Routing algorithm for the control layer of flow-based biochips

Pathfinder

Tinna Schmidt Rasmussen

103440

Technical University of Denmark

Department of Informatics and Mathematical Modelling (IMM)

Supervisor: Paul Pop

26th June 2014

Abstract

Routing channels on the control layer of a flow-based biochip is a troublesome problem standing in the way of biochips living up to their real potential. A manual design approach is labour intensive and error prone as the amount of channels that needs to be routed can be very high. Therefore algorithms that can do most of the work in the design process is very helpful.

This paper contains theory about a routing algorithm for the control layer of flow-based biochips called Pathfinder. I discuss the implementation of the algorithm in Java and looks at how the algorithm can improve the routing of the control channels.

Contents

Introduction

As an alternative to the conventional biochemical laboratories, different biochips have emerged over the past couple of years. Compared to traditional laboratory work, micro fluidic biochips can decrease the consumption of reagent used. Such biochips are already in use in many areas. One of the main alternatives is the continuous flow-based micro fluidic biochip. The technology behind the creation of continuous flow-based biochips is microfluidic Very Large Scale Integration, or mVLSI for short.

Continuous flow-based microfluidic biochips are customized biochips designed for a given task. The task is carried out by manipulating liquid as a continuous flow from an inlet through closed flow channels to components placed on the biochip. Both the flow channels and the components are placed on the layer of the biochip called the flow layer. The biochip contains several valves that can be opened or closed in order to guide the liquids in the right direction. The valves are controlled by the control layer using air pressure. A biochip can have just a single control layer or two, one at the top and one at the bottom of the flow layer.

Small biochips have the potential to revolutionize bioscience. However, there is an obstacle that needs to be overcome in order for the field to really advance move forward; the design process. Even though some automated tools are emerging, the design and fabrication process is still mostly manual. This is usually done by manually placing components on the flow layer, manually drawing lines representing the flow channels as well as placing the valves and drawing the lines representing the control channels. The manual design process is both labour intensive and often error prone. In order to design the infrastructure of the biochip, an extensive knowledge of the application of the chip being design is required.

In this paper I will describe different algorithms, which may help easing the customized labour-intensive design process of the control layer of a flow-based biochip. I will analyse and implement one of these algorithms, called Pathfinder and discuss its advantages and how it can be improved.

Preliminaries and theory background

The design process

The design process is divided into a number of different steps.

This paper focuses on the step with routing of the control channels, but will also touch the subject of routing the flow channels.

Routing of the channels

Routing of channels on the flow layer and control layer are similar. In both cases the channels needs to be routed from a source to a sink, while avoiding obstacles such as other channels.

On the flow layer the channels needs to go from a specific source (e.g. an output on one component) to a specific sink (e.g. an input on another component). The flow channels have to move around components. It is possible for flow channels to intersect with one another, which can either be done to shorten the total length of flow channels or because it is necessary in order to be able to route the flow channel at all. This is done by introducing a switch, which needs to be controlled with valves. There are two kinds of switches, the T switch and the X switch, which can be seen in Figure 1.

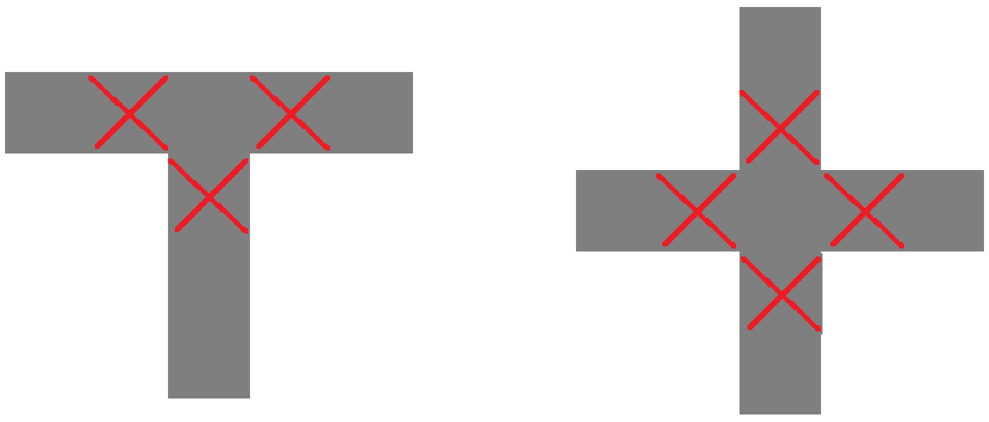


Figure 1 showing an illustration of a T switch and an X switch on the flow layer

The flow channels intersect and needs valves, marked with a red cross, to open and close them in order to guide the liquid in the right direction. If possible it is better to have one X switch instead of two T switches. An example of two T switches that could be replaced by a single X switch is shown if Figure 2. This reduces the number of valves by 2. However, the length of the flow channel has to be taken into consideration and sometimes it might be preferred to have a shorter route length and introduce more valves.

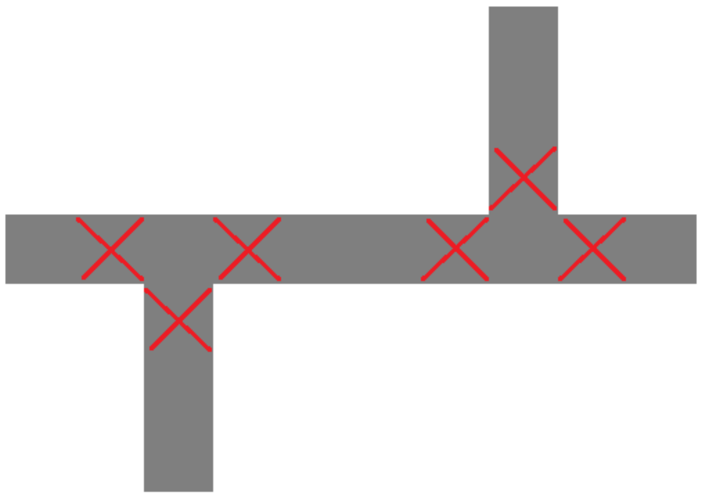


Figure 2 showing an example of two T switches which could be replaced by a single X switch

On the control layer the channels need to go from a source, being a valve, to a sink, being the inlets for the air. A specific valve doesn't necessary need to be connected to a specific inlet. The control channels have to avoid running into other control channels and valves, but they are not affected by the components or flow channels located on the flow layer. This means that the routing of the control channels only have to take the placement of the valves and air inlets into consideration. Control channels are not allowed to intersect, since there isn't a way to control a control channel the same way the flow channels can be controlled by the T and X switch. However it is possible that some valves can share the same inlet, in which case their control channels are allowed to intersect. This will be discussed further in the section "Improvements" under "Let valves share inlets".

Algorithms

There exist several algorithms that have been used to solve similar problems to the routing of the control channels on a biochip. Some of these make use of a rip up and retry method, where routes are ripped up if they conflict with other routes and then rerouted. The problem with this method is that it might not always be just the routes that are conflicting which needs to be ripped up in order to find a solution, and in this case ripping up conflicting routes and rerouting them won't always be able to solve the problem.

The pathfinder algorithm is created to route signals on FPGAs. However, it can be applied to routing of the control channels on a biochip as the problems are similar. The pathfinder's method is to let routes use the same nodes for a while, but eventually this node becomes more expensive to use. All of the routes keep being rerouted until a solution is found.

Analyses of the problem

The algorithm Pathfinder that is used in this paper is based on the algorithm Negotiated Congestion but with a few alterations[i]. This implementation of the algorithm considers all channels to be equally important and will be given the same advantages when routed. The pseudo code the implantation is based on is shown below.

Pseudo code

```
While shared resources exist (global router) [1]
    Loop over all signals i (signal router) [2]
            Rip up routing tree RTi [3]
            RTi <- si [4]
            Loop until all sinks tij have been found [5]
                Initialize priority queue PQ to RTi at cost of nodes [6]
                Loop until new tij is found [7]
                    Remove lowest cost node m from PQ [8]
                    Loop over fanouts n of node m [9]
                        Add n to PQ at cost cn + Pim [10]
                End [11]
            End [12]
            Loop over nodes n in path tij to si (backtrace) [13]
                Update cn [14]
                Add n to RTi [15]
            End [16]
        End [17]
    End [18]
End [19]
```

Explanation

The pathfinder consists of two parts: a global router and a signal router. The global router keeps track of what nodes/resources are used and by how many signals (channels) at once. The global router calls the signal router which routes one signal at a time using a breath first search to find the cheapest path from a source (valve) to a sink (inlet). Seeing as each valve only needs to be connected to one inlet, there is only one sink for each signal.

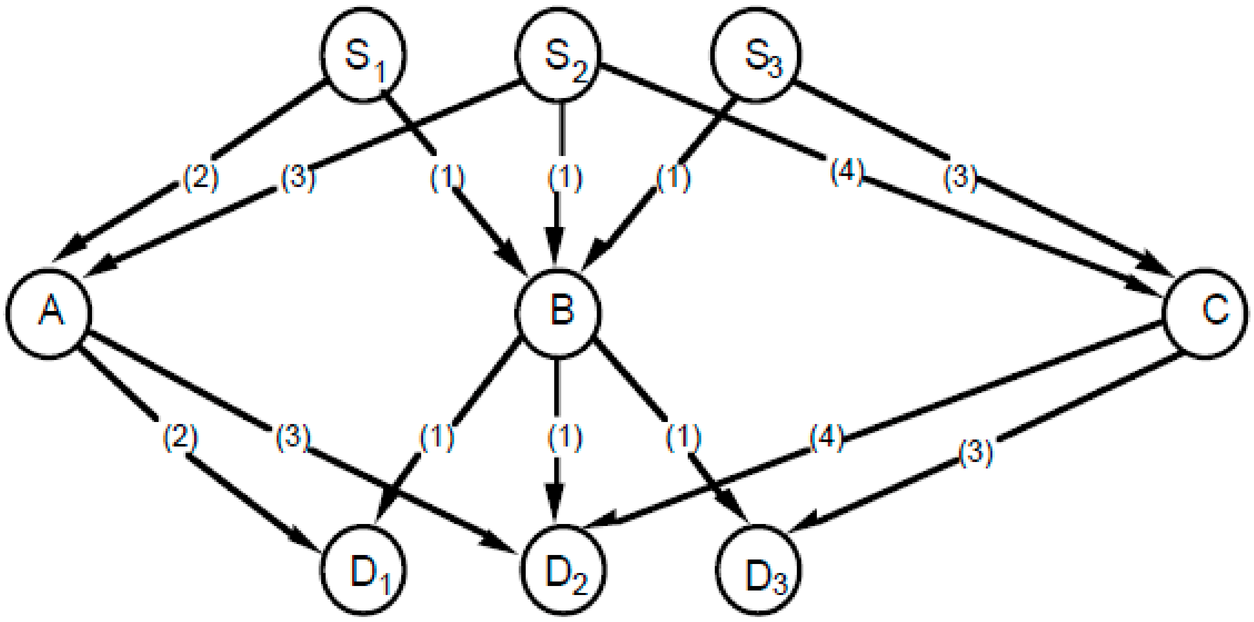
At first all the channels are routed so that they achieve the shortest path possible. However some channels might be using the same nodes in order for them to get their shortest possible path. Since control channels aren't allowed to intersect, this needs to be changed. This is done through several calls to the signal router that keeps rerouting all the channels until the global router sees that there are no more shared nodes.

In order for the signal router to know which nodes are used by more than one channel, a cost is introduced to each node. Whenever a node is used by one or more channels, the cost of using said node is increased. This means that nodes that are used by a lot of channels will be more expensive, and some of these channels will therefore be rerouted in another direction.

If a node is no longer used by any channels, the cost will slowly decrease so that the node might be appealing later on.

Examples with text

In this section I will explain how the algorithm works with a small example of first order congestion (borrowed from the Pathfinder paper [i]).



Assume we have 3 valves: S1, S2 and S3 as well as 3 inlets: D1, D2, D3. It is irrelevant what inlet a valve is connected to, but it is wanted to get the shortest total length of the channel paths. There are 3 paths available: A, B and C. All valves can go through B and reach any inlet, however, S1 can't reach path C and S3 can't reach path A. The arcs leading to and from the nodes are partial paths that the channels need to go through before reaching A, B and C. The length of these partial paths is shown in parentheses.

The pathfinder works by first finding the shortest path for all valves. This means that all channels runs through path B. Since B is shared by many channels at once, its cost increases and it slowly gets more and more expensive. Eventually S1 will see that the path through A is cheaper and use that path instead. Later S3 will see that the path through C is cheaper.

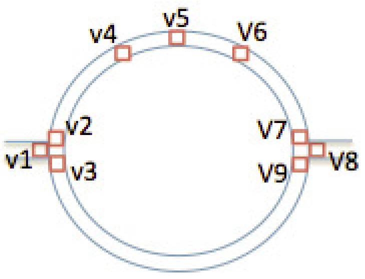
It is important that the cost doesn't increase too fast. In the above example, if the cost in increased by 1 at each iteration, that's how the channels will be routed. However, if the cost is increased by 10 each iteration then both S1 and S2 will choose to go through A and S3 will go through C. Then A will be overloaded and the channels need to be rerouted again.

Placement of valves

All the placements are based on the paper "BioChip Simulater Components Design" by Morten Schmidt[ii] and the sizes of the components are based on a library file[iii].

Mixer

The component called mixer is used to mix two fluids together. The mixed fluid can then be used in other components. The mixing process is controlled by opening and closing valves in a certain order. The mixer component contains 9 valves in total.

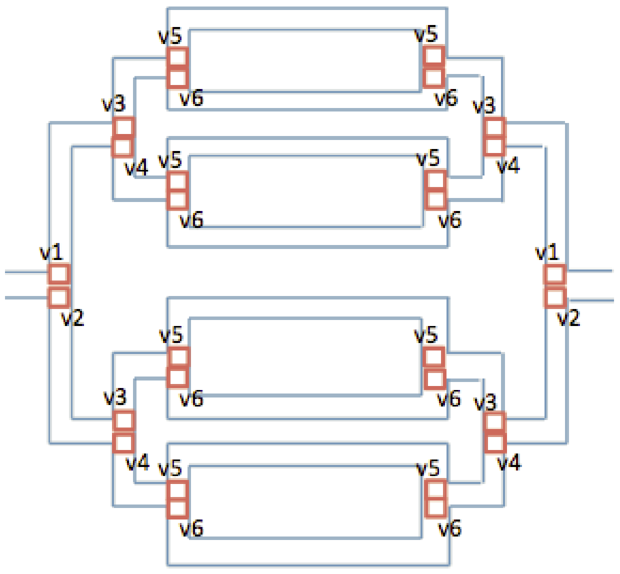


The mixer component has the size 30 x 30 in the XML files. The 9 valves are placed at the following coordinates relative to the top left corner of the mixer component:

V1: (0, 15)	V2: (2, 13)	V3: (2, 17)
V4: (5, 2)	V5: (15, 0)	V6: (24, 2)
V7: (27, 13)	V8: (29, 15)	V9: (27, 17)

Storage

The component called storage is used for storage of fluids on the biochip. These stored fluids can be used later on. The transportation of fluids inside the storage is controlled by opening and closing valves. The storage component contains 28 valves; however, some of these valves are always open or closed at the same time. This means that some of these valves can be "combined" through local routing and thereby share the same inlet source. This means that there are in fact only 6 distinct valves that need to be routed on the biochip.

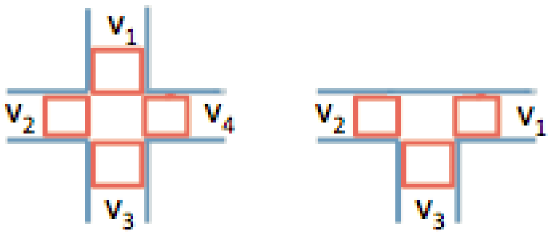


The storage component has the size 90 x 30 in the XML files. The 6 distinct valves are placed at the following coordinates relative to the top left corner of the mixer component as (0, 0):

V1: (0, 0)	V2: (0, 29)	V3: (45, 0)
V4: (45, 29)	V5: (89, 0)	V6: (89, 29)

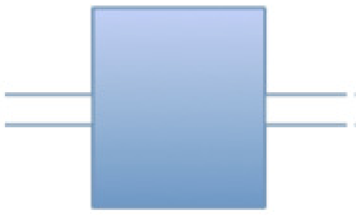
These valves are spaced out over the size of the component, so that they are all at the edge of the component.

Switch



There are two types of switches as described in the section “Preliminaries and theory background” under “Routing of channels”: the T switch and the X switch. The size of a valve is 100 µm, which mean that in the unit used in the XML files would have to be changed and be done more precisely in order to incorporate switches properly, since 1 unit in the XML files at the moment are 150 µm.

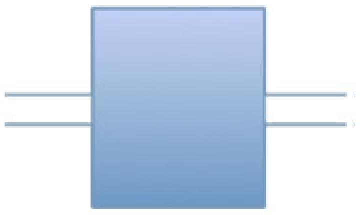
Detector



The component called detector is able to use a detect operation on the biochip for a specified amount of time. The detector component has no internal valves but it has been decided that a valve should be placed at the input and output of the detector.

The detector component has the size 20 x 20 in the XML files. In order to place the valves at the input and output, a valve should be placed at the point (0, 10) and (19, 10), relative to the top left corner of the detector component.

Heater

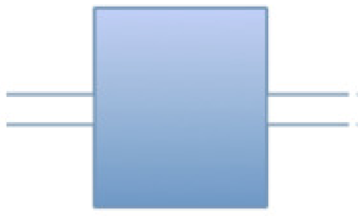


The component called heater can heat up liquid for a specified amount of time. The heater component has no internal valves but it has been decided that a valve should be placed at the input and output of the heater.

The heater component has the size 40 x 15 in the XML files. In order to place valves at the input and output, a valve should be placed at the point (0, 7) and (39, 7), relative to the top left corner of the heater component.

Filter

The component called filter is able to use a filter operation on liquid on the biochip. The filter component has no internal valves but it has been decided that a valve should be placed at the input and output of the filter.



The filter component has size 120 x 30 in the XML files. In order to place valves at the input and output, a valve should be placed at the point (0, 15) and (119, 15), relative to the top left corner of the filter component.

Rotation of components

In order to rotate the components on the biochip, a rotate function is introduced. The rotate function is divided into two methods: rotX and rotY, who return the x-coordinate and y-coordinate after rotation respectively.

The rotate methods takes as input the original coordinates of the valve (oldX and oldY), the angle (which depends on the orientation of the component) and the coordinates of the top left corner of the component (here called xCenter and yCenter, since this point will be the center of the rotation).

The formulas for calculating the rotated coordinates are:

$$rotX = (xCenter + \cos(Angle) * (oldX - xCenter) - \cos(Angle) * (oldY - yCenter)$$

$$rotY = (yCenter + \sin(Angle) * (oldX - xCenter) + \cos(Angle) * (oldY - yCenter)$$

The components are rotated counter-clockwise. The orientation 0 gives the angle 0, the orientation 1 gives the angle$\pi/2$, the orientation 2 gives the angle $2 * \pi/2$ and the orientation 3 gives the angle $3 * \pi/2$ in radians. Those are the only 4 possible rotations.

When rotating the component it is possible that a valve will get a new coordinate that isn't a real number. However, the coordinates are integers meaning that any decimals will be cut off.

Implementation

In order to solve the main problem with routing the control channels, it is necessary to not only implement the algorithm itself, but also implement a way for the program to read and understand input files as well as displaying the result in an easily accessible manner. In the following I will describe the overall structure of the program as well as go into detail and describe how the program reads input, runs the algorithm and displays the result.

The overall structure

The program consists of the following classes: Driver.java, Grid.java, Listener.java, Node.java, RoutingAlgorithm.java, SortedNodeList.java, View.java.

Class: Driver

The driver contains the method main(String[]) which creates a new driver object. From here the method readXML(String) is called, which takes the name of an XML file as input. How the reading of XML files works is explained under "Reading XML files". The information from the XML file is stored in an two-dimensional array called ArcComponents.
When the XML file is read, a frame to display the biochip is created by creating a new View object.

Lastly a new RoutingAlgorithm object is created and given the view and grid as input.

Class: Grid

The class grid is responsible for creating a grid of nodes that represent the biochip. The grid is initialized with the method initialize(), that first create the grid of width*height nodes. For each node, the node to the left, right, top and bottom are added to its neighbour list, except for the nodes that are located at the edge of the grid, in which case not all 4 neighbours are added.

Next the method placeValvesAndInlets() are called. The two-dimensional array ArcComponents created in the driver class is used to place the valves. ArcCompnents has the following information about each component i:

- ArcComponents[i][0] – The type of the component, indicated by a number.
- ArcComponents[i][1] – The x coordinate for the top left corner of the component.
- ArcComponents[i][2] – The y coordinate for the top left corner of the component.
- ArcComponents[i][3] – The orientation of the component.

A switch is used to call methods place1(), place2() and so on, depending on the component type. Each of these methods takes the coordinates and orientation as input.

Type 1 – Mixer

The mixer component contains 9 valves that are to be placed as described under "Analysis of the problem".

Type 2 – Storage

The storage component contains 6 valves that are to be placed as described under "Analysis of the problem".

Type 3 and 4 – Switches

The type with number 3 and 4 indicates switches that are used when flow channels on the flow layer intersect. Number 3 is a switch with 3 valves, called a T-switch and number 4 is a switch with 4 valves, called an X-switch.

Whereas placement information of all the components should be given as the placement of the top left corner, the placement information of the switches should be given as the placement of the centre of the switch. The methods place3() and place4() will then place the valves around the switch. For an X-switch the orientation is irrelevant since a valve will be placed on all 4 sides. For a T-switch, however, the orientation is important. Orientation 0 is the standard orientation and is implemented as a valve to the left, right and bottom (to that it simulates an actual T-shape). Orientation 1 is the entire T turned 90 degrees counter-clockwise, meaning a valve is to be placed to the top, bottom and right and so on.

Type 5, 6 and 7 – Detector, Heater and Filter

These 3 components do not contain any inner valves, but it has been decided to put a valve at the input and output of the components. This means that each of the 3 components has 2 valves each that are to be placed as described under "Analysis of the problem".

Case 8 and 9 – Input and Output

These components do not need any valves. They are read into the program from the XML files, but they have no effect on the placement of valves.

More components

Other components can easily be added to the program. All that is needed is to create a method in the file Grid that places the valves relative to the components top left corner. In the file Driver the name of the component needs to be connected to a number for ArcComponentType.

Class: Listener

The class Listener is used to update the time in the GUI but has no effect on the algorithm itself.

Class: Node

The class Node create the Node object that is used to represent a single spot in the grid. The Node class contains the following information about a Node:

- x, y – the coordinates of the node in the grid
- isStart – A Boolean indicating whether the node is a valve (true) or not (false)
- isGoal – A Boolean indicating whether the node is an inlet (true) or not (false)
- name – The coordinates written as a string. It is used for printing to the console
- subACost – The subtotal cost of the node, calculated by how many channels are using the node and for how long the node has been used.
- curACost – The number of channels currently occupying the node. It is used to check if there are any shared resources. If curACost for all nodes are less than or equal to 1, no channels use the same node.
- totalMinCost – The total minimum cost of the node, meaning it's subACost plus the cost from the valve to the node.

Class: RoutingAlgorithm

The class RoutingAlgorithm contains a few helping methods as well as the method that contains the overall implementation of the algorithm, called runAlgorithm(). Furthermore the class contains methods that are used to collect the result of the program and print it to the console.

Below I will briefly describe the usage of the various methods.

- **runAlgorithm()** –
  This method contains the overall algorithm for routing the channels. It will be described in detail under "The Pathfinder algorithm" later in this section.
- **routeSignal1()** –
  This method takes a start node (valve) as input and routes the channel to a goal node (inlet). This is done with a breath first search. It initialises the start nodes totalMinCost to 0 and adds it to a queue, sorted by cost. As long as the queue isn't empty, the first node is the queue it extracted. If the node is a goal node, then the method getShortestPath(node) is called. Otherwise the node is expanded and the neighbours are investigated and added to the queue.

- **routeSignal2()** –
  This method is similar to routeSignal1(), but there is an important difference. The method routeSignal1() didn’t take the subACost into account. This meant that routeSignal1() found the cheapest possible path in terms of distance. The method routeSignal2(), on the other hand, takes the cost of a node into account so that if a node is really expensive it might prefer to take a longer (but cheaper) path around it.
- **getShortestPathTo(Node)** –
  This method takes a goal node (inlet) as input and then backtracks the path, looking at the previous node all the time until there are no more previous nodes. This will be when a start node (valve) is reached. This list of nodes are stored in an ArrayList and returned. During the backtracking of the route, which will now be the current channel from the given goal node to start node, the aCost is incremented, by calling that method in the Node class.
- **ripUp(ArrayList<Node>)** –
  This method takes an ArrayList of nodes as input. This ArrayList is a routing tree from a single valve to an inlet. The method visits every single node in the path and decrement the aCost, by calling that method in the Node class.

Below the methods that handle the collection of the result is described in detail.

- **collectResults()** –
  This method collects all the relevant information after the algorithm has terminated and prints them to the console. It also calls the methods printAllminRT() and printAllRT(). The information that is collected are:
    - Number of valves
    - Number of intersections on the first run
    - Number of minimum intersections (in case the algorithm doesn’t finish with a solution, then the minimum number of intersections might have been found on another iteration than the last one)
    - Number of intersections on the last run (this is the solution unless the algorithm was terminated because it reached a limit)
    - Number of reruns, which is the number the signal router of the algorithm is called
    - Running time
    - The length of each routed channel as well as the total route length

- **printAllRT()** – This method prints out all the channels in the last routing tree RT. If a solution is found then that is what is printed. If a solution isn't found because a limit is reached then the method prints the latest routing tree, which might not be the most optimal routing tree it has found. It is possible that it during tit's runtime has found another routing tree with fewer intersections than the one being printed.

Class: SortedNodeList

The class SortedNodeList creates an ArrayList with Nodes. The Nodes are sorted by their total minimum cost (subACost of the node + path cost from valve to the node) so that the node with the lowest total minimum cost is first in the list.

The SortedNodeList class contains methods to make it easy to retrieve information, such as size(), get(int), getList() and more. Furthermore the class contains methods to change the list, such as clear(), add(Node n), remove (Node n) and more.

Class: View

The class View is responsible for the visual representation of the biochip and the routing of the channels. It uses the Swing library to produce a window frame containing a field showing running time, reruns, number of intersections as well as the field containing the grid representing the biochip. It contains functions to update the labels, which can be called from other classes, as well as a function that returns the current time, used when the algorithm is finished.

The grid is made in the class Grid and the view is updated from the class Node, whenever a node changes is status, e.g. is set as a valve.

Reading XML files

The program needs information about valve placement on the control layer as well as the placement of inlets. The placement of the inlets depends on the biochip used and is built directly into the program. In my thesis it is assumed that all the inlets are placed on two opposing side of the biochip.

The placement of valves depends on the placement of components on the flow layer as well as the routing of the flow channels. This information is given to the program through an XML file.

The XML file should have the following structure:

```
<Architecture>
    <Size>
        <Width>[BIOCHIP WIDTH]</Width>
        <Height>[BIOCHIP HEIGHT]</Height>
    </Size>
    <ListOfArcComponents>
        <ArcComponentProperties>
            <ArcComponentType>[COMPONENT NAME]</ArcComponentType>
            <Position>
                <X>[COMPONENT X]</X>
                <Y>[COMPONENT Y]</Y>
            </Position>
            <Orientation>[COMPONENT ORIENTATION]</Orientation>
        </ArcComponentProperties>
    </ListOfArcComponents>
</Architecture>
```

First the program reads the [BIOCHIP WIDTH] and [BIOCHIP HEIGHT] in order to create the grid representing the biochip. The scaling unit is 150 um, however in my test on some benchmarks I have assumed the scaling 20 um, so the grid would be less detailed and therefore more manageable for my computer to handle. This means that the valves as well as the width of the control channels are larger than they would be on the actual biochip. Because of this my tests might not be able to find a solution if one exists, however, if a solution is found in my tests then that will be a possible solution to the actual biochip. This will be discussed more throughout in the section “Improvements”.

Next all the information about the components on the biochip is read into the program. For each component the following information is obtained:

[COMPONENT NAME] – The name of the component's type. This is used to determine how many valves should be placed in the grid and where, relative to the component's top left corner. Different component have different valve placement which will be described under "Placement of valves".

[COMPONENT X], [COMPONENT Y] – The placement of the component's top left corner on the biochip. This corner is used when turning the component relative to its orientation.

[COMPONENT ORIENTATION] – The orientation of the component on the biochip. Orientation 0 is the standard orientation. Orientation 1 is orientation 0 turned 90 degrees counter-clockwise, orientation 2 is orientation 1 turned 90 degrees counter-clockwise and so on.

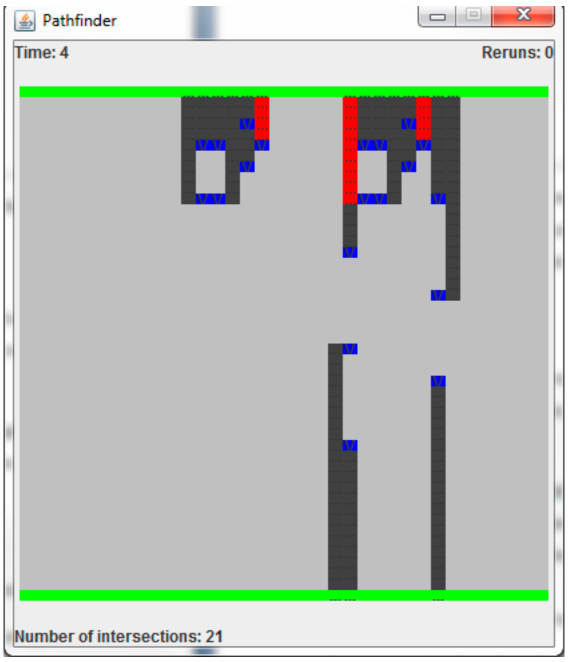
Placement of valves

From the XML file the program gains information about a component's type, top left corner placement and orientation. The placement of the valves is described in the section "Analysis of the problem" under "Placement of valves".

Displaying the result

When the program finishes, either because a solution is found or a limit is reached, the program outputs the result in the console. The output contains:

- Number of valves on the biochip
- Number of intersections in first try
- Number of intersections in the best solution (will be 0 if a working solution is found)
- Number of reruns (the number of time the loop "global router" is run)
- The running time
- The total length of all control channels
- The length of each control channel
- Each control channels route written as a list of coordinates from valve to inlet



Furthermore a GUI is implemented which shows a grid simulating the biochip. The biochip itself has the colour LIGHT_GRAY. The inlets have the colour GREEN and are placed at the top and bottom of the grid.

The valves have the colour BLUE. A control channel has the colour DARK_GRAY; unless control channels are intersecting in which case the spot of the intersection turns RED. The GUI also displays the run time, number of reruns and current number of intersections between channels.

The Pathfinder algorithm

The main implementation of the pathfinder algorithm is done is the class RoutingAlgorithm.java in the method runAlgorithm().

The ArrayList<ArrayList<Node>> RT contains all the current routings of the channels. The method routeSignal1() is called to find the shortest path for all channels and then adding them to the routing tree RT. The method checks if a solution is already found, because in that case there is no need for the algorithm in that routing.

Otherwise the pathfinder algorithm starts with the global router, and the signal router inside the loop. The signal router rips up a channel from RT, reroutes it now that the cost is updated and adds the new path to RT.

After the signal router is done for all channels, the global router checks if a solution is found and otherwise updates the cost of all nodes. If a limit is reached then the loop of the global router is terminated and the current result printed.

How and when to change the cost of a node

The cost for the nodes needs to be updated to that it corresponds to the number of channels currently using the nodes as well as for how long the node as been used.

To measure the cost I use 2 variables: curACost and subACost. The variable curACost is the number of channels currently using the node. This variable is increased by 1 whenever the method getShortestPathTo(node) is called, because then every node in the path has its curACost incremented with 1.The variable curACost is decremented whenever the method ripUp(ArrayList<Node>) is called. It is possible that that same route will be found when its rerouted, but that doesn't matter as the nodes will be incremented again through the getShortestPathTo(node) method if that's the case.

The variable subACost is trickier as it indicates how wanted the node is over time. Whenever curACost is incremented by one, subACost is incremented by a cost called costX. However, when curACost is decremented, subACost is only decremented with costX if no channels use the node anymore, meaning if curACost = 0. If more channels use the same node then the subACost won't be decremented, but it will be further incremented if the channels chooses the same route once again, thereby making the node even more expensive.

Every time all the signals have been routed one, the global router in the algorithm calls subCostDec(). This method checks all the nodes if they are currently occupied and if they aren't the subACost of the nodes are decremented by costY. This way, nodes that aren't used for a longer period of time will start getting cheaper again until eventually they might become attractive when a channel is rerouted and be used again.

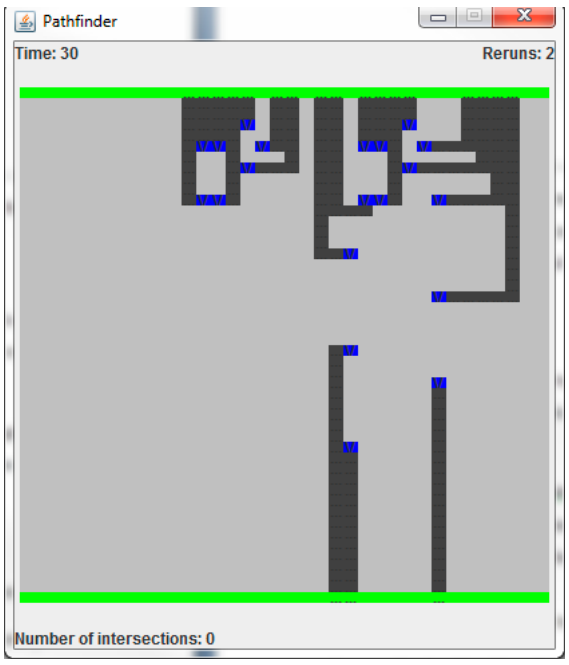
The impact of costX and costY

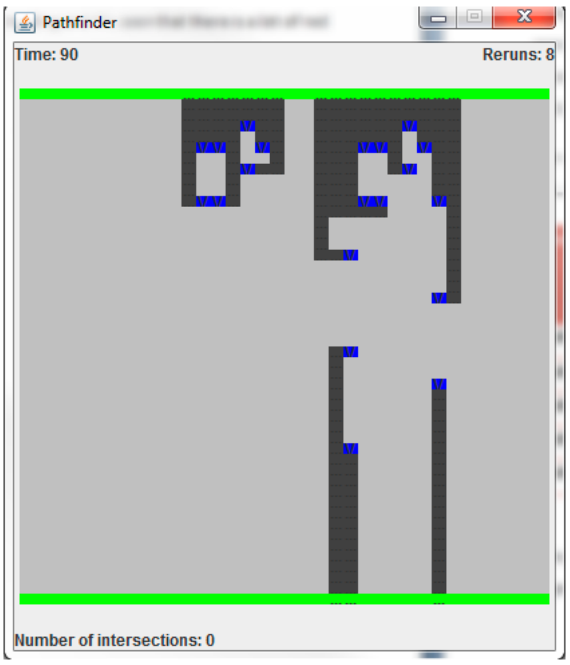
The subACost of a node is changed with two costs: costX and costY. The costX is the cost for the node for each channel who wants to use it. The costY on the other hand is the decrease in cost over time when the node is unused.

The costY should decrease slowly, as if it decreases too quickly, the routing algorithm will try to use the same paths too often and too soon after it becomes available.

The costX should also change slowly, as too big a change would mean that too many channels would avoid the given node.

Below is an example of the same test. To the left is the costX = 5 and to the right the costX = 1.





As seen a solution is found in both cases, however, they are different:

- The left example finds a solution after only 2 reruns. The total route length is 242.
- The right example finds a solution after 8 reruns. The total route length is 226.

This shows that if costX is larger, then the channels will spread out faster and if there's enough space as they are spreading out, a solution will be found faster. However, this solution might not be optimal as the high increasing costX makes the channels avoid nodes that might actually give the solution. It takes some time for the unused nodes to get a reasonable cost again if the costY is a lot smaller than costX, in this case 1.

It is a trade-off between running time and route length.

Evaluation

In order to run a test, the program needs to know which XML-file to use. This is specified in the class Driver.java in the first line of the driver constructor. In both the class Driver.java and Grid.java there's an integer constant called resize. This constant is used to resize the grid, making the entire more or less precise. I've run my tests with a resize factor of 5, meaning that one node represents the size 750 µm. The resize constant is 5, because the scaling in the XML files is 150 µm and one node represents 750 µm, and 150/750 = 1/5.

The tests are run with costX = 5 and costY = 1, in order to decrease running time.

I have evaluated the program on some benchmarks created by Wajid Hassan Minhass[iv]. The outputs and results of these evaluations are included in the appendix. Here I will show the overall result.

Benchmark	Number of valves	Number of reruns	Running time	Total length	Number of intersections (first run)	Number of intersections (min)
Arch10-1s	43	4	82	3037	419	0
Arch10-2s	39	6	42	872	109	0
ArchIvd2	50	16	588	2114	405	0
ArchSB2	48	50 (limit)	877	2286	140	2
ArchSB2s	47	9	94	2614	169	0

The program ran until a solution was found or a limit was reached. I set the limit as 50 iterations of the algorithm.

Discussion

The results from the benchmarks show that even though the number of valves is similar, the difficulty of routing all the channels differs. That's because it's the placement of the valves relative to one another that's important. If the valves are spaced out and don't share a lot of nodes or can easily find another path, then the algorithm doesn't have to rerun too many times before a solution is found.

However, if the shared nodes are trapped between other occupied nodes then it will take a long time before all the channels are routed properly. First the channels using shared nodes will spread out, which will mean other the occupied nodes around them will be shared instead. Then the channels using these nodes will have to spread out on so on until no nodes are shared. This can take a while if there are mane shared and occupied nodes next to each other.

Benefits

The benefit of the Pathfinder algorithm is that it lets channels share nodes to begin with and then calculates how high the demand for a certain node is and then set the cost accordingly. This assures that slowly but surely the channels will spread out and start using different nodes. The routing isn't so much depending on which order the channels are routed in, which is a good thing. Channels who might have nodes all to themselves to begin with might later on want to find another path as to make room to other channels which are spreading out. If the algorithm only considered the channels that were conflicting then it would be difficult to figure out that an entire different channel needed to be rerouted as well in order to get a solution.

Improvements

Two control layers

In my implementation there is only a single control layer. However, it is possible to have a biochip with a control layer on top of the flow layer and at the bottom. The benefit of having two control layers is that it is easier to get a better routing, since there will be more possible ways to place the control channels. Instead of a channel having to be routed around another channel, and thereby increasing the length of the channel, is it possible to place the channels on different layers.

If a failed routing on a single control layer is found, the problem being a single intersection of channels, one of these channels could be moved to a second layer and the routing could then be used. However, this is not an optimal solution. If two layers are available it would be important to use them as efficiently as possible.

In order to use the two layers efficiently it would be likely to assume that both layers should contain about the same amount of routed channels. A way to do this would be to put every second channel on the top layer and the rest at the bottom layer. However, it might result in a bad routing if a lot of troublesome routings are placed at the same layer for no reason, so I don’t think this solution would be efficient on its own.

I think a good solution to using two layers would be to decide what layer to place a channel on when it’s routed and maybe move it to the other layer when it’s rerouted. In my implementation this could be done by introducing another cost for nodes, so that each node has a costA for the top layer and a costB for the bottom layer. When a channel is routed it should try to do the routing on both layers, trying to use both the costA and costB, and then choose the cheapest solution.

By combining the two suggestions, one get a solution that places the same amount of channels on each layer to begin with. But when the channels are rerouted, which they will be in every iteration of the algorithm, the channels will be placed to the cheapest layer. This means that if a lot of channels are using the same nodes in an area on one layer then some of them are likely to be moved to the other layer to avoid too long routes.

Let valves share inlets

Valves needs to be placed for all the components on the biochip, as well as around all the switches introduced by the routing of the flow channels. This can become a rather large number, meaning that there should be a lot of inlets to the small biochip. If it’s known when all the valves should be open and closed on the biochip, it is possible to figure out which valves are always open or closed at the same time. These valves could in principle share an inlet source, which would cut down the amount of inlets needed.

It should be taken into account that just because two valves can share an inlet, it doesn’t automatically mean that this is the most efficient solution. If the valves are placed far apart it can be a problem if their channels have to connect, because the channel lengths might become unnecessary long. Before inlet-sharing is introduced in the program, it should first be discussed when the algorithm should favour a short channel length and when a shared inlet is to be preferred.

More precision

As of right now the precision in my program isn’t very good. The valves and control channels don’t have the proper size compared to one another and the size of the biochip. In my implementation one node represents the size750 µm. This means that no valve or channel can be smaller than this. Since a valve is

100 µm on both side and a control channel is 30 µm with 40 µm between each channel, this isn't very precise.

This could be improved by making the grid contain more nodes, so that each node represents a smaller space. However, this would also increase the memory used by the program as well as the running time.

A smart algorithm

It is possible to make the algorithm smarter in a few ways.

As of right now channels are routed with a breadth first search that takes the cost into account. It could be a good idea to include a heuristic function so that the algorithm doesn't have to search through quite as many nodes before it finds a path.

Another way to make the algorithm smarter would be not to reroute channels that are already perfectly routed and has no impact on the rest of the channels. Assume we have a chip that contains a cluster of valves both to the left side of the biochip as well as to the right side. If the cluster of valves to the left side has been routed and there are no intersections and they use the inlets all the way to the left, then it's likely to assume that only the valves to the right would need to be rerouted in order to find a solution for the entire biochip.

Conclusion

The pathfinder algorithm is able to route the control channels on a flow-based biochip. In some cases it might take a lot of retries, but it keeps moving closer to a solution at each rerun of the algorithm. The use of a cost for shared nodes works as it means the algorithm doesn't just rip up and reroute random channels. At each run of the algorithm it becomes clearer what areas of the biochip are wanted by the channels and what areas are free to use, at the cost of a longer route length.

References

[i] Larry McMurchie, Carl Ebeling

PathFinder A Negotiation-based Performance-driven Router for FPGAs

[ii] File: library.xml, included the in zip-folder.

[iii] File: BioChip Simulator - Component Design.pdf, included in the zip-folder.

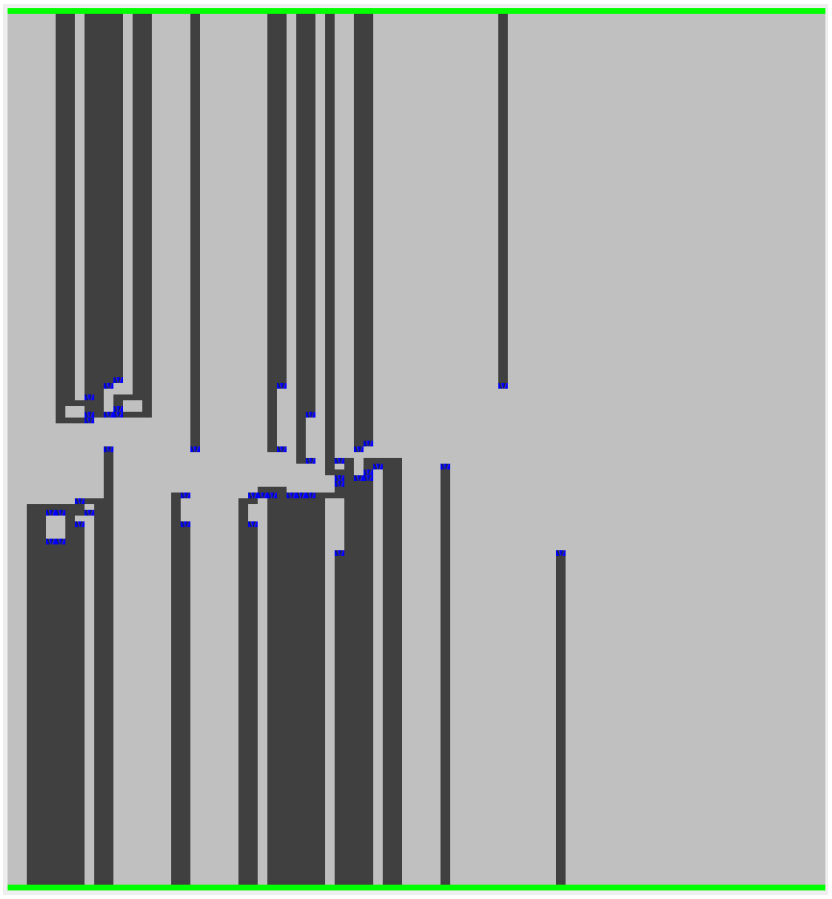
[iv] W. H. Minhass, P. Pop, and J. Madsen

System-Level Modeling and Synthesis Techniques for Flow-Based Microfluidic Very Large Scale Integration Biochips," Technical University of Denmark, Department of Information Technology.

Naveed A. Sherwani. Algorithms for VLSI Physical Design Automation, chapter "Grid routing"

Appendix

Arch10-1s



Number of valves: 43

Number of intersections: 0

FIRST RUN Number of intersections: 419

MIN Number of intersections: 0

Number of reruns: 4

Running time: 82

Total route length: 3037

Route 1 length: 65	Route 5 length: 68	Route 9 length: 74
Route 2 length: 66	Route 6 length: 72	Route 10 length: 71
Route 3 length: 66	Route 7 length: 73	Route 11 length: 75
Route 4 length: 66	Route 8 length: 72	Route 12 length: 76

Route 13 length: 77

Route 14 length: 77

Route 15 length: 78

Route 16 length: 77

Route 17 length: 80

Route 18 length: 76

Route 19 length: 75

Route 20 length: 74

Route 21 length: 80

Route 22 length: 83

Route 23 length: 72

Route 24 length: 72

Route 25 length: 73

Route 26 length: 70

Route 27 length: 70

Route 28 length: 73

Route 29 length: 69

Route 30 length: 69

Route 31 length: 69

Route 32 length: 69

Route 33 length: 70

Route 34 length: 67

Route 35 length: 67

Route 36 length: 74

Route 37 length: 64

Route 38 length: 64

Route 39 length: 64

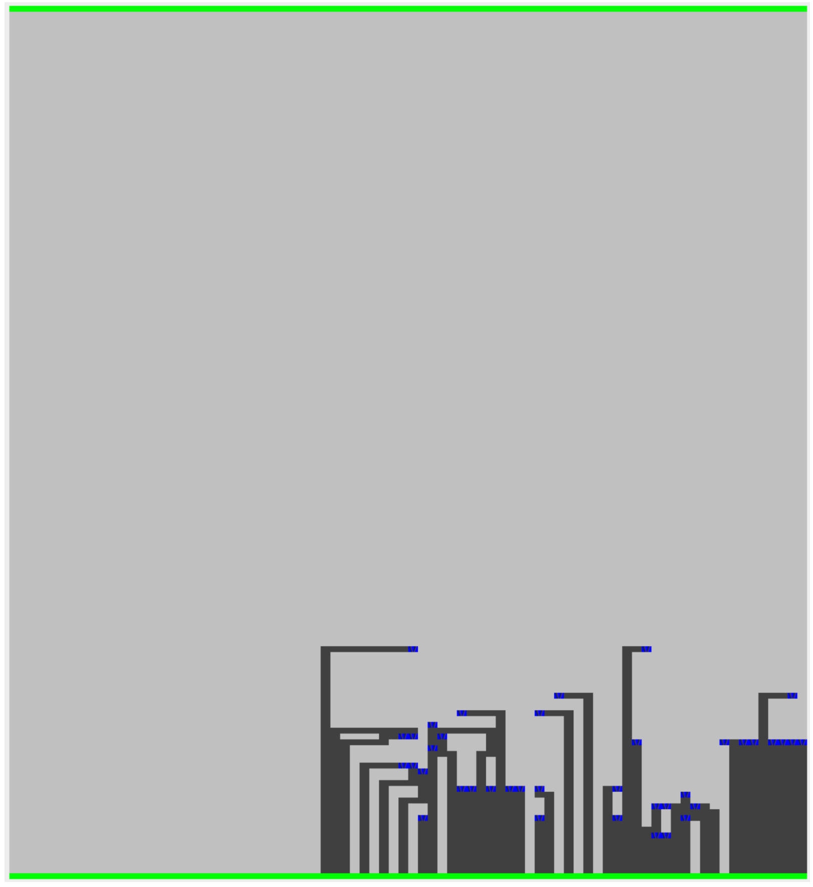
Route 40 length: 61

Route 41 length: 61

Route 42 length: 59

Route 43 length: 59

Arch10-2s



Number of valves: 39

Number of intersections: 0

FIRST RUN Number of intersections: 109

MIN Number of intersections: 0

Number of reruns: 6

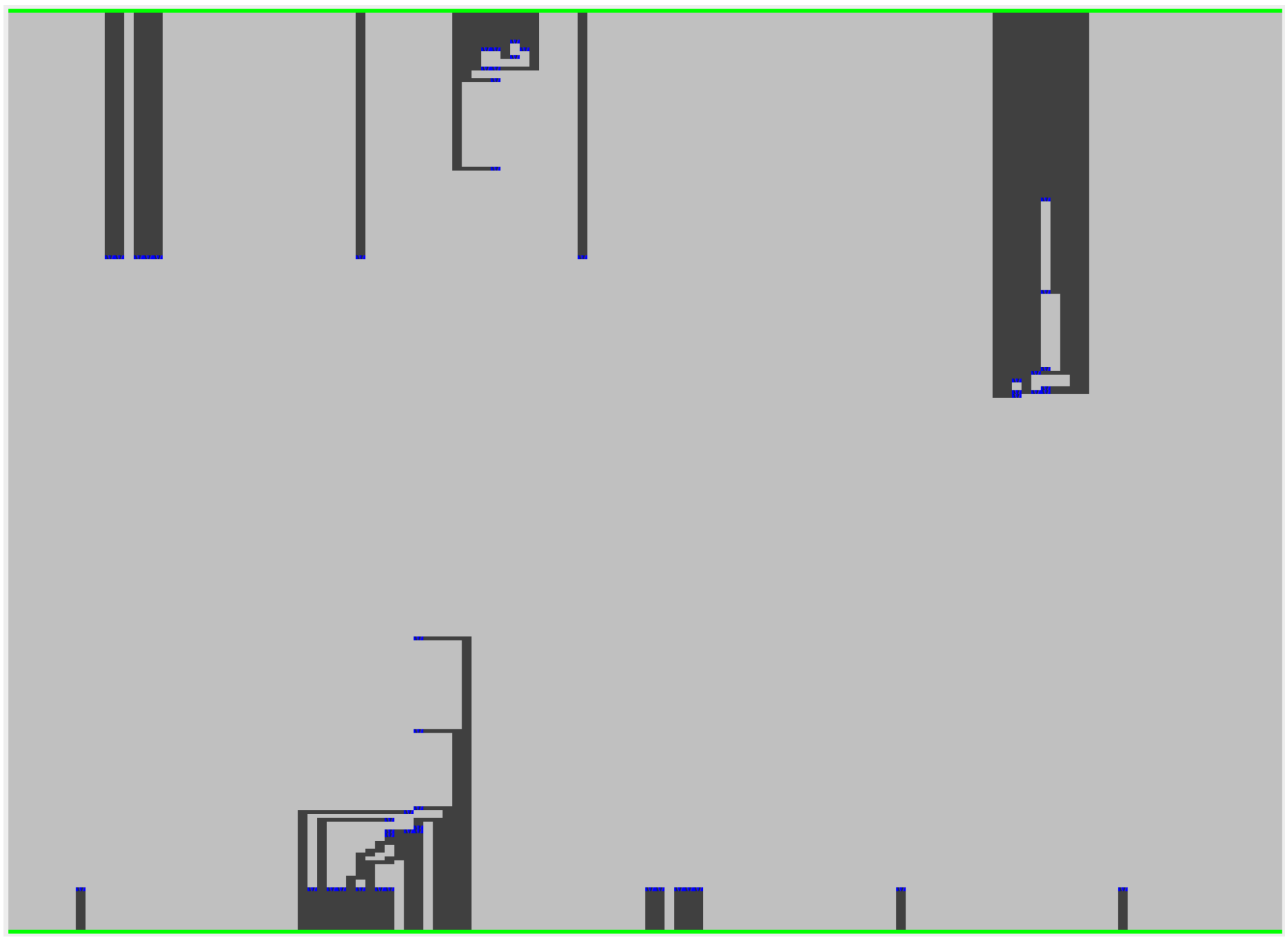
Running time: 39

Total route length: 872

Route 1 length: 49	Route 6 length: 32	Route 11 length: 24
Route 2 length: 42	Route 7 length: 29	Route 12 length: 25
Route 3 length: 35	Route 8 length: 31	Route 13 length: 24
Route 4 length: 35	Route 9 length: 35	Route 14 length: 24
Route 5 length: 33	Route 10 length: 33	Route 15 length: 24

Route 16 length: 24	Route 24 length: 16	Route 32 length: 14
Route 17 length: 24	Route 25 length: 16	Route 33 length: 15
Route 18 length: 24	Route 26 length: 16	Route 34 length: 11
Route 19 length: 23	Route 27 length: 16	Route 35 length: 11
Route 20 length: 24	Route 28 length: 17	Route 36 length: 11
Route 21 length: 23	Route 29 length: 17	Route 37 length: 11
Route 22 length: 21	Route 30 length: 17	Route 38 length: 8
Route 23 length: 16	Route 31 length: 14	Route 39 length: 8

ArchIvd2



Number of valves: 50

Number of intersections: 0

FIRST RUN Number of intersections: 405

MIN Number of intersections: 0

Number of reruns: 16

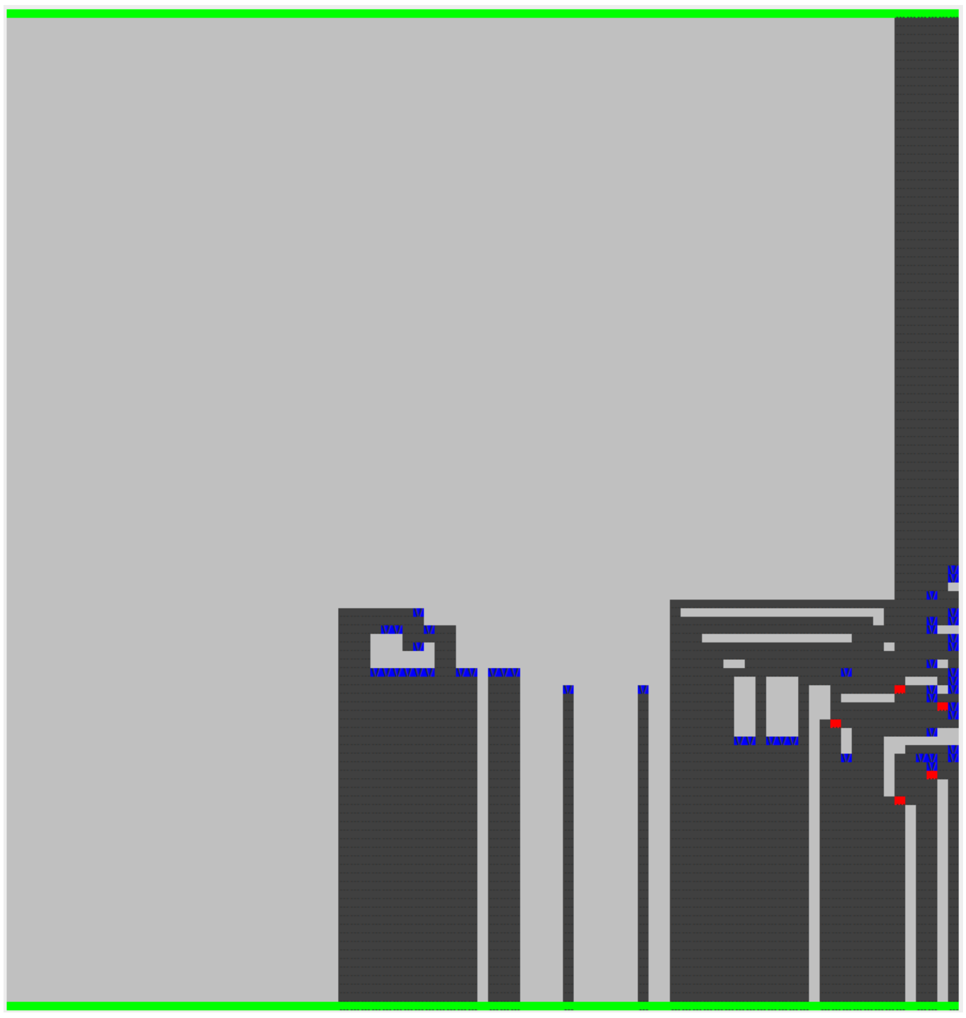
Running time: 588

Total route length: 2114

Route 1 length: 9	Route 3 length: 11	Route 5 length: 14
Route 2 length: 11	Route 4 length: 11	Route 6 length: 17

Route 7 length: 20	Route 22 length: 102	Route 37 length: 12
Route 8 length: 22	Route 23 length: 101	Route 38 length: 12
Route 9 length: 46	Route 24 length: 101	Route 39 length: 12
Route 10 length: 50	Route 25 length: 104	Route 40 length: 12
Route 11 length: 65	Route 26 length: 103	Route 41 length: 12
Route 12 length: 65	Route 27 length: 82	Route 42 length: 12
Route 13 length: 65	Route 28 length: 57	Route 43 length: 12
Route 14 length: 65	Route 29 length: 36	Route 44 length: 12
Route 15 length: 65	Route 30 length: 43	Route 45 length: 12
Route 16 length: 65	Route 31 length: 37	Route 46 length: 12
Route 17 length: 65	Route 32 length: 34	Route 47 length: 12
Route 18 length: 75	Route 33 length: 31	Route 48 length: 12
Route 19 length: 95	Route 34 length: 27	Route 49 length: 12
Route 20 length: 98	Route 35 length: 27	Route 50 length: 12
Route 21 length: 97	Route 36 length: 30	

ArchSB2



Number of valves: 48

Number of intersections: 5

FIRST RUN Number of intersections: 140

MIN Number of intersections: 2

Number of reruns: 50

Running time: 877

Total route length: 2286

Route 1 length: 66

Route 2 length: 68

Route 3 length: 70

Route 4 length: 49

Route 5 length: 73

Route 6 length: 75

Route 7 length: 76

Route 8 length: 47

Route 9 length: 51

Route 10 length: 47

Route 11 length: 75

Route 12 length: 73

Route 13 length: 58

Route 14 length: 71

Route 15 length: 63

Route 16 length: 40

Route 17 length: 40

Route 18 length: 40

Route 19 length: 40

Route 20 length: 40

Route 21 length: 40

Route 22 length: 40

Route 23 length: 40

Route 24 length: 40

Route 25 length: 40

Route 26 length: 40

Route 27 length: 48

Route 28 length: 69

Route 29 length: 64

Route 30 length: 38

Route 31 length: 38

Route 32 length: 47

Route 33 length: 50

Route 34 length: 53

Route 35 length: 45

Route 36 length: 43

Route 37 length: 38

Route 38 length: 32

Route 39 length: 32

Route 40 length: 32

Route 41 length: 32

Route 42 length: 32

Route 43 length: 36

Route 44 length: 30

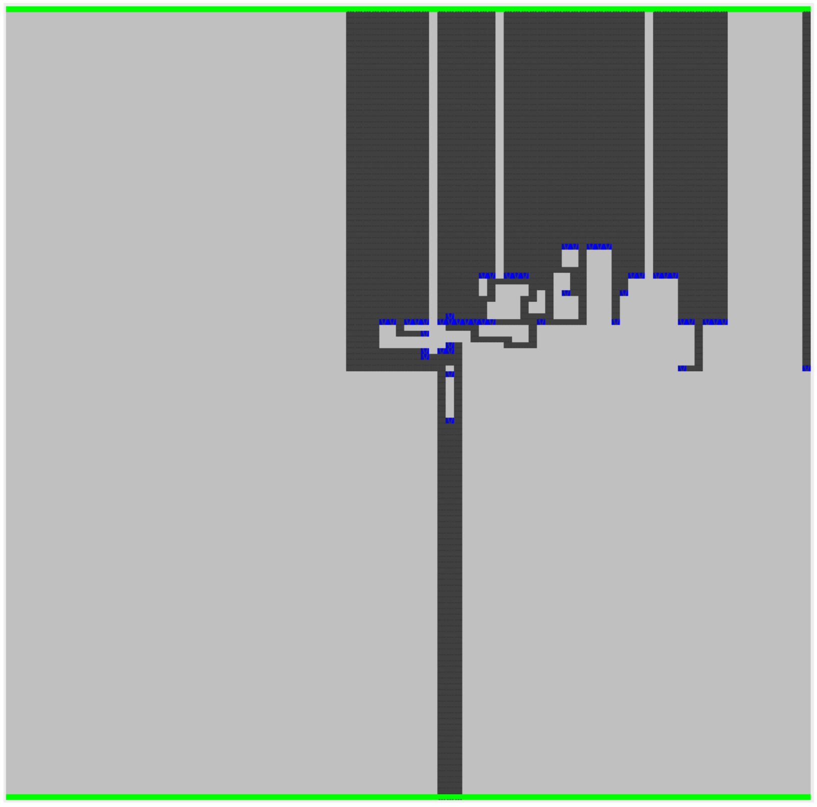
Route 45 length: 33

Route 46 length: 33

Route 47 length: 30

Route 48 length: 29

ArchSB2s



Number of valves: 47

Number of intersections: 0

FIRST RUN Number of intersections: 169

MIN Number of intersections: 0

Number of reruns: 9

Running time: 94

Total route length: 2614

Route 1 length: 42	Route 5 length: 42	Route 9 length: 47
Route 2 length: 42	Route 6 length: 47	Route 10 length: 47
Route 3 length: 42	Route 7 length: 47	Route 11 length: 47
Route 4 length: 42	Route 8 length: 47	Route 12 length: 47

Route 13 length: 47

Route 14 length: 47

Route 15 length: 47

Route 16 length: 53

Route 17 length: 50

Route 18 length: 54

Route 19 length: 55

Route 20 length: 55

Route 21 length: 55

Route 22 length: 55

Route 23 length: 55

Route 24 length: 55

Route 25 length: 75

Route 26 length: 55

Route 27 length: 55

Route 28 length: 55

Route 29 length: 61

Route 30 length: 61

Route 31 length: 60

Route 32 length: 55

Route 33 length: 55

Route 34 length: 55

Route 35 length: 55

Route 36 length: 55

Route 37 length: 55

Route 38 length: 60

Route 39 length: 80

Route 40 length: 66

Route 41 length: 74

Route 42 length: 78

Route 43 length: 68

Route 44 length: 65

Route 45 length: 63

Route 46 length: 75

Route 47 length: 66