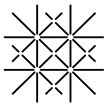


University of Basel

Temporal Planning for Droplet Routing on Microfluidic Biochips

Master thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Malte Helmert
Supervisor: Florian Pommenering

Fabian Burch
f.burch@unibas.ch
2015-916-760

November 17, 2022

Acknowledgments

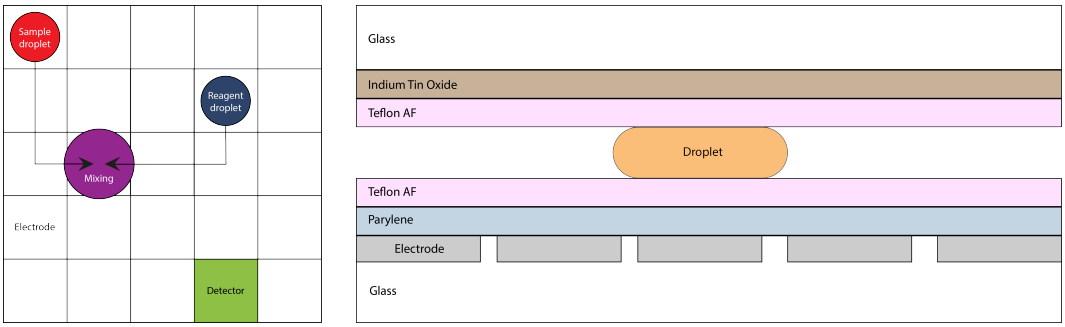
First, I would like to thank Florian Pommerening for being the most helpful and supportive advisor I could wish for. I would also like to thank Patrick Schnider for proofreading this work. Finally I would like to thank my partner for taking some load off my shoulders during the whole process.

Abstract

A Digital Microfluidic Biochip (DMFB) is a digitally controllable lab-on-a-chip. Droplets of fluids are moved, merged and mixed on a grid. Routing these droplets efficiently has been tackled by various different approaches. We try to use temporal planning to do droplet routing, inspired by the use of it in quantum circuit compilation. We test a model for droplet routing in both classical and temporal planning and compare both versions. We show that our classical planning model is an efficient method to find droplet routes on DMFBs. Then we extend our model and include spawning, disposing, merging, splitting and mixing of droplets. The results of these extensions show that we are able to find plans for simple experiments. When scaling the problem size to real life experiments our model fails to find plans.

Table of Contents

1 Introduction



(a) (b)

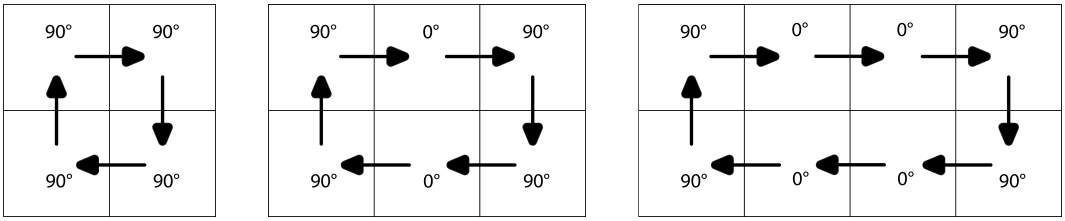
Figure 1.1: (a) Schema of a simple DMFB. A 5-by-5 grid of electrodes is used to move two droplets to the same location, mixing them. They then can be moved to a detection site. (b) Side view of a DMFB. Droplets are enclosed between two plates in a filler fluid (usually silicone oil) [7]. Open chip designs are also possible and cheaper to achieve.

Advances in microfluidic technologies have enabled the production of Digital Microfluidic Biochips (DMFBs), a fully electronically controllable lab on a chip. A DMFB essentially is a 2D-grid of electrodes on which small droplets of biochemical assets can be moved, mixed and measured, as can be seen in Figure 1.1(a). Whole biochemical experiments can be performed on such chips without the need for human interaction. This technology provides various advantages over classical laboratory experimentation, such as miniaturization, automation, repeatability, pruning human errors as well as being controllable by software. DMFBs are based on the effect of electrowetting [16]. By applying a voltage to an electrode, the contact angle of a droplet close to the electrode can be affected, effectively drawing the droplet onto the electrode [17]. This allows completely digital control of liquid droplets with the help of a grid of electrodes. A variety of biochemical operations can be performed just by using this phenomenon. They include, but are not limited to, movement, mixing, splitting, storage and detection. But such a powerful platform also comes with many challenges. In chapter 2 we will discuss the issues that come with the design of DMFBs and what has been done so far in trying to solve those issues. Then in chapter 3 we will present how we model droplet routing on DMFBs in PDDL, the language used for planning. In chapter 4 we present the results of testing our model. Chapter 5 goes into how we can extend our model to implement

more parts of DMFB design. The results of how those extensions performed are presented in chapter 6. Finally we discuss our results and where future work could expand into in chapter 7.

2
Related Work

When designing a DMFB experiment, one has to take into account various factors. Let us assume we want to mix two droplets. This can for example be achieved by a 2x2 mixing field (see Figure 2.1(a)) in which the assets to be mixed are moved in circular fashion [3]. Usually a guard ring is applied around a mixer, increasing the occupied area to a 4x4 square for a 2x2 mixer. Three different sizes of mixing modules can be seen in Figure 2.1. Bigger mixers generally achieve lower mixing times. Mixers, detectors, dilutors and storage units are examples for DMFB modules. Modules of all sizes are stored, alongside any accompanying information such as size and execution time, in a module library [4]. All the biochemical operations of an experiment have to be mapped to the DMFB modules. This is called binding. After finding appropriate modules for all operations, a schedule for the order of the operations has to be made and the modules have to be allocated to electrodes on the chip. Due to the dynamic nature of DMFBs, the modules are not to be seen as static but as dynamic objects that can be allocated to certain electrodes for the duration of the operation. If we return to our mixer example we would allocate a 4x4 area on the chip for the mixing operation. After completing the mixing process, the 4x4 space can be used for a completely different operation with different droplets. Droplets have to be moved to their target destinations between operations. A freshly mixed droplet may need to be moved to a detection site where its color will be measured. Droplets that are not meant to merge have to obey fluidic constraint rules in order to avoid unwanted mixing [22] during moving on the chip, as can be seen in Figure 2.2. For example if a droplet needs to be moved to an adjacent electrode, but this electrode is also adjacent to another droplet, both droplets will



(a) 2x2 module (b) 2x3 module (c) 2x4 module

Figure 2.1: Differently sized mixers. Generally speaking, bigger mixers will mix droplets faster than smaller ones.

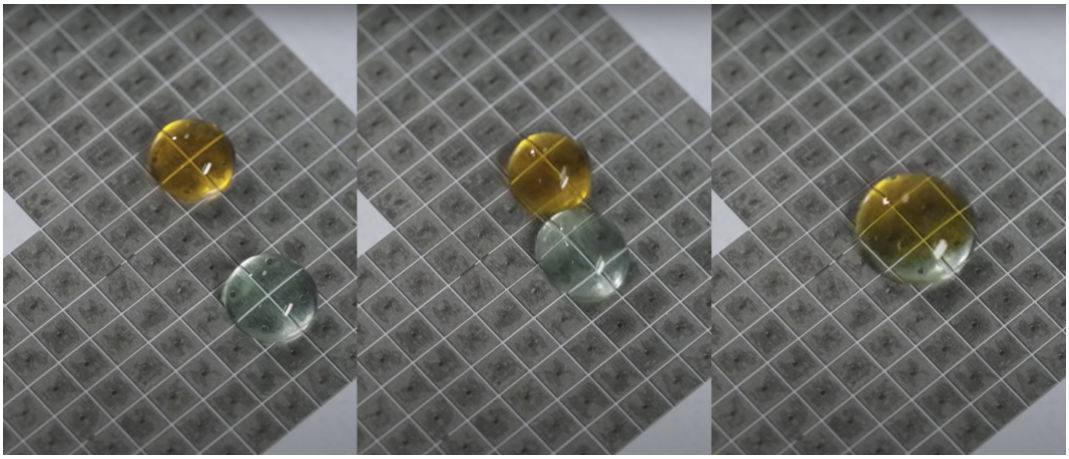


Figure 2.2: Two droplets that get too close and merge. The improper mixing can be seen as a color gradient on the big droplet. Pictures are taken from Umapathi and Ishii [23].

be drawn to the same electrode.
The problem of droplet routing has been covered by several papers with different approaches. Inspired by Very-Large-Scale Integrated Circuits (VLSI) design, several papers have used a modified version of the Lee algorithm [20] for finding shortest paths between two points [13, 22, 25, 27]. Keszöcze et al. have described droplet routing as a Boolean Satisfyability (SAT) problem and used the Z3 [15] SAT solver to find minimal paths [10]. A similar approach was used by Böhringer [2]. He describes the routing problem as a graph search problem and uses an A* search to obtain solutions.
Another topic that has been addressed by researchers is electrode control. Each electrode has to be connected to a control pin, that delivers the signals from the control computer. Pin assignment maps each electrode on the chip to a control pin. One possible way to control the electrodes is to connect the DMFB to a PCB (Printed Circuit Board). A PCB that can control every single electrode on the biochip is possible, but expensive. PCBs can be optimized for specific setups though. By doing a minimal pin assignment, fewer wires are needed to connect the control pins to the electrodes. In an effort to minimize the number of layers of a PCB and therefore also the production cost, McDaniel et al. proposed another modified Lee algorithm [13]. With a pin assignment as input, their algorithm tries to find a single-layer PCB wire routing. Whenever it cannot find a solution, it increases the number of layers by one until it finds a valid design. The process of finding a suitable wiring for a given pin assignment is called escape routing. Although escape routing and electrode control are interesting topics we do not cover them in this work.

It is apparent that there have been various approaches to try and solve the challenges of cost efficient and convenient DMFBs. From VLSI approaches to SAT solvers, several algorithms have been proven effective. We want to take a new approach that has previously been used in quantum circuit design.

Quantum computers use quantum effects to solve problems conventional computers struggle with. They use quantum bits, called qubits for short, that have a set of interesting proper-

ties. Unlike classical bits, qubits are not just either a 0 or a 1, but a superposition of the basis states $|0\rangle$ and $|1\rangle$. A qubit is in a quantum state $|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ where $\alpha, \beta \in \mathbb{C}$ denote probability amplitudes. When a qubit is measured, meaning a measurement device on the chip looks at the state of qubit, its quantum state collapses and either $|0\rangle$ or $|1\rangle$ is returned. The probability that a $|0\rangle$ is returned is $|\alpha|^2$, whereas the probability for $|1\rangle$ to be returned is $|\beta|^2$. States like $|\Psi\rangle$ are called pure or coherent states [5]. However, as soon as we use real-life representations of qubits, we diverge from this idealized model. For example, IBMs 5-qubit quantum chip is based on superconducting qubits [11]. Those superconducting qubits, as well as any other qubit representations that are currently available, are not perfectly isolated from their surroundings and will, over time, exchange information with their surroundings. This will make the qubits no longer pure. This loss of information in qubits is called decoherence [5]. The longer a qubit state is unmeasured, the less likely it is that a measurement returns the $|0\rangle$ or $|1\rangle$ state with their correct probabilities $|\alpha|^2$ or $|\beta|^2$. Therefore it is desirable to minimize the execution time of a quantum circuit.
Gate-model quantum computing chips, like the 5-qubit quantum chip by IBM, are an up-and-coming technology that is scalable and therefore able to run any quantum algorithm [24]. A quantum algorithm is performed by a quantum circuit, which is a sequence of applying quantum gates or measurements on qubits [14]. Quantum gates are the building blocks that quantum circuits are made of. Also called quantum logic gates, they are similar to classical logic gates, with the main difference being that quantum logic gates are reversible. They can be applied to single or multiple qubits, altering their states, just like logic gates. A quantum chip basically is a set of gates that can be applied to the qubits present on the chip. In Figure 2.3 a simple quantum chip design is shown.

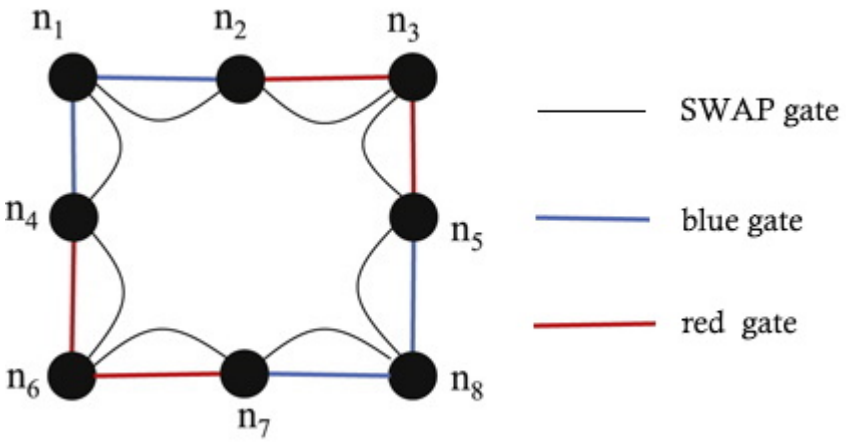


Figure 2.3: An abstract representation of a quantum chip. The black dots represent positions that qubits can be in (labelled n_1 through n_8). The lines connecting those positions represent gates, that are applied on the qubits in these positions. Picture taken from Venturelli et al. [24].

The gates, that are shown as lines connecting two dots, will affect the qubits on the two positions they connect. The most important gate is the SWAP gate, which changes the position of the two qubits it connects. This allows for two specific qubits to be moved to the positions that are connected by another gate that then can be applied to them. Each qubit is connected to two neighbours by SWAP gates and either red or blue gates, which represent

some specific two-qubits-gates. With this design it is possible to apply red or blue gates to any pair of qubits by executing a sequence of SWAP gates to move them into position.
Similar to a biochemical experiment that needs to be designed to be performed on a DMFB, a quantum circuit can be designed to perform a certain quantum algorithm on a quantum chip. This makes it interesting for us, as we can try to apply the methods used in quantum circuit design to droplet routing on DMFBs. There are parallels like SWAP gates that correspond to droplet movement and red or blue gates that correspond to modules like mixers or sensors. As mentioned before, quantum computing chips and circuits suffer from non-neglectable decoherence [24]. To counteract this, quantum circuits have to be designed to take minimal time to execute. Venturelli et al. have proposed temporal planning to find circuits with minimal execution time [24]. By describing hardware limitations, possible actions as well as start and goal state in PDDL (Planning Domain Definition Language), they were able to use any off-the-shelf temporal planner compatible with PDDL. For small circuits (less than 22 qubits) they were able to obtain good results in reasonable time. Electrode grids of the size 8x8 are a standard size for DMFBs [3]. Considering that even small mixers already take up a 4x4 square, the problem size is comparable and we should be able to obtain some results in reasonable time.

3

PDDL Model for Droplet Routing

Planning is the problem of finding a course of *actions* that connects the initial world state with a desired goal state. Two sub-areas of planning, which we will consider in this thesis, are *classical planning* and *temporal planning*.

3.1 Classical Planning

A set of binary *state variables* describes the current state of the world and are changed by the performed actions. An example for state variables for our problem would be "Droplet 1 is at position (4,2)" or "Position (3,1) is occupied". In classical planning actions consist of two sets, the *preconditions* and the *effects*. The preconditions are a list of all variables that need to be true, in order for the action to be able to be executed. The effects are a list of changes to the state variables that is executed if the action is performed. An example for an action would be "Move droplet 1 from position (1,1) to position (1,2)". The preconditions of this action are "Droplet 1 is at position (1,1)" and "Position (1,2) is not occupied". The effects are "Droplet 1 is not at position (1,1)", "Position (1,1) is not occupied", "Droplet 1 is at position (1,2)" and "Position (1,2) is occupied". An action has a cost, allowing some actions to be less attractive than others. A planning algorithm, a *planner*, will try to find a sequence of actions that leads from an initial state to a desired state called goal state. This is called a (valid) plan. An optimal plan is a valid plan with minimal cost. Depending on the problem, there might be several possible goal states, of which only one has to be reached for the problem to be considered solved. For one sort of problem we discuss, the initial states consist of a starting position for each droplet and of blocked positions, that cannot be used for droplet movement. The goal state is the state where all droplets have to be at their target position.

3.2 Temporal Planning

In classical planning actions take immediate effect. Temporal planning introduces slightly different actions called durative actions. Durative actions have a duration defining the time it takes for the action to complete. Preconditions of durative actions are split into three categories.

s	*at-start*:	conditions that must be true at the start of the action
e	*at-end*:	conditions that must be true at the end of the action
o	*over-all*:	conditions that must be true over the whole duration of the action

Similarly the effects can either be applied at the beginning or at the end of the action. Instead of a cost, temporal actions have a duration and in contrast to classical planning, actions can be executed simultaneously as long as their preconditions allow it. As a result the relevant metric for temporal plans is the duration from the beginning of the first action to the end of the final action. This duration is called the makespan.

3.3 PDDL Representation

In order for a planner to find a valid plan, we have to provide it with a logical representation of our problem. The *Planning Domain Description Language (PDDL)* [8] is a modelling language that was introduced to standardize the input of planning algorithms. PDDL is based on first-order logic, also known as predicate logic. In PDDL we define *objects* (like droplets and positions) and *predicates* (like *droplet-at(?d, ?p)*). The question mark indicates a placeholder for an object. Predicates that can not be changed are called static predicates and are written in all capital letters by convention. A predicate with objects (such as *droplet-at(droplet2, position5)*) is called a (Boolean) atom. Atoms are either true or false representing characteristics and relations between objects. To represent a state where *droplet1* is at *position5* the atom *droplet-at(droplet1, position5)* is true whereas the *droplet-at*-atom for *droplet1* and any other position than *position5* is false, as a droplet can only be at one position at a time. A state is a set of (true) atoms. Any atom not contained in the set is considered false. The initial state is a set of atoms that are true at the beginning of a plan. Actions change the current set of atoms, removing them from the set and adding others to it. The goal state is a list of atoms that need to be true at the end of the sequence of actions.

A planning task is formulated in a *domain* and a *problem* file. A domain serves as a common task description for several, different problems. Constants are objects that are present for every problem. They are defined in the domain file, along with predicates and actions. In the problem file the initial state, the goal state, problem-specific objects and static atoms are defined. If we want to model the domain for a 4 by 4 grid DMFB we would need to define $4 * 4 = 16$ position constants, p1 through p16. Then we define the predicates *droplet-at(?d, ?p)* and *occupied(?p)*. Finally we define the action *move(?d, ?p1, ?p2)*, where *?d* is the droplet, *?p1* the origin position and *?p2* the target position. But we also have to consider that we must provide some relation of the positions to each other. PDDL does

not automatically assume positions to be adjacent, but we have to define those relations via predicates. We define a new, static predicate *NEIGHBOURS(?p1, ?p2)* and add an atom for every pair of directly adjacent positions to the problem file. Then we can add that predicate as a precondition to our *move* action. In a process called "grounding" the planner will generate the atoms from the predicates for all specified objects. For example, the *droplet-at(?d, ?p)* predicate will generate an atom for every combination of droplet and position. Most planners will try to detect atoms that are only true in unreachable states and will not generate them.
In the problem file, the initial state, the goal state(s) and any additional problem-specific objects are defined. As an example we can define a simple problem with a single droplet at position *p1* that wants to end up at position *p5*:

```
(:init
    (droplet-at droplet1 p5)
    (occupied p5)

    (NEIGHBOUR p1 p5)
    (NEIGHBOUR p5 p1)
    ...
    (NEIGHBOUR p15 p16)
    (NEIGHBOUR p16 p15)
)

(:goal
    (droplet-at droplet1 p5)
)
```

By separating domain and problem into two files we can easily define a large number of problems for the same domain.

Now let us look at the problem of droplet routing on DMFBs in greater detail. On DMFBs droplets are moved by applying a current to a neighbouring cell. This means we have to pay attention when moving droplets, as all droplets in the vicinity of the activated cell are pulled towards it. We can formulate a set of rules that eliminates unwanted behaviour due to this way of moving. Let (X_i^t, Y_i^t) be the coordinates of droplet i at time t.

1.

$$|X_i^t - X_j^t| \geq 2 \text{ or } |Y_i^t - Y_j^t| \geq 2 \quad \text{for all droplets } i, j \text{ with } i \neq j, \text{ for all } t \geq 0 \quad (3.1)$$

The new (or initial) locations of two droplets may not be adjacent.

2.

$$|X_i^{t+1} - X_j^t| \geq 2 \text{ or } |Y_i^{t+1} - Y_j^t| \geq 2 \quad \text{for all droplets } i, j \text{ with } i \neq j, \text{ for all } t \geq 0 \quad (3.2)$$

The target cell for a droplet may not be adjacent to another droplet.

Let us look at an example. In Figure 3.1 two droplets and the moves they intend to do are shown. It is clear that the cell marked with an X cannot be activated, as in that case both droplets would be drawn towards that cell. The first rule does not disallow that, as the cells around the target positions of both droplets are free. The second rule though is violated. The target of the blue droplet has a distance of less than 2 in both y- and x-coordinate to the current position of the red droplet. In conclusion we can only perform the blue move after the red move has been completed.

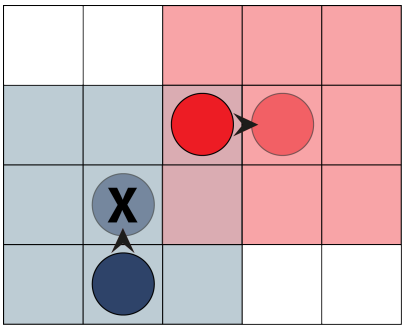


Figure 3.1: The blue droplet wants to move north, the red droplet wants to move east. The red droplet is too close to the target position of the blue droplet. Moving the blue droplet north would violate the second rule. We first have to move the red droplet. After that the blue droplet can be moved without violating a rule.

Venturelli et al. [24] used temporal planning for their quantum circuit compilation. We will now argue that classical planning is sufficient for the droplet routing problem. In an ideal setup, droplet movement has the same duration for all droplets and cells on the DMFB.[1] This is what we assume in equation (3.2), when we compare positions before and after one timestep. A cell is considered occupied by a droplet i until the move is completed. So we can not start a move with a different droplet j at $t - \delta$ where $-1 < \delta < 0$ if it would violate equation (3.1) for $t - \delta$. The main difference between classical and temporal planning in that aspect is that temporal planning is able to recognize moves that can be performed at the same timestep without interfering each other. Classical planning would list these moves subsequently but finding and scheduling parallel moves could be done in a post-processing step. The total number of moves, called the plan length, is the same for both approaches. Classical and temporal planning optimize for different solutions though. Classical planning minimizes the total cost of a plan. Since we have only one type of action with the standard cost of one, plan length and plan cost are the same. Temporal planning on the other hand minimizes the makespan. It might for example move one droplet some extra times to move it out of the way for another droplet with a very long total path. We decided to perform both classical and durative planning on a test set and compare the plan length, keeping in mind that is not an optimal metric.

[1] In reality, bigger droplets take longer to move than smaller droplets, as Bhattacharjee and Najjaran have shown [1].

3.4 Pre-Grounding or *forall*-Statements

If we want to implement the required rules in PDDL, we have to check for every cell, that if it contains a droplet, it is not in the vicinity of the target cell. One intuitive way of doing this is using a *forall* precondition. But this is not the most optimal approach as it limits planner options. For example in Fast Downward [9], one of the planners we use, we can not use certain heuristics that would guarantee to find optimal plans, as long as we have *forall* statements in the domain or problem. To avoid *forall* statements we introduce a different approach: we manually perform the grounding operation for the actions where a *forall* statement would be required, thus eliminating them. Instead of having only one move action with three parameters (droplet, origin-position and target-position), we include only legal moves, so moves to neighbouring cells. Any good grounder will also automatically do this. Additionally we explicitly require every cell in the vicinity of the target to be unoccupied as a precondition. Since the eight cells around the origin position are already required to be unoccupied from the last move or the initial setup (due to equation (3.1)) five cells from the eight surrounding the target position are already guaranteed to be unoccupied. This way we only have to check a maximum of three cells per move, as showcased in Figure 3.2.

For a more formal proof let us assume we have a valid initial configuration, meaning all droplets satisfy both rules. Since the first rule holds, all cells that in either x- or y-direction are different by only one or zero, have to be empty. Now we perform a new move, say moving droplet i in positive x-direction, so

$$X_i^{t+1} = X_i^t + 1 \text{ and } Y_i^{t+1} = Y_i^t \tag{3.3}$$

If we plug this into equation (3.2) the new condition we have to satisfy is

$$2 \leq |X_i^{t+1} - X_j^t| = |(X_i^t + 1) - X_j^t| \text{ or } 2 \leq |Y_i^{t+1} - Y_j^t| = |Y_i^t - Y_j^t|. \tag{3.4}$$

For a fixed X_i^t and Y_i^t there are 9 possible combinations of X_j^t and Y_j^t that satisfy both equations (3.1) and (3.4). Because the only change we have between equations (3.1) and (3.4) is the shift by 1 in the x-coordinate, those two sets of 9 cells have an overlap of 6 cells. Those 6 cells are already guaranteed to be unoccupied, so to satisfy equation (3.4) we require the 3 remaining cells to be unoccupied. This requirement is made in the precondition of our move actions.

A PDDL action that moves any droplet from cell (2,2) to cell (2,3) looks like this:

```
(:action move_22_23
    :parameters (?d - ?droplet)
    :precondition (and
        (droplet-at ?droplet x2 y2)
        (not (occupied x1 y4))
        (not (occupied x2 y4))
        (not (occupied x3 y4))
    )
    :effect (and
```

```
        (not (droplet-at ?droplet x2 y2))
        (droplet-at ?droplet x2 y3)
        (not (occupied x2 y2))
        (occupied x2 y3)
    )
)
```

The durative version of this action looks very similar:

```
(:durative-action move_22_23
    :parameters (?d - droplet)
    :duration (= ?duration 1)
    :condition (and
        (at start (droplet-at ?d x2 y2))
        (over all (not (occupied x1 y4)))
        (over all (not (occupied x2 y4)))
        (over all (not (occupied x3 y4)))
    )
    :effect (and
        (at start (not (droplet-at ?d x2 y2)))
        (at end (droplet-at ?d x2 y3))
        (at end (not (occupied x2 y2)))
        (at start (occupied x2 y3))
    )
)
```

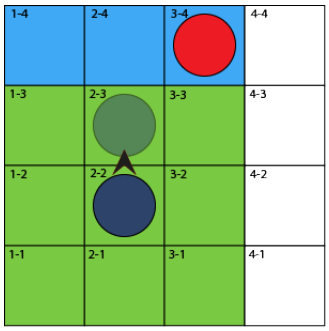


Figure 3.2: The blue droplet wants to move from cell 2-2 to cell 2-3. Assuming any initial configuration to not violate rule 1, the 9 surrounding fields (marked green) do not have to be checked, as any other droplet moving there would fail this very check in their move. Only the new 3 fields surrounding the target location have to be checked (marked blue). In this example, the move would not be possible as the target location would be in the vicinity of the red droplet.

Note that in the effect, we start occupying the target cell immediately and release the origin cell only at the end of the effect. This represents that the droplet is moving between the two cells during the action and is occupying both until the move is completed. Similarly as soon as the move starts, the *droplet-at* predicate is updated, disallowing new moves to start until the action is completed.

3.5 Coordinates or Position Labels

Another option we decided to include is the use of coordinates, so a split between a x- and y-coordinate object, versus a single location object. On a 4 by 4 chip we would either have 4 x- and 4 y-coordinate objects or 16 "position" objects, one for each cell. This is a trade-off between having fewer objects and more action arguments versus more objects and less action arguments. So our domains could have a reduced amount of objects when using coordinates:

```
(:types
    droplet coordinate - object
    xcoord ycoord - coordinate
    x1 x2 x3 x4 - xcoord
    y1 y2 y3 y4 - ycoord
)

(:action move
    :parameters (?d - droplet ?xo ?xt - xcoord ?yo ?yt - ycoord)
    :precondition (and
        ...
    )
    :effect (and
        ...
    )
)
```

Or when using a single, sequential coordinate each move has two parameters less:

```
(:types
    droplet coordinate - object
    c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 - coordinate
)

(:action move
    :parameters (?d - droplet ?co ?ct - coordinate)
    :precondition (
        ...
    )
    :effect (and
        ...
    )
)
```

3.6 Generating the PDDL Files

With this, we ended up with three options to build a configuration from: durative or classical planning, pre-grounded or "lifted" actions and coordinates or sequential position labels. This results in eight different configurations. For each configuration, a separate domain

and problem file has to be generated. For this we created a Python script, allowing us to automatically generate any desired configuration from parameters.
Stoppe et al. [21] have proposed a grammar for DMFB designs called BioGram alongside a program called BioViz that can visualize designs using that grammar. Our script is able to read files in the BioGram format and generate PDDL files from it. Plans that are found by the planners can then again be coded into the BioGram format and visualized using BioViz. Since we only do droplet routing so far, we use "blockages" to represent inaccessible cells such as mixer or detector cells.

4

Droplet Routing Results

To test and compare our eight different configurations, we generated random routing problems of different sizes, as shown in Table 4.1. Starting locations and destinations for the droplets are randomly placed on the chip. The starting position of any droplet is chosen not to be in the vicinity of another starting position but it may coincide with the target location of any droplet (including itself). This initial configuration is guaranteed to satisfy equation (3.1). There is no guarantee that the generated configuration is solvable, however every generated instance of our testset was solved by at least one configuration.
There are blockages on the chips that represent the occupied space of mixers and other modules. These blocked areas may not be used by any droplet movement. All chips are square and the blockages are rectangular with side lengths between 2 and $\lfloor (x-2)/2 \rfloor$ where x is the side length of the chip. For each configuration of the testset (so for each column of the table), 100 instances are generated. The aim of the test set is to compare the performance for changes in either size, number of droplets or number of blockages, with the two other parameters fixed.

side length	9	9	9	9	9	10	11	12	13	14	15	15	15	15
droplets	5	6	7	8	9	7	7	7	7	7	7	7	7	7
blockages	3	3	3	3	3	3	3	3	3	3	3	4	5	6

Table 4.1: Table of the test set parameters.

The tests were performed on a cluster of nodes with Intel Xeon E5-2660 processors. The memory was limited to 3500 MB per instance. Classical instances were solved with Fast Downward [9], temporal instances with Temporal Fast Downward (TFD) [6]. In a first round of tests, both planners were configured to search for any (potentially sub-optimal) solution and then stop. For Fast Downward we used the first iteration of the LAMA configuration [18] and for TFD we used the default arguments[2] except for anytime search, which was disabled. The time limit was set to 30 minutes per instance. For each instance a score was

[2] cyclic CG heuristic, epsilonize internally, reschedule plans

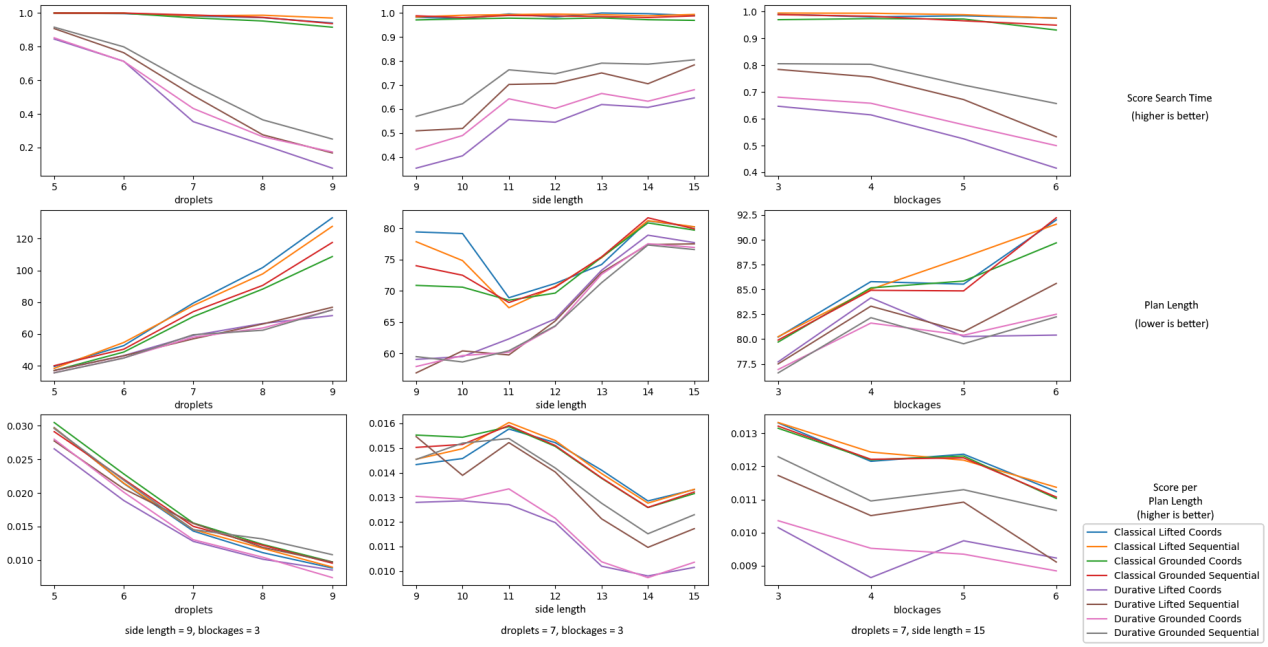


Figure 4.1: Test results for a search that aborts after the first found solution. In each row a different metric is presented. The first row shows the logarithmic score for the search time. The second row shows the plan length. The third row shows the score per plan length, combining the first two metrics. In the columns we fix two parameters and only change the third.

calculated according to

$$score = \begin{cases} \frac{\ln t_s - \ln t_{max}}{\ln 1 - \ln t_{max}} & 1 \leq t_s \leq t_{max} \\ 1 & t_s < 1 \\ 0 & t_s > t_{max} \end{cases} \tag{4.1}$$

where t_{max} is the time limit which is 1800 s and t_s is the search time in seconds. All search times that are below $1s$ are awarded a perfect score of 1. Unsolved instances are awarded a score of 0. With this logarithmic score we account for the fact that more complex problems take exponentially more time to solve. We then calculate the mean score for each of the eight configurations and for each problem type. In Figure 4.1 in the first row the scores are plotted. In each column we fix two parameters and change the third. For example in the first column the side length and the number of blockages are fixed and the number of droplets varies from 5 to 9. When looking at the first row it is clear that the classical planner achieves much higher scores in all cases. In the second row we take a look at the plan lengths. Here we can see that the temporal planner achieves lower plan lengths. This is remarkable as TFD minimizes the makespan and not necessarily the plan length. So the temporal version in general takes longer to find a plan, but the plan will be better than the one found by the classical version. That is why in the third row we combine score and plan length into a single metric "score per plan length". This way we can evaluate if the extra invested time of the temporal planner pays off in terms of reducing the plan length. As we can see temporal and classical planning move closer together, although the classical version still slightly outperforms the temporal one.

Looking only at the classical planning results we can see that they are generally very close.

We can see that the grounded versions (red and green) are overall worse than their lifted counterparts (yellow and blue). Similarly the sequential versions (red and yellow) perform better than their coordinates counterparts (green and blue). This trend is also observed in the temporal versions.
In Figure 4.2 we see survival plots for 4 types of problems. The x-scale is logarithmic and displays the search time. The plot shows how many (accumulated) instances were solved in the given time. Good curves rise fast. The top row displays problems on the biggest grid (15 by 15) with either 3 or 6 blockages. The temporal planner struggles to get solutions early, but catches up later. In the bottom row problems on the smallest grid (9 by 9) with either 5 or 9 droplets are shown. Problems with only 5 droplets are rather easy to solve. Classical Fast Downward finds most of its solutions in the first 0.1 seconds. TFD again starts out slowly, but finds solutions for over 90% of the problems after about 60 seconds. Both planners begin to struggle when the number of droplets is increased. With 9 droplets Fast Downward takes 10 seconds to find 90% of the solutions whereas TFD is not able to achieve a rate over 40% after 100 seconds. From these graphs it is obvious that at any point in time the classical version outperforms the temporal.

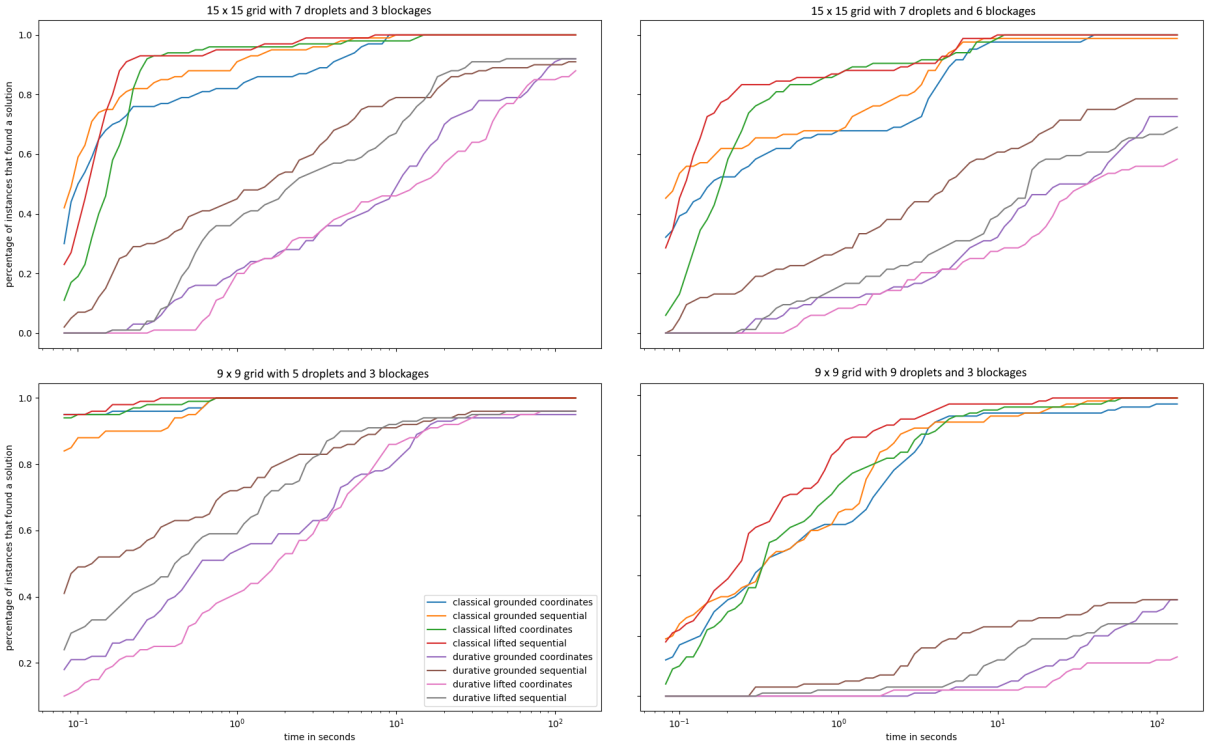


Figure 4.2: Survival plots for four configurations. The accumulated percentage of runs that found a solution over time is displayed.

In a second round of tests we ran the planners in anytime search configurations. When conducting an anytime search, the planner will not stop after finding an initial solution but continue to try to find better solutions. For Fast Downward we used the LAMA configuration. For TFD we enabled the anytime option. We set a time limit of 300 s for both

planners. The plan length of any found plan is saved along with its (total) search time, so any plan found after the first is bound to have a higher search time than the ones before. Like before we calculated a score for each found plan length according to

$$score = \begin{cases} \frac{\ln l_{plan} - \ln l_{max}}{\ln 1 - \ln l_{max}} & l_{min} \leq l_{plan} \leq l_{max} \\ 0 & l_{plan} > l_{max} \end{cases} \tag{4.2}$$

where l_{plan} is the found plan length, l_{min} is the minimal possible plan length and $l_{max} = 4 * l_{min}$. We calculate the minimal possible plan length from the difference in x- and y-coordinates from start and goal positions of the droplets, ignoring any blockages:

$$l_{min} = \sum_i (|x_i^{start} - x_i^{goal}| + |y_i^{start} - y_i^{goal}|) \tag{4.3}$$

In Figure 4.3 the normalized sum of the scores over time is shown. Each run starts with a score of zero. Over time plans are found for which a score is calculated. When a better plan is found the new, higher score replaces the old one. In the graph the scores of all runs are summed and divided by the number of runs. If a line would reach 1.0 this would mean that every run was eventually able to achieve a perfect score. An optimal curve would start very steep and reach the maximum early.

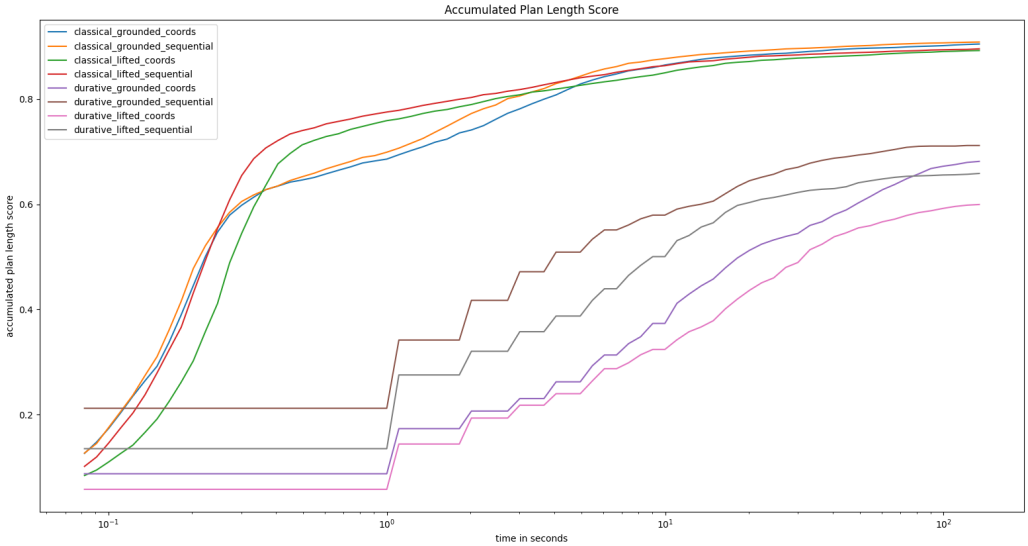


Figure 4.3: The summed search scores over time for all eight configurations. Optimal curves start steep and reach the maximum early. Durative curves are jagged because of rounding in the search times.

We can see that classical Fast Downward achieves high scores early but still improves late. This is close to optimal. Temporal Fast Downward displays flatter curves implying slow improvement. The jagged curve for the durative configurations come from TFD rounding their search times to integers.

5 Extensions

With the promising results from the test set, we were optimistic to extend beyond just droplet routing. We test different extensions trying to approach a model of a full DMFB experiment. From now on we only use classical planning and avoid pre-grounding, as these configurations proved to be more efficient. Also we stick with coordinates as they provide us with some advantages when merging and splitting droplets.

5.1 Extension 1: Spawning and Disposing, Splitting and Merging

In a first step we add actions for spawning and disposing droplets as well as for merging two droplets into one, big droplet and splitting a big droplet into two small ones. Since we want to model a real life application we test a model of the Polymerase Chain Reaction (PCR) bioassay [12] as it was also used by Chakraborty and Chakraborty [3]. The reaction starts with eight reagents that get mixed pairwise to four first order mixtures (M1 through M4) which are then mixed into two second order mixtures (M5 and M6) and those into one final mixture (M7). This is visualized in Figure 5.1. Around the 8x8 DMFB there are eight dispensers, each for one type of fluid, and one waste outlet where excess droplets can be disposed. The dispensers are arranged such that ingredients of first order mixtures are close to each other. The circles on the DMFB serve the purpose of visualizing how the different reagents are mixed to obtain the desired mixture M7 and do not represent actual droplet movement.

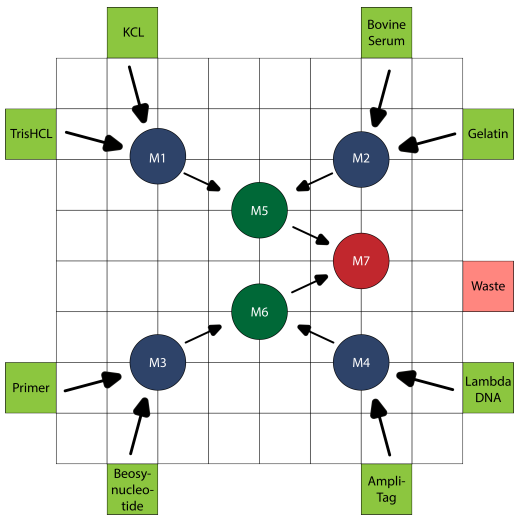


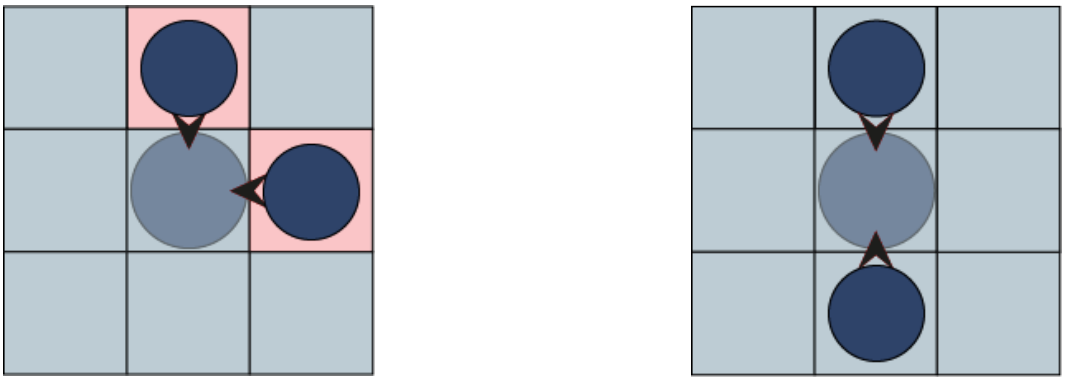
Figure 5.1: A possible DMFB setup for the PCR experiment. There are 8 dispensers for the different reagents and one waste outlet which can be used to dispose of excess droplets.

Adjusting our PDDL representation to be able to perform these new actions requires a change of paradigm. Since we now need to be able to spawn new and dispose unwanted droplets, our old *droplet-at(?d, ?p)* predicate is not sufficient anymore. We considered having a pool of spare droplets from which we can draw if needed but the bookkeeping is hard to do in PDDL. Instead we decided to change *droplet-at(?d, ?p)* to *reagent-at(?r, ?p)*. With this change we discard the droplet object as the unique identifier and instead only mark the type of fluid at a position. This allows us to have several reagents of the same type to coexist on the chip. As we still need a unique identifier to distinguish different droplets of possibly the same type we made the position the unique identifier. All other droplet attributes are attached to the position and need to be updated every time a droplet moves. One of those new attributes we need is the size of a droplet. We have to distinguish between small and big droplets, so we introduce the *small(?p)* predicate. The predicate is true if a droplet at position *?p* is small and false if the droplet is big or there is no droplet present. We can still distinguish between a big droplet and no droplet by checking the *reagent-at(?r, ?p)* predicate for the same position.
Further we need some more static predicates. We need a *MIX(?r1 ?r2 ?r3)* predicate that determines which reagents (*r1 and r2*) can be mixed into a product (*r3*). So for example in our problem file we would define *MIX(HCL, KCL, M1)*. Instead of *NEIGHBOUR(?p1, ?p2)* we now use *ISNORTH(?y1, ?y2)*, *ISWEST(?x1, ?x2)* and so on. *ISNORTH(?y1, ?y2)* is true if *?y1* is exactly one step north of *?y2*. This on one hand reduces the number of static predicates and on the other the number of parameters in our actions.
For spawning in the reagents as well as for disposing droplets we add the following actions:

```
(:action spawn_HCL    ; according actions for KCl, Gelatin etc.
    :parameters ()
    :precondition (and
        ; check that the spawn position is free
        ; as well as all surrounding cells
        (not (occupied x1 y6))
        (not (occupied x1 y7))
        (not (occupied x1 y8))
        (not (occupied x2 y6))
        (not (occupied x2 y7))
        (not (occupied x2 y8))
    )
    :effect (and
        ; spawn a small droplet with type HCL
        (reagent-type hcl x1 y7)
        (occupied x1 y7)
        (small x1 y7)
    )
)
```

```
(:action dispose
    :parameters (?r - reagent)
    :precondition (and
```

```
        ; check that there is a droplet with the specified type present
        (reagent-type ?r x8 y4)
    )
    :effect (and
        ; delete all atoms at that position
        (not (reagent-type ?r x8 y4))
        (not (occupied x8 y4))
        (not (small x8 y4))
    )
)
```



(a) illegal configuration

(b) legal configuration

Figure 5.2: Two setups for a merge operation. In (a) the two droplets are in an invalid configuration before merging. The configuration in (b) is possible as the droplets have enough space between them before merging.

When merging two droplets into one there are not a lot of possible positions from which we are allowed to merge. Merging is done by activating a cell that is adjacent to two droplets, pulling them both to the same location and merging them into one, big droplet. This is only possible if the droplets are on one line with one empty cell between them, as illustrated in Figure 5.2(b). Merging two droplets in an L-shape as seen in 5.2(a) is not possible as the droplets would violate equation (3.1). This leaves us with only two possible moves: merging in x- and in y-direction.

```
(:action merge_x
    :parameters(?r1 ?r2 ?r3 - reagent ?x1 ?x2 ?xt - xcoord ?yt - ycoord)
    :precondition (and
        ; check that the correct reagents are present
        (reagent-type ?r1 ?x1 ?yt)
        (reagent-type ?r2 ?x2 ?yt)
        ; check that the droplets are both small
        (small ?x1 ?yt)
        (small ?x2 ?yt)
        ; check that three cells are next to each other
        (ISEAST ?x1 ?xt)
        (ISWEST ?x2 ?xt)
        ; ckeck that the ?r1 and ?r2 can be mixed into ?r3
```

```
        (MIX ?r1 ?r2 ?r3)
    )
    :effect (and
        ; delete all atoms of the droplets that are merged
        (not (reagent-type ?r1 ?x1 ?yt))
        (not (reagent-type ?r2 ?x2 ?yt))
        (not (small ?x1 ?yt))
        (not (small ?x2 ?yt))
        (not (occupied ?x1 ?yt))
        (not (occupied ?x2 ?yt))
        ; add atoms for the newly merged, big droplet
        (reagent-type ?r3 ?xt ?yt)
        (occupied ?xt ?yt)
    )
)
```

Merging in y-direction is done accordingly. Splitting is very similar to merging when it comes to the arrangement of the droplets. Splitting a big droplet into two small ones is done by activating cells on two opposite sides of the cell where the big droplet sits [26]. Activating neighbouring cells that are not opposite to each other can lead to uneven splitting. We also distinguish between splitting in x- and y-direction.

```
(:action split_x
    :parameters (?r - reagent ?xo ?xl ?xr - xcoord ?yo - ycoord)
    :precondition (and
        ; check that the correct reagent is present
        (reagent-type ?r ?xo ?yo)
        ; check that three cells are opposite to each other
        (ISWEST ?xl ?xo)
        (ISEAST ?xr ?xo)
        ; check that the droplet is big
        (not (small ?xo ?yo))
        ; check that all cells around the two target cells are unoccupied
        (forall (?x - xcoord)
          (forall (?y - ycoord)
            (imply (and
                (not (and (= ?x ?xo) (= ?y ?yo)))
                (or
                    (VICINITY ?x ?y ?xl ?yo)
                    (VICINITY ?x ?y ?xr ?yo)
                )
            )
                (not (occupied ?x ?y))
            )
          )
        )
    )
```

```
    :effect (and
        ; delete the atoms from the cell where the big droplet was
        (not (reagent-type ?r ?xo ?yo))
        (not (occupied ?xo ?yo))
        ; add the two new, small droplets
        (small ?xl ?yo)
        (reagent-type ?r ?xl ?yo)
        (occupied ?xl ?yo)
        (small ?xr ?yo)
        (reagent-type ?r ?xr ?yo)
        (occupied ?xr ?yo)
    )
)
```

We call this extension "Merge-Only". Note that we completely neglect mixing merged droplets here. Any droplet that results from a merge action is immediately considered completely mixed. For proper results in a real experiment a freshly merged droplet first needs to be mixed, for example in a mixing module. This is what we implement next.

5.2 Extension 2: Mixing Modules

In Extension 2 we add mixing modules. We add two 2x4 mixers, one in the top right and one in the bottom left corner. The mixing area may be used freely for droplet movement when no mixing takes place. The mixing operation itself can only be executed if the area is free from droplets including the cells around the actual mixing area. We introduce the predicate *is-mixed(?x ?y)* and add it to the precondition of the split actions.

```
(:action mix_top_right
    :parameters (?r - reagent ?x - xcoord ?y - ycoord)
    :precondition (and
        (reagent-type ?r ?x ?y)
        (not (small ?x ?y))
        ; check that the padding cells are free
        (not (occupied x4 y8))
        (not (occupied x4 y7))
        (not (occupied x4 y6))
        (not (occupied x5 y6))
        (not (occupied x6 y6))
        (not (occupied x7 y6))
        (not (occupied x8 y6))
        (or
            ; check that all cells except (x, y) are free
            (and
                (= ?x x5)
                (= ?y y8)
                (not (occupied x5 y7))
                (not (occupied x6 y8))
```

```
                (not (occupied x6 y7))
                (not (occupied x7 y8))
                (not (occupied x7 y7))
                (not (occupied x8 y8))
                (not (occupied x8 y7))
            )
            (and
                ; same for all other cells
                ...
            )
            ...
        )
    )
    :effect (and
        (is-mixed ?x ?y)
    )
)
```

We call this extension "Merge-Mixer".

5.3 Extension 3: Mixing Model Without Mixers

Chakraborty and Chakraborty [3] have shown how droplets can be mixed while being moved to their target location, effectively eliminating the need for mixing-modules. Without the need for dedicated areas during the mixing process, they were able to significantly reduce the time for a DMFB experiment to complete. Several differently sized mixing modules (as in Figure 2.1) were tested on how many moves they need to achieve 100% mixing. In $2 \times N$ mixing modules droplets are moved in circular shape. There are several types of movement in such modules that increase the mixing percentage by different amounts. The amount depends on the last type of movement made. The movements can be divided into 90° moves and straight-line moves. For example a move in northern direction made directly after a move in western direction is a 90° move. The mix percentage of a straight-line move, so a move in the same direction as the last one, depends on how many consecutive straight-line moves have been done before the current one. By measuring the number of movements in a 2×2 mixing module the mixing completion percentage for 90° moves was calculated:

$$p_{90^\circ} = \frac{p_{2\times 2}}{4} \tag{5.1}$$

where p_{90° is the mixing percentage of a 90° move and $p_{2\times 2}$ is the mixing percentage of one full loop in a 2×2 mixing module. The percentages for straight-line moves were then calculated from the consecutively larger $2 \times N$ modules:

$$p_{0^\circ_N} = \frac{p_{2\times(N+2)} - p_{2\times(N+1)}}{2} \tag{5.2}$$

where $p_{0^\circ_N}$ is the N^{th} consecutive straight-line move and $p_{2\times N}$ is the mixing percentage of one loop in a $2 \times N$ mixing module. The results are presented in Table 5.1. Ignore the

last column for now. Whereas the efficiency of each consecutive straight-line move rises initially, any 0°_N shift after $N = 3$ sees a decrease in the mixing completion percentage. Thus Chakraborty and Chakraborty do not consider any shift after 0°_4 in their approach. We were convinced to be able to apply this module-less approach in our planning framework. With the mixing completion percentages from Table 5.1, we can keep track for each droplet if it is completely mixed.

Shift patterns	Mixing completion percentage	Modelled percentage
(90°)	0.625 %	0%
(0°_1)	1.875 %	0%
(0°_2)	5.125 %	5%
(0°_3)	5.946 %	5%
(0°_4)	5.929 %	5%

Table 5.1: Mixing completion percentages for different types of movement patterns. The shift patterns describe the relation of the direction of the last performed move to the direction of the current move. An example for a 90° shift is a movement in eastern direction after a movement in southern direction. The indices of the 0° moves denote the number of consecutive shifts in the same direction.

The additional attributes we need are the mixing percentage, the direction of the last move and the number of consecutive moves in the same direction. The predicates therefore are

- *mix-percentage(?m ?x ?y)* is true if the droplet at position (x,y) is mixed m percent
- *moved-west(?x ?y)* is true if the droplet at position (x,y) made its last move in western direction
- *moved-east(?x ?y)* is true if the droplet at position (x,y) made its last move in eastern direction
- *moved-north(?x ?y)* is true if the droplet at position (x,y) made its last move in northern direction
- *moved-south(?x ?y)* is true if the droplet at position (x,y) made its last move in southern direction
- *moved-once(?x ?y)* is true if the droplet at position (x,y) made one consecutive move in the same direction
- *moved-twice(?x ?y)* is true if the droplet at position (x,y) made two consecutive moves in the same direction
- *moved-three-times(?x ?y)* is true if the droplet at position (x,y) made three consecutive moves in the same direction
- *moved-four-times(?x ?y)* is true if the droplet at position (x,y) made four consecutive moves in the same direction
- *moved-five-plus-times(?x ?y)* is true if the droplet at position (x,y) made five or more consecutive moves in the same direction

When a droplet has moved into a direction once, it has not moved into that direction before, so it performed either a 90° or a 180° shift. A droplet that moved into a direction twice has performed exactly one 0° shift and so forth. After five moves the mixing percentage does not increase anymore, therefore such a droplet keeps being in this five-plus state.

We have to decide on a discretisation of the mixing percentages that keeps the number of generated atoms low but is able to represent the numbers from Table 5.1 as good as possible. We ended up with steps of 5% as it fits the numbers fairly well and was manageable by the planners. With the chosen model we underestimate the actual mixing percentage. An example of this can be seen in Figure 5.3. The black numbers show how much percentage of mixing is added in each step according to the calculations from Chakraborty and Chakraborty [3]. The totally achieved mixing percentage is 37.446%. The red path numbers are the mixing percentages of our model. It ends up with 25% mixing.

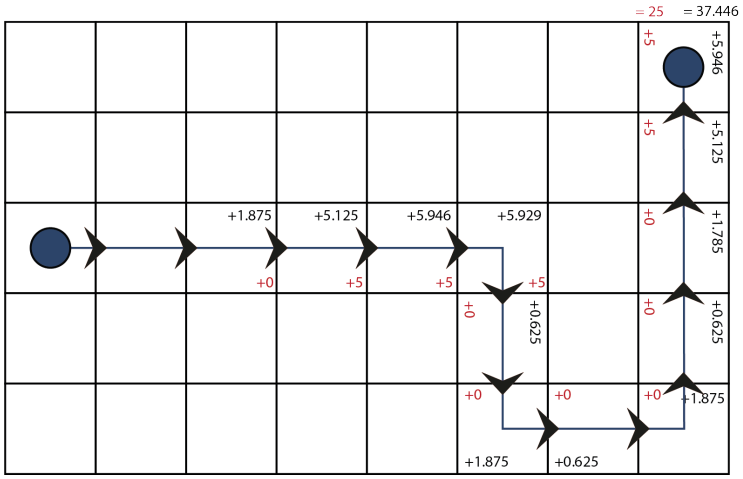


Figure 5.3: Mixing percentages along a path of 90° and 0° shifts. The black numbers show the experimentally observed percentages. The red numbers show the percentages of our model.

By strictly underestimating the mixing percentages the found plans are always valid plans in real experiments. To keep track of the last direction we split the move action into four actions - one for each direction. We use the *move-north* action to go over the details of the PDDL representation:

```
(:action move_north
    :parameters (
    ?r - reagent ?xo - xcoord ?yo ?yt - ycoord ?p1 - percentage)
    :precondition (and
        (reagent-type ?r ?xo ?yo)
        (ISNORTH ?yt ?yo)
        (mix-percentage ?p1 ?xo ?yo)
        (not (blocked ?xo ?yt))
        (forall (?x - xcoord)
          (forall (?y - ycoord)
            (imply (and
                (not (and (= ?x ?xo) (= ?y ?yo)))
                (VICINITY ?x ?y ?xo ?yt)
            )
                (not (occupied ?x ?y))
            )
          )
```

```
        )
    )
```

In the precondition we check that a droplet of type r with mixing percentage p1 is present at cell (x,y). We check that the target cell is located north of the origin cell and that it is not blocked. Then we perform our usual check for unoccupied cells around the target position.

```
    :effect (and
        (when (small ?xo ?yo) (and
            (not (small ?xo ?yo))
            (small ?xo ?yt)
            )
        )
        ...
```

First we have a conditional effect. This is a part of the effect that only takes place if another condition is met. Here the condition is that the droplet at the origin cell is small. If this is true we delete this atom and add the *small* predicate at the target location instead.

```
        ...
        (forall (?p2 - percentage) (and
            (when (and
                (moved-north ?xo ?yo)
                (moved-four-times ?xo ?yo)
                (NEXTPERCENTAGE ?p1 ?p2)
            ) (and
                (not (moved-north ?xo ?yo))
                (not (moved-four-times ?xo ?yo))
                (not (mix-percentage ?p1 ?xo ?yo))
                (moved-north ?xo ?yt)
                (moved-five-plus-times ?xo ?yt)
                (mix-percentage ?p2 ?xo ?yt)
            )
            )
            (when (and
                (moved-north ?xo ?yo)
                (moved-three-times ?xo ?yo)
                (NEXTPERCENTAGE ?p1 ?p2)
            ) (and
                (not (moved-north ?xo ?yo))
                (not (moved-three-times ?xo ?yo))
                (not (mix-percentage ?p1 ?xo ?yo))
                (moved-north ?xo ?yt)
                (moved-four-times ?xo ?yt)
                (mix-percentage ?p2 ?xo ?yt)
            )
            )
            (when (and
```

```
            (moved-north ?xo ?yo)
            (moved-twice ?xo ?yo)
            (NEXTPERCENTAGE ?p1 ?p2)
        ) (and
            (not (moved-north ?xo ?yo))
            (not (moved-twice ?xo ?yo))
            (not (mix-percentage ?p1 ?xo ?yo))
            (moved-north ?xo ?yt)
            (moved-three-times ?xo ?yt)
            (mix-percentage ?p2 ?xo ?yt)
        )
        )
    ))
    ...
```

Next we cover the moves that add 5% to the total mixing percentage. This happens when we move into the same direction for the third, fourth or fifth time which corresponds to a second, third or fourth consecutive straight-line move. For each of these three cases we add another conditional effect. In the condition we make sure that the droplet has made the correct amount of moves in the same direction, so in this case the northern direction. To guarantee that that the percentage is increased by exactly 5% we go through all *percentage* objects and check the static predicate *NEXTPERCENTAGE*. This predicate is defined in the problem file by atoms such as *NEXTPERCENTAGE(p5, p10)*. The effect then simply removes all atoms covering the origin position and adds the atoms that represent the updated attributes at the target position.

```
    ...
    (when (and
        (moved-north ?xo ?yo)
        (moved-five-plus-times ?xo ?yo)
    ) (and
        (not (moved-north ?xo ?yo))
        (not (moved-five-plus-times ?xo ?yo))
        (not (mix-percentage ?p1 ?xo ?yo))
        (moved-five-plus-times ?xo ?yt)
        (moved-north ?xo ?yt)
        (mix-percentage ?p1 ?xo ?yt)
    )
    )
    (when (and
        (moved-north ?xo ?yo)
        (moved-once ?xo ?yo)
    ) (and
        (not (moved-north ?xo ?yo))
        (not (moved-once ?xo ?yo))
        (not (mix-percentage ?p1 ?xo ?yo))
        (moved-north ?xo ?yt)
```

```
            (moved-twice ?xo ?yt)
            (mix-percentage ?p1 ?xo ?yt)
        )
        ...
```

Then we cover the remaining cases where the percentage is not increased.

```
        ...
        )
        (when (and
            (not (moved-north ?xo ?yo))
        ) (and
            (not (moved-east ?xo ?yo))
            (not (moved-west ?xo ?yo))
            (not (moved-south ?xo ?yo))
            (not (moved-once ?xo ?yo))
            (not (moved-twice ?xo ?yo))
            (not (moved-three-times ?xo ?yo))
            (not (moved-four-times ?xo ?yo))
            (not (moved-five-plus-times ?xo ?yo))
            (not (mix-percentage ?p1 ?xo ?yo))
            (moved-north ?xo ?yt)
            (moved-once ?xo ?yt)
            (mix-percentage ?p1 ?xo ?yt)
        )
        )
        ...
```

If the last move was not made in northern direction the mixing percentage is not increased. In that case we make sure all atoms covering the origin position are deleted.

```
        ...
        (not (reagent-type ?r ?xo ?yo))
        (reagent-type ?r ?xo ?yt)
        (not (occupied ?xo ?yo))
        (occupied ?xo ?yt)
    )
)
```

Finally we update the *reagent-type* and *occupied* predicates which are always changed regardless of any previous move.
We call this extension "Merge-No-Module".

6
Extensions Results

To test our extensions we used different stages of the polymerase chain reaction (PCR). In Figure 6.1 the sequence of the mixtures in the PCR are shown. Mixtures 1 to 4 are the easiest to achieve, as they only need to mix two reagents each. Mixtures 5 and 6 use a total of four reagents in two steps. Finally mixture 7 uses all eight reagents. We create seven different problems. The goal for every problem is to have a small droplet of mixture N at position (8,1), so the bottom right corner. For each problem only the dispensers for reagent types that are needed in the desired mixture are made available. Because we require the droplet to be small, every plan needs to include at least one *split* and therefore also at least one *merge* action.

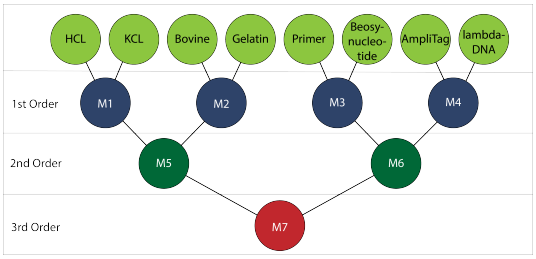


Figure 6.1: Sequence graph for the polymerase chain reaction (PCR).

The tests were performed on a cluster of nodes with Intel Xeon Silver 4114 processors. The memory was limited to 6000 MB per instance. We used Fast Downward in the LAMA configuration and set a time limit of 1 hour. The results of these tests can be seen in Figure 6.2. On the logarithmic x-scale the search time is shown. The y-scale displays a score calculated for the plan length according to equation (4.2). Here l_{min} is the manually calculated minimal plan length for each problem and $l_{max} = 10 * l_{min}$. Each line represents a run and each point stands for a found plan. The LAMA configuration will continue to find plans with lower plan length than the previously found plan.

The first extension ("Merge-Only", green) where we neglect mixing found plans for all seven mixtures. For the first-order mixtures M1 to M4 good scores are found fast. Second-order mixtures M5 and M6 take slightly longer but still achieve good scores. Third-order mixture M7 is also fast but the plan quality is suffering with the final plan only achieving a score of roughly 0.7.

The second extension ("Merge-Mixer", orange) where we add fixed mixing modules found plans for all mixtures except the final one. Similar to the "Merge-Only" extension plans

for the first-order mixtures are found fairly fast (under 1 minute) and achieve high scores. Second-order mixtures take significantly longer though. The best plan score is much better for M6 (0.88) than for M5 (0.61).

Finally the last extension ("Merge-No-Module", purple) where we mix freely without using fixed modules only found plans for mixtures M2, M3, M4 and M5. Interestingly no plan for M1 was found despite M1 being an ingredient for M5 for which a plan was found. All four runs only found a single plan. For the first-order mixtures plans were found between 5 s and 30 s with scores between 0.6 and 0.75. For M5 it took just under 9 minutes to find a plan with a score just below 0.5. The exact results can be found in Table 6.1.

To find a reason why certain scores are so low we take a look at the generated plans. The "Merge-No-Module" extension should perform five consecutive moves in the same direction and then change direction to be efficient. As shown in Figure 5.3 changing directions frequently will not lead to a significant increase in mixing percentage. In our model the mixing percentage will not increase at all since we round the small percentages to zero. When we look at the plan for mixture M5 we see that it is very inefficient in that regard. In Figure 6.3(a) we can see how the plan is increasing the mixing percentage of M2 (purple droplet). The droplet is moved in circles where on the top it first performs three consecutive straight-line moves in eastern direction, increasing the mixing percentage by 5%. Then it takes one step back, goes south and performs two consecutive straight-line moves which does not increase the mixing percentage before moving north. This leads to a total increase of only 5% per cycle. Additionally the red droplet is moved east to not get too close to the purple

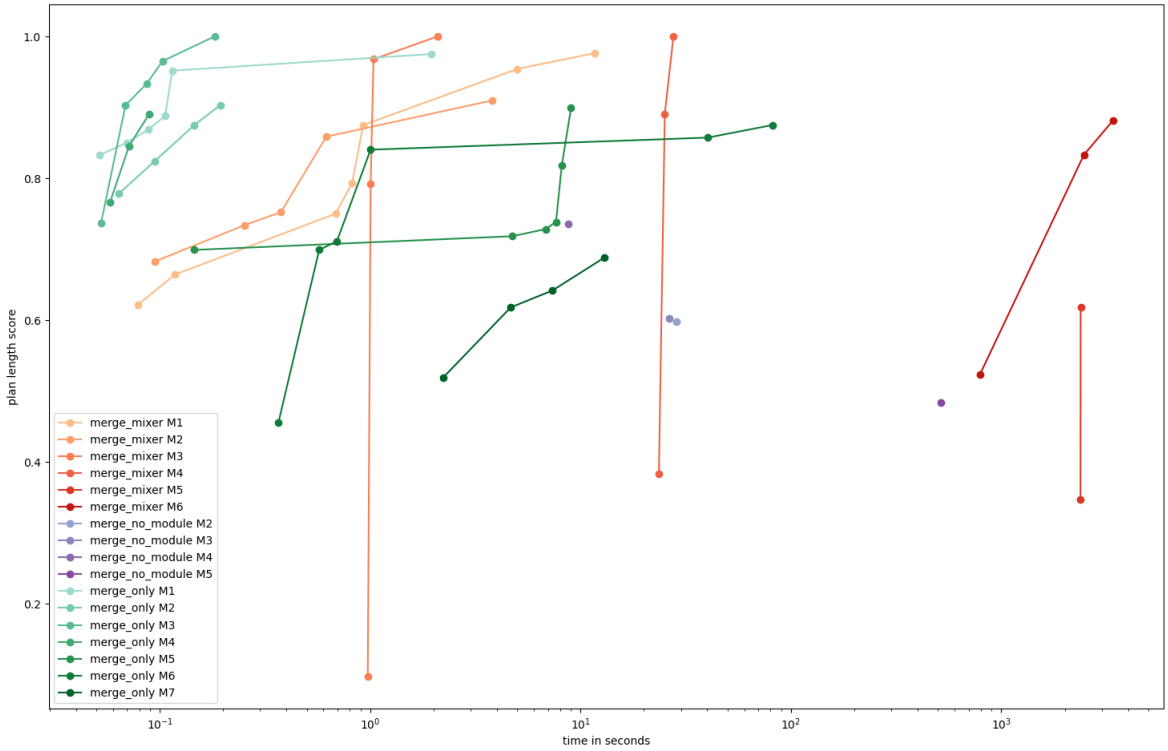
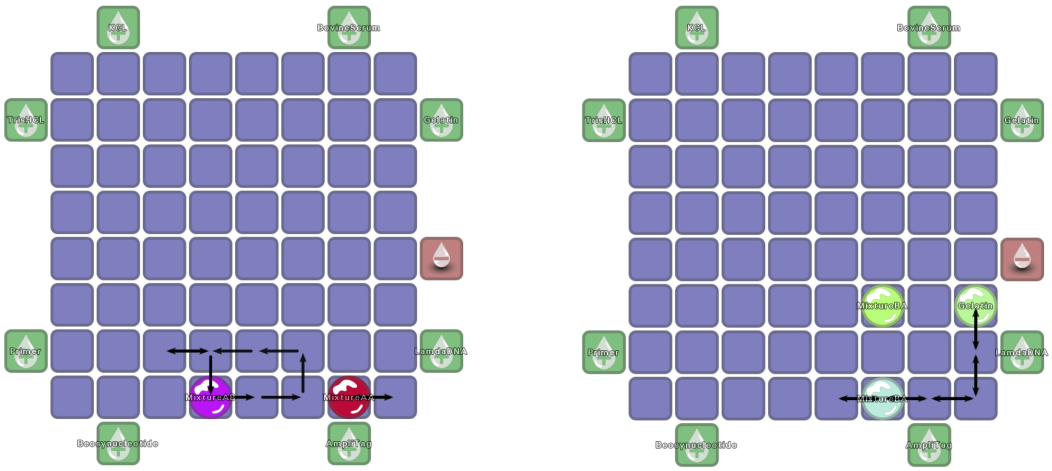


Figure 6.2: Achieved scores per search time. Each color group (orange, purple and green) covers one of the three extensions. M1 to M4 are the easiest problems requiring only one merging operation. M5 and M6 are harder and M7 is the hardest problem (see Figure 6.1).

1st Order	merge-only			merge-mixer			merge-no-module		
	l_{plan}	l_{min}	score	l_{plan}	l_{min}	score	l_{plan}	l_{min}	score
Mixture 1	18	17	0.975	19	18	0.977			
Mixture 2	15	12	0.903	16	13	0.910	101	40	0.598
Mixture 3	12	12	1.000	13	13	1.000	100	40	0.602
Mixture 4	9	7	0.891	14	14	1.000	79	43	0.736
2nd Order	merge-only			merge-mixer			merge-no-module		
	l_{plan}	l_{min}	score	l_{plan}	l_{min}	score	l_{plan}	l_{min}	score
Mixture 5	29	23	0.899	82	34	0.618	420	128	0.484
Mixture 6	24	18	0.875	42	32	0.882			
3rd Order	merge-only			merge-mixer			merge-no-module		
	l_{plan}	l_{min}	score	l_{plan}	l_{min}	score	l_{plan}	l_{min}	score
Mixture 7	80	39	0.688						

Table 6.1: Exact results from the extensions test. The plan lengths of the best found plans are listed under l_{plan}. Under l_{min} the optimal plan length is given. Empty cells indicate that no plan was found for this extension and mixture.

droplet - but is then moved back every time. Optimally the mixing percentage would be increased by 15% for every five moves, reaching 90% after 30 moves and then 100% after 34 moves. This plan increases the mixing percentage by 5% per 10 moves, reaching 100% after 200 moves.



(a) mixing M2 (purple)

(b) mixing M5 (turquoise)

Figure 6.3: Depiction of the moves made in the plan for M5 of the "Merge-No-Module" extension. This representation is from the BioViz program [21] and is made by converting the generated plans into .bio files compatible with BioViz.

A similar inefficiency is found when mixing M5, displayed in Figure 6.3(b) (light blue droplet). Here an increase of 10% per 10 moves is made. The same behaviour is seen in the mixing of M1.

The "Merge-Mixer" extension also has low score for mixture 5. The main reasons for this are that the two reagents for mixture 2 are both moved to the lower right corner before being mixed and that some unnecessary back and forth movement is made afterwards.

Finally when looking at mixture 7 in the "Merge-Only" extension the biggest issue that affects plan length is the spawning of an extra TrisHCL droplet. It is not used in any mixture, clutters the chip and therefore needs some extra moves to get out of the way of other

droplet movement. In all found plans the dispose action was never used.

These issues are not caused by our model. If the planner would find better plans they would be forced to avoid those inefficient moves. When scaling up the problem complexity it gets harder for the planner to find high quality plans. Our model allows and favors efficient plans but it is hard for the planner to find them.

7

Conclusions and Future Work

Our initial motivation for this work was to use temporal planning for droplet routing on DMFBs. When trying to model the problem in PDDL, we quickly realized that the temporal aspect is not strictly needed. The main difference between temporal and classical planning is the metric they optimize for. Temporal planners minimize the makespan which is the time between the start of the first and the end of the last action of a plan. Classical planners minimize the plan cost which is the sum of the cost of all actions in a plan. By setting the cost of all classical actions to one and the duration of all durative actions also to one we can bring the two metrics closer together. The main difference remains the parallel execution of actions. For a temporal planner it makes no difference if two droplets are moved to their target first and then merge or if they are merged first and the resulting droplet is then moved to the target. A classical planner will prefer to mix the droplets first as this will reduce the number of actions. There is no advantage though in delaying the merging action of two droplets. One might even argue that merging droplets late clutters the chip unnecessarily. Therefore we see no significant advantage in using temporal planning over classical planning.

In our tests for droplet routing we achieved good results. We were able to consistently find good plans fast. We tested different configurations and compared them in terms of plan quality and search time. The classical planner generally outperformed the temporal planner, sometimes significantly. Overall planning proves to be well suited for the task of droplet routing on DMFBs.

As planning is very flexible we tried to model more aspects of DMFB design into our planning domains. In a first extension we added the ability to spawn, dispose, merge and split droplets. This extension achieved good results being able to solve even more complicated problems. However the found plan for the full PCR experiment is far from optimal.

The next extension is using predefined mixing areas to mix merged droplets. With this addition we are very close to implementing full real-life experiments. Finding plans for this extension took significantly longer than when mixing is ignored. For the hardest problem, the full PCR, no plan was found. With this we fall just short of finding a plan for an entire, physically executable experiment.

The third and final extension then discards the concept of mixing modules and replaces it

with mixing “on-the-go”. This approach proved to be too ambitious as often no plans are found and the plans that were found are inefficient.

There are still many areas that can explored that we did not or only partially cover. We did not do any pin assignment or escape routing. We only did mixing with fixed position mixing modules but finding areas for mixing dynamically can lead to completely different and new plans. A strength of planning domains is that further extensions such as pin assignment can easily be added.
One aspect that is present and dominant in all domains is symmetry [19]. The grid structure of DMFBs leads to different chains of actions that have the same cost, origin state and goal state. This inflates the search space with many states that lead to plans with the same cost. We do not perform any kind of symmetry breaking that would favor some routes over others. It would be interesting to see how much of an impact introducing symmetry breaking would have on the performance of the planners.

Bibliography

[1] Biddut Bhattacharjee and Homayoun Najjaran. Size dependent droplet actuation in digital microfluidic systems. In *Micro-and Nanotechnology Sensors, Systems, and Applications*, volume 7318, pages 96–107. SPIE, 2009.

[2] Karl F. Böhringer. Towards optimal strategies for moving droplets in digital microfluidic systems. In *IEEE International Conference on Robotics and Automation Proceedings. ICRA'04.*, volume 2, pages 1468–1474, 2004.

[3] Sarit Chakraborty and Susanta Chakraborty. An efficient module-less synthesis approach for digital microfluidic biochip. *SN applied sciences*, 2(8), 2020.

[4] Wen-Chun Chung, Pei-Yi Cheng, Zipeng Li, and Tsung-Yi Ho. Module placement under completion-time uncertainty in micro-electrode-dot-array digital microfluidic biochips. *IEEE Transactions on Multi-Scale Computing Systems*, 4(4):811–821, 2018.

[5] Lu-Ming Duan and Guang-Can Guo. Preserving coherence in quantum computation by pairing quantum bits. *Physical Review Letters*, 79(10):1953, 1997.

[6] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[7] Richard Fair, Vijay Srinivasan, H. Ren, Philip Paik, Vamsee K. Pamula, and Michael G. Pollack. Electrowetting-based on-chip sample processing for integrated microfluidics. In *IEEE International Electron Devices Meeting 2003*, pages 32.5.1–32.5.4. IEEE, 2003.

[8] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL — the planning domain definition language. 1998.

[9] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[10] Oliver Keszöcze, Robert Wille, Tsung-Yi Ho, and Rolf Drechsler. Exact one-pass synthesis of digital microfluidic biochips. In *Proceedings of the 51st Annual Design Automation Conference*, 2014.

[11] Natalie Klco, Martin J Savage, and Jesse R Stryker. SU (2) non-Abelian gauge field theory in one dimension on digital quantum computers. *Physical Review D*, 101(7): 074512, 2020.

[12] Martha F Kramer and Donald M Coen. Enzymatic amplification of DNA by PCR: standard procedures and optimization. *Current protocols in molecular biology*, 56(1): 15.1.1–15.1.14, 2001.

[13] Jeffrey McDaniel, Zachary Zimmerman, Daniel Grissom, and Philip Brisk. PCB escape routing and layer minimization for digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):69–82, 2016.

[14] Michele Mosca. Quantum algorithms. *Computational Complexity: Theory, Techniques, and Applications*, 2008.

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] Frieder Mugele and Jean-Christophe Baret. Electrowetting: from basics to applications. *Journal of physics: condensed matter*, 17(28), 2005.

[17] Michael G. Pollack, Alexander D. Shenderov, and Richard B. Fair. Electrowetting-based actuation of droplets for integrated microfluidics. *Lab on a Chip*, 2(2):96–101, 2002.

[18] Silvia Richter, Matthias Westphal, and Malte Helmert. LAMA 2008 and 2011. In *International Planning Competition*, pages 117–124, 2011.

[19] Jussi Rintanen. Symmetry reduction for SAT representations of transition systems. In *ICAPS*, pages 32–41, 2003.

[20] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation - Theory and Practice.* IEEE, 1995.

[21] Jannis Stoppe, Oliver Keszöcze, Maximilian Luenert, Robert Wille, and Rolf Drechsler. Bioviz: An interactive visualization engine for the design of digital microfluidic biochips. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 170–175. IEEE, 2017.

[22] Fei Su, William Hwang, and Krishnendu Chakrabarty. Droplet routing in the synthesis of digital microfluidic biochips. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 323–328. IEEE, 2006.

[23] Udayan Umapathi and Hiroshi Ishii. Programmable droplets, Jan 2018. URL https://www.media.mit.edu/projects/programmable-droplets.

[24] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Compiling quantum circuits to realistic hardware architectures using temporal planners. *Quantum Science and Technology*, 3(2), 2018.

[25] Qin Wang, Zeyan Li, Haena Cheong, Oh-Sun Kwon, Hailong Yao, Tsung-Yi Ho, Kwanwoo Shin, Bing Li, Ulf Schlichtmann, and Yici Cai. Control-fluidic codesign for paper-based digital microfluidic biochips. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016.

[26] Tao Xu and Krishnendu Chakrabarty. Functional testing of digital microfluidic biochips. In *2007 IEEE International Test Conference*. IEEE, 2007.

[27] Tao Xu and Krishnendu Chakrabarty. Integrated droplet routing in the synthesis of microfluidic biochips. In *Proceedings of the 44th annual Design Automation Conference*, pages 948–953, 2007.