Univerza v Ljubljani
Fakulteta za elektrotehniko

Memory allocation optimization via MILP

Ime Priimek: Danilo Pejović

Predmet: Optimizacija v telekomunikacijah

Mentor: Andrej Košir

Datum: 19.5.2021

Table of contents

1. Abstract

Modern applications are only becoming more complex with each passing day, and that complexity is followed by increase in diversity of available components needed for those applications to work. That necessitates structured method in selection of components as well as resource allocation. This paper tackles part of latter problem. Using Mixed Integer Linear Programming (MILP) this paper will calculate minimum needed time to run certain processes given upper bound of memory allocation and based on that evaluation system designer can decide if he needs to allocate more memory to that process. This paper will be split into three main parts – first will give brief overview of use of our method and brief explanation of Two-Dimensional Rectangular Strip Packing problem (and how it relates to problem paper is addressing), second will contain detailed explanation of methodology and last part will contain results of simulations based on system explained in second part.

2. Introduction

Memory allocation and scheduling influences power consumption as well as overall cost of our system – making evaluation necessary. Unfortunately, until relatively recently memory was seen as less of a variable resource but rather a constraint – which makes sense when looking at history of computing and how increase in available memory quickly rose. That means that in (especially less robust systems, example of that being system on chip, SoC) systems mapping was often left to heuristics like list scheduling.

Focus of the paper will be relationship between assigned memory and time it takes to finish the process. Goal is not just to find quickest way to finish the process but also to be cognizant of memory allocated as long as process is guaranteed to finish before maximum allowed time. During this report memory and time will be referred to as unsigned variables, as report is focusing on their relation.

Presented method can be used as part of starting evaluation for systems – other evaluations are still necessary since this method has very specific point of interest. A lot of work has been done on hard real time systems, which at least tangentially relate to theme of this paper – so methods such as [1] and [2] can serve as either addition or replacement to method presented in this paper, even if they do not tackle problem of scheduling from same angle it is tackled here.

2.1 Two-Dimensional Rectangular Strip Packing problem (2D-SPP)

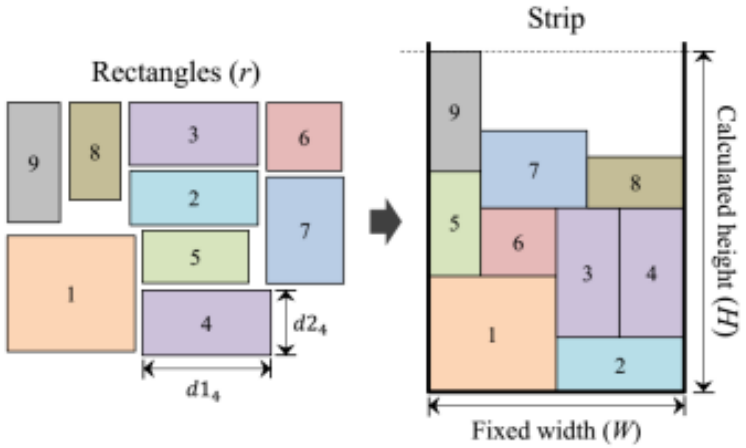

Figure 1. Illustration of (2D-SPP) [1]

Two-Dimensional Rectangular Strip Packing problem is very famous and often studied problem in optimization. Goal is to arrange blocks of differing sizes, so they take us as little of area as possible given a constraint (shown in Figure 1. problem has fixed width and variable height).

2D-SPP is NP-complete which due to similarity with problem this paper is addressing means that MAO problem also carries that label – meaning that complete range of solvable problems in finite/reasonable time is limited.

2D-SPP has come constraints that have to be considered when solving it [3]:

- Objects must not overlap.
- Objects have to be fully contained in the strip.
- Objects can be rotated by n times 90˚, where n is an integer

Major difference between memory allocation optimization (MAO) and 2D-SPP is that MAO contains processes that can only be placed after certain other process has finished which has to be properly accounted for when evaluating the input. Also, we are unable to just flip our rectangles as in our case their dimensions have physical meaning.

3. Formulation of the problem

3.1 Assumptions

While assumptions taken during this paper are mostly reasonable it still is worth it to point them out.

1. Every process contains explicit start and an end – as in, there is obvious sub-process that will be carried out first and obvious sub-process that will be carried out last.
2. Sub-processes cannot be interrupted. Every sub-process that starts must finish before we remove it from memory.
3. Application is given as a graph/algorithm which explains dependencies between sub-processes.
4. Time sub-process takes to complete is not only active part of process but also time it takes to instantiate variables as well as ''clean up'' after itself when it is finished.
5. Method assumes that it was given maximal overall execution time as input. That means that method should not be given for/while loops of sub processes but instead time it takes and memory that that for/while loop requires in worst case scenario should be given as single sub process.

3.2 Constraints based on maximum values

Condition mentioned when explaining 2D-SPP that all objects have to be inside the strip hold true here too. That means that every object cannot occupy memory address higher than we allocated to the process and that object must finish before allocated time. That is how we get first 2 equations:

$$x_i + t_i \leq T \quad (1)$$

$$y_i + h_i \leq H \quad (2)$$

Where x_i is time when sub-process starts, t_i is time it takes to finish that sub-process, T is total time allocated to whole process. y_i is starting memory address for a sub-process, h_i is memory that that subprocess requires and H is total allocated memory. Something that is usually implied, but it does not hurt to add is that MILP problems assume that variables are bigger or equal to zero, thus we get 2 more implicit constraints:

$$x_i \geq 0 \quad (3)$$

$$y_i \geq 0 \quad (4)$$

3.3 Constraints based on relations to other sub-processes

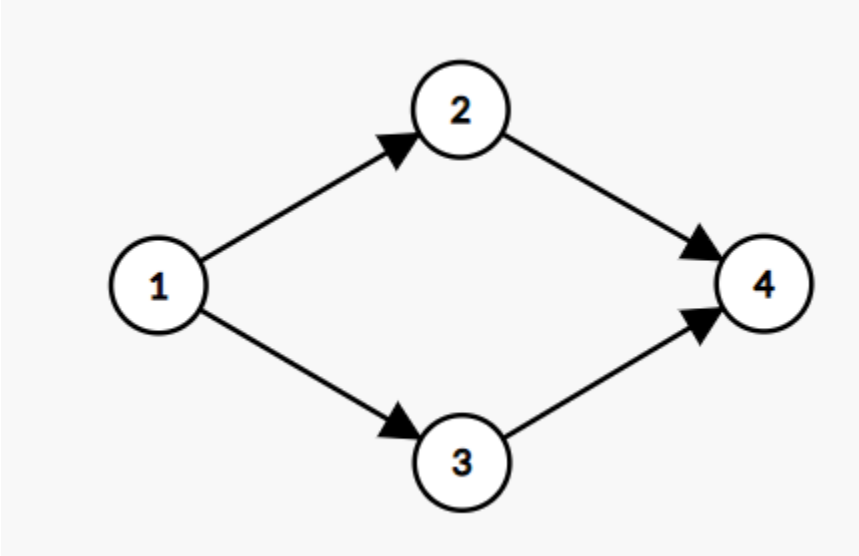

Figure 2. Example of a process

In example above sub-process 4 is dependent on all previous sub-processes, thus other than limitations in (1) and (2) it also has limitation that it cannot start before other 3 finish. For that purpose, two binary variables are introduced:

$$u_{i,j} = \begin{cases} 1\ , when\ sub - preocess\ i\ is\ below\ sub - process\ j \\ 0, when\ sub - preocess\ j\ is\ below\ sub - process\ i \end{cases} \quad (5)$$

$$b_{i,j} = \begin{cases} 1\ , when\ sub - preocess\ i\ is\ before\ sub - process\ j \\ 0, when\ sub - preocess\ j\ is\ before\ sub - process\ i \end{cases} \quad (6)$$

Those variables help us ensure that our sub-processes do not overlap, as it is impossible for 2 sub-processes to use same memory location at the same time. We ensure that overlapping does not happen with following equations:

$$x_i + t_i\ - T\ + T * b_{i,j} - x_j\ \leq 0 \quad (7)$$

$$y_i + h_i\ - H\ + H * u_{i,j} - y_j\ \leq 0 \quad (8)$$

$$b_{i,j} + b_{j,i} \leq 1 \quad (9)$$

$$u_{i,j} + u_{j,i} \leq 1 \quad (10)$$

$$b_{i,j} + b_{j,i} + u_{i,j} + u_{j,i} \geq 1 \quad (11)$$

Where at least one of equations (7) or (8) is not redundant for every pair of sub-processes i and y. [4]

Following all constraints laid out in this chapter and then simply minimizing for start time of last sub-process we would get following chart:

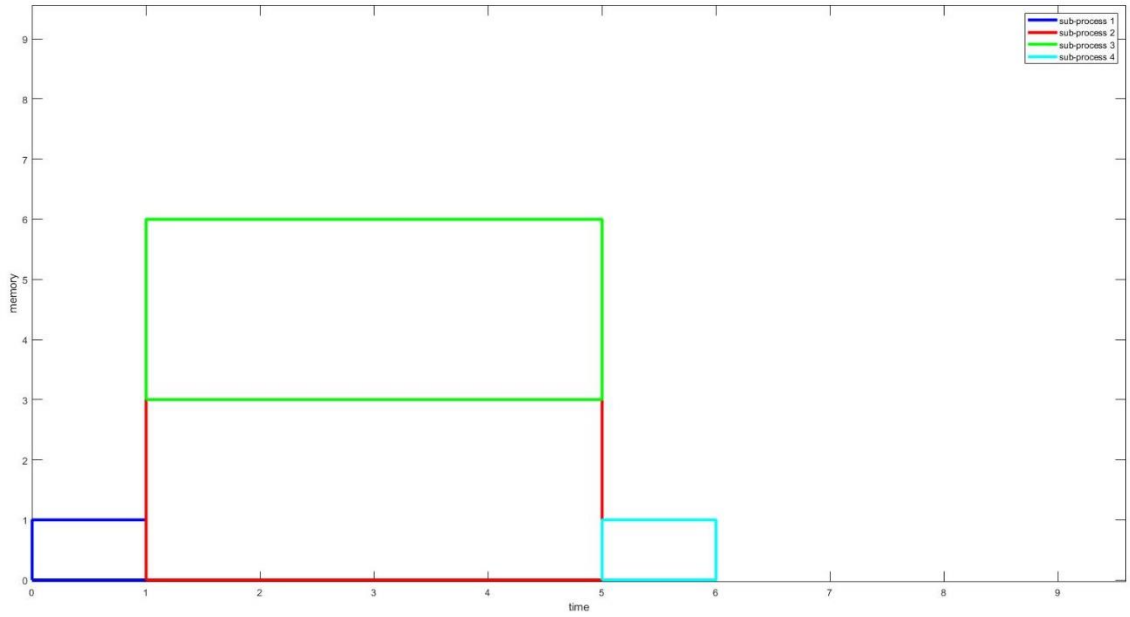

Figure 3. Memory allocation for process in Figure 2

Where we can see that sub-process 2 and sub-process 3 are allowed to run simultaneously, while 1 and 4 are not.

4. Implementation and results

Methodology was tested with few different types of graphs – one representing more realistic process (that has subprocesses that depend on some other sub-process) and one created to showcase usability of this methodology (where none, except starting and ending, sub-processes are dependent on one another).

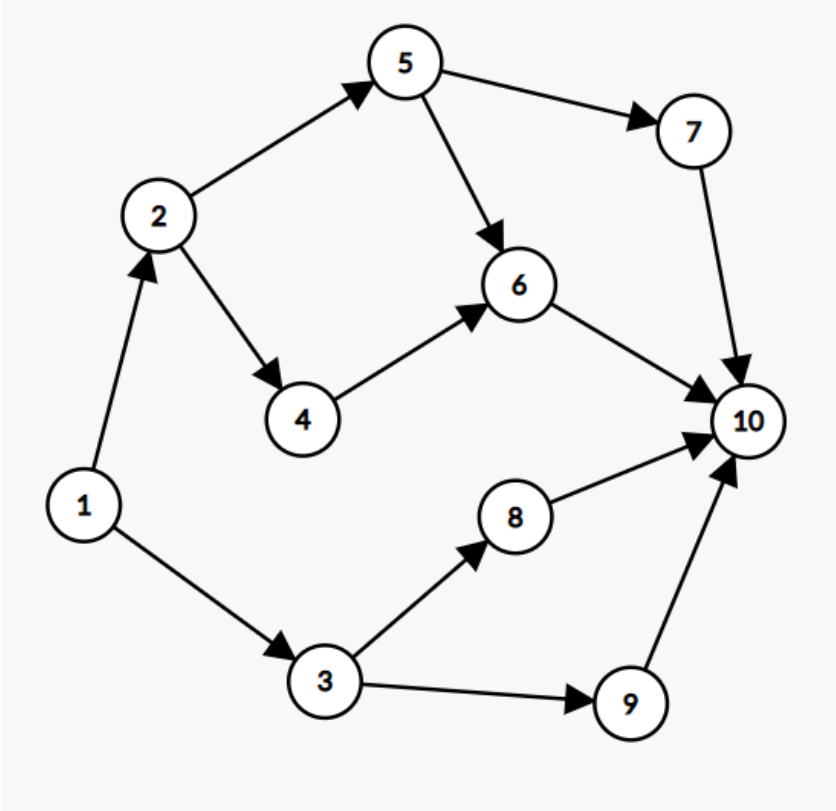

Figure 4. Graph of process with dependent sub-processes

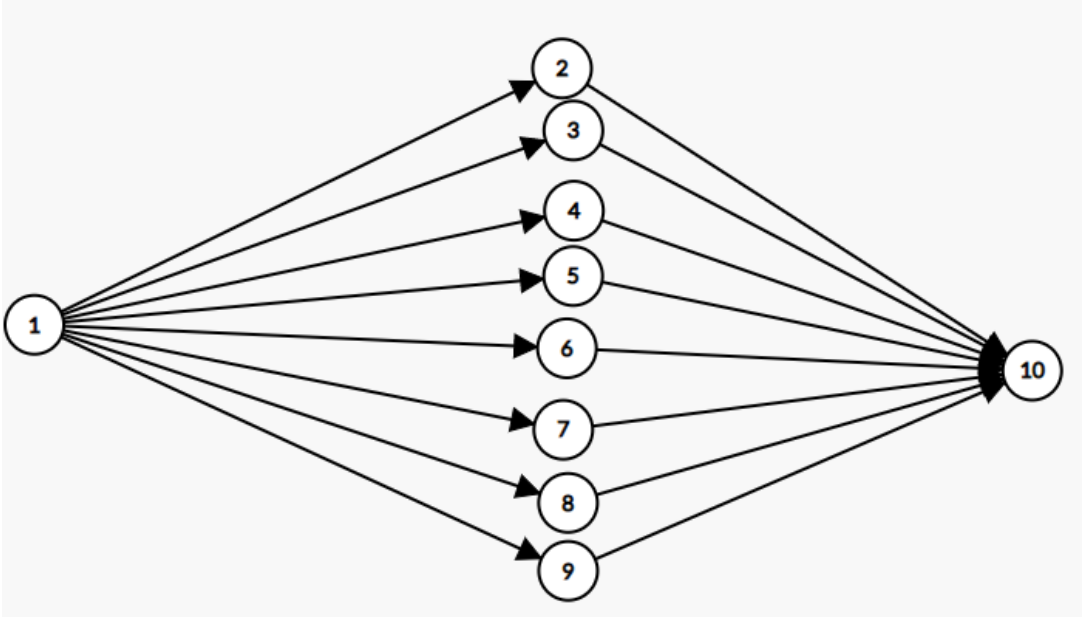

Figure 5. Graph where every process has no dependencies except start and end

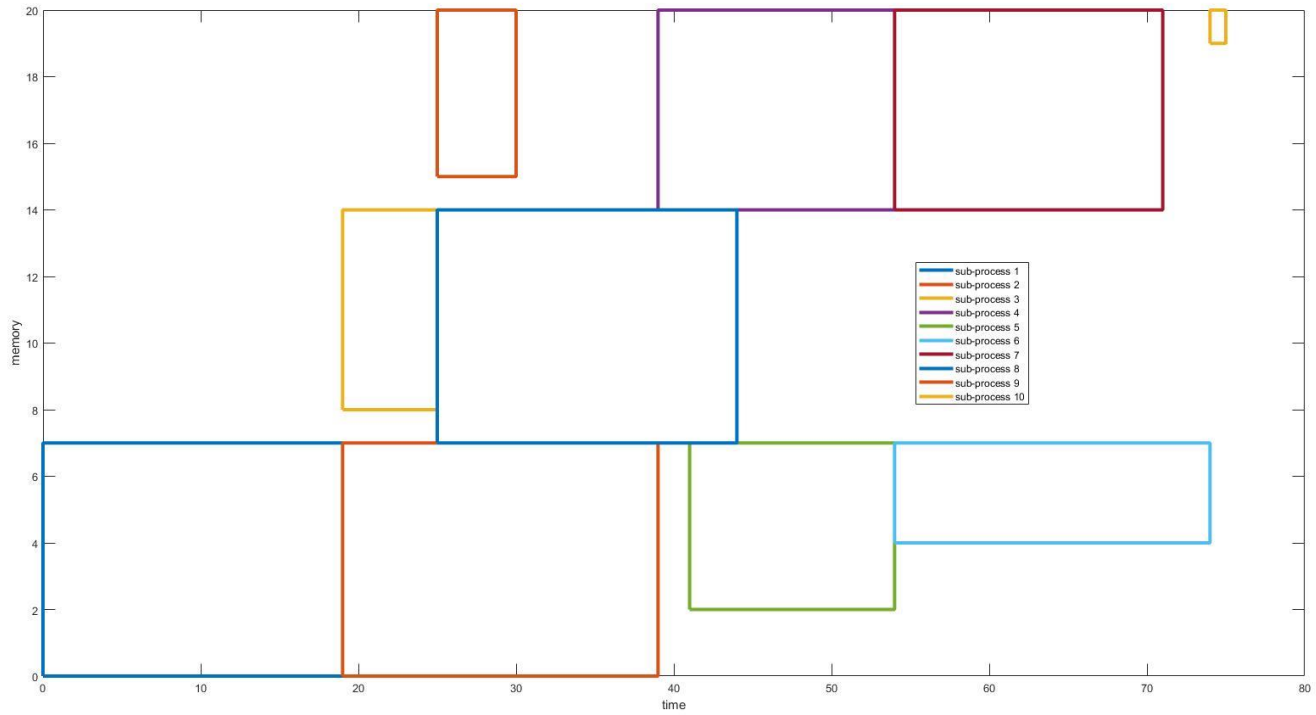

Figure 6. Memory allocation for example in Figure 4 with random variables as t_i and h_i

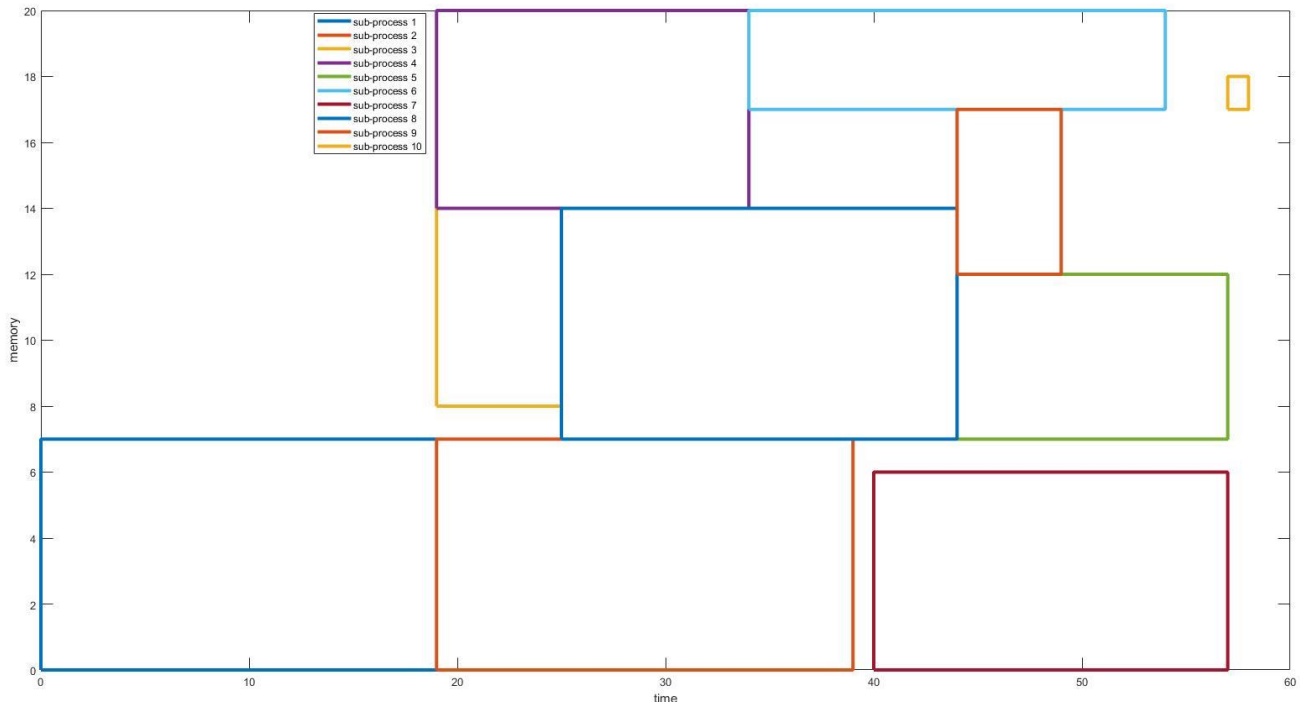

Figure 7. Memory allocation for example in Figure 5 with random variables as t_i and h_i

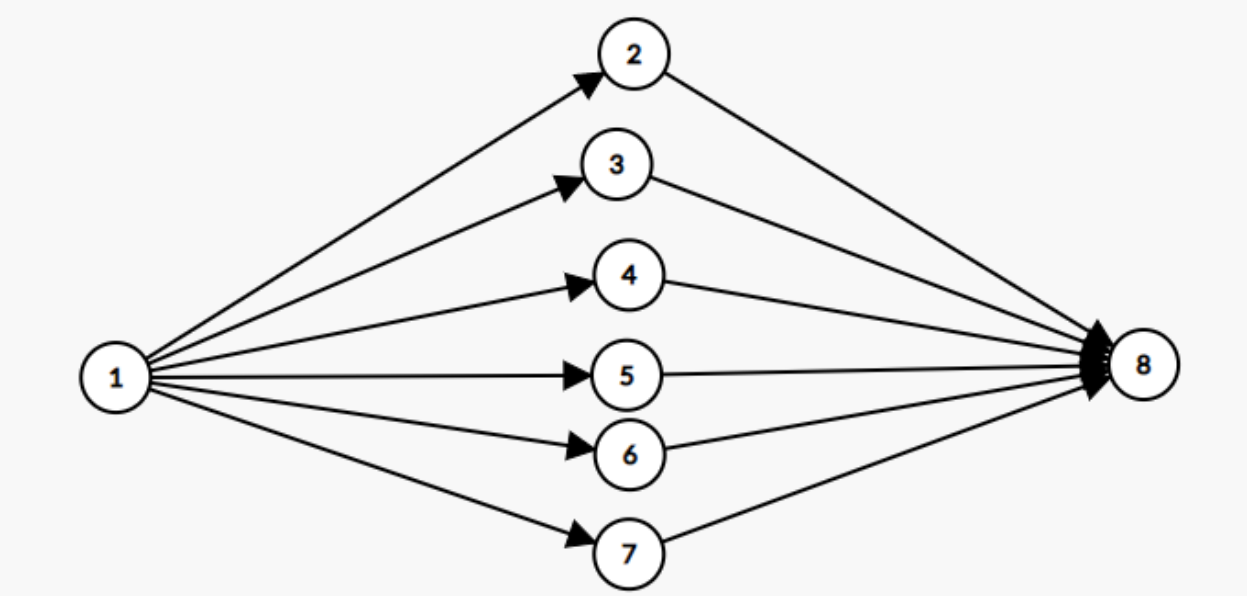

Figure 8. Simple 8 node process

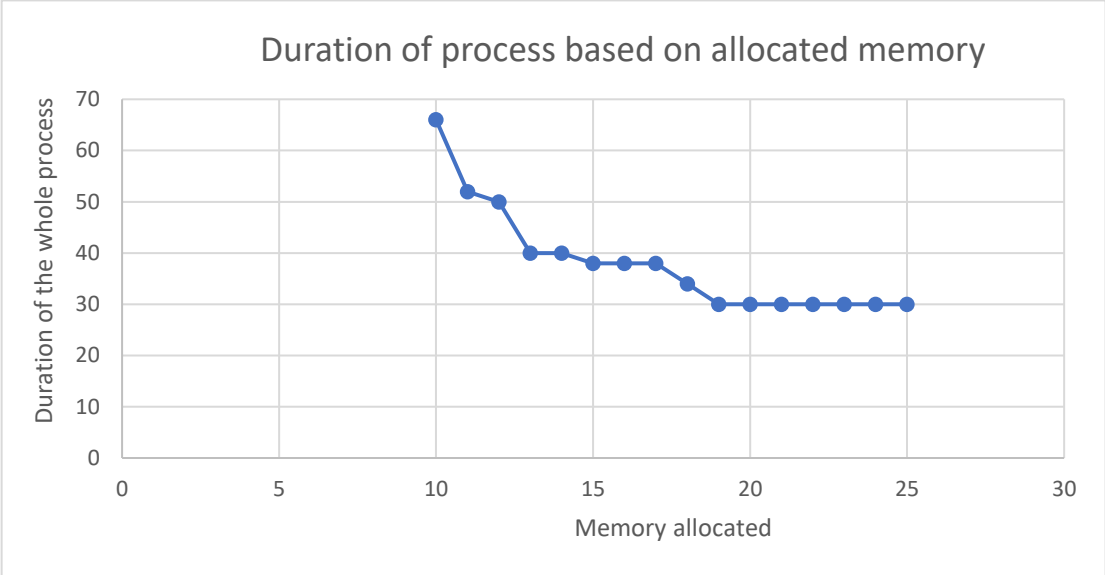

Chart 1. Duration of process based on allocated memory of example pictured on Figure 8.

In Chart 1. method with the same variables was tested with increased memory and duration of process was observed. From the chart, we can see that at some point allocating more memory is simply not worth it as the process stops getting quicker – but at the start, there is a sharp decline in the duration of the process with more memory, so when designing a system, you would probably want to allocate enough memory to take advantage of that decline in duration.

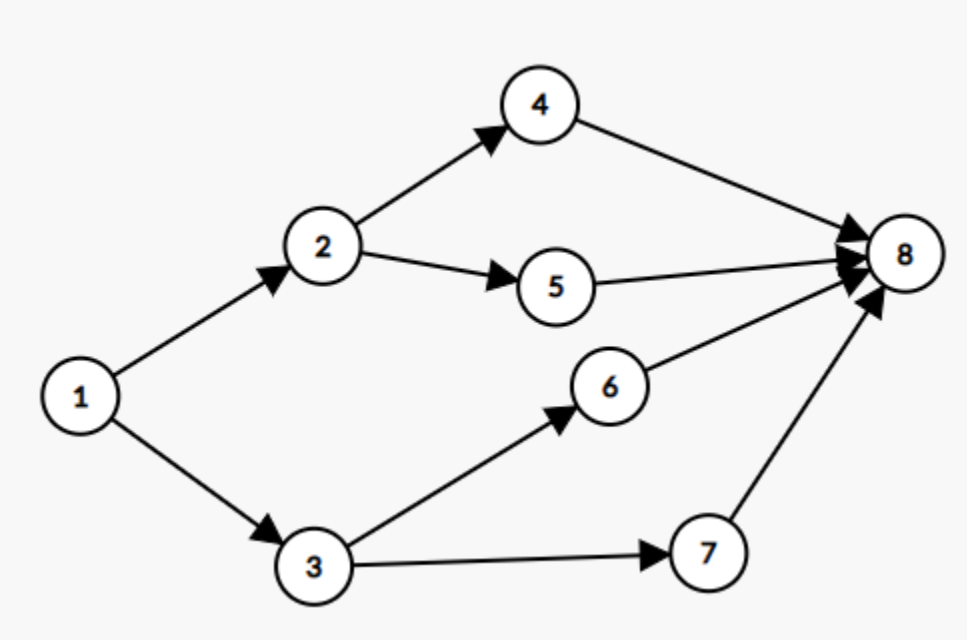

Figure 9. 8 node process

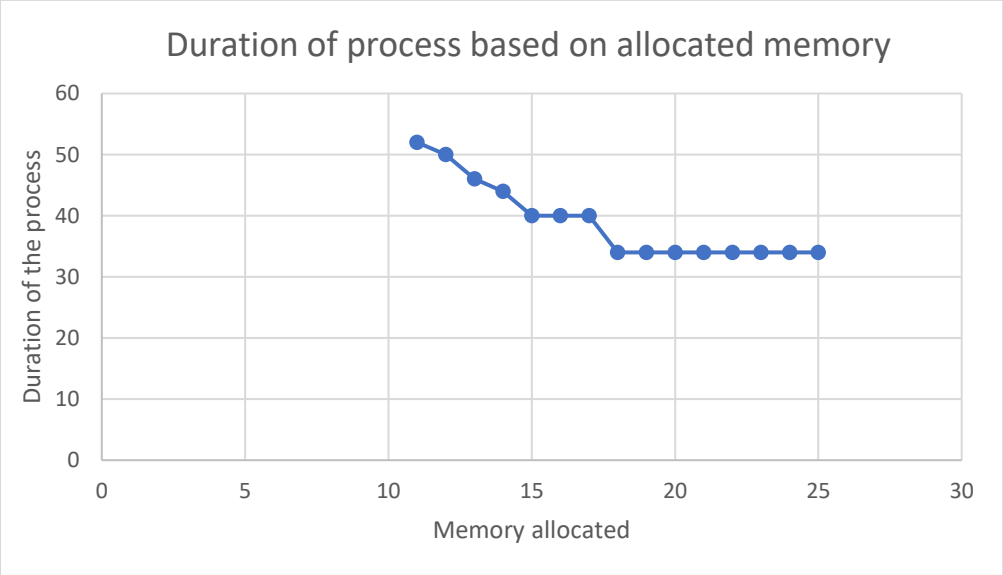

Chart 2. Duration of process on Figure 9 based on memory allocation.

As expected, chart on Figure 9. optimization was a bit less effective – as it already has hardcoded connections, so increases in allocated memory had less impact towards the end as bottleneck was established order of sub-processes and not allocated memory. Similarly, in figures 6 and 7 we can notice that the simpler process (or process that allows more parallel actions) is more effectively optimized.

Program does not always cleanly converge to a desired result when we have a lot of nodes – and will drop into what is effectively infinite loop, at that point it is reasonable to stop the program and take result it gives after 30 seconds- 1 minute (for graphs with less than 20 nodes) as final output. It is worth noting that that may not be optimized solution – for example, in process shown on Figure 5. LP solve would return 64 as optimized time relatively quickly, but if you let program run for unreasonable amount of time it would find solution shown in this report where minimized time is 57.

5. Conclusion

This paper has demonstrated a methodology for memory allocation of processes of very small size. As the program tries to evaluate processes with significantly more than 15 nodes, it is prone to getting stuck in calculations that last well over 30 minutes. That could be result of weaker machinery, but it could also mean that there is some flaw in the design of the system or that some problems are inherently not solvable in reasonable amount of time (as mentioned when explaining 2D-SPP).

Further improvements could be splitting every sub-process into few parts (like initialization, active use, and removal) which would interact with other sub-processes differently. As seen when evaluating graph on Figure 9. it is not necessarily memory allocation that is a bottleneck on a process – it may very well be something like CPU or having multiple subprocesses depend on each other/use same variables, it would be useful to account for those possible problems when evaluating memory allocation so we can get a better picture of a system as a whole.

6. References

1. Qi Zhu, Haibo Zeng, Wei Zheng, Marco Di Natale, Sangiovanni-Vincentelli; Optimization of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems; ACM Trans; December 2012
2. Layland, Liu; Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment; Journal of the Association for Computing Machinery; January 1973
3. Alvaro Luiz Neuenfeldt Junior; The Two-Dimensional Rectangular Strip Packing Problem; Department of Industrial Engineering and Management; University of Porto; 2017

4. Bastian Ristau, Gerhard Fettweis; An Optimization Methodology for Memory Allocation and Task Scheduling in SoCS via Linear Programming; SAMOS 2006; LNCS 4017; pp. 89-98; Springer-Verlag Berlin Heidelberg; 2006