CS4402

Practical 1 - Solar Power Company Problem

190004947

October 28, 2021

University of
St Andrews

Word Count:

Contents

1 Introduction

My Essence Prime implementation of a solver for the solar power company problem correctly finds solutions for valid parameter sets minimising the square of the energy transmitted between buildings and the grid. My program file has comments above each block to describe each one's function.

To avoid an extremely difficult debugging session, I developed my solver incrementally. I split the practical into three separate tasks: allocate panels to buildings, allocate workers to panels, and minimise the square of energy transmission. I completed each task in this order and would only work on that section until it was correctly implemented. The first of the three tasks seemed the most simple as power was not yet a relevant parameter and all I was trying to achieve was bin packing. However, it turned out to be the most difficult as at that point my understanding of Essence Prime was rather elementary. By the final task, I knew exactly what I was trying to do and how to achieve it, making the seemingly increasingly more difficult tasks much easier.

2 Variables and Domains

2.1 Parameter declaration

At the beginning of the file I declare all of the parameters from the param file and their domains. Each integer parameter is simply stored as an int type while the rest of the parameters are stored as matrices of int objects. Each of these matrices are indexed between one and the number of elements which is passed in the numPanels, numBuilds, numWorkers, and numDays parameters.

2.2 Domain declaration

Immediately after the parameters have been declared, I have defined four domains to be used in the rest of the program. These domains are called PANELS, BUILDS, WORKERS, and DAYS each spanning the number of elements of each. For example, if numWorkers is equal to three, the domain WORKERS will be int(1..3).

2.3 Non-decision variable declaration

Once the domains have been assigned, I have declared three non-decision variables. The first of which stores the sum of all the panel powers from the panelPower parameter matrix. This sum is used as a maximum bound for the energy transmission between buildings and the grid. I chose this as a maximum bound because in no problem instance should all the panels be on one building which requires zero energy; this is why the sum of panelPower elements should never be exceeded by the energy transmission values. The next non-decision variable I have defined is the minimum bound for the energy transmission with the grid. The minimum bound is calculated by finding the maximum value in the matrix of building power requirements and multiplying it by -1. This assumes that the most power hungry building is being supplied no energy - hence a minimum bound. Lastly, I have declared a variable to store an upper bound of the total square of the energy transmission; the value that we are trying to minimise. I have calculated am upper bound for this by taking the maximum value between either the sum of all panel powers squared or the sum of all building powers squared. The idea behind this is that the final value can never exceed the number returned when either panels or buildings are not producing/consuming energy.

2.4 Decision variable declaration

I have six decision variables in my implementation. Since I built my solution incrementally, These decision variables changed shape, size, and structure many times over the course of development. My initial attempt used a two-dimensional array indexed by panels and days that would store which worker was going to put up which panel and when. This proved to be a difficult way of accessing the required data when writing constraints for the daily budget. After many different trials, my final implementation consists of four one-dimensional matrices, one two-dimensional matrix, and one integer.

The solar power company problem can be considered three separate problems; allocating panels to buildings based on area, allocating workers to panels based on worker capacity and budget, and then minimising the squared energy transmission with the grid. My solution to the first part uses a one-dimensional matrix of integers between one and the number of buildings that is indexed by one to the number of panels there are to allocate. The idea behind this is that each index of the matrix is a panel and the element stored at each index is the building number that that panel will go on. For the second part of the problem, I have used three separate matrix decision variables. The first of which is called workerAssignment which works exactly as the buildingAssignment matrix does; each index is a panel and the element stored at that index is the worker that will install that panel. The next matrix is also similar in that it is indexed by panels but here the element stored at each index is the day on which that panel will be put up. Last of all we have the isWorking two-dimensional matrix of binary elements. It is indexed by number of workers and number of days. The rationale for this is that when constraining the daily budget I can look through that table and sum the daily cost of each worker that is working. Before I came up with the idea for this table I was trying to iterate through the workerAssignment and dayAssignment lists summing the daily costs but couldn't find a way around paying the same worker more than once per a day if they had put up multiple panels that day. After I built the isWorking matrix, the constraint for daily budget was fairly straight forward.

To calculate and constrain the energy transmission values I have a matrix indexed by buildings that uses my previously calculated upper and lower bounds to define a domain. This one-dimensional matrix stores the difference between the energy produced on a building and the energy consumed by a building. Once these values have been found, the totalSquareEnergy-Transmission integer decision variable will store the sum of each of values in the energyTransmission matrix squared. This variable will be minimised as the goal of the solver.

3 Constraints

3.1 Assigning panels to buildings

My entire solver comprises of six constraints. The initial constraint is to allocate panels to buildings. It asserts that for all buildings, the sum of all the panel areas on that building is less than the area of that building. To find this sum, I multiplied the panel areas by a boolean expression that checks if that panel is on the building we are currently evaluating. This acts as a sort of 'if' statement and is a crucial part of most of my constraints. Once the panels have been assigned to buildings, I have two constraints to find the values of energy transmission and then the total square of energy transmission values. With the goal of the solver already being declared as to minimise this total square sum, the constraints are quite simple. In calculating the energy transmission for each building, again I use a boolean expression as a multiplication operand to check that the panel power being summed is actually on the building we are currently evaluating. The totalSquareEnergyTransmission is is simply defined as the sum of each of these values squared.

3.2 Assigning workers to panels

The final three constraints are for allocating workers to keep within budget while also finishing the project within the given number of days. First, we must constrain the amount of work each worker does per day to their individual capacities. This can be achieved by using a nested forAll universal quantifier to check that for each day and for each worker, if the current worker put up the panel we are looking at on the current day we are evaluating, add the panel area to a sum which must equate to less than that worker's capacity. Again, the 'if' statement behaviour is simulated using two boolean expressions that will be multiplied to the panel area. The next two constraints are to ensure that the daily budget is not being exceeded. Firstly we need to find which workers are working on which days stored in the isWorking decision variable. To do this, we constrain that for each day, for each worker, there exists a panel that is put up by that worker on that day. The 'exists' keyword is used here to return a 1 if a worker is working on a given day and a 0 otherwise. We then use this matrix of who is working when to sum the worker cost of all workers that are working on each day and assert that it is less than the daily budget.

4 Example Solution

4.1 Parameters

Suppose we have 2 days to install 7 solar panels:

Solar Panel	Power Generated	Area
1	15	4
2	18	5
3	13	6
4	15	2
5	10	5
6	8	4
7	5	2

On 5 buildings:

Building	Power Demand	Roof Area
1	30	11
2	25	20
3	22	9
4	31	12
5	17	20

4.2 Solution

Using 3 workers and a budget of 50 per day:

Worker	Area installed each day	Salary per day
1	8	20
2	9	25
3	10	30

In a solution, each one of the panels must be placed on one of the two buildings without exceeding the building's roof area. One solution would place panels 1 and 7 on building 1, panel 2 on building 2, panel 3 on building 3, panels 4 and 6 on building 4, and panel 5 on building 5:

Building	Panels	Power Generated	Power Demand	Power Excess
1	1, 7	20	30	-10
2	2	18	25	-7
3	3	13	22	-9
4	4, 6	23	31	-8
5	5	10	17	-7

Worker 1 will install panels 1, 2, 4, and 6, while worker 2 will install panels 3, 5, and 7. Note that worker 3 doesn't install any panels in this solution.

Worker	Panels day 1	Panels day 2	Salary day 1	Salary day 2
1	1, 6	2, 4	20	20
2	3, 7	5	25	25
3	0	0	0	0

Note that the sum of salaries for both days (20+25 = 45) is less than the budget of 50 per day. On day 1, worker 1 meets their area capacity while worker 2 spares 1 metre squared of their capacity. On day 2, worker 1 spares 1 metre squared while worker 2 spares 4 metres squared and so at no point is worker capacity exceeded.

4.3 Result

In this solution, the total power sold to the grid is zero and the total power bought from the grid is 41kW. In fact this is an optimal solution and the total sum of squared power transmission is found to be 343kW.

5 Empirical Evaluation

To empirically evaluate my solver, I collected the result, SolverNodes, SolverSolveTime, and SavileRowTotalTime of my initial solution and then on the same solution with different orders of branching on decision variables.

5.1 Initial results

These are the results of my solver on each of the parameter sets before introducing a 'branching on' clause. The data will serve as a benchmark to compare the other results to. Note that the buildingAssignment and energyTransmission is calculated first in each instance as this part of the solution is completely self-contained and does not interact with any other decision variables.

spcp.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
4	25	0.000496	0.373

spcp2.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
343	37,341	0.338579	0.753

spcp3.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
17	138,659	0.922616	0.709

5.2 Results with branching order [buildingAssignment, energyTransmission, totalSquareEnergyTransmission, workerAssignment, dayAssignment isWorking]

As we can see in the tables below, this branching order produces solutions with the same number of SolverNodes as the base implementation. However, by forcing this branching order we see slight improvements on both the SolverSolveTime and the SavileRowTotalTime for all three parameter sets. Clearly, even though it uses the same number of SolverNodes, this is a more efficiently computable model.

spcp.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
4	25	0.000213	0.312

spcp2.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
343	37,341	0.296695	0.549

spcp3.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
17	138,659	0.927363	0.650

5.3 Results with branching order [buildingAssignment, energyTransmission, totalSquareEnergyTransmission, dayAssignment, workerAssignment, isWorking]

In this set of tables, we are branching on the dayAssignment before the workerAssignment. In this case, we actually see a significant reduction in the number of SolverNodes used. Along with less SolverNodes, we see similar solver times to the previous branching order. The slight difference in time is that the smaller parameter file one actually took a couple tenths of a second longer while the larger parameter file three took five tenths quicker; this could be a result of using less SolverNodes or could be down to natural variation. Regardless, this branching order produces the best solver model out of the orders I tested.

spcp.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
4	19	0.000333	0.337

spcp2.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
343	34,303	0.283703	0.600

spcp3.param			
Result	SolverNodes	SolverSolveTime	SavileRowTotalTime
17	90,491	0.676271	0.601

I think there might be an optimal solution that branches through dayAssignment and workerAssignment simultaneously but I could not find one.

5.4 Results when branching on isWorking before dayAssignment and workerAssignment

When testing the program branching on the isWorking matrix before the dayAssignment and workerAssignment decision variables (both ways around) the number of SolverNodes massively increased. These solutions were highly inefficient and took an excessively long time to run; especially on the third set of parameters.

5.5 Comments

After finding the best branching order, I ran tests with different optimisation levels using the -O0, -O1, -O2, and -O3 flags. Interestingly, the default level, -O2, produced the shortest SolverSolveTimes and SavileRowTotalTimes and so no optimisation level should be explicitly stated when running the solver.

6 Conclusion

To conclude, the solver I have built successfully computes results for parameter files one, two, and three. The final results, which can be seen in the Empirical Evaluation section above, are 4, 343, and 17 respectively which match the results posted on Microsoft Teams. By testing different variable orders for minion to search through, I discovered a variation that uses 22.3% less SolverNodes than if no branching order was defined. Additionally, SavileRowTotalTime was cut by an average of 15.1%.