Hybrid Black-box and model based optimization

Kevin Yu

Introduction

Optimization problems can be found in many disciplines, such as business, engineering, and biology. Some of these problems can be modelled using formal methods or computer languages, while the inner workings of others are too complex or unknown to be modelled. At the intersection, there are hybrid problems that possess both modellable and Black-Box components.

While there has been extensive research into approaches to solving both kinds of problems individually, there has not been much investigation into effective techniques to optimize problems that lie at this intersection.

Hence, the question that this literature review seeks to explore, and the research project to answer is:

Are there effective ways to combine the approaches to model-based and black-box optimization, to solve problems that are a hybrid of modellable and Black-Box ?

Introduction to modellable problems

As a motivating example, a classic modellable optimization problem is the nurse scheduling problem.

This problem consists of assigning a number of nurses to a number of shifts over a time period, and also subject to constraints such as nurses required per shift and each nurse's availability. (TOCITE: Google OR)

- Each day is divided into 3, 8 hour shifts; a day, evening and night shift
- Each day, every shift is assigned to a single nurse, and no nurse works more than one shift per day
- We want to make the shift distribution as equal as possible.

One way to solve this problem is to define a model, and let backend solvers solve the problem:

```
int: numNurses = 4;
int: numDays = 3;
int: numShifts = 3;
set of int: allNurses = 1..numNurses;
set of int: allDays = 1..numDays;
set of int: allShifts = 1..numShifts;

array[allNurses, allDays, allShifts] of var bool: roster;

constraint forall(d in allDays, s in allShifts)(
  let {var int: sm = sum(i in allNurses)(roster[i,d,s]);}
    in sm = 1
);

constraint forall(n in allNurses, d in allDays)(
  let {var int: sm = sum(i in allShifts)(roster[n,d,i]);}
    in sm <= 1
);

constraint forall(n in allNurses)(
  let {var int: sm = sum(d in allDays, s in allShifts)(roster[n,d,s]);
```

```
  } in sm >= minShiftsPerNurse /\ sm <= maxShiftsPerNurse
);
solve satisfy;
```

Figure 1 | A MiniZinc model for the nurse scheduling problem (Note that some parts of code have been omitted for brevity)

When run with the output code (omitted), the following would be outputted:

```
Nurse 1 works shifts: 3 on day 1; 2 on day 2; 3 on day 3;
Nurse 2 works shifts: 2 on day 1; 1 on day 2;
Nurse 3 works shifts: 3 on day 2; 1 on day 3;
Nurse 4 works shifts: 1 on day 1; 2 on day 3;
```

Figure 2 | Output of the nurse scheduling problem in MiniZinc

Above:

- 1 is a morning shift
- 2 is an evening shift
- 3 is a night shift

Introduction to unmodellable, Black Box problems

An example of an unmodellable problem is traffic light scheduling based on real world data. (TOCITE: https://www.sciencedirect.com/science/article/abs/pii/S1568494619301346) Within a road network, this problem aims to optimize the light cycle programs of traffic lights, to improve traffic flow and reduce pollution. As this problem involves a large volume of data and many variables, such as each individual moving car, it cannot be adequately modelled.

Instead, problems like traffic light scheduling using real world data often resort to simulation. (TOCITE: Reliable simulation-optimization of traffic lights in a real-world cities) Given some input such as each car's starting position and path, as well as the traffic light cycle program, it will output data on the performance of the simulation. Output data can include values such as the number of vehicles that reach their final destination, or the total travel time and waiting time for vehicles that complete their journey. These values are all valid candidates for optimization. The simulation function can be thought of as a Black Box function that can be optimized on its output variables.

Introduction to hybrid problems

At the intersection of modellable and unmodellable problems, there are hybrid problems. Usually, there are extensions of unmodellable black-box problems, where the constraints of the input to the problem can be reliably modelled. For example, in the above traffic light simulation problem, a constraint that can be introduced, is that the total wait time of any car should not exceed more than s seconds.

The following sections will include: TODO

Substantive Literature Review

Background

Background will introduce some concepts and definitions that will be referred to within this literature review.

Combinatorial satisfaction problems (CSP)

Each optimization problem can be built upon a base combinatorial satisfaction problem, therefore we will first define this. A combinatorial satisfaction problem is defined as a triple (X, D, C) where:

- X = {X_1,...,X_n} is a set of variables (also known as decision variables)
- D = {D_1,...,D_n} is a set of domains for each variable in X
- C = {C_1,...,C_n} is a set of constraints, where each constraint C_i is a relation defined over a subset of X, specifying a set of allowable combinations of values for the variables.

Definitions

- Assignment: A mapping of variables to their individual values
- Candidate solution: An assignment such that each variable X_i lies within its corresponding domain D_i
- Search space: The set of all candidate solutions
- Solution: A solution such that all constraints in C are satisfied

When there is at least one solution to a CSP, the CSP can be described as *feasible*. If not, then it is *infeasible*

Constraint optimization problems (COP)

A combinatorial satisfaction problem can be extended to be a combinatorial optimization problem by adding an objective function, that has input the set X of constraints (with domain D) and outputs a real value, known as the cost, or score.

Formally, an optimization problem can be defined as:

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && g_i(\mathrm{x}) \leq 0, \mathrm{i} = 1,...,\mathrm{m} \\ &&& h_j(\mathrm{x}) = 0, \mathrm{j} = 1,...,\mathrm{n} \end{aligned}$$

where

- $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function to be minimized
- $x \in \Omega \subset \mathbb{R}^n$, the parameter domain, where n is the number of parameters
- $g_i(\mathrm{x}) \leq 0$ are inequality constraints
- $h_j(\mathrm{x}) = 0$ are equality constraints
- $m \geq 0$ and $p \geq 0$

The above definition applies to both minimization and maximization problems, as a maximization problem can always be transformed into a minimization problem by negating the entire objective function.

Modellable problems

For problems that have an explicit problem formulation, modelling can be done using computer software or modelling languages. Modelling on software such as Matlab or MiniZinc TODO:CITE encapsulates the problem itself; rather than describing *how* exactly to solve the problem, it describes *what* the problem is. This description can then be sent to a backend solver which can find a solution.

This is in contrast to traditional methods where a specific imperative algorithm is devised to solve a problem.

To demonstrate its capabilities, here is an example taken from the MiniZinc Handbook [1]; imagine the scenario where we need to bake some cakes for a fete. We can make two sorts of cakes: A banana cake which takes 250g of self-raising flour, 2 mashed bananas, 75g sugar and 100g of butter, and a chocolate cake which takes 200g of self-raising flour, 75g of cocoa, 150g sugar and 150g of butter. We can sell a chocolate cake for $4.50 and a banana cake for $4.00. And we have 4kg self-raising flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa. The question is how many of each sort of cake should we bake for the fete to maximize the profit.

On one hand, we can devise an algorithm to solve this problem. This problem can be solved using the Simplex algorithm, however this would be fairly involved, as one would have to know the algorithm, its caveats, as well as implement it.

On the other hand, we can simply model the problem as it is, and allow one of many backend solvers to solve the problem for us. This enables us to focus on defining the problem, rather than coming up with solutions to it. See below for a model of this problem in the Constraint Programming modelling language MiniZinc.

```
% Baking cakes for the school fete

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b  <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;

% maximize our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = \(b)\n",
         "no. of chocolate cakes = \(c)\n"];
```

Figure 3 | A MiniZinc model for the school fete problem

The above MiniZinc model mirrors the definition of a COP as defined above:

- The parameter domain in the COP definition is equivalent to the `var 0..100: b;` variable declarations
- The constraints in the COP definition is equivalent to the `constraint` declarations
- The objective function is equivalent to the `solve maximize 400*b + 450*c` declaration

Introduction to optimization paradigms and modelling languages

There are many paradigms for optimization, such as constraint programming, integer programming, or satisfaction solving. Each technique is suited to its own type of problem. For example, constraint programming is suited to CSP problems while integer programming is suited to COPs where con-

straints and objective function are linear. Each optimization technique also has many solvers, which take as input a model written in the solver's language or library. Constraint programming has libraries such as Gecode, where models can be written in C++, with the use of Gecode's library, while Integer Programming has GLPK, where the MathProg language must be used.

While there are many approaches to model based optimization, in this literature review we will be focusing on constraint programming in particular. This firstly due to its flexibility, being able to model a variety of combinatorial optimization and satisfaction problems. Further, there is a mature and robust constraint modelling language - MiniZinc [2] which we can extend to conduct our research. More justification as to why MiniZinc was chosen will be provided in the following sections.

Introduction to constraint programming

In constraint programming, also known as constraint optimization, the user models the problem by defining the decision variables, their domains, and then stating the constraints on decision variables. Constraint Programming approaches are adept at solving CSPs such as the N-Queens problem TODO:CITE, since they directly work to resolve constraints on the problem. However, as a CSP can easily be extended to become a COP by adding an objective function, constraint programming can also be extended to solve problems within this domain. A simple way to do this is to evaluate the objective function at each solution in the search space and track the solutions which produce the minimum objective function value. Constraint problems are usually solved via complete search, with the aid of backtracking and constraint propagation to make the search more efficient. Note that complete search doesn't not mean exploring the entire search space, rather that that the solution produced by the solver is *provably* optimal, as often suboptimal areas of the search space can be logically removed.

TODO: Could talk a bit about branch and bound

MiniZinc TODO:TOCITE

MiniZinc enables users to model problems in a constraint programming paradigm, by defining the decision variables, domains, constraints. Solutions can then be found to either satisfy the model, or optimize it if an objective function is further supplied.

MiniZinc is different to the aforementioned modelling languages as it is *solver independent* - A single MiniZinc model can be mapped and solved by multiple different solvers, such as Gecode, or OR-Tools (An Integer Programming solver). This allows much more flexibility to test out different solving techniques for a given model, without having to rewrite the problem in each solver's unique language/ library. To map to different solvers, MiniZinc first compiles the model into a *FlatZinc* file, which is then passed to a solver via a solver specific *backend*.

Unmodellable problems - Black-Box Functions

In contrast, for a Black-Box problem, the nature of f or the domain of x is difficult or impossible to exploit. Most commonly, such Black-Box functions are found in computer simulations of complex real world scenarios.

Hybrid Black-Box and Modellable problems

At the intersection of these two types of problems are hybrid Black-Box and modellable problems. In such problems, the decision variables and their constraints can be modelled, but the internals remain a Black-Box function. This literature review aims to explore current model based optimization and black-box based optimization approaches, and how the merits of each can potentially be used to solve the problems at this intersection.

Black-Box problems

The main problems we aim to solve in this study are Black-Box problems with constraints that can be modelled. Some common areas where Black-Box problems arise are:

- Simulations of complex real world scenarios
- Optimization of hyper-parameters in Machine Learning research
- Assembly code

In the following section, we will be examining simulation based Black-Box problems by example.

Simulations of complex real world scenarios

In optimization of real world processes, simulations are used to analyze the scenario under different input variables.

A prominent example is Wind Farm Layout Optimization (WFLOP). This involves positioning wind turbines optimally within a designated wind farm area. The difficulty of this problem arises due to the "wake" of turbulence that individual turbines create, which reduces the wind speed, and consequently energy production of those turbines that are positioned downstream. [3]

From this scenario, a simplified version of this optimization problem can be formulated as follows; given an amount of farmland, and for a set of positions; the wind speed, direction and probability of this combination occurring, as well as the mathematical models for wake and related physical phenomena, determine an optimal positioning of Wind Turbines to maximize the power output of the wind farm.

Due to the complex relationship between turbines and physical forces, as well as the mathematical models involved in calculating an individual turbine's power output, the actual problem can only be represented through simulation. The representative Black-Box function is one where the inputs are the positions of each individual wind turbine, and the output is a measure of the power output that is produced by the wind farm in this particular arrangement.

In the formulation of this problem as described above, there are several constraints that come into play. The first constraint is that the turbines must lie within the area of the farm.[4] For a rectangular farm with length l and width w this constraint can be modelled as:

$$0 \leq x_i \leq l \land 0 \leq y_i \leq w, \forall i$$

The second constraint that must be taken into consideration is that any two turbines must not be within a certain distance MR of each other, where M is a proximity factor and R is the rotor radius:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq \text{MR}, \forall i \forall j$$

In real scenarios these two constraints are usually known and modellable. As the cost of simulation is high and there is usually an imposed limit on the number of simulations that can be made, WFLOP optimization algorithms aim to guarantee that solutions sent to the simulator are valid, rather than have the simulator report that the solution is infeasible.

This is either done by introducing a feasibility check in the objective function evaluation routine as seen for Genetic algorithms and Simulated annealing approaches in [3], or during the neighbor finding phase in local search techniques explored in [4].

Some approaches described in [3] use a discrete solution space - turbines can only be placed at specific points on a grid. This is an example of how constraints can be embedded into the problem formulation itself, such that they do not need to be explicitly dealt with. This however, does have some drawbacks. Since the positioning of each turbine is being restricted, many potentially better solutions are not considered.

Additionally, the WFLOP can be extended to be more realistic by adding more constraints. Current works do not factor in the installation phase, such as the roads (or lack thereof) connecting turbines which are used to transport construction material [3]. If there are no roads between two turbines, they will need to be built at an additional cost, and the roads that can be built also often carry additional constraints, for example that they must be built along crop boundaries. Therefore another objective that can be formulated is to build a connected network of roads between turbines, subject to constraints and while minimizing the cost of building roads. In this way, the wind farm problem can extended to a multi-objective optimization problem, where we are trying to maximize both the placement of the wind turbines (a Black-Box function), and minimize the cost of building roads between the turbines during construction (a problem that can be adequately modelled).

Approaches to Black-Box Optimization

On large and complex Black-Box problems, a complete search of the entire search space is simply impractical, often requiring too much compute and/or time. Therefore the focus of many Black-Box optimization techniques becomes to find a good enough solution in a reasonable amount of time/ resources. Many Black-Box optimization techniques cannot provably consider the entire search space, and therefore cannot guarantee optimality.

There are numerous approaches to Black-Box Optimization, such as direct search and hybrid optimization. In the following section we will examine local search, and some heuristic optimization methods.

Local Search methods

Local search is an optimization technique that is used to find a solution to a problem by exploring the solution space, iteratively improving a candidate solution. In local search, a starting solution s is chosen. At each step, the neighborhood of s is considered. From this neighborhood, a new solution s' is considered. It is accepted if it improves the objective function. This process continues until no further improvement can be made, or until a stopping condition is met.

A *heuristic* is utilized by a local search method to select a neighborhood of the current solution. In general, the neighborhood selection is a tradeoff between *exploration* and *exploitation*.

Exploration is defined as the ability to reach and test candidate solutions that are outside the local neighborhood of the current solution. This helps the algorithm escape from local minima. *Metaheuristics* are defined as techniques to escape these local minima.

Exploitation is defined as the ability to search in the local neighborhood of the current solution. This is done to refine the solution to the local minima.

Simulated Annealing [5]

A popular meta-heuristic used in local search is Simulated Annealing. In each move, a neighbor s' is chosen at random, and we make the move to s' with a certain probability. The probability is 1 if s' is better, and less than one if the move is inferior. This probability depends on T, which is temperature. At the start when the temperature is high, it is more likely that worse solutions are chosen. The temperature is gradually lowered, and consequently the probability of choosing worse solutions is also lowered. Towards the end, solutions chosen are almost always improving ones.

Can frame them as closely related search The techniques for black box are used with modellable AI. Its hybrid in that its local search with modellable. Its not hybrid as in black box with modellable, which is what we want.

Local search you can do in any function Talk about how this functions in Black Box.

Hybrid model and local search based approaches to optimization

TODO: Add motivation here

Large neighborhood Search (LNS) [6]

Large neighborhood search was one of the first techniques combining complete search techniques like constraint programming with local search based methods. The pseudocode for a simple LNS formulation is as follows:

1. Start with a feasible solution (using Constraint Programming)
2. Select a neighborhood (using Local Search)
3. Optimize within the neighborhood (using Constraint Programming)
4. Repeat from step 2 until end condition is met

The above process utilizes the strengths of each type of search technique. Since Constraint Programming is good at finding feasible solutions, is it used to find the initial solution.

In comparison to complete search techniques, Local Search is more suited to exploring the solution space, so it is used in the algorithm to find a suitable neighborhood to exploit. A neighborhood is chosen by fixing a subset of the variables, and this choice of subset is often problem-specific.

Choosing a neighborhood constricts the search space to a manageable size so that constraint programming can then be used to optimize the unset variables. This makes use of the fact that given a small enough search space, constraint programming is much better at finding the optimal solution, since it can guarantee optimality, as compared to local search.

Constraint Based Local Search (CBLS) [7]

Constraint based local search is a variation on local search that adapts ideas from CP. Similar to constraint programming, the constraints on the problem are used to guide the local search towards an satisfactory solution. Like in other variants of local search, the search still starts with a configuration which may or may not satisfy the constraints. The algorithm at each iteration identifies the violated constraints, and variables causing the most violations, and forms a neighborhood from this. It then chooses the neighboring solution that leads to the greatest reduction in constraint violations.

While the current solution is in a non-satisfying state, the algorithm will follow these steps to search for a satisfying state:

1. Calculate the constraint violations that each variable creates
2. Pick the variable x (that is not in the tabu list) that is contributing the most to the constraint violations
3. Set all variables except x, forming the neighborhood
4. Calculate the degree constraint violations at each neighbor. If a better one is found, then move to this neighbor.

If there is no better assignment, then add x to the tabu

This is continued until a satisfying solution is found, or until a maximum number of iterations is reached.

Note that the quantification of violation contribution depends on the type of constraint being violated. For example, for an equality constraint, the amount is $|x_1 - c_1|$ where the constraint is $x = c$. Therefore, the further away the variable x is from what it should be, c, the more likely it is that x will be chosen to be re-assigned.

Bibliography

[1] P. Stuckey, K. Marriott, and G. Tack, “Minizinc handbook.” https://www.minizinc.org/doc-2.5.5/en/modelling.html?highlight=cake#an-arithmetic-optimisation-example

[2] Nethercote, Peter J., et al., “Minizinc: towards a standard cp modelling language,” in *Princ. Pract. Constraint Program. -- CP 2007*, Berlin, Heidelberg, 2007, pp. 529–543.

[3] M. Samorani, “The wind farm layout optimization problem,” Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-642-41080-2_2

[4] M. Wagner, J. Day, and F. Neumann, “A fast and effective local search algorithm for optimizing the placement of wind turbines,” *Renewable Energy*, vol. 51, pp. 64–70, 2013, doi: https://doi.org/10.1016/j.renene.2012.09.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0960148112005745

[5] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Sci.*, vol. 220, no. 4598, pp. 671–680, 1983, doi: 10.1126/science.220.4598.671.

[6] P. Shaw, “Using constraint programming and local search methods to solve vehicle routing problems,” in *Princ. Pract. Constraint Program. --- Cp98*, Berlin, Heidelberg, 1998, pp. 417–431.

[7] P. Van Hentenryck, and L. Michel, *Constraint-Based Local Search*, 2005.