCraft has been a worldwide success, and as of time of publication, it remains the most sold RTS game with nearly

Florian Richoux is with the LINA of the Université de Nantes, France, and the JFLI of the University of Tokyo, Japan.

Alberto Uriarte is with the Computer Science Department at Drexel University, Philadelphia, PA, USA.

Jean-François Baffier is with the JST-ERATO Kawarabayashi Large Graph project at the National Institue of Informatics, Japan.

10 millions of copies distributed[1]. It has been (and still is until now) 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.

A player has to gather two types of resources (minerals and gas) in order to construct buildings. Those buildings are necessary to produce units, to upgrade properties such as damage points or armor, and to unlock technologies and special abilities. Producing units costs resources depending on their properties (hit points, damage points, or size).

The game is played on a rectangular map, discretized into two types of tiles: *walkable* tiles, composed of 8×8 pixels and *buildable* tiles composed of 4×4 walkable tiles, *i.e.*, 32×32 pixels. The difference between these two types of tiles would be explained in Section IV while introducing the wall-in problem. Like most RTS games, the map is covered by a fog of war that obscures players vision of the map everywhere beyond the range of their own (or allied) units/buildings. This means that the player cannot know at any time the full state of the game, unless the whole map is within their vision. 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.

The flow of time in StarCraft is complex: the game has 7 speed modes from “slowest” to “fastest” where “normal” corresponds to the regular frame rate (14.96 logical frames per second, against 23.81 for the fastest mode). This is why when we will write about in-game time, all along this paper and in particular in Section V, we will refer to logical frames (simply denoted as frames) rather than seconds to avoid confusions . In the fastest mode, one logical game frame takes 42 ms. In StarCraft AI competitions, it is necessary for bots to perform calculations in under 55 ms per frame; a bot that takes more than 55 ms per frame in 200 frames or more automatically loses the game.

C. Goals and summary

The research presented in this paper is an extension of Richoux et al. [7], where the problem to make a wall at a given chokepoint (a narrow passage) in StarCraft has been modeled and solved thanks to an ad-hoc algorithm.

The research presented in this paper has two goals:

- To present to the RTS AI community our framework 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 II, we give a short introduction 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 targeted, how to use it to solve an already encoded problem, and how to write your own problems via GHOST. Sections III, IV and V give details about three different problems, each one from a specific level of abstraction (from the lower to the higher: Reactive Control, Tactics and Strategy), how we model them into a CSP/COP and what results we obtain by applying GHOST. Section VI matches GHOST with state-of-the-art constraint solvers. Finally, this paper concludes with future work.

[1]http://www.msnbc.msn.com/id/18925251/

II. GHOST: A General meta-Heuristic Optimization Solving Tool

GHOST is a combinatorial optimization framework designed to tackle real-time problems, helping a programmer to model his/her problem and to solve it using the inner solver, without needing parameter tuning or advanced knowledge in Constraint Programming. RTS games exhibit many real-time combinatorial optimization problems that can be modeled and solved via a Constraint Programming framework. Moreover, a solver dealing with such problems must be fast: in the game industry, only a small percentage of CPU time is allocated for AI (usually no more than 10%). This is particularly true for RTS games, where the AI has a small amount of time (often less than 100 ms) to solve a problem. For instance, while some problems can be solved within several frames, other problems such us target selection or pathfinding need a solution as soon as possible (optimally within one frame).

A. A brief introduction to CSP/ COP

Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP) are two close formalisms extensively used in Artificial Intelligence to solve combinatorial optimization problems. Constraint Programming allows an intuitive and uniform way to model problems, as well as different general algorithms able to solve any (decidable) 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 one of them is the shortest.

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 as true and $\bot$ as false. Thus, regarding the value of the vector V^k, we say that c is either *satisfied* (equal to $\top$) or *unsatisfied* (equal to $\bot$).

Notice also that D should formally be the set of the domain for each variable in V, thus a set of sets 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** is 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 $q : V \to D$ 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 are two different kinds of algorithms to solve CSP/COP problems:

- **Tree-based search algorithms**, also called complete algorithms, completely explore the search space for solutions. These algorithms use smart moves (such as backtracking or forward checking) to localize and avoid dead-ends in the search space.
- **Meta-heuristics**, also called incomplete algorithms, are based upon local moves to find optimums, rather than looking at the problem as a whole. In practice, these algorithms are more suitable to handle 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.

GHOST uses a meta-heuristic algorithm, *Adaptive Search* from Codognet and Diaz [8], as the heart of its solver. The reason we have chosen a meta-heuristics is simple: to solve combinatorial optimization problems while playing a RTS game, the solver needs to find a solution very quickly, often within some tens of milliseconds, which is virtually impossible with tree-based search algorithms. Although it is not a well-known algorithm, we have chosen *Adaptive Search* because it is currently, to our knowledge [9], one of the fastest meta-heuristic algorithm.

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 since meta-heuristics are stochastic methods, two runs on the same problem instance can produce 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 often better than those produced by current state-of-the-art techniques.

B. GHOST *architecture*

GHOST is a C++11 framework[2], under the GNU GPL v3 license, designed to solve CSP/COP problems a game developer could implement, in particular those concerning AI and real-time strategy games . 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 the next sections. This user only needs to instantiate variables, the domain, constraints and possibly an objective to describe the instance of his problem, and to call the function `solve` of the inner 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 make easy the implementation of new problems without changing a single line in the solver and the existing code, and without expertise needed in Constraint Programming. Also, GHOST inner 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`[3].

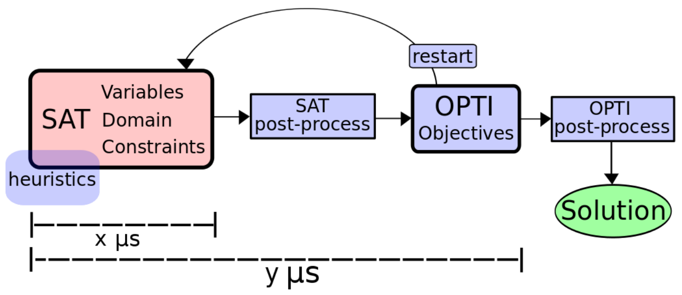

Fig. 1. GHOST architecture: In red, the satisfaction inner loop, running for x microseconds top. In blue, optimization mechanisms. The whole process, excluding optimization post-process, runs under y microseconds. Both x and y are user inputs

The function `solve` defined in `Solver` follows the steps exposed in Figure 1. 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 1, 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 a “real” objective function is set, then the optimization part, in blue in Figure 1, is triggered. Furthermore, it will influence the satisfaction part (finding a valid solution) if the objective implements optional heuristics to select the

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

variable to change and the value to assign, in case the current configuration is not a solution.

The optimization part also applies two optional post-process optimizations: one on the output of the satisfaction loop, and 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 IV.

The fundamental part 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 (which is mandatory) is the satisfaction timeout x in μs, *i.e.*, if a valid solution is not found within x μs, we leave the satisfaction loop without a valid one. The second parameter (which is optional) is the optimization timeout y in μs. 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 $m \leq n$ 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 μs are reached.

In a way, GHOST is applying a sort of Monte Carlo sampling. Notice that this is not MCTS. It would certainly be 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 is 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 implemented 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. During our experiments, we found that a tabu length of $|V| - 1$ provided optimal performance. It is then a priori not necessary to tune this parameter, however this remains 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 objective functions $1/f$ or $-f$. Our framework 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 framework 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 framework 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 emphasize that neither modifications or optimizations of the solver have been done to manage these different problems, as well as the resources allocation problem in Section VI. 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.

III. Reactive control problem: target selection

Reactive control problems have been fairly well studied for StarCraft, with many different techniques applied. Synnaeve and Bessière [10] propose a Bayesian model allowing units to move in groups without being in each other's way or finding the right distance to attack. Uriarte and Ontañón [11] use influence maps for kiting, *i.e.*, moving backward while the weapon is in cooldown and then attacking forward when it is ready to shoot. Finally, Churchill et al. [12], [13] present a heuristic search method that makes combat outcome predictions. These two last papers are also among the rare ones dealing with the target selection problem in StarCraft.

A. Problem statement and model

The target selection problem is a classical reactive control problem that a player has to deal with it several times in each fight. Its satisfaction version is simple; it assigns 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 the waiting time between shots, also known as *cooldown*. Each unit type has different cooldowns. 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. This problem has been studied by Furtak and Buro [14] as an *attrition game* where they proved that this problem is in P-SPACE.

Usually, RTS games define specific properties 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 I 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 inflict 5 damage points against a large unit like a Terran Tank, instead of 20 damage points as usual. Moreover, some units have a splash damage, afflicting damage to any unit inside an area. In StarCraft, there are two types of splash damage: the linear splash, where the damage area is a line to the target; and the radial splash, where the damage area is a circled shockwave around the target with three splash radiuses. The first radius afflicts 100% of damage, the second one 50% and the last one 25%.

TABLE I
DAMAGE EFFICIENCY MATRIX IN STARCRAFT

	Damage Type		
Size	**Concussive**	**Normal**	**Explosive**
Small	100%	100%	50%
Medium	50%	100%	75%
Large	25%	100%	100%

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

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

Notice that these two objectives are more complex than looking for the maximal damage or the maximal dead enemies each unit can do independently. For example, imagine a scenario where we have two units U_1 and U_2 and two enemies E_1 and E_2, such that U_1 can afflict 10 damage points to E_1 and 9 to E_2, U_2 can afflict 8 damage points to E_1, but E_2 is out of range, and E_1 has 5 hit points (HP) left. The best global assignment is U_1 to E_2 and U_2 to E_1, even if U_1 deals more damage to E_1.

B. GHOST *implementation and results*

Our instance is a mirror setup composed of four lines of units. Line 1 contains 5 marines; line 2 has 2 Goliaths and 2 Vultures; line 3 has 2 Siege Tanks in tank mode and 2 Ghosts; and line 4 has 1 Siege Tank in siege mode. We chose Terran units since many of them are long-range attack units and this makes the target selection problem more interesting. Also, we place them close to each other to make the Terran Siege Tank's splash damage more significant.

We use a custom simulator to emulate combats between two group of units. The simulator takes care of cooldowns, each unit HP, damages and Euclidean distances between units. Ranges, damage efficiency and splash damage are directly managed by GHOST. It is important to emphasize that, in this simulator, enemy units are applying very simple target selection heuristics: they aim the unit with the lowest current HP / initial HP ratio, or in other words, the unit which is the most likely to die soon. If several units share the same lowest ratio, then enemy units select randomly one among them.

Our simulator does not implement:

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

We ran 100 simulations for both objective functions until one group is completely annihilated, calling GHOST at each StarCraft unit time. Potentially, the two groups can kill each other, leading to a draw. At the end of the simulation we compute GHOST's win rate, the average number of living units and the average HP of those units. For these experiments, we first fixed the x satisfaction timeout and y optimization timeout parameters respectively to 1,000 μs and 3,000 μs, and we ran a new series of experiments with parameters $x = 2,000$ and $y = 5,000$. In [12], [13], Churchill et al. propose an Alpha-Beta search with a timeout of 5 ms (5,000 μs). This is why we have chosen to set the optimization timeout to 3,000 μs and 5,000 μs to be able to compare our results with the same timeout and with a shorter one. Although, a direct comparison is not possible since their goal is to guide the action selection and not only to decide the target selection. For this problem, we do not need any particular post-processing optimization.

Table II[4] shows that, if we allow 5 ms to GHOST to do computations, the Max Damage objective (shooting with the simple lowest HP ratio strategy exposed above) wins 99% of the time against the mirror unit group (and 98% within 3 ms). The Max Kill objective seems a bit less efficient with a win rate of 96% within 5 ms (94% within 3 ms). 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.

For both objectives, we see that GHOST victories are undeniable, with in average 2.75 remaining units after the simulation, whereas the rare losses are tight, with just one living enemy unit at the end of the combat. The average of total remaining HP is also very clearly in favor of GHOST.

The enemy target selection heuristics actually leads to a list of local optimums for each enemy unit independently, without considering the global situation, *i.e.*, choices taken for partners. At the opposite, GHOST allows to look at the big picture and search for a global optimum instead of a compilation of local ones. Results show this clearly make the difference.

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

TABLE II
AVERAGE RESULTS OVER 100 SIMULATIONS FOR BOTH OBJECTIVE FUNCTIONS. THE FIRST TABLE SHOWS EXPERIMENTS WHERE CALLS TO GHOST LASTS FOR 3 MS, AND THE SECOND TABLE CALLS LASTING FOR 5 MS

					GHOST Victory		Opponent Victory	
	Objective	Wins	Draws	Loses	# Avg. living units	Avg. HP	# Avg. living units	Avg. HP
3 ms	*Max Damage*	98	1	1	2.8	237.9	1.0	12.0
	Max Kill	94	5	1	3.0	250.8	1.0	3.0
5 ms	*Max Damage*	99	1	0	2.6	231.9	0.0	0.0
	Max Kill	96	4	0	2.6	233.6	0.0	0.0

C. Future work

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, or in another words trying to avoid to attack with a 20 damage weapon a unit with only 5 HP left. Even if GHOST is a mono-objective framework, we could craft new objectives by mixing already encoded ones, by simply applying a priority heuristics, like “maximize kills first, and consider maximizing damage as a tiebreaker”.

The current implementation only deals 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 [15] would make our experiments more accurate.

IV. TACTICS PROBLEM: WALL-IN

Up to our knowledge, tactics problems have not been intensively investigated for StarCraft. The exceptions are studies dealing with the wall-in problem like Certicky [16] and the wall-in solver by Richoux et al. [7]. As we already said, the latter provides the basis for the present work.

A. Problem statement and model

A classic tactic to defend a base in RTS games is to make a wall, *i.e.*, 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, which is easier to defend in case of invasion. In this section, we 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 for the width and h build tiles for the 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 is 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 go through. In this paper, we will call *significant gap* a gap that allows Zerglings, the smallest unit in StarCraft with 16×16 pixels, to cross a wall.

The wall-in problem has been first modeled in CSP by Certicky in [16]. Then, Richoux et al. proposed a different model, always in CSP, in [7]. 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 [7].

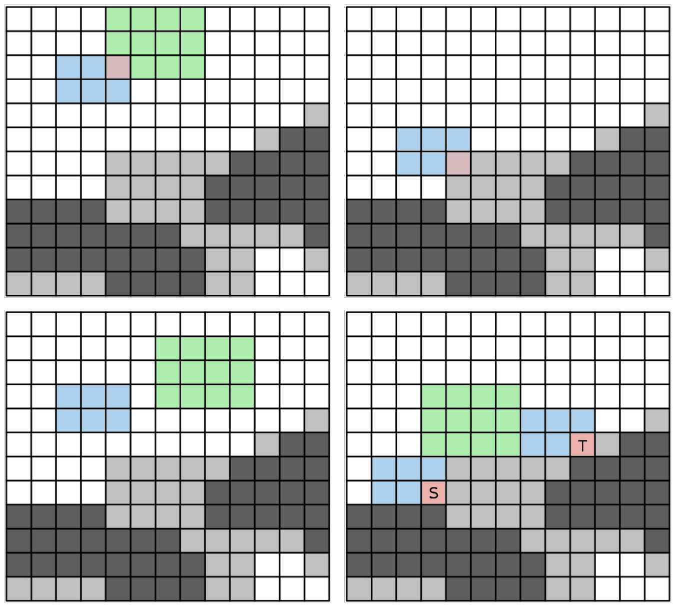

Fig. 2. Constraint are: Overlap (upper-left), Buildable (upper-right), NoHoles (bottom-left) and StartingTargetTile (bottom-right). Dark grey tiles represent unwalkable and unbuildable tiles. Light grey tiles are walkable but unbuildable tiles

For GHOST, the model of the wall-in problem is identical to the one in [7]:

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 model are illustrated in Figure 2. They are defined as follows:

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

structed 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 StartingTargetTile constraint to help the solver to find how to surround a chokepoint.

B. GHOST *implementation and results*

Like the target selection problem, we focus on the Terran race for our 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,
- **TechTree**: the required technology level in the game. In games like StarCraft, some buildings are unlocked after developing specific technologies. Therefore, 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 TechTree 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 TechTree 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. Satisfaction runs in Table III are GHOST runs

TABLE III
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 150 MS

Objective	Satisfaction Run	Optimization Run	% Solved (Opti)
Building	4.05	2.56	98.04%
Gap	1.32	0.03	97.50%
TechTree	1.99	1.35	97.54%

without any objective functions and with a satisfaction timeout of 160,000 μs, like in [7] with the difference that in Richoux et al.’s paper, they compiled 8 satisfaction runs of 20,000 μs (20 ms) 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 [7], optimization runs are slightly disfavored since their global timeout is 150,000 μs, thus 10,000 μs (10 ms) shorter than satisfaction runs. For optimization runs, x and y parameters where fixed to 20,000 and 150,000 respectively. We can see in Table III[5] that optimization runs lead to real improvements compared to satisfaction runs. This is particularly true with the Gap objective, where significant gaps are almost completely eliminated: 4,680 of 4,800 walls have been found (97.50%), and 4,527 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 [7], 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. Table IV shows the percentage of walls found in each

TABLE IV
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

of the seven maps from where chokepoints were extracted. These numbers correspond to pure satisfaction runs, since they do not differ significantly from 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 V
AVERAGE TIME OVER 20 RUNS TO FIND A SOLUTION

Chokepoint width (pixels)	Buildings needed	GHOST Avg. time (ms)	Clingo Avg. time (ms)
65	2	46.8	362.8
250	2	33.5	408.8

We also performed a comparison between GHOST and the state-of-the-art for the walling problem, in this case the work presented by Certicky [16]. First, we limited the number of possible buildings to be considered to the number of buildings in Certicky’s work (2 Barracks and 4 Supply Depots) and recorded the time to find a solution in two different cases: a small chokepoint (width of 65 pixels) and a big chokepoint (width of 250). In Table V[6], we can see the average results of both solvers. The times include the computation needed for setting the solvers parameters, the time to solve the problem, and the time to parse the solution (in the case of Certicky’s solution, we need to make an external call to Clingo solver). As we can see, GHOST is at least 7.8 times faster than Clingo

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

[6]Comparison experiments can be found at https://bitbucket.org/auriarte/bwta2/src

in our experiments. GHOST is faster in the wider chokepoint because the smaller one is a ramp and there is an overhead of computing the extremes of the ramp.

C. *Future work*

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 harder and related to the next section (it is somehow the wall-in problem combined with the build order problem).

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 with this feature in mind. The satisfaction part is executed in 20 ms, *i.e.*, it can be executed within one StarCraft frame in the fastest speed. 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 discussed more in detail in Section VII.

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

V. Strategy problem: The build order

The reader can find an extensive literature about build order planner and 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 [17] a strategy prediction and a build order adaptation system learning from replays, and in [18] Synnaeve and Bessière show a Bayesian model to predict the opponent build order.

A. *Problem statement and model*

A build order (BO) plan 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 focusing first on units in order to have an army as soon as possible.

A BO plan 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 requires *Air Weapons Upgrade level 1*, which requires itself a *Cybernetics Core*, which requires a *Gateway*.

The CSP model we propose for the BO plan problem is the following one:

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

B. 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 BO makespan. With this objective function, we had not 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 a BO, 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 where the user asks for a high number n of the same unit U, and none or a low number m of B.

Unlike the target selection problem, this time we need to integrate a simulator inside GHOST to emulate a game (without combat) and be able to compute the makespan of BOs. Thus, this simulator must emulate resource gathering, unit production (including workers), supply capacity and construction. Our simulator always tries to produce workers until reaching saturation (24 workers per base), as well as maintaining supply in order to never be "supply blocked" (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 BO 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 a bit too advantageous for the simulator. Then we closely analyzed some replays from Korean pro-gamers to refine our simulator settings, listed below in frames:

- Time to go build something: 74 frames (96 in [3])
- Time to go back gathering minerals after building something: 60 (0 in [3])
- Time to go from the base to mineral patches to start mining: 74 (0 in [3])
- Time for a worker to switch from mineral to gas: 74 (0 in [3])
- Mineral gathering rate: 0.045 mineral per worker per frame (like in [3])
- Gas gathering rate: 0.077 gas per worker per frame (0.07 in [3])

TABLE VI
OUR SIMULATOR EXECUTION MATCHED AGAINST THE PRO-GAMER BISU DURING THE FIRST 1900 FRAMES (*i.e.*, 80 SECONDS)

		Simulator in GHOST		Bisu	
Action	**Frames**	**Mineral**	**Supply (Used/Capacity)**	**Mineral**	**Supply (Used/Capacity)**
Simulator starts a probe	298	40.8	5/9	40	5/9
Simulator starts a probe	656	19.7	7/9	20	7/9
Simulator starts a probe	955	46.5	8/9	50	8/9
Simulator starts a pylon	1,119	-	-	-	-
Bisu starts a pylon	1,164	-	-	-	-
Simulator starts a probe	1,328	1.2	9/9	12	9/9
Simulator starts a probe	1,627	60.0	10/17	132	9/17
Bisu starts a gateway	1,731	-	-	-	-
Simulator starts a gateway	1,866	-	-	-	-
Simulator starts a probe	1,866	1.8	11/17	78	9/17

To be sure these parameters make our simulator realistic, we matched its execution against the first 1900 frames of games (*i.e.*, 80 seconds) played by Korean pro-gamers. Table VI gives an example of our simulator matched against the Protoss Korean pro-gamer "Bisu". Gas is not revealed since it remained 0 for both "Bisu" and the simulator during the first 1900 frames. 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 games are very similar, with a slight advantage to the simulator since its probe production is nearly perfect. For Table VII[7], GHOST has been run 10 times on each build order. We run experiments on two sets of build orders: those extracted from replays by Gabriel Synnaeve [19] and refined by Glen Robertson[8], and those extracted from top Korean pro-gamers.

In total, the first set is composed of 3,647 build orders: 768 Protoss versus Protoss, 2,043 Protoss versus Terran and 836 Protoss versus Zerg. Table VII shows that GHOST outperforms human BOs by far, with a mean of 544 frames of gain considering 10,000 frames BOs (about 7 minutes) and 394 frames of gain considering 7,800 frames BOs (about 5 minutes), all match-ups taken together.

We have also analyzed some replays played by top Korean pro-gamers: Protoss players "Bisu", "BeSt", "Violet" and "Cure"; Zerg players "Jaedong" and "sAviOr"; and Terran player "Flash". We have only downloaded 8 of them, giving us a set of 8 BOs to give to GHOST.

Results against these pro-gamers are shown at the very last line of Table VII. We can see that GHOST obtains better BOs than Protoss Korean pro-gamers listed above, with a gain of 689 frames for 10,000 frames BOs and 306 frames for 7,800 frames BOs. We have to lower the first result by emphasizing that pro-gamers are often already engaged in a fight before 10,000 frames and/or they are applying outside-the-book strategies, and thus minimizing the BO makespan may not be their first priority anymore. To a lesser extent, this is also true with 7,800 frames BOs.

Computation time is only 20 ms for satisfaction runs and 30 ms for the (global) optimization run. This means that GHOST can compute a highly optimized BO 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 BOs with the same makespan as pro-gamers in about 3.735 seconds (for build orders with a makespan up to 249s, *i.e.*, 5928 frames), giving thus a CPU time / makespan ratio of 1.5%. Considering GHOST is computing in average BOs with a makespan of 9250 frames within 30 ms. In the fastest mode, 9250 frames corresponds to about 388s; leading to a CPU time / makespan ratio of 0.007%.

Planning a BO is easier than a wall-in for GHOST because a BO plan 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 the wall-in problem, where even getting a non-optimized valid solution is hard.

C. *Future work*

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.

VI. MATCHING STATE-OF-THE-ART CONSTRAINT SOLVERS

One of GHOST's main goals is to provide a user-friendly and flexible interface to make the combinatorial problem modeling easier for non-specialists, using GHOST's inner solver as a blackbox to solve these models. A previous study has shown GHOST to be both robust and flexible [20], robust in the sense that GHOST inner solver shows good behavior to solve problems that it is not designed for, and flexible since proposing different models to the same problem only requires shallow modifications.

A. *State-of-the-art constraint solvers*

In this section, we compare GHOST inner solver performances with the state-of-the-art constraint solvers, namely

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

[8]scidrive.uoa.auckland.ac.nz/gameai/scdata/files.txt

TABLE VII
AVERAGE MAKESPAN OF HUMANS AND GHOST BOS IN FRAMES OVER 3647 GAMES. EACH CALLS OF GHOST LASTS FOR 30MS

Games till 10,000 frames					Games till 7,800 frames				
Match-up	**Humans**	**GHOST**	**% Solved**	**Gain**	**Match-up**	**Humans**	**GHOST**	**% Solved**	**Gain**
All	9,794	9,250	94.4	544	*All*	7,726	7,332	98.8	394
PvP	9,727	9,078	95.0	649	*PvP*	7,630	7,249	99.3	381
PvT	9,861	9,378	93.9	483	*PvT*	7,800	7,564	98.3	236
PvZ	9,692	9,097	95.0	595	*PvZ*	7,626	6,841	99.7	785
All pro	9,605	8,916	96.3	689	*All pro*	7,485	7,179	100	306

Opturion CPX[9] and *Gecode* [21]. We chose these two solvers for two reasons: first, both are able to parse MiniZinc code, a common language to model constraint optimization problems. Therefore, we only need to model a problem once and give this model to different solvers to measure their performances. Second, Opturion CPX and Gecode are well known state-of-the-art solvers, participation to the annual MiniZinc Challenges. Gecode won all gold medals in all categories (fixed, free and parallel) at the MiniZinc Challenges from 2008 to 2012. Starting from 2013, MiniZinc Challenges are composed of four categories: fixed, free, parallel and open[10]. Opturion CPX won two gold medals (fixed and free categories) and two bronze medals (parallel and open categories) in 2013, the four silver medals in 2014 and two gold medals (fixed and free), one silver medal (parallel) and one bronze medal (open) in 2015. Therefore, Gecode was the dominant constraint solver until 2012, and from 2013 Opturion CPX becomes and remains the current leading constraint solver.

However, these two solvers implement a complete algorithm, *i.e.*, an algorithm that explores the whole search space to find (and proof) the optimal solution. GHOST inner solver implements a local search meta-heuristics, *i.e.*, an incomplete algorithm. It is unable to prove the optimality of a solution, but in practice these algorithms are very efficient to quickly find an optimal or near-optimal solution. To match GHOST inner solver to other local search meta-heuristics, we also run experiments with *Oscar/CBLS* MinZinc interface [22]. We chose this solver again for two reasons: first, it is one of the rare solver programs that implements a meta-heuristics able to parse MiniZinc code. Other frameworks such as ORtools or EasyLocal++ can also parse MiniZinc code, but they are frameworks like GHOST, *i.e.*, they do not provide any executable, but a library to implement your own executable upon their solver. Second, Oscar/CBLS is a very recent solver (last release from late 2015), and the only local search-based algorithm listed on the MiniZinc software web page[11].

Rather than matching GHOST results with state-of-the-art solvers on the three problems presented above, we chose to compare these algorithms on a resources allocation problem. The reason is simple: the target selection and build order problems require a simulator to recreate the game environment, which is both complicated and time consuming for solvers we don't know so well, even for a simple simulator like the one used for target selection. The wall-in problem is actually quite complex to model through MiniZinc, with a lot of data related to the model variables (size of buildings, pixel gap size on each side, etc). Performances depend a lot on the quality of the MiniZinc model, where a good use of global constraints is critical. We lack the expertise in both MiniZinc and global constraints to assure to provide an efficient model for the wall-in problem. A model that is not well-optimized would have been a great disadvantage for other solvers. The MiniZinc model we use for the resources allocation problem is a slight modification of the `simple-prod-planning` model provided by MiniZinc authors, therefore we are sure to have a well optimized model for this problem.

B. Benchmark and experiments

Our resources allocation problem is the same as studied in [20]: given an amount of minerals, gas and supply, what units should we train to maximize the global damage per second (DPS) on ground units. In other words, what is the list of units you should train to maximize the sum of their ground DPS if we give you fixed stocks of resources. This is actually an instance of the multi-dimensional knapsack problem with three dimensions (one per resource type). The regular knapsack problem is well-known to be NP-complete, and its multi-dimensional version is even harder: unlike the original knapsack problem, there is no efficient polynomial-time approximation scheme starting from two dimensions (unless P=NP) [23].

To compare solvers on significant problem instances, we chose to optimize ground DPS for Zerg, Protoss and Terran factions with 20,000 mineral units, 14,000 gas units and 380 supply units. Although these values are unrealistic within a StarCraft game, they are large enough to challenge constraint solvers. Indeed, it is harder to match solvers if all of them return a solution within a couple of milliseconds.

Results matching GHOST inner solver with Opturion CPX and Gecode are compiled in Table VIII[12]. Since Opturion CPX and Gecode implement complete, deterministic algorithms, running them once on each problem instance is sufficient. That means runs on the same input are identical regarding solution quality and runtimes. For GHOST, we set the runtime to the lowest time (empirically found) where more than 50% of solutions are the optimal solution. We let Opturion CPX and

[9] www.opturion.com
[10] See www.minizinc.org/challenge.html for further details
[11] www.minizinc.org/software.html

[12] All experiment results can be found at github.com/richoux/GHOST_paper/tree/master/xp/other_solvers/resource and github.com/richoux/GHOST_paper/tree/master/xp/resource

TABLE VIII
SOLVERS' COMPARISON ON THE RESOURCES ALLOCATION PROBLEM

		Opturion CPX	**Gecode**	**GHOST** (100 runs)			**Oscar/CBLS** (10 runs)		
	Optimal DPS	**Runtime**	**Runtime**	**% Opt. Found**	**Mean DPS**	**Runtime**	**Mean DPS**	**Best DPS**	**Mean Runtime**
Zerg	11,400.00	200 ms	99,580 ms	59	11,387.70	80 ms	11,400.00	11,400.00	562 ms
Protoss	4,916.38	1,620 ms	-	54	4,907.70	1,300 ms	3,445.21	4,480.00	1,200 ms
Terran	6,632.73	3 h 19 min	-	53	6,619.30	130 ms	3,035.73	5,767.55	1,390 ms

Gecode solvers 6 hours for each problem instance to output a solution.

Results matching GHOST inner solver with Oscar/CBLS can be found in Table VIII. Notice that, like GHOST, Oscar/CBLS implements a stochastic algorithm. As such, we must run the same problem multiple times to compute a fair mean of the runtime and solution quality. Since Oscar/CBLS does not let us define a runtime timeout, we manually stopped the execution after one minute and then we collected, for each run, the runtime when it reached its highest DPS. This is the reason why we only have 10 Oscar/CBLS runs on each problem instance, against 100 runs for GHOST.

C. Results analysis

Table VIII clearly shows that GHOST inner solver outperform the state-of-the-art constraint solvers Opturion CPX and Gecode on our resources allocation problem instances. Gecode was only able to output a solution for the Zerg instance. For both Protoss and Terran instances, no solutions were found after 6 hours of computation. On the Zerg instance, Gecode found the optimal solution in 99.58 seconds, where GHOST found 59% of the time the optimal solution in 80 milliseconds. In average, GHOST finds a DPS of 11,387.7 where the optimal is 11,400. Thus, the average output quality from GHOST within 80 ms corresponds to 99.89% the optimal solution quality. Opturion CPX finds the optimal solution in 200 ms, *i.e.*, 2.5 times longer than GHOST needs to reach the optimal solution at least 50% of the time.

On the Protoss instance, GHOST found 54% of the time the optimal solution in 1.3 s with an average DPS of 4,907.7 where the optimal is 4,916.38, *i.e.*, 99.82% the optimal solution quality. Opturion CPX found the optimal solution in 1.62 s, thus slightly more time than GHOST needs to reach the optimal solution at least 50% of the time.

The real difference between GHOST and Opturion CPX occurs on the Terran instance. GHOST finds 53% of the time the optimal solution in 130 ms, with an average DPS of 6,619.3 where the optimal is 6,632.73, *i.e.*, 99.80% the optimal solution quality. Opturion CPX finds the optimal solution in 3 hours and 19 minutes!

Runtimes obtained on these three instances can be explained as follow: in StarCraft, 6 different Zerg and Protoss units can hit ground units, against 9 Terran units (without taking into account buildings such as Sunken Colony or Photon Cannon). This difference is sufficient to make the Terran instance search space considerably wider than the other two (about 1.57×10^{20} configurations for the Terran instance against 1.55×10^{13} and 1.33×10^{12} respectively for the Zerg and Protoss instances).

Although the Protoss instance has a smaller search space than the Zerg instance, all algorithms need more time to find a solution. This is because Zerg units' properties make the Zergling unit clearly more challenging to maximize the ground DPS, having by far the best DPS / cost ratio among Zerg units. Therefore, the strategy to maximize the Zerg ground DPS is trivial: just train as many Zerglings as resources allow. This is not the case with the Protoss instance where a composition of different units (for our instance, a mix of Zealots and Dark Templars) is required to reach the optimal solution. Notice that GHOST inner solver is faster to find a solution for the Terran instance rather than the Protoss instance, because Terrans have also a trivial optimal strategy: just produce as many Firebats as resources allow. However, for complete solvers, the search space remains too large to explore in order to prove the solution's optimality.

Table VIII shows that GHOST inner solver also outperforms the local search meta-heuristics implemented in Oscar/CBLS. Except for the Zerg instance, the best solution found by Oscar/CBLS was very far from the optimal solution, and finds in average poor-quality solutions usually within much longer time than GHOST to find a near-optimal solution. It requires about 562 ms to Oscar/CBLS to find the optimal solution of the Zerg instance, while GHOST finds the optimal or a near-optimal solution within 80 ms.

For the Protoss instance, GHOST needs 1.3 s to find the optimal (DPS = 4,916.38) or a near-optimal solution; in average, Oscar/CBLS requires 1.2 s to find a solution with a mean DPS of 3,445.21, that is, 70.08% of the optimal solution quality (against 99.82% for GHOST). The best solution found by Oscar/CBLS has a DPS of 4,480 (91.12% the optimal) while GHOST reaches 54% of the time the optimal solution.

Finally, for the Terran instance, GHOST computes the optimal (DPS = 6,632.73) or a near-optimal solution within 130 ms, while Oscar/CBLS takes 1.39 s in average to find a solution with a mean DPS of 3,035.73. This represents 45.77% of the optimal solution quality, against 99.80% for GHOST. The best solution found by Oscar/CBLS has a DPS of 5,767.55 (86.96% the optimal) when GHOST outputs 53% of the time the optimal solution.

We should also emphasize that Oscar/CBLS uses many cores to compute a solution[13], whereas GHOST inner solver remains sequential. Besides this difference, results compiled in Table VIII show that GHOST outperforms Oscar/CBLS both in terms of quality outputs and runtimes.

[13]Experiments have been conducted on an Intel i7 quad-core CPU, with 4GB of memory, under Ubuntu 14.04 64-bit.

VII. Discussion and conclusion

In this paper, we introduced GHOST, a combinatorial optimization framework to solve any (decidable) 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 II is now clear: looking for the absolute optimal solution may not be the best strategy for RTS games; this is confirmed by complete algorithms runtimes from Section VI. Fast meta-heuristics can output an "optimal enough" solution in some tens of milliseconds, and if no good enough solution has been found, the user can always re-run the solver on the next frame.

Some improvements we have in mind concern the implementation of a pause/resume system, that will allow GHOST to start a long computation and to hash it into small pieces fitting within one frame. 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 them 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. [9]). Indeed, this algorithm has been parallelized on a super-computer and it 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 the need to modify the already written classes. It is easy to implement and to include new problems in GHOST, and the authors highly encourage contributors to propose implementations of new problems to integrate into the library. We would like GHOST to be broadly used among both amateur and professional RTS AI developers. A proprietary C# version of GHOST has been transferred in favor of the game studio Insane Unity[14] for their MMORTS *Win That War!*[15] currently in alpha version. GHOST is used both for developing an adversary AI player, but also for making a taking-the-reins AI when the player is not connected, since this MMORTS is a persistent world.

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 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 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, up to our knowledge, none of these formalisms favor the design of efficient solvers. Thus, far beyond the scope of the present work, we would like to investigate on a new CSP formalism that could manage uncertainty efficiently.

[14] www.insaneunity.com
[15] www.winthatwar.com

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] G. Verfaillie and N. Jussien, "Constraint solving in uncertain and dynamic environments: A survey," *Constraints*, vol. 10, no. 3, pp. 253–281, 2005.
[7] F. Richoux, A. Uriarte, and S. Ontañón, "Walling in strategy games via constraint optimization," in *AIIDE*. AAAI Press, 2014.
[8] P. Codognet and D. Diaz, "Yet another local search method for constraint solving," in *SAGA*. Springer Verlag, 2001, pp. 73–90.
[9] 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.
[10] G. Synnaeve and P. Bessière, "A bayesian model for RTS units control applied to starcraft," in *CIG*. IEEE, 2011.
[11] A. Uriarte and S. Ontañón, "Kiting in RTS games using influence maps," in *AIIDE*. AAAI Press, 2012.
[12] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *AIIDE*. AAAI Press, 2012.
[13] 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.
[14] T. Furtak and M. Buro, "On the complexity of two-player attrition games played on graphs," in *AIIDE*. AAAI Press, 2010.
[15] D. Churchill, "Sparcraft: open source starcraft combat simulation," 2013. [Online]. Available: https://code.google.com/p/sparcraft/
[16] M. Čertický, "Implementing a wall-in building placement in starcraft with declarative programming," *arXiv*, 2013.
[17] 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.
[18] G. Synnaeve and P. Bessière, "A bayesian model for opening prediction in rts games with application to starcraft," in *CIG*. IEEE, 2011.
[19] G. Synnaeve and P. Bessière, "A dataset for StarCraft AI & an example of armies clustering," in *AIIDE*. AAAI Press, 2012.
[20] J. Fradin and F. Richoux, "Robustness and flexibility of GHOST," in *AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*. AAAI Press, 2015, pp. 9–14.
[21] C. Schulte and P. J. Stuckey, "Efficient constraint propagation engines," *Transactions on Programming Languages and Systems*, vol. 31, no. 1, pp. 2:1–2:43, 2008.
[22] G. Björdal, J.-N. Monette, P. Flener, and J. Pearson, "A constraint-based local search backend for minizinc," *Constraints*, vol. 20, no. 3, pp. 325–345, 2015.
[23] A. Kulik and H. Shachnai, "There is no eptas for two-dimensional knapsack," *Information Processing Letters*, vol. 110, no. 16, pp. 707–710, 2010.