Universitat Politècnica de Catalunya

June 11, 2013

Datacenter Power-Aware Management

ILP and Heuristic Optimizers

Alexandre Fonseca, Anh Thu Vu, Peter Grman

Contents

1 Introduction

This report aims to document and justify the development process our group underwent for the Communication Networks Optimization project where we implemented several solvers for an assignment problem and compared their performances and characteristics.

The problem we analyzed is related to the assignment of jobs to hosts keeping in mind CPU capacities, power consumption per CPU and Quality of Service (QoS) penalties.

We considered 2 different types of solvers: Integer Linear Program (ILP) solvers and heuristic solvers. For the latter, we developed our own heuristic solver frameworks and used several different heuristics such as GRASP, TabuSearch, Simulated Annealing and Path Relinking.

2 The problem

The problem we proposed to work on is that described in [2] where, in the context of a datacenter, we have a fixed amount of resources and a set of *jobs*. Thus, it is important to find an allocation to fit the *jobs* into our resources while maximizing our profit.

In our model, we consider the resources as a set of *hosts*, with each having a limited number of CPUs, and *jobs*, with each requiring a variable amount of CPU usage (bounded by *cpuMin* and *cpuMax*). If a *job* is assigned to a *host* then we'll benefit from that job's revenue (which varies from job to job). The actual revenue obtained from a specified job should take into account its QoS. We consider the QoS (or health) of a job to be a value between 0 and 1 which relates its actual CPU usage to its bounds:

$$\text{QoS} = (1 - \frac{ActualCPUUsage}{MaximumCPUUsage}) \tag{1}$$

We also take into account the power cost for active hosts (those with assigned jobs). Each of the CPUs utilized will consume a certain amount of energy. It was shown in [1] that turning on an extra CPU in an already-running *host* consumes less energy than turning on an inactive *host*. This characteristic is reflected in our model by ensuring the amount of the power consumed by the $(i+1)^{th}$ CPU is less than the i^{th} and the i^{th} must be fully utilized before the $(i+1)^{th}$ CPU is activated.

Lastly, we do not consider splitting a *job* over multiple *hosts*.

3 Integer Linear Program

We modeled the above characteristics into an ILP using the GNU MathProg modeling language and tested it with GNU Linear Programming Kit (GLPK). The

model consists of 9 *parameters*, together with some corresponding *sets*, 4 *variables*, 10 constraints and an objective function as can be observed in the complete model in subsection 3.5:

3.1 Parameters

- *maxCPUs*: A positive integer indicating the maximum number of CPUs that any host can have.
- *noHosts*: A positive integer indicating the total number of hosts.
- *noJobs*: A positive integer indicating the total number of jobs.
- *powerCost*: A non-negative real number indicating the cost of one unit of power.
- *cpus[h]*: A vector of *noHosts* positive integers indicating the number of CPUs in the corresponding host.
- *pwr[i]*: A vector of *maxCPUs* positive real numbers indicating the amount of power consumed by the i^{th} CPU in a host.
- *consmin[j]*: A vector of *noJobs* positive integers indicating the minimum CPUs requirement of the j^{th} job, in percentage unit.
- *consmax[j]*: A vector of *noJobs* positive integers indicating the maximum CPUs requirement of the j^{th} job, in percentage unit.
- *revenue[j]*: A vector of *noJobs* positive real numbers indicating the revenue of the corresponding scheduled job.

3.2 Variables

- *schedule[h,j]*: A matrix of *noHosts* x *noJobs* binary values, each indicates whether the h^{th} host is allocated for the j^{th} job.
- *pr[h,i]*: A matrix of *noHosts* x *maxCPUs* binary values, each indicates whether the i^{th} CPU at the h^{th} host is turned on.
- *jcpu[j]*: A vector of *noJobs* non-negative integers indicating the amount of CPUs allocated for the j^{th} job if it is scheduled. If the job is not scheduled, it can be any value within the [*consmin*, *consmax*] range, which will effectively give it the value of the corresponding *consmax* in order to maximize the objective function.
- *quota[h,j]*: A matrix of *noHosts* x *noJobs* non-negative integers. Each of them will take the value of the corresponding *jcpu[j]* if the job is scheduled at host h or zero otherwise. The sum of *quota[h,j]* over the all h is essentially the actual amount of CPUs assigned to job j.

3.3 Constraints

- *Processor*: This constraint ensures that a $(i+1)^{th}$ CPU in a host is turned on only if the i^{th} CPU of the same host is already turned on.
- *Unique*: This constraint ensures that a job is either scheduled at exactly one host or not scheduled.
- *MaxCPU*: This constraints ensures that the number of CPUs utilized at a host does not exceed the total number of CPUs at that host.
- *Capacity*: This constraints ensures the total amount of CPUs consumed by all the jobs allocated at a host does not exceed the total number of CPUs at that host.
- *MarginCPU1-2*: These two constraints restrict the value of *jcpu* to be within the range of the corresponding *consmin* and *consmax*.
- *QosAux1-4*: These constraints enforce the relation between the value of *quota* variables and the corresponding *jcpu*.

$$quota[h,j] = \begin{cases} jcpu[j] & if schedule[h,j] = 1 \\ 0 & if schedule[h,j] = 0 \end{cases}, \forall h \in \{1..noHosts\}, j \in \{1..noJobs\}$$

3.4 Objective Function

Our objective function is to maximize the final profit which consists of the total revenue of all scheduled jobs, subtracting the cost of the power consumed by all the utilized CPUs and the QoS penalty for those jobs that the *consmax* threshold couldn't be satisfied. This penalty is computed as a function of *jcpu*, *consmax* and *revenue*.

3.5 Model

```
/* PARAMETERS & SETS*/
param maxCPUs, integer, >0;
set I := 1..maxCPUs;
set I_sub := 2..maxCPUs;
param pwr {i in I}, >0;

param noHosts, integer, >0;
set H := 1..noHosts;
param cpus {h in H}, integer, >0;

param noJobs, integer, >0;
set J := 1..noJobs;
param consmin {j in J}, integer, >0;
param consmax {j in J}, integer, >0;
```

```
param revenue {j in J}, >0;

param powerCost, >=0;

/* VARIABLES */
var schedule {h in H, j in J}, binary ;
var quota {h in H, j in J}, integer, >=0 ;
var pr {h in H, i in I}, binary ;
var jcpu {j in J}, integer, >=0 ;

/* OBJECTIVE FUNCTION */
maximize z : sum{h in H, j in J} schedule[h,j] * revenue[j]
             - sum{h in H, i in I} pr[h,i] * pwr[i] * powerCost
             - sum{j in J} (1 - (jcpu[j] / consmax[j])) * revenue[j];

/* CONSTRAINTS */
s.t. Processor {h in H, i in I_sub} : pr[h,i-1] >= pr[h,i] ;
s.t. Unique {j in J} : sum{h in H} schedule[h,j] <= 1 ;
s.t. MaxCPU {h in H} : sum{i in I} pr[h,i] <= cpus[h] ;
s.t. Capacity {h in H} : sum{j in J} quota[h,j] <= (sum{i in I} pr[h,i]) * 100;
s.t. MarginCPU1 {j in J} : jcpu[j] >= consmin[j] ;
s.t. MarginCPU2 {j in J} : jcpu[j] <= consmax[j] ;

s.t. QosAux1 {h in H, j in J} :
    quota[h,j] >= schedule[h,j] ;
s.t. QosAux2 {h in H, j in J} :
    quota[h,j] <= schedule[h,j] * maxCPUs * noHosts * 100;
s.t. QosAux3 {h in H, j in J} :
    (quota[h,j] - jcpu[j]) <= (1 - schedule[h,j]) ;
s.t. QosAux4 {h in H, j in J} :
   (jcpu[j] - quota[h,j]) <= (1 - schedule[h,j]) * maxCPUs * noHosts * 100;

end;
```

However, as we noticed an exponential increment in execution time even with small-medium size data sets while testing our model with GLPK, we decided to translate our data sets into a more universal format, i.e. Mathematical Programming System (MPS), in order to conveniently solve the instances with different solvers, i.e.: GLPK, Gurobi, Coin-OR Branch and Cut (CBC), and evaluate their performances. The result of this comparison is shown in subsection 5.2.

4 Heuristic Solvers

In this section, we'll discuss the implementation details and algorithms used in the construction of our heuristic solvers.

4.1 Frameworks

During our implementation, we ended up creating 2 different heuristic frameworks, one in C++11 and another in C# with fundamental differences regarding neighbourhood generation and, thus, complexity and flexibility.

4.1.1 C++11

Our C++11 framework was developed in a completely Object-Oriented manner with every entity in the problem (*Job*, *Host*, *Solver*, *Solution*, …) being represented by an individual class.

In this framework, the *Solver* is an entity which has 3 different responsibilities/execution steps:

- *Construction* — Construct an initial solution.
- *Local Search* — Explore the resulting solution neighbourhood.
- *Path Relinking* — Explore the paths between the best identified solutions.

A solver *round* corresponds to a sequential execution of the above 3 steps. The stopping condition of the solver is either a fixed number of these rounds, a time limit, or both.

A *Solution* object contains all the data related to job-to-host assignments and respective CPU usages. Connections from a specific solution to all neighbour feasible solutions (we do not consider unfeasible ones) can be obtained through the *Solution.getPossibleOperations()* method. This method returns a list of operations that can be applied to a *Solution* object to turn it into a new *Solution*. The operations we considered in this heuristic framework were:

- *AssignmentOperation* — The assignment of a job to a host. The job can either be unassigned or assigned to a different host before the execution of this operation. The target host of the assignment must, however, have enough CPU capacity left to be able to run the job at its current CPU usage. If the job was unassigned before, the current CPU usage corresponds to its minimum CPU usage.
- *SwapOperation* — This operation allows the swapping of jobs in 2 different hosts provided that those hosts have enough CPU capacity to accommodate the swapped operations.
- *CPUAssignmentOperation* — This operation affects the CPU usage of a certain assigned job. This operation can either increase or decrease a job's CPU usage by a fixed increment (which is a parameter of the heuristic framework) provided that the minimum and maximum bounds are respected at all times. In the event, that the increment would lead the CPU usage above/below a bound, the increment is truncated to match the bound. The utility for each solution is calculated using the exact same formula as that described in the ILP section.

4.1.2 C#

Although the C# framework has a very similar object oriented structure as the C++ framework, it is simplified a bit to provide higher performance. A *Solution* object only contains a list of *HostWithJobs-Objects*. A *HostWithJobs-Object* is a specific host and a list of the assigned jobs to this host.

Unlike the C++ framework, the C# framework doesn't have the possibility to retrieve all possible operations. This is due to the huge number of these operations and the slow execution of the .Net Framework. Also it wasn't necessary for the heuristic implemented with this framework. However it was necessary to retrieve a random operations, so there is a workaround to this performance problem. Instead of getting all operations which would generate feasible neighbouring solutions, the C# framework can calculate the number of possible operations. The calculation is based on the possible combinations of switching jobs between hosts, reassigning jobs to different hosts or changing the CPU assignments of jobs. A random operation can afterwards be retrieved by index and the framework calculates what kind of operation it is and the corresponding parameters.

Since operations can lead us to unfeasible solutions where hosts are overloaded, mechanisms were required to deal with these solutions. While penalising them with an extremely low value, so that the heuristic wouldn't consider choosing it, is one implemented part, it turned out not to be enough. Much better heuristic results were measured after a second mechanism was implemented which tried to make the solutions feasible. This is done, by reducing the assigned CPU time to all jobs within a host until the solution is feasible. Jobs with a low reward per CPU-consumption are considered first. Jobs with a higher reward later.

4.2 GRASP+TS Solver

This solver was implemented uniquely in C++11 and combines 3 different heuristics: GRASP for construction, Tabu Search for local searching and Path Relinking for combining elite solutions.

4.2.1 GRASP

Our GRASP implementation follows a sample greedy construction as described in algorithm 1. GRASP's parameter p determines the size of the random sample taken from the total number of possible operations from which the best operation will be chosen. If $p \to 1$ we get a more randomized construction whereas with

$p \rightarrow |ops|$ we get a greedier construction.

Algorithm 1: Sample Greedy Grasp

```
input  : p
output: solution

solution ← new Solution();
while not all jobs assigned or number of applied operations < |Jobs| * 4 do
    ops ← solution.getPossibleOperations();
    opsSample ← sample(ops, p);
    bestOp ← getBest(opsSample);
    solution.apply(bestOp);
```

4.2.2 Tabu Search

Our tabu search implementation is pretty much default as can be seen in algorithm 2. It works as a pure greedy algorithm until a local maximum is reached. When this happens, it will move unto worse solutions with the hope of navigating a small valley and be able to reach higher maxima.

The tabu set maintained by this algorithm determines the set of operations which the algorithm is not allowed to perform to prevent it from constantly coming back to the same local maximum. The tabu set works as a simple FIFO fixed-length structure.

Regarding stopping conditions, tabu search will stop when it has navigated through a certain number of solutions without obtaining any improvement.

Algorithm 2: Greedy Tabu Search

```
input  : maxItersWithoutImprovement, tabuSetSize, initialSolution
output: bestSolution

solution ← initialSolution;
bestSolution ← solution;
tabuSet ← new TabuSet(tabuSetSize);
numItersWithoutImprovement ← 0;
while numItersWithoutImprovement < maxItersWithoutImprovement do
    ops ← solution.getPossibleOperations();
    bestOp ← getBestNotIn(ops, tabuSet);
    solution.apply(bestOp);
    tabuSet.add(bestOp);
    if solution.getUtility() > bestSolution.getUtility() then
        bestSolution ← solution;
        numItersWithoutImprovement ← 0;
    else
        numItersWithoutImprovement ++;
```

4.2.3 Path Relinking

The best solution found with Tabu Search is then subjected to a Path Relinking algorithm. What our implementation of Path Relinking does is select a random solution from our elite set (set of best solutions found so far, one per round) and use that solution as the destination solution and our currently found solution as the source one. We then greedily apply the operations which bring the source solution closer to the destination one with hopes of finding other good solutions in that path. The returned solution is the best one found during its execution. The complete algorithm is described in algorithm 3.

Algorithm 3: Path Relinking

```
input  : sourceSolution
output: bestSolution

solution ← sourceSolution;
bestSolution ← solution;
destinationSolution ← randomChoice(eliteSet);
while true do
    ops ← solution.getPossibleOperationsTowards(destinationSolution);
    if |ops| == 0 then
        break;
    bestOp ← getBest(ops);
    solution.apply(bestOp);
    if solution.getUtility() > bestSolution.getUtility() then
        bestSolution ← solution;
```

The best solution returned by the path relinking algorithm is then added to the elite set. Since the elite set has a fixed maximum size, its procedure for deciding which solutions to keep and which to remove is based on the distance between solutions (an approximate number of operations needed to get from one solution to the other) and their utility. In this sense, the removed solution is the one that is worse than the newly added solution and from the set of worse solutions, the one which is closer to the newly added solution. In this sense, we attempt to keep in the elite set the best most distant solutions from one another as they allow us to explore a greater solution space than if they were all clustered in the same region.

4.3 SA Solver

This solver implements a Simulated Annealing heuristic. It was written for both frameworks, in C++11 and C#. The implementation is very similar, only the construction of the initial solution differs.

4.3.1 Simulated Annealing

The idea of Simulated Annealing is to allow a search focused on better solutions but prevent it from getting caught in a local optimum. This is done by having a

chance to also choose worse solutions. This chance depends on a special parameter called temperature. The higher the temperature, the better the chance of choosing a worse solution. The temperature gradually cools so less and less worse solutions will be accepted.

While it is recommended to use a logarithmic function to calculate the temperature cooling, and the Boltzmann distribution to calculate the likely hood of keeping a worse solution, it wasn't feasible to implement them in our heuristics due to the increased complexity. Instead we used an exponential function ($T_{n+1} = \alpha * T_n$). You can choose α to be closer to 1, like 0.9999 or further away like 0.9 and control with this the number of the iterations. The number of iterations in fact is $\log_{alpha}(\frac{T_{min}}{T_0})$.

Instead of the Boltzmann Distribution we used a simplified custom algorithm algorithm 4. We choose a random value between 0 and the initial temperature (T_0). This value is the highest allowed value and the random part of our computation. Then we divide the utility difference between the new and old solution by the current temperature ($T_{current}$). Since $T_{current}$ is lower after each iteration, the final result will be higher and the chance for the random value to be higher will sink. With T_0 and $T_{current}$ you can control the magnitude of utility difference you want to accept.

Algorithm 4: Likelyhood Computation

highestAllowedValue $\leftarrow$ `random` (0, T_0);
utilityDifference $\leftarrow utility_{old} - utility_{new}$;

if utilityDifference$/T_{current} <$ highestAllowedValue **then**
  return continue with new solution
else
  return roll back to old solution

4.4 SA Solver Implementation

As you can see in algorithm 5, we take an initial solution and keep it as the current one. In each iteration a random operation is chosen from the current solution. If the random operation improves the current solution, it is kept. If the new solution is better than the old best solution is is also remembered as the best solution.

Otherwise if the new solution is worse, it can be still kept by chance as described in subsubsection 4.3.1. If it won't be kept, the operation is undone.

In any case a new temperature is calculated and the iteration repeats until the

minimal temperature is reached.

Algorithm 5: SA Solver in C++11

```
input  : initialSolution, T_0, T_min
output: bestSolution

currentSolution ← initialSolution;
bestSolution ← initialSolution;
currentTemperature ← T_0;

while currentTemperature ≥ T_min do
    randomOperation ← currentSolution.getRandomOperation ();
    oldUtility ← currentSolution.getUtility ();
    randomOperation.execute (currentSolution);
    newUtility ← currentSolution.getUtility ();

    if newUtility > oldUtility then
        if newUtility > bestSolution.getUtility () then
            bestSolution ← currentSolution;
    else
        if Not keepSolutionByChance (newUtility, oldUtility) then
            randomOperation.undo (currentSolution);

    currentTemperature ← recalculateTemperature ();
```

4.4.1 Construction in C++11

The C++ implementation doesn't have an explicit construction phase, instead it simply uses and empty solution. Due to the fact, that unassigned jobs are penalised, the heuristic as such tries to assign all the jobs, to reach the optimal value.

4.4.2 Construction in C#

In C# there exist two different construction heuristics. One is based on pure random assignment of jobs to hosts, not considering any boundaries of the hosts resources.

The other is a greedy domain specific heuristic, which simply orders the hosts by the number of CPUs they offer and the jobs by the CPU-Time they require. Afterwards, it tries to assign the jobs, which require the most CPU-Time to hosts which offer the most CPUs. Also because the first CPU-Core used always requires the same amount of energy, it is better to start filling up hosts with a higher number of CPUs, since it might be possible to shut down some other weaker hosts later.

This second heuristic achieves very good results, when the problem domain is simple. However if the objective function would change, the heuristic would need to be rewritten from scratch.

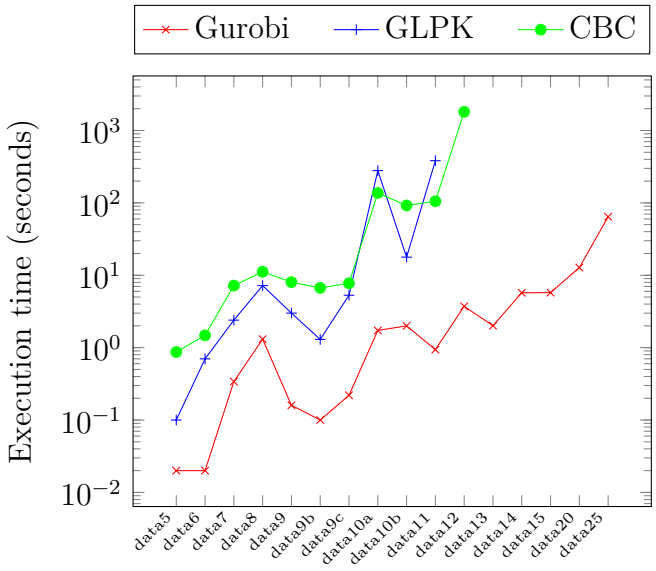

Figure 1: Execution time of different ILP solvers

5 Evaluation

5.1 Data sets

We evaluate our implementations of heuristics solvers against the different ILP solvers base on a set of random generated data sets. The *data-generator* script reads in the number of hosts and number of jobs and randomly decides on the other parameters, i.e.: number of CPUs for each host, minimum and maximum requirement of CPUs of each job. The maximum number of CPUs in a node and the cost per power unit are kept constant in all the data sets as 8 and 15, respectively. Since both number of hosts and number of jobs are proportional to the complexity of the problem, we keep the relation between these two parameters constant (*numOfHosts* = *numOfJobs)* in most of our data sets while increasing them from 5 to 25. Since there seems to be a reduction in complexity in the case of 9 hosts and 9 jobs as well as a drastic increment of complexity when the numbers are set to 10, we repeated the test several times for these two cases (data9, data9b, data9c and data10a, data10b). In the end, our data collection consists of 17 data sets, in which 16 have *numOfHosts* = *numOfJobs* and the most complex one has 20 hosts and 50 jobs (data20-50).

5.2 Different ILP solvers

Due to a better parallelization scheme and a more sophisticated branch-and-cut algorithm, Gurobi outperformed the other two solvers as shown in Figure 1 (note the logarithmic scale).

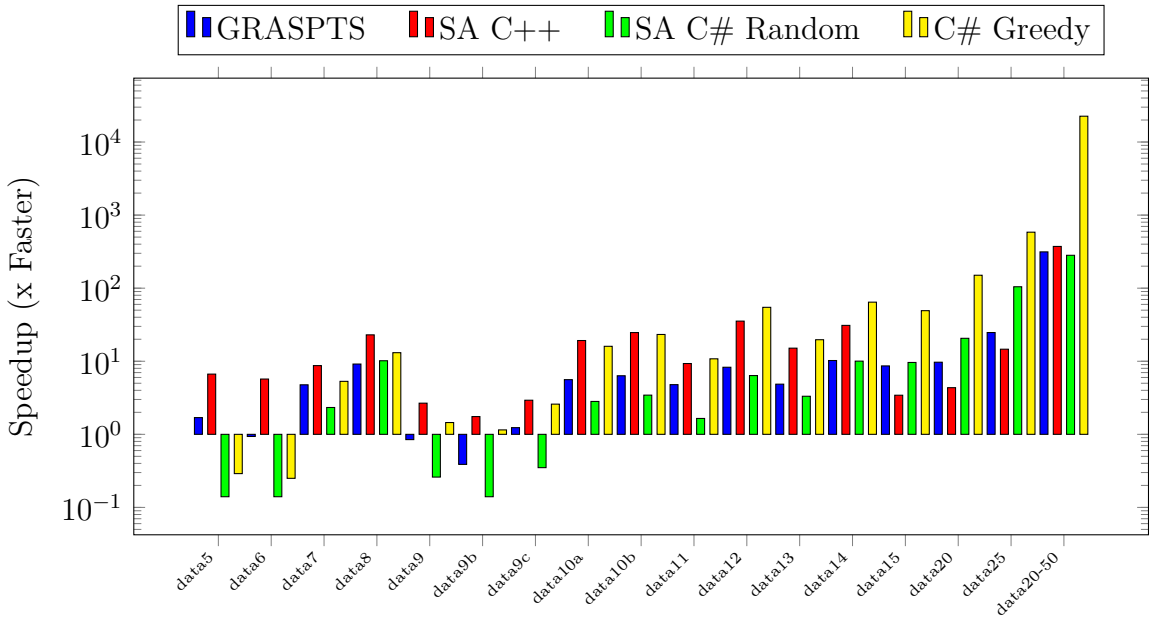

Figure 2: Speedup of heuristic solvers vs Gurobi (C# Greedy is only the greedy construction heuristic)

5.3 Heuristic vs ILP

In this section we compare the execution times and found utilities between the best of the ILP solvers (Gurobi) and our heuristic solvers.

As you can see in Figure 2, for bigger data, the heuristics achieve speedups in the order of hundreds or even thousands of magnitude. While the heuristics implemented in C# are relatively slower for small datasets, with larger datasets the difference between languages is attenuated.

In Figure 3 you can see that the error of our heuristics is usually below the 3% mark. However you can also see that there are few spikes, independent of the dataset size. This shows that, while our heuristics in general can return very good results, we cannot guarantee their quality with absolute certainty.

5.4 GRASP+TS parameters

In this section we examine the effects of the several different parameters that can be changed in the GRASP+TS heuristic solver, seeing how they affect not only execution time but also the maximum utility found. Unless otherwise stated, the results shown are from executions of the solver with a dataset containing 20 hosts and 50 jobs (*data20-50* in the previous graphs). The default parameter values used were:

- CPU increment = 25
- GRASP $p = 100$
- Number iterations without improvement = 200

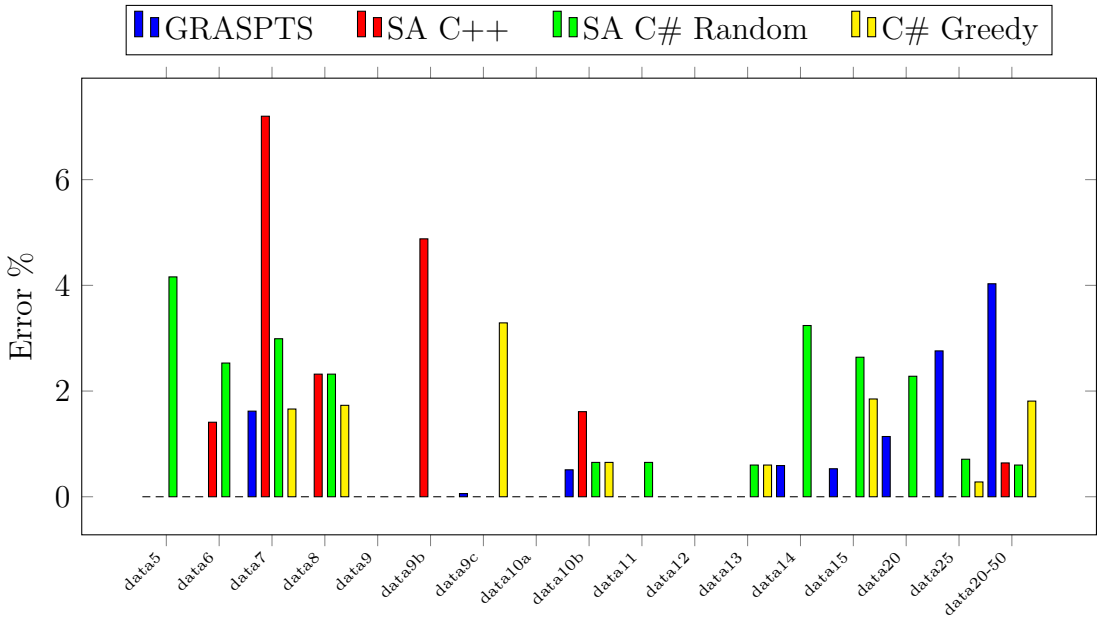

Figure 3: Error % of best solution when compared to optimal ILP solution. (C# Greedy is only the greedy construction heuristic)

- Tabu set size = 400
- Total number of rounds = 5
- Elite set size = 10
- Time limit = -1 (no time limit)

5.4.1 CPU Increments

Intuitively, the bigger the CPU increments, the lesser the depth of a solution's neighbourhood and the more probable it is that we miss optimal arrangements in situations where greedy assignments do not yield the best result. Lower CPU increments allow us to also examine these cases but at the cost of having a bigger solution neighbourhood depth and thus taking longer to reach optimum solutions where all jobs can run at their maximum CPUs.

Our intuition is supported by the results we obtained through experimentation with different values of this increment under 2 different scenarios: a non-overload scenario where jobs can run at their maximum CPU without overloading the available hosts and an overload scenario where only a handful of jobs fit into the hosts using their maximum CPU and, thus, the optimal solution corresponds to a more balanced CPU assignment. These results can be seen in Figure 4, Figure 5 and Figure 6.

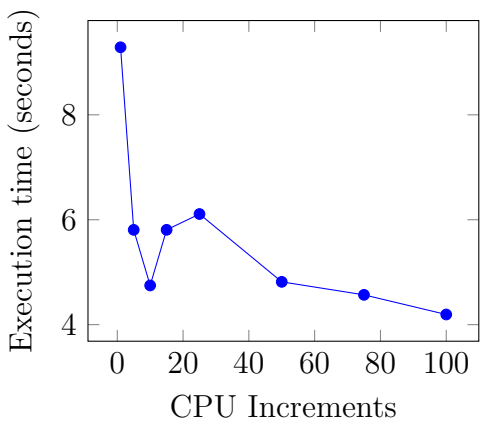

Figure 4: Execution time for different CPU increments in a non-overload scenario.

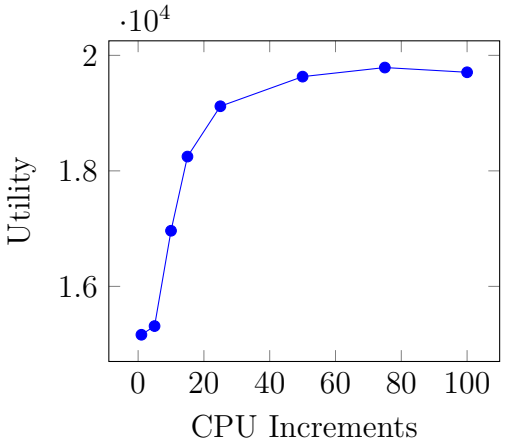

Figure 5: Maximum utility found for different CPU increments in a non-overload scenario.

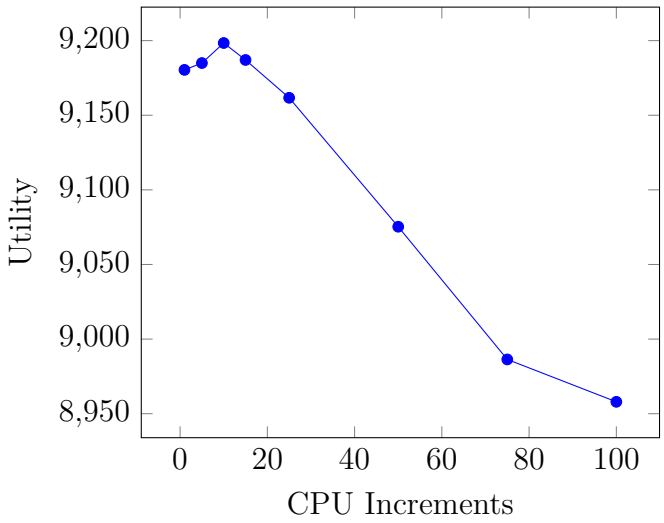

Figure 6: Maximum utility found for different CPU increments in an overload scenario.

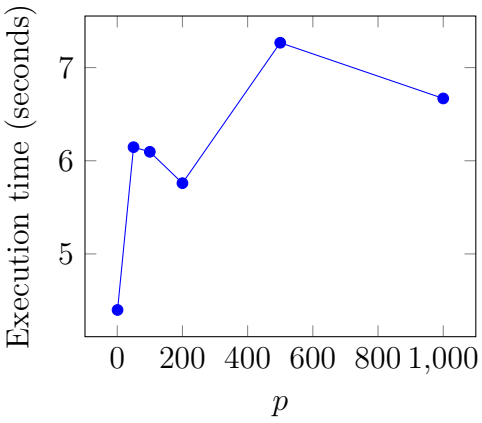

Figure 7: Execution time for different values of p.

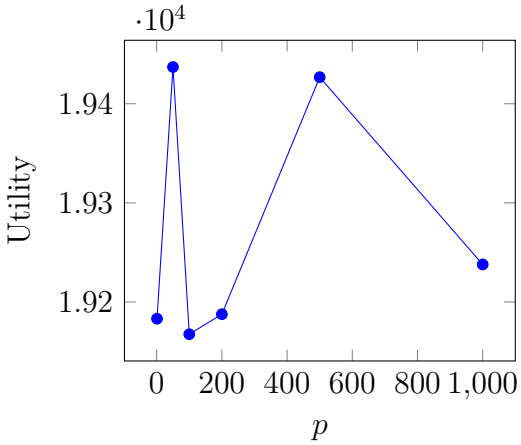

Figure 8: Maximum utility found for different values of p.

5.4.2 GRASP's parameter p

As was described in subsubsection 4.2.1, the p parameter of grasp influences the greediness of the constructed solution. The closer it is to 1 the less greedy and more random it is.

Figure 7 and Figure 8 show the effects of varying p keeping everything else constant. In terms of utility, there doesn't appear to be any significant changes but in terms of time it does appear that, to a certain extent, the lesser the p the less time it takes to execute the 5 rounds. We hypothesize that by following more random approaches, GRASP is less likely to get stuck in the same zone of local minima and thus Tabu Search has less probability of having to navigate several local minima before exiting.

In Figure 9 we see the resulting utility by using GRASP in conjunction with a purely greedy non-TS approach. The obtained results appear to support our previous hypothesis in that, regarding this specific data set, the more greedy the approach, the more likely it is that the algorithm only checks a very limited and sub-optimal part of the solution space while with increased randomization we manage to explore and detect new maxima.

5.4.3 Number of iterations without improvement

In Figure 10 and Figure 11 we can see the results of running the GRASP+TS heuristic solver with different values of the number of iterations without improvement. To make this change relevant we also set the tabu set size to double the number of iterations without improvement otherwise Tabu Search would just spend those iterations navigating the same peak. It is clear from observation of the aforementioned figures that the bigger this value is, the better solutions Tabu Search is able to find at the expense of a worse execution time. We can also see that, while the previous sentence is true, with values bigger than 200, the improvement of utility is negligible compared to the effect on the execution time.

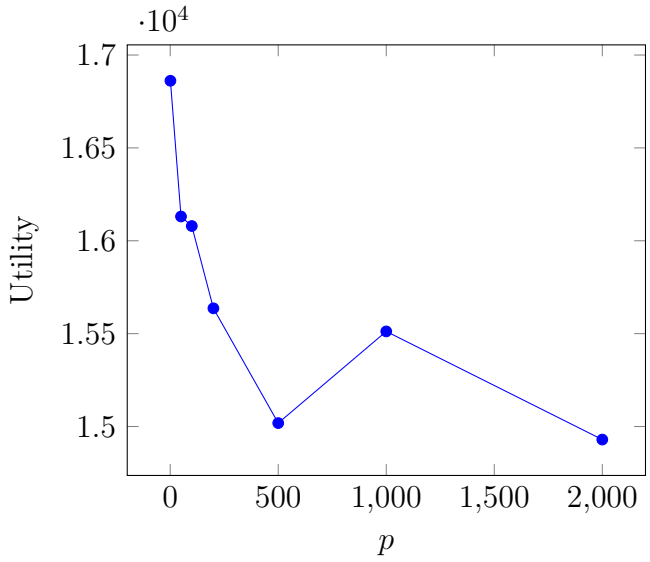

Figure 9: Maximum utility found for different values of p with a pure greedy local search.

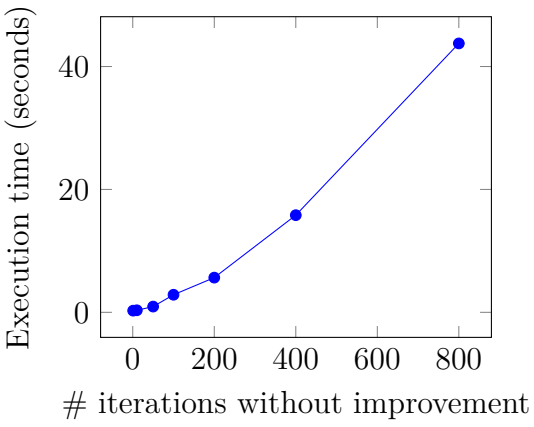

Figure 10: Execution time for different values of the number of iterations without improvement and tabu set size.

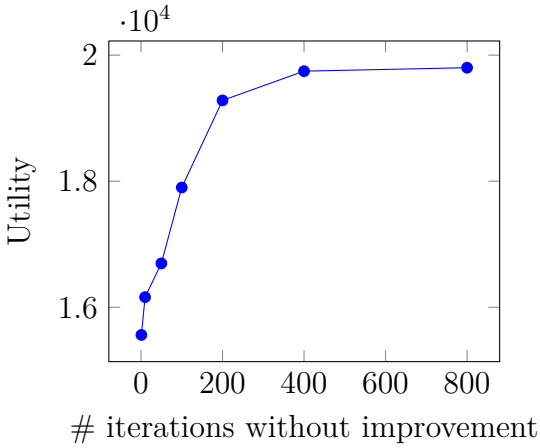

Figure 11: Maximum utility found for different values of the number of iterations without improvement and tabu set size.

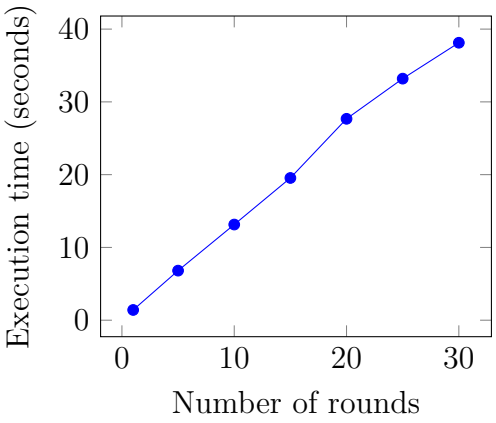

Figure 12: Execution time for different values of the maximum number of rounds.

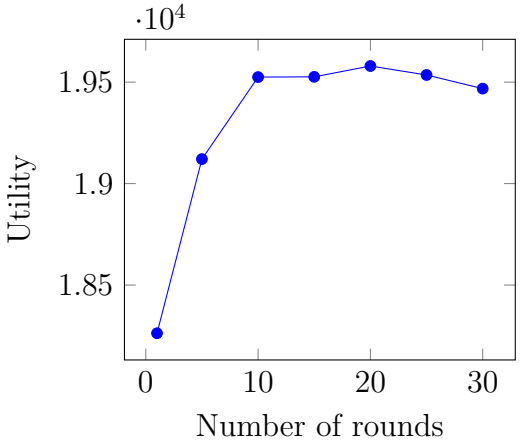

Figure 13: Maximum utility found for different values of the maximum number of rounds.

5.4.4 Number of rounds

In this section, we examine the effect of the total number of rounds (i.e, total number of construction-local search-path relinking executions) on the execution time and utility. Figure 12 and Figure 13 show the collected results for time and utility, respectively.

As we can see, and as was expected, the execution time grows linearly with the number of rounds as going from 1 round to 2 essentially means doubling the workload. The quality of the solutions found also grows almost linearly with the number of rounds up to a value of 10 rounds. This happens because different rounds are more likely to explore different solution spaces. However, with more than 10 rounds, it becomes less and less likely that a new round manages to explore a totally different neighbourhood and thus the profit from running those extra rounds decreases.

5.4.5 Elite set size

By varying the elite set size, we examine the efficiency of the path relinking algorithm in finding new good solutions. With an elite set size of 1, path relinking is essentially disabled while bigger elite sets allow greater variability in terms of distance between elite solutions.

Figure 14 and Figure 15 show us that path relinking has an apparent negligible effect in terms of execution time and also does not appear to have any significant effect in terms of quality of the solutions found. This leads us to believe that the combination of GRASP and Tabu Search is already sufficient to explore most of the optimal solutions in the solution space of this particular data set. However, since the effect in execution time is practically non-existent, it's still useful to leave it active as a safety net in case grasp+ts fail to reach solutions of adequate quality.

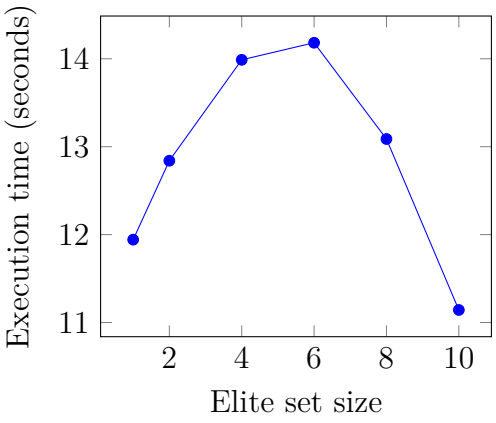

Figure 14: Execution time for different values of the elite set size.

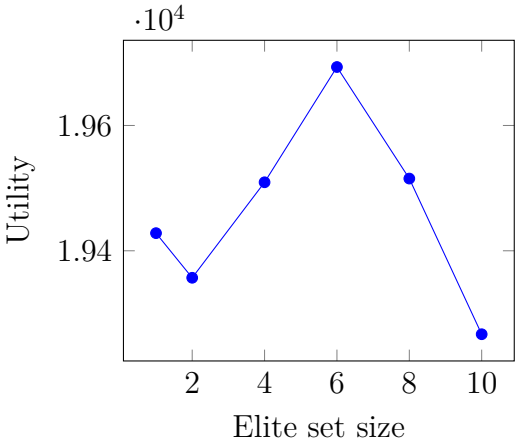

Figure 15: Maximum utility found for different values of the elite set size.

5.4.6 Time limit

In this experiment, we explore the effects on solution quality resulting from the limiting of execution time. We ran this experiment with a bigger data set composed of 100 hosts and 300 jobs which, without a time limit, takes approximately 10 minutes to run using GRASP+TS with the default parameters. Figure 16 shows what was already expected: the stronger the limit on execution time, the lower the solution quality. However, the increase in the utility of the best solution found appears to experience a logarithmic growth which means that the difference in solution quality from letting the solver execute in 5 seconds or 32 seconds is way bigger than the difference with execution times of 32 seconds and 60 seconds.

5.5 Simulated Annealing

Due to it's simplicity, the Simulated Annealing heuristic has only 3 different parameters you can adjust. In our specific case there is also the CPU increments parameter for the solver. Since we already measured the affect of this parameter on the GRASP+TS heuristic, we decided to use the optimal value measured there, which is 25.

The three parameters we measured are:

- α - the factor with which the temperature is kept
- T_0 - the initial temperature
- T_{min} - the minimum temperature

5.5.1 Factor to Keep the Temperature

As we expected and as you can see in Figure 17 and Figure 18 the closer the factor (α) is to 1, the more iterations the heuristic does and the better the results we get. Of course, the computation time also goes up with the number of iterations.

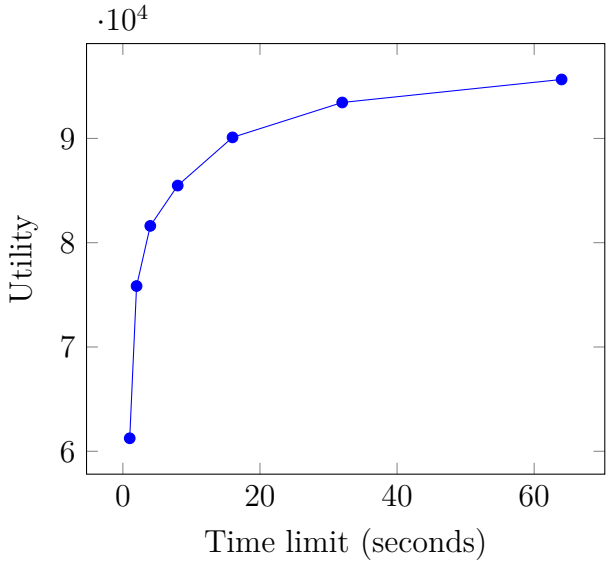

Figure 16: Maximum utility found for different values of time limit for the execution of GRASP+TS.

For C# there are 2 different values for the different construction heuristics (greedy and random). However the solver with the randomly constructed initial solution only starts to create feasible solutions from $\alpha \geq 0.999$.

Also we can see that there is not too much difference between the C++ and C# implementation. In addition, the greedy construction phase generates quite some overhead in the beginning. However, this overhead is worth it as it is able to find much better solutions.

5.5.2 Temperature Boundaries

As we have already mentioned, the temperature boundaries T_0 and T_{min} describe how much worse the utility can be for us to still accept a worse solution.

As shown in Figure 19 we had the best results with $T_0 : 10$ and $T_{min} : 1$. These low temperatures didn't allow the heuristic to search in neighbourhoods with too bad solutions and focused it on better solution spaces. Higher temperatures allowed the heuristic to search in any neighbourhood and, with a high T_{min}, the heuristic basically turned into a random search.

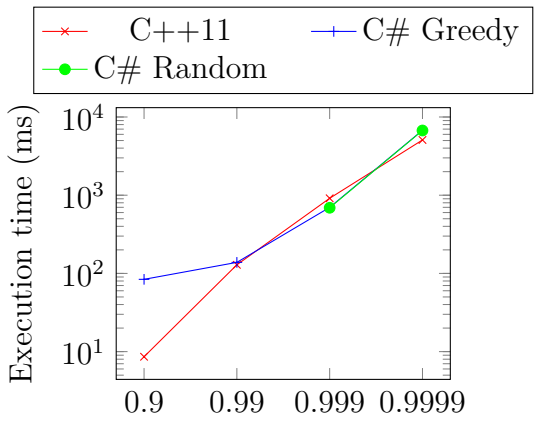

Figure 17: Different execution times for different values of α

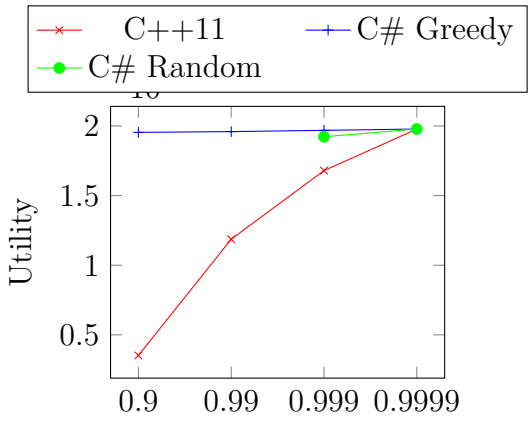

Figure 18: Different utilities computed for different values of α

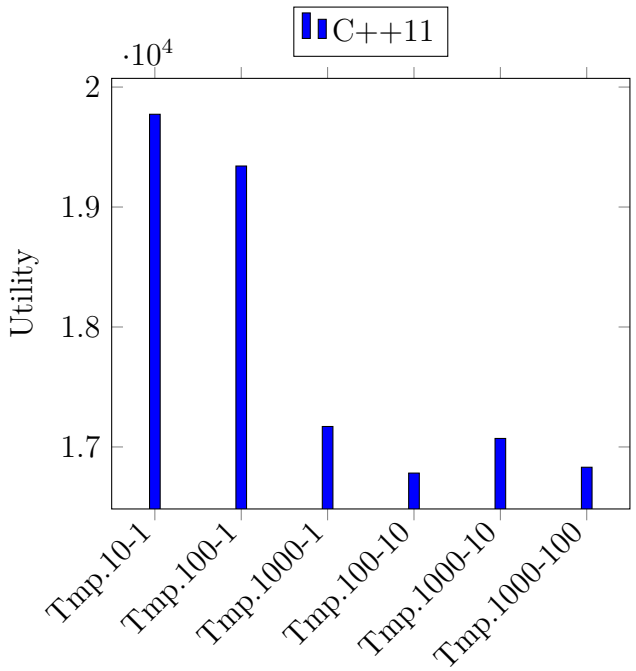

Figure 19: Difference of utilities by temperature in C++11 for α 0.9999

6 Conclusion

Based on our collected results, we can say with absolute certainty that the use of ILP solvers for obtaining realtime solutions to complex problems like the one we studied is not feasible. Where those solvers win in terms of accuracy, they more than lose in terms of execution time.

The answer to obtaining realtime solutions for these problems relies on heuristic solvers. While these seldomly give the best possible solution, the solutions they provide usually have tolerable error margins (less than 4%) and are found tens or even hundreds of times faster than their ILP counterparts.

Nevertheless, heuristic solvers have lots of different parameters which have to be carefully tuned. While a good generic solver might be obtained by choosing balanced values for these parameters, it may be possible to obtain better solutions by using different parameters for different data sets (refer back to subsubsection 5.4.1 for an example). These different parameters might be previously computed and applied based on properties of the particular data set being calculated or may be discovered through machine learning techniques in the actual solver which learn from past executions and autonomously decide which parameter values to use.

References

[1] Fit, J. Oriol, ênigo Goiri, and Jordi Guitart. “SLA-driven elastic cloud hosting provider.” In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, pp. 111-118. IEEE, 2010.

[2] Goiri, Inigo, Ferran Julia, Ramon Nou, Josep Ll Berral, Jordi Guitart, and Jordi Torres. “Energy-aware scheduling in virtualized datacenters.” In Cluster Computing (CLUSTER), 2010 IEEE International Conference on, pp. 58-67. IEEE, 2010.