Solvers for Optimization Problems in Operations Research

Ali Khudiyev

May 2021

Contents

1 Introduction

There are many problems related to operations research which are considered the NP type which means this problems are not solvable in a polynomial time or have non-deterministic algorithm(s) which works in a polynomial time. Such problems include Knapsack Problem, Vechile Routing Problem(VRP)[15] or Travelling Salesman Problem(TSP), Graph Coloring[2], Sum of Subsets[13], Hamiltonian Cycle[3] and so on. These problems have (deterministic) algorithms which work in an exponential time rather than polynomial. Some of such problems are usually explicitly linked to a problem which generalizes all of them, and if the generalized problem is solved then any of those problems can be solved in polynomial time. We can also try to show that two problems are strongly linked with each other by proving that polynomial time solution of either of those leads to the polynomial solution of the other one. There is also another problem called Boolean Satisfiability Problem[1](SAT or BSAT) which generalizes problems such as Knapsack Problem.

”Verification versus Solution time relationship” has always been an unanswered question up until now; in fact, **NP=P?**[10] is one of the most popular questions. There are some problems that we can verify a proposed solution in a ”polynomial time” or relatively quickly but cannot solve that quickly(SAT or Knapsack problem) and there are also some problem that we cannot verify whether a proposed solution is correct or not in a polynomial time(i.e. the best move in chest, an optimal solution for Vehicle Routing Problem). However, there are also terms like **NP-complete**[8] and **NP-hard**[9] which refer to the different type of problems. **NP-complete** problems are the ones which cannot currently be solved in polynomial time, however, can be verified in polynomial time and can be used to solve any other problem. SAT or Knapsack problem can be given as examples of this class. **NP-hard** problems are the ones which cannot currently be either solved or verified in polynomial time. *Finding an optimal route in a vehicle routing problem* can be given as an example of this class.

Note: There are also other sets of problems as known as NP-easy, NP-intermediate and NP-equivalent.

Personal thoughts and big MAYBEs. We currently measure the time complexity[14] by using Ω notation which is related to our current technology and its fundamental principles which make it what it is. For example, we could easily observe that if we were to do the same thing(S) several times(n) then the run-time complexity of such algorithm would be $\Omega = nt_S$ and obviously, we would not be wrong about it. This is due to the reason our underlying technology work sequentially processing one unit of information at a time and that is why it would take it nt_S time to terminate the algorithm. If we were using a technology which is able to use parallelism and our algorithm contains strongly independent operations then we would end up with less time complexity, however, this happens mainly due to two reasons: *potential parallelism in the developed algorithm* and *technology which is able to use parallelism.* In contrast, we cannot parallelize a sequential algorithm even if we had the technology for it; however, it is very interesting to me how the brain works while implementing such sequential algorithms. Does it work in parallel but creates an abstraction that makes it seem like sequential to us? Or can it work in a truly sequential manner? Maybe the underlying technology that we currently have has not evolved yet to work as much efficient as it potentially could. Maybe we can develop a technology due to which we would not not worry about the ”for loops” and memory accesses(and it is true that most of the time we do not consider the heavy time consumption of memory accesses while calculating the time complexity of an algorithm which may give inaccurate results); then we potentially could have a different perspective on Ω notation.

2 Optimization Problems

2.1 Linear Programming Problem

Linear Programming(Optimization) Problems[7] are a branch of optimization problems and play an important role in many real-world applications such as business management, vehicle routing, networking and so on. This type of problems has the general form as shown below:

$$\begin{aligned} &\max\ C^T x \\ &\text{s.t. } Ax \le b \\ &x \ge 0 \end{aligned}$$

To find an optimal x vector, there is an algorithm called Simplex[12, 11, 4] which has been developed by George Bernard Dantzig. Since the algorithm has already been discussed before, I am going to give the algorithm which I have implemented in this project directly.

2.2 Knapsack Problem

Knapsack problem[6] is also an integer linear programming[5] problem whose solution is given by $S_n \in \{0,1\}^n$. Solving a knapsack problem by using brute-force approach would take 2^n iterations of computation where n is the number of items. However, we can optimize the algorithm by representing the solutions in a binary tree structure where the items are nodes and the choice of whether picking a specific item or not is the outcoming edge from the node.

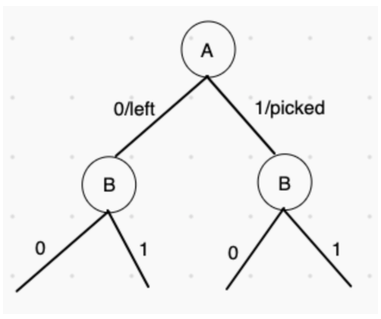


Figure 1: Binary Tree representation of Knapsack Problem

By using tree-based approach, we can actually prune many possibilities depending on the given problem. For example, imagine we know that we cannot choose a specific node(item) because it exceeds the maximum capacity, then we know that we can exclude or prune the right side of the tree beginning from that node and it would save time as the search space is now smaller than it was before. Other that pruning the tree, we could save some precomputed results in the memory for the sake of avoiding repetitive computations. So, dynamic programming and pruning methodologies can be very beneficial to solve a binary knapsack problem.

Algorithm 1: Knapsack Solver

```
Result: Binary solution vector s
if level == n then
    return bestResult;
end
currentWeight ← currentWeight + weightOf(level); currentValue ← currentValue + valueOf(level);
if currentWeight ≤ capacity then
    if currentValue ¿ maxValue then
        maxValue ← currentValue;
        bestResult ← s;
    end
    s[level] = 1;
    Solver(s,level+1,capacity,maxValue);
end
s[level] = 0;
Solver(s,level+1,capacity,maxValue);
```

2.2.1 Relationship with Integer Linear Programming

Let's assume that we have to find an optimal solution to the following problem:

$$\begin{aligned}
\max\ & \sum_{i=1}^{n} \lambda_{0i} x_i \\
\text{s.t.}\ & \sum_{i=1}^{n} \lambda_{1i} x_i \le b_1 \\
& ... \\
& \sum_{i=1}^{n} \lambda_{mi} x_i \le b_m \\
& x \in \{0, 1, ..., k-1\}
\end{aligned}$$

If we had a polynomial-time algorithm to solve the *0/1 Knapsack Problem* then we would have solve the above given problem in polynomial time as well. To do that, we can try to convert the solution space from n possibilities to only 2 possibilities(0 or 1) and it is obvious that, we could do it by representing the numbers in binary instead of decimal.

$$\begin{aligned}
x = & \sum_{i=0}^{\lceil \log_2 k \rceil - 1} 2^i y_i \\
y \in & \{0, 1\}
\end{aligned}$$

Since we can represent any given x with only 0's or 1's as shown above, we can rewrite the original problem as shown below:

$$\begin{aligned}
\max\ & \sum_{i=1}^{n} \left(\lambda_{0i} \sum_{j=0}^{\lceil \log_2 k \rceil - 1} 2^j y_{ij} \right) \\
\text{s.t.}\ & \sum_{i=1}^{n} \left(\lambda_{1i} \sum_{j=0}^{\lceil \log_2 k \rceil - 1} 2^j y_{ij} \right) \le b_1 \\
& ... \\
& \sum_{i=1}^{n} \left(\lambda_{mi} \sum_{j=0}^{\lceil \log_2 k \rceil - 1} 2^j y_{ij} \right) \le b_m \\
& y \in \{0, 1\}
\end{aligned}$$

Since we have used more variables than we had before (due to binary representation), we have to consider this fact in the final time-complexity measurement. It is obvious that 1 non-binary variable expands to $\lceil \log_2 n \rceil$ binary variables and therefore, we have to have $n\lceil \log_2 k \rceil$ binary variables in total. This means, we can easily use an invented polynomial time algorithm for the knapsack problem for this and it is going to take polynomial time again.

3 API specification

The project encapsulates all of its objects namespaces called **lp** and **knapsack** for linear programming and binary knapsack problems respectively. Here is the list of all objects(classes) inside **lp** and **knapsack**:

- lp::Variable - *to define a decision variable*
- lp::Solver - *to define a linear programming problem solver*
- lp::Tableau - *to define a tableau for a given problem*
- knapsack::Solver - *to define a binary knapsack solver*

In fact, there are three main classes that you will frequently use in that namespace:

- lp::Variable
- lp::Solver
- knapsack::Solver

3.1 Defining decision variables

```
    Variable(double min=0, double max=LP_INFINITY);
```

After declaring your variable(s), you can play with its boundaries by using the following member functions of the **Variable** class:

```
    Variable& less_than(double number);
    Variable& less_equal(double number);
    Variable& greater_than(double number);
    Variable& greater_equal(double number);
    Variable& equal_to(double number);
    void satisfy(double min, double max);
```

3.2 Defining an objective function

```
    set_objective(const std::vector<double>& coefficients,
                  goal_t goal);
    set_objective(const std::initializer_list<double>& coefficients,
                  goal_t goal);
```

3.3 Defining constraints

To add a constraint to the problem, you can use a couple of member functions of **Solver** class:

```
    subject_to(const std::vector<double>& coefficients,
               operator_t operator_,
               double number);
    subject_to(const std::initializer_list<double>& coefficients,
               operator_t operator_,
               double number);
```

To remove an already-defined constraint, you can use the member functions shown below:

```
    remove(const std::vector<double>& coefficients,
           operator_t operator_,
           double number);
    remove(const std::initializer_list<double>& coefficients,
           operator_t operator_,
           double number);
    remove(size_t index=-1);
```

3.4 Optimizing the problem

For an LP problem there is a function called solve which can be invoked after the proper set up of objective function as well as the constraints.

```
    status_t solve(bool verbose=false);
```

For a Binary Knapsack problem all you need to do is to call the below shown method of the **knapsack::Solver** class with a couple of arguments(items and maximum capacity).

```
    using solution_t = std::vector<bool>;
    using result_t = std::pair<solution_t /*solution*/, unsigned /*value*/>;

    result_t solve(items_t items, unsigned int capacity);
```

3.5 Macros

There are a few macros all of which begins with the prefix **LP**.

Macro	**Description**	**Type** (optional)	**Used by** (optional)
LP_LESS_THAN_OR_EQUAL	$\leq$	-	Used by Solver::subject_to() method
LP_LESS_THAN	$<$	-	Used by Solver::subject_to() method
LP_GREATER_THAN_OR_EQUAL	$\geq$	-	Used by Solver::subject_to() method
LP_GREATER_THAN	$>$	-	Used by Solver::subject_to() method
LP_EQUAL	$=$	-	Used by Solver::subject_to() method
LP_OPTIMAL	To indicate that the found solution is an optimal one	status_t	Returned by Solver::solve() method
LP_FEASIBLE	To indicate that the found solution is feasible and not optimal	status_t	Returned by Solver::solve() method
LP_NONE	To indicate that there is no feasible solution	status_t	Returned by Solver::solve() method

Table 1: Macros

4 Examples

4.1 LP Solver

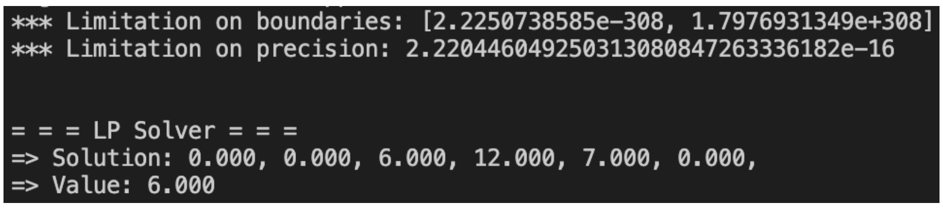
Let's say we want to optimize the problem shown below:

$$\begin{aligned} \max \quad & -3a - b + c \\ \text{s.t.} \quad & -3a - b + 5c \geq 18 \\ & -a - b + 2c \geq 5 \\ & -a + b + c \leq 6 \end{aligned}$$

Here is the code for it:

```
#include <iostream>
#include "solver.hpp"

int main(){
    LP_PRINT_INFO;
    printf("\n= = = LP Solver = = =\n");

        lp::Variable a, b, c;
        lp::Solver solver;

        solver.add_variable(a);
        solver.add_variable(b);
        solver.add_variable(c);

        solver.set_objective({-3, -1, 1}, LP_MAXIMIZE);
        solver.subject_to({-3, -1, 5}, LP_GREATER_THAN_OR_EQUAL, 18);
        solver.subject_to({-1, -1, 2}, LP_GREATER_THAN_OR_EQUAL, 5);
        solver.subject_to({-1, 1, 1}, LP_LESS_THAN_OR_EQUAL, 6);

        auto status = solver.solve(/*true*/);

        if(status == LP_NONE){
                printf("Could not find any solution!\n");
                return -1;
        }

        auto solution = solver.get_solution();
        printf("=> Solution: ");
        for(size_t i=0; i<solution.size()-1; ++i){
                printf("%.3lf, ", solution[i]);
        }
        printf("\n=> Value: %.3lf\n", solution.back());
    return 0;
}
```

The answer looks like this:



```
*** Limitation on boundaries: [2.2250738585e-308, 1.7976931349e+308]
*** Limitation on precision: 2.220446049250313080847263336182e-16


= = = LP Solver = = =
=> Solution: 0.000, 0.000, 6.000, 12.000, 7.000, 0.000,
=> Value: 6.000
```

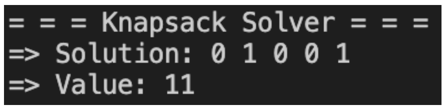
Figure 2: Result shown on the terminal

4.2 Knapsack Solver

Let's try to solve the problem where there are 5 items with the values of 3, 5, 10, 2, 6 and the weights of 4, 3, 9, 3, 8 respectively and the weight limitation or the capacity of a bag is set to 11. What items should we pick to get the maximum possible value without overloading the bag? Here is the code for it:

```
#include <iostream>
#include "solver.hpp"

using namespace std;

int main(){
    printf("= = = Knapsack Solver = = =\n");
    using namespace knapsack;

    items_t items {
        item_t(3, 4),
        item_t(5, 3),
        item_t(10, 9),
        item_t(2, 3),
        item_t(6, 8),
    };
    unsigned capacity = 11;
    auto solver = Solver();

    result_t result = solver.solve(items, capacity);
    for(const auto& bit: result.first)
        cout << bit << ' ';
    printf("\nValue: %d\n", result.second);
    return 0;
}
```

The answer looks like this:



```
= = = Knapsack Solver = = =
=> Solution: 0 1 0 0 1
=> Value: 11
```

Figure 3: Result shown on the terminal

References

[1] “Boolean Satisfiability Problem”. In: (). URL: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[2] “Graph Coloring”. In: (). URL: https://en.wikipedia.org/wiki/Graph_coloring.

[3] “Hamiltonian path problem”. In: (). URL: https://en.wikipedia.org/wiki/Hamiltonian_path_problem#:~:text=The%5C%20Hamiltonian%5C%20cycle%5C%20problem%5C%20is,if%5C%20there%5C%20is%5C%20no%5C%20Hamiltonian.

[4] idk. “Simplex Method”. In: (). URL: http://www.lokminglui.com/lpch3.pdf.

[5] “Integer Programming”. In: (). URL: https://en.wikipedia.org/wiki/Integer_programming.

[6] “Knapsack Problem”. In: (). URL: https://en.wikipedia.org/wiki/Knapsack_problem.

[7] “Linear programming”. In: (). URL: https://en.wikipedia.org/wiki/Linear_programming.

[8] “NP-completeness”. In: (). URL: https://en.wikipedia.org/wiki/NP-completeness.

[9] “NP-hardness”. In: (). URL: https://en.wikipedia.org/wiki/NP-hardness.

[10] “P versus NP problem”. In: (). URL: https://en.wikipedia.org/wiki/P_versus_NP_problem.

[11] Spyros Reveliotis. “An Introduction to Linear Programming and the Simplex Algorithm”. In: (). URL: https://www2.isye.gatech.edu/~spyros/LP/LP.html.

[12] “Simplex Algorithm”. In: (). URL: https://en.wikipedia.org/wiki/Simplex_algorithm.

[13] “Subset Sum Problem”. In: (). URL: https://en.wikipedia.org/wiki/Subset_sum_problem.

[14] “Time complexity”. In: (). URL: https://en.wikipedia.org/wiki/Time_complexity.

[15] “Vehicle Routing Problem”. In: (). URL: https://en.wikipedia.org/wiki/Vehicle_routing_problem#:~:text=The%5C%20vehicle%5C%20routing%5C%20problem%5C%20(VRP,given%5C%20set%5C%20of%5C%20customers%5C%3F%5C%22.&text=The%5C%20objective%5C%20of%5C%20the%5C%20VRP%5C%20is%5C%20to%5C%20minimize%5C%20the%5C%20total%5C%20route%5C%20cost..