Middleware Processes Optimization

Pedro Pexirra
Instituto Superior Técnico – Universidade de Lisboa

***Abstract* — this work address research and development of an application created to solve a specific optimization problem regarding de execution of middleware maintenance Tasks which often are difficult to plan due to time, resource and dependency constraints.**

The developed solution maps the optimal schedule using a Branch and Bound approach along with a graph based model, searching algorithms and specific heuristics to improve its performance in various possible scenarios. The input is received in an Excel document with the Task details and the results are presented in a HTML document containing the schedule with minimum loss.

Keywords: *Middleware, Scheduling with constraints, RCPSP, Branch and Bound, Graph search algorithms*

1 INTRODUCTION

Big companies nowadays depend on various systems communicating between them to perform various Tasks. To facilitate that communication many of the companies introduce another component in the fold which serves as a middle-man. It is aptly called Middleware or Integration System.

Like any other system inside the company, the middleware needs to be maintained and updated as the other systems evolve and the requirements change. Most of the time, these changes have temporal constraints, resource constraints (human or computation wise) and dependencies between them, which makes it difficult for the responsible teams to plan their execution.

These changes are called Integration Tasks and are included in a class of well-known problems called Resource-Constrained Project Scheduling Problem.

Each Task is defined as a group of activities that must be executed in order to complete a specified purpose.

It has the following properties:

Begin date – day it's execution begins

Limit date – last day for the Task to be completed without loss

Daily Loss – the amount that is lost for each day the Task is not completed after the limit date

Each is Integration Task is composed of activities, called Phases that can range from development, to documentation or installation or whatever activity is needed to perform for the execution of the Task. Only when all of its Phases are executed is the Task completed.

A Phase has the following properties:

Begin date – day it's execution begins

Duration – the amount of time required to execute the Phase

Resources – A list of resources required to execute the Phase

Dependency – A list of other Phases that need to be completed before this one can be executed (can be empty)

The Resources mentioned above can refer to human resources like programmers, testers or supporters and it can also refer to computational resources like machines, cpu or ram or whatever finite resource is needed to complete a Phase.

The presentation of an application that optimally schedules these Integration Tasks, is the focus of this paper.

This paper is organized as follows. In section 2, an overview of the related work in optimization algorithms is presented. Section 3 introduces the implemented solution. Then, in section 4 the evaluation is illustrated with some case studies, and, finally, in section 5, the conclusions are shown.

2 RELATED WORK

There have been many approaches for solving optimization problems with constraints, like the RCPSP, being one of the most famous the simplex algorithm of linear programming problems

Linear programming is a method used to solve optimization problems when we require the maximization or minimization of an objective function subject to a limited number of resources under some type of constraints.

To solve the problems it's necessary to specify our objective as a linear equation and the constraints on the resource as a group of equalities or inequalities over the variables in our objective equation.

For example, we can formulate a linear programming problem to find the x and y values that maximize the sum of x +y under the constraints:

$$x + 3y \leq 5$$

$$4x + y \leq 10$$

$$-x + y \leq 1$$

$$\text{with } x \text{ and } y > 0$$

The solution in a simple case like these is obtained through analysis of the border of the "walled off" section of the graphic obtained through the inequality constraints, searching for the intersection points that maximize (or minimize) the value.

The method is used to solve various real world problems such as maximizing the profit in a company or Task scheduling minimizing costs

Another process applied in solving optimization problems is dynamic programming algorithms. It's a recursive method that solves an optimal solution to a problem by solving the solution to its sub-problems.

One of the most interesting properties of this approach is the fact that a solution to a sub-problem is only evaluated once, being stored in a table or similar structure to be accessed whenever the sub-problem shows up in a recursion.

Dynamic programming can be used when the optimization problem has the following characteristics:

Optimal Substructure – when the optimal solution for a problem contains within it optimal solutions to its sub-problems

Overlapping sub-problems – when the number of different sub-problems is small the solution tables are accessed more often.

One of the most popular techniques for solving these types of scheduling problems is the Branch and Bound method. It consists in enumerating all the possible schedule solutions within a set limit. The result of this enumeration is often thought of as a tree in which each leaf represents a sequence of execution of the various activities.

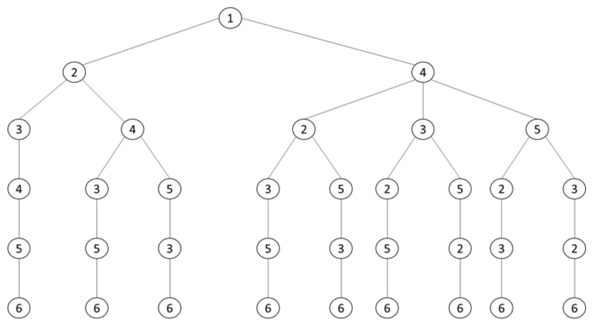


Figure 1: Scheduling tree

The number of branches in the tree grows quickly with the number of activities to schedule. To fight this node explosion and decrease the size of the tree this method is often used along with several pruning heuristics.

Graphs are also very popular representation methods for optimization problems like the RCPSP, due to its particular intuitive structure for relationship modelling and the large amount of algorithms that exist to search through their edges and discover vertexes.

An interesting property of graph searching algorithms is the ability to find the shortest routes between vertexes, like the BFS and DFS for non-weighted graphs and the Dijkstra for weighted graphs.

Simulated Annealing algorithms are another approach widely used for solving RCPSP problems. A process analogous to the metallurgy process with the same name, which consists of heating and then gradually cooling down a material in order to strengthen it.

Beginning with a random solution, the algorithm works by trying to find a better solution in the neighbourhood of the current one. The concept of gradually cooling down is achieved using a variable which represents the probability of a worse solution being chosen and thus decreasing the chance of a local optimal solution being found.

As the algorithm progresses the variable is decreased and the algorithm converges to an optimal solution.

Genetic Algorithms are also a common technique for tackling complex optimization problems like the RCPSP.

They are search methods that try to mimic natural occurring phenomenon like biological evolution, being the premise that the fittest survive and the weakest do not.

Generally they are defined as iterative procedures in which a population of candidate solutions is maintained and with each iteration of the algorithm the fitness of the members of the population is evaluated and based on those fitness values a new population is generated. The new candidate solutions are obtained by applying a number of genetic operators to the current one, like reproduction, mutation or crossbreeding.

3 TASK SCHEDULING

This section explains the algorithm used to solve the loss minimization problem and the main application components.

A Branch and Bound technique along with a set of heuristics was developed, using a graph modelling of the problem, where:

Each vertex has information about a scheduling of Phases.

Each edge has a weight regarding the loss that exists between the schedule in the destiny vertex and the origin vertex.

An optimal Task scheduling solution is obtained following the shortest path along the edges until we reach a vertex where all the Phases are scheduled.

A vertex is then represented as a **Scheduling State**, defined by the existing Tasks and available resources

$$E = [\,T_1, T_2, \dots, T_N \;-\; R_1, R_2, \dots, R_M\,]\; where\ N = nº\ tasks\ and\ M = nº\ resource\ types$$

A Task is made of a group of Phases

$$T_i = \{\,f_1, f_2, \dots, f_n\,\}\;\; where\ n = nº\ phases\ of\ the\ task\ T_i$$

$$f_i = \begin{cases} -1, & phase\ not\ scheduled \\ d, & phases'end\ day, d > 0 \end{cases}$$

A Task is scheduled if all its Phases end day is bigger than 0.

And for each resource type

$$R_i = \{\,r_1, r_2, \dots, r_m\,\}\;\; where\ m = nº\ resources\ of\ type\ R_i$$

$$r_i = \begin{cases} -1, & resource\ not\ allocated \\ d, & resource\ allocation\ end\ day, d > 0 \end{cases}$$

The initial scheduling state is the states where all Phases are not scheduled and all resources are not allocated

$$\forall_{f \in E[T]}\, f = -1\ e\ \forall_{r \in E[R]}\, r = -1$$

Two scheduling states are said to be equivalent if the scheduling day of their Phases are equal

$$\forall_{f \in E[T]}\ e\ \forall_{f' \in E'[T]}\, f = f'$$

An edge is a **Transition** between two scheduling states. Two states connected by a Transition can only differ in the value of one Phase’s day. This means we will only schedule one Phase at a time. The weight of the edge (or Transition) is the value of the loss represented between the scheduling of the two Phases.

Using the concepts of Schedule State and Transition, the minimal cost complete schedule is obtained by building a graph representing a set of possible complete schedules and then finding the shortest path between the first node and the one with a complete schedule.

The base version of the algorithm follows the following steps:

Creation of the initial scheduling state;

Process Schedule State (while states to be processed exist):

- Obtains Phases that can be scheduled;
- For each Phase generates a new Schedule States from the current one where that Phase is scheduled;
- Adds the states to the graph (if they are not in it already);
- Adds transitions between the current state and the new ones;
- If any of the new states is a complete schedule its marked as such;

When there are no more states to process, adds a final state to the graph and transitions from all the complete Schedule States to it (this is necessary due to the choice of shortest path algorithm to use: The Dijkstra Algorithm);

After that, the shortest path between the initial state and final state is computed.

And finally the optimal schedule is built by going through the shortest path and analysing each of the Schedule States.

Task	Limit Day	Loss per day
Task 1	5	100
Task 2	15	500

Resource	Quantity
Programmer	2
Machine	2

Task	Phase	Duration	Precedence	P	M
Task 1	F11	5	-	2	1
Task 2	F21	3	-	2	2
Task 2	F22	3	F21	1	1

Table 1: Tasks input example

For the above example of Tasks, the following results would be obtained.

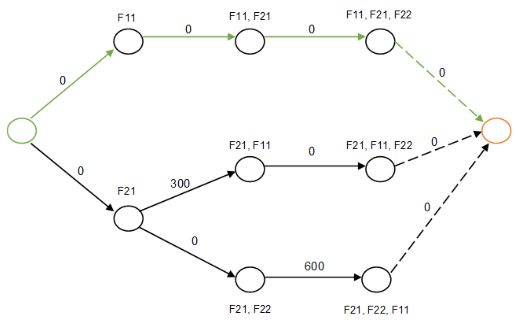
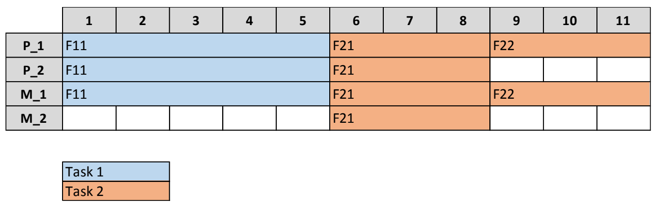


Figure 2 Schedule graph example

.



	1	2	3	4	5	6	7	8	9	10	11
P_1	F11					F21			F22		
P_2	F11					F21					
M_1	F11					F21			F22		
M_2						F21					

🟦	Task 1
🟧	Task 2

Figure 3: Minimal schedule Gantt chart

We can guess that like this, this algorithm will perform poorly except for the most trivial cases. This is because we are generating many possible combinations of scheduling states and a good part of those are not part of an optimal scheduling solution.

To fight this and prune the number of generated Schedule States, a number of heuristics were developed.

Limit the allowed loss per scheduled Phase

The first heuristic is the usage of allowed losses per scheduled Phase, meaning that a restriction in the allowed weight of a Transition was introduced.

Initially the allowed loss is 0. If the algorithm cannot find a completed Schedule State then the allowed loss is increased by a minimum value and the search starts again from the beginning.

$$\begin{aligned}&new\ loss\ limit\\&= actual\ loss\ limit\\&+ minimum\ loss\ required\ to\ schedule\ a\ phase\end{aligned}$$

Using these limits in the previous example we would obtain the following schedule graph.

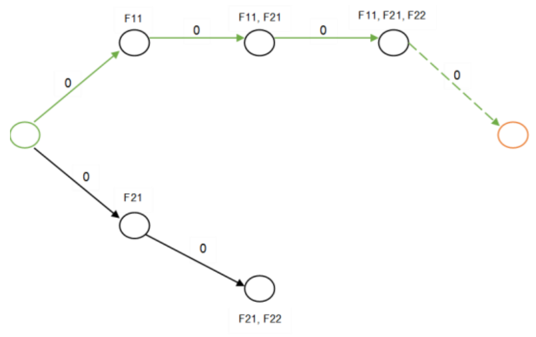


Figure 4: Schedule graph using limits

Discard of invalid Schedule States

Looking at the figure above, we can see that if it is not possible to schedule the Phase F11 immediately after the Phase F21 then we will never be able to schedule F11 through a sequence of schedules in which F21 was scheduled first.

Using this knowledge, we can increase the performance of the algorithm and reduce the number of generated states by discarding these types of schedule states. Applying this heuristic to the example we obtain the following schedule graph:

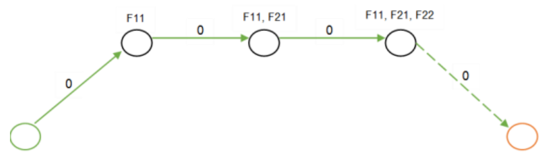


Figure 5: Schedule graph with invalid states discarded

Priority Phases and Potential States

To decide the order in which the algorithm processes the Schedule States

Which Phases should be scheduled first?

- The ones that have a lower limit date
- The ones that represent a higher loss per day
- The ones that are a requirement for other Phases

With these factors taken into account the following formula for quantifying the priority of a Phase was reached:

$$Phase\ priority = \frac{\log(TaskLossPerDay)}{TaskLimitDay} + priority\ of\ the\ Phases\ for\ which\ it\ is\ a\ requirment$$

Having quantified the importance of a Phase, we can now establish that the possibility of a Schedule State originating another one of minimal loss increases along with the increase of number of Phases (of high priority) that are scheduled within it. To the value that represents this possibility we call Potential and it is given by the expression:

$$State\ potencial = \sum priority\ f\ , for\ \forall_{f \in E[T]}\ where\ f > 0$$

Validate existent schedule

This heuristic consists on doing an extra validation before processing a Schedule State. If the schedule contained in the state we are about to process already exists in another state of the schedule graph, then it is not processed. This is used to avoid multiple calculations of schedule sequences that lead to the same final schedule.

It is more effective if used along with the Potential States technique.

Break: Trying to schedule with loss

To avoid the building of complete schedule graphs when an optimal solution is not possible to find within the allowed loss limit a maximum value for the number of times one can try to schedule a Phase that exceeds the allowed loss. When this break value is reached, the algorithm stops, re-calculates the allowed loss limit and begins the search using the new limit.

The value of this break variable is studied in the evaluation section.

The ideal configuration for this value would be 1, meaning that if an attempt to schedule beyond the allowed loss limit is made, then it is not possible to reach an optimal solution using the current loss limit.

Break: Number of complete states found

The possibility to stop the building of the graph when a set number of completed scheduled states are found was also implemented.

The value of this break variable is also studied in the evaluation section.

And the ideal configuration for this value would also be 1, which means that the first complete Schedule State reached is an optimal one (one with minimal loss).

4 EVALUATION

It's not possible to completely, or even exhaustively, test the developed application due to the infinite combination of inputs. As such it was decided to test the application through several test cases created to validate the behaviour of the application under a specific set of conditions.

What we intend to evaluate:

Results:

- Obtained schedule is one of minimal loss
- Resources are correctly occupied
- Precedence relations are held

Performance:

- Execution time
- Used memory

- Correct result with minimal calculations

With and without using the break parameters for number of complete states found or number of times the loss limit was exceeded while building the schedule.

Case 1:

6 Tasks, 17 Phases and 8 Precedence relations

Minimum loss schedule possible with only one iteration of the build of the graph

Break: Limit	Break: States	Loss	Time (s)	Graphs Built	Total States	Total Transitions
0	0	0	10,79	1	2887	3182
0	1	0	0,72	1	110	109
1	0	0	10,66	1	2887	3182
1	1	0	0,70	1	110	109

Table 1: Results of Test Case 1

Looking at the first line we see that it was possible to find a minimal loss schedule building only one graph. In the second line, that the first completed schedule found is one of minimal loss and that it has a considerable better than when the graph is fully built. In the third line, that during the graph construction there were no attempts to schedule Phases which incurred loss and in the fourth we confirm what could already be concluded form the other lines, that using [1|1] breaks we produce an optimal schedule with superior performance.

Case 2:

20 Tasks, 60 Phases and 0 Precedence relations

Existence of only one possible minimal loss schedule

Break: Limit	Break: States	Loss	Time (s)	Graphs Built	Total States	Total Transitions
0	0	-	-	-	-	-
0	1	0	54,15	1	1832	1831
1	0	-	-	-	-	-
1	1	0	58,82	1	1832	1831

Table 2: Results of Test Case 2

It was verified that the execution of the algorithm does not end in useful time in the situations in which is required to fully build the graph. This happens due to the lack of precedence relations and the high number of activities to schedule.

When using the [1|1] break configuration the optimal schedule is found.

Case 3:

10 Tasks, 28 Phases and 16 Precedence relations

Minimum loss: 4700, several graph builds required

Break: Limit	Break: States	Loss	Time (s)	Graphs Built	Total States	Total Transitions
0	0	-	-	-	-	-
0	1	-	-	-	-	-
1	0	-	-	-	-	-
1	1	41450	1,34	2	285	283
28	224	4700	24,91	6	5437	5654
56	224	4700	25,897	5	5240	5458
54	448	4700	44,80	5	7823	8265
112	448	4700	67,37	5	11006	11448

Table 3: Results of Test Case 3

For the first three break configurations the program did not complete in useful time.

And in this case, for the [1|1] break configuration and optimal solution is not found. The potential states calculation is not optimal.

		Break: States						
		1	28	56	112	224	448	896
Break: Limit	1	41450	40400	40400	39400	39400	34650	32750
	28	10750	8700	8300	8650	4700	10600	9900
	56	10750	8700	8300	7500	4700	4700	9900
	112	10750	8700	8300	7500	4750	4700	9900
	224	10750	8700	8300	7500	4750	4700	4700
	448	10750	8700	8300	7500	4750	4700	4700
	896	10750	8700	8300	7500	4750	4700	4700

Table 4: Break trials for Test Case 3

After experimenting with the break values we verify that the minimal loss solutions are found as we increase both values of the breaks. Being that the biggest concentration of optimal results is found when the "Break: Limit" is bigger than 56 and the "Break: States" is equal to 448.

Analysing the obtained results, we notice that even though for some cases it is possible to obtain an optimal solution using the [1|1] parametrization for the Breaks that does not happen in all the cases. This means the Priority and Potential calculus is not adequate for every case.

Looking at the experimentation table of the Case 3, we verify that there must be a balance between the values assigned to the breaks. If we assign a large value to the completed states break, like the one in entry [28|448], there is the risk of building graphs where the allowed loss is too high, this happens because the value chosen for the states break is too high and the limit break is always reached first and thus the schedule graph is rebuilt allowing for more and more schedules that are not part of the optimal solution.

An ideal parametrization for these breaks is not obvious, for it depends greatly on the type of input received, it is necessary to perform more tests to study these values, something that is not simple due to the difficulty of creating the test cases with known solutions and executing them.

5 CONCLUSIONS

This paper proposed the development of an application capable of solving a scheduling problem relating to the problems arising in middleware process development nowadays. To achieve this we defined the concept of Scheduling State and several algorithms for optimal schedule search using a Branch and Bound approach with a graph model for representation along with several heuristics for node pruning. Besides the algorithms, modules for input processing through Excel documents, results presentation through html pages and logging where also implemented.

The obtained results show that for every case the resource occupation is always correct and the precedence relations are always upheld.

Without the use of Breaks in the building of the schedule graph the optimal solution is only obtained in useful time for simple cases, with few activities or many dependences among them.

Using Breaks while searching for the optimal solution greatly improves the application performance, but for the current Priority and Potential calculations it is not obvious the parametrization of these values.

We can also conclude that the developed application, having been motivated by the scheduling of Tasks for a middleware development team, is generic enough to be applied for similar RCPSP,

REFERENCES

[1] David S. Linthicum, Enterprise Application Integration, 1st edition, Addison Wesley, 1999.

[2] Sacha Krakowiak, Middleware Architecture with Patterns and Frameworks, 2009.

[3] Christian Artigues, Sophie Demassey , Emmanuel Neron, Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications, Wiley-ISTE, 2008.

[4] Sönke Hartmann, Project Scheduling under Limited Resources, Springer, 1999.

[5] Richard Walter Conway, William L. Maxwell, Louis W. Miller, Theory of Scheduling, Courier Corporation, 2003.

[6] Peter Brucker, Scheduling Algorithms, Springer, 5th Edition, 2007.

[7] J. Blazewicz, J.K. Lenstra e A.H.G. Rinnooy Kan, “Scheduling Subject to Resource Constraints: Classification and Complexity,” Discrete Applied Mathematics 5, pp. 11–24, 1983.

[8] James M. Crawford, "An Approach to Resource Constrained Project Scheduling,"

[9] Michael Held, Richard M. Karp, "A Dynamic Programming Approach to Sequencing Problems," Journal of the Society for Industrial and Applied Mathematics, Vol. 10, No. 1, pp. 196-210, 1962.

[10] Peter J. M. van Laarhoven, Emile H. L. Aarts, Jan Karel Lenstra, "Job Shop Scheduling by Simulated Annealing," Operations Research, Vol. 40, No. 1, pp. 113-125, 1992.

[11] S. Diana, L. Ganapathy, Ashok K Pundir, "An improved Genetic Algorithm for Resource Constrained Project Scheduling Problem," International Journal of Computer Applications (0975 – 8887), Vol. 78, No.9, 2013.

[12] Wei Zhao, Krithi Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," The Journal of Systems and Software 7, pp. 195-205, 1987.

[13] W. Herroelen, E. Demeulemeester, B. De Reyck, "RESOURCE-CONSTRAINED PROJECT SCHEDULING A survey of recent developments," COMPUTERS & OPERATIONS RESEARCH, 1998

[14] Rainer Kolisch, Rema Padman, "An Integrated Survey of Project Scheduling," 1997.

[15] Ruey-Maw Chen, Shih-Tang Lo, "Using an Enhanced Ant Colony System to Solve ResourceConstrained Project Scheduling Problem," IJCSNS International Journal of Computer Science and Network Security, Vol.6, No.11, 2006.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd Edition, 2009.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 2nd Edition, 2001.

[18] Robert Sedgewick, Phillipe Flajolet, An Introduction to the Analysis of Algorithms, 2nd Edition, 2013.

[19] Robert Sedgewick, Kevin Wayne, Algorithms, 4th Edition, 2011.