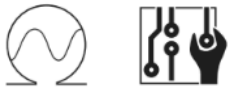
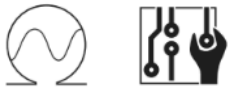


Technische Universität München
Department of Electrical Engineering and Information Technology
Institute for Electronic Design Automation

Introducing a new approach for mixer assignment, routing, and storage of fluid in Microfluidic Biochips

Master Thesis

Yasamin Moradi



Technische Universität München
Department of Electrical Engineering and Information Technology
Institute for Electronic Design Automation

Introducing a new approach for mixer assignment, routing, and storage of fluid in Microfluidic Biochips

Master Thesis

Yasamin Moradi

Supervising Professor: Prof. Dr.-Ing. Ulf Schlichtmann

Abstract

Microfluidic biochips technology is an appealing concept that has been developing in a wide range of applications over the past years. In this concept, a miniaturized biology laboratory is implemented on a single chip. These chips consist of different operating components and mixers are one of the most important components of biochips. Flow-based biochips use micro mechanical valve and tubes to control the flow of liquid. By actuating valves in specific order, liquid can be led to different paths and modules on the chip. By routing, mixing and performing other operations on the input liquid, the chip is able to execute different assays. Compared to transferring liquid samples between equipment in a big laboratory, using small biochips can provide more precise and reliable result in much shorter time and smaller area. Biochips need very small amount of reagent, and reduce the cost and effort of an experiment significantly. Emergence of these devices in many complicated chemical and biological experiments has increased the need for computer aided design and synthesis techniques. Since the concept of operation assignment to modules and routing in biochips and electronic chips are relatively similar, many techniques of electronic chip design has been modified and applied for biochips automated design. Various algorithms and methods have also been introduced for automated design of these chips, but a number of problems are not solved yet. One of the most neglected problems in biochip design is the fact that valves can only be actuated reliably for a few thousand times before they wear out and affect the whole function. In order to increase the performance and optimize the chip, it is required to decrease the number of actuations for the valve that is used the most. Recently a research has been done on reducing the number of valve actuations by building dynamic paths and components on the chip. In this method, valves and tubes form storage units and mixers. Compared to these dynamic mixers, conventional mixers are privileged by their performance. The purpose of this work is to introduce a new architecture for flow-based biochips, based on a matrix of valves and conventional mixers. This architecture combines the advantages of conventional mixers and dynamic generation of paths and storage units. The valve matrix can be programmed to move liquid in certain patterns within the matrix, and connect mixers by generating dynamic paths. Therefore biochips with this architecture can be programmed to carry out different assays, without the need for changing the design of the chip.

Acknowledgement

I would like to thank Professor Schlichtmann for supervising this master thesis, and Dr. Bing Li for his help and guidance through this project.

I am very grateful to my family and loved ones for their unconditional support throughout my education. This work would not have been possible without them.

Contents

List of figures

List of tables

1 Introduction

1.1. Lab on a Chip

Experimenting on liquid samples is a common step in the process of a biomedical or chemical research, during a medical exam, in the process of treatment, and in numerous other applications. As an example, a biologist needs to perform certain experiments on liquid sample to determine specific characteristics of the sample or its composition. A biomedical researcher may need to perform tests on a sample to understand the impact of a new medicine. A doctor may need to examine any of the body liquids from a patient to find out the type of an illness, or to prescribe the right medicine. And a patient may need to monitor the amount of certain substance in their blood to know they are in a stable situation. Or in a simple case, one might want to determine their blood type by mixing their blood sample with certain chemicals. There are countless of such applications in which an experiment is required to be done on a sample liquid. In conventional way, the experiment is done in a laboratory, with the help of laboratory equipment. Sample and the entire reagent are transported from one large device to the other by the laboratory staff and observed carefully.

The term Lab On a Chip (LOC) refers to a device capable of executing one or more of such laboratory functions, and is designed to help automation and high throughput execution of functions [1]. These chips are very small in size, and deal with very small amount of samples, meaning microliters and nanoliters. In LOCs, liquid is carried inside channels, and different operations are done on the liquid in different modules on the chip. For example two liquid samples can be carried through channels from two input ports to a specific module on the chip, to get mixed. The structure of this channels and modules will be discussed later in this chapter. Manipulation of liquid in LOCs has been made possible by advancement of microfluidics, which is the area of science focusing on significantly small amounts of liquid and its behavior [2].

As it is mentioned earlier, the first advantage of LOCs over conventional laboratories is the significantly small size of the chip. A conventional laboratory is usually a large room, with complicated equipment, while a LOC is a tiny chip, which dimensions are usually in range of a few centimeters. The other important advantage of LOCs is consumption of very low liquid volume. While for an experiment in a laboratory, some milliliters of a sample might be required. By using LOCs this amount will be reduced to micro and nanoliters, meaning the whole experiment can be done using a drop of the sample and reagents. This is very important when dealing with precious samples and expensive chemicals, and can reduce the cost of the assay. Decreasing the required volume of a reagent from one milliliter to one microliter can decrease the cost of input liquid to about 0.1% of the previous cost which is a significant reduction. Since these chips are working with very small amount of liquid and due to the very short distance that the liquid needs to travel inside the chip, LOCs can carry out the experiment faster and have a shorter response time. For example identifying target pathogens takes at least 2 days in

conventional laboratories. But for doing this experiment using LOC, only a few minutes are required [3]. Some specific operations such as heating and detection can also be executed faster because of the low liquid volume. These chips are also more precise and can provide and much more accurate result [4]. Laboratory on a chip can also provide a safer platform for dealing with dangerous or toxic chemicals, since these chemicals are out of reach after they are inserted into the chip. Even if they were reachable their volume was very small. Another important advantage of these chips, which should not be dismissed, is the very low cost of fabrication. This low cost allows producing cheap and disposable devices that can be used fast and easy once and disposed afterwards. This has a big influence on the usage of LOCs in clinical diagnostic and in at home applications, just as instance. LOCs can also be reused by using some methods for cleaning the channels and modules inside the chip before the experiment.

Although there are several advantages in using LOCs over conventional laboratories, there also exist some disadvantages to them. First of all, laboratory on a chip is a new concept and is not fully developed yet. Therefore not all the experiments can be carried out on these chips and the methods can still improve a lot. There are several unsolved problems in this field that researchers are still working on. A very important issue is that a significantly small sample might not represent the whole target liquid very well. For example a nanoliter of a blood sample may not contain the same portion of a certain substances that a milliliter does. As a result, all the advantages and disadvantages of laboratory on a chip must be considered carefully before usage in an application.

Microfluidics-based biochips are one type of laboratory on a chip. These chips work based on flow of fluid and are used for biochemical analysis and in biomedical applications, and are recently receiving much attention [4]. Advancements in microfluidics-based biochip design is making it possible to use these chips in various applications. Enzymatic analysis, DNA analysis, clinical diagnostics, and continues and real time monitoring of biochemical toxins level in air or water for finding any threat, are just a few examples of these applications [4], [5], [6].

Digital microfluidics-based biochips are another type of Laboratory on a chip, which operate on discrete droplets of liquid. The focus of this work is on flow-based biochips. Figure 1 from [5], and Figure 2 from [2] show examples of digital microfluidics-based biochips and flow-based microfluidics-based biochips.

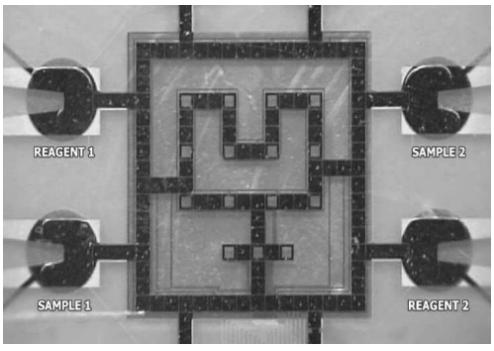


Figure 1

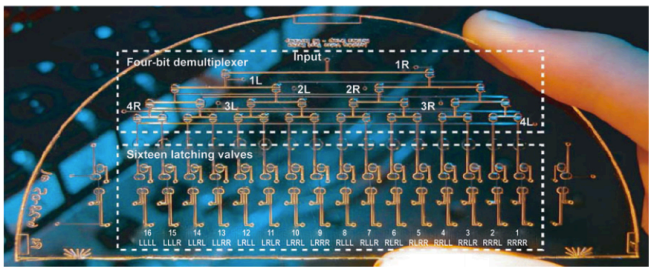


Figure 2

1.2. Biomedical assay

An assay is basically a procedure for analysis of a sample. This analysis is usually done to assess presence of a substance in the sample, or for measuring its amount. This procedure is a sequence of single operations that are carried out in the right order and under the right circumstance. Therefore an assay can be modeled and shown as a directed graph, called sequencing graph. in this graph, nodes represent the operations and arrows (directed edges) represent the transportation of the result of an operation to the next.

Figure 3 shows an example of the sequencing graph for an assay. In this example, liquid 1, liquid 2, and liquid 3 are the inputs of the assay as in1, in2, and in3. In the procedure of this assay, first in1 and in2 are mixed, then the result of this mixing is heated up. At last the heated result is mixed with in3. The result of the last mixing operation is the output. The presence of some chemical can be measured in this output as an instance.

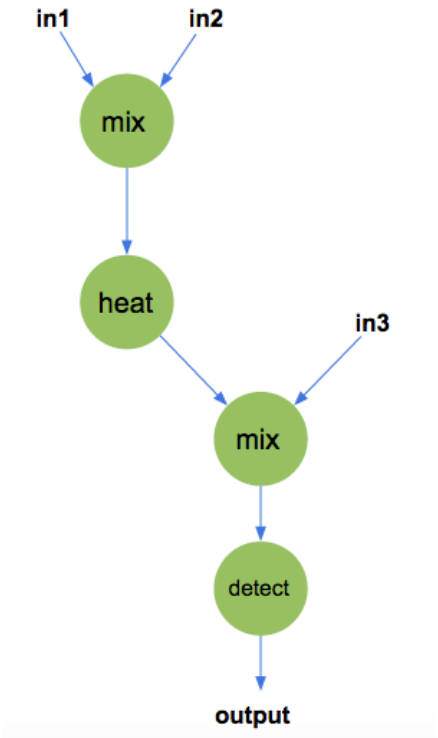


Figure 3

1.3. Modules

Biochips are consisted of various modules that each are able to perform a specific operation. On a biochip, liquid can be transferred to one of these modules for a specific operation. When the liquid reaches the module, execution of the operation is started. Different operations have different execution time. The execution for one operation can also vary depending on the module.

Since there are various types of operations defined in bioassays, there are also various types of modules introduced to perform these operations. Some of the most used modules are dispenser, mixer, heater, detector, storage, etc. which are respectively responsible for dispensing liquid from a source into the chip, mixing two liquids, warming the reagent, detecting the presence of some chemical in the liquid, storing the result of an operation until further use. Among all these modules, mixers are one of the most widely used. In this work, the focus is mainly on the mixers as well.

A conventional mixer has a round shape and consists of a circular flow channel and 9 valves. Structure of the valves will be discussed in the next section. Layout of a conventional mixer and its appearance after fabrication can be seen in Figure 4 (b) from MIT presentations. As it is shown in this figure, micro valves in a mixer are grouped as control valves and pump valves. Although the physical structures of these valves are identical, they differ in their role. Control valves are responsible for letting liquid into and out of the mixer. They basically act as doors. As an example, by opening red valves on the left in Figure 4 (a), liquid can flow the mixer's pipes. The other group of valves, pump valves, are responsible for circulation of the liquid inside the mixer, which results in mixing. By actuating the pump valves in Figure 4 (a) in specific sequence and for specific number of times, liquid will circulate inside the mixer's pipes [7].

After a mixing operation is over inside the mixer, by actuating the control valves, liquid can leave the mixer.

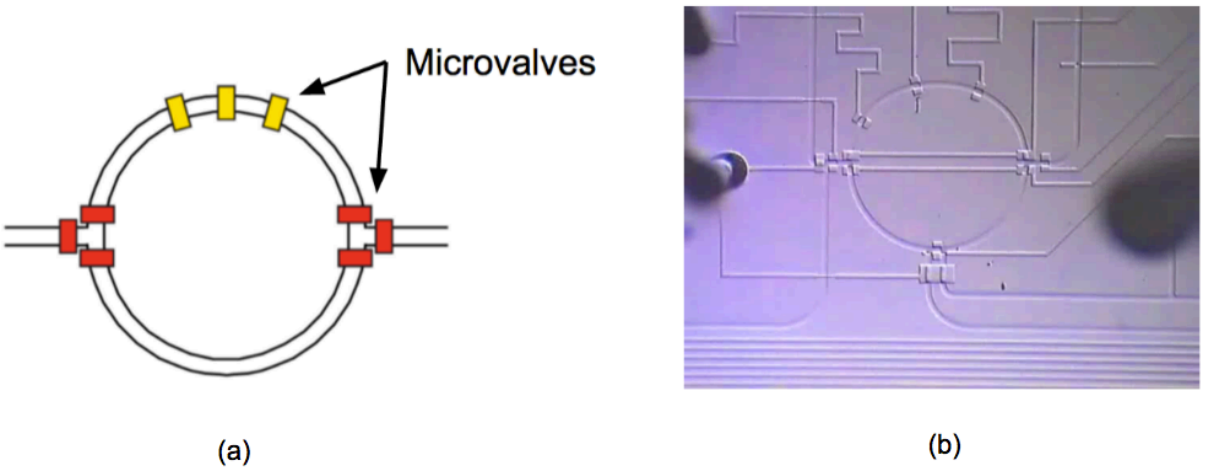


Figure 4

1.4. Valve structure

Valves are one of the basic components in microfluidic biochips, and are the fundamental unit that controls the flow of liquid inside the channels. In the first designs for biochips, silicon microelectromechanical systems technology was used for implementing valves [2]. The structure of valves and the technology of valve implementation have changed over time. Since the creation of the first microfluidic chips, several methods have been introduced for implementation of valves. Today valves and biochips are manufactured multilayer soft lithography [2]. Figure 5 from [8] shows the structure of these valves, which consists of two layers: flow layer (shown in blue in the picture) and control layer (shown in red in the picture).

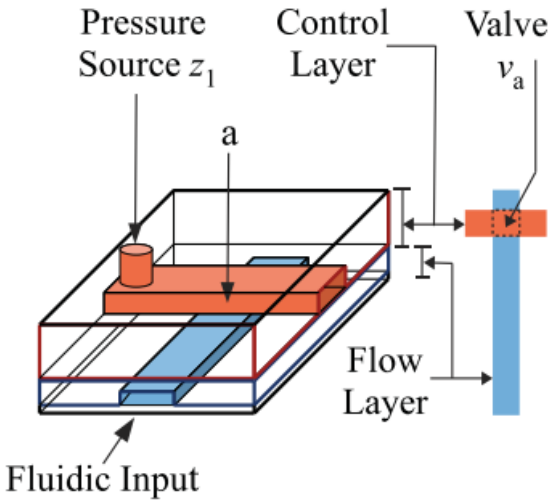


Figure 5

The flow layer is connected to the channels for flow of fluid. The control layer is connected to air pressure source with air channels. The two channels are separated with a flexible membrane, which can stretch towards the flow channel. Figure 6 from [2] shows the side cut of the channel. Position of the flexible membrane can be seen in this figure. When the air pressure in control channel is high, the flexible membrane will bend towards the flow channel and block the way for liquid to go through. This state is known as closed valve. When the air pressure source is inactive, valve is considered to be open and the fluid can flow through freely [8].

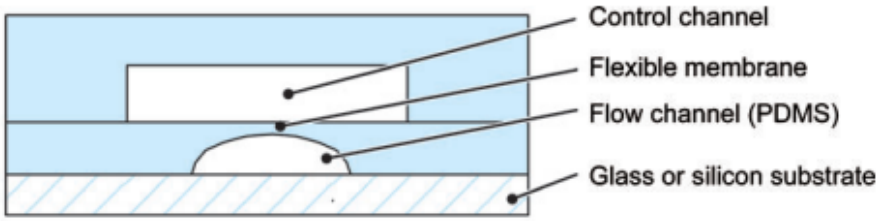


Figure 6

Different modules such as switch and multiplexer, can be implemented by putting valves together. As it was discussed in the previous section, mixer can be implemented by combining valves as well.

1.5. Fabrication

Because of the development of electronic chip industry and advancement of semiconductor fabrication process, early lab-on-a-chips used to be developed using silicon processing. But this process could not fulfill all the requirements for a biochip. The cost would rise more than it could be affordable for a single assay to perform. The chip was not compatible with some chemicals or bio organism. Also special optical characteristics could not be implemented in this process. Therefore new processes were developed for fabrication of biochips. Today's chips are mostly fabricated by photolithography, and with simple method that allow creation of channels and implementation of other biochip characteristics.

1.6. Computer Aided Design

During the recent years, much attention has been drawn to biochips and their advantages. Small area, fast and reliable response, safety, and low cost, are only a few of their benefits, which has convinced biologists to tend to use these chips. Emerging applications for biochips require further advancement in their structure and design. Performing large assay, as well as multiple and concurrent assays on a chip results in increasing complexity of biochips. On the other hand, time to market and fault tolerance of these chips is a critical issue. Therefore full-custom design techniques cannot keep up well with this growing scale, and the support of advanced Computer Aided Design (CAD) tools is becoming more important. Using a well-developed CAD tool in design process can decrease the time to market significantly, and increase the reliability of the design. This can also help in the way of integrating microfluidic biochips with micro electronic components in the next generation.

While CAD tools for biochip design are still in the early stage of developments, very advanced CAD tools are already used in semiconductor technologies. By modeling microfluidic biochips, many similarities can be found between these chips and electronic chips conceptually. As an example, valves in flow-based biochips are conceptually similar to switches in electronics, and flow channels are similar to wires and connections on the board. Therefore many of the methods and approaches that are already being used for designing electronic chips, can also be modified and applied for biochip design. This work is an effort to advance Computer Aided Design methods, for solving one of the existing problems of design process.

2 Problem formulation

Although the process of biochip design has been improved significantly, there still exist problems that are unsolved or neglected in the research. In this section, some of these problems will be addressed. Specially the main focus of this research, and the purpose of this work will be discussed.

2.1. Valve actuation

The actuation of valves in a mixer was introduced in the first chapter, as a key act for mixer's functionality. As it was shown, valves control the flow of liquid inside and through the mixer by actuating in the right order.

One of the problems that is often neglected during biochip design, is the fact that a valve can only be actuated a limited number of times (usually a few thousand times) before it wears out and cannot be used anymore or starts malfunctioning [9]. A broken valve can even impact on the result of the assay. By the growing scale of assays that are carried out on biochips, the number of actuations for a valve will soon become a critical parameter in design, if not already.

When designing a chip to carry out an assay, or when programming a chip to do so, sometimes a few valves are used more often that the others. These valves become the critical ones in the design when the number of their actuations increases. If one valve is used too much and is broken, the whole functionality of the chip can be affected regardless of the number of times that the other valves have worked. This means a chip is useless if only one valve (which is still needed in the function) is broken, even if the rest of the valves are only used a few times. Such design is obviously very inefficient. This is also the case in conventional mixers. Figure 7 (a) shows the number of valve actuations required to perform one mixing operation [3]. As it can be seen in this figure, pump valves are actuated 40 times after the first mixing operation, while the control valves are only actuated 2 to 4 times for bringing the liquid inside and taking the result outside [10]. The difference between the number of valve actuations increases dramatically with the next mixing operations that are carried out on the mixer. Figure 7 (b) shows the number of valve actuations after the second mixing operation. It can be seen that the pump valves are actuated 80 times while the control valves are actuated 4 to 8 times. This means the pump valves are actuated about 36 times more than the control valves for each mixing operation. The mixer as a result, will be out of order and useless as soon as the pump valves are actuated for the maximum allowed number of times, even though the control valves have been actuated much less and can still be used.

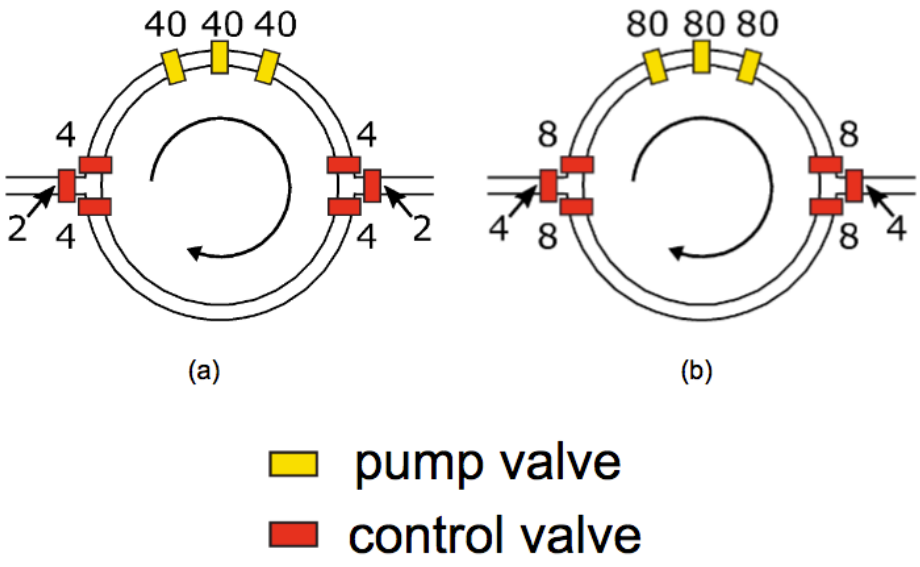


Figure 7

The problem described in this section is often neglected in the design process. In the following sections some solutions will be introduced that are a result of some previous research.

2.2. Previous work

It was described in the previous section that the imbalanced number of actuations for different valves in a design can cause early damage to the chip and take it out of order, while it could potentially be still in use. This issue is caused by inefficient design. In this section, a new approach that was introduced by [3] from Electronic Design Automation (EDA) department of Technische Universität München (TUM) will be described.

2.2.1. Valve role changing

Inside a conventional mixer, the difference between the number of actuations of a pump valve and a control valve does not let the mixer be used efficiently and up to its true potential. But it is not necessary to group the valves in this way and split their responsibility so unequally. As it was mentioned in the first chapter, all the valves inside a mixer have the same physical structure, and therefore they are able to perform the same tasks. But in fact some are assigned to only control the flow of liquid through the mixer, which needs a few numbers of actuations, while the others are assigned to circulating the fluid, which required significantly larger number of actuations. If there was a way to actuate the valves in a mixer more balanced, the potential of the mixer could be utilized well. This was the idea for the work done at the EDA department of TUM.

In this work, the concept of valve's role changing is proposed. It suggests that the role of the valve does not necessarily need to be fixed all the time. Since the physical structures of the

valves are the same, they can carry out each other's responsibilities at different points of time throughout the assay execution. This can be done if the valves are placed at the topologically same locations, allowing them to do the same job.
From the 9 valves that are used in the structure of a mixer, 7 are located at the round pipe. This means these 7 valves can block the way of liquid through the pipe. But only 3 of these 7 are used for circulation of the fluid. Figure 7 shows that the other valves can also play the same role and perform fluid circulation. Knowing this, the circulation of the fluid can be once assigned to one group of these valves, and the other time to another. Meaning for the first mixing operation, valves in group A act as pump valves and perform the circulation, and for the second mixing operation, valves from group B do the job. It can be seen in Figure 8 that by doing so, the actuations of the valves are more balanced. The actuation of the pump valves are almost halved over time, and the actuation of the other group of valves is increased to equate. Therefore the lifetime of the mixer can almost increase to double.

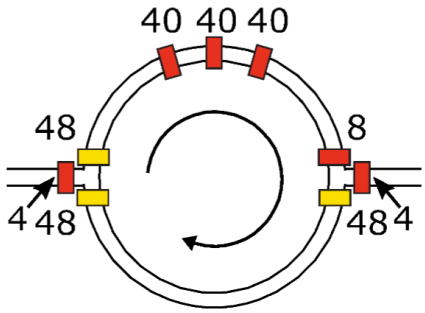


Figure 8

2.2.2. Square mixers and Valve array structure

The idea of valve matrix was first proposed and manufactured by [9]. In this architecture, valves are arranged in rows and columns, forming the shape of a matrix. Flow channels and other components of the chip are basically constructed using these valves. A group of values that are located regularly in the same row can construct a straight flow channel if they are all open to let the fluid go through. By programming these valves to actuate at certain points of time, transportation of the liquid can be controlled. This is very similar to the matrix of switches in an electronic chip. If all the switches in one row are in their ON state, electrical current can pass through the switches.
In the work represented in [3], valve matrix is transformed to valve-centered architecture. In this architecture, virtual valves are assumed to be arranged regularly in shape of a matrix. These valves are used to construct different components dynamically. This means that for executing any operation, suitable module will dynamically be generated at any position on the chip at that point of time. Some of the valves in the matrix work as boundaries of this module, to prevent the liquid from leaving the module. These valves are kept closed during the execution of an operation. After the execution is completed, valves must actuate in the right order to transport

the liquid to its next destination, and the module will be automatically removed. This area of the chip (or this group of valves) can be used later for any other purpose, provided that it has been cleaned from contamination. By observing dynamically generated modules on the chip, it can be seen that they might have been generated in the same area, but in different stages during the performance of the essay. In order to decrease the distance between modules and reduce the time of execution, the next module that the target liquid must go to, can be generated at the same position. This will help avoiding unnecessary transportation of liquid inside the chip. Therefore it can be seen that some modules may overlap at some point of time. By using this architecture, the number of actuations for all the valves can be controlled and the generation of modules and channels can be done with awareness of the number of actuations, and in a way to reduce the maximum number of actuations and balance the actuation of different valves in the matrix. After all the scheduling and routing is completed for an essay, unused virtual valves can be removed from the design. The final chip can then be fabricated and used.
Figure 9 from [3] shows a 4×4 valve centered architecture, and a mixer generated in this matrix. As in can be seen in the figure, wall valves (marked in yellow) are dedicated to create the borders of the mixer, and pump valves (marked in red) circulate the liquid inside the module by actuating in the right order.

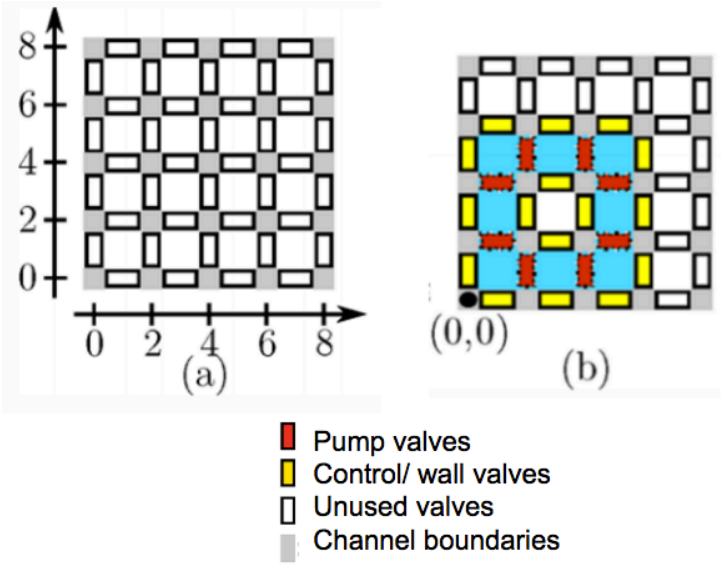


Figure 9

2.3. Problem with square mixers

Using a group of valves as a mixer has many advantages. It increases the flexibility of the chip, reduces the size of the chip, and has a significant impact on the balance of valve actuation among the valves. But it also raises problems, which are not solved yet. The first problem is during the execution of the mixing operation. Due to the square structure, new mixers have 4 sharp corners. Liquid trapped in those areas does not circulate with the same velocity, and does

not get mixed well with the rest of the sample inside the square mixer. Therefore, some substance might have non-uniform concentration in the final mixture. This theory has increased the sensitivity and decreased the trust of biologists in using square mixers.

Another issue regarding this method is the cleaning of the area of the mixer. After the mixing of operation is fully executed inside a dynamically generated mixer, the result should be carried out and the area of that mixer must be cleared of any residual before that area can be used for any other purpose. If the channels are not cleaned properly and some of the liquid from the mixing operation remains in the channel, it might be dissolved into the next liquid passing through that channel and can contaminate the liquid. For example during a blood test, amount of a substance inside the sample might be reported higher than its true value, due to the mixture of the sample with the remaining droplet from the previous sample. This problem is very well known and the solution for cleaning the channels and the modules on a chip has been introduced as washing method. In this method, a special liquid known as wash liquid is inserted to the chip and moved around in the target channel or module. This liquid dissolves any residual of previous operations into itself and carries the contamination outside of the chip. But in a square mixer, the contamination trapped in the sharp corners might not mix into the wash liquid and will remain untouched in the channel after the cleaning, decreasing the reliability of the result. The out of reach area in the corners are depicted in Figure 10 (a). It can be seen in picture (b) in this figure, how contamination remains in the corners after washing. The third picture shows how this contamination can be mixed with the next sample and cause changes in its content.

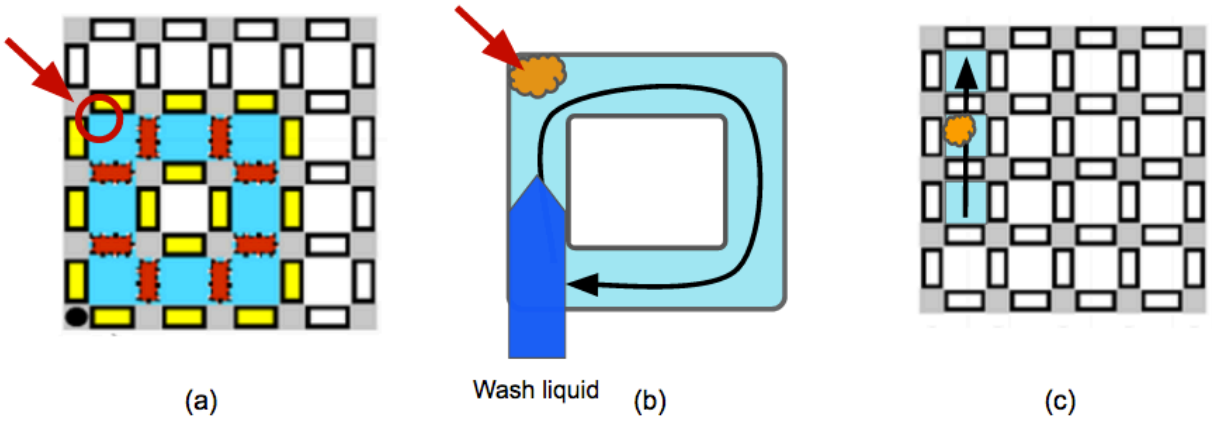


Figure 10

3 Scheduling with valve array and conventional mixers

3.1. Idea

In the previous section, it was explained how increasing number of valve actuations can affect the reliability of a design. Previous work done for solving this problem was also described in that section. It has been shown how valve's role changing technique can make a balance in actuations of valves, and how dynamically generated modules and storage units can increase the reliability and flexibility of the design. Benefits of using valve centered architecture were discussed in detail. Advantages of conventional mixers were also covered. Considering all these advantages and disadvantages, the question is if an approach can be introduced to combine these architectures and benefit from the advantages of both?

The idea of this work is to find a combination of valve centered architecture and conventional mixers, that meets the requirements and constraints. This architecture would ideally use conventional mixers for executing mixing operation, in order to provide a reliable infrastructure for mixing, and washing afterwards. This architecture should also take advantage of valve matrix in routing and storage generation.

The suggested architecture in this work is a combination of valve matrix and conventional mixers. Figure 11 shows a model of the layout of this architecture. The valve matrix in this design plays the role of a router between the modules. Paths can be generated through this matrix by opening and closing specific valves. Mixers are arranged in two rows around the mixer, and the other two sides of the rectangle are connected to input ports (reservoirs) and output ports (where final result is received) of the chip. This regular design makes it flexible to be connected to other circuits as well. The reason for arranging valves in regular structure is to provide a flexible space for paths and storages to be generated, by keeping all the valves connected to each other. It also makes it independent of the connected modules, so the size and functionality of the matrix stays intact if the connected modules change. Therefore the design is easy to expand.

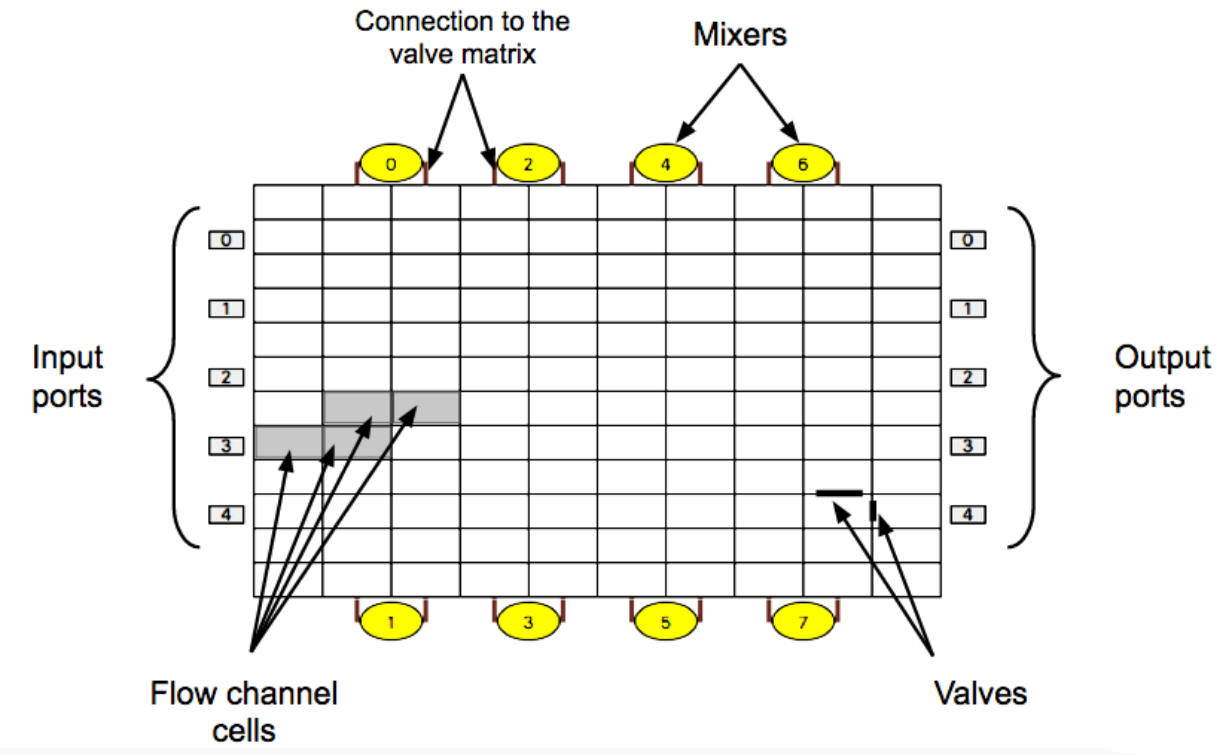


Figure 11

Besides using a matrix of valves to have the ability of generating paths and storages dynamically, and guarantee functionality by using conventional mixers, the goal that is tried to be achieved in this work is to minimize the cost of the design, by minimizing the execution time of the assay and size. In the next section, methods used for minimizing the execution time, cost calculation and optimization are discussed.

3.2. Algorithm

The algorithm that is developed to carry out the idea of this work is described in this section. Required inputs, provided output, and strategies for every stage are explained in details.

3.2.1. Inputs

As it was said before, the goal of the algorithm is to plan an assay to obtain a solution fulfilling the constraints. The first input that is required to start the work on, is the assay itself. The sequencing graph model together with table of operations details, provide sufficient information about the assay in this case. The sequencing graph contains information about operations, such as inputs and outputs, as well as the connection between operations. The detail for each operation like the volume of inputs and duration of the operation can be gained from the table. Therefore this information is given to the algorithm as input, by reading from a text file. Details about the format of these input files can be found in section 3.3 and also in appendix II.

The other important information that the algorithm needs to work with is the constraints and rules that must be applied and obeyed during the planning. The necessary information that is required in this work are the constraints about size of the valve matrix (meaning the number of valves allowed to be used in the design), number of mixers allowed to be used, maximum time that is allowed for performing the assay, maximum number of actuations allowed for each valve, and the cost of each element of the design. The latter is required for calculating the cost of the design based on the technology and application. The important elements affecting the cost of the design that are concerned here are time of the performance of the assay, number of mixers, and size of the valve matrix. The cost for each of these elements may vary by the technology and purpose of building the chip. In some cases time might be the critical parameter in design, while in other cases the size of the chip is more important. The effect of each element in cost has been discussed more into detail in section 3.2.5. Since the value of each element is not constant for different applications, it can be given to the program as an input. This provides more flexibility, because the cost is not modeled with fixed parameters. It also provides the option of comparison since by changing this values different results can be obtained. Information about the constraints and limits are provided in another text file as input for the program. The format of this file and some samples can be found in section 3.3 and appendix II.

3.2.2. Scheduling

Given the inputs that were discussed in the previous section, and using the architecture introduces earlier, this high-level synthesis focuses on scheduling assay operations under resource constraint, in order to maximize parallelism and decrease execution time. This problem can be formulated as follows: given a sequencing graph model of an assay, the architecture of the chip including characterization of modules, and also a set of constraints, a solution (determining the binding of resources to operation, binding of the edges to dynamically generated flow channels, and the timing details for all operations), is to be provided to minimize the time of execution for the assay, while satisfying the dependency, resource constraints, valve actuation constraint, and also routing constraints. This problem can be considered equivalent to resource constrained scheduling with non-uniform weights, which is NP-complete [11], [12].
Constraint programming approach can provide a solution for such problem. But due to the increasing complexity of biochip architectures and biomedical assays, a constraint programing approach becomes computationally infeasible. Therefore for solving this problem, a heuristic method has been used in this work. The objective of heuristic approach is to employ human intelligence, experience, and common sense, and apply certain rules, to provide a solution that is good enough for solving the problem, in a reasonable time frame and with reasonable effort. The produced solution might not be the best solution, but the effort is to make it close to the optimum, and it is valuable since finding the solution did not require a prohibitively long time [13]. Thus, the heuristic approach is generally used when deterministic techniques are not available, economical, or practical. Since the execution time for linear optimization will not be

very long for large assays, and also because heuristic method can provide acceptable solution, a heuristic method is used in this work for solving the problem.

To schedule operations of the assay and assigning resources for executing these operations, a modified list scheduling algorithm is used in this work. In this method, given certain architecture for the chip, the effort is made to minimize the execution time of the assay. To be able to follow what happens to each operation during the scheduling phase, it can be assumed that each operation goes through a set of states during execution. State of operation is changed by occurrence of certain events. Each state shows in which stage of scheduling is the operation, and what is happening to it at the time. Figure 12 shows the state diagram for one operation. In this algorithm, each operation is looked at independently and in parallel. At the beginning of the scheduling, all operations are at Start state. The state of an operation changes to Ready when its inputs are ready to be used. If the inputs are supposed to be received from input ports, then the input is ready when the port is ready. If the input of the operation is a product of another operation, then the successor operation must wait until the previous operation is fully executed and the product is ready to be used. Only then the state of the second operation changes to Ready. In this state, algorithm tries to find the most suitable mixer available at the moment for this operation. This is done using a method explained in the next section. Depending on the assigned mixer, operation can go to two different states. If the assigned mixers already contains one of the inputs, meaning that input was supposed to come from the result of another operation, and the mixers executing that operation is assigned to this new operation, the state will change to Reserve. In this state, operation is kept on hold until the rest of the product leaves the mixer, and only the liquid required for this operation remains in the mixer. If this is not the case, meaning both inputs must be transferred to the assigned mixer, state of operation changes to Active 1 if a path is found for transporting one the inputs. If no mixer is available at the moment, operation stays at the Ready state until one is found. When the operation is at reserved state, meaning the mixer is reserved for this operation, its state changes to Active 1 when the mixer is free and ready to be used. When going to Active 1, mixer is assigned to the operation. The path for one input is also assigned if needed at this stage. Since the mixer has only one input port, inputs can only enter the mixer one by one. Therefore when one input is entering the mixer, the second input must be kept at some other location. This can be at the source (mixer or source input ports), or in a storage unit. For reducing the execution time, second input is transported and stored in the nearest place to the mixer while the first input is being transported into the mixer. And operation stays in the Active 1 state. Storage generation is done by a method described in section 3.2.4. By moving the liquid close to the mixer, transportation time is reduced, comparing to the situation where the liquid is kept at the source until the first input is fully received, and then the transportation starts. When the first input has entered the mixer successfully, and a free path for transporting the second input is found, this path is assigned to the input and the state of operation changes to Active 2. Execution of the operation inside the mixer starts right after the second input is received at the mixer's input port. Duration of execution is specified in the description of operation, provided at the beginning of scheduling

from the input file (see section 3.2.1). After the execution time is over and mixing operation is successfully executed, state of the operation changes to Finished. At this state, mixing has been done and the result is ready, and the successor operations (those that need the result of current operation as input) are notified that their input is ready, but the result has not left the mixer yet. When the next operation is ready to take a portion of the result as its input, and the rest of the result is transported out of the mixer towards output ports of the chip (to be discarded as waste, or to be delivered as a final result), state of the operation changes to Removed. This operation is now fully executed and has been removed from the list of operations that are being scheduled. Mixer will be marked as free at this point. Figure 12 only shows the states and the events that change the state. Actions taken at each transition and in each state are not shown on the diagram.

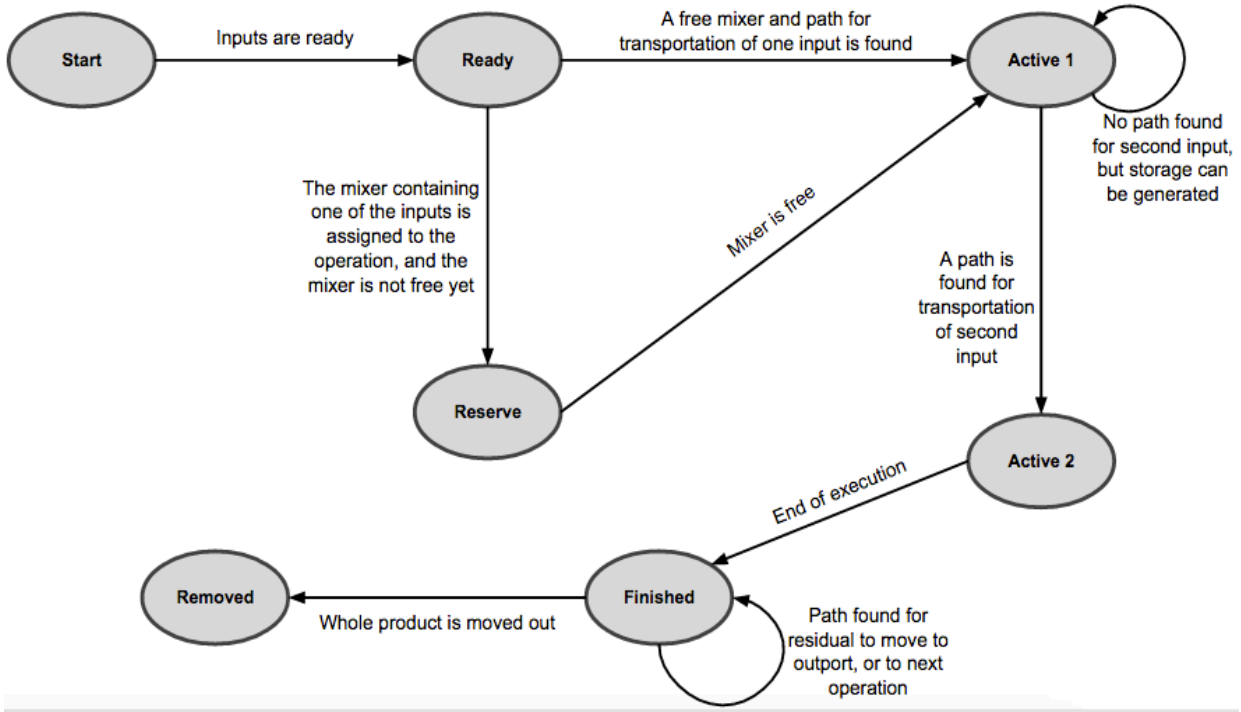


Figure 12

3.2.3. Mixer assignment

As it was mentioned before, in this method, operations are picked by priority and scheduled one by one. The first step in scheduling an operation is finding a suitable mixer module for this operation. Two criterions must be considered when choosing the right mixer. First, the number of actuations for a valve in mixer structure must not exceed the allowed value. Number of actuations can be calculated from the number of times that the mixer was used. By using valve's role changing method presented in [3], the maximum number of actuations for one valve in

mixer after one operation is 44. This number increases to 48 after second time the mixer is used. When evaluating a mixer to assign to an operation, maximum number of actuations must be considered, and it should not step over the limitations. As an example, if the maximum allowed number of actuations for one valve provided by the constraint file is 100, then a mixer can only be used 4 times reliably. It is also important to choose the mixer in a way that the transportation to that mixer takes less time.

To guarantee the first constraint, namely number of actuation for valves in mixer structure, mixer usage must be checked before assigning the mixer to an operation. Mixers can be classified into three groups, regarding their usage. first group are those that have been used at least once but less than the limit. These mixers can be used one or more times (depending on the number of times they have been used so far). Second group of mixers are the ones that have not been used so far in the design. Choosing a mixer from this group is similar to adding a new mixer to the design. And the third group represents those that have been used to the limits, and cannot be used anymore. When a mixer is required to be assigned to execute an operation, mixers from the first group are the best candidates. They have already been used, meaning the cost of adding those mixers has already been added to the total cost. And they have not been used too much. Therefore they can be used once more reliably. Using these mixers minimizes the cost of the design and also guarantees the reliability of the mixers. Only if no suitable mixer among this group is found, then a new mixer should be added to the design. By choosing a mixer from the second group, the cost of a new mixer is added to the total cost. This can include the cost of integrating control channels and circuits, and increase in area. The mixers from the third group cannot be used any more. Therefore if no suitable mixer was found in the first and second group, execution of the operation cannot start at the moment, and must be postponed. Figure 13 shows a summary of this classification.

Group	Description
1	Mixer usage > 1 and Mixer usage < limit
2	Mixer usage = 0
3	Mixer usage > limit

Figure 13

In order to minimize the execution time of the assay, the shortest paths must be chosen, so the transportation time is minimized. To find a mixer for each operation, group of mixers are investigated with priority, and in each group, the free mixer (meaning it is not executing another operation at the moment, and the product of previous execution is totally removed from the mixer) with minimum distance to the sources is chosen to be assigned to the operation. This

minimum distance is equal to the sum of the distance to the first and second input source of the operation.
Following is an example of mixer assignment for getting a clearer picture of the process. Figure 14 shows the current state of a design. In this time step, mixers 2 is currently busy executing a mixing operation. Mixer 1 has been used to its limits and is not usable anymore. Mixers 0 and 3 are free to be used, and both are classified in the first group. The inputs of the operation must come from input ports 6 and 7. Distance of these ports from mixer 0 is equal to 9 and 10, and their distance from mixer 3 is equal to 6 and 7. Therefore the sum of the distances from mixer 0 is 19 and for mixer 3 is 13. Mixer 3 is chosen to be assigned to the next operation in order to minimize the time of execution by minimizing the time of transportation.

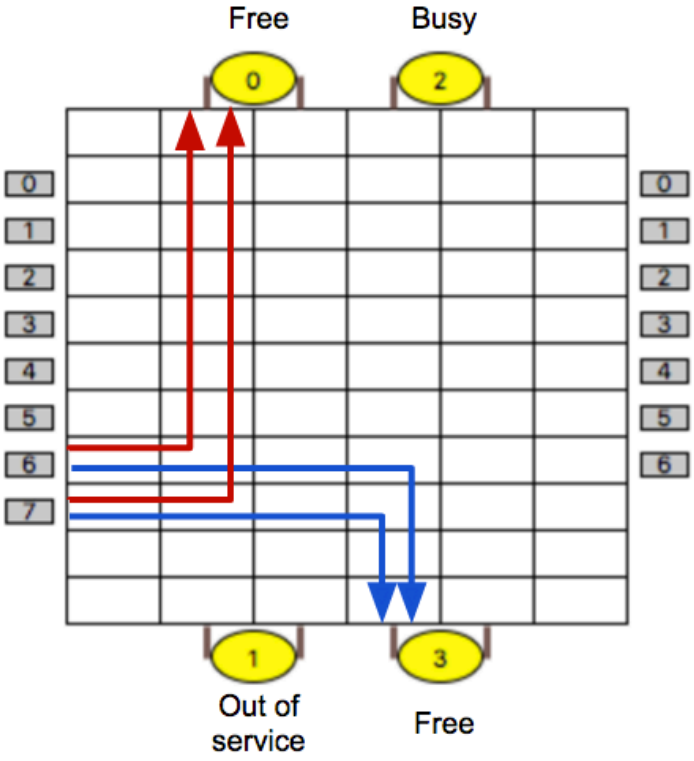


Figure 14

3.2.4. Routing and storage generation

Routing is one of the stages during which time minimization can be considered. In order to do so, at every point of time where a path is required to be generated, the short possible path is found and assigned for the task. This is done using the famous Lee algorithm [14]. Lee algorithm is a simple algorithm that guarantees that if a path exists between two points in matrix, it will find it. It also guarantees to find the shortest path between these points. Lee algorithm provides an approach for routing, based on a bitmap representation of data and a cost function, which can be modified to adopt the application. The drawback of this algorithm is that it is slow and requires large memory. Lee algorithm is used in this work since it provides the

shortest path which is important to reduce the execution time, and also because the runtime of the program and memory usage are not critical in this application. To apply Lee algorithm on the valve matrix, this matrix is seen as a matrix of cells. The space between four valves is considered as a cell, which can hold liquid inside it when the surrounding valves are closed. Figure 15 shows cells in a valve matrix.

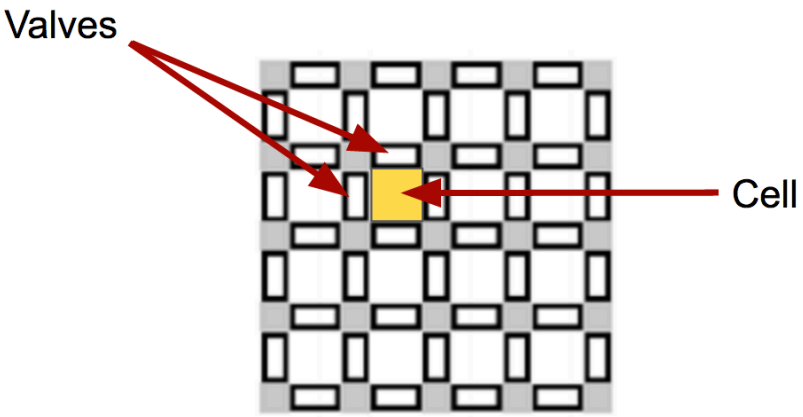


Figure 15

To use Lee algorithm in this work, it has been modified to adopt the application. In this method, one of the points is considered as starting point, and is labeled by 1. This point is the only point in the front line at the beginning. Next, all the adjacent point of the starting point that are not occupied, are marked with 2 and added to the front line points. At each step of the algorithm, every available adjacent point of every point in the front line is marked with a label, which is the label of the front line point plus one, if the point is free and not visited before. This procedure goes on until the destination point is reached, or no other point is left to explore. In the second case, no path was found.

Before starting this stage, all the occupied points are marked with -1. These points might be occupied by another path or a storage unit. Also the output port of mixers that are receiving input must be left free, because it is used for letting the air leave the mixer while the liquid enters. These points are marked with -1 as well. Cells marked by -1 are considered to be unusable during the routing of one path, and are simply ignored.

After the labeling stage, a backtracking stage is required to find the path using the labels. This stage is started from the point, which was the destination previously. This point is added to the path. Then from the adjacent points, the one with a label that is only 1 unit less than the label of current point is chosen, added to the path, and considered as the current point in the next step. This procedure goes on until the starting point is reached. Figure 16 shows the labeling and backtracking procedure in a simple example. The first figure shows how cells are labeled in the first stage. Notice that three of the cells are marked as unusable. They might represent an storage unit generated in the matrix. Figure 16 (b) and (c) show two of the possible paths that can be found by backtracking, but notice that the length of the paths are equal, therefore in this

example any of them can be chosen. In the modified algorithm used in this work though, the paths are chosen in a way that the path is horizontal near source inputs and output ports of the matrix, and vertical near mixers, if possible. The reason for this is to avoid blocking adjacent ports or mixers. This method resulted in shorter execution time when compared to other ways of choosing the path during the experiments. Therefore this strategy is used here.

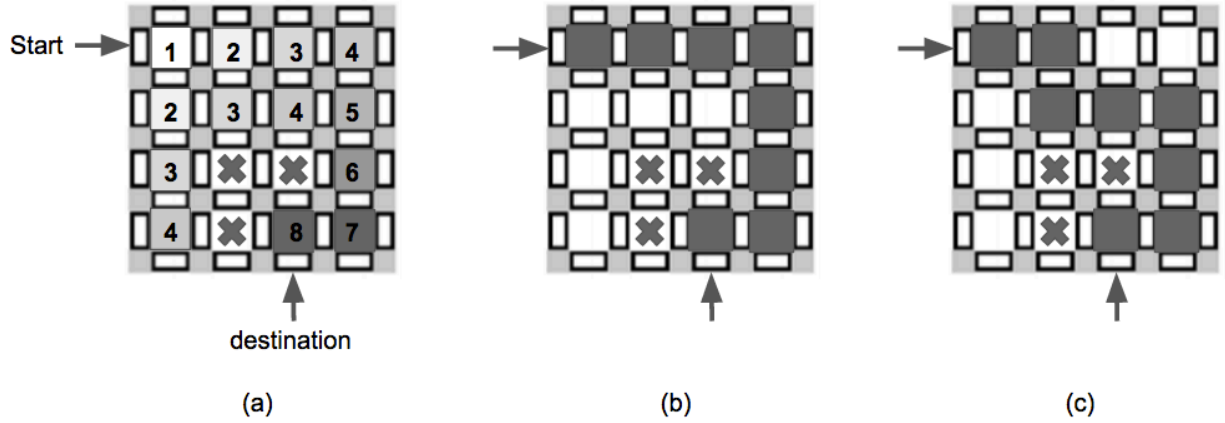


Figure 16

I must also be mentioned that the number of valve actuations is considered during the routing process. For finding each path, during the labeling and backtracking, the number of actuations for the valve connecting current point to the next point is checked. Next cell is only chosen if the valve's actuations are less than the limit. Therefore the maximum number of actuations for a valve is guaranteed to not exceed the constraint value. Figure 17 shows an example of routing with this algorithm. In this example, 4 paths are generated for 4 different liquid transportations. Paths can be identified with the label. 3 out the paths are routed from an input port to a mixer, and one is routed from an input port to a dynamically generated storage unit.

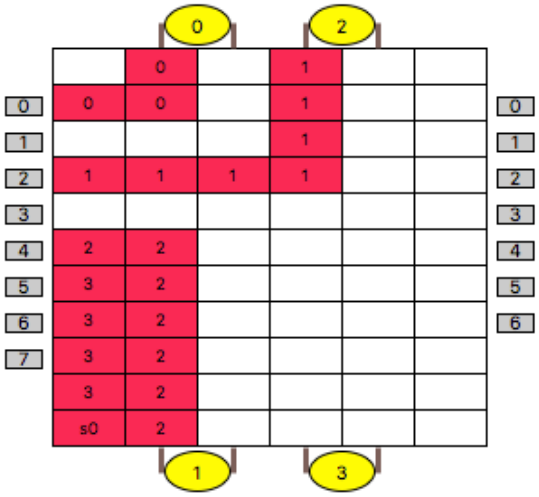


Figure 17

Not always a path is found for transporting liquid to a destination (mixer, or output port), because the path might be blocked by another path at the moment. In such cases, one approach is to wait until a free path is found. An alternative is to move the liquid to a place as close as possible to its destination. This would decrease the time of transportation significantly, since the by the time the path is free to the destination, the liquid has already traveled a part of the way instead of waiting at the source.

When a path is not found, algorithm tries to find a path between the source and another point close to the destination. This point is found by a greedy approach, which is finding the closest point to the destination on the shortest path from source to destination. This might not be the optimal solution, but it guarantees the best choice with the available information. Since in the heuristic algorithm only the information of the current step is available, decision has to be made upon that. One can argue that it might take long for path to be available between the storage and destination, while it might appear sooner from another point that was a few steps away and could be chosen for storage instead. But the purpose of this greedy algorithm is not to find the optimal solution, but is to find the local optimum based on the information provided, in a reasonable time. By breaking the problem to two steps, first finding the shortest path to the closest point to the destination, and second find the shortest path from that point to the destination, the greedy algorithm guarantees the local optimal solution, with purpose of reducing execution time. Since the operations are scheduled with priority given to the first operations in the sequencing graph, when the path is free between the storage and destination, it will be assigned to this transportation. Therefore this solution is close the global optimum solution in practice. This is also implemented using the same modified Lee algorithm that was used for routing and was introduced earlier in this section. When a path is found, storage is generated on this path. The reason for generating the storage in the shape of the path is to guarantee traveling to the destination through the shortest path. The size of this storage is equal to the number of cells that is required to store the liquid with specific volume. As an example, if the volume of the target liquid is 5 units, and each cell in the valve matrix can hold 1 unit of liquid inside itself, then 5 cells are required for storing the liquid, and the size of the dynamically generated storage unit is equal to 5 accordingly. Therefore storage can only be generated if the length of the path is long enough for keeping the liquid stored. Figure 18 shows three examples of dynamically generated storage units. Cells labeled with “s” represent storage cells, and arrows show the paths.

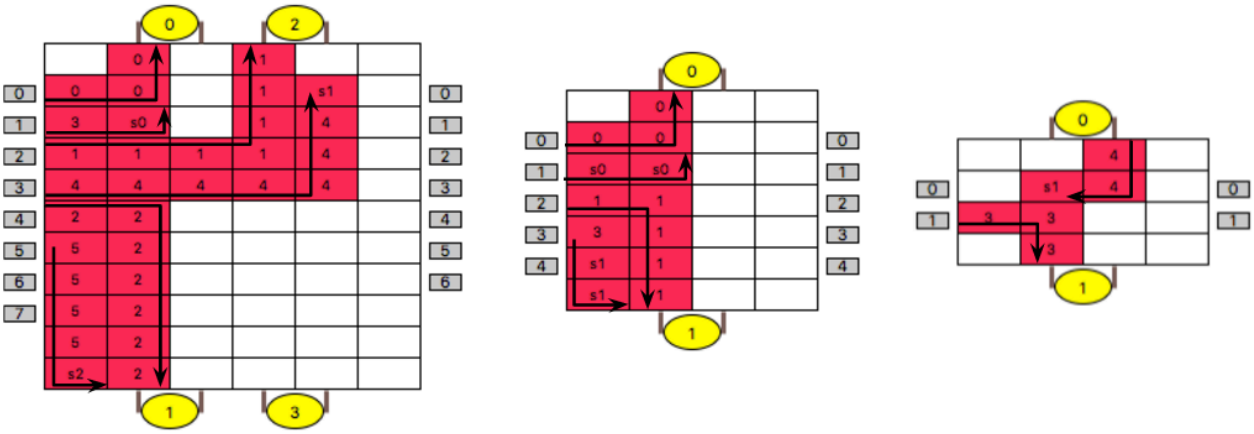


Figure 18

3.2.5. Optimization strategies

One of the goals of this work is to minimize the cost of the design while fulfilling the requirements and limitations. The first step for decreasing the cost of the design is to find a model for the cost calculation. The important elements affecting the cost of the design that are concerned here are time of the performance of the assay, number of mixers, and size of the valve matrix. Other parameters such as the material used in fabrication, can also affect the cost of the chip, but those parameters are not relevant to this stage of the design, and therefore not concerned.

The cost for each of relevant elements in cost may vary by the technology. As an instance, in one method adding control channels for one valve can be more expensive comparing to another method, therefore the size of the valve matrix will have a higher importance. On the other hand, if the cost of adding a new valve is less than actuating a valve for a certain number of times, then it is preferred to use more valve, that each actuate less, or to have more mixers, each being used more.

The application and purpose of building the chip can also have an impact on the value of each element in cost calculation. As an example, if the input samples for an assay are very sensitive to the condition and their characteristics can change rapidly when used, then the assay must be carried out in the shortest time possible. This means that time has a higher value in this case, than the number of valves. In another case, it can happen that the size is critical for the application. The chip must be as tiny as possible to be used in special conditions. Size of the matrix and number of mixers has a higher value in such case.

Cost of the design can be modeled as a weighted summation of the parameters. Weighted sum model is probably the best known and simplest model for evaluating decisions in a multi-criteria decision analysis [15]. In this model, for every criterion, a relative weight of importance is defined. Each alternative decision is obtained by summation of the product of these assigned

weights with the value of the criterion. A model with n criterion and m alternative values, can be defined as bellow:

$$Alternative_i = \sum_{j=0}^{n} w_i * a_{ji} \quad , for\ i = 0, 1, 2, \ldots, m$$

By using this model, the cost of the design can be modeled as follows:

$$Cost = w_{time} * Time + w_{mixer} * (Number\ of\ mixers) + w_{valve} * (Size\ of\ valve\ matrix)$$

Where time denotes the duration of the assay, number of mixers denotes the number of mixers used in the design, and size of valve matrix denotes the number of valves used in the design. w_{time}, w_{mixer}, and w_{valve}, which are the cost of each unit of the parameters, meaning weight of importance for each parameter, are given as inputs.

After the cost is calculated using the model, the minimum cost that can be obtained using this algorithm can be found by iteration over sensible values for parameters and finding all the alternatives. For this purpose, the optimization algorithm starts by scheduling and routing using the maximum allowed number of mixers, rows, and columns given as input, meaning the constraints. The cost of the design is calculated with regards to the obtained values for parameters, namely the duration of the assay, number of rows and columns used in the matrix, and number of used mixers. Next, the constraint for the number of mixers is reduced by one and the procedure is done all over again. For each number of mixers, different number of rows and column are examined as well. These numbers are decreased until the duration of the assay or the number of actuations for one valve exceeds the constraints. At the end of this progress, a table of cost regarding to the number of mixers and size of matrix is obtained. The minimum cost in this table, is the minimum that could be achieved using the heuristic algorithm with the given constraints. This cost, together with the number of used mixers, size of the valve matrix, and duration of the assay is reported to the user at the end.

3.3. Program description

In this work, the presented algorithm has been implemented using Python. The full script can be found in appendix I. For implementing this algorithm, some data structures have been used and some classes were defined that are fully described in this section. Information about input and output files can also be found in this part. This information not only help understanding the implementation better, but also makes the program easy to use and expand for the further research on this topic.

3.3.1. Classes

For describing elements of the design, a few classes have been built and used in this work. The “Operation” class is used to describe an operation in the program. An object of this class has a

name, which is of type integer and shows the number of the operation in the sequencing graph. “input1”, “input2” together with “in1type” and “in2type”, determine the type and number of the sources for this operation. The type of an input in this case can be another operation, an input source, or a storage unit. Attributed “in1ready” and “in2ready” show if the sources are ready or not. The volume of the liquid that needs to be provided by each of the sources can be found in “volume1” and “volume2”. Attributes start, duration, and end appoint the timing of the operation. As described before, “start” shows the soonest time that the operation can be executed. But the actual start time of the execution depends on the inputs and status of the mixers and valve matrix. The “end” attribute reports the time at which the operation is fully executed and ready to pass the result. The attribute “assignedMixer” contains the index of the mixer that is assigned to execute this operation. And the last attribute, which is “destinations”, shows all the destinations for the liquid in this mixer.

Next class that will be described is the “Mixer” class. An object of this class has the attributes “busy”, showing if the mixer is busy at the time, “currentOp”, showing the name of the operation that is currently being executed inside this mixer, and “ops”, containing a list of all operations that have been assigned to this mixer so far. The start and finish time of executing the current operation can be found using attributes “start” and “end”. The last two attributes are two integers named x and y, that represent the position of the mixer in the valve matrix. Value of x shows if the mixer is connected at the first or last row of the matrix, and value of y shows to which column is the input port of this mixer connected. Attribute “yo” also shows the position of the output port of the mixer. If the mixer is reserved for some operation during the run time, the attribute “reserved” would hold a Boolean value of True. The number of times that this mixer was used can be derived from the used attribute. It should be mentioned that an optional attribute named “typ” has been introduced in this class. This attribute can be used to identify the type of the mixer if various types of mixer were used in the architecture, so that each operation is assigned to a mixer of the right type.

The third class implemented in this program, is to represent dynamically generated storage units. This which is named “storage”, holds the value of x and y, representing the position of the storage unit on the valve matrix, in an attribute called “head”. Attribute “path” contains a list of all the cells of the matrix that are part of this storage unit. It basically shows in what shape and where this storage is generated and used. Attribute “v” shows the volume of the liquid in this storage unit. Attribute “sourceOp” shows from which operation the liquid has come to be stored in this storage, and attribute “destOp” shows for which operation this liquid is going to be used. “start” and “end” show the time of generation and termination of the storage unit. And finally “readyTime” is the time when the liquid has fully received at the storage unit and is ready to be transported to the destination from this point.

3.3.2. Functions

To provide flexibility and reusability and also simplification of the program, it is arranged modular. In other words, each functionality has been implemented in a separate function. By

combining these functions, the whole algorithm can be implemented. Modular implementation makes the program easy to use later, and easy to read and debug. In this section, important functions are introduced and described briefly. More information about the functions can be found in appendices.

readOpFile, readConstraints:
These two functions are used to read the information about the operations, and constraints from text files. The only required input for these functions is the path of the file. First function, arranges operations (from type operation) in a list. Second function returns the constraints in order.

init:
This function is implemented for initialization of the arrays, matrices, and lists. By getting the number of rows, columns, mixers, sources, and output ports, this function creates the matrix of valves, locates sources, output ports and mixers at the right location on the matrix, initialized the list of mixers, storages, paths, and ports.

findMixer:
This function can be used for finding the most suitable mixer available at the current time step during the scheduling. It is the implementation of the method described in section 3.2.3.

findAPath:
This function is used to check if there exists any free path between source and destination, which are inputs of this function. If a path exists, this function returns the shortest path between these two points. Otherwise it returns False value. The complete algorithm for this function is explained in section 3.2.4.

assignPath:
When a path is found successfully, and other conditions are considered as well, it should be dedicated to the transportation of the target liquid for a specific amount of time. This function marks the cells on the matrix as blocked for a certain period. It also marks the valves that must be actuated for this purpose.

store:
When the path between two points is blocked for some reason, the liquid can be transported and stored near the destination using this function. It gets the location of source and destination, and the volume of liquid as inputs, and tries to generate a storage unit near destination for storing the liquid. It returns the position of the storage and the path for reaching the storage unit from source, if the storage unit was generated successfully. If no suitable location is found for storage, it returns a False value. Complete method for storage is described in section 3.2.4.

Plan:
The whole scheduling algorithm is implemented in this function. It basically controls the state of each operation according to the state diagram, and perform changes at each time step according to the state of operations, mixers, and cells in valve matrix. This function assigns mixers to the operations by using "findMixer" function, and assigns paths for transporting liquid by using "findAPath" function, which have been described earlier.

detectUsed:
This function is implemented to find out which rows and columns of valve matrix are actually used after the scheduling is finished. Unused rows and columns are then removed from the design to save space.

drawPaths:
This function shows one or more specific paths in the matrix, as a picture. It can be used for debugging and understanding the process better. It gets the list of path indices as input, and depicts a graphical output.

snapShot:
By using this function, the current state of the cells can be depicted on a canvas. By putting the pictures from a sequence of points in time, changes can be seen much clear. It gets the desired time step as input, and shows graphical output. This function finds the cost of design for different configurations by iterating over the number of mixers and size of matrix, and scheduling operations using the "plan" function.

findMinCost:
The optimization method described in section 3.2.5 is implemented in this function. This function gets the path for operation and constraint files as input, and returns a report of the design with minimized cost. This report includes run-time of the program, number of mixers, rows and columns used in the design, execution time of the assay, maximum valve actuations, maximum usage of a mixer, and the cost of the design. The cost of the design and time of execution with different number of mixers, rows, and columns is written to a text file as well.

3.3.3. Used libraries

For implementing the algorithm, 4 special libraries have been used in this work, which will be briefly introduced in this section.
The first library used in the program is the Queue library. Queue has implemented the structure of the queue and required functions for using it, such as "put", "get", and "epmty". This library was used in generating queues during routing stage.

Second library that was used in this work is named copy. It has been used to safely copy and array into another array, and breaking their connection. If an array is copied without using this library, only the pointer to the array is copied into the new variable, and therefore any changes made to the copied array, impacts the original one as well.
Third library that is going to be introduced in this part is Tkinter. This library was used for drawing the chip and presenting graphical outputs.
The fourth and last library, is called time, and is used for computation of the run time of the program.

3.3.4. Input file format

The program gets two text files as inputs. The first file should contain the information about the assay, meaning the sequencing graph and information about the operations. In each line of this file, one operation of the sequencing graph is described. Name of the operation, its inputs, and name of the successor operations, are presented in this line. By using this information, the sequencing graph can be derived easily. Volume of the inputs, starting time of the operation, and duration of the operation are also included in this line. This information is necessary for scheduling of the assay. The header line of this file shows the order and format in which this informations must have been stored. This header is as follows:

name input1 input2 volume1 volume2 start duration destinations

The name of the operation must start with character ‘d’, followed by a number (e.g. d0, d1, ...). Inputs must clarify the type of the input and its number. The type of the input is shown by ‘s’ for source and with ‘d’ for operations. If the type of an input is ‘s’, then the liquid must be transported from a source to the assigned mixer. Otherwise it should be routed from the assigned mixer to the previous operation. The type of the input is followed by the number of that input (e.g. s1, d0, ...). The fields volume1 and volume2 show the volume of input1 and input2, which are integers. Starting time of the operation, is the first step of time in user wants the operation to be executed. During the scheduling, an operation may start at the same stage or a few stage later depending on the state of the mixers and valve matrix. In other words, an operation is executed at the soonest time after start time, that its inputs are ready, a free mixer is found and a free path is assigned to the inputs to reach the mixer. Duration of an operation indicates the number of time steps that it takes for the inputs to get mixed in the mixer. As an instance, if this number is 10, the liquid will circulate inside the mixer for 10 time steps. During this time, control valves of the mixer are closed, and the pump valves are actuated to move the liquid. And at last, the destinations field includes all the destinations that the liquid inside the mixer must go to afterwards. This would be the character ‘d’ followed by a number, if the destination is the next operation, and would be ‘o’ followed by a character if a portion of the liquid needs to be transported to the outports as waste or as a result. In this case, the volume of the liquid to be transported to the outports must also be indicated. Destinations are separated by

comma, and for outport destinations, the volume is indicated after a colon sign. As an example, the destination of an operation can be written as “d0,d1,o3:2” , which means a portion of the liquid will be taken for operation 0 and 1, and a portion with volume of 2 must be transported to outport 3. User must pay attention to empty a mixer from any waste liquid by adding an outport destination for the operation. This is very important since the liquid remains in the mixer otherwise. For example, if an operation is mixing a liquid with volume 4, and only a portion with volume 2 is used for the next operation, the rest of the liquid must be carried out to an outport to be discarded. The following line shows the information for an arbitrary operation in file. It represents operation number 5, which takes inputs from operation 3 and 4, with volumes 1 and 1. This operation can start at stage 1, and it takes 10 time steps for executing this operation in the mixer. The liquid will then be taken for operation 6, and a portion with volume 1 will be carried to outport 5.

d5 d3 d4 1 1 1 10 d6,o5:1

The second file that this program needs, is a test file containing information about the design constraints. These constraints include the limitations about the area, time, and modules. They describe the boundaries in which the chip can be designed. The header of this file is as follows:

valveMatrixRows valveMatrixColumns #mixers #sources #outports maxDuration maxActuations CostCoefficients(time, mixer, size)

The first and second field indicates the maximum size allowed for the final design of the chip. It shows the maximum number of rows and columns that can be used for the valve matrix in the design. By changing these values, simple the number of valves is changed in the design. The number of valves is an important factor in the cost of the chip. But reducing the number of valves may also result in more actuations for other valves. The third field indicates the maximum number of mixer modules that can be used in the design. This value also impacts the final cost of the chip. Higher number of mixers means higher number of valves and control circuit for them, and lower number of mixers can result in higher usage of one mixer. The next two fields indicate the number of source ports and outports, which are necessary in the design. maxDuration is the maximum allowed time for performance of the assay. During the scheduling and routing, number of time steps is not allowed to exceed this value. maxActuations represents the maximum allowed number of actuations for each one valve. Therefore in the design process, it is assured that the number of actuations for any single valve in the matrix or in the structure of mixers, does not exceed this limit. This is highly noticed during the mixer assignment and routing. The last three fields, named CostCoefficients, are used for calculating the cost of the design, and minimizing it. the y represent the weight of the value of number of time steps, number of mixers, and size in the overall cost. Basically, they represent the cost of each time

step, each mixer, and each valve in the matrix. The cost can then be calculated and minimized according to these values. An example of content in the constraint file is shown below:

valveMatrixRows valveMatrixColumns #mixers #sources #outports maxDuration maxActuations CostCoefficients(time, mixer, size)
20 12 8 9 7 1000 100 2 5 1

This line is interpreted in the program as follows. The valve matrix can have 20 rows and 12 columns at maximum. Not more than 8 mixers can be used in the design. The chip has 9 source inports and 7 outports. The execution time for the assay cannot exceed 1000 time steps, and in the final design, no valve should have been actuated more than 100 times. The cost for one time step is 2, for each mixer is 5, and for each single valve in the matrix is 1. The formula for the cost will therefore be as bellow:

$$cost = 2 * (time\ steps) + 5 * (number\ of\ mixers) + 1 * (size\ of\ the\ valve\ matrix)$$

Content of the operation and constraint files for test cases can be found in the appendix II.

3.3.5. Output format

The developed program presents different text outputs and graphical outputs. Some of these outputs are shown right after execution, and some are optional and can be requested by the user later. In this section, output content, their format and instructions to use them are provided.

3.3.5.1. Text output

After executing the program,some information regarding the execution time and design specification is provided. This information consists of run time of the program, which is dependant on the system running the program, and also the test case, provided constraints, cost formula, specifications for the optimal solution, meaning the solution fulfilling the requirements with the lowest cost. This specification includes number of mixers used in the design, size of the valve matrix, time steps required to carry out the assay on this platform, maximum number of actuations for one valve, maximum usage of a mixer, and finally the overall cost of the design according to the cost formula.
Besides this, user can use several other functions to achieve more detailed information about the design. Functions printOps and print platform provide information about the operations and the constraints respectively. By using the functions printMixers, printPaths, and printQInfo, detailed information about the assignment of operations to mixers, generated paths and their source and destinations, and the content of queues for handling operations at each point of time (which basically shows the state of the operations at each point of time) can be obtained.

3.3.5.2. Graphical output

Besides text outputs containing information of the design, a simple model of the design is also provided in this platform. The two functions drawPaths and snapShot provide this graphical output. By calling the first function, a image is drawn, depicting the valve matrix, mixers and their connection to the matrix, ports, and the paths for each transportation between the ports and mixers. By calling the second function, status of the chip (status of valve matrix) can be seen at an exact point of time. More information about these functions, and an example of the graphical outputs can be found in appendix IV. These functions can be used to trace the route for an exact input, or to trace the status of the matrix over a period of time. Using these images can give a better understanding of the design and algorithm to users.

4 Results

In this section, first the test cases used for experiments are introduced. After that, results are explained in details for a simple example for clarification. At the end of the section, results mentioned test cases are shown and discussed.

4.1. Introduce test cases

4.1.1. Polymer Chain Reaction (PCR)

Polymerase Chain Reaction (PCR) is an easy and low cost technique in molecular biology for DNA analysis. This technique is used for amplification of a particular DNA sequence, and generation of copies of that specific fragment [16]. One of the most important applications of PCR is in diagnosis and monitor of genetic diseases. It is also used in forensics to identify criminals.

PCR process consists of different stages. The mixing stage of this process has been used as a test case for this work. The sequencing graph of this stage of PCR can be seen in Figure 19 from [16]. This assay has 8 inputs, 7 operations and one output.

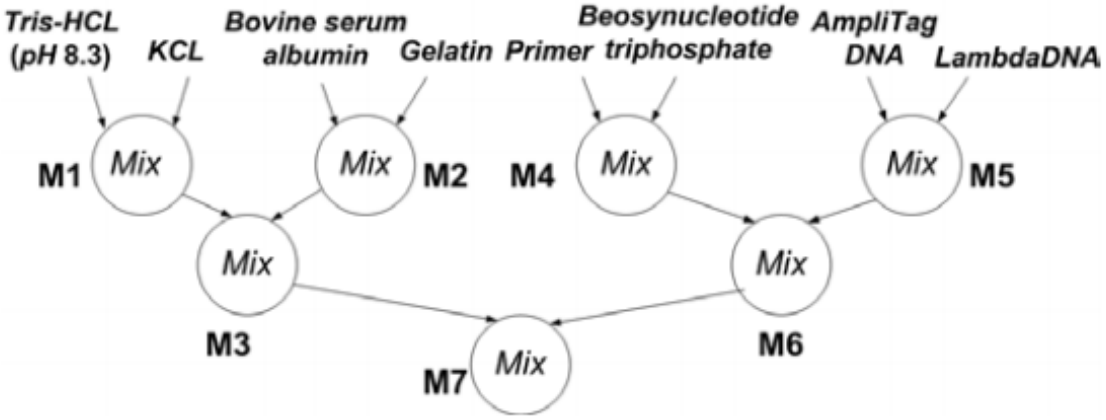


Figure 19

To adopt this sequencing graph to the program, it has been modeled as shown in Figure 20. Notice that inputs have been named with number for simplification, and an output has been added to each operation for removing unused liquid from the mixers after the operation is fully executed.

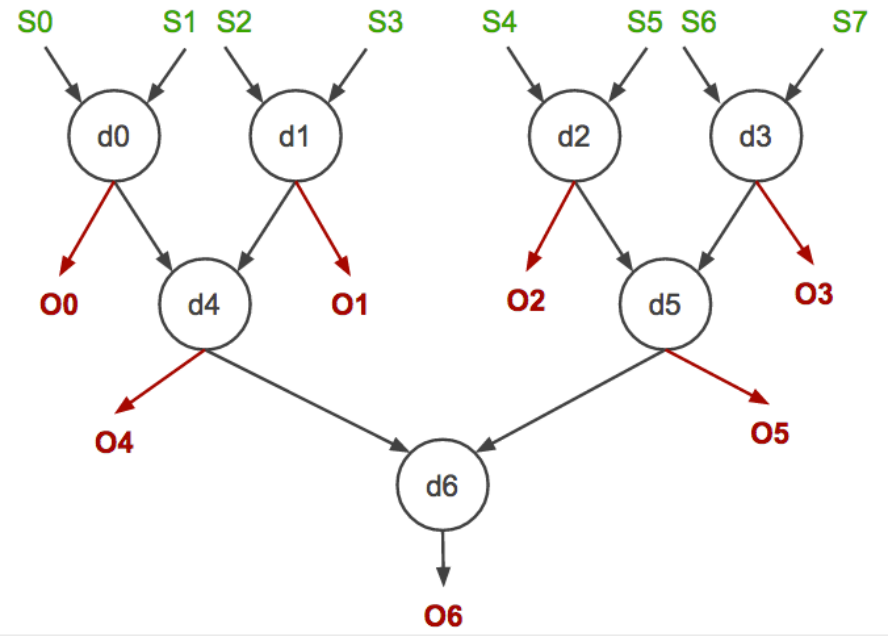


Figure 20

4.1.2. Multiplexed diagnostics

Measuring different metabolites plays an important role in clinical diagnosis. Irregularities in the metabolic parameters in the blood can be a symptom of or a sign for monitoring organ damage or dysfunction. As an instance, glucose can be detected in an enzymatic reaction based assay called glucose assay. This assay is performed in three steps: transportation, mixing, and detection. At the first stage, sample and reagent are transported to a mixer. In the second stage, sample and reagent are mixed and the enzymatic reaction happens. At the last stage, the product is transported to a detector for glucose concentration detection. In addition to glucose, lactate, and glutamate, are only two of the other examples of these metabolites. Figure 21 from [17] shows an example of a multiplexed diagnostic for measuring these metabolites in four different body fluids.

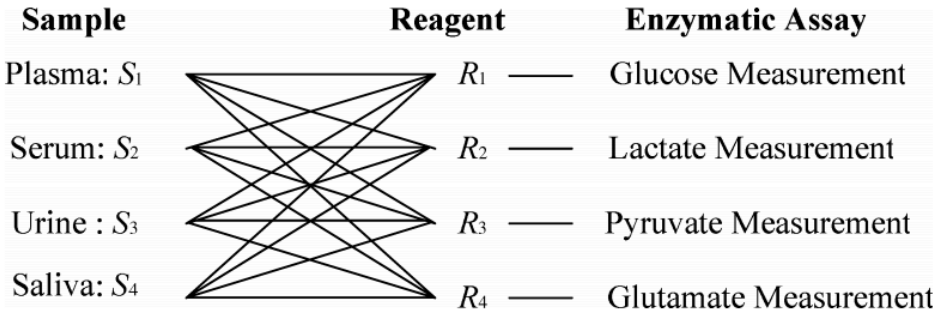


Figure 21

Figure 22 from [17] presents the sequencing graph model of multiplexed diagnostics. In this graph, the operations in the first level are representing input operation, meaning the process in which the liquid is inserted to the chip. Second level operations are mixing operations. And the last level of operations, are detecting operations.

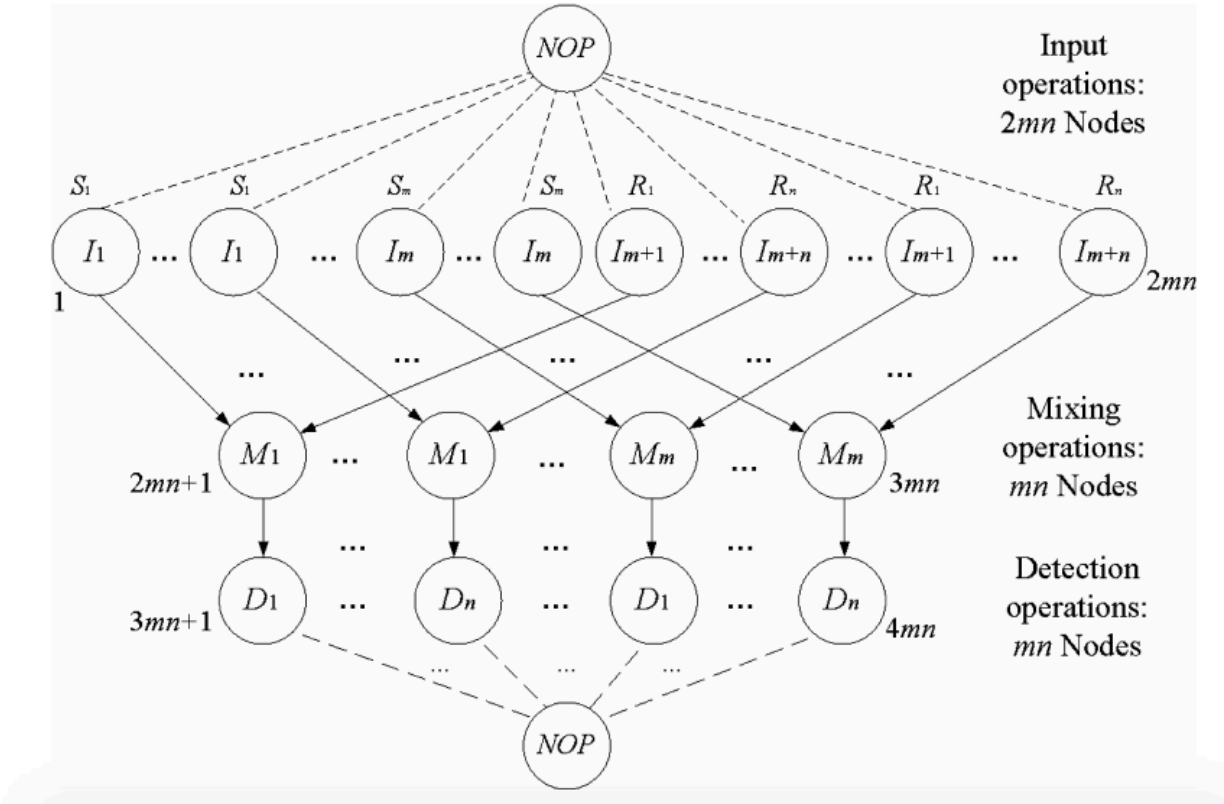


Figure 22

The sequencing graph for one instance of multiplexed diagnostics with two samples and three reagents for detecting three types of metabolites that are used for testing the algorithm presented in this work can be seen in Figure 23. In this model, input operations are removed for simplifications, since the input sources are attached to the valve matrix directly. We can also assume that the detector modules are connected at the outports of the matrix. Therefore detection operations are replaced with transportation to outport. In this way, the mixing stage of the assay can be scheduled using the presented algorithm.

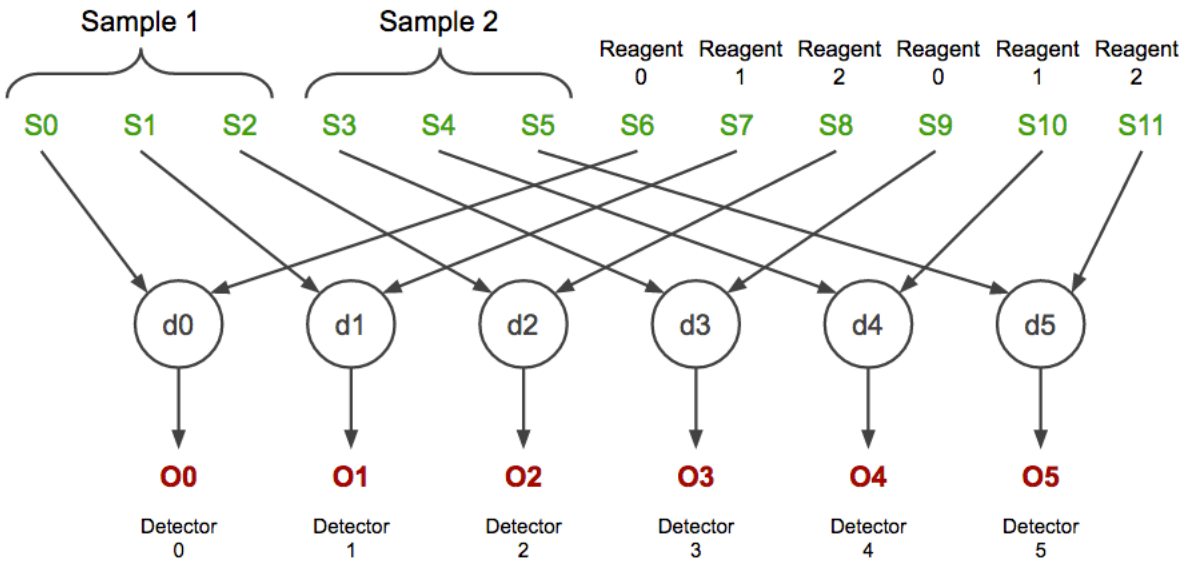


Figure 23

4.1.3. Protein assay

The protein assay is an analytical procedure that is done on a solution for measuring the concentration of protein. The protocol for protein assay based on the Bradford reaction [18], is also a colorimetric glucose assay. The difference between this example and the previous one is a new type of operation, called dilution operation, in the sequencing graph. Protein assay is also more complicated and has more steps comparing to the previous examples. The sequencing graph of protein assay can be seen in Figure 24 from [17]. Nodes denoted by DsS, DsB, and DsR in this picture, represent insertion of sample, buffered liquid and reagent to the chip respectively. Nodes denoted by Dlt, represent dilution operations. And finally, Mix nodes show mixing operations. The last level of operations represents optical detection stage, in which the protein is detected using optical detectors.

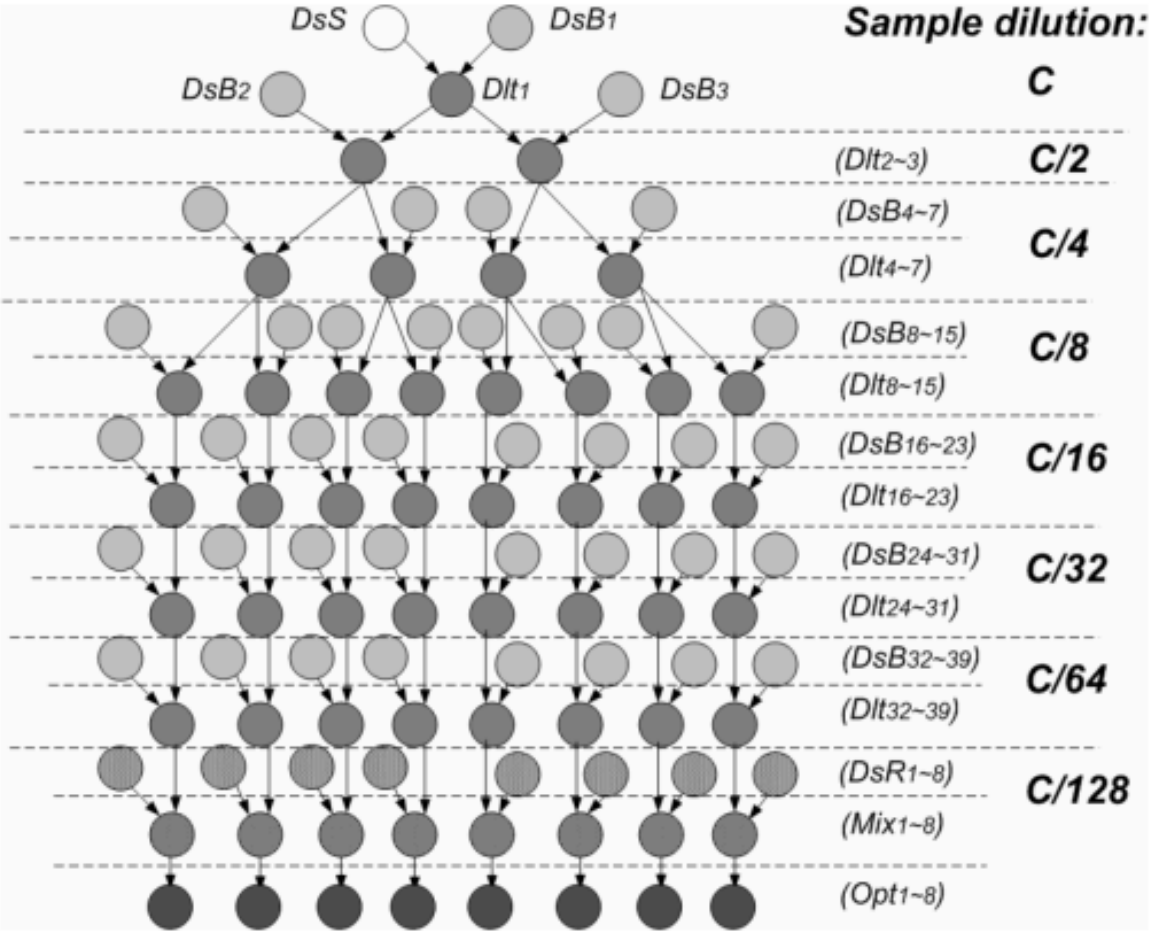


Figure 24

In the dilution operation, an excess amount of liquid is used to dilute the sample, which contains protein, in order to obtain a liquid with desired dilution factor. This liquid is later mixed with reagents. The dilution operation can be seen as a mixing and splitting operation. In the platform presented in this work, dilution can be done in the mixer modules. In order to do so, sample and excessive liquid can be mixed in the mixer, and the desired amount can then be used in the next operation and the unusable amount can be discarded. Therefore in this work, dilution operations can be replaced by mixing operations in the sequencing graph. As it was explained in the previous examples descriptions, detection stage is omitted in the sequencing graph for testing this work, since the detectors are examples of special modules that can be connected to the output ports without changing the concept and affecting the algorithm. It is also assumed in this work that all the mixing operations are the same type, meaning having the same volume of input, so they can be executed in all the mixer modules available. Therefore, excessive liquid is taken to the waste through output ports by adding it to the destination of each operation.

Modified sequencing graph of protein assay that is used in this experiment can be seen in Figure 25 bellow.

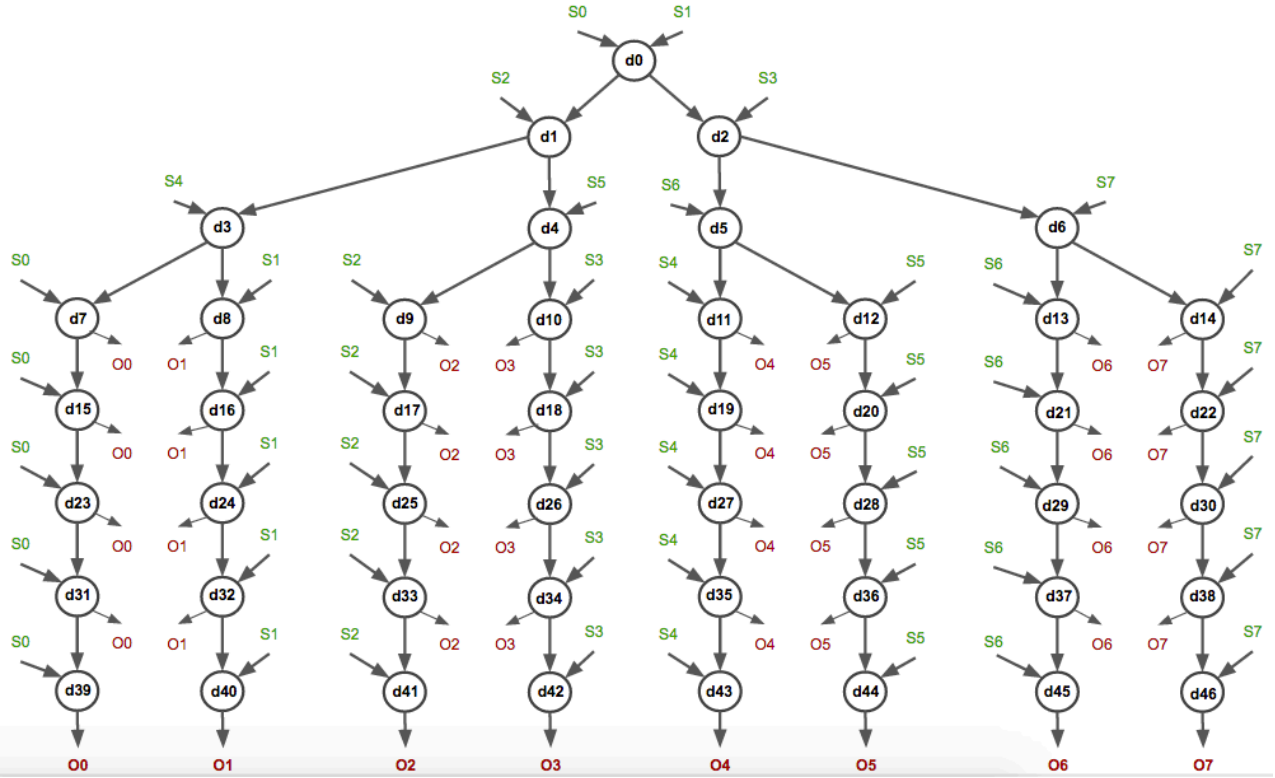


Figure 25

4.2. Overview of the output for one example

In this section, a very simple case is introduced, and the procedure of scheduling this assay is explained step by step. This is only to give a better understanding of the procedure by an example. The simple test case consists of three operations, modeled as a sequencing graph shown in Figure 26. This assay has 4 inputs, one output product, and two waste portion of liquid. Volume of inputs and duration of the operations can be seen in the next table.

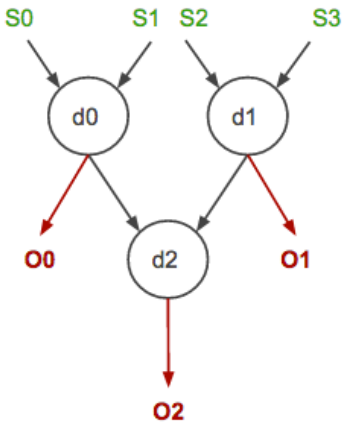


Figure 26

Table 1

Operation	Volume of input 1	Volume of input 2	Duration (in time steps)
0	2	2	40
1	2	2	40
2	3	3	50

The cost formula is set as below in order to minimize the execution time.

$$cost = 1 * (time\ steps) + 0 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

At the beginning, operation 0 is assigned to mixer 0 and operation 1 is assigned to mixer 1, because of the distance of these mixers to their sources. Two of the inputs are routed to enter the mixers, and the other inputs are routed to be stored in the closest point to the destination Figure 27 (a)). The second inputs are routed to the mixer when the first inputs are completely received by the mixer (Figure 27 (b)). After all inputs are delivered to the mixer, execution of the operation starts. This takes 40 time steps at this stage. After 40 time steps, unneeded portion of the product is routed to an output port (Figure 27 (c)).

Next, operation 2 is assigned to mixer 0, which was executing operation 0 previously. This mixer already contains the product of operation 0, therefore only the product of operation 1 has to be transported to this mixer now (path 8 shown in Figure 27 (d)). When the liquid is transported to mixer 0, operation 2 starts, and finishes in 50 time steps. At last, the product of this operation is routed to the output port to be presented as the result of the assay (Figure 27 (e)).

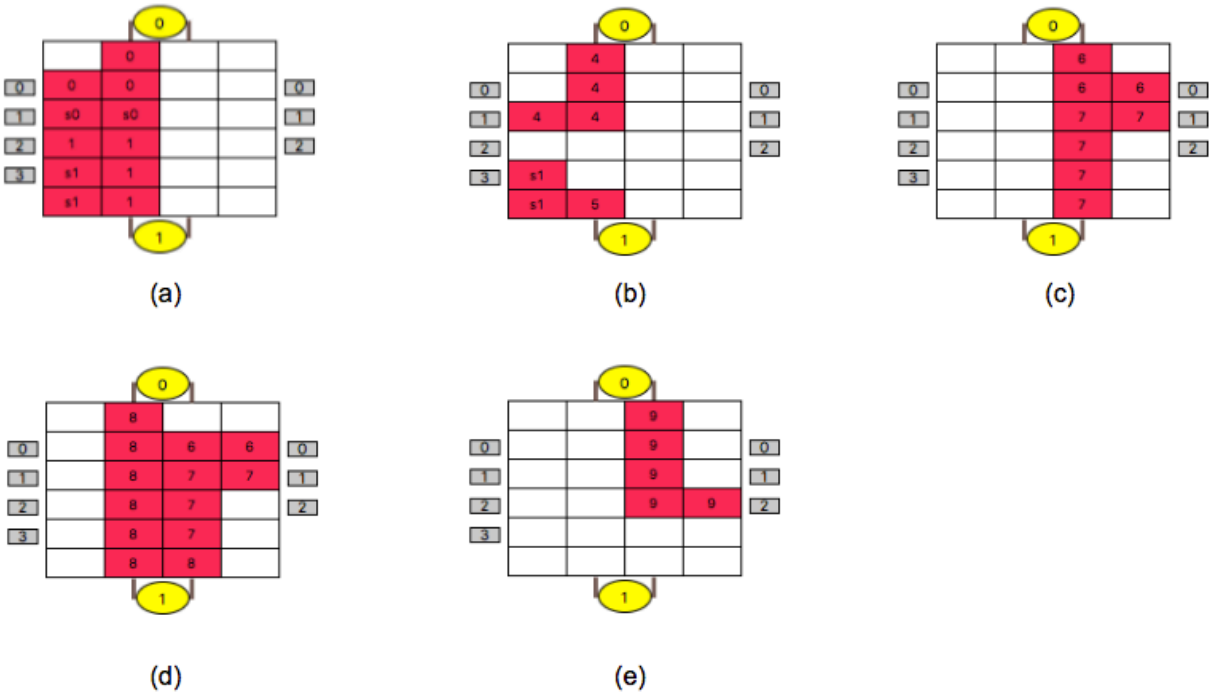


Figure 27

The cost of the solution found by the algorithm is as shown below. Figure 28 also shows the structure of the designed chip, and the cells in valve matrix that are used. Run time of the program for finding this solution was 1.3 second.

Table 2

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
120	2	6	4	120	14

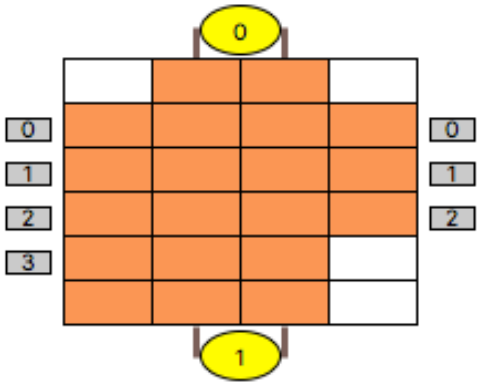


Figure 28

4.3. Results for test cases

As it was discussed earlier, this work presents a method for minimizing the cost of the design, based on the value of the parameters in cost that are provided depending on application. These parameters are time, number of mixer modules, and number of valves. The solution guarantees satisfying introduced constraints. In this section, results of experimenting on test cases will be reported. The effect of change in the value of each parameter, as well as the impact of constraints is shown for each test case separately.

4.3.1. PCR

To run the algorithm on the test case, some configurations must be done at the beginning. The sequencing graph of the assay has been derived from [16]. Since the volumes and number of actuations for mixing operation (time steps) is not mentioned, without loss of generality, sensible assumptions are made. The volume of all the inputs for all operations is assumed to be 2. Therefore half of the product of each operation is moved to the next, and the other half is discarded. Duration of operations is reported in this paper as shown in the following table. These values are reported in seconds. In this work, a more general solution is provided, independent of the speed of valve actuations. Therefore the duration of mixing is calculated by the number of valve actuations, or time steps in other words. This provides flexibility for the design to be implemented using any technology, with different actuation speeds. The duration of mixing operations is assumed as shown in the following table, to be relative to the actual time periods, and close to the time of mixing reported in [3]. Also the maximum number of actuations is assumed to be 100 and maximum allowed time steps for performing this assay is assumed to be 300. Maximum number of mixers allowed is 7, which is equal to the number of operations, and maximum size of valve matrix is 20x20. This number is found by experience, because the number of rows and columns used in every solutions is less than this number.

Table 3

Operation	**Reported duration in seconds**	**Assumed duration in time steps**
Op 0	10	80
Op 1	5	40
Op 2	6	48
Op 3	5	40
Op 4	5	40

Op 5	10	80
Op 6	3	24

In the first test, the objective is to minimize the time regardless of the size of the chip. This can be tested by assigning values 1, 0, and 0 to the coefficients of parameters in the cost formula. Therefore the cost formula will be:

$$cost = 1 * (time\ steps) + 0 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm is as shown below. Figure 29 also shows the structure of the designed chip, and the cells in valve matrix that are used. Run time of the program for finding this solution was 1 second.

Table 4

Cost	**Number of mixers**	**Matrix rows**	**Matrix columns**	**Duration (time steps)**	**Maximum valve actuation**
200	4	11	6	200	22

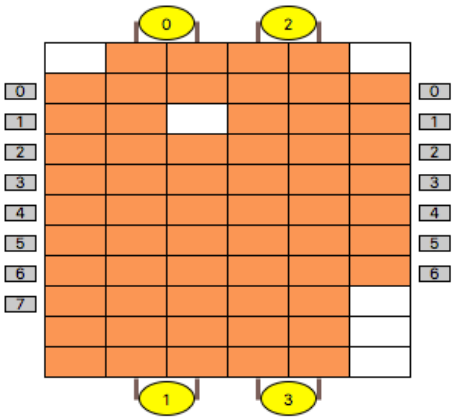


Figure 29

If the assay was to scheduled with no delay between operations, full parallelism, and with neglecting the time of transportation, the durations of the assay would be 138 time steps. Considering only the transportation of liquid between operations (neglecting the transportation of waste to the output ports) in ideal order, the execution time would be 165 time steps. By taking transportation of waste to the ports into account, this number increases to at least 175 time steps. By considering resource constraints and conflict of paths, this duration increases even more. The found solution suggests 14% increase in the ideal execution time (without considering conflict of paths), and less increase comparing to more realistic case. This increase

in not all because of the heuristic nature of the algorithm but is also due to the conflict of paths and added delays that are unavoidable even in the most ideal case. Therefore the solution provides an acceptable duration and resource usage. Following table shows the achieved execution duration for a few different numbers of mixers and size of matrix using the presented algorithm:

Table 5

Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
3	10	6	281	16
4	10	6	212	22
4	11	6	200	22
4	11	7	201	22
4	12	6	203	22
5	11	8	202	22

In the second and third test, the focus will be on the number of mixers and size of valve matrix. To consider minimizing number of mixers and valves used, the coefficients of these parameters in the cost formula can be set to 1. Cost formula for minimizing number of mixers:

$$cost = 0 * (time\ steps) + 1 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm in this case is as shown below. Figure 30 also shows the structure of the designed chip, and the cells in valve matrix that are used.

Table 6

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
3	3	10	6	281	16

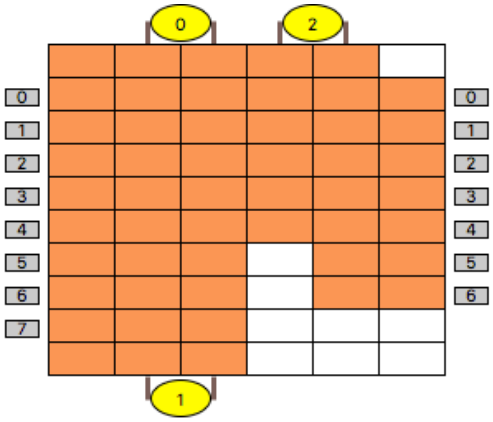


Figure 30

If the aim is to minimize the size of the valve matrix, cost formula for minimizing number of valves:

$$cost = 0 * (time\ steps) + 0 * (number\ of\ mixers) + 1 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm in this case is as follows:

Table 7

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
60	3	10	6	281	16

As it can be seen, the number of mixers and size of the valve matrix is reduced to minimum while still satisfying the time constraints.
By combining all three parameters with same weight to find the cost, the solution will be as follows:

Table 8

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
270	4	11	6	200	22

By giving more value to the size of the matrix, following result is obtained:

Table 9

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
340	4	10	6	212	22

It can be seen that in the later solution, time has increased and number of rows has decreased, and the cost is minimized by this trade-off.
All the tests so far has been done with a loose constraint on valve actuations and duration. Although the constraint on valve actuations is far less than the real limitation of valve, for showing the impact of a tight bound on actuations, this limit is decreased to 10, and tested again. The result is as follows:

Table 10

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
297	5	11	8	204	10

As it can be seen, the maximum number of actuations is decreased to 10. This is done by rerouting the paths, and increase in size of the matrix. Solution still executes the assay within the time frame requested.

4.3.2. Multiplexed diagnostics

Same as the previous test case, to run the algorithm on this test case, some configurations must be done at the beginning. The sequencing graph of the assay has been derived from [17]. Since the volumes and number of actuations for mixing operation (time steps) is not mentioned, without loss of generality, sensible assumptions are made. The volume of all the inputs for all operations is assumed to be 1. The volume of output for all operations is assumed to be 2, meaning all the product is transported to the output ports as result. Duration of operations is reported in this paper as shown in the following table. These values are also reported in seconds. Therefore the duration of mixing operations is approximated as shown in the following table, to be relative to the actual time periods, and close to the time of mixing reported in [3]. Also the maximum number of actuations is assumed to be 100 and maximum allowed time steps for performing this assay is assumed to be 300. Maximum number of mixers allowed is 6, which is enough for executing all operations in parallel, and maximum size of valve matrix is 30x30. This number is found by experience, because the number of rows and columns used in every solution is less than this number.

Table 11

operation	Reported duration in seconds	Assumed duration in time steps
Op 0	5	50
Op 1	5	50
Op 2	5	50
Op 3	3	30
Op 4	3	30
Op 5	3	30

Same way as for the previous test case, in the first test for this case too, the objective is to minimize the time regardless of the size of the chip. This can be tested by assigning values 1, 0, and 0 to the coefficients of parameters in the cost formula.

$$cost = 1 * (time\ steps) + 0 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm is as shown below. Figure 31 also shows the structure of the designed chip, and the cells in valve matrix that are used. Run time of the program for finding this solution was also 1 second.

Table 12

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
91	6	14	8	91	10

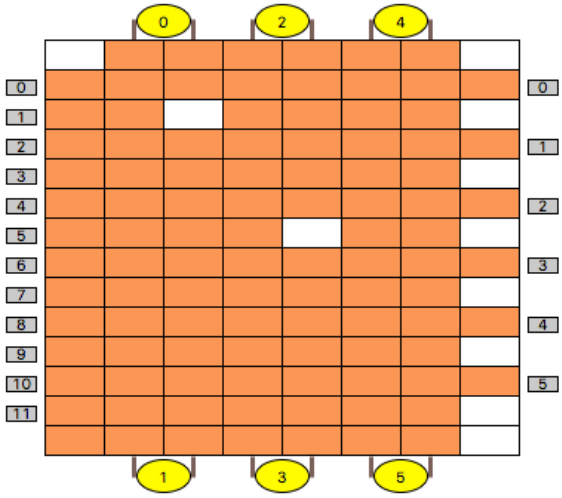


Figure 31

The ideal duration for the assay is when all the operations are executed in parallel (since the execution time of each operation is larger than transportation times) and transportations are done in the shortest possible way. Therefore without considering the conflict of paths, transporting inputs to mixers takes 15 time steps, execution of the longest operation takes 50 time steps, and transporting results to the output ports takes 7 time steps. As a conclusion, the whole assay would approximately take 72 time steps to be performed. By considering resource constraints and conflict of paths, this duration increases even more. The found solution suggests 26% increase in the ideal execution time (without considering conflict of paths), and less increase comparing to more realistic case. This increase in not all because of the heuristic nature of the algorithm but is also due to the conflict of paths and added delays that are unavoidable even in the most ideal case. Therefore the solution provides an acceptable duration and resource usage. Achieved execution duration for a few different numbers of mixers and size of matrix using the presented algorithm is also provided in the following table.

Table 13

Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
6	14	8	91	10
6	15	8	94	10
6	14	10	93	10
5	14	8	105	10
4	15	6	119	8

To consider minimizing number of mixers and valves used, the coefficients of these parameters in the cost formula can be set to 1.
Cost formula for minimizing number of mixers:

$$cost = 0 * (time\ steps) + 1 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

Cost formula for minimizing size of the matrix:

$$cost = 0 * (time\ steps) + 0 * (number\ of\ mixers) + 1 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm in these two cases are as shown below. The first result is by aiming reduction in number of mixers:

Table 14

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
2	2	14	4	189	18

The cost of the solution by focus on minimizing the size of matrix is as follows:

Table 15

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
56	2	14	4	189	18

As it can be seen, the number of mixers and size of the valve matrix is reduced to minimum while still satisfying the time constraints. By combining all three parameters with same weight to find the cost, the solution will be as follows:

Table 16

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
209	6	14	8	91	10

And by giving more value to the size of the matrix, following result is obtained:

Table 17

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
303	2	14	4	189	18

It can be seen that in the later solution, time has increased and number of rows has decreased, and the cost is minimized by this trade-off. All the tests so far has been done with a loose constraint on valve actuations and duration. Although the constraint on valve actuations is far less than the real limitation of valve, for showing the impact of a tight bound on actuations, this limit is decreased to 10, and tested again. The result is as follows:

Table 18

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
96	6	14	8	96	6

As it can be seen, the maximum number of actuations is decreased to 6. This is done by rerouting the paths, and increase in size of the matrix. Solution still executes the assay within the time frame requested. By increasing the pressure on valve actuation constraint by 40% (comparing 6 actuations to 10, which was the maximum number of actuations in the first test), the execution time has increased less than 1%.

4.3.3. Protein assay

Same as the two previous test cases, to run the algorithm on this test case, some configurations must be done at the beginning. The sequencing graph of the assay has been derived from [17]. Since the volumes and number of actuations for mixing operation (time steps) is not mentioned, without loss of generality, sensible assumptions are made. The volume of all the inputs for all operations is assumed to be 1. Therefore for operation 0 to 6, half of the product of mixing must be transported to each of the successor operations, for operations 7 to 38, half of the operation must be transported to the next operation and the other half must be discarded, and for operations 39 to 46, all the product must be transported to the output ports as the result. Duration of operations is reported in this paper as shown in the following table. This values are also reported in seconds. Therefore the duration of mixing operations is approximated as shown in the following table, to be relative to the actual time periods, and close to the time of mixing reported in [3]. Also the maximum number of actuations is assumed to be 100 and maximum allowed time steps for performing this assay is assumed to be 1000. Maximum number of mixers allowed is 20, which is enough for executing all operations in parallel, and maximum size of valve matrix is 30x30. This number is found by experience, because the number of rows and columns used in every solution is less than this number.

Table 19

operation	Reported duration in seconds	Assumed duration in time steps
Op 0 to 38	5	50
Op 39 to 46	3	30

Same way as for the previous test case, in the first test for this case too, the objective is to minimize the time regardless of the size of the chip. This can be tested by assigning values 1, 0, and 0 to the coefficients of parameters in the cost formula.

$$cost = 1 * (time\ steps) + 0 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm is as shown below. Figure 32 also shows the structure of the designed chip, and the cells in valve matrix that are used. Run time of the program for finding this solution was 60 second.

Table 20

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
552	11	11	14	552	24

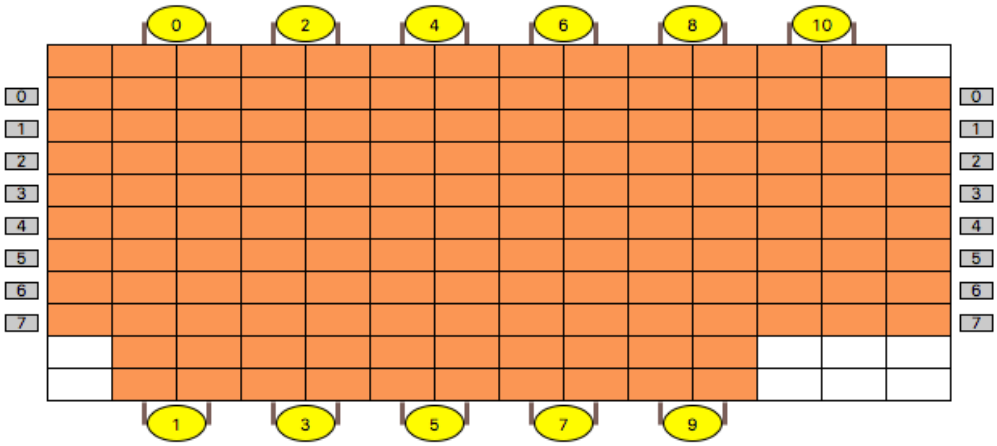


Figure 32

The ideal duration for the assay is when all the operations are executed right after another and transportations are done in the shortest possible way. Therefore without considering the conflict

of paths, transporting inputs to mixers takes at least 5 time steps, execution 7 dilution operations and 1 mixing operation consecutively takes 380, and transporting results to the output ports takes at least 13 time steps. The whole assay would approximately take 400 time steps to be performed in an unrealistic and ideal situation. Taking into account that this assay is relatively large and complicated, when considering resource constraints and conflict of paths, this duration increases significantly. The found solution suggests 38% increase in the ideal execution time (without considering conflict of paths), and less increase comparing to more realistic case. This increase in not all because of the heuristic nature of the algorithm but is also due to the complexity of the assay, conflict of paths, and added delays that are unavoidable even in the most ideal case. Therefore the solution provides an acceptable duration and resource usage. Achieved execution duration for a few different numbers of mixers and size of matrix using the presented algorithm is also provided in the following table.

Table 21

Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
11	11	14	552	24
10	11	13	586	30
11	12	14	574	30
12	11	14	566	24

To consider minimizing number of mixers and valves used, the coefficients of these parameters in the cost formula can be set to 1.
Cost formula for minimizing number of mixers:

$$cost = 0 * (time\ steps) + 1 * (number\ of\ mixers) + 0 * (size\ of\ the\ valve\ matrix)$$

Cost formula for minimizing size of the matrix:

$$cost = 0 * (time\ steps) + 0 * (number\ of\ mixers) + 1 * (size\ of\ the\ valve\ matrix)$$

The cost of the solution found by the algorithm in both of the two cases is as shown below.

Table 22

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
10	10	12	602	20	10

As it can be seen, the number of mixers and size of the valve matrix is reduced to minimum while still satisfying the time constraints. By combining all three parameters with same weight to find the cost, the solution will be as follows:

Table 23

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
717	11	11	14	552	24

All the tests so far has been done with a loose constraint on valve actuations and duration. Although the constraint on valve actuations is far less than the real limitation of valve, for showing the impact of a tight bound on actuations, this limit is decreased to 15, and tested again. The result is as follows:

Table 24

Cost	Number of mixers	Matrix rows	Matrix columns	Duration (time steps)	Maximum valve actuation
565	12	11	14	565	14

As it can be seen, the maximum number of actuations is decreased to 14. This is done by rerouting the paths, and increase in number of mixers. Solution still executes the assay within the time frame requested. By increasing the pressure on valve actuation constraint by 37% (comparing 15 actuations to 24, which was the maximum number of actuations in the first test), the execution time has increased less than 1%.

5 Summary and conclusion

5.1. Summary

Using biochips for performing biomedical assays is getting more attention in the recent years, and much development has been made in automation of the design of these chips. Growing complexity of the assays performed on biochips still demands more attention to the generation of suitable CAD tools for designing biochips efficiently, fast, and reliably. There are still a number of problems unsolved or neglected in the methods proposed for biochip design. One of the most neglected problems is to consider that a valve can only be actuated a limited number of times reliably. Previous work presented in section 2, has introduced a new architecture for biochips, which guarantees preventing the maximum number of actuations for valves to exceed the limit. But this architecture uses a new structure for mixer modules, which implies new problems, such as the quality of mixing and washing.

The effort of this work was to provide a solution that benefits from the advantages of valve centered architecture, dynamic routing and storage generation, and combines it with conventional mixers, which perform mixing operations and washing reliably. The algorithm for scheduling assays to be performed on this architecture, and the methods used in every step of this algorithm was described in section 3. In this method, sequencing graph and specifications of the operations are taken as input. Constraints on the number of actuations, maximum area, and maximum duration are also introduced to the algorithm at the beginning. Algorithm tries to minimize the cost, using the cost formula described in section 3. Finally, the result of experimenting on three famous assays was reported in section 4. It was seen that this algorithm has successfully provided acceptable solutions for minimizing the cost of the design. Since a heuristic method is used in this work, suggested design does not necessarily present the optimal solution, but introduces a solution close to the optimal one, in a feasible amount of time.

In the next section, some suggestions are given for future work on this topic. These works could not be done within the limited time for implementing this project.

5.2. Suggestions for future work

Implementing washing method

One of the reasons for using conventional mixers in this work was to simplify washing the modules reliably. Washing method has not been implemented in this work, but guaranteed to be implementable. A suggested method for applying washing during the performance of an assay, is to insert wash operations in the sequencing graph. This special operation must be identified in the algorithm and assigned to the same path and mixer that were used to execute previous

operation. In this way, wash liquid cleans the path and the module right after the operation is executed. Wash liquid must be transported to an output port to be discarded.

Removing unused valves

At the end of the cost optimization, when the solution is introduced, used and unused valves can be identified easily. In order to minimize the size of the chip even further, unused valves can be removed from the design. However this implies irregularity to the valve matrix. Those valves that have only been used as wall for the generated flow channels can also be replaced by actual walls to reduce the cost of design by removing the control circuit for the valve. This is similar to the last step of the work in [3].

Introducing mixers of different types

Although in this work, all mixers connected to the valve matrix are assumed to be identical for simplification, in reality different types of mixers might be required to execute different mixing operations. Also it might be the case that only one operation requires a large mixer and the rest of the operations can be performed on smaller mixers. In this case, using different mixers can also reduce the cost of the design.

This can easily be implemented in the algorithm presented in this work. By adding a field specifying the type of the mix operation in the input file, and also a field specifying the type of the mixers, suitable mixer can be found for each operation by paying attention to the type of the operation. To do so, in the step of finding a mixer for an operation, type of the operation and mixer must be checked to match, and the best mixer with the same type must be chosen for the operation.

Change the shape of the storage

At the moment, storage units are generated in the shape of the shortest path to the destination. It can be experimented to find out if other shapes of storage unit would help more reduction in duration of the assay. As an example, if the storage unit has a more compact shape, such as a square, it may block fewer paths, and therefore fewer operations to be executed. Also if a no long enough path is found for storage, the storage unit can be generated in the shape of a square at a free location.

Reference

[1] Volpatti, L. R., Yetisen, A. K. (Jul 2014): "*Commercialization of microfluidic devices,*" Trends in Biotechnology 32 (7): 347–350.

[2] Jessica Melin, Stephen R. Quake: "*Microfluidic Large-Scale Integration: The Evolution of Design Rules for Biological Automation,*" Annu. Rev. Biophys. Biomol. Struct. 2007. 36:213–31

[3] Tsun-Ming Tseng, Bing Li, Tsung-Yi Ho, and Ulf Schlichtmann: "*Reliability-aware synthesis for flow-based microfluidic biochips by dynamic-device mapping,*" (2015) 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)

[4] Fei Su, Krishnendu Chakrabarty, and Richard B. Fair: "*Microfluidics-Based Biochips: Technology Issues, Implementation Platforms, and Design-Automation Challenges*," IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 25, NO. 2, FEBRUARY 2006

[5] V. Srinivasan, V. K. Pamula, and R. B. Fair: "*An integrated digital microfluidic lab-on-a-chip for clinical diagnostics on human physiological fluids*," Lab Chip, vol. 4, no. 4, pp. 310–315, Aug. 2004.

[6] S. Venkatesh and Z. A. Memish: "*Bioterrorism: A new challenge for public health*," Int. J. Antimicrob. Agents, vol. 21, no. 2, pp. 200–206, Feb. 2003. [9] M. G. Pollack

[7] M. A. Unger, H.-P. Chou, T. Thorsen, A. Scherer, and S. R. Quake.: *"Monolithic microfabricated valves and pumps by multilayer soft lithography*," Science, 288(5463):113-116, 2000.

[8] W. H. Minhass, P. Pop, and J. Madsen: "*System-level modeling and synthesis of flow-based microfluidic biochips,*" in *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2011.

[9] W. H. Minhass, P. Pop, J. Madsen, M. Hemmingsen, and M. Dufva: "*System-level modeling and simulation of the cell culture microfluidic biochip procell*," In IEEE DTIP, pages 91-98, 2010.

[10] L. M. Fidalgo and S. J. Maerkl: "*A software-programmable microfluidic device for automated biology*," Lab on a Chip, 11:1612-1619, 2011.

[11] Paul Pop, Wajid Hassan Minhass, Jan Madsen, "*Microfluidic very large-scale integration (VLSI)*", Springer; 1st ed. 2016 edition (February 8, 2016)

[12] Ullman, D.: "*NP-complete scheduling problems*," J. Comput. Syst. Sci. 10,384, 393 (1975)

[13] Jon Louis Bentley (1982): "*Writing Efficient Programs*," Prentice Hall. p. 11

[14] C. Y. LEE: "*An Algorithm for Path Connections and Its Applications*," IRE Pm. on Electron. Computer, vol. EC-IO, pp. 346- 365, Sept. 1961

[15] Triantaphyllou, E. (2000): "*Multi-Criteria Decision Making: A Comparative Study,"* Dordrecht, The Netherlands: Kluwer Academic Publishers (now Springer). p. 320.

[16] Fei Su and Krishnendu Chakrabarty: *"Design of Fault-Tolerant and Dynamically-Reconfigurable Microfluidic Biochips*," Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)

[17] Fei Su and Krishnendu Chakrabarty: "*High-Level Synthesis of Digital Microfluidic Biochips*," ACM Journal on Emerging Technologies in Computing Systems, Vol. 3, No. 4, Article 16, Pub. date: January 2008.

[18] SRINIVASAN, V., PAMULA, V. K., PAIK, P., AND FAIR, R. B.: (2004) "*Protein stamping for MALDI mass spectrometry using an electrowetting-based microfluidic platform*," In Proceedings of the SPIE. 5591, 26–32.

Appendix

I: code of the project

```
import queue
import copy
from tkinter import *
import time

class Operation:
    def __init__ (self, name, input1, input2, in1type, in2type, volume1, volume2,
start, duration, destination):
        self.name = name
        self.input1 = input1 #sources
        self.input2 = input2
        self.in1type = "op"
        self.in2type = "op"
        self.in1type = copy.copy(in1type)
        self.in2type = copy.copy(in2type)
        self.in1ready = False
        self.in2ready = False
        self.volume1 = int(volume1) #amount of reagent to use
        self.volume2 = int(volume2)
        self.start = int(start)
        self.duration = int(duration)
        self.end = 0 #specified later when the operation is assigned to mixer
        self.assignedMixer = 100
        self.destinations = copy.copy(destination)

        if in1type == "source":
            self.in1ready = True
        if in2type == "source":
            self.in2ready = True

class Mixer:
    def __init__(self, x, y, typ):
        self.typ = typ #type of the module(for further generalization)
        self.busy = False
        self.used = 0
        self.ops = []
        self.currentOp = []
        self.start = 0
        self.end = 0
        self.x = x #row inport
        self.y = y #column inport
        self.yo = y + 1 #column outport
        self.reserved = []
```

```
class Storage:
    def __init__(self, head, path, sourceOp, destOp, start, readyTime):
        self.head = copy.copy(head)
        self.path = copy.copy(path)
        self.sourceOp = sourceOp
        self.destOp = destOp
        self.start = start #time of storage creation
        self.readyTime = readyTime #time that storage is ready to be used as a source
        self.end = 0 #time of storage termination

def init (rows, cols, mixercount, sourceCount, outportCount):
    global Mem, mixers, sources, storages, outports, allPaths, valves, acLim
    Mem = [ [ [0, 0] for i in range(cols) ] for j in range(rows) ]
    srcDist = (rows - 2)//sourceCount
    outDist = (rows - 2)//outportCount
    mixers = [Mixer((i%2)*(rows-1), (i//2)*2 + 1, 0) for i in range(mixercount)]#all
mixers type 0 for now
    sources = [(i*srcDist + 1, 0) for i in range(sourceCount)]
    outports = [(i*outDist + 1, cols - 1) for i in range(outportCount)]
    storages = []
    allPaths = []
    valves = []
    for i in range(len(Mem)):
        valves.append([0 for j in range(len(Mem[0]))])#horizontal valves
        valves.append([0 for j in range(len(Mem[0])+1)])#vertical valves
    valves.append([0 for j in range(len(Mem[0]))])#last row

def readOpFile(operationsFile):
    f = open(operationsFile ,'r')
    global ops, originalOps
    ops = []
    s1 = []
    s2 = []
    s1type = []
    s2type = []
    dest = []
    l = f.readline()
    l = f.readline()
    while l != '':
        l = l.split(' ')
        s1 = int(l[1][1:len(l[1])])
        if l[1][0] == 's':
            s1type = "source"
        elif l[1][0] == "d":
            s1type = "op"

        s2 = int(l[2][1:len(l[2])])
        if l[2][0] == 's':
            s2type = "source"
        elif l[2][0] == "d":
            s2type = "op"

        dest = l[7][0:-1].split(',')
        for i in range(len(dest)):
```

```
            if dest[i][0] == "d":
                dest[i] = (dest[i][0], int(dest[i][1:len(dest[i])]))
            elif dest[i][0] == "o":##TODO out
                dest[i] = dest[i].split(':')
                dest[i] = (dest[i][0][0], int(dest[i][0][1:len(dest[i][0])]),
int(dest[i][1][0:len(dest[i][1])]) )
        #(name, input1, input2, volume1, volume2, start, duration, destination)
        ops.append(Operation(int(l[0][1:len(l[0])]), s1, s2, s1type, s2type,
int(l[3]), int(l[4]), int(l[5]), int(l[6]), dest))
        l = f.readline()

    ops.sort(key = lambda ops: ops.name) #sort operations by name
    originalOps = copy.copy(ops)
    f.close()

def printOps():
    global ops
    for op in ops:
        print("op"+str(op.name)+" start= "+ str(op.start)+ ", inputs: " + op.in1type
+ str(op.input1) + "," + op.in2type + str(op.input2) + ", out:",op.destinations)

def readConstraintFile(PlatformFile):
    #read platform specifications
    f = open(PlatformFile, 'r')
    l = f.readline()
    l = f.readline()
    l = l.split(' ')
    f.close()
    return int(l[0]), int(l[1]), int(l[2]), int(l[3]), int(l[4]), int(l[5]),
int(l[6]), int(l[7]), int(l[8]), int(l[9][0:len(l[9])])

def printPlatform(a1, a2, a3, a4, a5, a6, a7):
    print("Specifications: ", "rows = ", a1, ", columns = ", a2, ", #mixers = ", a3,
", #sources = ", a4, ", #outports = ", a5, ", max. duration = ", a6, ", max.
actuations = ", a7)

def plan():
    global ops, mixers, Mem, sources, outports, allPaths, t, tEnd, qInfo, originalOps
    ops = copy.copy(originalOps)
    #queues for managing operations
    tempOps = []
    ready = []
    reserved = []
    act1 = []
    act2 = []
    finished = []
    removed = []
    finishedWait = [-1 for i in range(len(ops))]
    qInfo = "" #q info at each point of time

    tempOps = copy.copy(ops)

    #keep track of time
```

```
    t = 0

    nOfOps = len(ops)
    while t < tEnd and (len(removed) != len(ops) or Mem != [[[0, 0] for i in
range(len(Mem[0])) ] for j in range(len(Mem))]):
        changed = False
        #move finished ops to removed when all liquid is taken
        i = 0
        n = len(finished)
        while i < n:
            ii = 0
            nn = len(finished[i].destinations)
            while ii < nn:
                if finished[i].destinations[ii][0] == "o":
                    #move result out...
                    path = []
                    path = findAPath((mixers[finished[i].assignedMixer].x,
mixers[finished[i].assignedMixer].yo), "op", finished[i].destinations[ii][2],
outports[finished[i].destinations[ii][1]], "outport")
                    if path != False:
                        assignPath(path, finished[i].destinations[ii][2])
                        allPaths.append(("d"+ str(finished[i].name), 'o' +
str(finished[i].destinations[ii][1]), copy.copy(path),
copy.copy([(Mem[p[0]][p[1]][0], Mem[p[0]][p[1]][1]) for p in path]), t))#allPath
                        finishedWait[finished[i].name] =
max(finishedWait[finished[i].name], finished[i].destinations[ii][2]+t)
                        del finished[i].destinations[ii]
                        nn = nn - 1
                    else:
                        ii = ii + 1
                else:
                    ii = ii + 1

            if  finished[i].destinations == [] and finishedWait[finished[i].name] ==
t:
                #user must care about the reagent left in mixer

                if mixers[finished[i].assignedMixer].reserved != []:
                    mixers[finished[i].assignedMixer].busy = True
                    mixers[finished[i].assignedMixer].start = t

mixers[finished[i].assignedMixer].ops.append(copy.copy(mixers[finished[i].assignedMix
er].reserved ))
                    mixers[finished[i].assignedMixer].currentOp =
copy.copy(mixers[finished[i].assignedMixer].reserved )
                    mixers[finished[i].assignedMixer].reserved = []
                else:
                    mixers[finished[i].assignedMixer].busy = False
                    mixers[finished[i].assignedMixer].currentOp = []

                changed = True
                removed.append(copy.copy(finished[i]))
                del finished[i]
                n = n - 1
```

```
            else:
                i = i + 1
        #move act2 ops to finished if fully executed and t = op.end
        i = 0
        n = len(act2)
        while i < n:
            if act2[i].end != 0 and act2[i].end == t: #reached the end
                #notify destinations
                for j in range(len(tempOps)):
                    if tempOps[j].in1type == "op" and tempOps[j].input1 ==
act2[i].name:
                        tempOps[j].in1ready = True
                    elif tempOps[j].in2type == "op" and tempOps[j].input2 ==
act2[i].name:
                        tempOps[j].in2ready = True

                changed = True
                finished.append(copy.copy(act2[i]))
                del act2[i]
                n = n - 1
            else:
                i = i + 1

        #move ops to ready if inputs are ready
        i = 0
        n = len(tempOps)
        while i < n:
            if (tempOps[i].in1ready and tempOps[i].in2ready):
                changed = True
                ready.append(copy.copy(tempOps[i]))
                del tempOps[i]
                n = n - 1
            else:
                i = i + 1

       #move ops from reserved to act1:
        i = 0
        n = len(reserved)
        while i < n:
            if mixers[reserved[i].assignedMixer].currentOp.name == reserved[i].name:
                changed = True
                act1.append(copy.copy(reserved[i]))
                del reserved[i]
                n = n - 1
            else:
                i = i + 1



        #move ready ops to act1 if mixer and free paths is available
        i = 0
        n = len(ready)
        while i < n:
            m = -1
            m = findMixer(ready[i])
```

```
            if  m != -1: #mixer found
                #setup for findPath function
                temps1, temps2, tempv1, tempv2, temptg = findAPathSetup(ready[i], m)

                #search for paths
                path1 = []
                temppath1 = []
                temppath2 = []

                if ready[i].in1type == "op":
                    if m == ops[ready[i].input1].assignedMixer:
                        temppath1 = "same" #execute in the same mixer as input
                    else:
                        temppath1 = findAPath(temps1, ready[i].in1type, tempv1,
temptg, "mixer")
                else:
                    temppath1 = findAPath(temps1, ready[i].in1type, tempv1, temptg,
"mixer")
                if ready[i].in2type == "op":
                    if m == ops[ready[i].input2].assignedMixer:
                        temppath2 = "same" #execute in the same mixer as input
                    else:
                        temppath2 = findAPath(temps2, ready[i].in2type, tempv2,
temptg, "mixer")
                else:
                    temppath2 = findAPath(temps2, ready[i].in2type, tempv2, temptg,
"mixer")

                #if path for in1 is not found but a path is found for in2, swap
inputs
                if temppath1 == False and temppath2 == False:
                    None
                elif (temppath1 == False and temppath2 != False) or (temppath2 ==
"same"):
                    #swap inputs
                    path1 = copy.copy(temppath2)
                    ready[i].input1, ready[i].input2 = ready[i].input2,
ready[i].input1
                    ready[i].in1type, ready[i].in2type = ready[i].in2type,
ready[i].in1type
                    ready[i].volume1, ready[i].volume2 = ready[i].volume2,
ready[i].volume1
                else:
                    path1 = copy.copy(temppath1)

                if not path1: #path not found
                    i = i + 1
                else:
                    #update op
                    ready[i].assignedMixer = m
                    ops[ready[i].name].assignedMixer = m

                    if path1 == "same":
                        mixers[m].reserved = copy.copy(ready[i])
```

```
                    else:
                        #assign mixer
                        mixers[m].busy = True
                        mixers[m].start = t
                        mixers[m].ops.append(copy.copy(ops[ready[i].name]))
                        mixers[m].currentOp = copy.copy(ops[ready[i].name])

                    #assign path1
                    if path1 != "same":
                        assignPath(path1, ready[i].volume1)
                        allPaths.append((ready[i].in1type + str(ready[i].input1),
"d"+str(ready[i].name), copy.copy(path1), copy.copy([(Mem[p[0]][p[1]][0],
Mem[p[0]][p[1]][1]) for p in path1]), t))

                    #delete op from list of the destinations for all finished ops
                    if ready[i].in1type == "op":
                        for k in range(len(finished)):
                            if finished[k].name == ready[i].input1:
                                finished[k].destinations.remove(('d',ready[i].name))
                                if path1 == "same":
                                    finishedWait[finished[k].name] =
max(finishedWait[finished[k].name], t + 1)
                                else:
                                    finishedWait[finished[k].name] =
max(finishedWait[finished[k].name], ready[i].volume1+t)
                        if ops[ready[i].input1].destinations.count(('d',
ready[i].name)) > 0:
                            ops[ready[i].input1].destinations.remove(('d',
ready[i].name))

                    changed = True
                    if path1 == "same":
                        reserved.append(ready[i])
                    else:
                        act1.append(ready[i])
                    del ready[i]
                    n = n - 1
            else: #mixer not found
                i = i + 1

        #move ops from act1 to act2
        i = 0
        n = len(act1)
        while i < n:

            if act1[i].in2type == "storage" and storages[act1[i].input2].readyTime >
t:
                #storage is not ready to use
                path2 = False
            else:
                #find a path from source to the mixer
                temps1, temps2, tempv1, tempv2, temptg = findAPathSetup(act1[i],
act1[i].assignedMixer)
                path2 = []
```

```
                path2 = findAPath(temps2, act1[i].in2type, tempv2, temptg, "mixer")
            #check path
            if path2 != False:
             #change storage cells to normal
                if act1[i].in2type == "storage":
                    for k in range(len(storages[act1[i].input2].path)-2, -1,-1):
                        path2.insert(0,storages[act1[i].input2].path[k])
                    storages[act1[i].input2].end = t + act1[i].volume2 -1
                    for p in storages[act1[i].input2].path:
                        Mem[p[0]][p[1]][1] = 0

             #assign path
                assignPath(path2, act1[i].volume2)
                allPaths.append((act1[i].in2type + str(act1[i].input2),
"d"+str(act1[i].name), copy.copy(path2), copy.copy([(Mem[p[0]][p[1]][0],
Mem[p[0]][p[1]][1]) for p in path2]), t))

                #update sources
                if act1[i].in2type == "op":
                    for k in range(len(finished)):
                        if finished[k].name == act1[i].input2:
                            finished[k].destinations.remove(('d',act1[i].name))
                            finishedWait[finished[k].name] =
max(finishedWait[finished[k].name], act1[i].volume2+t)

                    if ops[act1[i].input2].destinations.count(act1[i].name) > 0 :
                        ops[act1[i].input2].destinations.remove(('d',act1[i].name))

                #update op
                if act1[i].in2type == "storage":
                    act1[i].end = t + len(path2) + act1[i].duration
                    ops[act1[i].name].end = t + len(path2) + act1[i].duration
                else:
                    act1[i].end = t + len(path2) + act1[i].volume2 - 1 +
act1[i].duration
                    ops[act1[i].name].end = t + len(path2) + act1[i].volume2 - 1 +
act1[i].duration

                changed = True
                act2.append(copy.copy(act1[i]))
                del act1[i]
                n = n - 1
            else:
                #path not found
                #store
                if act1[i].in2type != "storage":
                    path2 = []
                    path2 = store(temps2, tempv2, temptg, t, act1[i].input2,
ops[act1[i].name].name)
                    if path2 == False:
                        None
                    else:
```

```
                        allPaths.append((act1[i].in2type + str(act1[i].input2),
"storage"+str(len(storages)-1), copy.copy(path2), copy.copy([(Mem[p[0]][p[1]][0],
Mem[p[0]][p[1]][1]) for p in path2]), t))
                        #update sources
                        if act1[i].in2type == "op":
                            for k in range(len(finished)):
                                if finished[k].name == act1[i].input2:

finished[k].destinations.remove(('d',act1[i].name))
                                    finishedWait[finished[k].name] =
max(finishedWait[finished[k].name], act1[i].volume2+t)

                            if ops[act1[i].input2].destinations.count(act1[i].name) >
0 :

ops[act1[i].input2].destinations.remove(('d',act1[i].name))

                        act1[i].in2ready == len(path2)-act1[i].volume2
                        act1[i].in2type = "storage"
                        act1[i].input2 = len(storages) - 1
                        ops[act1[i].name].in2type = "storage"
                        ops[act1[i].name].input2 = len(storages) - 1

                i = i + 1

        #info of queues queues for tracking
        if changed:
            qInfo = qInfo + '\n' + "/////////////// t = " + str(t)
            qInfo = qInfo + '\n' + " ready: " + str([o.name for o in ready]) + "
reserved: " + str([o.name for o in reserved]) + " act1: " + str([o.name for o in
act1]) + " act2: " + str([o.name for o in act2]) + " finished: " + str([o.name for o
in finished]) + " removed: " +str([o.name for o in removed]) + '\n'

        #Mem -1
        for i in range(len(Mem)):
            for j in range(len(Mem[i])):
                if Mem [i][j][1] == 0 and Mem[i][j][0] > 0:
                    Mem[i][j][0] = Mem[i][j][0] - 1
        t = t + 1

def findAPathSetup(op, mixerIndex):
    global mixers, sources, storages
    #s1
    if op.in1type == "op":
        temps1 = (mixers[ops[op.input1].assignedMixer].x,
mixers[ops[op.input1].assignedMixer].yo)
    elif op.in1type == "source":
        temps1 = copy.copy(sources[op.input1])
    elif op.in1type == "storage":
        temps1 = copy.copy(storages[op.input1].head)
    #s2
    if op.in2type == "op":
```

```
        temps2 = (mixers[ops[op.input2].assignedMixer].x,
mixers[ops[op.input2].assignedMixer].yo)
    elif op.in2type == "source":
        temps2 = copy.copy(sources[op.input2])
    elif op.in2type == "storage":
        temps2 = copy.copy(storages[op.input2].head)
    #v1,v2
    tempv1 = op.volume1
    tempv2 = op.volume2
    #tg
    temptg = (mixers[mixerIndex].x , mixers[mixerIndex].y)

    return temps1, temps2, tempv1, tempv2, temptg

def calcDist(op, m):
    global sources, ops, mixers

    if op.in1type == "source":
        s1 = sources[op.input1]
    elif op.in1type == "op":
        s1 = (mixers[ops[op.input1].assignedMixer].x,
mixers[ops[op.input1].assignedMixer].yo)

    if op.in2type == "source":
        s2 = sources[op.input2]
    elif op.in2type == "op":
        s2 = (mixers[ops[op.input2].assignedMixer].x,
mixers[ops[op.input2].assignedMixer].yo)

    dist1 = abs(m.x - s1[0]) + abs(m.y - s1[1])
    dist2 = abs(m.x - s2[0]) + abs(m.y - s2[1])
    return dist1, dist2

def findMixer(op):
    global mixers, sources, ops
    x1 = 5 #the point where mixer cannot be used anymore

    m1 = 0.1
    m2 = 0.1
    if op.in1type == "op":
        m1 = op.input1
    if op.in2type == "op":
        m2 = op.input2

    priority = []
    #G1 mixers: those that were used, but not too much
    for m in mixers:
        if (m.busy == False or (m.reserved == [] and (m.currentOp.name == m1 or
m.currentOp.name == m2))) and len(m.ops) > 0 and len(m.ops) < x1: #busy or containing
source op
            dist1, dist2 = calcDist(op, m)
            priority.append((mixers.index(m), dist1+dist2))
    if len(priority) > 0:
```

```
            priority.sort(key = lambda tup:tup[1])
            return priority[0][0]

        #G2 mixers: those that are not used yet (new mixers)
        for m in mixers:
            if m.busy == False and len(m.ops) == 0: #busy
                dist1, dist2 = calcDist(op, m)
                priority.append((mixers.index(m), dist1+dist2))
        if len(priority) > 0:
            priority.sort(key = lambda tup:tup[1])
            return priority[0][0]

        return -1

    def findAPath(s, stype, v, tg, tgtype):
        global Mem, t, mixers, valves, acLim

        tempmem1 = [[0 for j in range(len(Mem[0]))] for i in range(len(Mem))]
        for i in range(len(Mem)):
            for j in range(len(Mem[0])):
                tempmem1[i][j] = [copy.copy(Mem[i][j][0]), copy.copy(Mem[i][j][1])]
                if tempmem1[i][j][0] != 0 :
                    tempmem1[i][j][0] = [-2]
        #not use mixers outport
        if tgtype == "mixer":
            if (tempmem1[tg[0]][tg[1]+1] != [0,0]):
                return False
            else:
                tempmem1[tg[0]][tg[1]+1] = [-3,0]

        for m in mixers:
            if m.busy == True and (m.x, m.yo) != s:
                tempmem1[m.x][m.yo] = [-3,0]

        cp = (s[0], s[1]) #current point

        if stype == "storage":
            if tempmem1[cp[0]][cp[1]][1] != 's':
                return False
            else:
                tempmem1[cp[0]][cp[1]] = [1, 0]
        else:
            if tempmem1[cp[0]][cp[1]] != [0, 0]:
                return False
            else:
                tempmem1[cp[0]][cp[1]][0] = 1

        q = queue.Queue()
        while cp != tg:
            #check right
            if cp[1] < len(tempmem1[0])-1 and tempmem1[cp[0]][cp[1]+1] == [0, 0]:
                if valves[cp[0]*2+1][cp[1]+1] <= acLim - 2*v:
                    tempmem1[cp[0]][cp[1]+1][0] = tempmem1[cp[0]][cp[1]][0] + 1
```

```
                q.put((cp[0], cp[1]+1))
        #check left
        if cp[1] > 0 and tempmem1[cp[0]][cp[1]-1] == [0, 0]:
            if valves[cp[0]*2+1][cp[1]] <= acLim - 2*v:
                tempmem1[cp[0]][cp[1]-1][0] = tempmem1[cp[0]][cp[1]][0] + 1
                q.put((cp[0], cp[1]-1))
        #check down
        if cp[0] < len(tempmem1) - 1 and tempmem1[cp[0]+1][cp[1]] == [0, 0]:
            if valves[cp[0]*2+2][cp[1]] <= acLim - 2*v:
                tempmem1[cp[0]+1][cp[1]][0] = tempmem1[cp[0]][cp[1]][0] + 1
                q.put((cp[0]+1, cp[1]))
        #check up
        if cp[0] > 0 and tempmem1[cp[0]-1][cp[1]] == [0, 0]:
            if valves[cp[0]*2][cp[1]] <= acLim - 2*v:
                tempmem1[cp[0]-1][cp[1]][0] = tempmem1[cp[0]][cp[1]][0] + 1
                q.put((cp[0]-1, cp[1]))

        if q.empty() == False:
            cp = q.get()
        else:
            return False #found no path

    #Back tracking
    cp = copy.copy(tg)
    nxt = [] #possible choices
    path = []
    path.append(cp)

    #if routing from source->go horizontal first
    if stype == "source":
        while cp != s:
            nxt = []
            #down
            if cp[0] < len(tempmem1)-1 and tempmem1[cp[0]+1][cp[1]][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]+1][cp[1]][1] == 0):
                if valves[cp[0]*2+2][cp[1]] <= acLim - 2*v:
                    nxt.append((1,0))
            #up
            if cp[0] > 0 and tempmem1[cp[0]-1][cp[1]][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]-1][cp[1]][1] == 0):
                if valves[cp[0]*2][cp[1]] <= acLim - 2*v:
                    nxt.append((-1,0))
            #right
            if cp[1] < len(tempmem1[0])-1 and tempmem1[cp[0]][cp[1]+1][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]][cp[1]+1][1] == 0):
                if valves[cp[0]*2+1][cp[1]+1] <= acLim - 2*v:
                    nxt.append((0,1))
            #left
            if cp[1] > 0 and tempmem1[cp[0]][cp[1]-1][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]][cp[1]-1][1] == 0):
                if valves[cp[0]*2+1][cp[1]] <= acLim - 2*v:
                    nxt.append((0,-1))
            if len(nxt)>0:
                cp = (cp[0]+nxt[0][0],cp[1]+nxt[0][1])
```

```
                path.append(cp)
            else:
                print("ERROR!!! findAPath: backtracking: no next point found!",
stype, s, tg)
                break

    #else-> go vertical first
    else:
        while cp != s:
            nxt = []
            #right
            if cp[1] < len(tempmem1[0])-1 and tempmem1[cp[0]][cp[1]+1][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]][cp[1]+1][1] == 0):
                if valves[cp[0]*2+1][cp[1]+1] <= acLim - 2*v:
                    nxt.append((0,1))
            #left
            if cp[1] > 0 and tempmem1[cp[0]][cp[1]-1][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]][cp[1]-1][1] == 0):
                if valves[cp[0]*2+1][cp[1]] <= acLim - 2*v:
                    nxt.append((0,-1))
            #down
            if cp[0] < len(tempmem1)-1 and tempmem1[cp[0]+1][cp[1]][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]+1][cp[1]][1] == 0):
                if valves[cp[0]*2+2][cp[1]] <= acLim - 2*v:
                    nxt.append((1,0))
            #up
            if cp[0] > 0 and tempmem1[cp[0]-1][cp[1]][0] ==
tempmem1[cp[0]][cp[1]][0]-1 and (tempmem1[cp[0]-1][cp[1]][1] == 0):
                if valves[cp[0]*2][cp[1]] <= acLim - 2*v:
                    nxt.append((-1,0))
            if len(nxt)>0:
                cp = (cp[0]+nxt[0][0],cp[1]+nxt[0][1])
                path.append(cp)
            else:
                print("ERROR!!! findAPath: backtracking: no next point found!",
stype, s, tg)
                break

    path.reverse()
    return path

def assignPath(path, vol):
    global Mem, storages
    cp = []
    for cp in path:
        Mem[cp[0]][cp[1]][0] = vol + path.index(cp)
    addActuations(copy.copy(path), vol)

def addActuations(path, vol):
    global valves
    p = copy.copy(path)
    for i in range(len(p)-1):
        cp = p[i]
        np = p[i+1]
```

```
        #up
        if np[0] == cp[0] - 1 and np[1] == cp[1]:
            valves[cp[0]*2][cp[1]] += vol*2
        #down
        if np[0] == cp[0] + 1 and np[1] == cp[1]:
            valves[cp[0]*2+2][cp[1]] += vol*2
        #left
        if np[0] == cp[0] and np[1] == cp[1] - 1:
            valves[cp[0]*2+1][cp[1]] += vol*2
        #right
        if np[0] == cp[0] and np[1] == cp[1] + 1:
            valves[cp[0]*2+1][cp[1]+1] += vol*2

def store(src, vol, destination, t, sourceOp, destOp):
    global Mem

    if src[1] == 0:
        srcType = "source"
    else:
        srcType = "op"

    dest = copy.copy(destination)
    g = [0, 0]
    if src[0] - dest[0] > 0:
        g[0] = 1
    elif src[0] - dest[0] < 0:
        g[0] = -1
    if src[1] - dest[1] > 0:
        g[1] = 1
    elif src[1] - dest[1] < 0:
        g[1] = -1
    grad = (g[0], g[1])
    if grad[1] != 0:
        grad = (0, grad[1])
    path = findAPath(src, srcType, vol, dest, "storage")
    candidates = []
    blocking = False
    for i in range(min(src[0], dest[0]), max(src[0],dest[0])+1):
        for j in range(min(src[1], dest[1]), max(src[1],dest[1]+1)+1):
            if candidates.count( [ (i, j), (abs(dest[0] - i) + abs(dest[1] - j) ),
(abs(src[0] - i) + abs(src[1] - j) ) ] ) == 0:
                blocking = False
                for m in mixers:
                    if (i, j) == (m.x, m.yo):
                        blocking = True
                if blocking == False:
                    candidates.append( [ (i, j), (abs(dest[0] - i) + abs(dest[1] - j)
), (abs(src[0] - i) + abs(src[1] - j) )  ] )
    candidates.sort(key = lambda tup:tup[2])
    candidates.sort(key = lambda tup:tup[1])
    i = 0
    while path == False:
        if i < len(candidates) - 1:
            i = i + 1
```

```
            dest = copy.copy(candidates[i][0])
            path = findAPath(src, srcType, vol, dest, "storage")
        else:
            return False


    assignPath(path[0:len(path)-vol], vol) #assign path from source to the storage

    storages.append(Storage(path[len(path)-1], path[len(path)-vol:len(path)],
sourceOp, destOp, t, t + len(path)-1))
    assignPath(path[len(path)-vol:len(path)], 1) #assign storage path

    #set mem cells as storage
    cp = []
    for i in range(len(path)-vol,len(path)):
        cp = copy.copy(path[i])
        Mem[cp[0]][cp[1]][1] = 's'

    return path

def printResult():
    global allPaths, t, mixers, Mem, ops, storages, tEnd
    usedMixerCount = 0
    for m in mixers:
        if len(m.ops) > 0:
            usedMixerCount = usedMixerCount + 1

    usedRows = []
    usedColumns = []
    tempMem = [[0 for i in range(len(Mem[0]))] for j in range(len(Mem))]
    for p in allPaths:
        for cp in p[2]:
            tempMem[cp[0]][cp[1]] = 1

    usedRows, usedCols = detectUsed()
    maxVal, maxIndices = maxActuation()
    maxMixer, maxMixerIndex = maxUsedMixer()

    print('\n' + "////////////////////////////SUMMARY//////////////////////////")
    if t == tEnd:
        print("***UNSUCCESSFUL!***" + '\n')
    else:
        print("Total time = " + str(t))
        print("Max. actuations = " + str(maxVal)+ ",  for valve(s)", maxIndices)
        print("Total number of mixers used = " + str(usedMixerCount) + " out of " +
str(len(mixers)))
        print("Max. mixer usage = " + str(maxMixer) + ",  for mixer " +
str(maxMixerIndex))
        print("Total number of rows used = " + str(len(usedRows)) + ",  Columns used
= " + str(len(usedCols)))
        print("Total number of paths = " + str(len(allPaths)) + '\n')
```

```
def printMixers():
    global ops
    print('\n' + "////////mixers////////")
    for op in ops:
        print("op"+str(op.name)+" -> mixer" + str(op.assignedMixer))

def printPaths(with_details = False):
    global allPaths, Mem
    print('\n' + "////////Paths////////")
    for p in allPaths:
        print("path"+str(allPaths.index(p))+",", p[0], p[1])
        if with_details:
            print(p[2])

def printQInfo():
    global qInfo
    for line in qInfo.splitlines():
        print(line)

def detectUsed():
    global Mem, allPaths, sources, outports
    usedRows = []
    usedCols = []

    usedCols.append(0)
    for s in sources:
        if usedRows.count(s[0]) == 0:
            usedRows.append(s[0])
    for o in outports:
        if usedRows.count(o[0]) == 0:
            usedRows.append(o[0])

    for p in allPaths:
        for i in range(len(p[2])-1):
            cp = p[2][i]
            np = p[2][i+1]
            if cp[0] != np[0]: #col used
                if usedCols.count(cp[1]) == 0:
                    usedCols.append(cp[1])
            if cp[1] != np[1]: #row used
                if usedRows.count(cp[0]) == 0:
                    usedRows.append(cp[0])
    for m in mixers:
        if len(m.ops)>0:
            if usedCols.count(m.y) == 0:
                    usedCols.append(m.y)
            if usedCols.count(m.yo) == 0:
                    usedCols.append(m.yo)

    usedRows.sort()
    usedCols.sort()
    return usedRows, usedCols
```

```
def countActuations():
    global allPaths
    valves = []
    for i in range(len(Mem)):
        valves.append([0 for j in range(len(Mem[0]))])#horizontal valves
        valves.append([0 for j in range(len(Mem[0])+1)])#vertical valves
    valves.append([0 for j in range(len(Mem[0]))])#last row

    for p in allPaths:
        vol = p[3][0][0]
        for i in range(len(p[2])-1):
            cp = p[2][i]
            np = p[2][i+1]
            #up
            if np[0] == cp[0] - 1 and np[1] == cp[1]:
                valves[cp[0]*2][cp[1]] += vol*2
            #down
            if np[0] == cp[0] + 1 and np[1] == cp[1]:
                valves[cp[0]*2+2][cp[1]] += vol*2
            #left
            if np[0] == cp[0] and np[1] == cp[1] - 1:
                valves[cp[0]*2+1][cp[1]] += vol*2
            #right
            if np[0] == cp[0] and np[1] == cp[1] + 1:
                valves[cp[0]*2+1][cp[1]+1] += vol*2
    return valves

def maxActuation():
    global valves
    maxval = max([max(i) for i in valves])
    maxindices = []

    for i in valves:
        if i.count(maxval) > 0:
            for j in i:
                if j == maxval and maxindices.count((valves.index(i), i.index(j))) ==
0:
                    maxindices.append((valves.index(i), i.index(j)))
    return maxval, maxindices

def maxUsedMixer():
    global mixers
    maxMixer = 0
    maxIndex = 0
    for m in mixers:
        if len(m.ops) > maxMixer:
            maxMixer = len(m.ops)
            maxIndex = mixers.index(m)

    return maxMixer, maxIndex

def drawPaths(tempPaths = []):
    global allPaths, mixers, sources, outports, Mem
```

```
    master = Tk()
    win = Canvas(master, width = 1200, height = 750)
    win.pack()
    w = 40
    l = 20
    b = 30

    if tempPaths == []:
        paths = [i for i in range(len(allPaths))]
    else:
        paths = copy.copy(tempPaths)

    #draw valve matrix
    for r in range(len(Mem)):
        for c in range(len(Mem[0])):
           win.create_rectangle(c*w+b,r*l+b, (c+1)*w+b, (r+1)*l+b, fill = "white")

    #draw input ports
    for s in sources:
        win.create_rectangle(4, s[0]*l+b+l/3, 24, s[0]*l+b+l/3+10, fill = "grey")
        win.create_text(4+10, s[0]*l+b+l/3+5, font = ("purisa", 10), text =
str(sources.index(s)) )

    #draw output ports
    for o in outports:
        win.create_rectangle(len(Mem[0])*w+b+6, o[0]*l+b+l/3, len(Mem[0])*w+b+26,
o[0]*l+b+l/3+10, fill = "grey")
        win.create_text(len(Mem[0])*w+b+6+10, o[0]*l+b+l/3+5, font = ("purisa", 10),
text = str(outports.index(o)) )

    #draw mixers
    for m in mixers:
        if m.x == 0:
            win.create_oval(m.y*w+w/2+b+2, m.x*l+b/4-2, m.yo*w+w/2+b-2, m.x*l+l+b/4,
fill = "yellow")
            win.create_text(m.y*w+b+w, m.x*l+b/4+l/2, font = ("purisa", 10), text =
str(mixers.index(m)) )
            win.create_line(m.y*w+w/2+b, m.x*l+l/2+b/4-2, m.y*w+w/2+b, m.x*l+l+b/4+2,
fill = "#674c47", width = 3)
            win.create_line(m.yo*w+w/2+b, m.x*l+l/2+b/4-2, m.yo*w+w/2+b,
m.x*l+l+b/4+2, fill = "#674c47", width = 3)
        else:
            win.create_oval(m.y*w+w/2+b+2, (m.x+2)*l+b/2-2, m.yo*w+w/2+b-2,
(m.x+2)*l+l+b/2, fill = "yellow")
            win.create_text(m.y*w+b+w, (m.x+2)*l+b/2+l/2, font = ("purisa", 10), text
= str(mixers.index(m)) )
            win.create_line(m.y*w+w/2+b, (m.x+2)*l+b/2-4, m.y*w+w/2+b,
(m.x+2)*l+l/2+b/2, fill = "#674c47", width = 3)
            win.create_line(m.yo*w+w/2+b, (m.x+2)*l+b/2-4, m.yo*w+w/2+b,
(m.x+2)*l+l/2+b/2, fill = "#674c47", width = 3)

    #draw paths
    for i in paths:
```

```
        for cp in allPaths[i][2]:
            win.create_rectangle(cp[1]*w+b,cp[0]*l+b, (cp[1]+1)*w+b, (cp[0]+1)*l+b,
fill = "#ff8243")
            win.create_text(cp[1]*w+b+w/2, cp[0]*l+b+l/2, font = ("purisa", 9)), text
= str(allPaths[i][4]) + ',' + str(allPaths[i][4] +
allPaths[i][3][allPaths[i][2].index(cp)][0]) )

    master.mainloop()

def snapShot(t):
    global allPaths, mixers, sources, outports, Mem, storages
    master = Tk()
    win = Canvas(master, width = 1200, height = 750)
    win.pack()
    w = 40
    l = 20
    b = 30
    activeInT = []

    #draw valve matrix
    for r in range(len(Mem)):
        for c in range(len(Mem[0])):
           win.create_rectangle(c*w+b,r*l+b, (c+1)*w+b, (r+1)*l+b, fill = "white")

    #draw input ports
    for s in sources:
        win.create_rectangle(4, s[0]*l+b+l/3, 24, s[0]*l+b+l/3+10, fill = "grey")
        win.create_text(4+10, s[0]*l+b+l/3+5, font = ("purisa", 10), text =
str(sources.index(s)) )

    #draw output ports
    for o in outports:
        win.create_rectangle(len(Mem[0])*w+b+6, o[0]*l+b+l/3, len(Mem[0])*w+b+26,
o[0]*l+b+l/3+10, fill = "grey")
        win.create_text(len(Mem[0])*w+b+6+10, o[0]*l+b+l/3+5, font = ("purisa", 10),
text = str(outports.index(o)) )

    #draw mixers
    for m in mixers:
        if m.x == 0:
            win.create_oval(m.y*w+w/2+b+2, m.x*l+b/4-2, m.yo*w+w/2+b-2, m.x*l+l+b/4,
fill = "yellow")
            win.create_text(m.y*w+b+w, m.x*l+b/4+l/2, font = ("purisa", 10), text =
str(mixers.index(m)) )
            win.create_line(m.y*w+w/2+b, m.x*l+l/2+b/4-2, m.y*w+w/2+b, m.x*l+l+b/4+2,
fill = "#674c47", width = 3)
            win.create_line(m.yo*w+w/2+b, m.x*l+l/2+b/4-2, m.yo*w+w/2+b,
m.x*l+l+b/4+2, fill = "#674c47", width = 3)
        else:
            win.create_oval(m.y*w+w/2+b+2, (m.x+2)*l+b/2-2, m.yo*w+w/2+b-2,
(m.x+2)*l+l+b/2, fill = "yellow")
            win.create_text(m.y*w+b+w, (m.x+2)*l+b/2+l/2, font = ("purisa", 10), text
= str(mixers.index(m)) )
```

```
            win.create_line(m.y*w+w/2+b, (m.x+2)*l+b/2-4, m.y*w+w/2+b,
(m.x+2)*l+l/2+b/2, fill = "#674c47", width = 3)
            win.create_line(m.yo*w+w/2+b, (m.x+2)*l+b/2-4, m.yo*w+w/2+b,
(m.x+2)*l+l/2+b/2, fill = "#674c47", width = 3)

    #draw paths
    for i in range(len(allPaths)):
        if  t >= allPaths[i][4]:
            for cp in allPaths[i][2]:
                if allPaths[i][3][allPaths[i][2].index(cp)][1] == 0 and t <=
allPaths[i][4] + allPaths[i][3][allPaths[i][2].index(cp)][0] - 1:
                    if activeInT.count(cp) > 0:
                        print("snapShot: error! node:", cp, "path"+str(i))
                    else:
                        activeInT.append(cp)
                    win.create_rectangle(cp[1]*w+b,cp[0]*l+b, (cp[1]+1)*w+b,
(cp[0]+1)*l+b, fill = "#ff0243")
                    win.create_text(cp[1]*w+b+w/2, cp[0]*l+b+l/2, font = ("purisa",
9), text = str(i) )

    #draw storage cells
    for s in storages:
        if t >= s.start and t < s.end:
            for cp in s.path:
                win.create_rectangle(cp[1]*w+b,cp[0]*l+b, (cp[1]+1)*w+b,
(cp[0]+1)*l+b, fill = "#ff0043")
                win.create_text(cp[1]*w+b+w/2, cp[0]*l+b+l/2, font = ("purisa", 9),
text = "s" + str(storages.index(s)) )

    master.mainloop()

def checkCrossings(start, end):
    global allPaths
    for t in range(start, end):
        activeInT = []
        for i in range(len(allPaths)):
            if  t >= allPaths[i][4]:
                for cp in allPaths[i][2]:
                    if t <= allPaths[i][4] +
allPaths[i][3][allPaths[i][2].index(cp)][0] - 1:
                        if activeInT.count(cp) > 0:
                            print("checkCrossing: error! node:", cp, "path"+str(i))
                        else:
                            activeInT.append(cp)

def scheduleOnce(files = ['operations_one.txt','platform_one.txt'], nMixers = 0,
doPrintQInfo = False, doPrintPaths = False):
    #nMixers = number of mixers, print qInfo if doPrintQInfo = True, print paths if
doPrintPaths = True
    global tEnd, acLim
    opFile = files[0]
    platformFile = files[1]
    start_time = time.time()
```

```
    readOpFile(opFile)#read operations
    #printOps()
    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10 = readConstraintFile(platformFile)#read
platform: rows, columns, mixers, sources, outports
    tEnd = a6
    acLim = a7
    if nMixers == 0:
        nMixers = a3
    init(a1, a2, nMixers, a4, a5)
    printPlatform(a1, a2, nMixers, a4, a5, a6, a7)
    plan()
    if doPrintQInfo == True:
        printQInfo()
    printMixers()
    if doPrintPaths == True:
        printPaths()
    printResult()
    runTime = int((time.time()-start_time)*100)/100
    print('\n' + "*** RUN TIME =", runTime, "s ***")

def findMinCost(files = ['operations_one.txt','platform_one.txt',
'results_one.txt']):
    global t, tEnd, acLim
    print('\n'+ "*************************Test Case: " + files[3] +
"**********************")
    opFile = files[0]
    platformFile = files[1]
    resultFile = open(files[2], "r+")
    readOpFile(opFile)#read operations
    #read platform: rows, columns, mixers, sources, outports, maxTime, maxAct,
costCoefficients
    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10 = readConstraintFile(platformFile)

    tEnd = a6
    acLim = a7
    table = []

    optRows = a1
    optCols = a2
    optMixers = a3
    optTime = a6
    optVA = a7
    optMU = 0
    opPath = []

    CTime = a8
    CMixer = a9
    CSize = a10

    cost = CTime*optTime + CMixer*optMixers + CSize*(optRows*optCols)
    minCost = cost
    t = 0
```

```
    #printOps()
    printPlatform(a1,a2,a3,a4,a5,a6,a7)
    print("Cost = " + str(CTime) + "*Time + " + str(CMixer) + "*nMixers + " +
str(CSize) + "*Size")
    start_time = time.time()
    print("start...")
    for nmixers in range(1, a3 + 1):
        #print("nmixers: ", nmixers)
        readOpFile(opFile)
        init(a1 , a2, a3, a4, a5)
        plan()
        ii , jj = detectUsed()
        ii, jj = len(ii), len(jj)
        for rows in range(max(a4, a5) + 2, max(max(a4, a5) + 3, ii)):
            #print(" rows: ", rows)
            for cols in range((nmixers//2 + nmixers%2)*2 + 2, max((nmixers//2 +
nmixers%2)*2 + 2 + 1, jj)):
                #print("     cols: ", cols)
                readOpFile(opFile)
                init(rows , cols, nmixers, a4, a5)
                plan()
                if t < tEnd:
                    maxValveActuation = maxActuation()[0]
                    maxMixerUsage = maxUsedMixer()[0]
                    table.append((nmixers, rows, cols, t, maxValveActuation,
maxMixerUsage))
                    cost = CTime*t + CMixer*nmixers + CSize*(rows*cols)
                    if cost < minCost:
                        minCost = cost
                        optRows = rows
                        optCols = cols
                        optMixers = nmixers
                        optTime = t
                        optVA = maxValveActuation
                        optMU = maxMixerUsage
                        opPath = copy.copy(allPaths)
                else:
                    break
    #plan optimal solution to have the otimal arrays at the end
    readOpFile(opFile)
    init(optRows , optCols, optMixers, a4, a5)
    plan()

    run_time = int((time.time() - start_time)*100)/100
    runTimeSec = run_time%60
    runTimeMin = int(run_time//60)
    print('\n' + "*** RUN TIME =", runTimeMin, "m", runTimeSec, "s ***")
    print("Writing results to file..." + '\n')
    resultFile.seek(0)
    resultFile.truncate()
    resultFile.write("info: ""nMixer, " + "Row, " + "Col, " + "Time, " + "ValveAct, "
+ "MixerUsage." + '\n')
    resultFile.write('\n' + "nM" + '\t' + "Row" + '\t' + "Col" + '\t' + "T" + '\t' +
"VA" + '\t' + "MU")
    for line in table:
```

```
        resultFile.write('\n' + str(line[0]) + '\t' + str(line[1]) + '\t' +
str(line[2]) + '\t' + str(line[3]) + '\t' + str(line[4]) + '\t' + str(line[5]))

    resultFile.close()
    print("Solution:")
    print('\t' + "Cost =", minCost)
    print('\t' + "Number of mixers = ", optMixers)
    print('\t' + "Size of the valve matrix (rows * columns) = ", optRows , " * ",
optCols)
    print('\t' + "Total time steps of execution = ", optTime)
    print('\t' + "Maximum valve actuation = ", optVA)
    print('\t' + "Maximum mixer usage = ", optMU)
    print('\n' + "Finish." + '\n' +
"**********************************************************************")

#Test Cases
PCR = ('operations_PCRmix.txt', 'platform_PCRmix.txt', 'results_PCR.txt', "PCR Mixing
Stage")
mux_diagnosis = ('operations_multiplexed_diagnosis.txt',
'platform_multiplexed_diagnosis.txt', 'results_mux_diagnosis.txt', "Multiplexed
Diagnosis")
protein_assay = ('operations_protein_assay_opt.txt',
'platform_protein_assay_opt.txt', 'results_protein_assay.txt', "Protein Assay")

#findMinCost(PCR)
#findMinCost(mux_diagnosis)
#findMinCost(protein_assay)
```

II: input files for test cases

PCR

Operations:

```
name input1 input2 volume1 volume2 start duration destinations(separated by ',')(if outport: o2:5 = outport number 2, volume = 5)
d0 s0 s1 2 2 0 80 d2,o0:2
d1 s2 s3 2 2 0 40 d2,o1:2
d2 d0 d1 2 2 1 48 d6,o4:2
d3 s4 s5 2 2 0 40 d5,o2:2
d4 s6 s7 2 2 0 40 d5,o3:2
d5 d3 d4 2 2 1 80 d6,o5:2
d6 d2 d5 2 2 2 24 o6:4
```

Constraints:

```
valveMatrixRows valveMatrixColumns #mixers #sources #outports maxDuration maxActuations CostCoefficients(time, mixer, size)
20 20 7 8 7 300 100 1 0 0
```

Multiplexed diagnostics

Operations:

```
name(from 1) input1 input2 volume1 volume2 start duration destinations(separated by ',')
d0 s0 s1 1 1 0 50 o0:2
d1 s2 s3 1 1 0 50 o1:2
d2 s4 s5 1 1 0 50 o2:2
d3 s6 s7 1 1 0 30 o3:2
d4 s8 s9 1 1 0 30 o4:2
d5 s10 s11 1 1 0 30 o5:2
```

Constraints:

```
valveMatrixRows valveMatrixColumns #mixers #sources #outports maxDuration maxActuations CostCoefficients(time, mixer, size)
30 30 6 12 6 300 100 1 0 0
```

Protein assay

Operations:

```
name(from 1) input1 input2 volume1 volume2 start duration destinations(separated by ',')
d0 s0 s1 1 1 0 50 d1,d2
```

```
d1 s2 d0 1 1 1 50 d3,d4
d2 s3 d0 1 1 1 50 d5,d6
d3 s4 d1 1 1 2 50 d7,d8
d4 s5 d1 1 1 2 50 d9,d10
d5 s6 d2 1 1 2 50 d11,d12
d6 s7 d2 1 1 2 50 d13,d14
d7 s0 d3 1 1 3 50 d15,o0:1
d8 s1 d3 1 1 3 50 d16,o1:1
d9 s2 d4 1 1 3 50 d17,o2:1
d10 s3 d4 1 1 3 50 d18,o3:1
d11 s4 d5 1 1 3 50 d19,o4:1
d12 s5 d5 1 1 3 50 d20,o5:1
d13 s6 d6 1 1 3 50 d21,o6:1
d14 s7 d6 1 1 3 50 d22,o7:1
d15 s0 d7 1 1 4 50 d23,o0:1
d16 s1 d8 1 1 4 50 d24,o1:1
d17 s2 d9 1 1 4 50 d25,o2:1
d18 s3 d10 1 1 4 50 d26,o3:1
d19 s4 d11 1 1 4 50 d27,o4:1
d20 s5 d12 1 1 4 50 d28,o5:1
d21 s6 d13 1 1 4 50 d29,o6:1
d22 s7 d14 1 1 4 50 d30,o7:1
d23 s0 d15 1 1 5 50 d31,o0:1
d24 s1 d16 1 1 5 50 d32,o1:1
d25 s2 d17 1 1 5 50 d33,o2:1
d26 s3 d18 1 1 5 50 d34,o3:1
d27 s4 d19 1 1 5 50 d35,o4:1
d28 s5 d20 1 1 5 50 d36,o5:1
d29 s6 d21 1 1 5 50 d37,o6:1
d30 s7 d22 1 1 5 50 d38,o7:1
d31 s0 d23 1 1 6 50 d39,o0:1
d32 s1 d24 1 1 6 50 d40,o1:1
d33 s2 d25 1 1 6 50 d41,o2:1
d34 s3 d26 1 1 6 50 d42,o3:1
d35 s4 d27 1 1 6 50 d43,o4:1
d36 s5 d28 1 1 6 50 d44,o5:1
d37 s6 d29 1 1 6 50 d45,o6:1
d38 s7 d30 1 1 6 50 d46,o7:1
d39 s0 d31 1 1 7 30 o0:2
d40 s1 d32 1 1 7 30 o1:2
d41 s2 d33 1 1 7 30 o2:2
```

```
d42 s3 d34 1 1 7 30 o3:2
d43 s4 d35 1 1 7 30 o4:2
d44 s5 d36 1 1 7 30 o5:2
d45 s6 d37 1 1 7 30 o6:2
d46 s7 d38 1 1 7 30 o7:2
```

Constraints:

```
valveMatrixRows valveMatrixColumns #mixers #sources #outports
maxDuration maxActuations CostCoefficients(time, mixer, size)
30 30 20 8 8 1000 100 1 0 0
```

III: text output samples

Following samples of text output are provided using PCR test case.

Output of cost minimization:

```
******************Test Case: PCR Mixing Stage***************
Specifications:   rows =   11 , columns =   7 , #mixers =   4 ,
#sources =   8 , #outports =   7 , max. duration =   300 , max.
actuations =  100
Cost = 1*Time + 0*nMixers + 0*Size
start...

*** RUN TIME = 0 m 0.55 s ***
Writing results to file...

Solution:
     Cost = 212
     Number of mixers =  4
     Size of the valve matrix (rows * columns) =  10  *  6
     Total time steps of execution =  212
     Maximum valve actuation =  22
     Maximum mixer usage =  3

Finish.
************************************************************
```

Output of printPaths() function:

```
////////Paths////////
path0, source0 d0
path1, source2 d1
path2, source1 storage0
path3, source3 storage1
path4, source4 d3
path5, storage0 d0
path6, source5 storage2
path7, source6 d4
path8, storage1 d1
path9, source7 storage3
path10, storage2 d3
path11, storage3 d4
path12, d1 o1
path13, d3 o2
path14, d4 o3
```

```
path15, op3 storage4
path16, storage4 d5
path17, d0 o0
path18, op1 d2
path19, d2 o4
path20, d5 o5
path21, op2 d6
path22, d6 o6
```

Output of printQInfo() function:

```
/////////////// t = 0
 ready: [3, 4] reserved: [] act1: [0, 1] act2: [] finished: []
removed: []

/////////////// t = 3
 ready: [4] reserved: [] act1: [0, 1, 3] act2: [] finished: []
removed: []

/////////////// t = 4
 ready: [4] reserved: [] act1: [1, 3] act2: [0] finished: []
removed: []

/////////////// t = 7
 ready: [] reserved: [] act1: [1, 3, 4] act2: [0] finished: []
removed: []

/////////////// t = 9
 ready: [] reserved: [] act1: [3, 4] act2: [0, 1] finished: []
removed: []

/////////////// t = 16
 ready: [] reserved: [] act1: [4] act2: [0, 1, 3] finished: []
removed: []

/////////////// t = 22
 ready: [] reserved: [] act1: [] act2: [0, 1, 3, 4] finished: []
removed: []

/////////////// t = 52
 ready: [] reserved: [] act1: [] act2: [0, 3, 4] finished: [1]
removed: []
```

```
////////////// t = 63
 ready: [] reserved: [] act1: [] act2: [0, 4] finished: [1, 3]
removed: []

////////////// t = 66
 ready: [] reserved: [5] act1: [] act2: [0] finished: [1, 3, 4]
removed: []

////////////// t = 69
 ready: [] reserved: [] act1: [5] act2: [0] finished: [1, 3]
removed: [4]

////////////// t = 71
 ready: [] reserved: [] act1: [5] act2: [0] finished: [1]
removed: [4, 3]

////////////// t = 78
 ready: [] reserved: [] act1: [] act2: [0, 5] finished: [1]
removed: [4, 3]

////////////// t = 88
 ready: [] reserved: [2] act1: [] act2: [5] finished: [1, 0]
removed: [4, 3]

////////////// t = 91
 ready: [] reserved: [] act1: [] act2: [5, 2] finished: [1]
removed: [4, 3, 0]

////////////// t = 93
 ready: [] reserved: [] act1: [] act2: [5, 2] finished: []
removed: [4, 3, 0, 1]

////////////// t = 151
 ready: [] reserved: [] act1: [] act2: [5] finished: [2]
removed: [4, 3, 0, 1]

////////////// t = 169
 ready: [] reserved: [6] act1: [] act2: [] finished: [2, 5]
removed: [4, 3, 0, 1]
```

```
////////////// t = 172
 ready:  []  reserved:  []  act1:  []  act2:  [6]  finished:  [2]
removed: [4, 3, 0, 1, 5]

////////////// t = 174
 ready: [] reserved: [] act1: [] act2: [6] finished: [] removed:
[4, 3, 0, 1, 5, 2]

////////////// t = 199
 ready: [] reserved: [] act1: [] act2: [] finished: [6] removed:
[4, 3, 0, 1, 5, 2]

////////////// t = 204
 ready: [] reserved: [] act1: [] act2: [] finished: [] removed:
[4, 3, 0, 1, 5, 2, 6]
```

Output of printMixers() function:

```
////////mixers////////
op0 -> mixer0
op1 -> mixer1
op2 -> mixer0
op3 -> mixer3
op4 -> mixer2
op5 -> mixer2
op6 -> mixer2
```

IV: graphical output functions and samples

Following samples of graphical output are provided using PCR test case.

drawPaths()

drawPath function can be used to see all the paths generated during scheduling at one view. Since paths overlap in different points of time, some are hidden under the others in the picture.

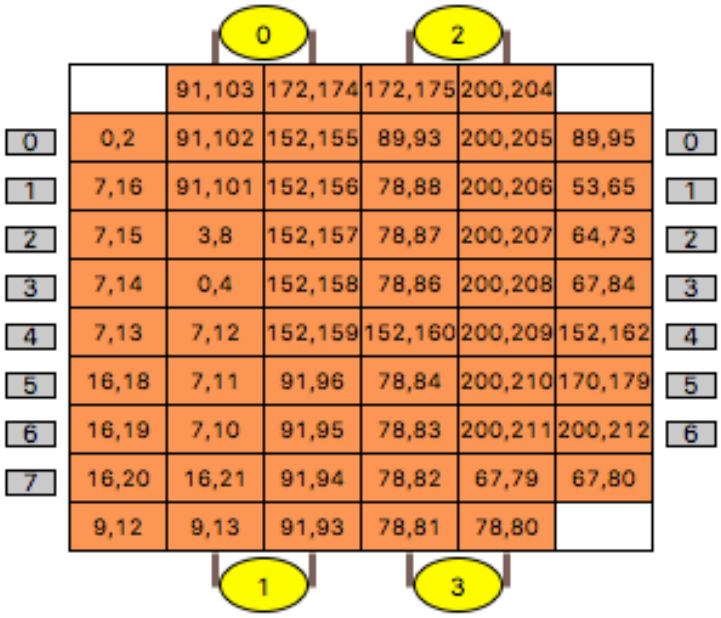


Figure 33

drawPaths([0,7])

This function can also be used to view specific path or paths separately. This can be done by listing desired paths in the input of the function. In this example, only path 0 and 7 are shown on the picture.

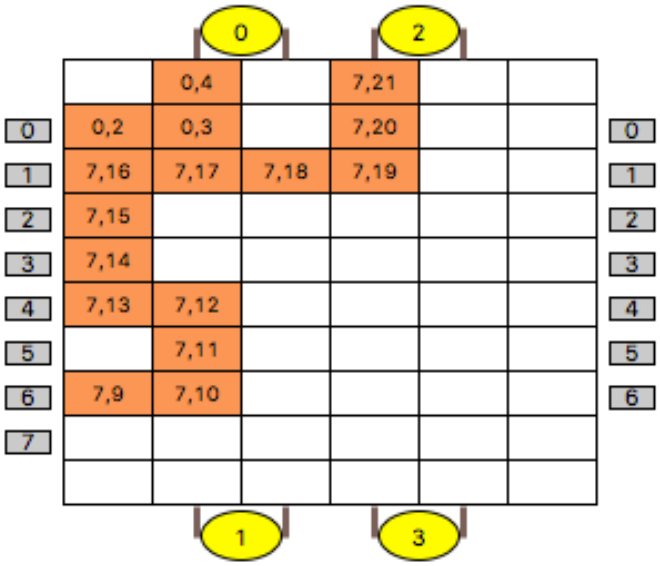


Figure 34

snapShot(0)

This function can be used to view the state of valve matrix at a certain point of time. In this picture, all the active paths and storage units are illustrated. Following figure shows the state of the matrix at time 0.

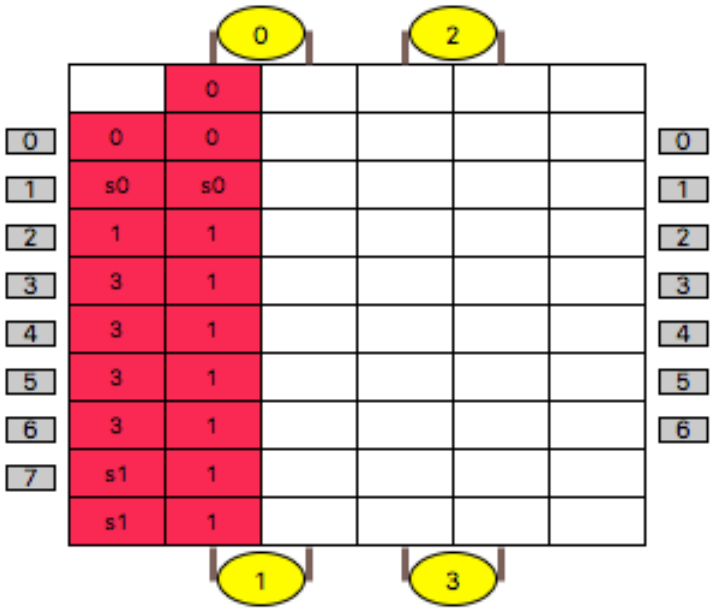


Figure 35