Robustness and Flexibility of GHOST

Julien Fradin and **Florian Richoux**
LINA - UMR 6241, TASC - INRIA
Université de Nantes, France
julien.fradin@etu.univ-nantes.fr and florian.richoux@univ-nantes.fr

Abstract

GHOST is a framework to help game developers to model and implement their own optimization problems, or to simply instantiate a problem already encoded in GHOST. Previous works show that GHOST leads to high-quality solutions in some tens of milliseconds for three RTS-related problems: build order, wall-in placement and target selection. In this paper, we show the robustness of the framework, having very good results for a problem it is not designed for (pathfinding), as well as its flexibility, where it is easy to propose different models of the same problem (resource allocation problem). The goal of the paper is not to improve the state-of-the-art on these problems, but to use them as benchmarks to test GHOST properties.

Introduction

This paper shows the robustness and the flexibility of GHOST, a framework including a skeleton to implement our own constraint satisfaction problems and constraint optimization problems (CSP/COP), a constraint solver and a set of ready-to-use problems (Richoux, Baffier, and Uriarte 2015). This framework is available as a C++11 library under GNU GPL v3 licence, available at `https://github.com/richoux/GHOST`

GHOST deals with CSP/COP which are well-known to be NP-complete problems. To solve them, it includes a meta-heuristics algorithm to quickly approach an optimal solution. GHOST's purpose is to help the game developer coding an AI to get a high-quality solution of its problem within some tens of milliseconds, since it is not realistic to allocate much computation power dedicated to AI in a game.

To demonstrate our claim, we have included two new ready-to-use problems: the pathfinding problem and the resource allocation problem. The first one is a polynomial-time problem. GHOST's solver has no way to capture and exploit its intrinsic easiness, although it finds very good paths within 1 millisecond only, even if we make the original problem more complex by adding passable dangerous zones to avoid if possible. These good results show the robustness of the framework. GHOST is also flexible since we can implement very different models for the same problems

by changing less than 10 C++ lines only, while keeping excellent results. This is illustrated with resource allocation problems now included into the framework, with StarCraft: Brood War as a testbed.

GHOST: A General meta-Heuristic Optimization Solving Tool

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 solution seems better than any other 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 $c : V^k \to \{\top, \bot\}$ 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 $\bot$).

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.

For GHOST, we have chosen a meta-heuristic algorithm to be the heart of the solver, *Adaptive Search* from (Codognet

and Diaz 2001). The reason we have chosen a meta-heuristics is simple: to solve combinatorial and optimization problems while playing an 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, although it is not a well-known algorithm, it is one of the fastest meta-heuristics at the moment, up to our knowledge (Caniou et al. 2014).

It is essential to keep in mind that meta-heuristics are stochastic methods: two runs on the same problem instance may lead to two different solutions within the same runtime. This is why results in this paper are the average of 100 repeated experiments.

GHOST architecture

GHOST is implemented around five main C++ classes: `Variable`, `Domain`, `Constraint`, `Objective` and `Solver`. Implementing a CSP model means inheriting its own classes from the three first classes above. Implementing a COP model means making an additional class inheriting from `Objective`.

The solving process implemented in class `Solver` is composed of two main loops: the outer loop for optimization, containing the inner loop for satisfaction. The satisfaction part 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. The optimization part is triggered when an objective function has been implemented in a descendant class of `Objective`. 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. The optimization part also applies two optional post-process optimizations, one on the output of the satisfaction loop to "clean up" the raw solution found, one on the final output to apply to ad-hoc, last-minute optimizations if possible, leading to the solution returned by the solver.

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 is the satisfaction timeout x in milliseconds. It means that, after x ms top, we leave the satisfaction loop, certainly without a valid solution since the loop stops as soon as it finds a solution. The second parameter 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 shorter than y).

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.

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 solver, however the objective function can be dynamically changed between two calls of the solver. The choice of designing a mono-objective solver is pragmatic: multi-objective solvers are in general significantly slower, since they are dealing with more complex problems. Multi-objective are also making models more complex, which goes against one goal of GHOST: to propose a solver both easy to use and easy to extend by implementing new problems.

The pathfinding problem

The pathfinding problem is the very classical problem of finding a path (usually the shortest one) from a starting point s to a target point t. We would not debate if pathfinding is an AI problem or not; our approach is practical: numerous games including RTS games need pathfinding, and GHOST aims to be a well-spread library for helping game developers to solve AI and/or optimization problems. Including a ready-to-use pathfinding solution into GHOST can only be beneficial for users.

Problem statement and model

Before introducing our pathfinding model, we need two notions: the notion of walkable tile and the notion of marked tile.

In StarCraft like many other RTS games, a grid is applied on the map where each cell of the grid, also called tile, can be either walkable or not by ground units. Since pathfinding is relevant for ground units only (flying units do not encounter any obstacles so they can move straight toward their target), a path can only be composed of walkable tiles.

A marked tile is simply a tile where we associate a number. We will see below we only mark some specific tiles in order to find a path.

Therefore, we propose the following pathfinding model:

CSP model for the pathfinding problem:
Variables: Each tile on the map.
Domain: Binary: either the variable contributes to the path (value 1) or it does not (value 0).
Constraints: A unique constraint: does a series of n variables assigned to 1, going from the starting point and reaching the target point such that each variable is marked by a unique value in $[1, \ldots, n]$, exists?

If variables and domains in our model are clear, the constraint deserves further explanations. We did not write how we mark tiles yet, and it is the right moment to do it. Let's say that a variable assigned to the value 1 is a *selected variable*. Let's call *neighbors* the eight tiles around a given tile. We only mark tiles if they are selected, following this simple algorithm: Mark all selected neighbors of the starting tile with the number 1. Then, until all selected variables reachable from the starting tile are marked, repeat: for a variable
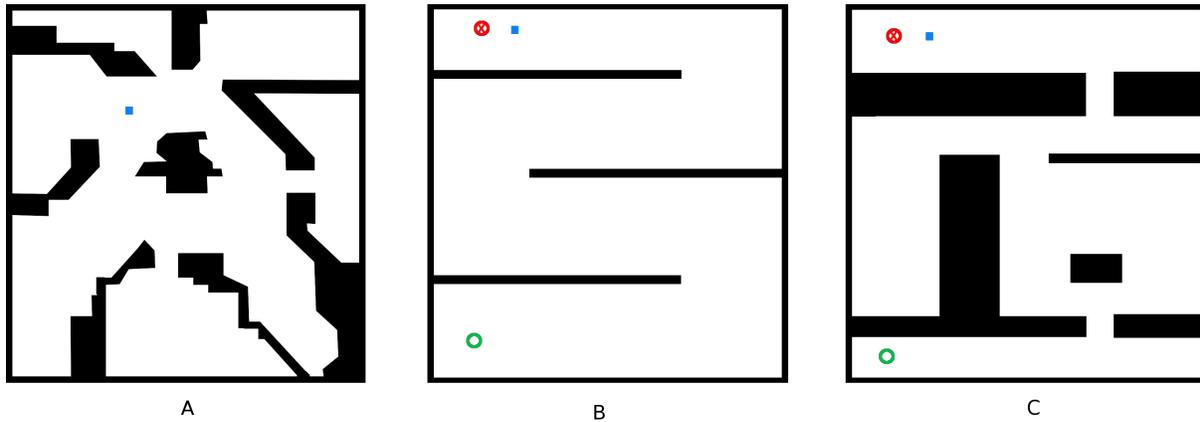
Figure 1: Three crafted maps A, B and C used for our path-finding experiments. The blue rectangle shows the walkable tile size. For B and C, the red crossed circle is the starting point and the green empty circle the target.

marked with the number k, marked all its unmarked selected neighbors with the number $k + 1$. To give a picture, we obtain waves of marks equal to 1, then 2, then 3 and so on, beginning from the starting point. This is actually a breadth-first search done only upon selected variables reachable from the starting point. Notice that this is done by GHOST: it is not a precomputed process.

What our constraint does is, intuitively, looking if a series of marks (1, 2, 3, ..., n) exists between the starting and the target points.

To give insights about how we model this problem, we had at the beginning two constraints: the first one to check if each tile except the starting and target tile has exactly two selected neighbors, the second one to check if we can reach the target point from the starting point passing by selected variables only. The unique constraint based on mark tiles in our current model assures these properties as well, but in a more efficient way.

For this problem, we have considered two objectives. The first one is obvious: find a path as short as possible from the starting point to the target one. The second one also try to find a path as short as possible, but taking also into account some dangerous zones (like tiles too close from the enemy army) we try to avoid without strictly discarding them.

GHOST implementation and results

For the pathfinding problem, we did not need to apply the classical scheme *satisfaction loop + optimization loop* in GHOST. Actually, only the satisfaction loop is important to find a rough path between the starting and the target point. Thus, smart post-processes can be applied to modify our path in order to fit the objective. They are computed very quickly and are powerful enough to avoid to repeat the satisfaction process: these post-processes leads to optimal or near-optimal paths according to the applied objective.

Thus, we ran 100 experiments over 8 instances from hand crafted maps illustrated by Figure 1. Maps B and C have their starting and target points fixed and look like small maze one can find in action games for instance. Experiments have
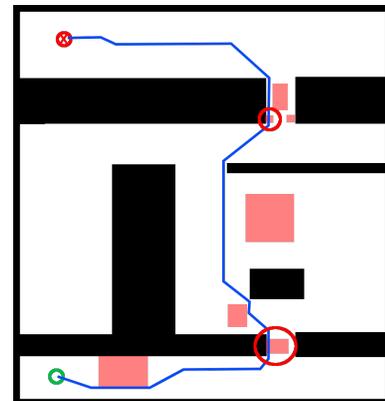

Figure 2: A computed path (in blue) in map C where dangerous zones are marked by red blocks. Parts where the path crossed dangerous zones are circled in red.

been conducted by running GHOST 100 times on map B and map C. Map A looks like more to an RTS map like we have in StarCraft. For this map, we used 6 different instances where the starting and target points where manually such that they are not in the same area, forcing a path to bypass walls or unwalkable blocks.

Since it was not necessary to exploit the optimization loop to solve this problem, we run our experiments with a time-out of 1ms for the satisfaction part and also 1ms for the optimization part, thus bounding the global runtime to 1ms top. Within 1ms, GHOST always find a optimal or near-optimal paths. Figure 2 shows an example of path found by GHOST while considering dangerous zones to avoid if possible, that is, if it makes longer the path reasonably only.

Both source code and experiments results for the pathfinding problem can be found at the following address: http://www.runmycode.org/companion/view/1391.

Discussion

Both versions of pathfinding presented above, with or without dangerous zones, are in P. This is easy to see: finding the shortest path from s to t can be done by marking all tiles the way we do (without taking into account the notion of selected tile), and then stop as soon as the target in reached. You get your path by following down a series of tiles marked from n to 1. This is done in linear time regarding the number of tiles in your grid. If you consider dangerous zones, you can add some extra tiles to artificially increase the distance between two dangerous tiles, according to the danger level. Then you run the same algorithm looking for the shortest path.

GHOST is designed to output in a couple of milliseconds a high quality solution to NP-complete problems, by doing effective descents into the search space landscape. It has no way to capture and exploit the pathfinding computational easiness, unlike ad-hoc polynomial-time algorithms such as A*-like algorithms. However, GHOST reveals itself to be very effective, finding optimal or near optimal paths within 1ms, even for more complicated pathfinding problems dealing with passable dangerous zones.

At the beginning of this project, we aimed to model pathfinding not for a single unit but for a group, keeping group coherence while moving, to avoid situations like faster units leaving slower ones way behind them. Usually, this is handled by flocking algorithms, potential fields or a mix of both (Hagelbäck 2015). Due to lack of time, we failed to find a satisfying CSP/COP model for this problem but this would be an interesting extension with direct applications for game developers.

The resource allocation problem

Resource allocation and management in RTS games such as StarCraft has been extensively studied, since it is at the heart of game mechanisms for this kind of games. Among others, the reader can find works about an agent applying resource-based strategies (Lemaître, Lourdeaux, and Chopinaud 2015), a resource production planner (Chan et al. 2007) and a decision-making process involved in resource management (Thiago, Ramalho, and Queiroz 2014).

In StarCraft, resources can be used to obtain units, buildings, upgrades and technologies. Two resources are needed for buying these: minerals and gas, both gathered by workers but differently (gas needs a special building to be extracted, unlike minerals). For training units, a third resource is required: the supply resource (also named "food"), limiting the number of units you can currently possess. Notice we could eventually consider a fourth kind of resource for training Zerg units: larvae. However we would not consider this in our model, since this resource (not universally acknowledged as a real resource) is race-specific.

Problem statement and model

In this section, we focus on the following problem: given an amount of minerals, gas and supply, what units should we train to maximize, say, the global damage per second (DPS).

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) (Kulik and Shachnai 2010).

We have tried two different models for this problem: a 0-1 multi-dimensional knapsack model (*i.e.*, an operational research-oriented model) and a constraint programming-oriented model. Let's call them Model 1 and Model 2, respectively.

In Model 1, variables represent individual units we can train. Thus, the number of variables must be large enough to cover all possible combinations our current resources allow. For domains, we just need binary values $\{0, 1\}$ to express the fact we plan to train the considered unit or not.

CSP model for resource allocation - Model 1:
Variables: Each **unit** we can train.
Domain: Binary: 1 means we plan to train the unit, 0 means we do not.
Constraints: A unique constraint to check if the sum of the desired units' costs does not exceed one or more resource stocks.

For Model 2, variables do not represent individual units but unit types we can train, such that each possible unit type is represented by a unique variable. Here, the value of a variable represents the number of units of the concerned type we plan to train. Thus, domains must to be large enough to cover all possible combinations.

CSP model for resource allocation - Model 2:
Variables: Each **unit type** we can train.
Domain: $\mathbb{N}$ (or a sufficiently large subset) representing the number of units of the considered type we will train.
Constraints: Same as Model 1.

Actually, we also have tried a third model which was a variation of Model 2: variables were also unit types, but with the noticeable difference that several variables can concern the same unit type. Domains were of the same nature than Model 2 but smaller. The idea was the following: instead of having a variable for Terran Marines for instance, with a domain $[1, \ldots, 30]$, this model would accept, say, three variables for that unit type with domains $[1, \ldots, 10]$. This was to force some diversity in the solution, avoiding the solver to fall into a greedy behavior, training for instance 30 Marines to the detriment of other units. In fact, experiences shown it does not lead to a better diversity than Model 2, and since this third model never output results of higher quality than the second model, we have simply discarded it.

GHOST implementation and results

For this problem, we consider two objectives: ground DPS and air DPS, that is respectively, the damage per second we can inflict on ground enemies and on flying enemies. Indeed in StarCraft, many units cannot attack both kind of units or use different weapons against ground and flying targets.

Table 1: Mean of 100 (1,4)-runs for the resources allocation problem with 500 mineral units, 200 gas units and 9 supply units

		Model 1				Model 2			
		DPS	minerals	gas	supply	DPS	minerals	gas	supply
Protoss	ground	97.9	50.2	0.0	1.0	98.0	50.0	0.0	1.0
	air	64.0	0.0	0.0	1.0	64.0	0.0	0.0	1.0
Terran	ground	116.7	48.2	25.0	0.1	143.1	65.0	9.0	0.3
	air	94.7	26.5	8.0	0.0	95.9	50.0	1.5	0.0
Zerg	ground	212.7	11.0	118.5	0.1	270.0	50.0	200.0	0.0
	air	94.7	56.0	52.0	3.0	96.0	50.0	50.0	3.0

Table 2: Mean of 100 (1,4)-runs for the resources allocation problem with 1000 mineral units, 700 gas units and 19 supply units

		Model 1				Model 2			
		DPS	minerals	gas	supply	DPS	minerals	gas	supply
Protoss	ground	211.3	29.5	161.5	1.4	236.2	36.0	11.5	3.0
	air	141.4	40.5	23.0	2.0	148.0	25.0	50.0	1.0
Terran	ground	215.3	57.2	193.0	0.4	326.8	56.0	221.0	0.1
	air	200.9	39.2	103.7	0.0	205.9	91.7	144.7	0.0
Zerg	ground	335.8	20.0	303.7	1.7	567.9	53.5	700.0	0.0
	air	169.9	47.2	231.7	4.8	208.0	25.0	375.0	6.0

Unfortunately, we did not find in the literature experimental results we could compare with. Some articles show results such as win rate percentages to evaluate resource-oriented agents, but we think too many factors are in game to estimate how good the resource planning is by just looking if a bot win or loose.

We could also compare our results with other constraint-based solvers on our resource allocation problem instances, but we do not think it is a pertinent approach. GHOST should not be reduced to a solver: it is an entire library aiming first and foremost to help game developers modeling and implementing (and then solving automatically) their own combinatorial optimization problems, or to let them simply reuse an available model in the library. We have the feeling that comparing GHOST with CSP/COP solvers like one usually does in constraint programming papers is not scientifically relevant. However, it would have been relevant to compare GHOST results on resource allocation planning for RTS games with other Game AI methods, if such results are available.

Tables 1 and 2 show the average over 100 runs of the resource allocation problem with 500 minerals, 200 gas and 9 supply (medium problem instance), and 1000 minerals, 700 gas and 19 supply (large instance) respectively, considering all anti-ground and anti-air units for each race. All experiments have been done given 1ms for the satisfaction timeout and 4ms for the optimization timeout. Therefore, the total time of each run is 4ms.

Although we did not compare GHOST results with other solvers on this problem, we solved instances from Tables 1 and 2 with a complete solver, GNU Linear Programming Kit (GLPK), to compute the optimal solution for each instance. This is impossible to compute with GHOST because its solver is a meta-heuristics, i.e., an algorithm which does not scan the full search space and then cannot certify the op-

timality of a solution. However within 4ms and in average among 100 runs, GHOST finds solutions very close to the optimum. Table 3 shows optimal DPS values found by GLPK, and matches them with the average DPS computed by our library.

Both source code and experiments results for the resource allocation problem can be found at the following address: http://www.runmycode.org/companion/view/1391.

Discussion

We can see Model 2 always outperforms Model 1. This is not surprising, since Model 2 is a more compact way to express this problem, leading to greatly smaller search space. To give an example: consider we can train 10 units from 3 unit types (so 30 units in total). Model 1 leads to 30 binary variables, so a search space of size 2^{30}, which is a bit more than one billion combinations. For Model 2, we have 3 variables only, each taking a value in the range $[1, \ldots, 10]$. This leads to 3^{10} combinations, i.e., a bit less than 60,000. Some methods like SAT solvers are optimized to find solutions in Boolean domains, but this is not the case of GHOST's solver, which implements the Adaptive Search meta-heuristic designed to handle any kind of finite domains.

Besides GHOST's efficiency on the resource allocation problem, the goal of this section was to show GHOST's flexibility. Indeed even if Models 1 and 2 differ greatly by their variables and domains nature, but in the code they only differ from 8 C++ lines in the implementation of `Domain` class. Remaining code is unchanged.

An obvious extension can be modeling and implementing other objectives. Focusing on DPS only is very limiting since it does not take into account the diversity one should expect in an RTS game army. This can be done by only modifying `Objective` class's descendants, without modifying

Table 3: Optimal DPS versus average GHOST DPS over 100 runs
P=Protoss, T=Terran, Z=Zerg, m=medium instance (500/200/9), l=large instance (1000/700/19), g=ground, a=air

	Pmg	Pma	Plg	Pla	Tmg	Tma	Tlg	Tla	Zmg	Zma	Zlg	Zla
Optimal	98.0	64.0	241.4	148.0	149.2	96.9	331.6	207.2	270.0	96.0	570.0	208.0
GHOST	98.0	64.0	236.2	148.0	143.1	95.9	326.8	205.9	270.0	96.0	567.9	208.0
wrt opt.	100%	100%	97.8%	100%	95.9%	99.0%	98.5%	99.4%	100%	100%	99.6%	100%

the variable, domain and constraint parts of both the model and the implementation.

Future work

To conclude, we have shown GHOST to be both robust, able to deal with problems it is not designed for (*i.e.*, problems in P, whereas its solver is designed to tackle NP-complete problems) and flexible, showing that changing 8 C++ lines only is sufficient to express very different models to the same problem.

We think the next step is to broadcast GHOST and make it more available, reachable to game developers. A C# version exists (see `https://github.com/richoux/GHOST_C_sharp`), with a lighter, simpler structure than the C++ version, making it more user-friendly in particular for developers with no prior knowledge in constraint problems modeling. The first step would be to bring these simplifications into the C++ version, and reciprocally to include the pathfinding problem and the resource allocation problem into the C# version. Then, we would like to propose the C# GHOST as a Unity plugin, to reach the broadest game developer audience as possible, and to make it really easy to use within the famous game engine.

Finally, a proprietary C# version of GHOST is currently under a technology transfer process for the game company Insane Unity (`http://www.insaneunity.com`) for their MMORTS *Win That War!* in alpha version. GHOST would be 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 would be a persistent world.

References

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

Chan, H.; Fern, A.; Ray, S.; Wilson, N.; and Ventura, C. 2007. Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. The AAAI Press.

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

Hagelbäck, J. 2015. Hybrid pathfinding in StarCraft. *IEEE Transactions on Computational Intelligence and AI in games*.

Kulik, A., and Shachnai, H. 2010. There is no eptas for two-dimensional knapsack. *Information Processing Letters* 110(16):707–710.

Lemaître, J.; Lourdeaux, D.; and Chopinaud, C. 2015. Towards a resource-based model of strategy to help designing opponent AI in RTS games. In *Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART)*. LNCS Springer.

Richoux, F.; Baffier, J.-F.; and Uriarte, A. 2015. Ghost: A combinatorial optimization solver for rts-related problems. Working paper: https://hal.archives-ouvertes.fr/hal-01152231.

Thiago, S.; Ramalho, G.; and Queiroz, S. 2014. Resource management in complex environments: Applying to real time strategy games. In *Proceedings of the Brazilian symposium on Computer Games and Digital Entertainment*.