Advanced Programming for Scientific Computing

Implementation of branch and bound algorithm for stochastic scheduling problems —using bob++ library

Student: Lei Liu, 10769500
Supervisor: Prof. Marcello Urgo

Politecnico di Milano
Academic Year 2019-2020

Abstract

This project aims at the development of a branch and bound algorithm to solve stochastic scheduling problems in the remanufacturing process of turbine blades with the environment of the bob++ library developed by LeCun et.al [2]. In particular, we develop an efficient algorithm to solve the two-machine flow shop scheduling problems with exponential distributed processing times, minimizing the value-at-risk of makespan. The test instances are proposed and statistical results of the problem and algorithm are collected. Additionally, several C++ libraries are introduced and implemented, related scientific computing topics are presented.

Contents

1. Manufacturing System

In recent years, increasing attention has been devoted towards enhancing the sustainability of manufacturing processes by reducing the consumption of resources and key materials, energy consumption and environmental footprint, while also reducing costs and increasing competitiveness in the global market. Re-manufacturing, which can be defined as "the rebuilding of a product to specifications of the original manufactured product using a combination of reused, repaired and new parts", is a form of product recovery process entailing the repair or replacement of worn out components to obtain re-manufactured products with the same characteristics as new products. The re-manufacturing paradigm is aimed at supporting sustainability challenges in strategic manufacturing sectors for high-value products whose residual value is high, such as aeronautics, automotive, electronics, consumer goods, and mechatronics.

Re-manufacturing processes, compared to the original manufacturing processes, entail higher degrees of uncertainty, complexity and dynamics, due to the unpredictable and variable conditions of the used parts to be processed. This significantly affects process and production planning, as well as the requirements driving the design of systems operating re-manufacturing activities. Thus, low production efficiency, unstable product quality, frequent abnormal production accidents and high product rework rates are typical characteristics of re-manufacturing systems. To address these operating scenarios, smart approaches are needed to match production management and control decisions with the updated state of production processes, resources and requirements.

Industry 4.0, which is one of the most trending topics in manufacturing area, considers smart manufacturing as its central element, and relies on the adoption of digital technologies such as Internet of Things (IoT), cloud services, big data and analytics, to gather data in real time and to analyze it, providing useful information to the manufacturing system. To hedge against unexpected events in the manufacturing system, predictive and reactive approaches can take advantage of advanced sensor technologies providing monitoring capability and artificial intelligence supporting analyses and decisions. Taking advantage of the described scenario, smart management approaches for re-manufacturing systems are required to support management, planning and scheduling, within the circular economy paradigm.

Gas turbine (Figure 1.1), in which burning of an air-fuel mixture produces hot gases that spin a turbine to produce power, is one of the most widely-used power generating technologies. Turbine blades (Figure 1.2), whose individual price is close to a middle-class car (i.e. about ten-thousand euros), is one of the most important and expensive parts in

a gas turbine. To maximize the performance of a gas turbine, they are constituted by multiple stages, each of them equipped with specially-designed turbine blades. According to the specific OEM (Original Equipment Manufacturer), the number of stages can vary, as an example, a F-class turbine [3] needs about 400 blades for 3 to 6 stages. Blades belonging to the same stage of the turbine are usually manufactured/re-manufactured in batch to guarantee homogeneous characteristics and the balancing of the portion of the rotor for the stage.

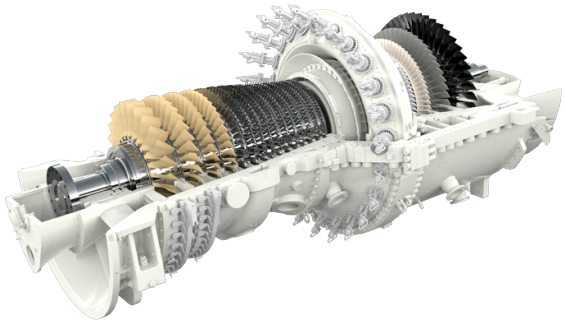

Figure 1.1: AE 94.3A gas turbine manufactured in Ansaldo Energia [4]

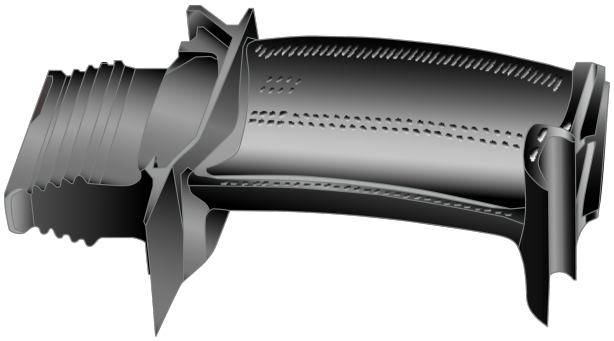

Figure 1.2: New turbine blade [10]

During the functioning of the turbine, blades are exposed to an extremely strenuous environment (high temperatures, stresses, vibration and corrosion) constituting a major cause for wear and failures of the blades (Figure 1.3) and, consequently, reduced energy conversion efficiency, and potentially disruptive failures of the whole turbine.

Figure 1.3: Damaged blades resulting from exposure to high temperature and stress [10]

Thus, planned maintenance operations are enforced for gas turbines, requiring all the blades in the rotor to be disassembled, inspected and, if needed, re-manufactured or replaced with new ones. Taking the economy cost into consideration, extending the service life of blades through re-manufacturing is a feasible and attractive choice, since their replacement with a new one is far more expensive.

The re-manufacturing of turbine blades are processed in batches, consisting of a set of blades belonging to the same stage of the turbine, it involves multiple processes and technologies, e.g., visual inspection, non-destructive testing, machining, additive manufacturing, grinding, heat-treatments, coating, etc. The whole process can be summarized in the following main steps:

1. Blades are disassembled at the production site and shipped to the OEM to be refurbished/renewed.

2. Upon arrival at the OEM, blades are inspected to assess the severity of wear and damages. Blades that cannot be repaired are discarded and the supply/production of new blades is issued. On the contrary, for reparable blades, the re-manufacturing process starts.

3. The re-manufacturing process consists of removing the damaged (e.g., cracked) parts of the blades, restore the removed material through welding/additive technologies, machining/grinding the blade to reconstruct the desired shape. During the material removal phase, as well as at the end of the processing, non-destructive tests are operated to verify the complete removal of the defects.

4. The blades undergo the definition of small-size features (e.g., internal cooling ducts, specific shapes in the terminal side or in the coupling with the rotor) and are successively coated with high-resistance materials.

5. Finally, the blades belonging to the same stage are assembled together and balanced. Hence they are shipped back to the operating site of the turbine where they will be re-assembled and be ready to go into production again.

Within the described process, two of the most relevant re-manufacturing activities are the defects removal process and the following welding process adding materials. Blades are repaired in batches, A set of jobs, representing batches of blades, must be processed on two machines, i.e., defect removal and laser welding in sequence. The sequence of the jobs on the two machines is the same (See Figure 1.4).

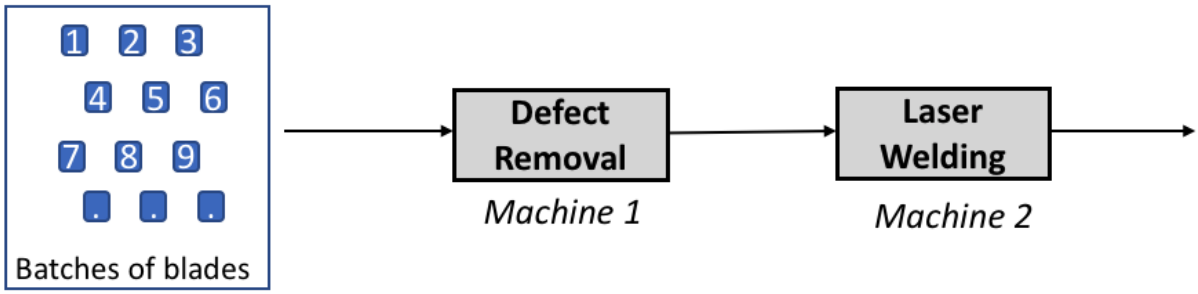

Figure 1.4: Re-manufacturing process

One of the peculiar requirements of re-manufacturing processes is the capability to cope with the intrinsic uncertainty affecting the parts and products to process. This drives the need for re-manufacturing plants able to operate in variable conditions with high degree of efficiency awhile guaranteeing the quality of the processed products. The main sources of uncertainty to manage can be summarized in the following classes:

1. A major source of uncertainty is embedded in the characteristics to be processed. Used parts arrives from different use-phase conditions and, thus, could be worn differently. The blades with a higher degree of damages will likely entail longer processing times and different process parameters respect to the less severely damaged blades. The severity of the damages can be partially estimated in the initial inspection phase but, while the re-manufacturing process is operated, deviations from what initially estimated could emerge. These deviations can just affect the processing time of specific operation.

2. Moreover, grounding on their characteristics, the number of parts whose re-manufacturing is feasible or attractive could be different, thus impacting on the actual volumes of parts to be re-manufactured. Thus, the number of blades to be processed as well as their processing times are not known in advance.

The sources of uncertainty described above primarily affects the planning and scheduling of re-manufacturing operations, responsible of matching the requirements of the customers (delivery dates) as well as defining planned execution of operations matching the characteristics and constraints within the re-manufacturing system. Thus, it would not

be possible to obtain a realistic schedule of re-manufacturing operations without being able of taking into consideration the complexity and uncertainty of the described re-manufacturing environment, entailing the need of the development and adoption of robust scheduling approaches, which are preventive schedules minimizing the effects of disruptions on the performance measure and ensuring that the predictive and realized schedules do not differ drastically, while maintaining a high level of schedule performance.

2. Problem Definition

The problem under study is a two-machine permutation flow shop where a set of n jobs $\{1, 2, ..., n\}$ are processed on two machines in series. The routing of the jobs through the shop is given and the processing time of job j on machine i, denoted as $p_{ij}, i = 1, 2; j = 1, ..., n$, is modeled as an independent random variable following a general exponential distribution, see Figure 2.1. Due to the uncertainty affecting processing times, the makespan, i.e., the length of time that elapses from the start of work to the end of this project, is also a random variable depending on p_{ij}, as well as on scheduling decisions. The aim of the scheduling approach is to mitigate the impact of extreme values of the processing times on the makespan. To this aim, the minimization of the VaR(value-at-risk) of the makespan is used as the objective function.

The formulation of the scheduling problem and the associated objective function is operated according to the notations proposed in [8]. A schedule decision vector $\boldsymbol{x}$ defines the positions of the jobs in the sequence while a vector of random variables $\boldsymbol{y} = \{p_{11}, ..., p_{n2}\}$ models the random processing times. These variables are governed by a probability measure P on Y and are independent of scheduling decision $\boldsymbol{x}$. The probability distribution of the makespan, $f_{C_{max}}(\boldsymbol{x}, \boldsymbol{y})$, depends on the values of $\boldsymbol{x}$ and $\boldsymbol{y}$. For a given schedule $\boldsymbol{x}$, the resulting cumulative density function (cdf) for the makespan is defined as:

$$F_{C_{max}}(\boldsymbol{x}, \zeta) = P(f_{C_{max}}(\boldsymbol{x}, \boldsymbol{y}) \leq \zeta | \boldsymbol{x}) \tag{2.1}$$

Then, the value-at-risk α (VaR_α) of C_{max}, associated with a schedule decision $\boldsymbol{x}$, denoted as $\zeta_\alpha(x)$, is defined according to the following:

$$\zeta_\alpha(\boldsymbol{x}) = min\{\zeta | F_{C_{max}}(\boldsymbol{x}, \zeta) \geq \alpha\} \tag{2.2}$$

Given that the scheduling decision vector $\boldsymbol{x}$ is independent from the values of the stochastic variables in $\boldsymbol{y}$, and C_{max} is a regular scheduling objective function [7], $C_{max}(\boldsymbol{x}, \boldsymbol{y})$ is continuous and non-decreasing in $\boldsymbol{y}$, its value can only remain the same or increase when a new job is added to the schedule. i.e., the value of the objective function of a partial schedule is a lower bound to the values of the objective function of schedules adding new jobs on it [6], see Figure 2.2.

In the next section, we develop a branch and bound algorithm to minimize the value-at-risk of the makespan and obtain the best schedule from a risk measure perspective.

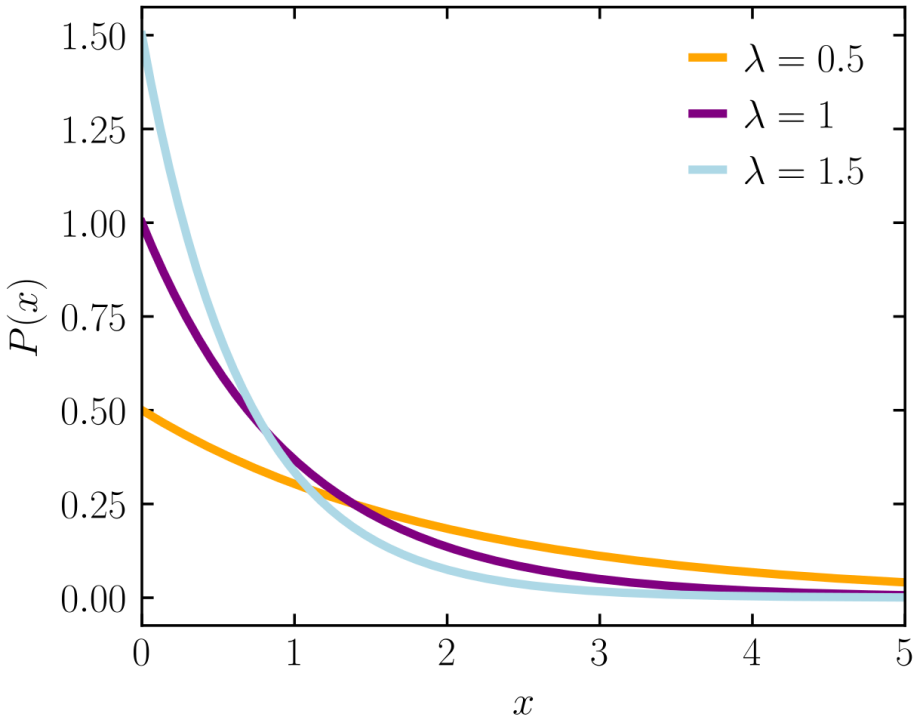

Figure 2.1: Samples of exponential distributions

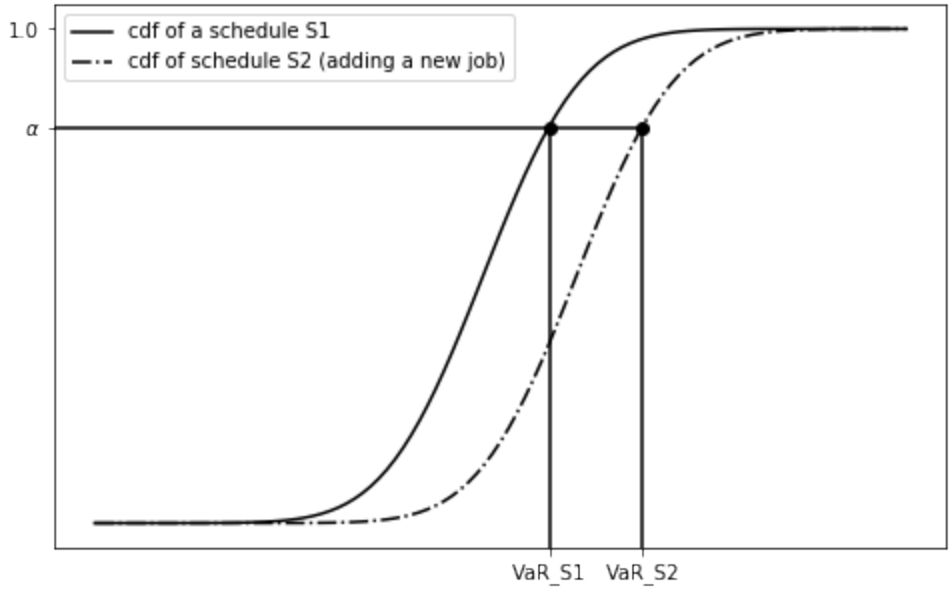

Figure 2.2: Lower bounds of VaR for a schedule

3. Branch and Bound Algorithm

Branch and bound algorithm (B&B) is an algorithm design paradigm for discrete and combinatorial optimization problems. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm [11].

The idea of the B&B algorithm is to detect the subspaces not containing the global minimizers and discard them from the further search. According to the B&B algorithm an initial problem solution should be initiated first. The initial search space D is subsequently divided into smaller subspaces D_i. There are several dividing/searching strategies, e.g., breadth first, depth first and best first. Each subspace is evaluated trying to find out if it can contain the optimal solution. For this purpose a lower bound for the objective function $LB(D_i)$ is calculated over the subspace and compared with the upper bound $UB(D)$ for the minimum value. If $LB(D_i) \geq UB(D)$, then the subspace D_i cannot contain the global minimizer and, therefore, it is rejected from the further search. Otherwise it is inserted into the list of unexplored subspaces. The algorithm terminates when there are no subspaces in the list.

In this chapter, we will introduce the branch and bound algorithm developed for the stochastic 2-machine flow shop scheduling problem. A branch scheme as well as the search strategy will be introduced, and non-leaves nodes evaluation, i.e., lower bound of partial schedule, and leaf nodes evaluation, i.e., full schedule, will be addressed in detail, followed by a simple initial solution schedule generation.

3.1 Branch Scheme and Search strategy

The branching scheme defines the decomposition strategy of a node in the branching tree. A forward branching scheme, which has the advantage of simple, is used, sequencing the jobs starting from the beginning of the schedule, see Figure 3.1

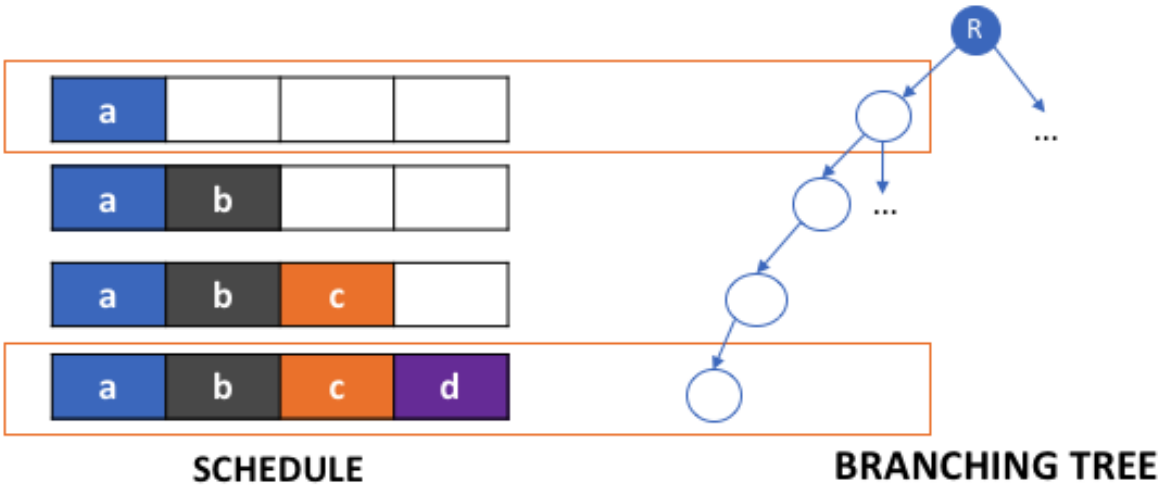

Figure 3.1: Forward branching scheme

As for the evaluation of the nodes, depth first search strategy, which starts at the root node and explores as far as possible along each branch before backtracking, is exploited. An example of depth first search strategy is presented in Figure 3.2, in which the dark node is already evaluated or exploring and orange nodes are not evaluated yet.

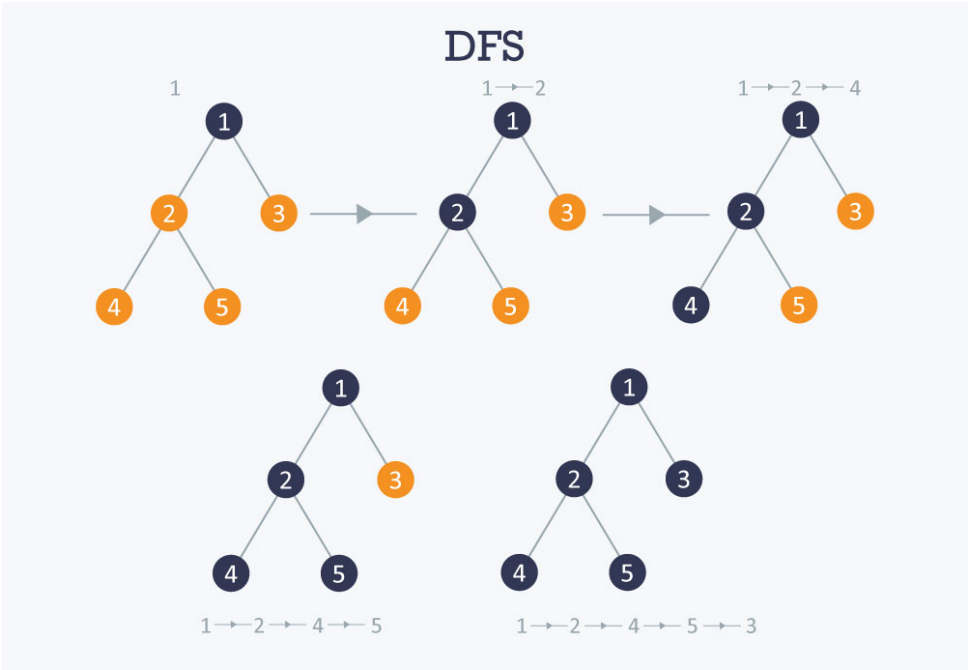

Figure 3.2: Depth first search strategy

3.2 Evaluation of the nodes

Nodes in the branching tree represent partial or complete solution to the scheduling problem. For leaf nodes, representing complete schedules, the value of the objective function has to be estimated. For the non-leaf nodes, representing partial schedules, proper bounds have to be derived to guide the exploration of the branching tree.

3.2.1 Evaluation of leaf nodes

Leaf nodes in the branching tree are associated with a complete schedule of the n jobs and entails the calculation of the value-at-risk α (VaRα) of the makespan of the schedule.

The makespan depends on the length of the critical path in the network of activities but, when stochastic processing times are considered, more than a single path have the probability to be critical. Hence, the estimation of the distribution of the makespan is intrinsically difficult to calculate. The evaluation of the nodes in the branching tree grounds on a Markovian Activity Network approach to calculate the distribution of the makespan, under the hypothesis that processing times are modeled through phase-type distributions.

The addressed 2-machine flow shop scheduling problem is represented with an Activity on Arc (AoA) network based on an acyclic directed graph $G = (V, A)$ with a set of nodes $V = v_1, v_2, ..., v_m$ and a set of arcs $A = a_1, a_2, ..., a_n$. Each arc in G represents an activity while the nodes in V represents states modeling the progress of the execution of the activities. At a given time t, an activity can only be in one of the following states [5]:

- active: it is being executed and it can be represented as (a)
- dormant: it has been completed but there is a uncompleted activity incident on the same destination node, this condition can be denoted (a^*)
- idle: it is neither active nor dormant

A 2-machine flow shop can be modeled with the AoA network in Figure 3.3. Grounding on this network, the set of states modeling the execution of the network can be obtained. Starting from the state that the first job is processing on machine 1, i.e., a_1, once activity a_1 is executed, it will transition to the state which both operations (first job on machine 2 and second job on machine 1) are processing, i.e., (a_2, b_1), and then it may transition to one of two independent states: first job finished on machine 2 and second job is processing on machine 1 (a_2^*, b_1); machine 1 has finished second job and has begun the third job, while first job is still processing on machine 2 (a_2, c_1), and so forth, until it goes into the absorbing state. For the problem under consideration, this scheme only depends on the number of jobs. Thus, for a given dimension of the scheduling problem, a general structure of this network is derived and used in all the different nodes, without the need of generating it multiple times, see Figure 3.4.

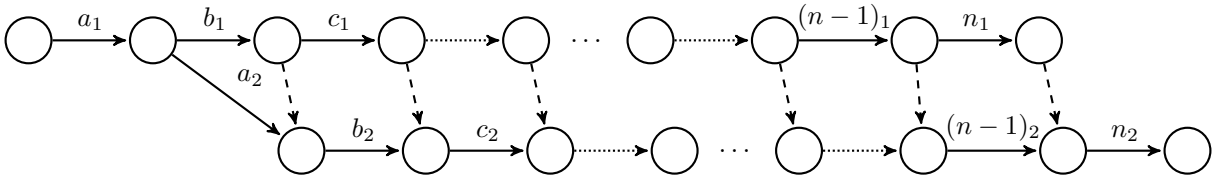

Figure 3.3: AoA activity network for a two-machine flow shop

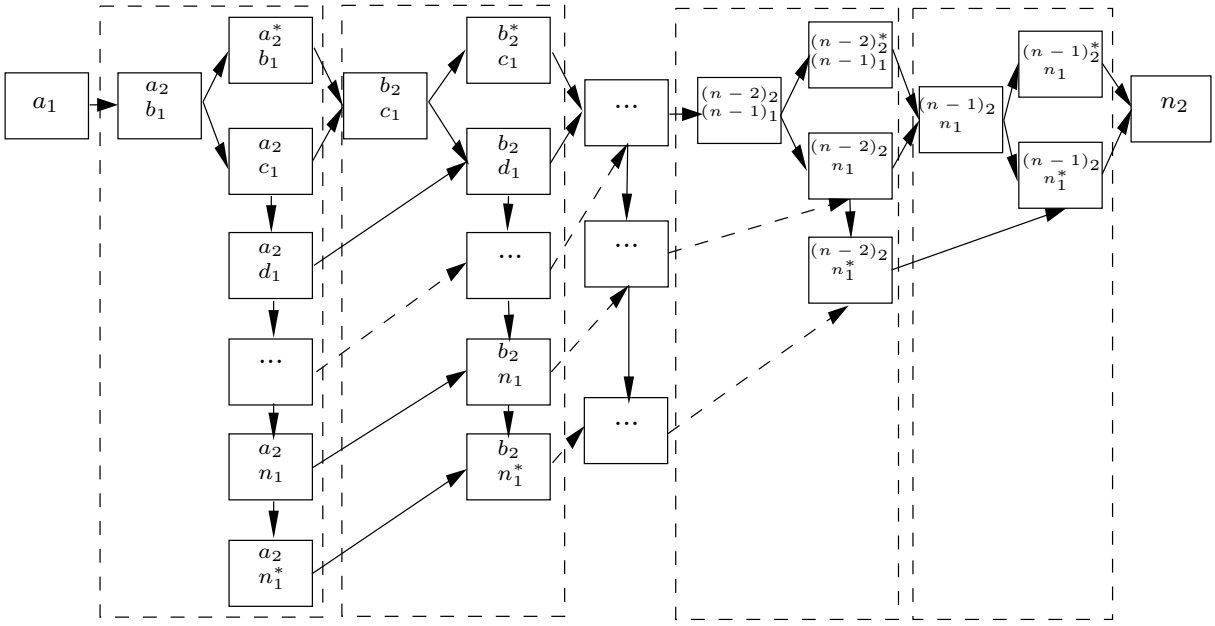

Figure 3.4: States generation scheme

Based on the states in Figure 3.4, the infinitesimal generator of the CTMC representing the execution of the network of activities with exponential distributions of the processing times can be obtained. Basically, in this project which all the distributions of operation follow exponential distribution with a single parameter λ, the size of the infinitesimal matrix equals to the number of states. The transition probability between two states is the parameter λ of the operation who was just finished. The diagonal values are calculated so that each row sums to zero.

Thus, the makespan of the network of activities is the time to absorption of the described CTMC, whose distribution can be calculated according to :

$$F(t) = 1 - \beta e^{Tt}\mathbf{1} \tag{3.1}$$

where β is the initial probability vector, T denotes the transition matrix not considering the absorbing state and $\mathbf{1}$ is an all-ones vector.

The quantile of this distribution corresponding to the VaR_α is obtained through a bisection method to find the root of:

$$\alpha = 1 - \beta e^{T*VaR}\mathbf{1} \tag{3.2}$$

where β, T and $\mathbf{1}$ are the same as in Equation (3.1), α is the considered risk level, and VaR is the value to be estimated. A lower and upper bound for the VaR to be estimated are also provided to the bisection method, to speed up the search for the root. The value of the VaR of the parent node is used as the lower bound value. In fact, since the VaR of makespan is a regular objective function, and due to the fact that a child node is obtained from the parent by inserting a new job in the schedule, then the VaR of the child node can only be larger or equal to the VaR of its parent node [7]. The upper bound value, on the contrary, is assigned the value of the current best solution. Thus, if the bisection method fails in finding the root, then it must be larger than the current best solution and the node can be pruned.

3.2.2 Evaluation of nodes representing partial schedules

For the nodes representing a partial schedule, i.e., only a subset of the jobs have been sequenced, bounds have to be obtained with respect to the considered objective function.

According to the described branching scheme in subsection 3.1, in a partial schedule s jobs have been already sequenced, while the sequencing of the remaining $n - s$ jobs is not decided yet. For the s assigned jobs, an approach similar to the ones described in subsection 3.2.1 is used to derive the associated network of activities (Figure 3.5).

On the contrary, for the not-sequenced jobs, their processing times on the second machine can be modeled as one single activity (k_2), thus leaving out the impact of the unknown precedence relations that could involve them. Grounding on the regular characteristic of Value-at-Risk of makespan objective function, this provides a lower bound for the leaf nodes derived from current node. The resulting AoA activity network is reported in Figure 3.5 and the same approach described in Sect. 3.2.1 is used to generate the state space, the CTMC considering phase-type distributed processing times and the estimation of the VaR.

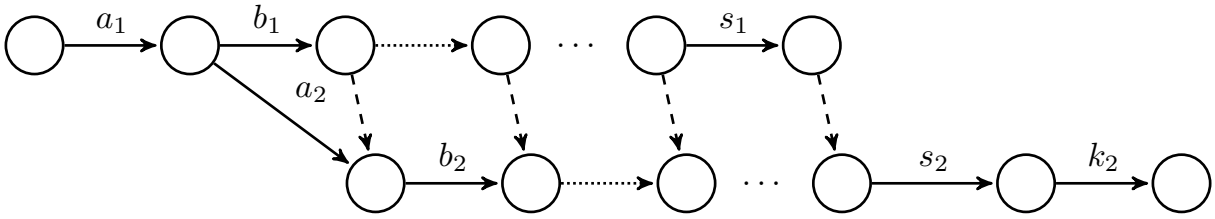

Figure 3.5: AoA activity network for partial schedule

3.3 Initial upper bound

With respect to the branch and bound algorithm for a minimization problem, an initial solution need to be obtained and the associated evaluation value set as an initial upper bound of the whole search space, the upper bound will be updated when smaller objective function value is explored during the search.

In this 2-machine flow shop scheduling project, the simplest initial schedule solution $[1, 2, 3, 4, ..., n]$ is exploited and the associated VaR value which is evaluated by leaf nodes evaluation approach in subsection 3.2.1 is set as the initial upper bound of the search space.

4. Bob++ Library

4.1 Bob++ framework

Bob++ is a C++ framework that facilitates the creation of sequential and parallel solvers based on node search algorithm such as Branch-and-Bound and Divide and Conquer algorithms [2]. The framework has been used to solve many optimization problems such as scheduling problems.

Bob++ library provides a set of C++ templates on top of which the user has to code some classes in order to configure the behavior of the algorithm. These templates are aimed at facilitating the development of search algorithms over irregular and dynamic data structures. Figure 4.1 shows how Bob++ interfaces between high-level applications and different possible parallel programming environments. The base classes of Bob++ and its parallel programming characteristic are introduced in the next sections.

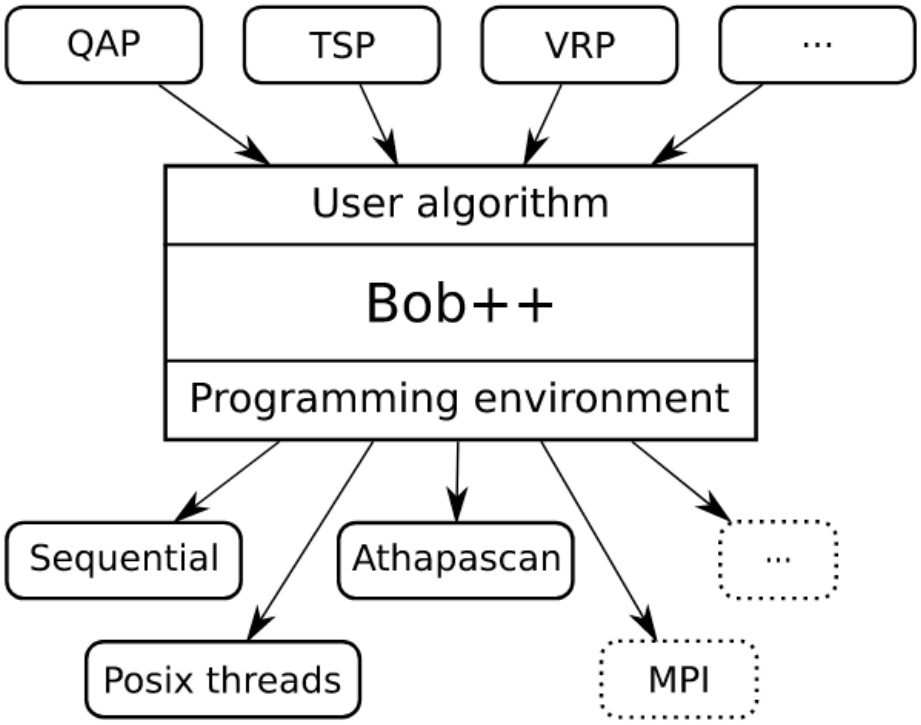

Figure 4.1: The structure of Bob++ applications

4.2 Base classes

In Bob++ library, each of the mentioned methods is implemented by four classes. The base classes for each of them are the following:

Instance:
The Instance class is used for the storage of all the global data of the application. This data is initialized at the beginning of the resolution. It is not allowed to modify its contents during the execution of the resolution.

Node:
The Node class enables the storage of the data and contains the methods associated to a node in the search space.

GenChild:
The GenChild class contains the method used to generate the child nodes of a given node.

Algo:
The Algo class contains the execution code of the main loop of the resolution.

The specialized *Instance*, *Node* and *GenChild* classes are called "the user classes", meaning that these classes must be customized by the user to implement the resolution of its own problem.

Bob++ makes an exhaustive use of templates in the class definition, the Bob++ classes are parametrized by a *Trait* class which only contains types definitions. The *Trait* must contain the definition for the *Node*, the *Instance*, the *GenChild* and the *Algo* but also for the *Stat* class and *Priority* class.

```
class MyTrait {
public:
    typedef MyNode Node;
    typedef MyInstance Instance;
    typedef MyGenChild GenChild;
    typedef Bob::BBAlgo<MyTrait> Algo;
    typedef Bob::DepthEPri<MyNode> PriComp;
    typedef Bob::BBStat Stat;
};

class MyNode : public Bob::BBIntMinNode { ....
};

class MyInstance : public Bob::BBInstance<MyTrait> {
    ....
};
```

```
class MyGenChild: public Bob::BBGenChild<MyTrait> {
....
};
```

Priority:
The *Priority* class contains the rules to schedule the search. Bob++ provides default classes derived from *Priority*. These default classes can be used when a standard node selection rule (depth first, best evaluation first, etc.) is enough. The Priority class must be defined in the Trait class.

Stat:
A *Stat* instance stores all the activities of an associated *Algo*. It is used for monitoring the execution of the resolution. The user can extend the definition of the default *Stat* class to add statistics or monitored values which correspond to its specific needs. The Stat is instantiated only once, hence the generated log data is global in this case. While the algorithm is running, a Stat object generates statistics about general data such as the number of generated nodes, evaluated nodes, or the number of calls to the generation method of the *GenChild*, as well as problem-specific information. It is possible to redirect the output of a Stat object to a file, hence the execution evolution can be analyzed after the execution is finished, or during the execution.

The Bob++ library allows different ways to schedule the search strategy (depth-first search, best-first search, etc.). The users can configure their search strategy by defining it in class *PFSTrait*.

In order to avoid the bottleneck caused by accessing the same set, several sets (priority queues) can be used. The dynamic load balancing strategy followed in Bob++ relies on work-stealing among sets in order to achieve better performance.

4.3 Parallel programming

The parallelizations of this library are based on Pthreads. The main idea leading the Bob++ design to obtain parallel solvers is a generalization of the Global priority Queue (GPQ). The principle is to have different instances of Algo that are executed on different processors. These different Algo communicate, and are synchronized using high level communication tools. These tools are mainly the data structures that store the sub-problems, the solutions or other information needed by a specific method. For example in the context of a B&B, these "parallel" instances of BBAlgo will mainly execute a loop where at each iteration the operator() of an instance of the user BBGenChild is executed on a pending node obtained from a Global priority Queue GPQ. The new generated nodes are re-inserted in GPQ. In the same way if a solution is found, the data structure called Global Solution GSol will be updated with this new solution. The different GPQ and GSol must communicate to ensure that each Algo has enough work to do and has the latest updated Solution. The GPQ and GSol can be considered from the Algo point of view

as high level communication tools since the different Algo instances only communicate through these tools.

5. Problem Solving with Bob++ Library

The implementation of branch and bound algorithm in solving the proposed stochastic 2-machine flow shop scheduling problem with the support of Bob++ library will be introduced in this chapter.

5.1 Project structure and source code files

Totally, five source code files are presented in this project, i.e., *main.cpp*,*pfs.h*,*pfs.cpp*,*lb.h*,*lb.cpp*. The details of each source code file will be presented in the following subsections.

5.1.1 main.cpp

```
//main function
int main(int m, char ** v){}
```

The project starts from *main* function, which will ask in the terminal window to input the path of the data file.

```
// data file name input from the command
std::cout << "Please enter the data file path : ";
std::string filename;
std::cin >> filename;
```

Multiple instances are presented in the data file, as well as in the experiments. Each instance will be solved by function *BBoneIns*. In this function, the instance data will be collected, bob++ library and branch and bound algorithm will be exploited.

```
// function BBoneIns to solve single instance
int BBoneIns(int n, char ** v, std::string filename, int line){}
```

Meanwhile, *Threaded* parameter is defined in this file, hence, multi-thread will be used in this project.

```
#define Threaded // Threaded parameter is defined and multi-thread will be used
```

```
#ifdef Threaded
  Bob::ThrBBAlgoEnvProg<PFSTrait> env;
  Bob::ThrEnvProg::init();
  Bob::core::Config(n, v);
  Bob::ThrEnvProg::start();
#else
  Bob::SeqBBAlgoEnvProg<PFSTrait> env;
  Bob::core::Config(n, v);
#endif
```

5.1.2 Data read and write

In this project, the data is stored in the data.txt file, which is at the same level folder of the codes, i.e., SUB-CLASS folder. The instances are defined by line, each line in the data file defines one instance. For each line, the first two integer number represent number of job and number of machine, respectively. The following float numbers denotes the parameter of exponential distribution, i.e., λ, of each operation, the order is as follows, job 1 on machine 1, job 1 on machine 2, job 2 on machine 1, job 2 on machine 2, and so on so forth.

The function to read data is at *read()* in *pfs.cpp*.

```
// read data function
void PFSInstance::read() {
  std::ifstream fr(file.c_str());
  if ( !fr.is_open() ) {
    std::cerr<< "Could not open file "<<file.c_str()<<" : exit\n";
    exit(1);
  }

  std::string feature;
  int count =0;
  while(!fr.eof()){
    getline(fr, feature);
    if(count == line){
      stringstream stringin(feature);
      try {
        stringin >> nbj;
        stringin >> nbm;
        if ( nbj != 0 ) {
          d_b = new double *[ nbj ];
        }

        for ( int i = 0; i< nbj ; i++ ) {
          d_b[i]= new double[ nbm ];
          for ( int j= 0; j < nbm ; j++ ){
```

```
            stringin >> d_b[i][j];
          }
         }
        }
        catch( ... ) {
          cout << "Error while loading data from stream..." << endl;
        }
       break;
    }else{
      count++;
    }
  }
  fr.close();
}
```

As mentioned, multiple instances are solved in this project, so a piece of code was prepared to store the results of each instance.The function to collect and write the results is at in *main.cpp*.

```
// The write function
void write_string_to_file_append(const std::string & file_string, const
    std::string str ){

  std::ofstream OsWrite(file_string, std::ofstream::app);
  OsWrite<<str;
  OsWrite<<std::endl;
  OsWrite.close();
}

// Collect the statistics
// Instance ID, Job_number, Threads_num, VaR_solution, Core_time, small_c,
    Big_C, Big_D, Big_E, Big_T, small_i, small_p,small_d,small_g
  std::ostringstream ss;
  ss << instanceID << "\t";    // Instance ID
  ss << nbj << "\t";           // Job_number
  ss << alpha << "\t";         // alpha
  ss << threads_num << "\t";   // Threads_num
  ss << bestG << "\t";         // VaR_solution
  ss << coreTime << "\t";      // Core time of this Instance
  ss << small_c << "\t";       // number of generated nodes
  ss << Big_C << "\t";         // number of generate call
  ss << Big_D << "\t";         // number of delete call
  ss << Big_E << "\t";         // Evaluation Time
  ss << Big_T << "\t";         // algorithm Time
  ss << maxBig_E << "\t";      // maxEvaluation Time
  ss << maxBig_T << "\t";      // max algorithm Time
```

```
  ss << small_i << "\t";       // inserted nodes
  ss << small_p << "\t";       // pruned nodes

// Write the collected results into the file
  write_string_to_file_append("../result.txt", ss.str());
```

5.1.3 pfs.h

Two header files are developed in this project, i.e., *pfs.h* and *lb.h*.

In the *pfs.h* header file, some classes which represent the structure of branch and bound algorithm, e.g., *PFSInstance*, *PFSNode* and *PFSGenChild*, are defined, the evaluation of leaf nodes is presented in class *Scheduled* as well.

To solve the stochastic 2-machine flow shop scheduling problem, the developed B&B algorithm is written by extending the three following "user" classes, *BBInstance, BBNode and BBGenchild*, of bob++ library. The details are described as follows:

1. To inherit the base class *Bob::BBInstance* which define a branch-and-bound instance from the original bob++ library, a virtual class *SchedulingInstance* is defined in this project, which will be a base class for any kind of scheduling problems wanted to be solved by this branch-and-bound scheme, e.g., single machine scheduling problem, flow shop scheduling problem and job shop scheduling problem. Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

2. The *BBInstance* class storage of all the information needed for the resolution of the specific problem. A class *PFSInstance* derived from *BBInstance* defines the permutation flow shop scheduling problem under investigation. Hence, the hierarchy would be like in Figure.

3. class *PFSNode* defines the nodes in the branch tree, a main function inside is to assign a new job to the node based on branching scheme rule. And the evaluation function of this node.

4. To be able to compare two *BBNode* objects according to their evaluation, the BBNode class is parametrized with the sense of minimsation. Shortcuts have been predefined, such that the user can choose one the different available default node classes i.e. BBMinIntNode, BBFloatMaxNode ... as a base class for its own Node class. In Bob++, only one type, one class is used to represent the subproblem or a solution of the problem.

5. class *PFSGenChild* inherited from *Bob::BBGenChild*, defines the branching scheme, e.g., forwarding branching, backward branching, alternative branching. In this project, forwarding branching scheme are exploited and developed in operator function.

6. class *Scheduled* defines a full schedule, i.e., leaf nodes with all jobs are assigned, the inside function can evaluate this schedule and update the current upper bound.

In the source code file *pfs.cpp*, the corresponding functions of each class are implemented.

5.1.4 lb.h

This header file defines a important class *OneMachine* which gives the lower bound value of each partial schedule, i.e., non-leaves node. The related functions are defined and implemented in file *lb.cpp*.

5.2 Compilation

This project is taking advantage of compiling tool $g++$, in line 113-114 of *Makefile* file, the command line are defined as follows.

```
CXX = g++  -I
CXXFLAGS = -g -O0 -Wall
```

g++ tool is exploited, data file name need to be input from Terminal command, *CXXFLAGS* controls the compiling

- -g - turn on debugging
- -Wall - turns on most warnings
- -O or -O3 - turn on optimizations
- -o <name> - name of the output file
- -I<include path> - specify an include directory

5.3 Test Case

In this section, a test case with full procedure from bobpp installation, data preparation to results collection is presented.

5.3.1 Libraries Installation and running guideline

As stated in README file of the project, the installation of bobpp in UNIX operating system is straightforward, here some basic installation steps are presented.

1. Open Terminal of operating system, "cd" to the folder "bobpp_lib"

2. input command "*./configure; make; make install*"

The library should be installed successfully now, By default, 'make install' installs the package's commands under '/usr/local/bin', include files under '/usr/local/include', etc, it can be checked as follows:

1. INCLUDE DIR: if you go in the folder "/usr/local/include", a folder named "bobpp-1,0/bobpp" was constructed, and here you will find the needed headers (.h files) that are needed to include the bob library in your projects

2. LIB DIR: in the folder "/usr/local/lib", a set of libraries (e.g., libbobpp-1.0.dylib) already compiled

Besides bobpp library which used for the branch and bound algorithm, Eigen library should be installed to do the matrix computation operations. The Installation of Eigen library should follow the steps:

1. In order to use Eigen, you just need to download and extract Eigen's source code, copy the Eigen folder into "/usr/local/include/"

2. If eigen has already been installed, check the path of MatrixFunctions, Core, Dense, i.e., /src/pfs.h, line 19,20,21, /src/lb.cpp, line 17,18,19("#include <Eigen/unsupported/Eigen/MatrixFunctions>", "#include <Eigen/Core>", "#include <Eigen/Dense>"), make sure it is consistant with your installation folders

Boost library is optional in this project, several alternative functions used it, it can be commented and run without boost library.

1. If you didn't install boost in your laptop, you can download it
   https://www.boost.org/users/history/version_1_79_0.html,
   and follow the guideline of
   https://www.boost.org/doc/libs/1_79_0/more/getting_started/unix-variants.html#get-boost

2. Defaultly, the path to the boost root directory is "/usr/local/boost_1_73_0", and it should be added in Makefile, which is located in folder src/Makefile, line 113, CXX = g++ -std=c++11 -I /usr/local/boost_1_73_0, here "/usr/local/-boost_1_73_0", if it's not the case in your laptop, replace this line with your own one

Note that old version of boost library and Eigen library is enough, no need the latest version. After libraries' installation, the project can be run.

1. Open Terminal, "cd" into the *src* folder

2. Compile the project with the command: $ make

3. After successful compiling, we can see executable file "pfs" in this same folder, otherwise the compiling was failed. This was tested under the g++ compile tool and Mac/Unix operating system

4. run the executable file, $ ./pfs

5. Data file name will be asked to be input, the data file was put in project folder, so input $: ../data.txt, then press return

6. The algorithm will be running in the Terminal, and the statistical results are output in the file PACS_Project/result.txt

5.3.2 Input and output

The input data file is put in the main folder, i.e., data.txt. Each line in this file denotes one instance, five instances with eight jobs, i.e., 16 activities, are prepared. The first four numbers of each line, denotes the number of jobs, number of machines, initial lower support for VaR calculation, and initial upper support for VaR calculation, respectively, followed by the λ parameter of exponential distributions for 16 activities, see Figure 5.1

```
6  2  1 500   0.056  0.2  0.143  0.25  0.2  0.027  0.027  0.024  0.032  0.033  0.017  0.016  0.023  0.024  0.017  0.02
6  2  1 500   0.1  0.143  0.25  0.067  0.059  0.023  0.022  0.031  0.03  0.03  0.014  0.023  0.015  0.017  0.024  0.013
6  2  1 500   0.062  0.25  0.053  0.077  0.059  0.029  0.03  0.029  0.021  0.032  0.019  0.022
6  2  1 500   1.263   1.998   1.211   1.68    1.247   0.306   1.225   1.884   0.856   0.485   1.261   0.441
6  2  1 500 0.2  0.25  0.077  1.0  0.083  0.021  0.021  0.031  0.032  0.024  0.016  0.017
```

Figure 5.1: Example of input data

The output file, i.e., result.txt, is in the same folder as data.txt, which collect the statistical results of each instance, several key indicators are presented, see Figure 5.2. The indicators are explained in Table 5.1.

InstanceID	Job_number	alpha	Threads_num	VaR_solution	Core_time	Evaluated Nodes
1	6	0.99	5	368.164	4.86515	556
2	6	0.99	5	380.859	6.1132	618
3	6	0.99	5	361.328	6.30843	716
4	6	0.99	5	16.6016	0.311966	82
5	6	0.99	5	382.812	4.36401	476

Figure 5.2: Example of output data

Table 5.1: Description of indicators of statistical results

Indicators	Description
InstanceID	Index of instance
Job number	Number of jobs
alpha	Value-at-risk level
Threads number	Number of threads
VaR solution	VaR value of optimal schedule
Core time	CPU time to solve this instance
Evaluated nodes	Number of nodes evaluated in the branch tree for this instance

6. Scientific Computing

Besides the algorithm development and project structure, several scientific computing topics are exploited in this project, especially, on the leaf nodes and partial schedule evaluation part.

6.1 Eigen library and CTMC infinitesimal matrix generation

Due to the large number of matrix related operations, Eigen library, which is a high-level C++ library of template headers for linear algebra, is exploited. Eigen is implemented using the expression templates metaprogramming technique, meaning it builds expression trees at compile time and generates custom code to evaluate these. Using expression templates and a cost model of floating point operations, the library performs its own loop unrolling and vectorization.

Specifically, matrix and vector with dynamic size, i.e., typedef MatrixXd and VectorXd, meaning a matrix/vector of doubles with dynamic size, is defined.

The following codes, i.e., function structureCon, generates the matrix corresponding with the CTMC is Figure 3.4. To save memory space, return *void* and reference MatrixXd &Qstar are used in this function, instead of return large matrix.

```
// The matrix generation function for leaf node
void Scheduled::structureCon(std::vector<int> PS, const PFSInstance *pfi,
    MatrixXd &Qstar){
  int n = PS.size();
  int states = (pow(n,2)+3*n+2)/2;
   MatrixXd Qstar1 = MatrixXd::Zero(states, states);
    // get the states number for each block
    std::vector<std::vector<int> > stateNumVec = getstateNum(n);

    // get the matrix for each block
    for (int k = 0; k < n-1; ++k){
      int count = stateNumVec[k].size()-1;

      Qstar1(stateNumVec[k][0], stateNumVec[k][1]) = pfi->d_b[PS[k]][1];
      Qstar1(stateNumVec[k][0], stateNumVec[k][2]) = pfi->d_b[PS[k+1]][0];
```

```
      Qstar1(stateNumVec[k][1], stateNumVec[k+1][0]) = pfi->d_b[PS[k+1]][0];
      Qstar1(stateNumVec[k][2], stateNumVec[k+1][0]) = pfi->d_b[PS[k]][1];

      if(stateNumVec[k].size() > 3){
        for (int i = 2; i < stateNumVec[k].size()-1; ++i){
          Qstar1(stateNumVec[k][i], stateNumVec[k][i+1]) =
              pfi->d_b[PS[n-count+i]][0]; // need a enumerate of jobs until last
              one
          Qstar1(stateNumVec[k][i+1], stateNumVec[k+1][i]) = pfi->d_b[PS[k]][1];
        }
      }
    }
    Qstar1(states-2, states-1)= pfi->d_b[PS[n-1]][1];
    Qstar1(0, 1) = pfi->d_b[PS[0]][0];


    for (int i = 0; i < states; ++i){
        for (int j = i+1; j < states; ++j){
            Qstar1(i,i) += (-1 * Qstar1(i,j));
        }
    }
    Qstar = Qstar1.topLeftCorner(states-1,states-1);
}
```

6.2 Matrix exponential operation

Grounding on the CTMC infinitesimal matrix, to calculate the Value−at−Risk, the matrix exponential operation, i.e., Equation 3.1, which is used to calculate the distribution of makespan, should be exploited. Note that matrix exponential operation is a really time consuming operation, especially when the size of matrix, i.e., the number of jobs, is large, which would be a bottleneck of the proposed algorithm. Hence, several matrix exponential approaches are proposed to speed up the matrix exponential operation, an analysis of different approaches are introduced as well.

6.2.1 Eigen toolbox exp()

The first approach is exploiting the toolbox from Eigen, i.e., T.exp()

```
const MatrixExponentialReturnValue<Derived> MatrixBase<Derived>::exp() const
```

The matrix exponential is computed using the scaling-and-squaring method combined with Padé approximation. The cost of the computation is approximately $20 * n^3$ for matrices of size n.The matrix is first rescaled, then the exponential of the reduced matrix

is computed approximately, and then the rescaling is undone by repeated squaring. The degree of the Padé approximant is chosen such that the approximation error is less than the round-off error.

This approach is easy to use thanks to the Eigen library, the accuracy is demonstrated by researchers, while when the matrix size is large, this approach is burdensome.

6.2.2 Krylov approach

Krylov approach is another popular approach to approximate the matrix exponential operation, it can be calculated as follows:

$$F(t) \approx \boldsymbol{e}_1^T \cdot \boldsymbol{V_m} \cdot \boldsymbol{e}^{H_m t} \cdot \boldsymbol{e_1} \tag{6.1}$$

where $\boldsymbol{V_m}$ and $\boldsymbol{H_m}$ are produced by the well-known Arnoldi process.

In the Matlab toolbox, this was packaged as ***expv*** and provided by RB Sidje [9], in this project, it has been translated to $C++$ codes and presented in file pfs.cpp.

6.2.3 CAM approach

CAM approach, which was proposed by AH Al-Mohy and NJ Higham [1] ten years ago, is another matrix exponential approximation approach, which is dedicated to large sparse matrix. Truncated Taylor series was exploited in this approach.

$$T_m(S^{-1}\boldsymbol{Q}) = \sum_{j=1}^{n} \frac{s^{-1}\boldsymbol{Q}^j}{j!} \approx \boldsymbol{e}^{s^{-1}\boldsymbol{Q}} \tag{6.2}$$

In the Matlab toolbox, this approach was packaged as ***expm***, similarly, it has been translated to $C++$ codes in this project and presented in file pfs.cpp.

6.2.4 Analysis

Based on the proposed matrix exponential approaches, experiments and analysis are conducted to compare these approaches.

Different sizes of matrix are tested under various approaches, and the efficiency of each approach can be concluded in Table 6.1.

Table 6.1: Solution time for a single matrix exponential operation(seconds)

Matrix size	Eigen	CAM	Krylov
100	0.008	0.015	0.01
150	0.014	0.018	0.01
300	0.05	0.03	0.02
600	0.26	0.05	0.16

Grounding on the experiments, the choice of matrix exponential approaches is concluded in Table 6.2. The Eigen ***exp*** can be used when the matrix is relatively small, with the increasing of matrix size, Krylov approach is preferable, the CAM approach would be exploited when the matrix is larger, while the normalization is needed in this approach, i.e., the smaller input value t is preferable.

Table 6.2: Choice of matrix exponential approaches

Matrix size	Approach
$<= 100$	Eigen ***exp***
$<= 500$	Krylov approach
> 500	CAM approach

6.3 Root finding operation

Besides the matrix exponential operation, finding the VaR value, i.e., quantile, from the cdf of makespan distribution, is another aspect to be improved to accelerate the algorithm. Hence, several root finding methods are developed and analyzed to find the root of a equation.

6.3.1 Bisection method

The most straightforward approach to find a root of an equation is bisection method. This approach has the advantage of easily implementation, and many libraries such as *boost* provides the related templates.

```
// boost template for bisection method
template <class F, class T, class Tol>
std::pair<T, T>
   bisect( // Limited iterations.
```

```
      F f,
      T min,
      T max,
      Tol tol,
      boost::uintmax_t& max_iter);
```

```
// The implementation of our project exploiting boost bisection method
double Scheduled::boost_Bisect(MatrixXd Qstar, VectorXd a,double x1lb, double
    x2ub, double VaRalpha){
    double root = x2ub;
    const boost::uintmax_t maxit2 = 50;
    boost::uintmax_t it2 = maxit2;
    tolerance tol = 1;
    // catch error if root not in this interval
    try{
        std::pair<double, double> found1 = boost::math::tools::bisect(
        [Qstar, a, VaRalpha](double x){return f(Qstar,a, x, VaRalpha);},
        x1lb,
        x2ub,
        tol,
        it2);

        root = found1.second;
    }catch(...){
        root = x2ub;
        it2 = 2;
    }
    if (root < x1lb+2){
        root= x1lb+2;
    }
    std::cout << it2 << std::endl;
  return root;
}
```

To save the iterations during the bisection, we set the lower interval support of bisection method as the VaR value of the parent node, and the upper interval support as the current best solution of the algorithm, see Figure 6.1. Thus, if the root of a node is not in this interval, it must be larger than current best solution and it should be pruned and no need to evaluate, the root not finding error will be caught and return the upper interval.

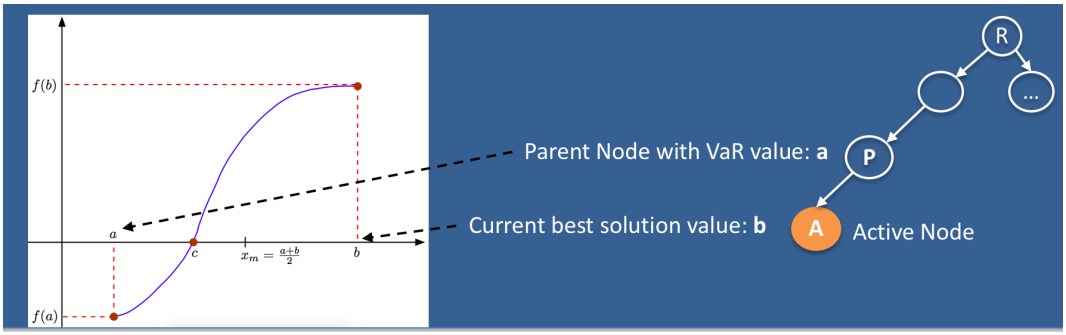

Figure 6.1: Bisection approach

6.3.2 Boost bracket and solve method

Boost library provide another root finding approach, i.e., bracket and solve method, which is a convenience function that calls TOMS 748 algorithm internally to find the root of f(x). It is generally much easier to use this function rather than TOMS 748 algorithm, since it does the hard work of bracketing the root for you. It's bracketing routines are quite robust and will usually be more foolproof than home-grown routines, unless the function can be analysed to yield tight brackets.

```
template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   bracket_and_solve_root(
      F f,
      const T& guess,
      const T& factor,
      bool rising,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);
```

The bracket and solve method was developed and tested in this project, since there is no lower/upper interval exploiting in this method, firstly the current best solution will be tested before finding the root, if the value of current best solution in this equation has been already large than 0, then this node can be pruned without evaluating.

6.3.3 False position method

False position method, or Regula-Falsi method, is another iterative methods of finding the roots of a non-linear equation. This method calculates new root from $[a_0,b_0]$ based on equation Eq.6.3. The false position method differs from the bisection method only in the choice it makes for subdividing the interval at each iteration. It converges faster to the root because it is an algorithm which uses appropriate weighting of the initial end points

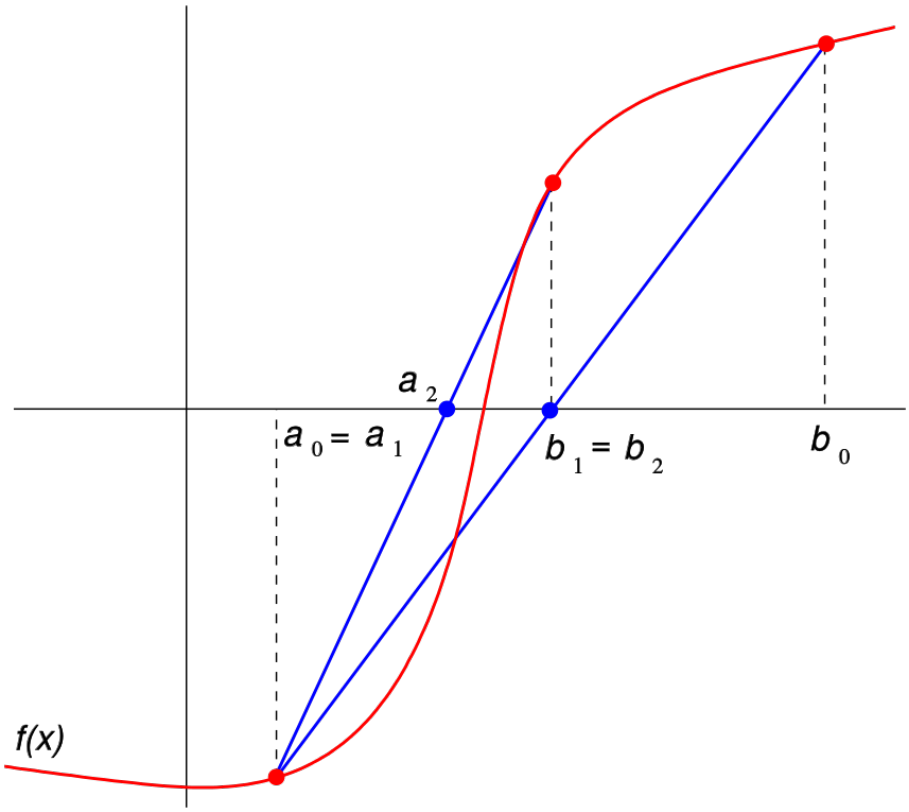

Figure 6.2: False position method

a_0 and b_0 using the information about the function, see Figure 6.2.

$$new\_root = \frac{b_0 f(a_0) - a_0 f(b_0)}{f(a_0) - f(b_0)} \tag{6.3}$$

6.3.4 Illinois method

The Illinois algorithm iterative procedure uses two estimates as in false position method to estimate a new estimate, the estimate which is replaced is the one for which the sign of the function is the same as the sign of the function for the new estimate, but if the same endpoint (right or left) is replaced in succession, then the ordinate of the opposite endpoint is halved, see Figure 6.3.

$$new\_root = \frac{b_0 f(a_0) - a_0 f(b_0)/2}{f(a_0) - f(b_0)/2} \tag{6.4}$$

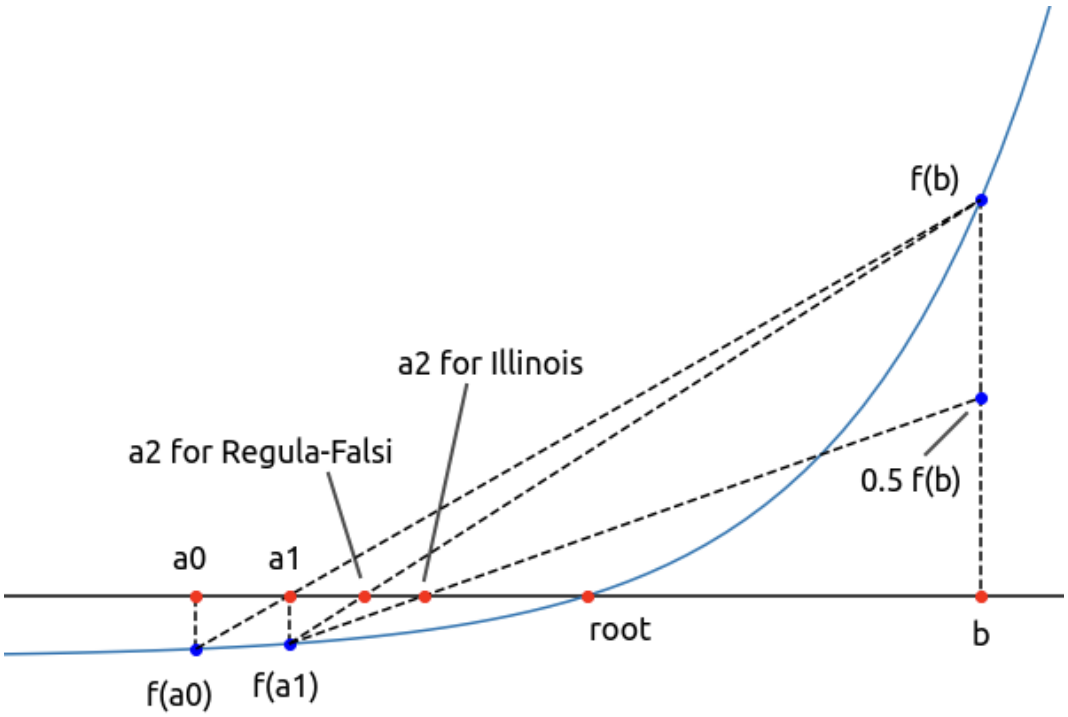

Figure 6.3: Illinois method

6.3.5 Analysis

Grounding on some tests on the instances, we concluded the number of iterations during each root finding operation for different approaches in Table 6.3, we can see that Illinois algorithm and Boost bracket and solve toolbox have the significant improvement comparing to other approaches, and these two will be the candidate approaches in this project.

Table 6.3: Number of iterations during the root finding

Iterations	Average	Min	Max	Median
Bisection	7.8	2	11	8
False Point	6.2	2	19	6
Illinois	4.9	2	10	5
Boost Bracket	5.1	1	9	5

7. Conclusions and Future Improvements

In this project, a stochastic 2-machine flow shop scheduling problem was studied, a branch-and-bound algorithm exploiting bobpp library was developed to solve this problem.

In the future, this simple exponential distribution can be extended to general phase-type distribution, and Kronecker algebra can be conducted. Furthermore, complex manufacturing systems can be studied under this structure, and corresponding hierarchy can be designed to solve more scheduling problems.

Bibliography

[1] Awad H Al-Mohy and Nicholas J Higham. Computing the action of the matrix exponential, with an application to exponential integrators. *SIAM journal on scientific computing*, 33(2):488–511, 2011.

[2] A. Djerrah, B. Le Cun, V.-D. Cung, and C. Roucairol. Bob++: Framework for solving optimization problems with branch-and-bound methods. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 369–370, 2006.

[3] Michael Ducker. The Fall of the F-Class Turbine. `https://www.power-eng.com/emissions/policy-regulations/the-fall-of-the-f-class-turbine/#gref`, 2015. [Online; accessed 19-January-2022].

[4] Ansaldo Energia. Ae94.3a gas turbine manufactured in ansaldo energia factory, genoa, italy. `https://www.ansaldoenergia.com/business-lines/new-units/gas-turbines/ae94-3a`. [Online; accessed 19-Dec-2021].

[5] Vidyadhar G Kulkarni and VG Adlakha. Markov and markov-regenerative pert networks. *Operations Research*, 34(5):769–781, 1986.

[6] Chenghu Ma and Wing-Keung Wong. Stochastic dominance and risk measure: A decision-theoretic foundation for var and c-var. *European Journal of Operational Research*, 207(2):927–935, 2010.

[7] ML Pinedo. Scheduling: Theory, algorithms, and systems fifth. *Cham: Springer International Publishing*, 2016.

[8] R Tyrrell Rockafellar and Stanislav Uryasev. Conditional value-at-risk for general loss distributions. *Journal of banking & finance*, 26(7):1443–1471, 2002.

[9] Roger B Sidje. Expokit: A software package for computing matrix exponentials. *ACM Transactions on Mathematical Software (TOMS)*, 24(1):130–156, 1998.

[10] Wikipedia. Turbine Blade. `https://en.wikipedia.org/wiki/Turbine_blade`, 2009. [Online; accessed 19-Dec-2021].

[11] Wikipidea. Branch and Bound. `https://en.wikipedia.org/wiki/Branch_and_bound`. [Online; accessed 19-May-2022].