IE 202 – Discrete optimization

Term Project Report

Fall 2024

Resource Constrained Project Scheduling

Ezzaddeen M.S. Mofarreh

FACULTY OF ENGINEERING

DEPARTMENT OF INDUSTRIAL ENGINEERING

Content

Content 2
INTRODUCTION 3
Notation 3
Data [6] 5
1. CPM algorithm [2] [3] 6
Algorithm code 8
Verification of ES and LS calculations 9
2. Solution of RG30 Instances using CPLEX Model 11
Original data format 11
Network diagram 13
CPLEX solution 13
Gantt chart 15
Other instances solutions retrieval: 16
3. Solution for RG300 Instance using CPLEX Model 17
Network diagram 17
CPLEX solution 18
Solution retrieval 19
4. Alternative Solution Approach for RG300 Instances 19
Genetic Algorithms 20
Key concepts of genetic algorithms 20
Basic steps of a genetic algorithm 22
Pseudocode 25
Advantages 27
Disadvantages 28
References 29
Appendix 31
Appendix A: RCPSP Data Files 31
Appendix B: Notebook for Charts and Additional Code 32

INTRODUCTION

In project management, the effective allocation of resources and scheduling of tasks are pivotal in ensuring project success. However, the Resource-Constrained Project Scheduling Problem (RCPSP) presents a formidable challenge in this domain. At its essence, the RCPSP encapsulates the intricate balance between project objectives, resource limitations, and task dependencies. With origins rooted in operations research and combinatorial optimization, the RCPSP has garnered extensive attention from researchers and practitioners alike due to its practical relevance and theoretical complexity. The RCPSP conceals its true complexity, as demonstrated by Blazewicz et al. (1983), who categorized it as an NP-hard problem [4].

This problem arises in scenarios where a project must be executed within a limited timeframe and under constraints imposed by the availability of resources such as manpower, machinery, and materials. The primary goal is to devise an optimal schedule that maximizes resource utilization while adhering to project deadlines and task dependencies. However, the interplay between these factors introduces many challenges, ranging from computational complexity to the need for innovative scheduling algorithms and optimization techniques.

In essence, the RCPSP involves the scheduling of a single project comprising n real activities, subject to precedence and resource constraints, with the overarching objective of minimizing the project makespan — the total time required for the completion of all activities. Activities subject to precedence constraints means each activity must be completed before any subsequent activity can begin. The project has a set of renewable resources, and each activity requires a certain amount of these resources, which it consumes during its execution. Each resource has a maximum capacity, meaning it can handle multiple activities simultaneously, but the total resource consumption by these activities cannot exceed the resource's maximum capacity.

Notation

- **A**: Set of activities, A = {1, 2, 3, ⋯, n}. 0 and n+1 are start and end dummy activities
- **p_i**: Duration of each activity i.
- **R**: Set of resources, R = {1, 2, 3, ⋯, m}.
- **ES_i**: Earliest start time value for activity i.
- **LS_i**: Latest start time value for activity i.
- **H**: Planning horizon, H = {0, 1, 2, ⋯, T}, where T = LS_{n+1}.

- **x_{it}**: Binary variable indicating if activity i starts at time t (x_{it} = 1 if activity i starts at time t, and x_{it} = 0 otherwise).
- **Graph Representation**: We consider a graph G(V,E), where:
    - V represents the set of nodes, with each node corresponding to an activity.
    - E represents the set of arcs, with each arc corresponding to a precedence relationship between activities. Precedence relations in set E are pairs (i,j), where (i,j) ∈ E indicates that the execution of activity i must precede that of activity j.
    - Additional Nodes: The set V contains two additional nodes, 0 and n+1, to denote the start and end of the project, respectively.
    - The duration of the start node 0 and end node n+1 is both 0, denoted as $p_0 = p_{n+1} = 0$.

In this report, we present the algorithm we used to compute bounds for the start time of each activity i, specifically the earliest start time (ES_i) and the latest start time (LS_i). These computed values are then applied in the Mixed-Integer Linear Programming (MILP) formulation outlined below.

$$\min \sum_{t=ES_i}^{LS_i} t x_{n+1,t} \qquad (3)\ \text{Objective function}$$

$$\sum_{t=ES_j}^{LS_j} t x_{jt} \geq \sum_{t=ES_i}^{LS_i} t x_{it} + p_i \quad \forall (i,j) \in E \qquad (4)\ \text{Precedence constraints}$$

$$\sum_{i=1}^{n} b_{ik} \sum_{\tau=\max(ES_i, t-p_i+1)}^{\min(LS_i, t)} x_{i\tau} \leq B_k \quad \forall t \in H, \ \forall k \in R \qquad (5)\ \text{Resource constraints}$$

$$\sum_{t=ES_i}^{LS_i} x_{it} = 1 \quad \forall i \in A \cup \{n+1\} \qquad (6)\ \text{Activity start constraints}$$

$$x_{00} = 1 \qquad (7)$$

$$x_{it} = 0 \quad \forall i \in A \cup \{n+1\}, \ t \in H \setminus \{ES_i, LS_i\} \qquad (8)$$

$$x_{it} \in \{0,1\} \quad \forall i \in A \cup \{n+1\}, \ \forall t \in \{ES_i, LS_i\}. \qquad (9)$$

We applied the MILP formulations to 20 RCPS instances (301-320) from the RG30 dataset using the CPLEX solver. These instances encompass data for 30 activities, including their durations,

resource usage, and capacities. The solutions were obtained by running the CPLEX code on a local machine, and the **`solutions.py`** file can retrieve all solutions for all instances. In this report, we present the solution for one instance, along with its Gantt chart and our observations. Additionally, we applied the same model to solve an instance with 300 activities from the RG300 dataset.

Lastly, we propose an alternative solution approach for the 300-activity instances. We provide a pseudocode for this alternative method and discuss its potential advantages and disadvantages.

Data [6]

We have two datasets:

- **RG30** [9] consists of instances with 30 activities. For our analysis, we are specifically working with RG30 set 1, focusing on 20 instances numbered from 301 to 320. These datasets are commonly used for benchmarking algorithms and evaluating the performance of scheduling techniques on smaller-scale problems.

- **RG300** [8] comprises instances with 300 activities, offering more challenging scenarios. Although we are primarily focusing on RG30, we also examine one instance from the RG300 dataset. This allows us to explore more complex instances and test the scalability and efficiency of scheduling algorithms on larger-scale problems.

Below is a brief summary of the datasets [10].

Dataset	Subsets	# Instances	Reference	Generator	Parameters	OR&S
RG300	RG300	480	Debels and Vanhoucke (2007)	RanGen1	OS, RU, RC	Yes
RG30	Set 1, Set 2, Set 3, Set 4, Set 5	1,800	Vanhoucke et al. (2008)	RanGen 2	I2, I3, I4, I5, I6	Yes

1. CPM algorithm [2] [3]

The Critical Path Method (CPM) formula comprises two essential components: the forward pass and the backward pass. These two complementary processes are integral to identifying the critical path—a sequence of tasks that collectively determine the minimum duration required to complete a project.

The forward pass involves traversing the project network from the starting point to the endpoint. During this phase, each activity's earliest start time (ES) and earliest finish time (EF) are calculated based on the durations of preceding activities and any imposed constraints. The ES of an activity is determined by the maximum EF of its preceding activities, while its EF is computed by adding its duration to its ES. This process continues until the endpoint of the project network is reached, resulting in the determination of the earliest possible completion time for each activity.

Conversely, the backward pass entails traversing the project network in the reverse direction—from the endpoint to the starting point. Here, the latest finish time (LF) and latest start time (LS) of each activity are determined. The LF of an activity is initially set to the project's total duration, and the LS is subsequently computed based on the LF of succeeding activities and any constraints imposed by the project timeline. This process continues until the starting point of the project network is reached, resulting in the determination of the latest possible start time for each activity.

The following are the key CPM concepts and their formulas used in code for the calculation of ES_i and LS_i for each activity i.

Earliest Start (ES): This is simply the earliest time that a task can be started in the project.

For the initial activities (activities with no predecessors), the earliest start time is 0.

For other activities:

$$ESi = \max(EFj\ for\ all\ j\ in\ predecessors\ of\ i)$$

This means the ES of an activity is the maximum EF of its predecessor activities.

Earliest Finish (EF): The earliest an activity can be completed, based on its duration and its earliest start time.

The earliest finish time of an activity is the sum of its earliest start time and its duration:

$$EF_i = ES_i + \text{Duration}_i$$

Latest Finish (LF): The latest an activity can be completed, based on its duration and its latest start time.

For the final activities (activities with no successors), the latest finish time is the project horizon (usually the maximum EF value). $T = LS_{n+1}$ = maximum EF value

For other activities:

$$LF_i = \min(LS_{j \text{ for all } j} \text{ in successors of } i)$$

This means the LF of an activity is the minimum LS of its successor activities.

Latest Start (LS): This is the very last minute in which an activity can start before it threatens to delay the project timeline.

The latest start time of an activity is the difference between its latest finish time and its duration:

$$LS_i = LF_i - \text{Duration}_i$$

Algorithm code

```python
# CPM Algorithm
def forward_pass(nb_activities, duration, successors):
    """
    Performs the forward pass in the Critical Path Method (CPM) to calculate
    the earliest start (ES) and earliest finish (EF) times for each task.

    Parameters:
    nb_activities (int): The number of activities in the project.
    duration (list): A list of activity durations.
    successors (list): A list of lists containing successors for each activity.

    Returns:
    tuple: A tuple containing:
        - ES (list): Earliest start times for each activity.
        - EF (list): Earliest finish times for each activity.
    """
    ES = [0] * nb_activities
    EF = [0] * nb_activities

    for i in range(nb_activities):
        EF[i] = ES[i] + duration[i]
        for succ in successors[i]:
            ES[succ] = max(ES[succ], EF[i])
            EF[succ] = ES[succ] + duration[succ]

    return ES, EF
```

```python
def backward_pass(nb_activities, duration, successors, EF):
    """
    Performs the backward pass in the Critical Path Method (CPM) to calculate
    the latest start (LS) and latest finish (LF) times for each activity.

    Parameters:
    nb_activities (int): The number of activities in the project.
    duration (list): A list of activity durations.
    successors (list): A list of lists containing successors for each activity.
    EF (list): Earliest finish times for each activity.

    Returns:
    tuple: A tuple containing:
        - LS (list): Latest start times for each activity.
        - LF (list): Latest finish times for each activity.
    """
    horizon = max(EF)
    LS = [horizon] * nb_activities
    LF = [horizon] * nb_activities

    for i in reversed(range(nb_activities)):
        if not successors[i]:
            LS[i] = horizon - duration[i]
            LF[i] = horizon
        else:
            for succ in successors[i]:
                LS[i] = min(LS[i], LS[succ] - duration[i])
                LF[i] = LS[i] + duration[i]

    return LS, LF
```

Verification of ES and LS calculations

To verify the accuracy of our code in calculating the ES and LS values, we cross-checked it against an activity network from the book "Project Management with Dynamic Scheduling" **[3].**

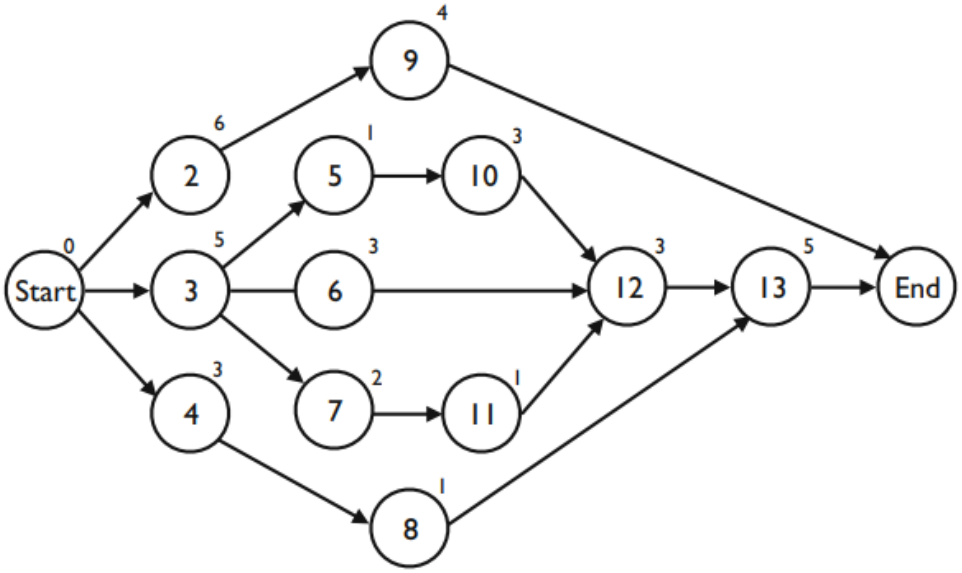


Fig. 2.12 The AoN example network of Table 2.2

Book's solution:

ES = [0,0,0,0,5,5,5,3,6,6,7,9,12,17]

LS = [0,7,0,8,5,6,6,11,13,6,8,9,12,17]

It is easy to verify that the earliest start times of the project activities of Table 2.2 are given by $es_1 = 0$, $es_2 = 0$, $es_3 = 0$, $es_4 = 0$, $es_5 = 5$, $es_6 = 5$, $es_7 = 5$, $es_8 = 3$, $es_9 = 6$, $es_{10} = 6$, $es_{11} = 7$, $es_{12} = 9$, $es_{13} = 12$ and $es_{14} = 17$. The overall minimal project duration equals 17 time units.

Given the project deadline of 17 time units, calculated as the earliest start of the end dummy activity in the previous step, the latest start times of each activity are given by $ls_1 = 0$, $ls_2 = 7$, $ls_3 = 0$, $ls_4 = 8$, $ls_5 = 5$, $ls_6 = 6$, $ls_7 = 6$, $ls_8 = 11$, $ls_9 = 13$, $ls_{10} = 6$, $ls_{11} = 8$, $ls_{12} = 9$, $ls_{13} = 12$ and $ls_{14} = 17$.

Algorithm's solution:

we got the same solution

```
# book's example - "Project Management with Dynamic Scheduling."
nb_activities = 14
duration = [0,6,5,3,1,3,2,1,4,3,1,3,5,0]
successors = [[2,3,4], [9], [5,6,7], [8], [10], [12], [11], [13], [14], [12], [12], [13], [14], []]

# mataching the index with the 0-index the algorithm work with
successors_0_index = [[1,2,3], [8], [4,5,6], [7], [9], [11], [10], [12], [13], [11], [11], [12], [13], []]
```

```
ES, EF = forward_pass(nb_activities, duration, successors_0_index)
LS, LF = backward_pass(nb_activities, duration, successors_0_index, EF)
print_activity_times(ES, EF, LS, LF, duration)
```

Activity	Duration	ES	EF	LS	LF
0	0	0	0	0	0
1	6	0	6	7	13
2	5	0	5	0	5
3	3	0	3	8	11
4	1	5	6	5	6
5	3	5	8	6	9
6	2	5	7	6	8
7	1	3	4	11	12
8	4	6	10	13	17
9	3	6	9	6	9
10	1	7	8	8	9
11	3	9	12	9	12
12	5	12	17	12	17
13	0	17	17	17	17

2. Solution of RG30 Instances using CPLEX Model

Let’s look closer at one of the RG30 instances we are working with, specifically the pat301.rcp instance. We will first look at the original data, followed by an Activity-on-Node (AoN) diagram. Finally, we will present the CPLEX model solution and its corresponding Gantt chart.

Original data format

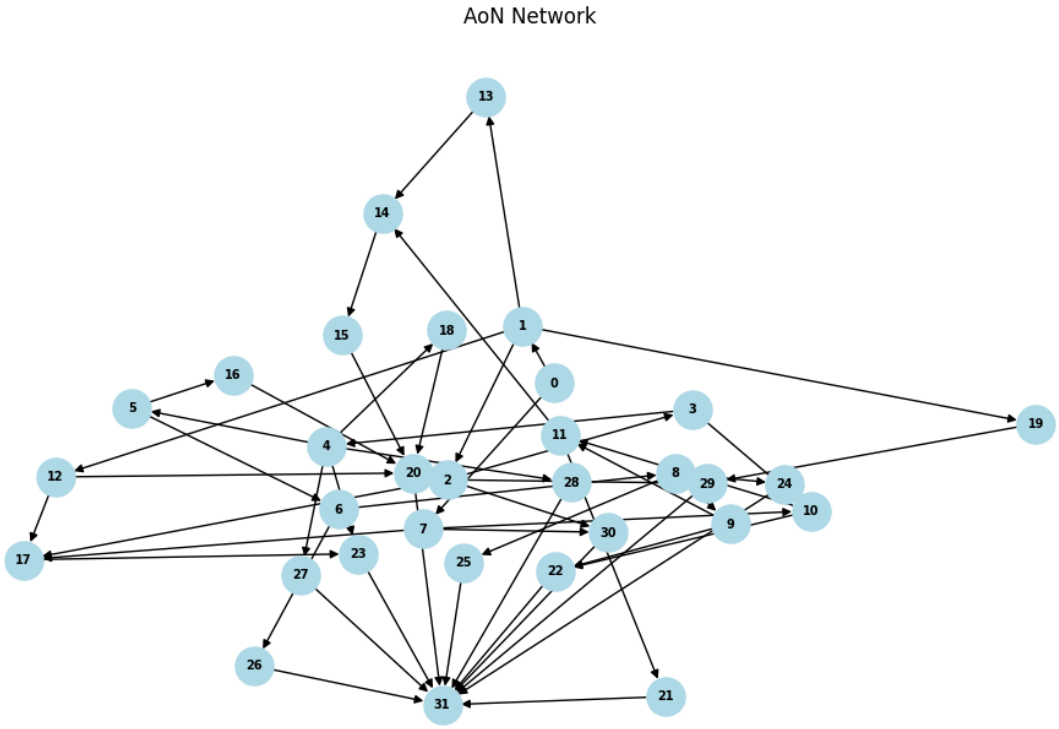
The original data is shown below in the Patterson format. For detailed instructions on how to interpret it, please consult Appendix A.

Pat301.rcp

```
 32    4
 10   10   10   10

  0   0   0   0   0   2 2 8
  1   1   1   0   1   4 20 14 13 3
  9   5   5   0   4   4 31 30 18 4
  7   7   3   0   4   2 11 5
  3   4   0   1   2   5 29 28 24 19 6
  8   0   2   4   4   2 17 7
  7   4   0   1   4   2 27 9
  4   6   6   0   8   3 31 18 11
 10   7   3   3   0   3 26 25 10
  2   1   2   0   5   2 23 12
  5   2   5   0   3   2 23 12
  4   5   0   4   1   2 22 15
  1   0   4   6   4   2 21 18
  8   0   2   4   5   1 15
  4   3   4   0   4   1 16
  1   2   0   3   6   1 21
  3   8   4   0   5   1 21
  7   2   7   0   3   1 24
  4   4   0   4   2   1 21
 10   0   4   6   5   1 30
  8   1   2   5   0   1 32
  6   4   0   8   4   1 32
 10   3   3   0   6   1 32
  2   0   3   5   4   1 32
  4   0   9   1   5   1 32
  2   5   3   3   0   1 32
  1   4   7   7   0   1 32
 10   7   0   2   3   1 32
  5   0   3   4   4   1 32
  2   3   5   3   0   1 32
  9   0   5   6   4   1 32
  0   0   0   0   0   0
```

Network diagram

The network graph for this instance is illustrated as an Activity-on-Node (AoN) diagram. Nodes represent activities, while arcs denote the precedence relationships between them.



CPLEX solution

The solution obtained from running the CPLEX model is as follows:

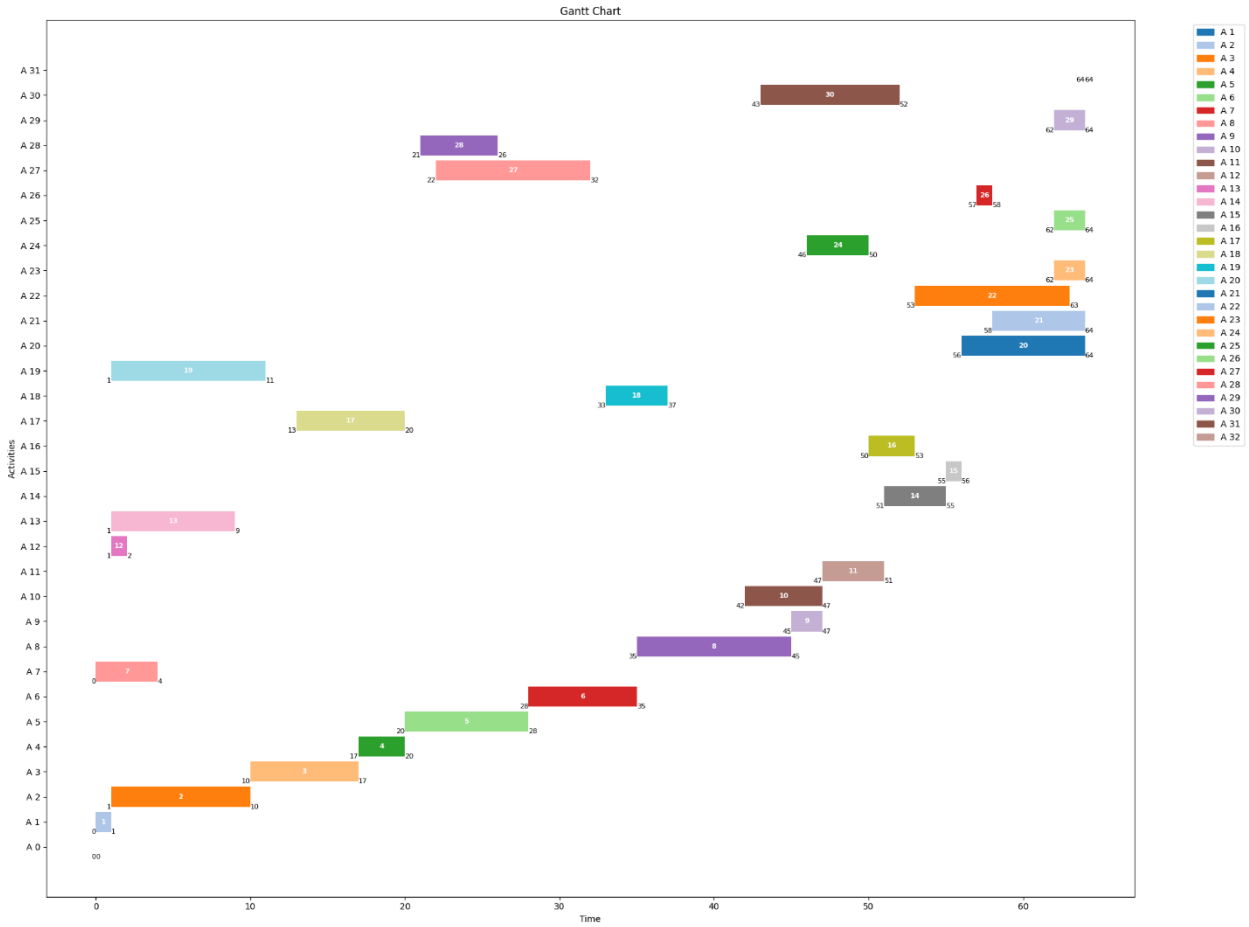
```
Model: RCPSP
 - number of variables: 2080
   - binary=2080, integer=0, continuous=0
 - number of constraints: 345
   - linear=345
 - parameters: defaults
 - objective: minimize
 - problem type is: MILP

Objective Function:
64x_32_64
Version identifier: 22.1.1.0 | 2022-11-27 | 9160aff4d
CPXPARAM_Read_DataCheck                          1
Tried aggregator 1 time.
MIP Presolve eliminated 242 rows and 1948 columns.
MIP Presolve modified 2560 coefficients.
Reduced MIP has 103 rows, 132 columns, and 1096 nonzeros.
Reduced MIP has 132 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.14 sec. (9.43 ticks)
Found incumbent of value 64.000000 after 0.17 sec. (11.73 ticks)

Root node processing (before b&c):
  Real time             =    0.19 sec. (11.80 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.19 sec. (11.80 ticks)
Solution found
```

```
activity 0 starts at time 0
activity 1 starts at time 0
activity 2 starts at time 1
activity 3 starts at time 10
activity 4 starts at time 17
activity 5 starts at time 20
activity 6 starts at time 28
activity 7 starts at time 0
activity 8 starts at time 35
activity 9 starts at time 45
activity 10 starts at time 42
activity 11 starts at time 47
activity 12 starts at time 1
activity 13 starts at time 1
activity 14 starts at time 51
activity 15 starts at time 55
activity 16 starts at time 50
activity 17 starts at time 13
activity 18 starts at time 33
activity 19 starts at time 1
activity 20 starts at time 56
activity 21 starts at time 58
activity 22 starts at time 53
activity 23 starts at time 62
activity 24 starts at time 46
activity 25 starts at time 62
activity 26 starts at time 57
activity 27 starts at time 22
activity 28 starts at time 21
activity 29 starts at time 62
activity 30 starts at time 43
activity 31 starts at time 64
Objective value = 64.0
```

The model solution reveals the optimal value attained, which represents the minimized completion time for all activities. In this case, **the objective value** is 64. It also provides insights into the computational efficiency, detailing the time taken to solve the problem—specifically, **the elapsed time** of 0.19 seconds for the total root and branch-and-cut processing. Each activity is assigned specific start and end times, meticulously determined to optimize project scheduling and resource utilization. The model's performance metrics include the number of variables and constraints involved, showcasing its complexity and the approach used to reach the solution.

Gantt chart

Gantt chart illustrating the start and end times of each activity in the solution.



Other instances solutions retrieval:

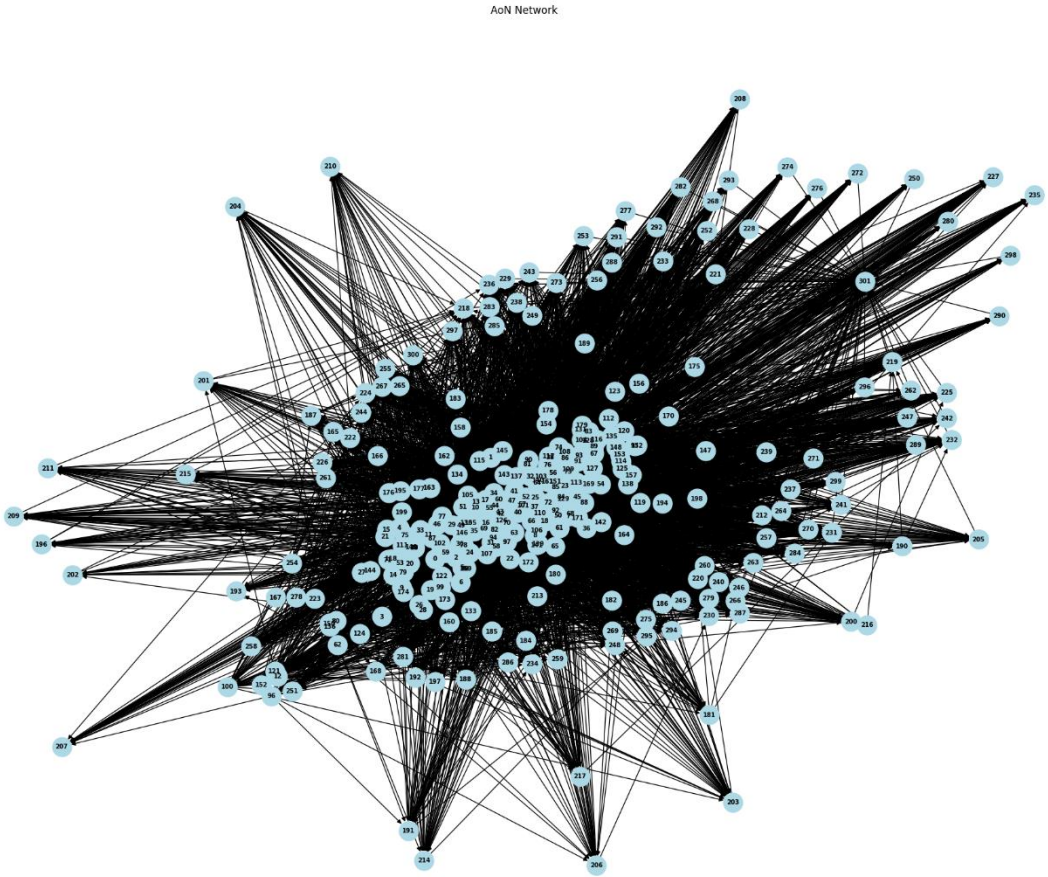
For solutions to other instances, please refer to the **`solutions`** folder. The solutions for the 20 instances are recorded in the file **`RG30_solutions.txt`**.

To obtain all solutions, run the **`solutions.py`** script.

3. Solution for RG300 Instance using CPLEX Model

We attempt to solve an instance from the RG300 dataset, specifically the RG300_1.rcp instance, which contains 300 activities. Using the same MILP model employed for the 30-activity instances, we run the CPLEX solver to tackle this larger and more complex problem. The original data is formatted similarly to the RG30 dataset using the Patterson format. Due to the extensive amount of data, it is impractical to display it fully here. However, we will present a graph to illustrate the complexity of the network. Despite the challenges in visualizing it clearly, this graph highlights the intricate nature of managing a project with 300 activities.

Network diagram



CPLEX solution

```
Model: RCPSP
 - number of variables: 13590
   - binary=13590, integer=0, continuous=0
 - number of constraints: 5690
   - linear=5690
 - parameters: defaults
 - objective: minimize
 - problem type is: MILP

Objective Function:
44x_302_44
Version identifier: 22.1.1.0 | 2022-11-27 | 9160aff4d
CPXPARAM_Read_DataCheck                          1
Tried aggregator 2 times.
MIP Presolve eliminated 2124 rows and 11000 columns.
MIP Presolve modified 142916 coefficients.
Aggregator did 11 substitutions.
Reduced MIP has 3318 rows, 2579 columns, and 61234 nonzeros.
Reduced MIP has 2579 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.70 sec. (273.08 ticks)
Probing fixed 1 vars, tightened 0 bounds.
Probing time = 0.11 sec. (7.82 ticks)
Cover probing fixed 0 vars, tightened 331 bounds.
Tried aggregator 2 times.
Detecting symmetries...
MIP Presolve eliminated 1202 rows and 235 columns.
MIP Presolve modified 18780 coefficients.
Aggregator did 1 substitutions.
Reduced MIP has 2068 rows, 2343 columns, and 40837 nonzeros.
Reduced MIP has 2343 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.47 sec. (113.52 ticks)
Probing time = 0.03 sec. (9.13 ticks)
Cover probing fixed 0 vars, tightened 10 bounds.
Clique table members: 18132.
Tightened 10 constraints.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 4 threads.
Root relaxation solution time = 0.28 sec. (88.37 ticks)
```

```
        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

      0     0       44.0000   354                   44.0000      657
      0     0       44.0000   172                 Cuts: 140      889
      0     0       44.0000   252                 Cuts: 337     1319
      0     0       44.0000    93                  Cuts: 21     1401
      0     0       44.0000   195                 Cuts: 211     1834
*     0+    0                           44.0000       44.0000            0.00%
      0     0        cutoff             44.0000       44.0000     1834   0.00%
Elapsed time = 6.39 sec. (1762.88 ticks, tree = 0.01 MB, solutions = 1)

GUB cover cuts applied:  69
Clique cuts applied:  60
Cover cuts applied:  3
Flow cuts applied:  22
Mixed integer rounding cuts applied:  22
Zero-half cuts applied:  34
Gomory fractional cuts applied:  4

Root node processing (before b&c):
  Real time             =    6.48 sec. (1763.40 ticks)
Parallel b&c, 4 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    6.48 sec. (1763.40 ticks)
Solution found
```

We can observe here that **the elapsed time** is 6.39 seconds, which is significantly higher compared to the time it took to solve an RG30 instance. This difference reflects the increased complexity of solving a larger RG300 instance with 300 activities.

Solution retrieval

For the solution to an instance from the RG300 dataset, please refer to the **`solutions`** folder. The solution for an instance is recorded in the file **`RG300_solutions.txt`**.

To obtain the solution, run the **`solutions.py`** script.

4. Alternative Solution Approach for RG300 Instances

To address the complexity of solving a 300-activity instance using the MILP model, we propose an alternative solution approach tailored to high-dimensional scheduling problems. This approach seeks to provide a more efficient method while acknowledging the computational limitations inherent in standard MILP formulations.

We introduce a well-regarded algorithm from the literature – the Genetic Algorithm (GA) – as an alternative approach, elucidate its key concepts and processes, and present pseudocode for its implementation in Resource-Constrained Project Scheduling (RCPS) problems. Furthermore, we discuss the advantages and disadvantages of this approach, offering a comprehensive perspective on its applicability and performance.

Genetic Algorithms

Genetic Algorithms (GAs) are search algorithms based on the mechanics of natural selection and genetics [12]. The concept of Genetic Algorithms was first developed by John Holland, his colleagues, and his students at the University of Michigan. John Holland's pioneering work [13] laid the groundwork for the development and popularization of GAs, influencing many subsequent researchers and practitioners in the field.

GAs are a popular type of evolutionary algorithm inspired by the process of natural selection. They are widely used for solving complex optimization and search problems, including the Resource-Constrained Project Scheduling Problem (RCPSP). Over the years, numerous variations of GAs have been developed to enhance their efficiency and effectiveness. GAs can be hybridized with methods like Local Search, Simulated Annealing, or Feasible Backward Improvement to enhance performance. Some common variations include GA with Local Search (GA(LS)), GA with Feasible Backward Improvement (GA(FBI)), and GA with Simulated Annealing (GA(SA(FBI))) [11].

Key concepts of genetic algorithms

Population: a set of potential solutions to the problem. Each individual in the population represents a candidate solution.

PRCPS Example:

The population is a collection of individual schedules for the project activities. Each individual in the population represents a different way to schedule the activities, taking into account resource constraints and dependencies.

Chromosome: The representation of an individual solution. It is typically encoded as a string of bits, numbers, or characters.

PRCPS Example:

A chromosome represents a single schedule within the population. It encodes the start times of all activities in the project. For a project with 5 activities, a chromosome might look like [3, 1, 4, 2, 5], where each number represents the start time of a corresponding activity.

Genes: the elements of a chromosome, representing the solution's variables.

PRCPS Example:

The individual elements of a chromosome represent the start time of a single activity. In the chromosome [3, 1, 4, 2, 5], the genes are 3, 1, 4, 2, and 5, each representing the start time of activities 1 through 5, respectively.

Fitness Function: a function that evaluates and assigns a fitness score to each individual based on how well it solves the problem.

PRCPS Example:

The fitness function evaluates how good each schedule (chromosome) is at minimizing the project's total duration while respecting resource constraints. It might calculate the total project duration and add penalties for any resource constraint violations. A lower fitness score would correspond to a better schedule.

Selection: the process of choosing individuals from the population to create offspring for the next generation. Fitter individuals have a higher chance of being selected.

PRCPS Example:

Selection involves choosing the best schedules to act as parents for creating the next generation. For example, using tournament selection, the schedules with the shortest project durations (and fewest constraint violations) are more likely to be chosen as parents.

Crossover (Recombination): a genetic operator used to combine the genetic information of two parents to generate new offspring. This mimics biological reproduction.

PRCPS Example:

Crossover combines two parent schedules to create a new schedule. If two parent schedules are [3, 1, 4, 2, 5] and [2, 3, 1, 5, 4], a single-point crossover might create offspring like [3, 1, 4, 5, 4] and [2, 3, 1, 2, 5].

Mutation: a genetic operator that introduces random changes to an individual's genes, ensuring genetic diversity within the population and helping to explore new solutions.

PRCPS Example:

Mutation introduces random changes to some schedules to maintain diversity. In a schedule [3, 1, 4, 2, 5], a mutation might change the start time of one task, resulting in [3, 2, 4, 2, 5].

Generation: one complete cycle of selection, crossover, and mutation processes resulting in a new population.

PRCPS Example:

Starting with an initial population, after applying selection, crossover, and mutation, you get a new population. This process represents one generation.

Basic steps of a genetic algorithm

Initialization: begin with a randomly generated population of schedules for the project activities.

RCPS Example: generate 10 random schedules for the RG30 dataset, each representing a different sequence of activity start times.

Evaluation: calculate the fitness of each schedule using the fitness function. Determine how well each schedule minimizes project duration and respects resource constraints.

RCPS Example: calculate the fitness of each schedule by computing the total project duration and adding penalties for any resource constraint violations.

Selection: choose schedules to be parents based on their fitness scores. Fitter schedules have a higher chance of being selected.

RCPS Example: use tournament selection to pick the top 5 schedules based on their fitness scores.

Crossover: combine pairs of parent schedules to produce new offspring schedules.

RCPS Example: pair up the selected schedules and perform crossover to generate 5 new schedules.

Mutation: introduce random changes to some schedules to maintain diversity.

RCPS Example: randomly adjust the start times of some activities in the new schedules to introduce variation.

Replacement: form a new population by replacing some or all of the old population with the new offspring.

RCPS Example: form a new population with the 5 old and 5 new schedules.

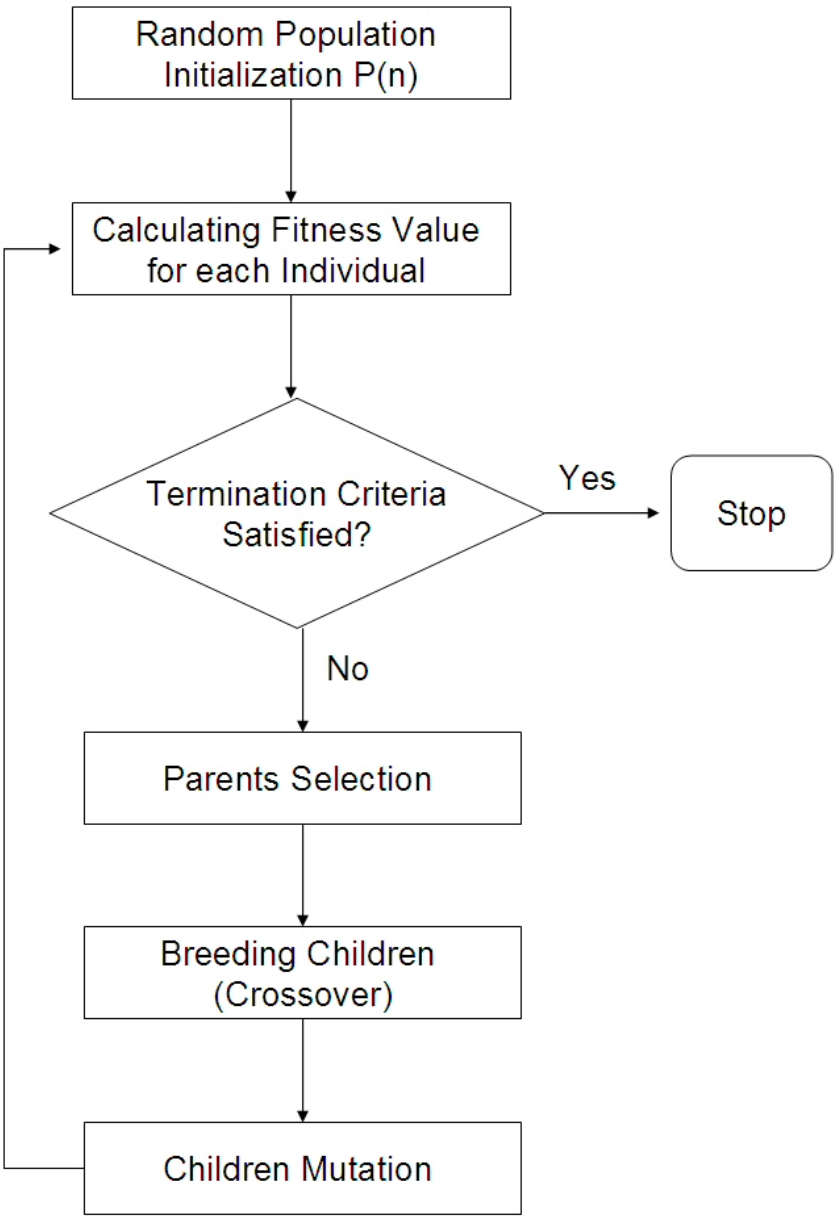
Termination: repeat the process until a stopping criterion is met (e.g., a schedule of acceptable fitness is found, or a maximum number of generations is reached).

RCPS Example: continue the process for 100 generations or until a schedule with a project duration below a certain threshold is found.

By following this process, the GA evolves a population of schedules with progressively shorter project durations, aiming to find the optimal or near-optimal schedule.

Flowchart [14]

The flowchart below provides a high-level visual representation to help understand how the general algorithm works. It outlines the sequence of steps involved in the genetic algorithm, making it easier to grasp the overall process and the flow of operations from initialization to termination.


Random Population Initialization P(n)
Calculating Fitness Value for each Individual
Termination Criteria Satisfied?
Yes
Stop
No
Parents Selection
Breeding Children (Crossover)
Children Mutation

Pseudocode

```
# Pseudocode for Genetic Algorithm to solve RCPSP

# Helper Functions

Function GenerateRandomSchedule(num_activities):
    - Initialize schedule with random start times for num_activities
    - Return schedule

Function EvaluateFitness(schedule, resource_constraints):
    - Calculate total project duration
    - Calculate penalties for resource constraint violations
    - fitness = project duration + penalties
    - Return fitness

Function TournamentSelection(population):
    - Select k individuals randomly from population
    - Return the two individuals with the best fitness

Function SinglePointCrossover(parent1, parent2):
    - Select a crossover point randomly
    - child1 = combine part of parent1 and part of parent2
    - child2 = combine part of parent2 and part of parent1
    - Return child1, child2

Function Mutate(schedule):
    - Select an activity randomly
    - Change its start time randomly
    - Return modified schedule

Function GetBestSchedule(population):
    - Find the schedule with the best (lowest) fitness value
    - Return best schedule
```

```
1. Initialize Parameters
    - population_size: Number of schedules in the population
    - num_generations: Number of generations to run the algorithm
    - mutation_rate: Probability of mutating a schedule
    - crossover_rate: Probability of crossover between schedules
    - num_activities: Number of activities in the project
    - resource_constraints: Constraints on the resources

2. Initialize Population
    - population = []  # List to store the population
    - For i in range(population_size):
        - schedule = GenerateRandomSchedule(num_activities)
        - population.append(schedule)

3. Evaluate Fitness
    - For each schedule in population:
        - fitness = EvaluateFitness(schedule, resource_constraints)
        - store fitness value
```

```
4. Repeat for num_generations generations:
    4.1. Selection
        - selected_parents = []  # List to store selected parents
        - While len(selected_parents) < population_size:
            - parent1, parent2 = TournamentSelection(population)
            - selected_parents.append(parent1)
            - selected_parents.append(parent2)

    4.2. Crossover
        - new_population = []  # List to store new population
        - For i in range(0, population_size, 2):
            - parent1 = selected_parents[i]
            - parent2 = selected_parents[i + 1]
            - If random() < crossover_rate:
                - child1, child2 = SinglePointCrossover(parent1, parent2)
            - Else:
                - child1, child2 = parent1, parent2
            - new_population.append(child1)
            - new_population.append(child2)

    4.3. Mutation
        - For each schedule in new_population:
            - If random() < mutation_rate:
                - Mutate(schedule)

    4.4. Evaluate Fitness
        - For each schedule in new_population:
            - fitness = EvaluateFitness(schedule, resource_constraints)
            - store fitness value

    4.5. Replacement
        - population = new_population

5. Termination
    - Best_schedule = GetBestSchedule(population)
    - Return Best_schedule
```

Advantages

- **Hybridization**: GAs can be hybridized with other optimization techniques like Local Search, Simulated Annealing, or Feasible Backward Improvement to enhance performance [11].
- **Parallelism**: GAs are naturally suited for parallel processing, which can significantly speed up computations [12].
- **Global Search Capability**: GAs excel at exploring the global search space, reducing the risk of getting trapped in local optima [12].

- **Robustness**: they are robust and can handle a variety of problem types, including non-differentiable, non-continuous, and multi-modal functions [13].
- **Versatility**: GAs can accommodate various types of objective functions and constraints, enhancing their applicability [13].
- **Adaptable**: they can easily adapt to different problem structures and constraints, making them suitable for a wide range of applications [13].

Disadvantages

- **Computational cost**: genetic Algorithms can be computationally expensive due to the need to evaluate a large number of candidate solutions across multiple generations [12].
- **Parameter sensitivity**: the performance of GAs is highly sensitive to the choice of parameters (e.g., population size, mutation rate, crossover rate). Careful tuning and experimentation are often required to find the optimal settings [15].
- **Convergence speed**: GAs may take longer to converge to an optimal or near-optimal solution compared to some other heuristics or exact methods. This can be an issue in time-sensitive applications [15].

AI Tools Assistance Acknowledgment

We would like to acknowledge the use of AI tools such as Gemini and ChatGPT for assistance in writing and coding.

References

1. Hexaly. "Resource Constrained Project Scheduling Problem (RCPSP)." Hexaly Documentation. Accessed June 6, 2024. (https://www.hexaly.com/docs/last/exampletour/resource-constrained-project-scheduling-problem-rcpsp.html)
2. ProjectManager. "Critical Path Method (CPM) in Project Management." ProjectManager.com. Accessed June 6, 2024. (https://www.projectmanager.com/guides/critical-path-method)
3. Vanhoucke, Mario. " Project Management with Dynamic Scheduling: Baseline Scheduling, Risk Analysis, and Project Control." 2nd ed., Springer, 2013. (https://www.google.com.tr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwis8Oy5pcqGAxWIVPEDHTkOLSAQFnoECDAQAQ&url=http%3A%2F%2Flibrary.wbi.ac.id%2Frepository%2F217.pdf&usg=AOvVaw2Sk62FtoPIu8vR4TsVT-ie&opi=89978449)
4. J. Blazewicz, J.K. Lenstra, A.H.G.Rinnooy Kan. " Scheduling subject to resource constraints: classification and complexity." Discrete Applied Mathematics, Volume 5, Issue 1, 1983. (https://www.sciencedirect.com/science/article/pii/0166218X83900124)
5. P2 Engine. "Data files: The Patterson format." P2 Engine. Accessed 8 June 2024. (https://www.p2engine.com/p2reader/patterson_format)
6. Vanhoucke, M., Coelho, J. and Batselier, J. "An overview of project data for integrated project management and control", Journal of Modern Project Management, 3(2), 6–21, 2016. (https://www.or-as.be/sites/default/files/files/blog_files/Vanhoucke et al%2C JMPM%2C 2016.pdf)
7. Random Network Generation | Operations Research & Scheduling Research Group (ugent.be)
8. Debels, Dieter and Vanhoucke, Mario. "A Decomposition-Based Genetic Algorithm for the Resource-Constrained Project-Scheduling Problem." 2007. ([PDF] A Decomposition-Based Genetic Algorithm for the Resource-Constrained Project-Scheduling Problem | Semantic Scholar)
9. Mario Vanhoucke, José Coelho, Dieter Debels, Broos Maenhout, Luís V. Tavares. "An evaluation of the adequacy of project network generators with systematically sampled networks." European Journal of Operational Research, Volume 187, Issue 2, 2008. (An evaluation of the adequacy of project network generators with systematically sampled networks - ScienceDirect)

10. Operations Research & Scheduling Research Group. "Random Network Generation Artificial project data". Accessed June 12, 2024. (https://www.projectmanagement.ugent.be/research/data/RanGen)
11. Robert Pellerin, Nathalie Perrier, François Berthaut, "A survey of hybrid metaheuristics for the resource-constrained project scheduling problem", European Journal of Operational Research, Volume 280, Issue 2, 2020, Pages 395-416, ISSN 0377-2217.(https://www.sciencedirect.com/science/article/pii/S0377221719300980)
12. Goldberg, D. E. "Genetic Algorithms in Search, Optimization, and Machine Learning". Addison-Wesley Publishing Company, Inc, 1989. (http://www2.fiit.stuba.sk/~kvasnicka/Free books/Goldberg_Genetic_Algorithms_in_Search.pdf)
13. Holland, J. H. "Adaptation in Natural and Artificial Systems". University of Michigan Press, 1975. (Second edition published in 1992 by MIT Press). (https://ieeexplore.ieee.org/book/6267401)
14. ResearchGate. Flowchart figure. Accessed June 14, 2024. Retrieved from https://www.researchgate.net/figure/Flowchart-of-evolutionary-algorithms_fig1_251822092
15. Mitchell, M. "An Introduction to Genetic Algorithms". Cambridge, MA: MIT Press, 1998. (https://direct.mit.edu/books/book/4675/An-Introduction-to-Genetic-Algorithms)

Appendix

Appendix A: RCPSP Data Files

Source of the Datasets

The RG30 and RG300 datasets were obtained from [Project Data | Operations Research & Scheduling Research Group (ugent.be)]. These datasets are commonly used for research and benchmarking in resource-constrained project scheduling.

The Patterson Format

The Patterson format is used to structure the RCPSP data files. Each file contains detailed information on the activities, their durations, resource usage, and capacities. The format is designed to facilitate easy parsing and application in project scheduling algorithms.

Below is an example of how the Patterson format looks and is read.

RCPSP Data files: The Patterson format

The datasets with instances for the **resource constrained project scheduling problem** (RCPSP) make use of the well-known Patterson format to represent an activity-on-the-node network with renewable resource use. The format is a simple text file and its structure is explained on the illustrative project network of figure 1. Each node above the node is assumed to be the activity duration.

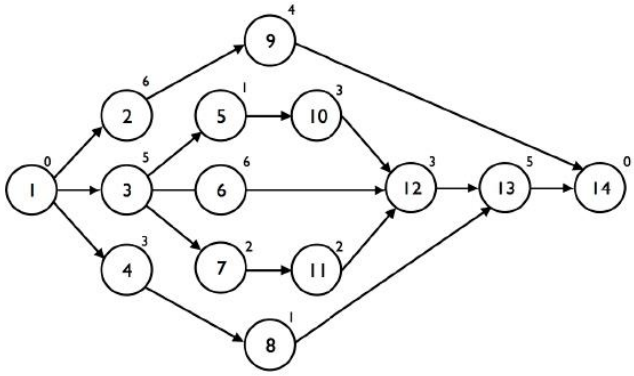


Figure 1. An illustrative activity-on-the-node network
(Source: Figure 7.1 and Table 7.2 of the book "Project Management with Dynamic Scheduling: Baseline Scheduling, Risk Analysis and Project Control")

The network of figure 1 has two dummy activities, i.e. dummy start node 1 and dummy end node 14, and hence, the network contains 14 activities in total, dummies inclusive. The Patterson format also makes use of start and end dummy nodes and is structured as follows:

Line 1:
- Number of activities (starting with node 1 and two dummy nodes inclusive)
- Number of renewable resources

Line 2: (one number for each resource)
- Availability for each renewable resource

Next lines from (one line for each activity, starting with a dummy start activity and ending with a dummy end activity)
- Activity duration
- Resource requirements for each resource type
- Number of successors
- Activity ID for each successor

It is assumed that the project network of figure 1 needs four renewable resource types. Consequently, the Patterson text file for the network of the figure is as follows:

```
14    4
10    20    8     10
0     0     0     0     0     1     2     3     4
6     7     15    2     6     1     9
5     1     8     4     8     3     5     6     7
3     5     8     3     3     1     8
1     6     15    2     6     1     10
3     1     13    0     3     1     12
2     2     16    2     0     1     11
1     2     9     4     4     1     13
4     8     12    5     5     1     14
3     6     17    5     0     1     12
1     2     10    2     5     1     12
3     6     5     5     4     1     13
5     8     10    3     7     1     14
0     0     0     0     0     0
```

As an example, activity 2 of figure 1 needs 7, 15, 2 and 6 units of resource 1, 2, 3 and 4, respectively. The availability of these resources is set at maximum 10, 20, 8 and 10 units.

Appendix B: Notebook for Charts and Additional Code

The detailed implementation, including Gantt charts, network graphs, and additional code related to this project, can be found in the Colab notebook available below.

Colab notebook:
https://colab.research.google.com/drive/1rMv37FqsbuDl0wVnFyS3hfTPO1sUGbF-?usp=sharing