UNIVERSITY OF CALIFORNIA

Los Angeles

Finger-powered Digital Microfluidics for Micro Droplet Manipulation

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Mechanical Engineering

by

Cheng Peng

2017

ABSTRACT OF THE DISSERTATION

Finger-powered Digital Microfluidics for Micro Droplet Manipulation

by

Cheng Peng

Doctor of Philosophy in Mechanical Engineering
University of California, Los Angeles, 2017
Professor Yongho Ju, Chair

Microfluidic devices that do not require bulky peripheral hardware, such as pumps and external battery/power supplies, are a suitable technology for portable applications in resource-constrained settings, such as point-of-care (POC) diagnosis in developed countries, environmental monitoring, and on-site forensic analysis, etc. The existing portable microfluidic devices are mostly based on microchannel structures, in which the pre-defined channels limit their functional flexibility, rendering them difficult to scale up. Digital microfluidics, on the other hand, can tackle this problem since they deal with discrete droplets individually and can therefore provide more on-demand flexibility and versatility. Most digital microfluidic devices, however, require external electric power sources.

We first propose finger-powered digital microfluidic (F-DMF) based on electrowetting on dielectric (EWOD). Instead of requiring an external power supply, our F-DMF uses piezoelectric elements to convert the mechanical energy produced by human fingers into electric voltage pulses for droplet manipulation. The voltage outputs of piezoelectric element mounted in cantilever beam configuration are studied theoretically and experimentally. Using this energy conversion scheme, the basic modes of droplet operations, such as droplet transport, splitting, and merging on EWOD devices are confirmed. The key assay steps involved in glucose detection and immunoassay are also successfully performed using F-DMF-EWOD.

Exploiting the same energy conversion scheme, F-DMF based on the electrophoretic transport of discrete droplets (EPD), which has the potential to overcome pinning and surface contamination often encountered in EWOD, is then presented. Successful EPD actuation, however, requires the piezoelectric elements to provide both sufficient charge and voltage pulse duration. These requirements are quantified using numerical models to predict the electrical charges induced on the droplets and the subsequent electrophoretic forces. The transport and merging of aqueous droplets as well as direct manipulation of body fluids is experimentally demonstrated using F-EPD-DMF. Further, a mechanical system and an efficient pin-assignment scheme are explored to facilitate the practical implementation of pre-programmed and functional actuation of droplets in the EPD-based system.

For the second part of this thesis, one practical issue in digital microfluidics biochip (DMFB) design is discussed: the droplet routing problem, which largely decides the performance and correctness of the system. The problem is formulated to a multi-agent path finding problem (MAPF) and an approximate algorithm based on Independent Detection (ID) is applied to solve the problem. The modified ID algorithm shows promising performance on selected benchmark problems with medium number of droplets (≤12). Overall, it achieves better timing result (~15% reduction) and total routing length (~50% reduction) with no compromise in fault tolerance (indicated by the total number of used cells), when compared with the previous best known results.

The dissertation of Cheng Peng is approved.

University of California, Los Angeles

2017

This work is dedicated to my mom.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This work would not have been possible without the support of many people. First and foremost, I would like to express my sincere appreciation and gratitude to my advisor, Dr. Yongho Ju, for his enthusiastic guidance and support throughout my PhD research, as well as his understanding and support in my future career development. I would also like to thank my committee members, Dr. Chang-Jin Kim, Dr. Pei-Yu Chiou, and Dr. Qibing Pei, for their valuable suggestions and encouragement.

I would like to thank my former and current colleagues in UCLA Multiscale Thermosciences Laboratory: Youngsuk Nam, Gillwan Cha, Tanye Tang, Katie Bulgrin, Stephen Sharratt, Yujia Zhan, Yanbing Jia, Jinda Zhuang, Yang Shen, Zezhi Zeng, Yide Wang, Chao Fan, Navid dehdari Ebrahimi, Abolfazl Sadeghpour and Quzheng Xian for their help during my stay at UCLA. I would also like to thank members from UCLA Micro-nano Manufacturing Laboratory and mechanical engineering department staff: Supin Chen, Lian-Xin Huang, Jia Li and Mr. Benjamin Tan for their advice and help in device fabrication. Also, I would like to thank Professor Richard Korf in UCLA computer science department, and Yuanlu Xu in UCLA Statistics Department for their advice and help in algorithm design.

Last but not least, I would like to thank my friends: Dan Duan, Luyao Xu, Fangting Xia, Lili Feng, Shengxin Jia and Muchen Xu for their encouragement and continued support over the years. My deepest gratitude goes to my family and my boyfriend for their love and support.

VITA

2011 - 2017	Graduate student researcher/Teaching Associate
	University of California, Los Angeles, USA
2014 summer	Summer Intern
	Sony Life Science Laboratory, Tokyo, Japan
2007 - 2011	B.S. Energy and Power Engineering
	Xi’an Jiao tong University, Shaanxi, China

PUBLICATIONS

C. Peng, Y. Wang, and Y. Sungtaek Ju, “Finger-powered electrophoretic transport of discrete droplets for portable digital microfluidics,” *Lab Chip*, vol. 16, no. 13, pp. 2521–2531, 2016.

C. Peng, Z. Zhang, C.-J. “CJ” Kim, and Y. S. Ju, “EWOD (electrowetting on dielectric) digital microfluidics powered by finger actuation,” *Lab Chip*, vol. 14, no. 6, p. 1117, 2014.

C. Peng and Y. S. Ju, “Finger-powered droplet actuation by electrophoretic force for portable microfluidics,” in *2015 Transducers - 2015 18th International Conference on Solid-State Sensors, Actuators and Microsystems (TRANSDUCERS)*, 2015, pp. 232–235.

C. Peng, C.-J. C. J. Kim, and Y. S. Ju, “Finger-triggered digital microfluidics,” in *2013 Transducers & Eurosensors XXVII: The 17th International Conference on Solid-State Sensors, Actuators and Microsystems (TRANSDUCERS & EUROSENSORS XXVII)*, 2013, pp. 388–391.

C. Peng and Y. Sungtaek Ju, “Finger-Powered Electro-Digital-Microfluidics,” in Biosensors and Biodetection: Humana Press, Ch. 20, Vol. 2, Forthcoming 2017.

S. Sharratt, **C. Peng**, and Y. S. Ju, “Micro-post evaporator wicks with improved phase change heat transfer performance,” *Int. J. Heat Mass Transf.*, vol. 55, no. 21, pp. 6163–6169, 2012.

Chapter 1 Introduction

1.1 Portable Microfluidic Devices

1.1.1 Motivation

Microfluidics refers to the technology of manipulating fluids at sub-millimeter length scales. It offers the advantages of low-volume sample consumption, high-throughput fluidic handling, and miniaturization, compared with conventional laboratory tests [1], [2]. Microfluidic devices that do not require external hardware have garnered significant attention as a suitable technology for a wide range of portable applications, such as point-of-care (POC) diagnostics in resource-constraint settings [3], wearable monitoring devices [4], on-site environmental analysis [5], and forensic analysis [6], [7], where a central laboratory and trained workers, as well as access to a reliable electric supply are unavailable.

However, the powering for various microfluidic components often required for fluidic control, such as micropump, microvalve, micromixer, and micro-separator, is becoming a major engineering challenge for portable applications [8]. Therefore, a fluid manipulation mechanism that does not require external pumping systems, power supplies, or other external support equipment is the key to the successful development of portable microfluidics.

1.1.2 Existing Portable Microfluidic Devices

The majority of existing portable microfluidic devices are based on microchannel structures, and can be categorized into three types based on their fluid pumping schemes: “passive” pumping, “active” pumping powered by battery, and powered by energy obtained from the surrounding environment (including human) [8].

Passive pumping

Capillary action is the ability of a liquid to flow in narrow channels without the help of external force and is a popular mechanism, which is widely exploited in passive pumping. For example, paper-based microfluidic devices use capillary action to guide fluids along the patterned “channel” on paper (Figure 1.1(a)). Recent studies further improved the fluid handling accuracy of paper-based microfluidics through the use of different components, including a magnetic-valve for timed fluidic control [9] and a delay valve made with sugar for the sequential delivery of fluids [10]. Using these fluidic components, various biomedical applications of paper-based microfluidics, such as the quantification of the nitrite levels in saliva [11] and the separation of hemolysis-free blood plasma [12], have been demonstrated. As an alternative to paper, different materials have also been proposed to improve the mixing further [13-15].

Degas-driven flow is another passive pumping mechanism that takes advantage of the free volume of PDMS. The potential energy of the evacuated PDMS drives the absorption of air in the sealed side of the microchannel, and thus pumps the fluid in the channel, as shown in Figure 1.1(b)[16-18].

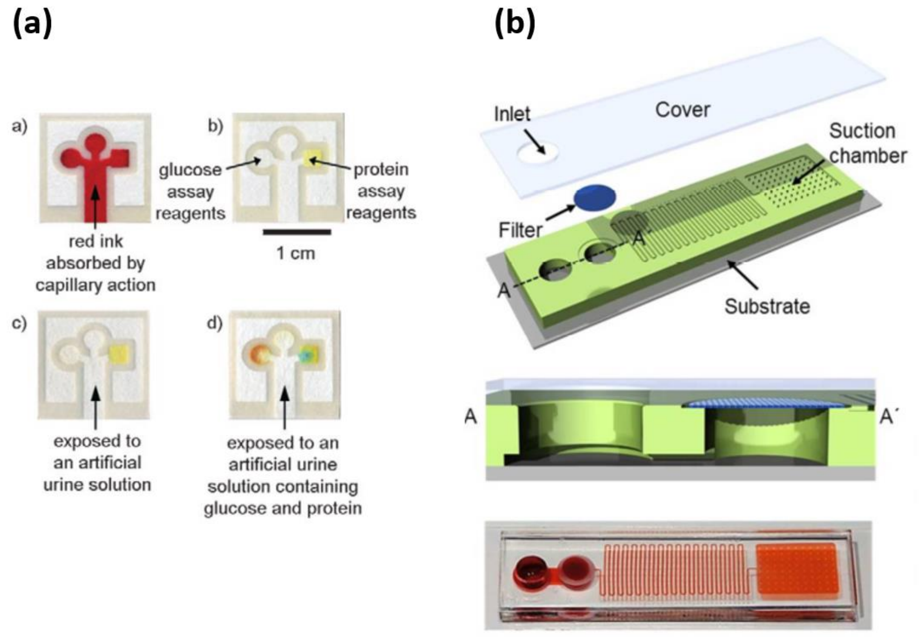


Figure 1.1 (a): Paper patterned with photoresist to actuate the fluid through capillary action and to carry out glucose assay [19]; (b): Schematic of a degas-driven microfluidic chip device for on-chip blood separation [17].

Active pumping powered by battery

Sometimes, active microfluidic components are necessary for precise fluid handling, which is required for certain bio-medical assays to achieve high precision and efficient reactions [8]. Developing low power consumption components that can be powered by the on-chip battery is one way to pursue this goal. For example, solenoid actuators and electro-osmosis (EO) pumps powered by a small powering unit are implemented on chips as valves and pumps for fluid injection and control [20] (Figure 1.2(a)). Similarly, a handheld microfluidic device incorporating multiple elastomeric microvalves and powered by a 9-V battery was demonstrated for horseradish peroxidase (HRP) amplification (Figure 1.2(b)) [21]. The use of battery to power the microsyringe pumps integrated in the portable system for fluid actuation, has also been reported [22].

Another interesting method for eliminating the bulky pumping system is to use centrifugal force, generated by the rotation of the microfluidic disk [23], [24]. These systems usually depend on a step motor for actuation, which can be potentially powered by a battery for portable applications.

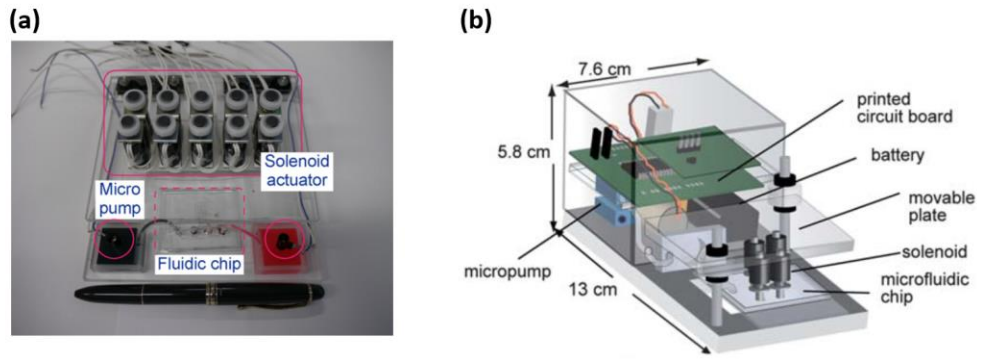


Figure 1.2 (a): A portable microfluidic device for potential parallel cell analysis. A lower power consumption is achieved with 10 solenoid actuators as valves and several electro-osmosis (EO)

pumps for fluid control [20]; (b): Schematic of a battery operated microfluidic system employing elastomeric valves for fluidic control [21].

Active pumping powered by environment/human

Reducing the power consumption can hardly be the ultimate solution, if the key components of microfluidics are to be adopted [8]. Energy conversion modules, combining various energy harvesters with a small rechargeable battery or other energy storage system, have been reported to replace the traditional battery for autonomous power supply [25], [26]. Table 1.1 summarizes the common types of energy sources, the approximate magnitude of the returned power, and their corresponding advantages/disadvantages [25], [26]. It is shown that vibration-based mechanical energy is more accessible than solar and thermal energy. As an example, the energy harvested from the movement of human body with a triboelectric nanogenerator is used to charge the lithium battery that powers glucose biosensors and water/ethanol detectors [27], [28].

Table 1.1: Source and harvested power for different energy sources [25], [26].

Source	**Harvested Power**	**Advantage/disadvantage**
Ambient light		High power
Indoor	~ 10 $\mu W/cm^2$	
Outdoor	~ 10 mW/cm^2	
Vibration/kinetic (Human)	~ 4 $\mu W/cm^2$	High accessibility
Thermoelectric (Human)	~ 30 $\mu W/cm^2$	Environmental dependent
Fuel cells	~ 50 $\mu W/cm^2$	*In vitro*

Energy conversion modules can also be used in POC devices to charge the battery, or even better, to directly manipulate fluids through various energy forms. A “light-driven” flow exploits the change in wettability of a polymer material with temperature, to pump and control fluids. The temperature difference is generated by the ambient light (Figure 1.3(a)) [29]. Moreover, the mechanical energy of the human body is directly used to provide the hydraulic pressure for fluid pumping in microchannels. Figure 1.3(b) shows a finger-powered device for continuous droplet formation and transportation [30]–[32]. Figure 1.3(c) illustrates a novel spring-driven mechanical actuator for the automation of fluid control in a lateral-flow based cassette through the opening/closing of a series of pre-evacuated air pouches [33]. A reprogrammable punch-card based portable microfluidic device has also been designed for low-cost and reconfigurable applications [34].

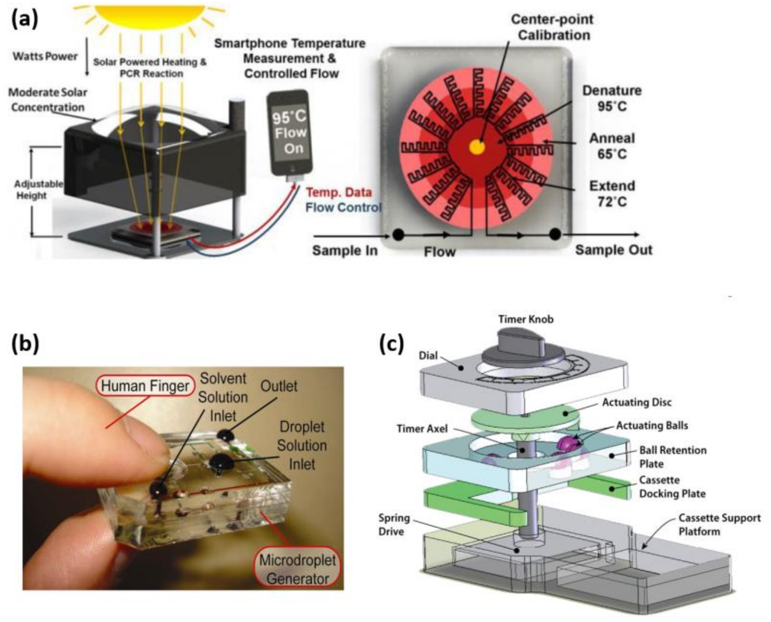


Figure 1.3 (a): “Light-driven” microfluidics for continuous-flow PCR, which exploits the change in wettability of a polymer material with temperature [29]; (b): Finger-powered droplet

microfluidic device based on hydraulic pressure, having an integrated deformable chamber that can be activated by a human finger press to pump multiple streams of fluids [32]; (c): A timer-actuated immunoassay cassette for biomarker detection in oral fluids. When the dial rotates under spring-force, the actuation balls pushes the on-chip air pouches to drive the fluids in the lateral flow chips [33].

1.2 Digital Microfluidics (DMF)

1.2.1 Digital-microfluidics and Applications

Digital microfluidics (DMF) is an intriguing alternative to channel-based (continuous) microfluidics, though most of the existing portable microfluidics are based on the latter. Digital microfluidics use discrete droplets to perform fluidic functions and possesses several salient features that are not present in continuous microfluidics. One is to allow various processes to perform in parallel within an often-compact design, providing better capability to scale up. The other is its reconfigurability, which allows researchers to design a diverse set of biomedical assays using one chip for the most part, offering more flexibility.

The most prominent example of DMF is electro-wetting on dielectric (EWOD) [35]. Figure 1.4(a) shows a typical EWOD device in closed configuration; the bottom substrate is patterned with electrodes (normally Au) using photolithography, and followed by a dielectric layer and a hydrophobic layer. A droplet is sandwiched between the top and bottom plates, and the top plate is usually coated with conductive ITO followed by a hydrophobic layer. The actuation force for the droplet is provided by the change in surface tension between the polarized/conductive droplet and the dielectric layer, which is often coated with a hydrophobic layer, when a voltage is applied between the two. Recently, EWOD is being widely used in many biomedical applications, including immunoassays (e.g. ELISA) [36-39], DNA-based applications (e.g. PCR) [40], [41], cell-based applications [42], [43], and chemical and enzymatic reactions (e.g. glucose) [44-46], [47]. A compact EWOD system has also been developed for point-of-care (POC) diagnostics, though it still requires DC plugin (Figure 1.4(b)) [48].

Other techniques of DMF include dielectrophoresis (DEP) [49], surface acoustic waves [50], magnetic force [51], [52], and thermocapillary force [53], the details of which are introduced elsewhere. For most DMF, however, an external powering unit is necessary for droplet actuation and control.

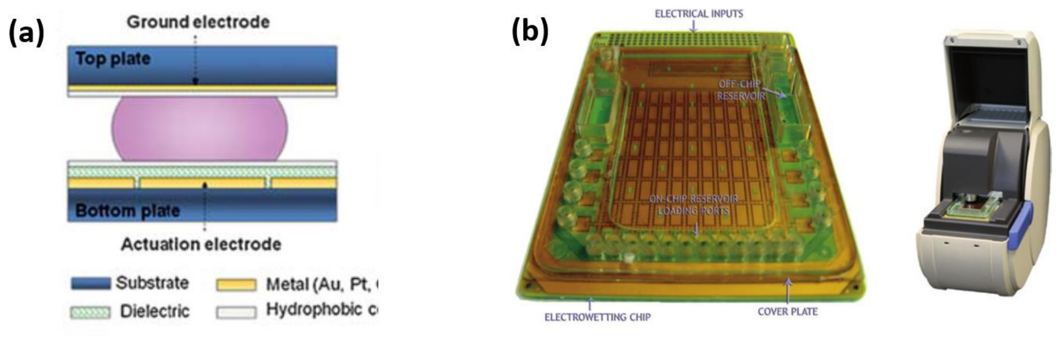


Figure 1.4 (a): Electrowetting on dielectric (EWOD) in closed configuration [54]; (b): A compact digital microfluidic platform for point of care (POC) testing [48].

1.2.2 Design Automation for DMFB

The digital microfluidic biochips (DMFB) design process refers to the conversion of biomedical protocols to an efficient chip design, through a series of sub-phases such as scheduling, resource placement, droplet routing (motion planning), pin assignment, and wire routing, etc. [55], [56].

Droplet routing is a key design issue for performing a large number of operations on a 2D biochip. The goal is to transport each droplet from its start position to the goal position, while ensuring that the droplets in motion do not accidentally collide with one another or with any functional modules on the chip. Many droplet routing methods have been developed since last decade, partially taking advantage of the strategies used in very-large-scale-integration (VLSI). Some of the early efforts include a two-stage algorithm that utilizes maze routing followed by random selection and re-scheduling for the selected paths [57]. A prioritized A^* search is reported by identifying the problem's resemblance to the multi-agent path problem (MAPF), and arbitrarily assigns priority to each droplet [58]. Many of the recent algorithms further improve the routing performance as well as the ability to solve harder problems with larger cardinality (evaluated by the number of droplets). A high-performance droplet routing algorithm based on *bypassibility* was reported, in which the droplet with high *bypassibility* is routed first and the possible deadlocks are solved by backing off some droplets followed by a final compaction step [59]. The global moving vector was proposed together with an entropy-based function for determining the routing order; the reported results show further reduction in the latest arrival time and the used cells [60]. A novel algorithm for concurrent path allocation to multiple droplets based on the Soukup's routing algorithm was also discussed, showing quite encouraging results [61]. In addition, a two-phase routing was proposed by defining a new metric of Interfering Index (II_{net}) and routed droplet with low II_{net} in the first phase, and another metric Routable Ratio (RR) for routing the remaining droplets in the second phase [62]. The test cases, however, are limited to problems with small cardinality. Whereas the algorithms mentioned above lead to heuristic results, "exact" methods targeting optimal solutions have also been reported. The latter mainly tackles the problem by formulating it to an Integer Linear Programming (ILP) problem [63-65] or a Boolean satisfiability (SAT) problem [66], [67]. Further efforts have also been made to tackle the problem of cross-contamination in routing heterogeneous droplets [68-71].

Another design issue of great practical importance is the efficient pin assignment (electrode addressing), which allows one pin to control multiple electrodes on a chip, thus significantly reducing the number of electrical interconnects and manufacturing difficulty. Given the droplet routing results, both broadcast addressing [72] and trace-based-partition method [73] can be used to reduce the number of pins needed, compared with the more straight-forward direct addressing strategy. Broadcasting addressing aims to group compatible electrodes together by examining its activation sequence. On the other hand, trace-based-partition method is based on the *"Connet-5"* algorithm, which automatically partitions the microfluidic array and each pin is assigned to one partition [73] (Figure 1.5(a)). Cross-referencing is another strategy used specifically for EWOD pin assignment, where an electrode on the microfluidic array is connected to two pins, one corresponding to a row and the other to a column (Figure 1.5(b)). Therefore, multiple droplets can be activated simultaneously on a cross-referenced EWOD chip. This method, however, is prone to unintentional droplet manipulation and high power consumption [74], [75]. In addition, a general-purpose pin assignment algorithm, which is not application specific, was developed in [76], [77].

It is also noticed that by combining two or more synthesis processes in the design stage, more optimal solutions can be found. For example, three objective functions including the total number of control pins, electrode usages, and routing completion time are considered simultaneously for co-optimization by combining the droplet routing and pin assignment stages [78], [79]. ILP has also been presented to solve the pin-constraint routing for DMFB in problems with small cardinality [80], [81].

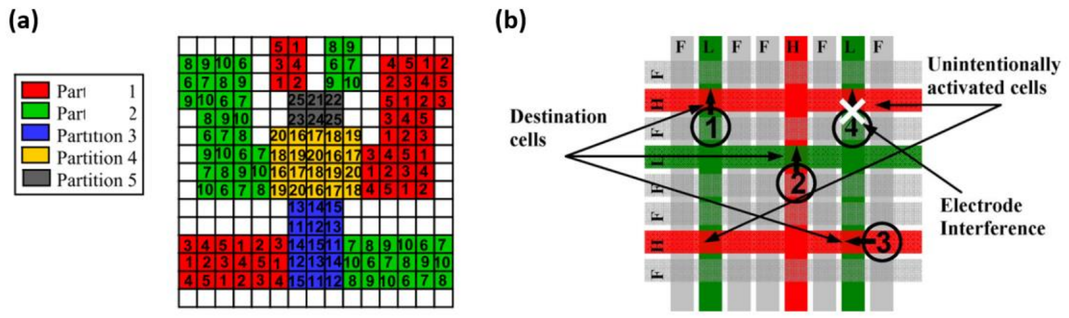


Figure 1.5 (a): Partition and pin assignment results for the multiplexed bioassay using droplet trace based array partitioning techniques and *"Connect 5"* algorithm [73]; each color represents one partition and the number indicates the different pins assigned; (b): Cross-referencing for pin assignment, droplet 1, 2, and 3 are activated simultaneously by one couple of the electrode, and droplet 4 indicates false inference [74].

1.3 Scope of Research

The present work investigates finger-powered digital microfluidics for fluid manipulation and one practical problem in the DMFB design: the droplet routing problem.

Chapter 2 presents electrowetting-on-dielectric (EWOD) digital microfluidics, which is powered by the electric energy converted from the mechanical energy of human finger press, through an array of piezoelectric elements. The output voltage of the piezoelectric element under different bending angles is characterized both theoretically and experimentally. EWOD devices of different thickness are then fabricated using the standard photolithography technique, and their actuation voltages are measured experimentally. Using finger actuation, the basic modes of droplet manipulation, such as transport, merging, and splitting are demonstrated, and key steps of bioassays are performed on EWOD devices.

Chapter 3 explores the electrophoretic transport of discrete droplets (EPD) powered by the human finger, exploiting the same energy conversion scheme as in Chapter 2. Finite element models are developed to predict the induced droplet charges and the subsequent electrophoretic force for a range of droplet sizes, electrode pitches, and actuation voltages, which are anticipated in typical microfluidics applications. The model helps in establishing the engineering criteria for successful EPD actuation. The EPD actuation of various aqueous and body fluids powered by human finger are experimentally demonstrated. Further, an auxiliary mechanical system is developed to facilitate the control of simultaneous deflection of multiple piezoelectric elements for practical implementation. The efficient pin-assignment scheme is further investigated, specifically for the EPD in our application to reduce the number of piezoelectric elements required. Both strategies are adopted for the experimental demonstration of the pre-programmed functional actuation of droplets on a 4×4 base electrode matrix.

Chapter 4 discusses a droplet routing algorithm based on Independent Detection (ID) to reduce the assay execution complexity (time), which is critical to the implementation of proposed portable devices. The problem of concurrent droplet routing on digital microfluidic chips is formulated to a multi-agent path-finding problem, with cost functions consisting of the total routing length and used cells. An approximate version of the original ID algorithm is implemented to solve selected hard test benchmarks with medium number of droplets. Finally, the key evaluation metrics including the latest arrival time and used cells are reported, and our results are compared with those of the state-of-the-art algorithms.

Chapter 2 EWOD (Electrowetting on Dielectric) Digital Microfluidics Powered by Finger Actuation

2.1 Introduction

The electrowetting on dielectric (EWOD) phenomenon is one of the most promising actuation mechanisms used in DMF. The basic fluidic operations such as droplet transporting, splitting, and mixing have been extensively studied through experiments and numerical/analytical models in previous literatures [82][83][84]–[86] [87]. However, the current EWOD devices still require an external high-voltage supply and switching circuitry, which entails custom development to realize compact systems.

In this chapter, the finger-actuated digital microfluidics (F-DMF) based on the manipulation of discrete droplets via the EWOD phenomenon is reported. Instead of utilizing an external power supply, F-DMF-EWOD uses piezoelectric elements to convert the mechanical energy produced by human fingers into electric voltage pulses for droplet actuation. Figure 2.1 schematically illustrates one possible implementation of our device concept, which uses an array of piezoelectric elements to convert mechanical energy pulses provided by human fingers into voltage pulses. Using this scheme, basic modes of droplet manipulation, such as transporting, splitting, and merging of water droplets were performed. Furthermore, to demonstrate its capability for biomedical applications, the key assay steps involved in glucose detection and an immunoassay were successfully performed.

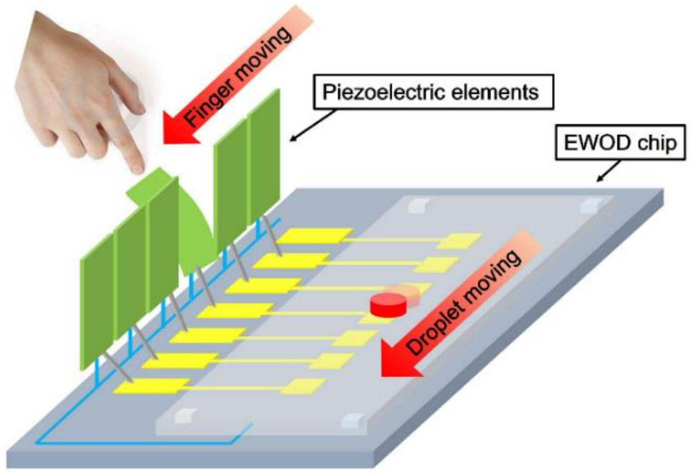


Figure 2.1: Schematic of one possible implementation of the finger-actuated digital microfluidic platform. The piezoelectric elements convert the mechanical energy imparted by human fingers into electrical energy to actuate the droplets confined between two parallel plates, through the EWOD phenomenon.

2.2 EWOD Device

2.2.1 Device Design and Fabrication

To successfully manipulate micro-droplets using EWOD, voltage pulses of sufficient amplitude must be generated to overcome the capillary (contact-line hysteresis), inertial, and viscous forces. The EWOD-induced contact angle change is related to the applied voltage by the Lippmann-Young equation:

$$cos\theta(V) - cos\theta_0 = \frac{\varepsilon_0 \varepsilon}{2\gamma_{LG} t} V^2 \qquad (2.1)$$

where θ_0 denotes the equilibrium contact angle at $V = 0$V, ε_0 is the permittivity of vacuum, ε is the dielectric constant of the dielectric layer separating the droplet from the electrode, t is its thickness, and γ_{LG} is the surface tension between the droplet and the surroundings. Since the contact angle change represents the actuation force along the surface, the higher the applied voltage, the stronger the actuation force that will drive the droplet against the above-mentioned resistant forces. As the capillary resistance that originates from the contact angle hysteresis of an aqueous droplet surrounded in air (as opposed to the popular oil environment) is larger than the inertial or viscous resistance in most cases, the performance of EWOD devices is often measured in the air environment without resorting to the filler oil or oil impregnation [87]. To manipulate water droplets in air, typical EWOD devices require voltage source of about 40 V [82].

Figure 2.2 shows the schematic of the cross section of our EWOD device, when a single droplet is being actuated. Our EWOD devices consist of two parallel glass plates that were separated by approximately 100 μm. The bottom glass plate contains an array of 1×1 mm^2 gold electrodes, which was fabricated using standard micro-fabrication processes. A 20-nm Cr/100-nm Au layer was first deposited on a glass wafer and the layer was patterned by wet etching. A dielectric layer of silicon nitride was then deposited by PECVD (plasma enhanced chemical vapor deposition). Next a solution of Teflon® AF (2% wt/wt in Fluorinert FC-40) was spin coated at 2000 rpm for 30 s, and it was baked at 180 °C for 10 min to obtain a ~100 nm-thick hydrophobic topcoat. A shadow mask was used to define the electrical contact pads. The top glass plate was coated with a transparent conductive ITO (indium tin oxide, <15 ohm/square) layer to form a counter electrode for EWOD. The counter electrode was also coated with a ~100 nm-thick layer of Teflon®. Figure 2.3 shows the optical image of the assembled EWOD device from the top. The EWOD electrodes on the bottom plate are connected individually to larger electrical contact pads to facilitate soldering or other wired connections to the piezoelectric elements. A silver paste was used to make an electrical connection between the counter electrode on the top plate to the ground.

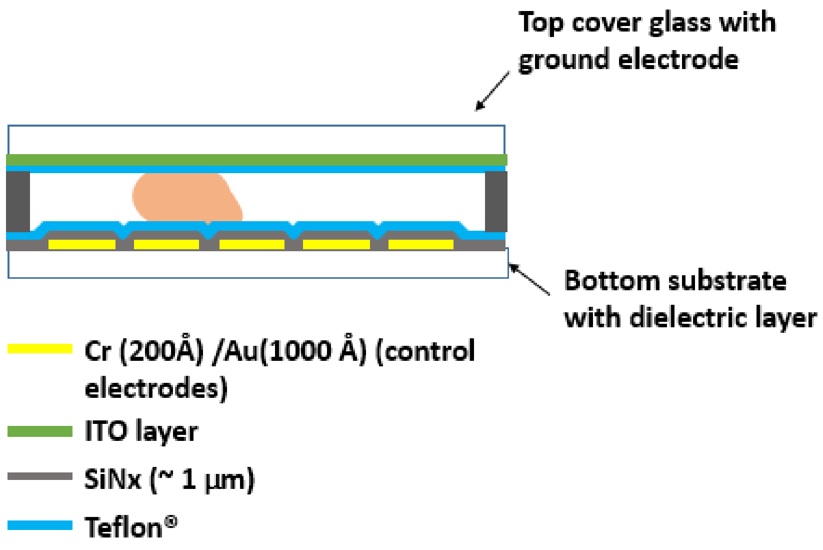

Top cover glass with ground electrode
Bottom substrate with dielectric layer
Cr (200Å) /Au(1000 Å) (control electrodes)
ITO layer
SiNx (~ 1 μm)
Teflon®

Figure 2.2: Cross section of the assembled EWOD device, illustrating the top plate with the transparent conductive ITO layer and the bottom plate with the metal electrode array.

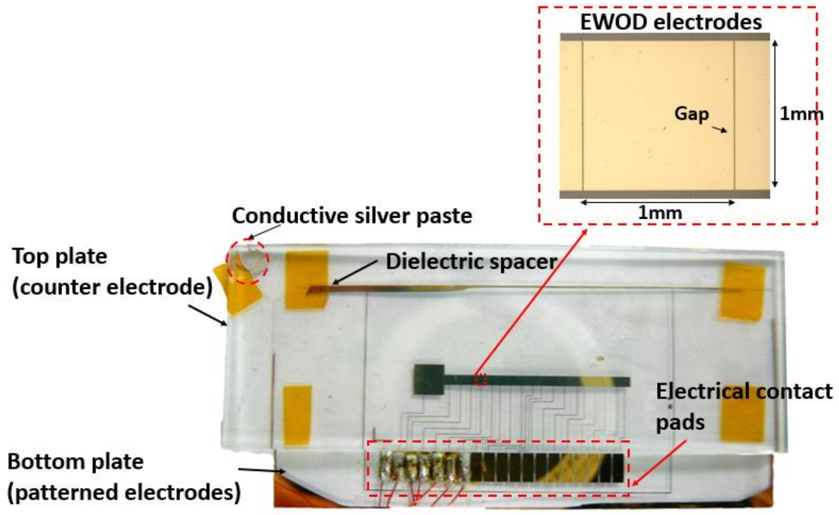


Figure 2.3: Top view of an assembled EWOD microfluidic device and an enlarged optical image of a part of the bottom EWOD electrode array. The overall dimension of the device is ~ 6 × 2.5 cm^2.

Due to the finite energy-conversion efficiency of our piezoelectric elements and the safety consideration for portable applications, a low actuation threshold voltage, and therefore a thin dielectric layer is desired. However, a thin dielectric layer is less robust and prone to pinholes and other defects, which lead to electrolysis-induced failure of the EWOD devices. The dielectric layer thickness is further limited by the capacitance allowed per EWOD electrode, which must be kept below that of the piezoelectric element to minimize the voltage dividing effect. With the above considerations in mind, SiN_x layers of thicknesses ranging from 0.8 μm to 2.5 μm were examined as our dielectric layers. The estimated capacitance was approximately 60pF per electrode for the EWOD devices with a 0.8 μm-thick dielectric layer, which is much smaller than that of the piezoelectric element (~ 1nF).

Another feature of our designed EWOD devices is that the entire voltage drop can be considered to be across the dielectric layer, due to the much smaller thickness of the hydrophobic coating layer compared with that of the dielectric layer in series connection; the voltage drop across the hydrophobic coating layer can be ignored.

2.2.2 EWOD Actuation Voltage

To characterize the threshold actuation voltage required for EWOD actuation as a function of the dielectric layer thickness, an external programmable power source was used to apply precisely defined voltage pulses. A water droplet of ~ 0.3 μL in volume was spotted onto the EWOD device and subsequently split into two nominally identical daughter droplets. After one of these droplets was positioned on one of the electrodes, the amplitude of the voltage pulse applied to the adjacent electrode was gradually increased until the droplet was successfully transported.

The threshold actuation voltage was recorded for each dielectric layer thickness and the results are plotted in Figure 2.4 (symbols). The solid line in Figure 2.4 indicates the voltages required theoretically (Equation 2.1) for the contact angle to change from 120° to 70°, which is an empirically determined range for droplet actuation in the given EWOD device. In general, the threshold actuation voltage increases with the increase in dielectric layer thickness. For EWOD devices with a PECVD SiN_x dielectric layer with a thickness of ~ 2.5 μm, a voltage as large as 70 V is required for successful actuation of a water droplet.

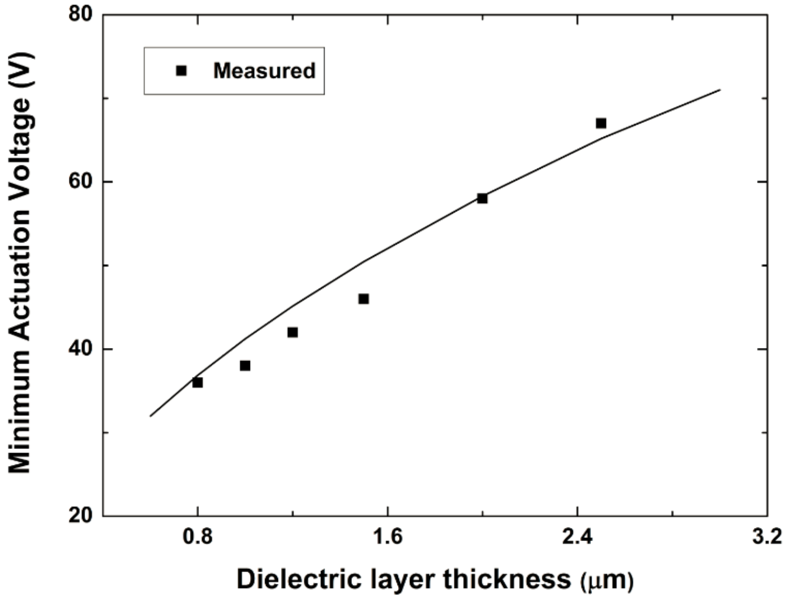


Figure 2.4: Threshold actuation voltage of a water droplet on EWOD devices as a function of the thickness of PECVD SiN_x dielectric layers.

During droplet splitting (cutting), the droplet is elongated in the longitudinal direction due to the actuation by the electrodes on the two sides. The middle electrode is non-activated to keep it non-wetting and the droplet pinches in the middle, as shown in Figure 2.5. The physical parameters describing the process can be expressed as:

$$\frac{R_2}{R_1} = 1 - (\frac{R_2}{d})\frac{\varepsilon_0 \varepsilon V_d^2}{2\gamma_{LG} t} \qquad (2.2)$$

where d is the distance between the top and bottom plates, R_1 and R_2 are the principal radii of curvature as shown in Figure 2.5. At the splitting point, $R_1 = -R_2$, and the critical d/R_2 was calculated for various dielectric thicknesses, and the corresponding actuation voltages measured above. The result shows that the critical d is ~ 200 μm, which is much larger than our separation of ~ 100 μm. Therefore, with actuation voltages exceeding the minimum values calculated above, successful splitting of droplets is expected on our EWOD devices.

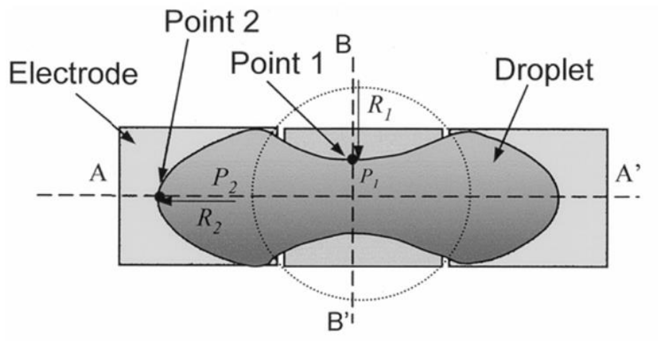


Figure 2.5: Droplet configuration for splitting [82].

2.3 Mechanical Energy Conversion

2.3.1 Piezoelectricity and Material Selection

When subjected to an external stress, the internal structure of piezoelectric material can be deformed, causing the separation of the positive and negative centers of the molecules and generating dipoles, as shown in Figure 2.6. The generated polarization forms an electric field, and the process converts the mechanical energy of the material deformation into electrical energy. Specifically, when electrodes are coated on two sides of the piezoelectric material, an electric voltage will be developed due to the free charge on the surfaces when deformation is generated [88].

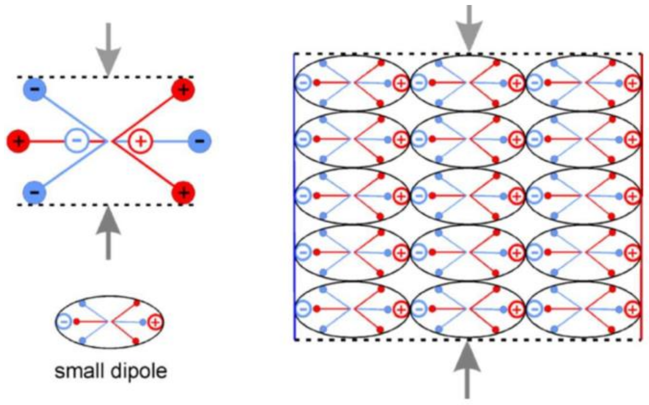


Figure 2.6: Separation of the positive and negative centers of the molecules under deformation in piezoelectric material. The facing polarities inside the material are mutually cancelled, leaving free charge on the surface [88].

Two common types of piezoelectric materials are ceramics and polymers. Traditionally, ceramics such as lead zirconate titanate (PZT) have been widely used for mechanical energy harvesting. However, one disadvantage of most ceramic materials is its extreme brittleness [28]. Moreover, for our finger-actuation application, the magnitude of force that can be imparted directly from human finger is limited. Therefore, materials with large piezoelectric constant g_{31} (3 indicates the common polarization direction), which is proportional to the ratio of the open-circuit voltage to the magnitude of the applied force in the *x* direction, are preferred. Therefore, PVDF (polyvinylidene fluoride), which is one of the most commonly used polymer piezoelectric materials, was chosen due to its large piezoelectric constant (~ 5 times higher than PZT [89]) as well as better reliability compared to ceramic materials.

2.3.2 Modeling of Voltage Output

Piezoelectric elements of 13×25 mm^2 were used to convert the mechanical energy input by human fingers into voltage pulses to actuate the micro-droplets in our EWOD DMF devices. Each piezoelectric element consists of a PVDF layer of thickness 28 μm (piezoelectric layer) laminated on a polyester layer of thickness 125 μm (substrate layer), as illustrated in Figure 2.7(a), to maximize the average strain in the piezoelectric layer and hence its output voltage. The length of the piezoelectric layer L_p is approximately 20.5 mm.

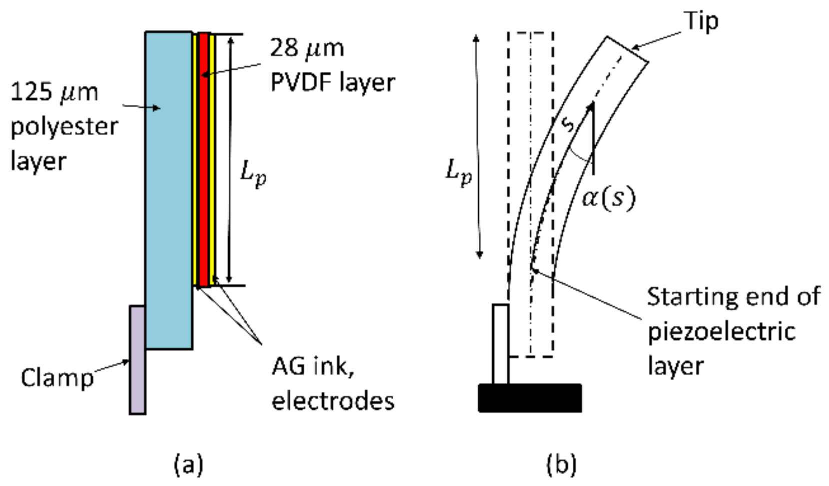


Figure 2.7: (a) Cross section of the main functional layers of a laminated piezoelectric element used in the present study; (b) Definitions of the length L_p and the bending angle α.

Each piezoelectric element was modeled as a Euler-Bernoulli beam. The element (beam) is mounted vertically with the clamped end fixed. The bending angle $\alpha(s)$ is defined as the rotation of the piezoelectric element beam, measured in radians at a distance *s* from the fixed end, as illustrated in Figure 2.7(b). For a single piezoelectric element, the open circuit voltage can be expressed as [90]:

$$V_0 = \frac{e_{31} h_p}{\varepsilon^S} \overline{S_1} \qquad (2.3)$$

Here, e_{31} is the electromechanical coupling coefficient of the piezoelectric layer, ε^S is the permittivity of the piezoelectric layer under constant strain, and h_p is the thickness of the piezoelectric layer. The average strain, $\bar{S}_1$, is defined as:

$$\overline{S}_1 = \frac{\int_{L_p} \int_{h1}^{h2} (\frac{\partial \alpha}{\partial s}) y dy ds}{L_p h_p} = \frac{(\alpha_{tip} - \alpha_0) \int_{h1}^{h2} y dy}{L_p h_p} \qquad (2.4)$$

where h_1 and h_2 are the distances from the neutral axis of the entire beam to the bottom and top of the piezoelectric layer, respectively; α_{tip} and α_0 are the bending angle $\alpha(s)$ at the tip and at the starting end of the piezoelectric layer, respectively (Figure 2.7(b)).

To characterize the energy conversion capability of our piezoelectric elements, the open-circuit voltage outputs were measured using an electrometer of input impedance >200 TΩ. Optical images of the side views of the element were captured to extract the bending angles along the beam.

Figure 2.8 shows the measured and predicted output voltages as a function of the tip bending angle α_{tip} for a single piezoelectric element. The prediction (straight line) agrees reasonably well with the experimental results (symbols) over the entire range. Output voltages greater than 40 V, which is sufficient to actuate a water droplet reliably in the EWOD device with a ~ 0.8 μm-thick SiN_x dielectric layer, can be generated at tip bending angles greater than 80°. However, larger deformations may not be desirable for field operations and may lead to the degradation of the piezoelectric elements. The forces required to achieve bending angles of 36°, 70°, and 108° were estimated to be approximately 0.06, 0.12, and 0.18 N, respectively.

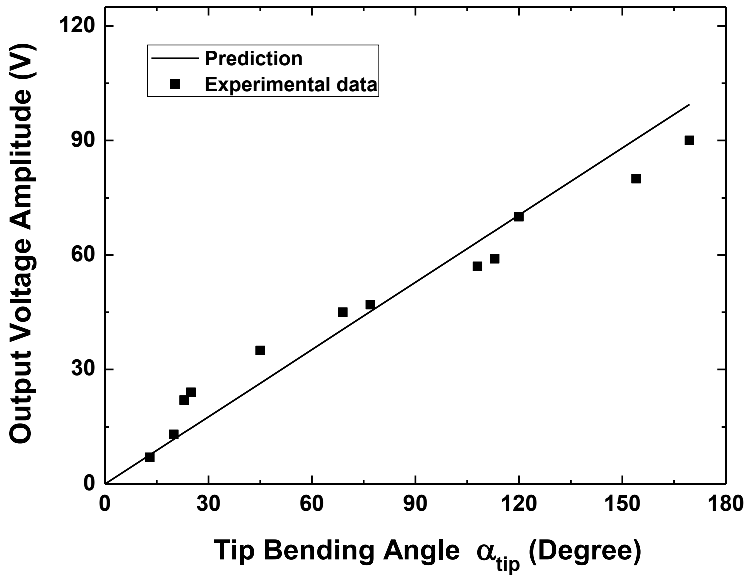


Figure 2.8: Open-circuit output voltages of a single piezoelectric element as a function of the tip bending angle α_{tip}.

If needed, multiple piezoelectric elements may be connected electrically in series and mechanically in parallel to increase the output voltage, while limiting the required deflection to an acceptable range. Figure 2.9 shows the total output voltage of single and two or three piezoelectric elements connected in series. The results are shown for three different bending angles of 15°, 45°, and 90°. At relatively small tip bending angles (~15°), the total voltage output increases linearly with each additional piezoelectric element. At larger tip bending angles (45° and 90°), adding more elements does not lead to a proportionate increase in the output voltage. This could be due to asynchronous bending and finite leakage currents. It is also reported that the parasitic capacitances formed by the insulation layers and derived from peripheral circuitry degrade the output voltages, setting a limit to the maximum output voltage with increasing number of elements in series connection [91]. Nevertheless, it was demonstrated that the output voltages on the order of 100 V can be reliably generated using piezoelectric elements connected in series with tip bending angles <90°.

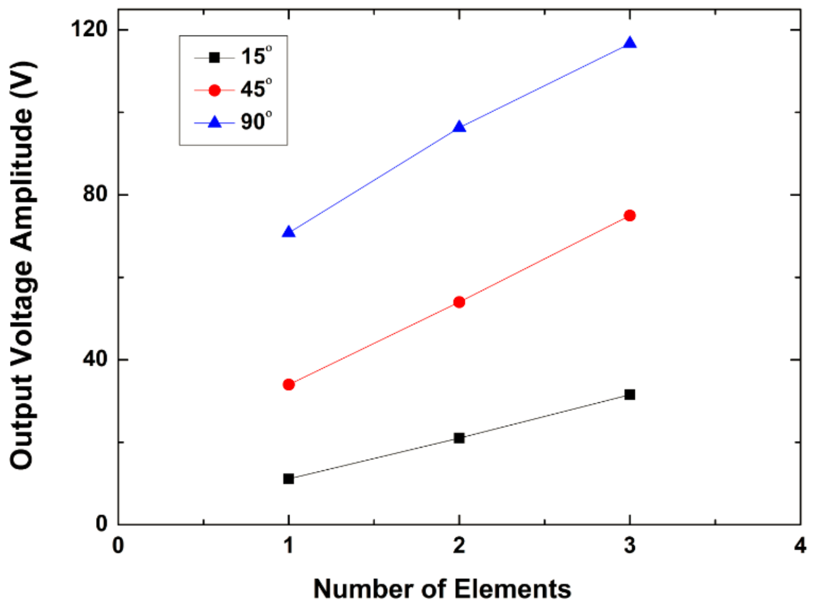


Figure 2.9: Total voltage outputs from multiple piezoelectric elements connected in series under different tip bending angles.

2.4 Basic Droplet Operations

Next, the successful finger-powered actuation of water droplets on EWOD devices with 0.8 μm-thick PECVD SiN_x dielectric layers is demonstrated. An array of single piezoelectric elements was used to convert the mechanical energy into voltage pulses. Figure 2.10 shows the optical images of a single water droplet (~ 0.15 μL) being transported over adjacent electrodes through a sequence of finger-driven deflection of the piezoelectric elements. To prevent droplets from being trapped on an inactive zone between the two adjacent electrodes, the release of the previously bent piezoelectric element is delayed while deflecting the neighboring element. For example, with reference to Figure 2.10, Element 2 was not entirely released when Element 3 was deflected, so that the front contact line of the droplet would stay across the gap between Electrodes 2 and 3. For such a delayed release, a holding force of approximately 0.06 N is sufficient.

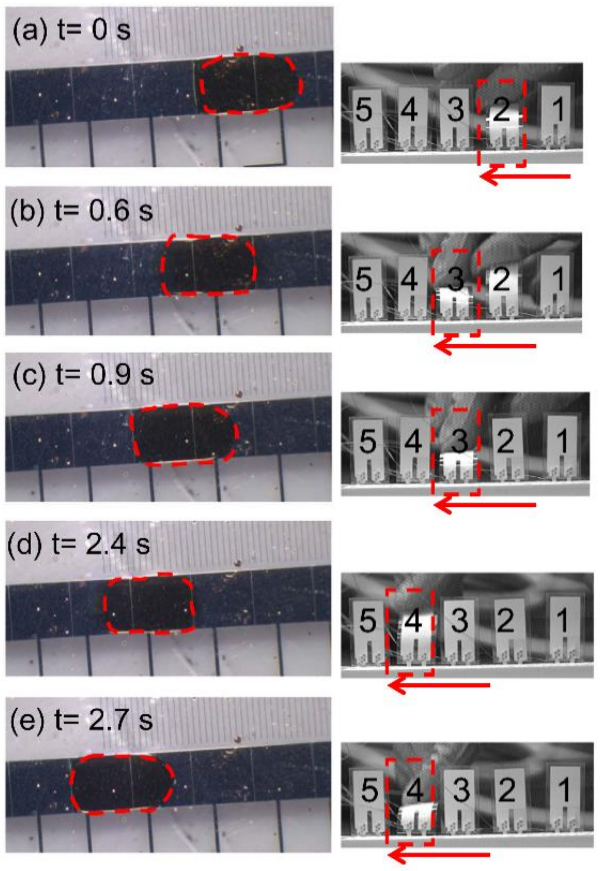
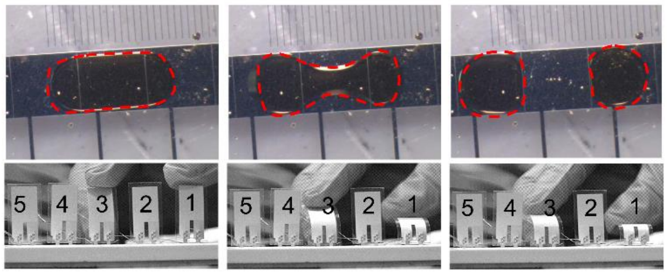


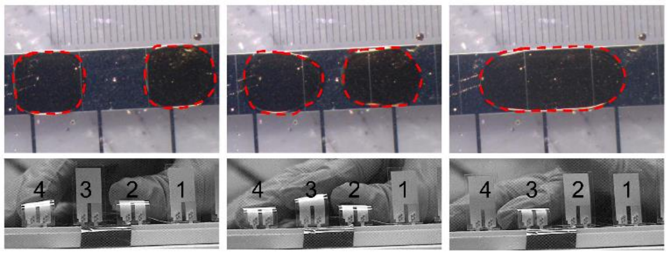
Figure 2.10: Finger-actuated EWOD transport of a water droplet, where the actuation voltage pulses were provided by bending a series of piezoelectric elements.

The droplet splitting was demonstrated by simultaneously deflecting two non-adjacent piezoelectric elements, while keeping the middle one non-deflected. Figure 2.11(a) illustrates the splitting of a water droplet (~ 0.3 μL) into two daughter droplets of similar sizes through the simultaneous deflection of element 1 and 3. When the splitting was in order, the droplet was elongated in the longitudinal direction by the wetting force exerted at the two ends, while the middle was kept non-wetting, as shown in Figure 2.11(a). The actuation voltage on either side was approximately 40–50 V, produced by one single piezoelectric element with a bending angle <90°. For a large droplet that covers multiple electrodes, asymmetric splitting can also be achieved by simultaneously deflecting a set of piezoelectric elements connected to the electrodes in an asymmetric manner, for example, elements 1 and 3, 4.

Figure 2.11(b) shows the merging of two droplets of similar sizes by asynchronous bending and the release of three piezoelectric elements (2, 4, and 3).



(a)



(b)

Figure 2.11 (a): Finger-actuated EWOD droplet splitting. The piezoelectric elements 1 and 3 are bent simultaneously for splitting; (b): Finger-actuated EWOD droplet merging.

2.5 Application to Biological Assays

As proof of principle demonstration of the biological applications of our F-DMF based on EWOD, the basic steps of glucose detection and immunoassay were performed. In these experiments, where higher EWOD voltages are necessary to actuate the droplets, silicon nitride layers of thickness 2.5 μm were used, because they help prevent electrolysis of water under the low-frequency (~1 Hz) finger-driven actuation scheme. Two piezoelectric elements, connected in series, were used to provide an actuation voltage of up to 100 V.

Glucose detection was demonstrated based on enzymatic oxidation, in which the color of an assay solution changes from clear to brown in the presence of glucose. Since the reagent solution may chemically attack the hydrophobic coatings, both the upper plate and bottom substrate of our EWOD devices were pre-treated with silicone oil. The reagent solution was prepared by adding 0.8 mL of o-Dianisidine Reagent to an amber bottle containing 39.2 mL of 1:5 horseradish peroxidase/glucose oxidase solution (15 units of protein per mL of solution). A droplet of 1mg/mL standard glucose solution and another one of reagent solution (both of approximately 0.15 μL) were spotted onto the bottom plate and covered with the top plate. The reagent droplet was then transported towards the glucose sample droplet and merged, as shown in Figure 2.12(b-d). An optical image was taken to verify the color change, which is indicative of the successful enzymatic oxidation reaction. Brown color was observed to start developing after the merging and was fully developed after a 5-min incubation, as shown in Figure 2.12(e).

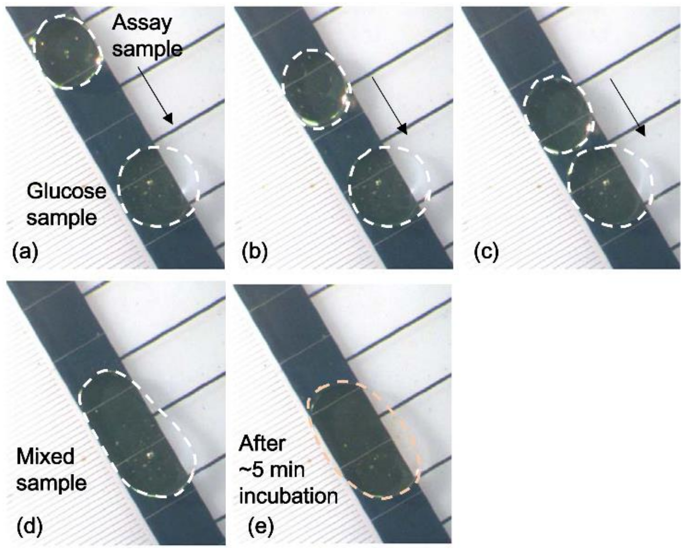


Figure 2.12: Snapshots of sample and reagent droplet during a glucose assay and enzyme catalyzed formation of colored product.

Next, an immunoassay-related enzyme-based colorimetric reaction was performed using the F-DMF-EWOD. 5-bromo-4-chloro-3-indolyl blue tetrazolium (BCIP/NBT) is a commonly used substrate for alkaline phosphate (ALP). In our experiment, the enzyme substrate was used to detect the ALP conjugated antibody. This mimicked the last step of signal detection and amplification in the ALP-based colorimetric ELISA [92]. To prepare the experiment, ALP conjugated IgG antibody (2.5mg/mL) was first diluted to approximately 13 μg/mL. The antibody was immobilized on the upper plate of our EWOD device by manual pipetting of ~500 nL of the diluted solution. The upper plate was chosen due to the smaller interference with actuation, compared with the bottom plate. The spots were allowed to air-dry before use. Approximately 0.15-µL aliquots of enzyme substrate (BCIP/NBT) solution (BCIP: 0.15 mg mL^{-1}, NBT: 0.3 mg mL^{-1}, Tris Buffer: 100 mM, and $MgCl_2$: 5 mM) was loaded onto the bottom plate of the EWOD device, as shown in Figure 2.13(a). Voltage pulses provided by finger-driven actuation were then used to move the sample droplet towards the immobilized antibody spot, as illustrated in Figure 2.13(b-c). After ~5-minute incubation, black-purple precipitates were confirmed to be produced (Figure 2.13(d)), indicating the detection of the ALP-conjugated antibody.

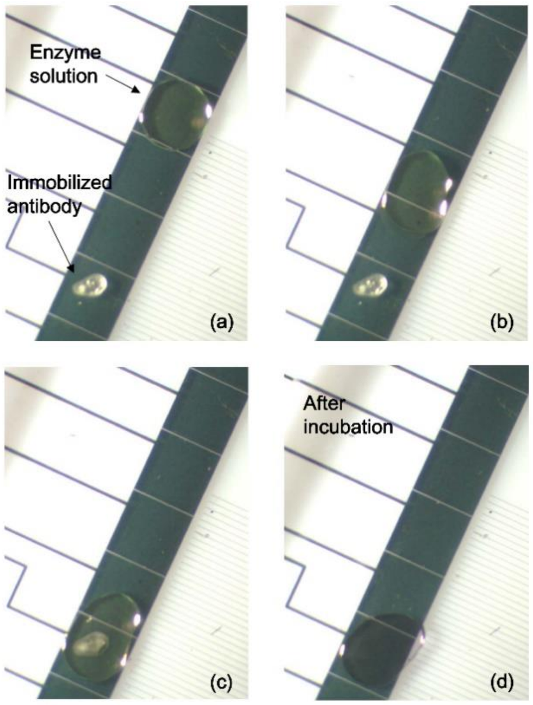


Figure 2.13: (a)-(c) Frames from a video depicting a droplet of BCIP/NBT enzyme solution transported towards immobilized antibody spot upon finger actuation and (d) purple precipitation observed indicating the detection of the ALP conjugated antibody.

2.6 Summary

In this chapter, finger-actuated digital microfluidics based on the EWOD phenomenon was demonstrated using piezoelectric energy conversion of human power. The generation of voltage pulses of amplitudes >100 V were demonstrated using laminated polymer piezoelectric elements connected in series. Using this scheme, the basic EWOD droplet operations such as droplet transport, splitting, and merging were confirmed, and an implementation of the basic assay steps in glucose detection and immunoassay were demonstrated. Due to the low-frequency nature of finger actuation, a relatively thick dielectric layer was used to help prevent possible electrolysis. This work offers a promising solution for expanded applications of EWOD-based digital microfluidics in portable systems.

Chapter 3 Finger-powered Electrophoretic Transport of Discrete Droplets for Portable Digital Microfluidics

3.1 Introduction

Electrophoretic control of discrete droplets (EPD) is a promising alternative approach for digital microfluidics. EPD utilizes the rapid charging of conductive droplets by adjacent electrodes and their subsequent electrophoretically induced motion [93], [94]. Both the droplet and electrodes are typically immersed in a dielectric fluid. This is advantageous, because EPD minimizes direct liquid-solid contacts, compared with other droplet actuation methods such as EWOD [87], thermomechanical [53], and surface acoustic wave (SAWs)-driven [95] actuations.

Previous studies of EPD investigated the electrophoretic force and the resulting trajectories of a droplet suspended between parallel plates or other macroscale electrodes [96-98]. EPD is typically thought to require very high voltages [99], making it ill-suited for portable applications. However, actuation voltages can be reduced to well below 500 V through miniaturization.

In this chapter, the finger-powered EPD digital microfluidics is introduced using a similar energy conversion scheme. An array of piezoelectric elements is connected in parallel to the electrodes immersed in dielectric fluids, as shown in Figure 3.1. When deflected by human fingers, the piezoelectric elements establish an electric field across adjacent EP electrodes to charge and actuate a droplet via electrophoretic force. The numerical models and their experimental validation are reported, to help develop design criteria for successful droplet actuation. The transport and merging of aqueous as well as various body fluids are experimentally demonstrated using our finger-powered EPD. Next, to facilitate practical system-level implementation of our concept, a mechanical system is designed and developed to facilitate controlled deflection of multiple piezoelectric elements. An efficient pin assignment scheme to reduce the number of piezoelectric elements required for practical purposes, is also explored. The pre-programmed functional actuation of droplets on a 4×4 base electrode matrix, integrating our mechanical system and the pin assignment scheme, is experimentally demonstrated.

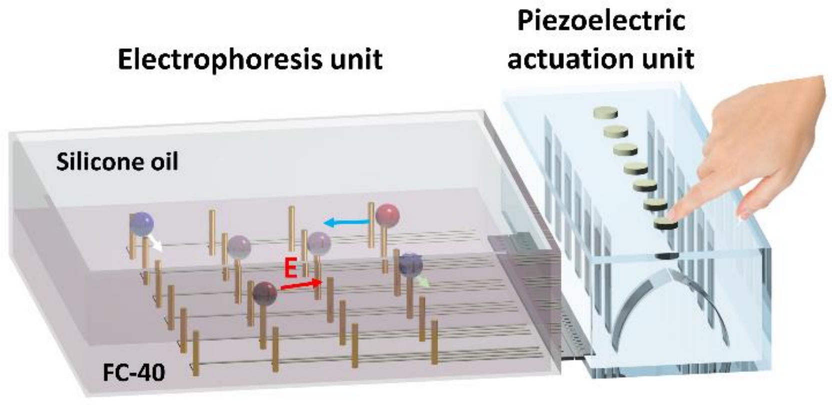


Figure 3.1: Schematic illustration of one implementation of our finger-powered EPD device. An electric field is established across adjacent EP electrodes when the corresponding piezoelectric elements are deflected by human fingers (or finger-powered mechanical levers in an auxiliary mechanical system).

3.2 Experimental Setup

The device, schematically illustrated in Figure 3.1, was used to study finger-powered EPD operations. A transparent acrylic cell is filled with two immiscible dielectric liquids. The two liquids are chosen to have densities and surface tensions such that the spherical aqueous droplets stay near the interface of the two liquids. In this study, silicone oil (DC 200F, υ = 5 cSt, $\sigma = 10^{-13}$ S/m, $\varepsilon = 2.8\ \varepsilon_0$) and Fluorinert FC-40 ($\varepsilon = 1.9\ \varepsilon_0$) were chosen. Insulated copper electrodes of diameter ~0.18 mm are assembled to form an array with pitch distance p. The top surfaces of the electrodes are exposed to allow for droplet charging. Aqueous droplets are placed inside the cell using a micropipette (Eppendorf, 0.1-2.5 μL). Different kinds of aqueous droplets are tested, including those of DI water (ρ = 0.1–1 MΩ), human body fluids (saliva and urine), and a sodium hydroxide solution, covering pH values from 5.6 to 9.

The actuation unit consists of laminated polymeric piezoelectric elements (Measurement Specialist, LDT Series) with active layers (Polyvinylidene fluoride) of thickness 28 μm and size 1.3×2.5 cm^2. A pair of piezoelectric elements are connected in series to increase the voltage output. The negative terminal of each piezoelectric unit is grounded, while the positive terminal is connected to each individual EP electrode, as shown in Figure 3.2. With this arrangement, when the adjacent units are deflected in opposite directions, a voltage differential of approximately 200 V can be generated across the two EP electrodes (see Section 2.3.2). This output is then used to charge a droplet and establish an electric field necessary for droplet actuation.

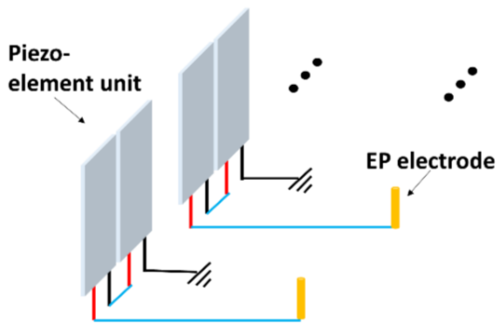


Figure 3.2: Electrical connections used to link the piezoelectric elements to the EP electrodes.

3.3 Droplet Charging and Actuation

3.3.1 Modeling of Induced Droplet Charge and Electrophoretic Force

For the successful transport of a droplet across adjacent EP electrodes, the piezoelectric elements need to provide sufficient charges and electric bias to generate appropriate electrophoretic forces. First, finite element models were developed to predict the induced droplet charges and the resulting electrophoretic force for a range of droplet sizes, electrode pitches, and actuation voltages anticipated in typical microfluidic applications.

Figure 3.3 illustrates our simulation domain and boundary conditions. Two cylindrical electrodes of radius r_c and pitch p and a spherical droplet of radius r were immersed in a dielectric fluid (Fluid 1). A second fluid of a higher density (Fluid 2) was used to separate the droplet from the solid surface at the bottom. The electrodes protrude into Fluid 1 by a finite gap of *h*. The dielectric constants of the two fluids are denoted as ε_1 and ε_2, respectively. A DC voltage of magnitude V_p is applied to Electrode 2 while Electrode 1 is grounded.

The droplet is initially in contact with the upper surface of Electrode 1. The droplet quickly reaches an equipotential state with the electrode, with electric charges of the same polarity distributed over the droplet surface. It is assumed that the droplet takes the maximum equilibrium charges before it leaves the charging electrode. A further discussion of this assumption is provided in Appendix 3-A. The charged droplet then detaches from the electrode under repulsive electrophoretic force acting on the acquired charges. The parameter d_x is defined as the location of the droplet center from its charging electrode (Electrode 1) along the *x*-axis.

The electric field $E = -\nabla V$ is obtained by solving the Laplace equation in both the upper and lower dielectric fluids:

$$\nabla^2 V = 0 \qquad (3.1)$$

The free charge density is set to be 0 in the dielectric fluids.

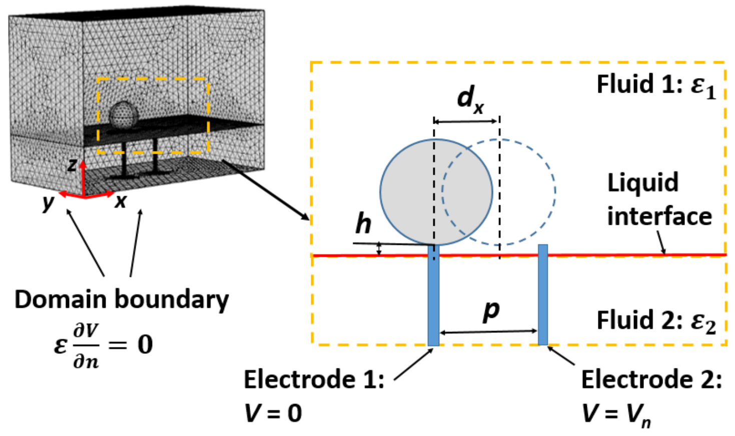


Figure 3.3: Finite element model used to predict the electric charges acquired by a droplet suspended between two biased electrodes and the resulting electrophoretic force.

The droplet is initially in contact with Electrode 1, and the following boundary conditions are specified.

$$V = 0 \qquad \text{on Electrode 1} \qquad (3.2)$$

$$V = V_n \qquad \text{on Electrode 2} \qquad (3.3)$$

$$V_\mathrm{d} = 0 \qquad \text{on droplet surface} \qquad (3.4)$$

At the outer boundaries, the zero charge or symmetry boundary condition is specified.

$$\varepsilon\frac{\partial V}{\partial n} = 0 \qquad (3.5)$$

Once the electric field E is obtained, the total charge Q_{eq} of the droplet is calculated by integrating the electric displacement over the droplet surface S_d:

$$Q_{eq} = \varepsilon \int \vec{E}\, dS \qquad (3.6)$$

The lateral electrophoretic force F_e along the x-axis is calculated by integrating the Maxwell stress tensor over S_d:

$$F_e = \frac{\varepsilon}{2} \int E_n^2 cos\theta\, dS \qquad (3.7)$$

Here E_n is the electric field normal to the droplet surface and θ is the angle between the surface normal vector and the x-axis.

Next, the electrophoretic force acting on the droplet at different positions between the two adjacent electrodes is determined, assuming that the total droplet surface charge is equal to that obtained in Eq. (3.6): $Q = Q_{eq}$. As the droplet moves, the electric field distribution is modified (E'). The Laplace equation for each droplet location is solved to determine E' and then the electrophoretic force is calculated using Eq. (3.7) under this new electric field distribution.

3.3.2 Results and Discussion

3.3.2.1 Droplet charges

Figure 3.4 shows the predicted equilibrium charges Q_{eq} for various combinations of droplet sizes and electrode pitches under an actuation voltage of 200 V. The droplet radius r was varied from 0.2 to 1 mm and the electrode pitch p from 2 to 8 mm. The actuation voltage of 200 V is chosen to be comparable to outputs from commercial piezoelectric elements used in this study.

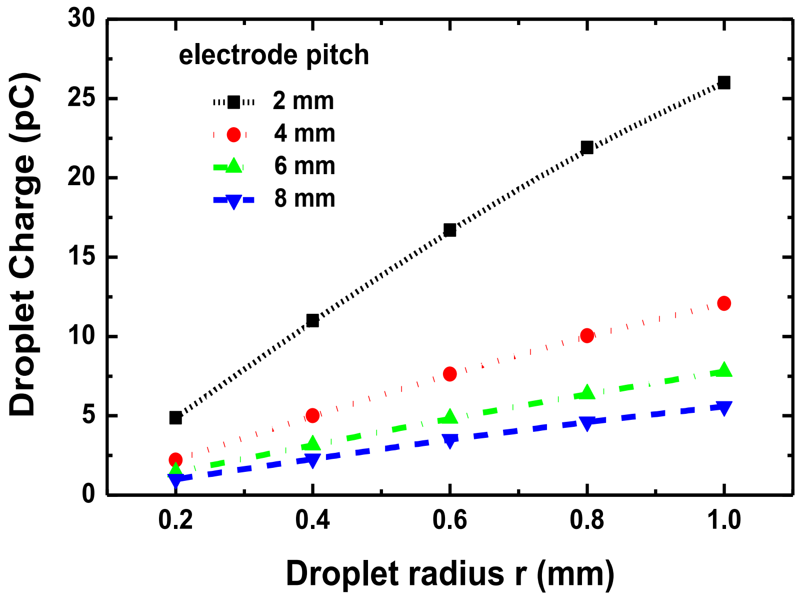


Figure 3.4: Predicted droplet equilibrium charge Q_{eq} as a function of the droplet size under an electrode bias voltage V_p of 200 V. The results are shown for different electrode pitches, varying from 2 to 8 mm.

The predicted charges are in the range of a few to a few tens of picocoulombs. These translate into equivalent capacitances of approximately 10^{-2} to 10^{-1} pF (given an applied voltage of 200 V) for our droplet/electrode system. To provide sufficient charges to the droplet while maintaining the electric bias field during droplet transportation, the capacitance of the piezoelectric elements must be much larger than this value. This represents one criterion, in terms of the minimum capacitance, in designing piezoelectric elements.

The capacitance of our piezoelectric elements currently used is approximately 1.3 nF, more than 4 orders of magnitude greater than the equivalent capacitances of the droplet/electrodes. The amount of charge generated by our piezoelectric elements at an output voltage of 200 V (approximately 2.6 x 10^{-7} C) is likewise more than 4 orders larger than the amount of charges acquired by the droplet. As a result, the flow of charges from the piezoelectric elements to the droplet would have minimal effect on the electrode bias voltages. The actuation voltages may therefore be approximated as a constant equal to the open circuit output voltage of the piezoelectric elements.

For reference, Table 3.1 lists the estimated capacitances per unit area for two common types of piezoelectric elements with two different thicknesses [100], [101].

Table 3.1: Capacitances of typical piezoelectric elements per unit area.

Material	Relative permittivity	Film thickness (μm)	Capacitance per unit area (pF /mm^2)
PVDF	12.4	25	4.4
PVDF	12.4	100	1.1
PZT-5A	1600	25	531
PZT-5A	1600	100	132

Figure 3.4 also reveals that, for a given droplet size, the total amount of charge decreases with increasing electrode pitches or decreasing nominal electric fields E_{nom} under a constant bias voltage. The nominal electric field is defined as $E_{nom} = V_p/p$. This is expected since the equilibrium droplet charges depend on the electric displacement on the droplet surface, which is in turn proportional to the electric field strength. It is also noted that the droplet surface charge density decreases rapidly with increase in droplet sizes, resulting in a nearly linear increase in the total droplet charges with the droplet radius. This is in part because droplet charging is governed primarily by non-uniform electric fields in the immediate vicinity of the electrode tip, whose magnitudes decrease rapidly with distance from the electrode tip.

To illustrate these points further, consider the case of a droplet of radius r suspended between two large parallel plate electrodes. The amount of equilibrium charges under this configuration $Q_{parallel}$ is given by:

$$Q_{\text{parallel}} = \frac{\pi}{6}(4\pi r^2)\varepsilon E. \qquad (3.8)$$

The predicted droplet charges are plotted for various combinations of droplet sizes and electrode pitches in Figure 3.5, for E_{nom} = 0.01 MV/m, 0.1 MV/m, and 1 MV/m. The normalized charges are approximately inversely proportional to the normalized droplet radius for the electrode and the geometric parameters of the droplet considered in the present study. This is consistent with the nearly linear relation between the amount of droplet charges and the droplet radius observed in Figure 3.4.

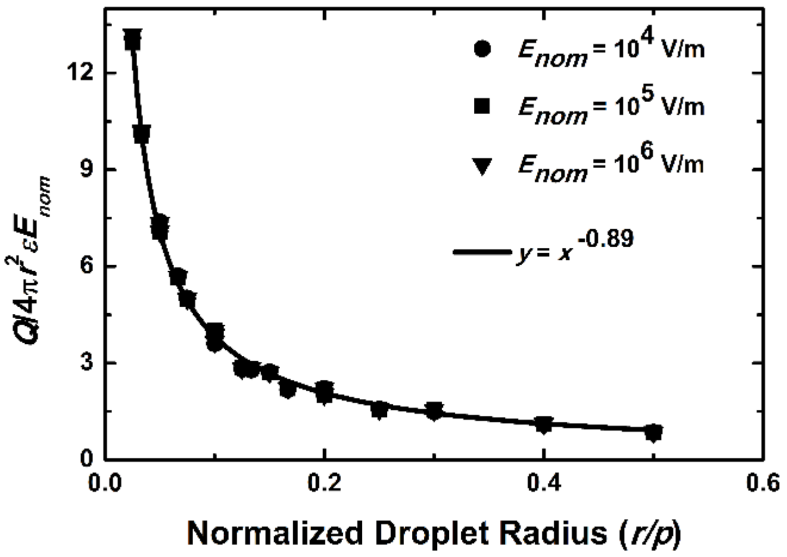


Figure 3.5: Normalized equilibrium droplet charge is approximately inversely proportional to the normalized droplet radius *r* under combinations of geometric parameters examined in the present study.

3.3.2.2 Charge dissipation time

For the successful actuation of a droplet through electrophoretic force, the charge acquired and carried by the droplet and the electric field established by our piezoelectric elements between the neighboring EPD electrodes must be maintained over the actuation process. The relaxation of electric charges carried by a droplet is governed by the charge dissipation through the surrounding dielectric liquid:

$$Q(t) = Q_0 e^{-\frac{t}{\tau_r}} \tag{3.9}$$

$$\tau_r = \varepsilon/\sigma \tag{3.10}$$

Here τ_r is the relaxation time constant, and is equal to the ratio between the permittivity and conductivity of the material. The estimated value of τ_r for silicone oil is >200 s.

To estimate the discharge time across the piezoelectric element due to finite leakage, the element is modeled as a capacitor (C ~ 0.8 nF) connected in parallel with a resistor of resistance *R*. The voltage decay can then be described as:

$$V(t) = V_0 e^{-t/RC} \tag{3.11}$$

The resistance *R* was measured using a source meter.

Figure 3.6 shows the measured I-V curves under sourcing voltages ranging from 0 to 120 V. The resistance *R* obtained from the linear fit is ~4×10^{10} Ω. The theoretical discharge time constant τ_p is then ~30 s, which is smaller than τ_r, and is thus is expected to dominate.

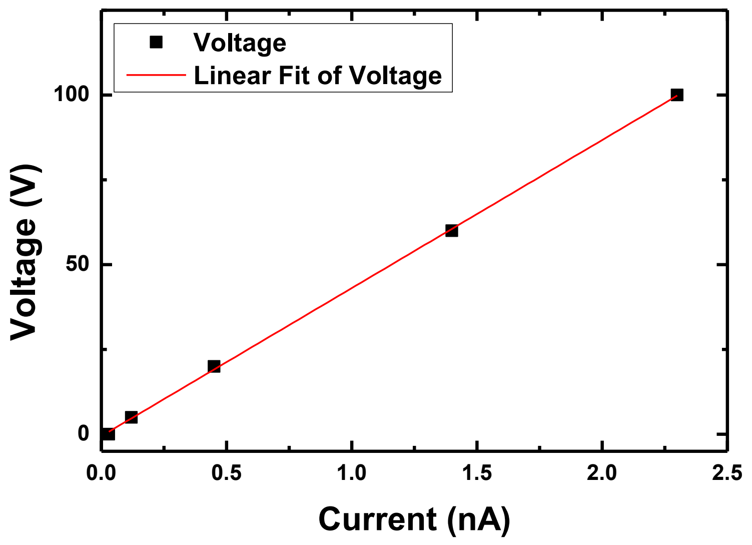


Figure 3.6: I-V curve for piezoelectric element resistance measurement.

This estimation was further confirmed by directly measuring the voltage decay curves. The piezoelectric element was deflected to a pre-selected bending angle and the resulting open circuit voltage output was measured as a function of time using an electrometer of input impedance $>$ 200 TΩ. The measurements were then repeated at different bending angles, and hence output voltages.

Figure 3.7 shows the temporal variations in the output voltage from a piezoelectric element for different bending angles in the log scale. The average measured time constant τ_p was 27.5 s with a standard deviation of 5.8 s, which is consistent with the value estimated from the resistance and capacitance of the piezoelectric elements.

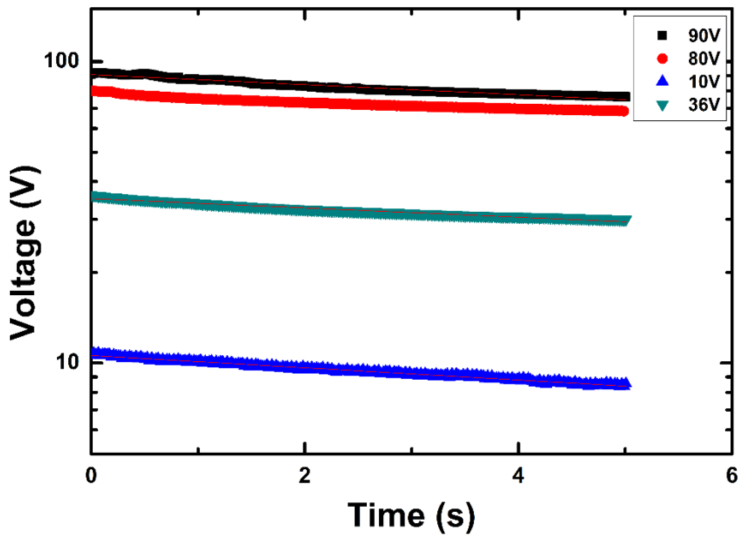


Figure 3.7: Decaying voltage output from a single piezoelectric element with increasing time.

3.3.2.3 Droplet velocity and electrophoretic force

To establish the baselines, experiments were conducted, in which the droplets were actuated using an external power supply. The droplet translational motions were recorded using a digital camera at 30 fps. The instantaneous droplet velocities at different positions between the electrodes were calculated through image analyses using ImageJ®. Each calculated velocity represents the average value over five independent trials (N = 5) performed under nominally identical bias voltage and geometric parameters. The estimated errors *e*, indicated by the error bars in this and subsequent figures, account for both the random error (S_N), as estimated from the standard deviations at the 95% confidence level, and the uncertainty in the measured droplet positions due to the finite spatial resolution of our imaging system.

$$e = \sqrt{S_N^2 + u^2} \tag{3.12}$$

The measured droplet velocities were compared with the so-called droplet terminal velocities, which were obtained by equating the predicted electrophoretic force at each droplet location to the steady-state drag force. The steady-state drag force F_x of a droplet moving parallel to a horizontal surface at a constant velocity U can be determined from [102], [103]:

$$F_x = 6\pi\mu r U f \tag{3.13}$$

$$f = \left(\frac{8}{15} + \frac{64}{375}\,\epsilon\right)\log\left(\frac{2}{\epsilon}\right) + 0.5846 \tag{3.14}$$

where μ is the viscosity of the surrounding dielectric fluid after correcting for the finite viscosity of the liquid droplet, r is the radius of the droplet, and ϵ is the ratio of the gap h (shown in Figure 3.3) to the droplet radius r.

In Figure 3.8, the filled symbols represent the measured instantaneous velocities of the droplets under three actuation voltages: 150 V, 200 V, and 250 V. These voltage values were chosen to be comparable to the outputs from our piezoelectric elements. The droplet radius was 0.63 mm and the electrode pitch was 1.76 mm. The lines correspond to the predicted droplet terminal velocities at different positions between the electrodes.

The predicted terminal velocities agree reasonably well with the experimentally measured velocities in the middle sections ($0.3 < d_x/p < 0.7$) for actuation voltages of 150 V and 200 V.

They deviate from the experimental data near the starting and terminal electrodes. This is mainly due to the fact that our steady-state model ignores the finite inertia of the droplets and the dynamic variations in droplet charges due to finite leakage. The model overpredicts the velocities at the highest actuation voltage (250 V) due to the incomplete initial droplet charging, as further discussed later in the section. The droplet translational velocity decreases quite substantially with decreasing bias voltages due to the combined effect of smaller droplet charges and smaller electric fields. For all the cases shown in Figure 3.8, the droplet velocity increases with d_x /p. That is, a larger electrophoretic force acts on the droplet as it approaches the terminal electrode of the opposite polarity, than it does as it departs from the charging electrode of the same polarity.

Aqueous droplets are also successfully actuated when the electrodes are biased using the piezoelectric elements. The measured droplet translational velocities, marked as the crosses in Figure 3.8, are similar to those obtained at 200 V using an external power supply.

The total droplet transit time across the two electrodes was <1 s for all the cases studied here. The voltage applied by the piezoelectric elements may be assumed to remain constant only when this total droplet transit time is much less than the discharging time of the piezoelectric elements. This consideration leads to a second criterion for reliable transport of droplets using our piezoelectric actuation scheme: the droplet transit time (between the electrode pairs) must be sufficiently short when compared with both the discharging time of the piezoelectric elements and the charge relaxation time of the dielectric medium. In our piezoelectric elements, the RC time constant is estimated to be 30 s from their measured electrical capacitance and resistance.

The estimated value of τ_r for the silicone oil used in the present study is >200 s, which is much larger than the RC time constant of the piezoelectric elements (see Section 3.3.2.2).

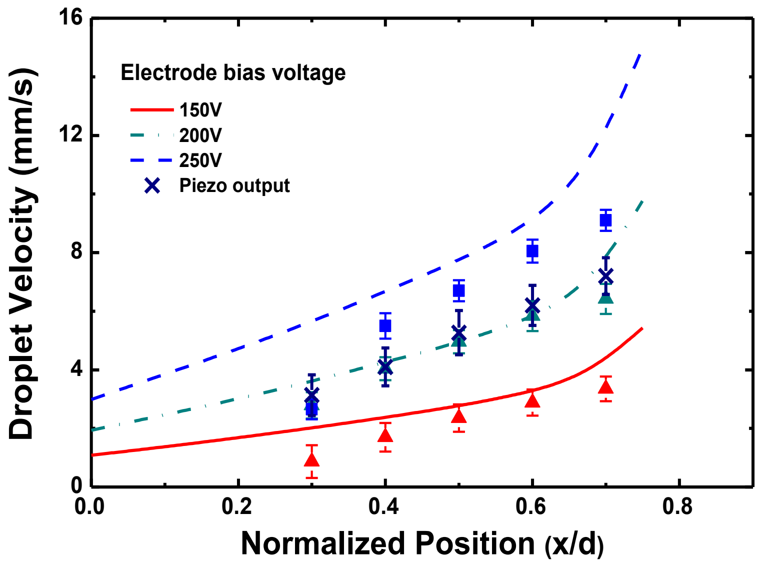


Figure 3.8: Measured (symbols) and predicted (lines) droplet velocities as a function of the normalized distance from the charging electrode under different applied voltages. The electrode pitch *p* = 1.76 mm and the droplet radius *r* = 0.63 mm.

Next, the experiments were repeated for different values of the electrode pitch. In

Figure 3.9, the solid lines show the predicted terminal velocities of droplets having a radius of 0.7 mm at an electrode bias voltage of 200 V. The electrode pitch *p* is varied from 2.2 to 5.9 mm. The predicted droplet terminal velocity decreases rapidly with the increasing pitches. The droplets were also successfully actuated with the piezoelectric elements for the electrode pitches of 2.2 and 3.34 mm. The measured translational velocities are plotted as filled symbols in

Figure 3.9. For the largest pitch (5.9 mm), the droplet transit time approaches the discharge time of the piezoelectric elements and a stable actuation voltage cannot be maintained. To help further discuss our results, an external power supply was used to obtain the droplet velocities for the case with the largest electrode pitch.

Figure 3.9 also shows that the spatial variations in the local droplet velocity along d_x qualitatively differ for different electrode pitches. For a large electrode pitch of 5.9 mm, the local droplet velocity stays relatively constant for $0.3 < d_x/p < 0.7$. In contrast, for the smaller pitches, the local droplet velocity increases monolithically with d_x/p, as the droplet continues to spatially "sample" highly non-uniform electric fields near the electrodes.

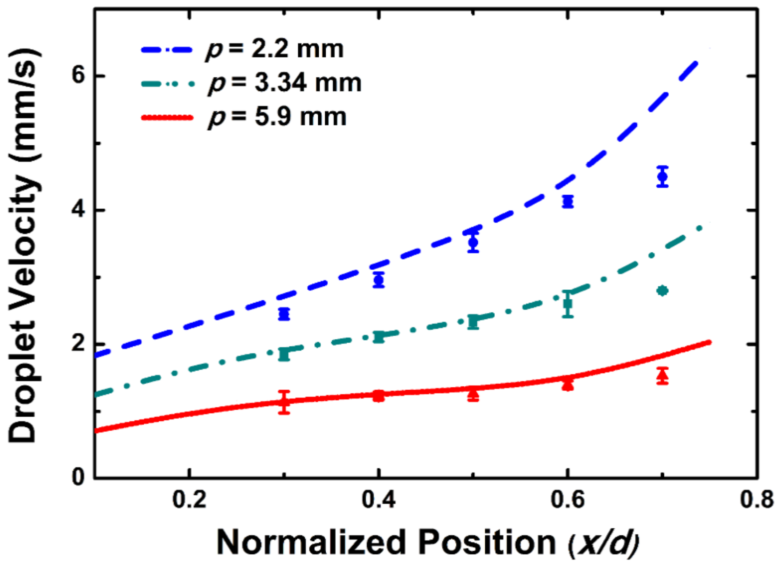


Figure 3.9: Variations in the local droplet translational velocity for different values of the electrode pitch. The actuation voltage is fixed at 200 V and the droplet radius is fixed at 0.7 mm.

Note that the above results were obtained by varying the electrode pitch for a fixed droplet radius. Alternatively, we may consider cases where the droplet radius is varied at the same time such that the ratio of the droplet radius to the electrode pitch *r/p* stays constant. In this case, the spatial variations in the local droplet velocity along *x* remain qualitatively similar for different electrode pitches considered in the present study. Representative simulation results for *r/p* = 0.1 are presented in Appendix 3-B.

Next, the droplet velocities at the mid-point between the two electrodes was determined as a function of the droplet radius, while keeping *r/p* constant (approximately 0.33). The experiments were repeated for three different bias voltages: 200 V, 250 V, and 300 V. The filled symbols in Figure 3.10 are the measured values, which agree reasonably well with the predicted terminal velocities (lines). With *r/p* being kept constant, the droplet velocity decreases with increase in droplet sizes, as the nominal electric field and charge density at the droplet surface decrease. For the largest droplet of radius 1.1 mm, the piezoelectric actuation is insufficient due in part to a large transit time and in part to a large droplet inertia.

At high actuation voltages (250 V and 300 V in Figure 3.10), it should be noted that the measured velocities are lower than the predicted values for smaller droplet sizes (and electrode pitches). Similar overprediction is also noted in Figure 3.8 at an actuation voltage of 250 V. One possible origin of these discrepancies is a partial or incomplete charging of droplets under high electric fields present in these situations. That is, the actual amount of droplet charges is less than Q_{eq}, which in turn leads to reduced electrophoretic force.

For a droplet with finite conductivity, a finite charging time is necessary for the droplet to reach equipotential with the charging electrode. If the droplet leaves the electrodes within this charging period, then the droplet will acquire only a fraction of Q_{eq}. The charging time is a function of the conductivity of the droplet, the dielectric properties of the fluids, and the contact area between the droplet and the charging electrode. Transient numerical simulations were used to estimate the charging time, and were found to be on the order of a few to a few tens of milliseconds for aqueous droplets of the sizes and electrical properties used in the present study (see Appendix 3-A). This is comparable to or larger than previously estimated contact times at local electric fields of approximately 3 kV/cm [104], [105]. Therefore, it is expected that the droplet is only partially charged before it is detached from the charging electrode under the high bias voltages. This in turn leads to decreased electrophoretic forces and hence smaller droplet translation velocities.

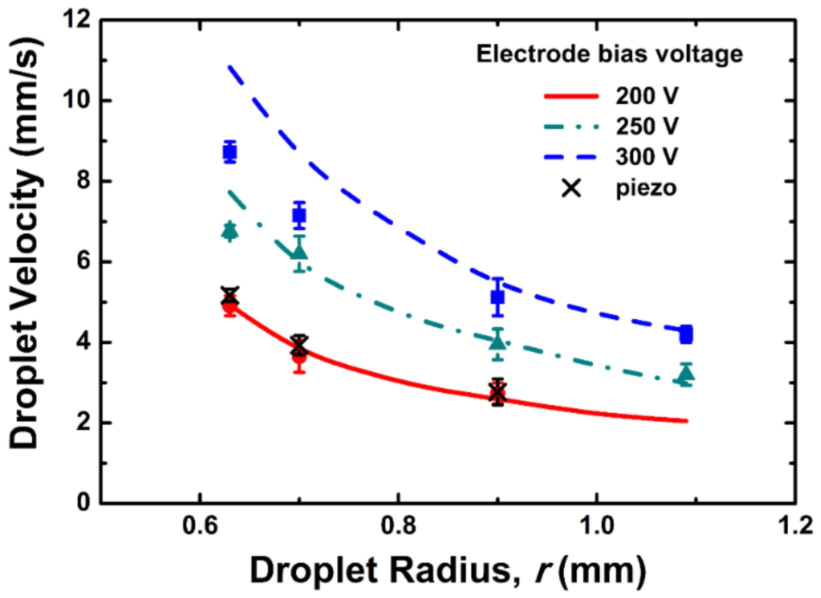


Figure 3.10: Variations in the droplet velocity at the middle point between the electrodes (*x/p* = 0.5) for four different droplet radii and three different actuation voltages. The *r/p* ratio is kept constant at 0.33 for all the cases.

3.4 Droplet Transport and Merging

By satisfying the design criterion for successful EPD actuation, the droplet transport and merging is demonstrated using finger-powered EPD DMF. Figure 3.11 shows the time-sequence optical images of a single water droplet having a volume of approximately 2 μL being transported between adjacent electrodes through a sequence of finger-powered ECD actuations. The electrode pitch *p* is ~ 2.08 mm and the droplet radius *r* is ~ 0.78 mm. To transport a droplet in a desired direction, one needs to sequentially alternate the relative polarities of the electrode pairs. For example, referring to Figure 3.11 (c) to (d), the polarities of Electrode 1 and 3 are interchanged when the droplet reaches the nearest approaching electrode (Electrode 2). This allows the positively charged droplet on Electrode 2 to continue moving to Electrode 3.

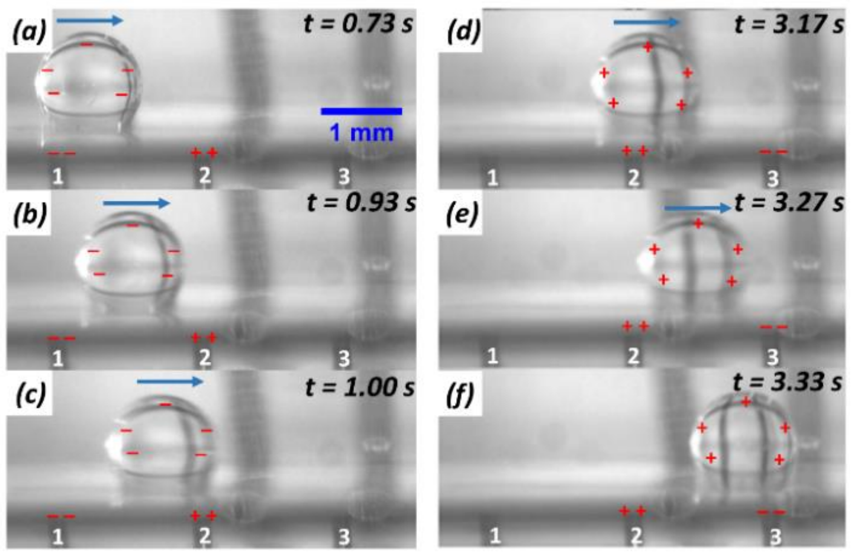
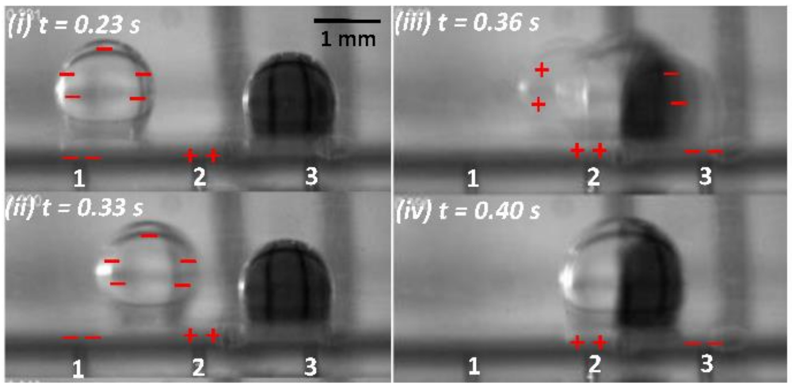


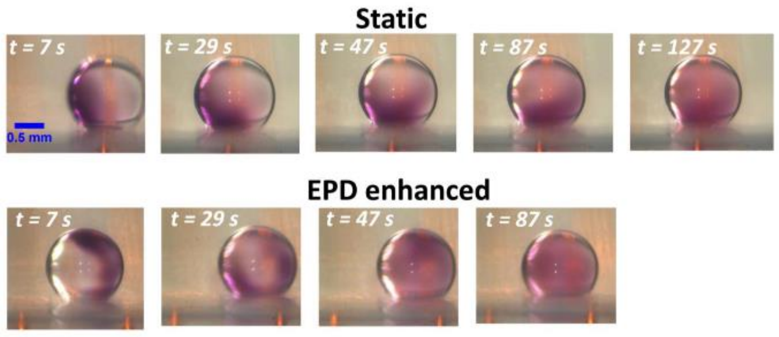
Figure 3.11: Time sequence showing continuous droplet transport by finger-powered EPD. The droplet volume is approximately 2 μL and the electrode pitch is 2.08 mm. The *r/p* ratio is ~ 0.37.

Figure 3.12(A) shows the merging of two DI water droplets using the same electrode configuration as above. The two droplets were oppositely charged in advance. The transparent droplet on the left was positively charged on Electrode 2, whereas the dyed droplet on the right was negatively charged on Electrode 3. When actuated to approach each other, the two droplets merge almost instantaneously upon contact through electrostatic interaction.

Enhanced internal mixing can be achieved by continuously transporting the droplet back and forth between the two electrodes and thereby inducing internal flows. This can be achieved readily in EPD microfluidics by simply maintaining the electrode bias, that is, by keeping the piezoelectric elements bent. The oscillatory motion is maintained, because the polarity of the droplet keeps reversing as the droplet alternatingly contacts one of the two electrodes. Enhanced mixing can be observed by mixing a dyed droplet with a clear droplet (Figure 3.12 (B)) with or without the sustained electrode bias (EPD enhanced *vs* static). The mixing time is reduced by approximately 30%. Further reduction in the mixing time may be achieved by breaking the symmetry and stirring more chaotic flows inside the droplet using a 2D array of electrodes rather than a linear array [106].



(A)



(B)

Figure 3.12 (A): Time sequence showing the merging of two oppositely charged droplets; (B): Mixing by pure diffusion (upper) compared with enhanced mixing using EPD actuation (lower). The radius of the merged droplet is ~ 0.75 mm.

One important challenge in the biomedical application of microfluidics originates from high viscosities, extreme pH values, or other unusual properties of the samples such as body fluids. Figure 3.13 shows the successful transport of droplets of human body fluids (saliva and urine) and an alkaline solution across three electrodes. The pH values for these droplets varies from 5.8 (human urine) to 9 (sodium hydroxide solution).

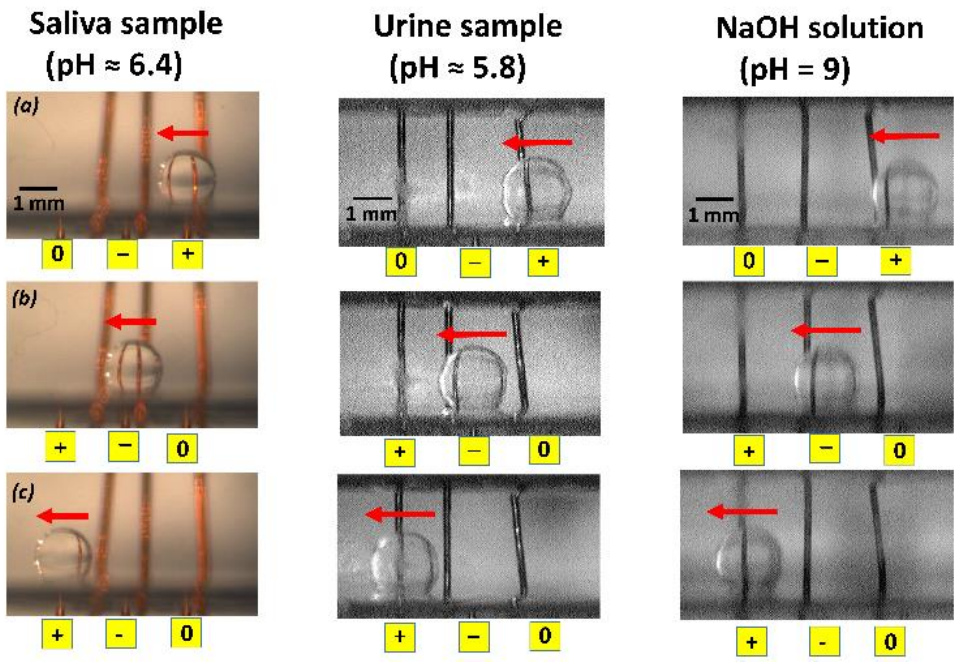


Figure 3.13: Sequential images of a saliva droplet, a urine droplet, and a NaOH droplet transported via finger-powered EPD. The droplet radius is ~0.7 mm and the droplet-radius to electrode-pitch ratio (*r/p*) is approximately 0.4.

3.5 Towards Practical System Implementation

3.5.1 Mechanical System for Programmed Operations

Relying on just human fingers to deflect multiple piezoelectric elements precisely in a complex sequence is not a realistic method for the practical implementation of our concept. To convert finger (or hand) motions into a sequence of controlled and reproducible deflections of the piezoelectric elements, a finger/hand-rotated drum system is proposed. Its design, shown in Figure 3.14, consists of a drum and an array of mechanical levers, all printed by high-resolution 3D printing. This design is analogous to that of a music box where pins (or embossed protrusions) formed on a cylinder are used to pluck an array of cantilever beams in a specific sequence. On the surface of the drum, there is a set of protrusions at pre-programmed locations. The surface of the drum is patterned like a toothed-gear, so that reusable plastic protrusions can be placed at desired locations. The mechanical levers are mounted in a see-saw configuration using a common shaft on a fixed fulcrum. One end of each lever is linked mechanically to the piezoelectric elements. As the drum is rotated manually, the protrusions push down on the mechanical levers, which in turn deflect the corresponding piezoelectric elements.

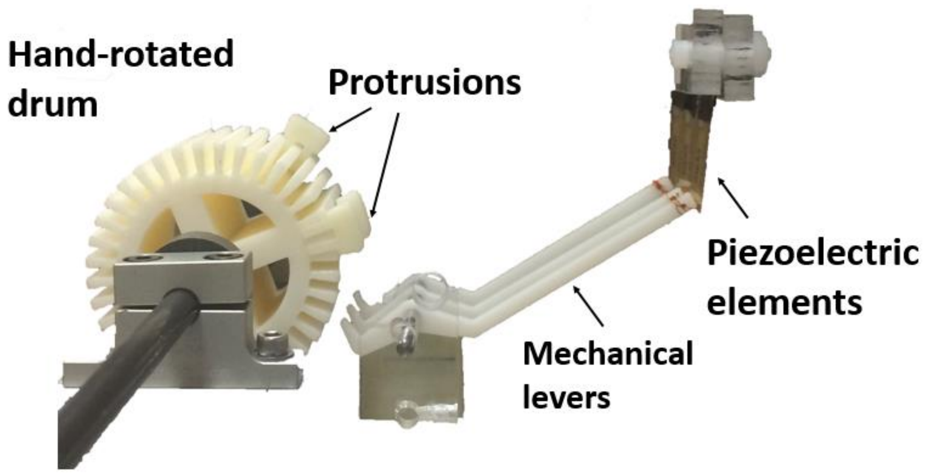
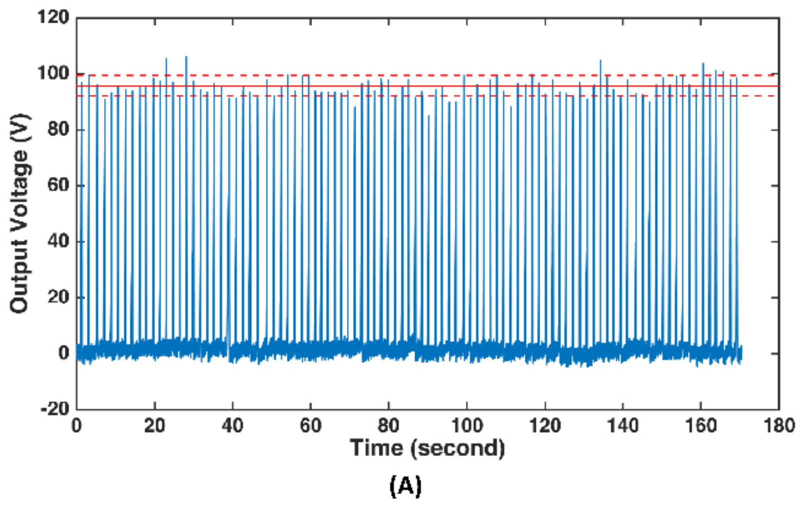


Figure 3.14: Finger/hand-rotated drum system, consisting of a drum with protrusions and an array of mechanical levers mounted in a see-saw configuration. One end of each lever is linked mechanically to a piezoelectric element. In this particular device, the outer diameter of the drum

is approximately 6 cm; the width and height of each protrusion are 3 mm; the length of the levers is approximately 10 cm; and the lever ratio is approximately 1:7.

The consistency of the voltage pulses generated using the mechanical drum system was experimentally examined. Each piezoelectric element unit consists of two piezoelectric elements connected electrically in series and mechanically to the same lever. The drum is rotated at approximately 100 degrees/second, resulting in voltage pulses of duration approximately 0.1 s. This duration is comparable to the typical droplet transit times across two adjacent electrodes. Figure 3.15 shows the voltage pulses measured during repetitive deflections of one of the piezoelectric units.

Table 3.2 summarizes the results obtained from three independent piezoelectric element units. The results show that our drum system provides fairly consistent voltage pulses (standard deviations of approximately 4%) with smaller amplitude variations than human fingers.



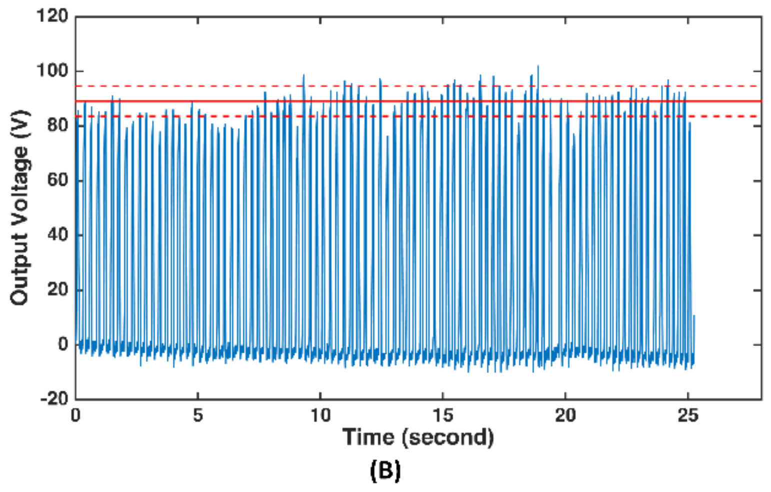


Figure 3.15: Voltage outputs from a single piezoelectric element unit over multiple deflections (A) by the mechanical drum system; (B) by a human finger. The solid red lines indicate the average voltage outputs and the dotted red lines represent one standard deviation.

Table 3.2: Measured voltage pulse outputs from 3 independent piezoelectric element units over 100 deflections either by the drum system or by human fingers.

Unit	Voltage (V) by drum	Voltage (V) by finger
1	90.2 ± 3.4	87.4 ± 7.9
2	97.2 ± 1.6	99.8 ± 7.3
3	93.8 ± 2.6	90.5 ± 6.3

3.5.2 Base Electrode Matrix and Electric Connection Schemes

The number of piezoelectric elements that can be used in practical portable EPD devices is limited. A base electrode matrix and its connection scheme were explored to realize different microfluidic functions using minimum numbers of piezoelectric elements. Similar schemes were explored for EWOD digital microfluidics [73]. The electric polarities of the electrodes, however, were not fixed in those studies, as they implicitly assumed the availability of external power supplies/switching circuits. In contrast, in our finger-powered EPD microfluidics, the polarity of each piezoelectric element unit is pre-fixed to facilitate their mechanical integration.

For a square electrode matrix of size $n \times n$ ($n > 3$), the minimum number of piezoelectric element units necessary to actuate a droplet located at any given position in any of the four independent directions (up, right, down, or left) is 8 (see proof in Appendix 3-C). Figure 3.16(A) shows one design of a base electrode matrix of size 4×4. Even numbers (red) are used to label the electrodes connected to the four piezoelectric elements of positive polarity, while odd numbers (blue) are used to label the electrodes connected to the remaining four piezoelectric elements of negative polarity. Figure 3.16(B) shows sample droplet transport paths that can be achieved using the base electrode matrix. Other possible path designs for operations such as droplet merging and storage are provided in the Supplementary Information in the corresponding paper. This basic matrix can be replicated multiple times (Figure 3.16(C)). A minimum of 8 piezoelectric elements may be used to actuate a droplet across a larger electrode matrix or to perform an identical set of actuations for multiple droplets in parallel (Figure 3.16(D)).

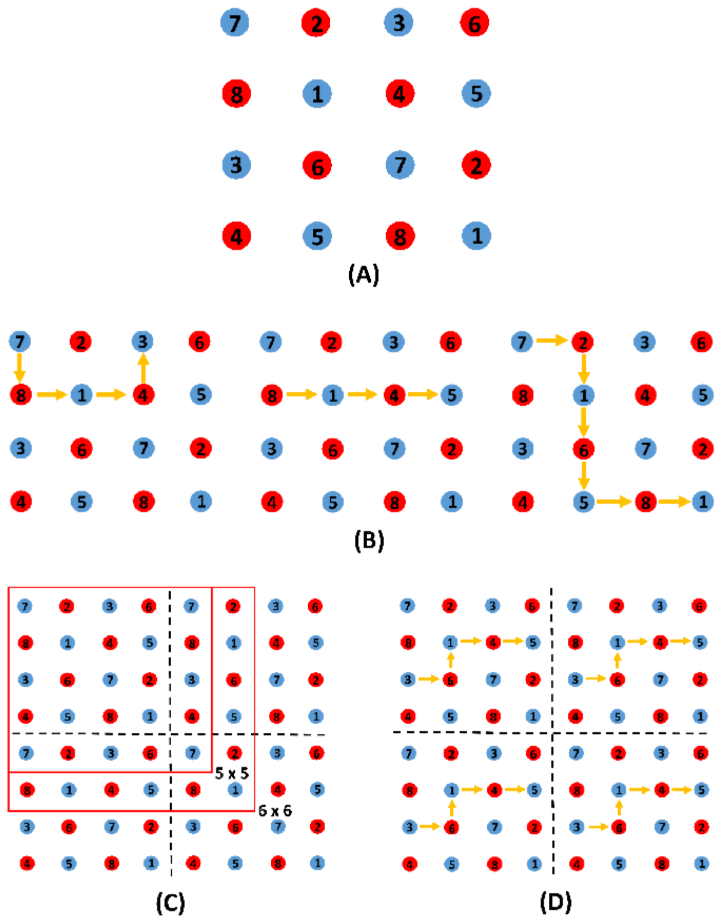


Figure 3.16: (A) Base electrode matrix of size 4×4; (B) Example droplet actuation paths that can be realized using the 4×4 base electrode matrix; (C) By replicating the base electrode matrix, one

can power a larger electrode matrix using 8 piezoelectric element units; (D) Parallel execution of a set of identical operations on 4 droplets.

Electrically connecting multiple electrodes to a single piezoelectric element unit allows significant reduction in the number of piezoelectric elements, and hence the size and cost of the overall system. However, the interference among these electrodes may present a potential issue.

To quantify the interfering influence of adjacent electrodes, additional numerical simulations were conducted. Figure 3.17 shows the predicted interfering forces at various positions between two driving electrodes for different values of *r/p* ratio under a typical electric field strength used in our experiments. The normalized magnitude of the interfering forces is larger for larger droplets (larger radius-to-electrode pitch ratios) and for smaller electric fields. For all the cases that were calculated, the magnitude of the predicted interfering forces was less than 10% of the main driving force for electric fields as small as 0.16 MV/m and *r/p* ratios as large as 0.45.

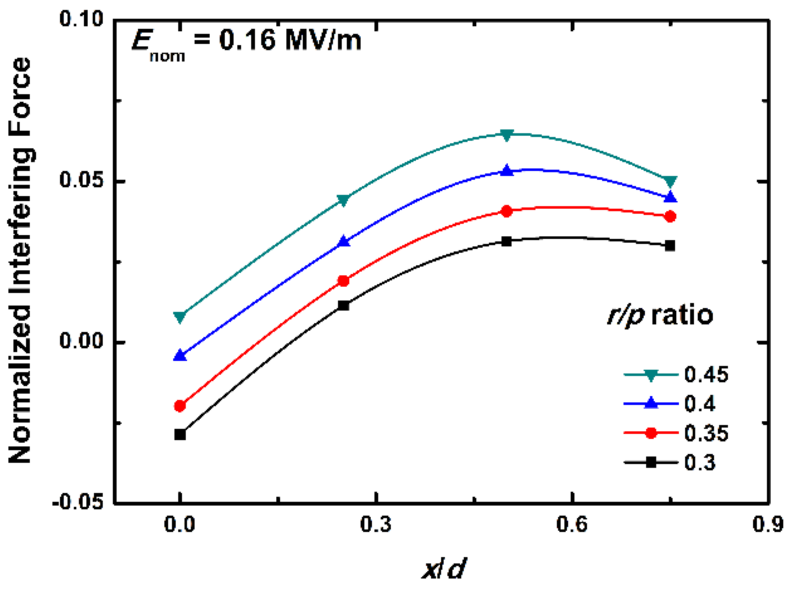


Figure 3.17: Predicted normalized interfering forces at different positions between two driving electrodes under different values of the ratio between the droplet radius and the electrode pitch, *r/p*, showing that the magnitude of interfering forces is less than 10% of the main driving force under typical actuation conditions used in the present study.

3.5.3 Pre-programmed Functional Actuation of Droplets

Finally, our 4×4 base electrode matrix, which is connected to 8 independent piezoelectric element units, was experimentally tested. The units were deflected in pre-programmed sequences using our mechanical drum system. Sequential droplet actuations were successfully demonstrated over multiple paths covering different electrode sites on the matrix. Figure 3.18(A) shows the snap shot images for the linear transport of the droplet for two typical paths, and

Figure 3.18(B) shows the snap shot images for the merging of two droplets and the subsequent enhanced mixing. The corresponding videos are also provided in the Electronic Supplementary Material.

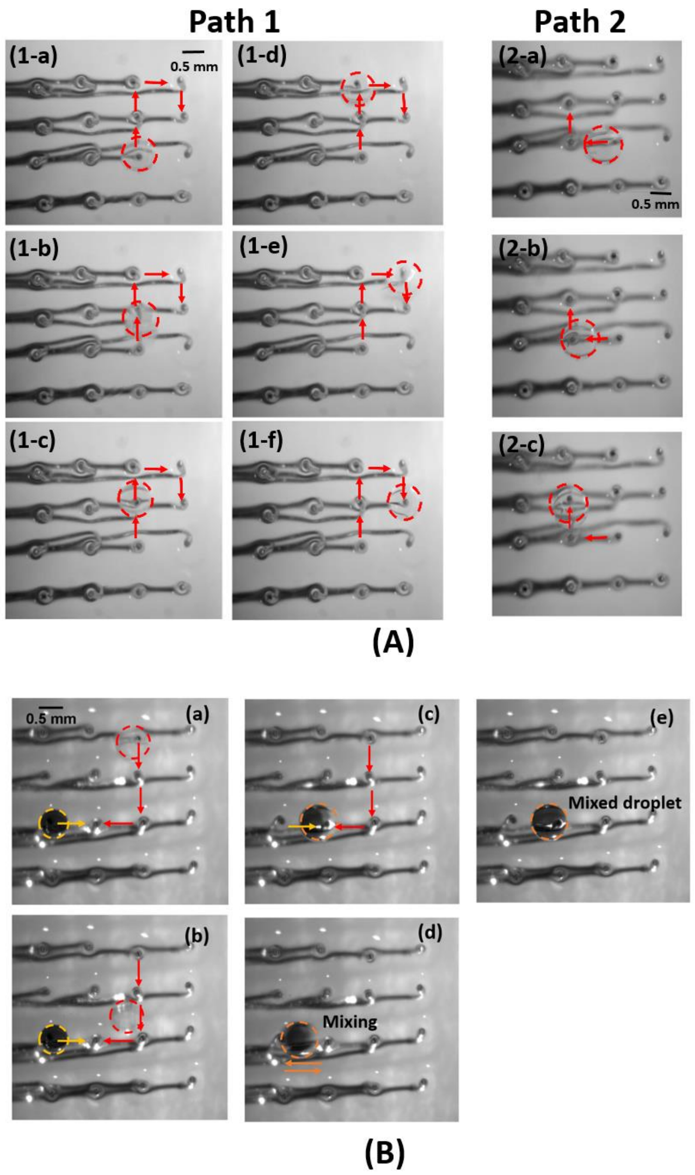


Figure 3.18: (A) Demonstration of droplet actuation along different paths on the base 4×4 electrode matrix, which is connected to 8 piezoelectric element units; (B): Demonstration of the merging and subsequent enhanced mixing of two droplets on the base electrode matrix.

3.6 Summary

In this chapter, the finger-powered electrophoretic transport of droplets (EPD) for digital microfluidics was demonstrated. The mechanical energy provided by human fingers can be converted using an array of piezoelectric elements into sufficient electrical energy to charge and electrophoretically actuate the droplets.

Numerical models for the electrical charging of the droplet and the resulting electrophoretic forces were developed and experimentally validated to establish the design criteria for finger-powered EPD actuation. The capacitance of the piezoelectric elements needs to be much larger than the droplet/electrode system, and the droplet transit time (across two electrodes) needs to be sufficiently smaller than the discharge time of the piezoelectric elements. The latter in turn limits the electrode pitch. The linear transport and merging of aqueous droplets were successfully demonstrated directly using finger actuation. The transport of human body fluids, such as saliva and urine droplets, was also demonstrated.

To facilitate the practical implementation of portable microfluidic devices based on our approach, a finger/hand-rotated drum system was designed to reliably control the deflections of multiple piezoelectric elements in a pre-programmed manner. A pin assignment scheme to implement different microfluidic functions while using a minimum number of piezoelectric elements was reported. Multiple pre-programmed droplet actuations were demonstrated using our integrated system, which consists of a 4×4 base electrode matrix and a mechanical drum with protrusions. This work establishes the engineering foundation for the systematic design and implementation of finger-powered EPD devices for portable microfluidic applications.

Appendix 3-A

In the main text in this chapter, it was assumed that the droplet acquires maximum equilibrium charges before it is detached from the charging electrode. To examine the validity of this assumption, the transient charging process for an aqueous droplet is directly simulated.

The model is shown schematically in Figure 3.19. A spherical droplet of radius *R*, conductivity σ, and permittivity ε_i is in contact with one of the electrodes through a contact area *A*. The droplet and electrodes are immersed in a dielectric oil. Subscripts "*i*," "*o*," and "*s*" are used to label the variables associated with the region inside the droplet, in the surrounding oil, and at the droplet interface, respectively. The electrical charges are assumed to be transported by Ohmic volume conduction within the droplet. It is also assumed that all the dielectric properties are constant.

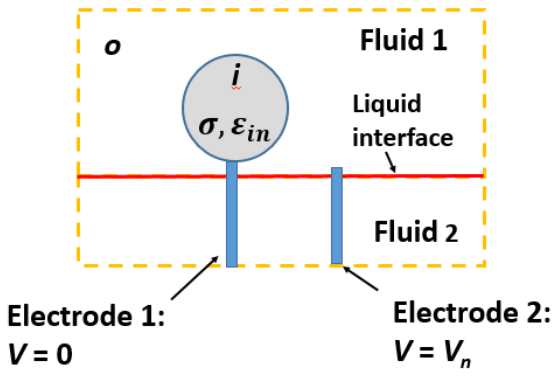


Figure 3.19: Schematic illustration of the model used for the simulation of transient droplet charging.

The transient continuity equation is solved:

$$\frac{\partial {\rho_s}^{(t+1)}}{\partial t} + \nabla \cdot \vec{J}^{(t)} = 0 \qquad (3.15)$$

The current density is related to the electric field within the droplet:

$$\vec{J}^{(t)} = \sigma \vec{E}^{(t)} = -\sigma \nabla V^{(t)} \qquad (3.16)$$

At each time *t*, the electric field $E^{(t)} = -\nabla V^{(t)}$ is obtained by solving the Laplace equation:

$$\nabla^2 V^{(t)} = 0 \qquad (3.17)$$

The boundary conditions are:

- Constant electric potentials at the two electrode surfaces:

$$V = 0 \qquad \text{on Electrode 1} \qquad (3.18)$$

$$V = V_n \qquad \text{on Electrode 2} \qquad (3.19)$$

- Continuity of the electrostatic displacement vector across the droplet surface, where ρ_s is the surface charge density:

$$\varepsilon_o \frac{\partial V_o{}^{(t)}}{\partial n} - \varepsilon_i \frac{\partial V_i{}^{(t)}}{\partial n} = \rho_s{}^{(t)} \tag{3.20}$$

- Symmetry boundary conditions on the outer boundaries:

$$\varepsilon_o \frac{\partial V}{\partial n} = 0 \tag{3.21}$$

The total charge at any given time *t* is obtained by integrating the surface charge density on the droplet surface S_d. The predicted temporal evolution of the amount of charges on the droplet is shown in Figure 3.20. The droplet radius is 0.63 mm, electrode pitch is 1.76 mm, and applied voltage is 100 V. The results are presented for three different contact areas *A*: 2.5×10^{-2} mm^2, 5×10^{-3} mm^2, and 2.5×10^{-3} mm^2. The largest value is equal to the area of the top surface of the electrode in our study, and the others to 1/5 and 1/10 of that value. The droplet charges approach the maximum (in magnitude) equilibrium value at different rates, depending on the contact area. The estimated charging times range from a few milliseconds to tens of milliseconds, increasing with a decrease in the contact area.

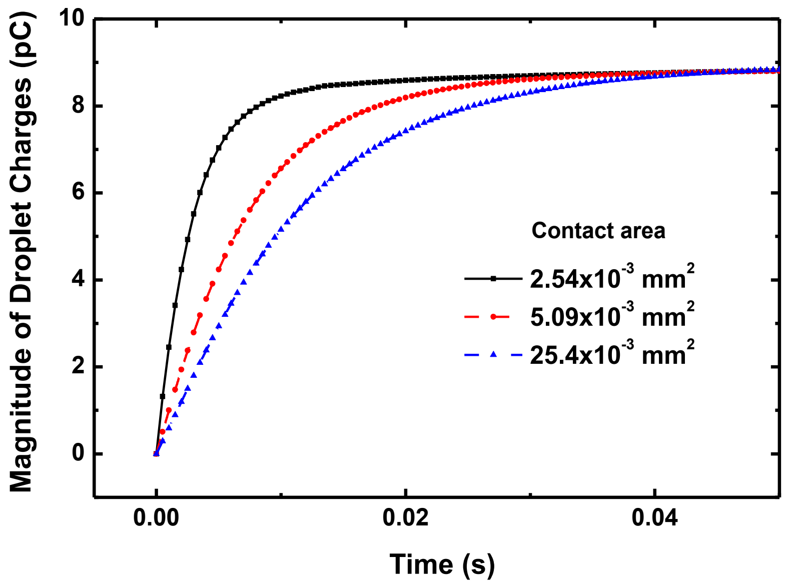


Figure 3.20: Predicted temporal evolution of the droplet charges for three different droplet-electrode contact areas.

Appendix 3-B

Figure 3.21 shows the predicted terminal velocities (in log scale) for a fixed *r/p* ratio of 0.1, under a bias voltage of 200 V. The droplet radius is varied from 0.2 to 0.8 mm and the electrode pitch from 2 to 8 mm. Note that the spatial variations in the local droplet velocity with increasing traveling distance (*dx/p*) are qualitatively similar for all the cases that were simulated.

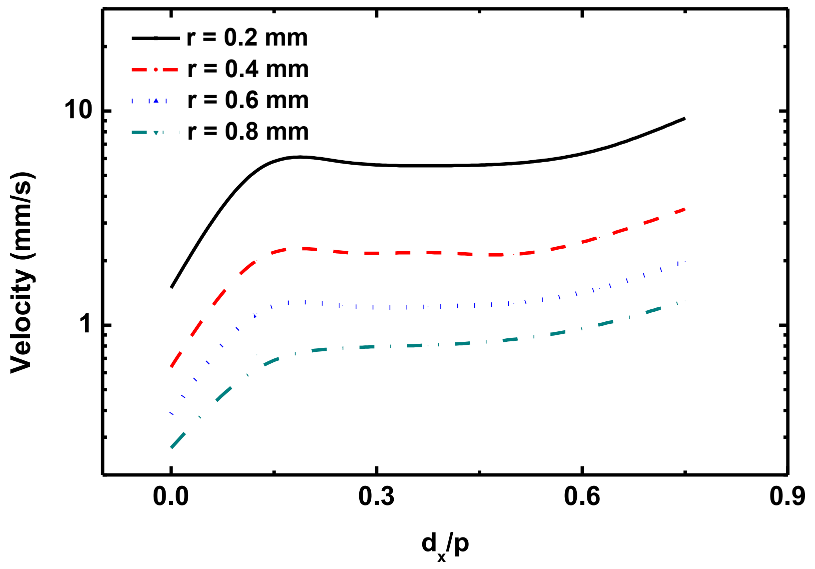


Figure 3.21: Predicted terminal velocities of electrophoretically actuated droplets with a fixed value of *r/p* = 0.1 for different combinations *r* and *p*.

Appendix 3-C

For a square electrode matrix of size $n \times n$ (with $n > 3$), we first show that the minimum number of piezoelectric elements necessary to actuate a droplet at any given position in any of the four independent directions (up, right, down, or left) is 8. (i, j) is used to indicate the row and column of each electrode, c(i, j) $\subset \{+, -\}$ to indicate its polarity, and p(i, j) to indicate the piezoelectric element connected to that electrode.

1. Consider electrode (i, j), where $i \neq 0$, $j \neq 0$, $i \neq n$, $j \neq n$. This center electrode is surrounded by four adjacent electrodes: (i-1, j), (i, j+1), (i+1, j), and (i, j-1), corresponding to each of the

four directions of possible linear droplet actuation. Here, Ω is used to represent the set of piezoelectric elements connected to these 4 surrounding electrodes:

$$\Omega = \{p(i-1,j), p(i,j+1), p(i+1,j), p(i,j-1)\}.$$

2. For the EPD actuation of a droplet initially located at (i, j), the destination electrode must have the opposite polarity. That is, we must have $c(i-1, j) = c(i, j+1) = c(i+1, j) = c(i, j-1) = -c(i, j)$. In other words, the four electrodes centered around one common electrode must have the same polarity, opposite to that of the center electrode.

3. Dummy electrode with no electric connection to a piezoelectric element is not allowed in the matrix.

4. It is first proved that $\min(|\Omega|) = 4$. Let us assume that for a particular center electrode m, $|\Omega_m| < 4$. At least 2 of the adjacent electrodes would then be connected to the same piezoelectric elements. Interference would occur when a droplet initially located on the center electrode m is to be actuated to one of these adjacent electrodes. Therefore, we must ensure that $|\Omega| \geq 4$ for any center electrode.

5. For a matrix of any size greater than 3×3, there are at least two adjacent center electrodes, denoted as A and B. In Steps 2 and 4, it is already shown that $|\Omega_A| \geq 4$ and $|\Omega_B| \geq 4$, and that the piezoelectric elements in each set must each have the same polarity. Since the polarities of

electrodes A and B themselves must be opposite (one electrode is adjacent to the other), so do the piezoelectric elements in the two sets. That is, $\Omega_A \cap \Omega_B = \emptyset$. Therefore, $|\Omega_A \cup \Omega_B| = 8$.

Chapter 4 Droplet Routing Using Independence Detection (ID) and Operator Decomposition (OD) Algorithm

4.1 Introduction

The minimization of assay completion time, i.e., the maximization of throughput is essential for portable applications, where sensors can provide early detection and warning. The demands for parallel execution of multiple droplets to achieve this also enables more efficient implementation given our proposed mechanical actuation scheme in portable device. Droplet routing is one of the key steps in digital microfluidic biochip (DMFB) design to optimize assay execution. The goal of droplet routing in a DMFB is to find an efficient route for each droplet from its source to the target, while optimizing the design targets and satisfying the fluidic constraints. In Section 1.2.2, various droplet routing methods were reviewed, which are used to minimize the latest arrival time (T_{la}), number of used cells (# cells), or total routing length [57], [58], [60], [61], [107]–[109]. Most of these methods are based on the decoupled approach, resulting in sub-optimal solutions. The Integer Linear Programming (ILP) or Boolean satisfiability (SAT) based methods, though aiming to find optimal solutions, are computationally expensive, and are only utilized for solving relatively easy problems with cardinality less than 5 [64], [110], [111].

To tackle this dilemma, it is noticed that droplet routing in DMFS is closely related to the multi-agent path finding problem (MAPF) [58]. Previous works on MAPF mainly consists of two types of approaches: the decoupled approach and the centralized approach. The decoupled approaches are mostly based on a reservation system, in which a reservation is placed on the required location and time slots for the droplet that has already been routed [112]. Other decoupled approaches allow agents at a given location to move only in a designated direction, similar to establishing a traffic law [113]. The decoupled approaches are often faster to solve and have better scalability. The centralized methods, however, plan the agents globally and often lead to a solution with completeness and optimality. A complete coupled algorithm combining Operator Decomposition (OD) with Independence Detection (ID) is proposed in [114] (referred to as OD+ID, hereafter), where the agents are divided into small groups, and the optimal solutions for the subgroups are found and finally merged. A novel increasing cost tree search (ICTS) is proposed in [115], where the high-level phase searches the increasing cost tree for a set of cost (per agent), and the low-level phase plans route for every agent under the cost constraint given in the high-level phase. A similar idea of Conflict Based Search (CBS) is proposed in [116], [117], where a constraint-tree (CT) is created and each node in the CT carries the constraints on time and location for each agent. New paths satisfying the constraints given by the node are found through a search algorithm. The performance of centralized algorithms depend on the given problem instance, such as the graph size and topology, the number of agents and branching factor, etc. [116].

In this paper, the droplet routing problem is formulated to an MAPF problem, with a cost function consisting of the total routing cost (length) and the total number of used cells. One of the centralized algorithms, OD+ID [114] is then applied with necessary modifications to solve the problem. Here, our discussions are limited to EWOD routing problems. However, the same algorithm can be applied to EPD with minor changes. The modified OD+ID algorithm shows promising performance on test benchmarks with medium number of droplets. It enhances the quality of results in terms of the latest arrival time, the total routing cost (length), as well as the total number of used cells, when compared with previous state-of-the-art droplet routers. In addition, the algorithm also provides an easy approach to handle the routing of droplets with different routing costs, for example, due to its different viscosity, pH values, etc.

4.2 Problem Formulation

4.2.1 Objective function

The main objective of the droplet routing problem is to transport all the droplets successfully from their start positions to their destinations with minimum routing cost (L_{tot}), i.e., the sum of routing costs for all droplets [58]. As droplets are routed in a concurrent fashion, some literatures alternatively minimize the latest arrival time (T_{la}) for all the routed droplets [57], [59]–[61], [107]. In addition, it is ideal to minimize the number of cells used (*# cells*) in assay operations, for fault tolerance. In this paper, we primarily consider the total routing length L_{tot} as well as the number of used cells (*# cells*), while indirectly minimizing T_{la}*.,* because the former reflects the optimality of the routing solution in general and provides flexibility to assign different costs to droplets of different types. The droplet routing problem can thus be formulated as follows.

Given *n* droplets $D = \{d_1, d_2, \dots, d_n\}$, their start locations and destinations as well as the locations of the other functional modules (blockages) in different time slots, route all droplets from their starts to their destinations while minimizing $\{L_{tot}, \#\, cells\}$, under fluidic and design constraints.

4.2.2 Fluidic Constraint

During concurrent routing of multiple droplets, a minimum spacing needs to be maintained between the droplets themselves as well as between the droplets and the functional modules on the chip, to avoid unwanted merging. For a functional module, a segregation region (usually one electrode distance) is added to the surrounding to separate them from the designed droplet routes. Let D_i and D_j denote two distinct droplets to be routed, and $X_i(t)$, $Y_i(t)$ be the two coordinates of droplet D_i on the microfluidic 2-D array at time *t*. To prevent the two droplets from accidental merging, the following fluidic constraints must be satisfied.

$$|X_i(t) - X_j(t)| \geq 2 \text{ or } |Y_i(t) - Y_j(t)| \geq 2 \qquad (4.1)$$

$$|X_i(t+1) - X_j(t)| \geq 2 \text{ or } |Y_i(t+1) - Y_j(t)| \geq 2 \qquad (4.2)$$

$$|X_i(t) - X_j(t+1)| \geq 2 \text{ or } |Y_i(t) - Y_j(t+1)| \geq 2 \qquad (4.3)$$

where Eq. (4.1) is static fluidic constraint, indicating that if one electrode is occupied by a droplet at time *t*, there will be no droplets on the adjacent eight electrodes at time *t*. Eq. (4.2–4.3) are the dynamic fluidic constraints, requiring that the activated cell for D_i at the next time step cannot be adjacent to the currently activated cell of D_j. The reason is that when more than one electrode surrounding D_j are activated, D_j may result in unpredicted movement.

4.3 Modified OD+ID algorithm

4.3.1 Preliminaries

Operator Decomposition (OD)

In standard A* algorithm with *n* droplets to be routed, the node expansion results in 5^n nodes, where each droplet can and must choose one of the following directions {N, E, W, S, wait} to move to the next time step. Operator Decomposition (OD) decomposes the standard operator in A* by using another representation of the search space, allowing only one droplet to move at each state and takes *n* intermediate states to advance to the next time step, thus reducing the branching factor from 5^n to 5*n* [114]. It significantly reduces the number of searched states compared with that of standard A*, since the intermediate state with higher costs are not expanded. OD is thus adopted in our algorithm to replace the standard A* operator. More detailed description of OD can be found in [114].

Heuristic function

In A* search, a heuristic function is used to estimate the remaining cost of transporting a droplet to its goal state. A good heuristic can reduce the search space and effectively speed up the searching process. In this paper, the shortest path of each cell on the microfluidic 2D array to all the destination cells of all droplets is pre-calculated using the standard breadth-first search, and the results are stored and used as the heuristic function.

4.3.2 Approximate algorithm based on ID+OD

Independence Detection

First, the independence detection (ID) in [114] is reviewed, as it is applied in this paper as the high-level algorithm to solve the routing problem. The pseudocode for ID implementation is provided in Table 4.1. ID first solves each droplet routing problem independently using standard A* algorithm (Table 4.1, line 1-2). Whenever a conflict is found between the two groups *i* and *j* that violates any fluidic constraint, and those two groups were not resolved for conflict previously, ID attempts to re-plan a path for one of the two groups (for example, group *j*). In doing so, ID first "reserves" the path for group *i* by placing reservations on the required positions and time slot in *Reservation*, and then a modified A* search is used to find an alternative path for group *j* without any violation to those time slots and positions already reserved in *Reservation*. (The details of modified A* search is explained later in this section). To guarantee the optimality of the solution, the cost of the newly re-planned path should be equal to the original cost.

Similarly, the route for group *i* is re-planned while reserving the original route of group *j* if the previous attempt fails (Table 4.1, line 11-20). Finally, if both the attempts fail, or if the two conflict groups were resolved for conflict before, the two groups are merged to a new group, and the new merged problem is solved using the modified A* search (Table 4.1, line 24). This whole process is repeated until no conflicts are found between any two groups, i.e., an optimal solution is found for the original problem.

The standard ID in [114] is complete and optimal in terms of minimizing the total path length (L_{tot}). However, in our routing problems, more "congestions" are encountered, and independent groups are found to be easily merged into a new group in our preliminary experiments. For some hard test cases, the cardinality of the newly merged group can easily exceed 5, resulting in excessive computational time and demands for memory to solve the merged problem. Therefore, an approximate algorithm derived from standard ID is adopted, similar to that proposed in [114]. The standard ID has two constraints: Firstly, the re-planned path has a cost limit that does not exceed that of the original path (Table 4.1, line 14, 19). Secondly, the modified A* algorithms expand the nodes in a non-decreasing order of the cost function *f*, which is determined by the total path length (L_{tot}), breaking ties using future conflicts with other groups and used cells. In the approximate algorithms, a maximum group size (MGS) is introduced as an input parameter, and the two constraints are dropped dynamically when the merging of the two conflict groups results in a group size larger than MGS (Table 4.1, line 8-10). Specifically, by dropping the second constraint, the nodes are expanded in a non-increasing order of future conflicts, resulting in less merging of groups and iterations in the future. Since the time for searching is determined by the re-planning of the group with the largest number of droplets, dropping both constraints helps increase the computation speed while sacrificing the optimality of the solution. To calculate the future conflicts, a *conflictAvoidanceTable* (Table 4.1, line 13, 18) is used to register the positions and required time slots for all the other routes except the one under re-planning.

Table 4.1: Independence Detection (approximate algorithm)

```
Algorithm 1: Independence Detection
 1:  Assign each droplet to a group;
 2:  Plan a path for each group using standard A* search;
 3:  Initialize conflict table to record conflict groups having been resolved;
 4:  WHILE $i$ is not the last group
 5:      IF no conflict of group $i$ with any other group
 6:          $i \leftarrow i + 1$; CONTINUE;
 7:      $i, j \leftarrow$ conflict groups;   //conflict exists between group $i$ and $j$//
 8:      IF MGS $\geq$ group_size($i$) + group_size($j$)
 9:          $f \leftarrow$ set future conflict as priority;
10:      ELSE   $f \leftarrow$ set total path length as priority;
11:      IF group $i$ and $j$ have not conflict before (according to conflict table)
12:          Fill Reservation table with path of group $i$;
13:          Fill ConflictAvoidanceTable with all other paths except group $j$;
14:          Find alternative path for group $j$ using modified A* search;
15:          $i \leftarrow \min(i, j)$; CONTINUE;
16:          IF failed to find alternative path for $j$
17:              Fill Reservation table with path of group $j$;
18:              Fill ConflictAvoidanceTable with all other paths except group $i$;
19:              Find alternative path for group $i$ using modified A* search;
20:              $i \leftarrow \min(i, j)$; CONTINUE;
```

```
21:         Merge $i$ and $j$ into one new group;
22:         Re-plan path for the new group without *Reservation*;
23:         Update *conflict table*;
24:         $i \leftarrow$ new group index;
25:    RETURN final paths for all groups;
```

Modified A search based on OD*

The pseudo code for modified A* search algorithm used for low-level path search is summarized in Table 4.2. In the standard A* algorithm, two sets are maintained: *Open* list and *Closed* list. The *Open* list is usually implemented using a priority queue to store the states that need to be expanded and the *Closed* list stores all the states that have been expanded previously, thus cutting off the redundant search space. The *Open* list contains one state initially: the start state A_s, which is determined by the starting positions of all the droplets; the *Closed* list starts empty (Table 4.2, line 1). A* search then tries to find a series of legal transition states to connect the start state A_s to the goal state A_g, essentially converting the routing problem into a graph searching problem.

As mentioned previously, *Reservation* stores the positions and corresponding time slots occupied by the droplets of the conflict group, and *ConflictAvoidanceTable* stores the positions and time slots occupied by all the other droplets. When new states are generated by moving one agent, those that are neither in violation with *Reservation* nor previously expanded (in *Closed* or *Open* list) are then added to the *Open* list (Table 4.2, line 15, 22, 23). Various state attributes (up to this state) such as the routing cost (length), used cells, and number of future conflicts with the other already-planned routes are also calculated (Table 4.2, line 16-20).

At each iteration, the most “promising” state is popped from *Open* list, depending on the cost function *f* and whether approximate search is adopted (Table 4.2, line 20). For optimal solutions, *f* is first determined by the total path length (cost), and breaking ties using the number of conflict and used cells ensures that the algorithm returns the path with smallest future conflicts and used cells among all the optimal solutions, with the smallest path cost (length). For approximate search, however, *f* is non-decreasing with future conflicts, and breaking ties using the total path cost (length) and used cells to avoid merging of groups provides a solution faster. The key modification compared with standard A* is highlighted in blue in Table 4.2.

Table 4.2: Modified A* search

```
Algorithm 2: Modified A* search
 1:  $A_s \leftarrow$ start state;  $A_g \leftarrow$ goal state;
 2:  $Reservation \leftarrow$ reserved routs; $conflictAvoidanceTable \leftarrow$ all other routs;
 3:  $Open \leftarrow \{A_s\}$; $Closed \leftarrow \{\ \}$
 4:  WHILE $Open \neq \emptyset$, BEGIN
 5:    $s \leftarrow$ pop state from $Open$ with smallest $f$;
 6:      IF $s = A_g$    // reach goal state
 7:          IF $s.longest\_time \geq Reservation.longest\_time$
 8:              RETURN $s$ and the path;
 9:          ELSE
10:              $s' \leftarrow s$ with time step + 1 and other remains the same;
11:              Put $s'$ into $Open$ if $s'$ does not conflict with $Reservation$;
12:      ELSE
```

```
13:         children ← assign valid moves to one agent in s;  // use OD state//
14:         For each q in children
15:             IF q has conflict with Reservation, CONTINUE;
16:             q.g ← s.g + cost for last step;
17:             q.h ← cost from q to goal position;
18:             q.used_electrode ← used electrodes of all steps in planned path until q;
19:             q.conflict ← all conflicts with conflictAvoidanceTable until q;
20:             q.f  = f (q.g + q.h, q.used_electrode, q.conflict);
21:             q.predecessor ← s;
22:             IF q' in Open and Closed such that q = q' and q.f < q'f,  CONTINUE;
23:             Put q to Open;
24:         End For
25:         Add s to Closed;
26:     End IF
27: RETURN ∅
```

4.4 Results

The ID-based droplet router was implemented in Python language on an 8-GHz Mac machine with 8 GB memory. To avoid long computation time (>2 h), we used MGS = 1. The experiments were performed on various benchmark suites, including four of the difficult cases in [59] (test 1-4), seven of the difficult cases in [60] (test a-g), and one bioassay: *in vitro*_1 [57], [59], [60]. The maximum number of droplets was 12 for all the chosen cases; however, problems with more droplets may be solved in a reasonable time with improved hardware.

Table 4.3 describes the size of the microfluidic array, the number of droplets (#D) to be routed, and the block area (#Blk) for each benchmark case. The routing results including the latest arrival time (T_{la}) and total number of used cells (*#cells*) were compared with those of the state-of-the-art algorithms [59]–[61]. The cells denoted by "-" are either failed or untested cases by the corresponding algorithms. For the sake of convenience, the results of fast route algorithm [60] was used as the base line for the comparison. Among all the algorithms, the novel droplet routing algorithm [61] resulted in the shortest T_{la} on average, at the expense of a 15% increase in the number of used cells compared with the baseline algorithm. For all the solved cases, the average T_{la} of the high-performance algorithm [59] was 2.56 times that of the baseline algorithm. The proposed algorithm decreases the average T_{la} by ~15% without increasing the number of used cells (~5% decrease), as highlighted in red in Table 4.3. It is also shown that our implemented algorithm achieves 100% routing completion for all the chosen cases with medium number of droplets. Overall, the results show that our algorithm can achieve better timing result and fault tolerance compared with the best-known algorithms on problems with medium number of droplets.

In addition, due to the lack of reporting of total routing cost (length) for most of the algorithms mentioned above, only the total routing length was compared with those reported in [59] for the solved cases (Test 1 and 4). The results show that our algorithm successfully reduces the total routing length by more than 50%, indicating a large improvement in overall routing efficiency.

Table 4.3: Experimental results for selected test benchmarks.

Benchmark Suite				High-Performance[59]		Fast Route [60]		Novel Routing[61]		Ours	
name	size	#D	#Blk	T_{la}	*#cells*	T_{la}	*#cells*	T_{la}	*#cells*	T_{la}	*#cells*
Test 1	12x12	12	23	100	67	39	73	23	63	28	75
Test 2	12x12	12	25	-	-	47	65	21	60	28	69
Test 3	12x12	12	28	-	-	41	48	22	73	33	75
Test 4	12x12	12	31	70	64	38	71	26	109	32	63
Test a	13x13	6	69	-	-	17	51	-	-	15	38
Test b	13x13	5	53	-	-	13	33	-	-	13	34
Test c	16x16	7	95	29	74	24	61	-	-	20	67
Test d	16x16	9	133	-	-	27	87	-	-	28	56
Test e	24x24	10	173	38	170	38	128	-	-	38	117
Test f	24x24	12	215	-	-	45	129	-	-	32	139
Test g	32x32	6	485	-	-	39	121	-	-	39	121
Total				2.56	0.92	1	1	0.6	1.15	0.86	0.95

Table 4.4 compares the results of the baseline algorithm [60] and that of ours for *in vitro_1* bioassay, which involves 11 sub-problems. The maximum latest arrival time (Max. T_{la}), average latest arrival time (Avg. T_{la}), and total number of used cells (*#cells*) among all the sub-problems are summarized. Due to the relative small number of droplets in all the sub problems (<7), the

improvement of our algorithm over [60] for Max. T_{la} and *#cells* are not significant. However, we can see a 20% reduction in Avg. T_{la}. Therefore, our algorithm also performs well on relatively easy bioassays with fewer droplets.

Table 4.4: Results for in vitro compared with [60]

				[60]			Ours		
name	size	#D	#Sub	Max. T_{la}	Avg. T_{la}	*#cells*	Max. T_{la}	Avg. T_{la}	*#cells*
in vitro	16x16	28	11	18	12.47	231	17	**10.43**	229

Figure 4.1 illustrates the routing result of one example test problem with 6 droplets. The start and goal positions of the droplets are marked as squares in different colors; the blockages are indicated by the blue blocks, and the resulted routes for different droplets are marked in red arrows. The stalls of droplets are denoted by "*s*" on the corresponding cells along the routes.

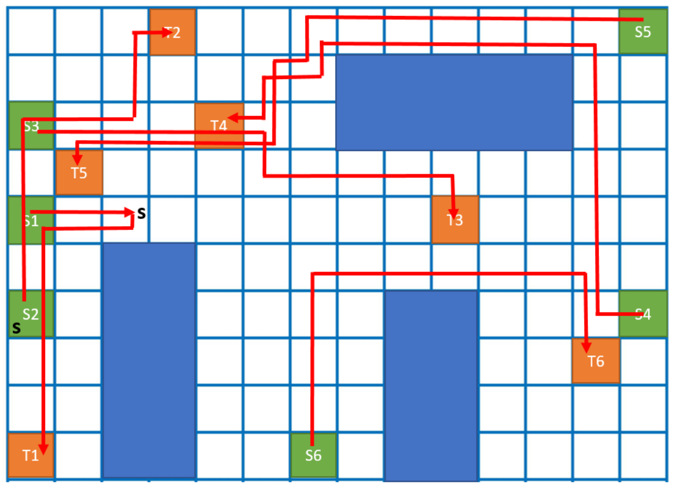


Figure 4.1: Sample droplet routing diagram with 6 droplets. The start and goal positions for all droplets are indicated by squares of green and orange, respectively; blockages are indicated by blue squares and resulted routes are marked using red arrows.

One advantage of our model is its ability to assign different cost to droplets of different types (pH, viscosity, etc.), by including the total routing length (cost) in our objective function. By varying the routing cost of the droplets, for example, by increasing the cost per step for a certain droplet from unit 1 to 10, we can prioritize the optimization for that droplet route at the expense of increasing total cost. Figure 4.2(a) and (b) demonstrate the re-routing results for droplet 4 and 5 for the same test case as in Figure 4.1, with routing cost set at 1 (dashed line) and 10 (solid line), respectively. The increased routing cost reduces the route length for that droplet. Table 4.5 lists the reduction in route lengths for different droplets in the same test case by increasing their routing cost from 1 to 10. Droplets 3 and 6 were not included, since their routes were already at the optimal level in the original solution, as shown in Figure 4.1. Therefore, no further improvement can be made when the routing costs are increased.

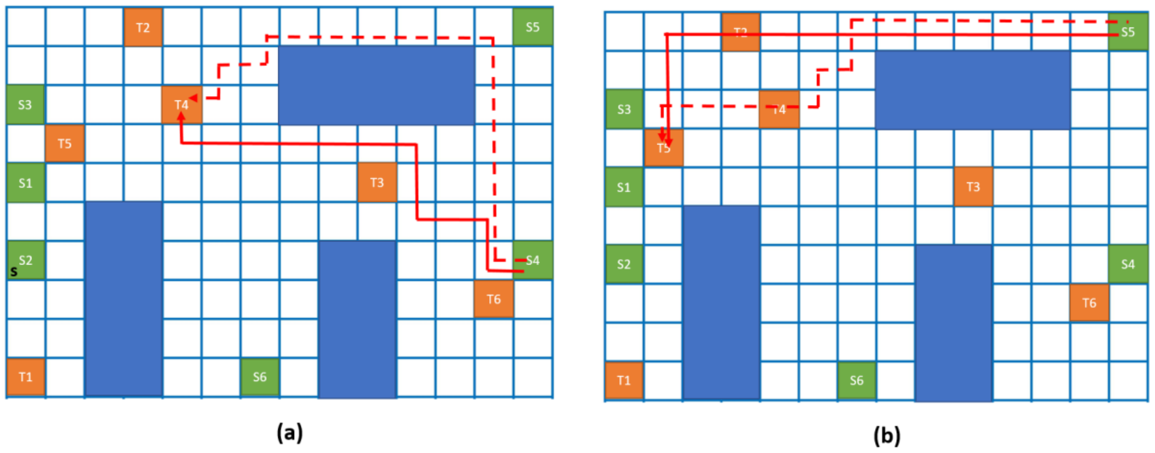


Figure 4.2: Routing results for (a): droplet 4 and (b): droplet 5 in sample test problem after increasing the routing cost from 1 to 10; solid line is the new route and dashed line is the original route.

Table 4.5: Reduction of droplet routing length for sample test problem with increasing routing cost.

No. Droplet	Original length (Cost = 1)	New length (Cost = 10)	Reduction
1	13	12	7.96%
2	12	11	9.09%
4	17	13	23.53%
5	15	12	20.00%
			15.1%

4.5 Summary

A droplet routing algorithm was implemented based on independence detection (ID) and operator decomposition (OD) to solve the routing problem in DMFB design. The basic ID functions by first dividing the droplets into independent groups and then routing them separately. The results are then merged sequentially and when conflict between two groups are found, the two conflict groups are merged as a new group, and legitimate paths for the new group is searched. To increase the computation speed, several constraints were dropped from the basic ID, resulting in an approximate algorithm for droplet routing on DMFB.

Experiments on selected benchmarks demonstrate that our algorithm can achieve 100% routing completion for problems with medium number of droplets ($\leq$12). Overall, the routing results show that our algorithm achieve better timing result and fault tolerance compared with the previous best-known results. In addition, the algorithm provides a flexible approach to route the droplets of different routing costs, while minimizing the total routing cost. All the Python codes used to obtain the results in this paper are available in Appendix 4-A and online at: https://github.com/sophiapeng0426/MAPF.

Appendix 4-A Source Code for droplet routing

Example to run …**/SingleAgent/Solver/IDSolver/EnhancedID.py**:

```
fileroot = '/Users/chengpeng/Documents/MTSL/ElectrodeDesgin/DMFB'
# filename1 = 'test_12_12'
# filename1 = 'benchmark_'
filename1 = 'in-vitro_2.'
for i in range(1, 2):
    # filename = filename1 + '{0}_minsik'.format(i)
    # filename = filename1 + '_{0}.in'.format(i)
    filename = filename1 + '{0}'.format(i)
    saveRoot = 
'/Users/chengpeng/Documents/MTSL/ElectrodeDesgin/Result/{0}/'.format(f
ilename)d
    if not os.path.exists(os.path.dirname(saveRoot)):
        os.makedirs(os.path.dirname(saveRoot))

    testProblem = generateProblem(os.path.join(fileroot, filename))
    testProblem.plotProblem()
    # save probleminstance
    with open(saveRoot + 'InitialProblem.pickle', 'wb') as f:
        pickle.dump(testProblem, f)

    f = open('{0}/discription.txt'.format(saveRoot), 'a+')
    f.write('agent num: {0}'.format(len(testProblem.getAgents())))
    f.close()
```

Sample input file:

```
grid 13 13 500

blk target = 115.2
#0
block 1 2 3 3
#1
block 0 6 3 8
#2
block 1 11 3 12
#3
block 5 3 7 7
#4
block 6 9 9 11
#5
block 9 3 10 4
#6
block 11 1 12 2
```

```
#7
block 11 6 12 7
#8
block 11 10 12 12

blk real = 175
nets 6
net 0 0 12 0 4 8 8
net 1 10 6 0 0 5 11
net 2 4 9 0 0 1 11
net 3 0 3 0 10 0 13
net 4 8 0 0 9 7 8
net 5 10 12 0 7 8 7
```

The following contains all classes used to solve droplet routing problems in this paper:

…/SingleAgent/Solver/Astar/GeneticAStar.py

```
from Queue import PriorityQueue
from SingleAgent.Solver.ConstraintSolver import ConstraintSolver
from SingleAgent.Utilities.StateClosedList import StateClosedList
from SingleAgent.Utilities.ProblemInstance import ProblemInstance


class GeneticAStar(ConstraintSolver):
    def __init__(self):
        """
        _openList: priority queue
        _closeList: stateCloseList
        _goalState: result goal state (used for reconstruct path)
        """
        super(GeneticAStar, self).__init__()
        self._openList = PriorityQueue()
        self._closeList = StateClosedList()
        self._goalState = None
        self._heuristic = None
        self._ignoreCost = False

    def solve(self, problemInstance, fileroot, cost, total):
        """ updated solve
        :param problemInstance:
        :param root:
        :param cost:
        :param total:
        :return:
        """
        import os
        import time

        startTime = time.time()
        assert isinstance(problemInstance, ProblemInstance),
"Initialize solve function require problemInstance"

        self.init(problemInstance)

        # ====== for debug ===========
        # alist = [str(x.getId()) for x in
problemInstance.getAgents()]
        # name = '_'.join(alist)
        # dirname = fileroot + '/log/'
        # # dirname =
'/Users/chengpeng/Documents/MTSL/ElectrodeDesgin/Result/test_16_16_1/l
og3.5t/'
        # if not os.path.exists(os.path.dirname(dirname)):
        #     os.makedirs(os.path.dirname(dirname))
        # f = open('{0}{1}.txt'.format(dirname, name), 'a')
        # f.write(dirname)
        # ===== end ======

        maxgValue = 0
        if total:
            for agent in problemInstance.getAgents():
                maxgValue += agent.getCost()*
problemInstance.getGraph().getSize() * cost
                # maxgValue = maxgValue * self._heuristic.nAgent()

        print("max cost: {0}".format(maxgValue))
        # f.write("max cost: {0}\n".format(maxgValue))

        root = self.createRoot(problemInstance)
        self.setHeuristic(root, 'trueDistance', self._heuristic)
        self._setTables(root, problemInstance)
        self._openList.put(root)

        elapse = 0
        while not self._openList.empty() and self._closeList.size() <
500000 and elapse < 1800:
            elapse = time.time() - startTime
            currentState = self._openList.get()

            if self._closeList.size() % 10000 == 0:
                toPrint = True
            else:
                toPrint = False

            if toPrint:
                print("\nOpenList size: {0};  closedList size ~:
{1}".format(self._openList.qsize(),

self.closeList().size()))

                print("timeStep: {0}, pop:
{1}".format(currentState.timeStep(), currentState))
            # f.write("\nOpenList size: {0};  closedList size ~:
{1}".format(self._openList.qsize(),
            #
self.closeList().size()))
            # f.write("\ntimeStep: {0}, pop:
{1}\n".format(currentState.timeStep(), currentState))

            self._closeList.add(currentState)

            # reach goal state
            if self.isGoal(currentState, problemInstance):
                if currentState.timeStep() >=
self.getReservation().getLastTimeStep():
                    self._goalState = currentState
                    return True
                else:
                    nextgoal =
currentState.generateNextGoal(problemInstance)
                    if self.getReservation().isValid(nextgoal):
                        print("put back {0}, timestep: {1}
".format(nextgoal, nextgoal.timeStep()))
                        # f.write("put back {0}, timestep:
{1}\n".format(nextgoal, nextgoal.timeStep()))

                        self.setHeuristic(nextgoal, 'trueDistance',
self._heuristic)
                        self._setTables(nextgoal, problemInstance)
                        self._openList.put(nextgoal)
                    else:
                        # delete goal in closelist
                        self._closeList.delete(currentState)
                        # ====== experiment with deleting all ========
                        # preS = currentState.getPreState()
                        # while preS is not None:
                        #     if preS in
self._closeList.getClosedList():
                        #         self._closeList.delete(preS)
                        #     preS = preS.getPreState()
                        # ======== end ==========
                        print("Conflict with reservation, do not put
back.")
                        # f.write("Conflict with reservation, do not
put back.\n")

            # not reach goal state
            else:
                potentialStates = currentState.expand(problemInstance)
                for s in potentialStates:
                    self.setHeuristic(s, 'trueDistance',
self._heuristic)
                    self._setTables(s, problemInstance)
                    # ======== each agent maxCost ======
                    lessValue = True
                    if not total:
                        for singleS in s.getSingleAgents():
                            if singleS.fValue() >
problemInstance.getGraph().getSize() * cost * singleS.getStepCost():
                                lessValue = False
                    else:
                        lessValue = s.gValue() + s.hValue() <
maxgValue
                    # ========= end ============
                    if self.getReservation().isValid(s):
                        if lessValue:
                            if toPrint:
                                print(s)
                            # f.write(str(s) + '\n')
                            # agents stays
                            if not self._closeList.contains(s):
                                self._openList.put(s)
                                self._closeList.add(s)
                                if toPrint:
                                    print("add to openlist/closelist")
                                # f.write("add to
openlist/closelist\n")
                            elif s.isStay(currentState):
                                self._openList.put(s)
                                if toPrint:
                                    print("add to openlist")
                                # f.write("add to openlist\n")
                    else:
                        if toPrint:
                            print(s)
                        # f.write(str(s) + "obey violations\n")
        # f.close()
        return False

    def init(self, problemInstance):
        """
        :param problemInstance:
        :param pathList:
```

```
        :return:
        """
        from TDHeuristic import TDHeuristic
        while not self._openList.empty():
            self._openList.get()
        self._closeList.clear()
        self._goalState = None
        # init TDHeuristic
        self._heuristic = TDHeuristic(problemInstance)

    def setIgnore(self, tf):
        if tf:
            print("AStarSolver set violation free as priority.")
        else:
            print("AStarSolver set cost as priority.")
        self._ignoreCost = tf

    def getHeuristicTable(self):
        return self._heuristic

    def setHeuristic(self, s, mode, input):
        s.setHeuristic(mode, input)

    def _setTables(self, s, problemInstance):
        """
        set s._conflictViolations and s._usedElectrode
        :param s: multiagentstate or ODstate, not implemented for
single agentstate
        :return:
        """
        nsize = problemInstance.getGraph().getSize()
        # if self.getUsedTable().isInitialized() is True:
        s.updateUsedElectrode(self.getUsedTable(), nsize)
        if self.getCAT().isEmpty() is False: #
self.getCAT().isInitialized() is True and
            s.updateCATViolations(self.getCAT())

    def isGoal(self, s, problemInstance):
        return s.goalTest(problemInstance)

    def getPath(self):
        """ Get list of states as paths
        :return:
        """
        pathList = []
        if self._goalState is None:
            return pathList

        s = self._goalState
        while s is not None:
            pathList.append(s)
            s = s.predecessor()
        return pathList[::-1]

    def printPath(self):
        """
        :return: print list of states as path
        """
        pathList = self.getPath()
        if len(pathList) == 0:
            print("No path to print")
        for s in pathList:
            print(s)

    def closeList(self):
        return self._closeList

    def createRoot(self, problemInstance):
        """create root node for AStar solver"""
        pass

    def __str__(self):
        return "AStar"
```

…/SingleAgent/Solver/Astar/BreadthFirstSearch.py

```
from GeneticAStar import GeneticAStar


class BreadthFirstSearch(GeneticAStar):
    def __init__(self):
        super(BreadthFirstSearch, self).__init__()

    def simpleInit(self):
        if self._ignoreCost:
            self.setIgnore(False)

        while not self._openList.empty():
            self._openList.get()
        self._closeList.clear()
        self._goalState = None

    def createRoot(self, problemInstance):
        from SingleAgent.States.SingleAgentState import
SingleAgentState

        assert len(problemInstance.getAgents()) == 1, 'breadthfirst
```

```
input > 1 agent problemInstance'
        agentid = problemInstance.getAgents()[0].getId()
        return SingleAgentState.fromProblemIns(agentid,
problemInstance)

    def simpleSolve(self, problemInstance):
        self.simpleInit()

        root = self.createRoot(problemInstance)
        self.setConsHeuristic(root, 0)

        self._openList.put(root)
        self._closeList.add(root)

        while not self._openList.empty():
            currentState = self._openList.get()
            self._closeList.add(currentState)

            potentialStates = currentState.expand(problemInstance)
            for s in potentialStates:
                self.setConsHeuristic(s, 0)
                if not self._closeList.contains(s):
                    self._openList.put(s)
                    self._closeList.add(s)
        return True

    def setConsHeuristic(self, s, cons):
        s.setHeuristic('constant', cons)

    def finalList(self):
        position = []
        distance = []
        for state, _ in self.closeList().getClosedList().items():
            position.append(state.getCoord().getNode().getPosition())
            distance.append(state.gValue())
        return position, distance
```

…/SingleAgent/Solver/Astar/SingleAgentAStar.py

```
from GeneticAStar import GeneticAStar
from SingleAgent.States.SingleAgentState import SingleAgentState


class SingleAgentAStar(GeneticAStar):
    def __init__(self):
        super(SingleAgentAStar, self).__init__()
    def createRoot(self, problemInstance):
        if not self._ignoreCost:
            assert len(problemInstance.getAgents()) == 1, "Not a
singleAgent problemInstance"
            return SingleAgentState.fromProblemIns(0, problemInstance)
        else:
            assert False, 'Not implement violation first.'
```

…/SingleAgent/Solver/Astar/MultiAgentAStar.py

```
from GeneticAStar import GeneticAStar
from SingleAgent.States.MultiAgentState import MultiAgentState


class MultiAgentAStar(GeneticAStar):
    def __init__(self):
        super(MultiAgentAStar, self).__init__()

    def createRoot(self, problemInstance):
        if not self._ignoreCost:
            assert len(problemInstance.getAgents()) >= 1, "Need agent"
            return MultiAgentState.fromProblemIns(problemInstance)
        else:
            assert False, 'Not implement violation first.'

    def visualizePath(self, problemInstance):
        """
        :param problemInstance:
        :return:
        """
        import copy
        # deep copy to prevent changing of map content
        mapContent =
copy.deepcopy(problemInstance.getMap().getContent())
        pathList = self.getPath()
        for state in pathList:
            for singleState in state.getSingleAgents():
                x = singleState.getCoord().getNode().getPosition()[0]
                y = singleState.getCoord().getNode().getPosition()[1]
                mapContent[y][x] = str(singleState.getAgentId())
        for i in mapContent:
            print(' '.join(i))
```

…/SingleAgent/Solver/Astar/ODAStar.py

```
from GeneticAStar import GeneticAStar
from SingleAgent.States.ODState import ODState
from SingleAgent.States.ODState_2 import ODState_2
from SingleAgent.States.SingleAgentState import SingleAgentState


class ODAStar(GeneticAStar):
    def __init__(self):
        """
        init _openList, _closeList and _goalState
        """
        super(ODAStar, self).__init__()

    def createRoot(self, problemInstance):
        assert len(problemInstance.getAgents()) >= 1, "Need agent"
        # if len(problemInstance.getAgents()) == 1:
        #     return
SingleAgentState.fromProblemIns(problemInstance.getAgents()[0].getId()
, problemInstance)
        # else:
        if not self._ignoreCost:
            return ODState.fromProblemIns(problemInstance)
        else:
            return ODState_2.fromProblemIns(problemInstance)

        def visualizePath(self, problemInstance):
        """ print path in map
        :param problemInstance:
        :return:
        """
        import copy
        # deep copy to prevent changing of map content
        mapContent =
copy.deepcopy(problemInstance.getMap().getContent())
        pathList = self.getPath()
        for state in pathList:
            if isinstance(state, ODState):
                for singleState in state.getSingleAgents():
                    x =
singleState.getCoord().getNode().getPosition()[0]
                    y =
singleState.getCoord().getNode().getPosition()[1]
                    mapContent[y][x] = str(singleState.getAgentId())
            elif isinstance(state, SingleAgentState):
                x = state.getCoord().getNode().getPosition()[0]
                y = state.getCoord().getNode().getPosition()[1]
                mapContent[y][x] = str(state.getAgentId())
        for i in mapContent:
            print(' '.join(i))
```

…/SingleAgent/Solver/Astar/TDHeuristic.py

```
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Utilities.Agent import Agent
from SingleAgent.Solver.AStar.BreadthFirstSearch import
BreadthFirstSearch
from SingleAgent.Utilities.Util2 import Util2


class TDHeuristic(object):
    def __init__(self, problemInstance):
        nAgent = len(problemInstance.getAgents())
        self._nsize = problemInstance.getGraph().getSize()
        self._idTable = {}

        # lookupTable[self._idTable[ID]][index] is the distance
        self._lookupTable = [[0 for i in range(self._nsize *
self._nsize)] for j in range(nAgent)]
        self._init(problemInstance)
        print("TDHeuristic initialized, size: {0} x
{1}".format(len(self._lookupTable), len(self._lookupTable[0])))

    def _init(self, problemInstance):
        """
        ID needs to be ranged from 0 to n
        :param problemInstance:
        :return:
        """
        goals = problemInstance.getGoals()

        k = 0
        for ID, goalPos in goals.items():
            newagent = Agent(ID, goalPos, None,
problemInstance.findAgent(ID).getCost())  # fake goal position
            newProblem = ProblemInstance(problemInstance.getGraph(),
[newagent])

            bfs = BreadthFirstSearch()
            bfs.simpleSolve(newProblem)
            position, distance = bfs.finalList()
            for i in range(len(position)):
                index = Util2().posToIndex(position[i], self._nsize)
                self._lookupTable[k][index] = distance[i]
                self._idTable[ID] = k
            k += 1

    def trueDistance(self, agentId, pos):
        index = Util2().posToIndex(pos, self._nsize)
        return self._lookupTable[self._idTable[agentId]][index]

    def nAgent(self):
        return len(self._lookupTable)
```

…/SingleAgent/Solver/IDSolver/EnhancedID.py

```
import pickle
import sys
import os
from IndependentDetection import IDSolver
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Solver.Utils import Util
from SingleAgent.Utilities.Util2 import Util2


class EnhandcedID(IDSolver):
    def __init__(self, solver, maxGroupSize):
        super(EnhandcedID, self).__init__(solver)
        self._mgs = maxGroupSize

        self._conflictInPast = []
        self._dir = None
        self._cost = 3
        self._total = True

    def solve2(self, problemInstance, root, cost, total=True):
        self._cost = cost
        self._total = total
        if total:
            self._dir = root + 'maxg{0}_{1}t'.format(self._mgs, cost)
        else:
            self._dir = root + 'maxg{0}_{1}'.format(self._mgs, cost)

        self._initialProblem = problemInstance
        if len(self._problemList) == 0:
            # start from scratch
            if not self.populatePath(self._initialProblem):
                return False
        self.save(self._dir, 'initial')

        for i in range(len(self.paths())):
            if self.paths()[i] is not None:
                self.solver().getCAT().addPath(self.paths()[i], i)
                self.solver().getUsedTable().addPath(self.paths()[i],
i)

        index = 0
        count = 0
        while index < len(self._pathList):
            conflict = Util().conflict(index, 0, self._pathList)
            if conflict is None or conflict.isEmpty():
                index += 1
            else:
                swallowed = False
                if not self.enhanced_resolveConflict(conflict,
swallowed):
                    return False
                # ======= for debugging ==========
                # else:
                #     return True
                # ======= end ===========
                if not swallowed:
                    # does not swallow
                    index = min(conflict.getGroup1(),
conflict.getGroup2())
                    # index = 0
                    print("-> Go to index: {0}".format(index))
                else:
                    print("-> Swallowed, index: {0}".format(index))

            if conflict is not None and not conflict.isEmpty():
                self.save(self._dir, str(count) + str(conflict))
                count += 1
            elif conflict is None and index == len(self._pathList)-1:
                # === check if last change does not make new
violations ===
                self.save(self._dir, 'solution')
                count += 1

        # ==== record final result =====
        totalCost, usedElectrode, finalPath =
Util().mergePaths(self._pathList, problemInstance)

        if not self.correctcheck(finalPath):
            print("=== Final check is wrong answer! ===")
            headline = "(Wrong)total cost: {0}, used electrode: {1}
\n".format(totalCost, usedElectrode)
            strPath = self.strPath(finalPath)
            print(headline + strPath + '\n')
            # write result to file
            self.recordResult(headline, strPath)
        else:
            # write result
            headline = "total cost: {0}, used electrode: {1}
\n".format(totalCost, usedElectrode)
            strPath = self.strPath(finalPath)
            print(headline + strPath + '\n')
            # write result to file
            self.recordResult(headline, strPath)
        return True

    """ ============ file IO ==============="""
    def recordResult(self, headline, strPath):
        f = open('{0}/result.txt'.format(self._dir), 'a+')
        f.write(headline + strPath + '\n')
        f.close()

    def strPath(self, pathList):
        """print paths"""
        strout = ''
        for t, singleAgents in enumerate(pathList):
            strout += "TimeStep: {0}".format(t)
            gValue = 0
            hValue = 0

            for singleAgent in singleAgents:
                gValue += singleAgent.gValue()
                hValue += singleAgent.hValue()

                strout += '; ' + str(singleAgent)
            strout += '; gValue: {0}; hValue: {1} \n '.format(gValue,
hValue)
        return(strout)

    def save(self, fileDir, filename):
        """save pathlist, problemlist, conflictInPast"""
        sys.setrecursionlimit(20000)

        if not
os.path.exists(os.path.dirname('{0}/{1}.pickle'.format(fileDir,
filename))):

os.makedirs(os.path.dirname('{0}/{1}.pickle'.format(fileDir,
filename)))

        with open('{0}/{1}.pickle'.format(fileDir, filename),
                  'wb') as f1:
            pickle.dump(self.paths(), f1)
            pickle.dump(self.problems(), f1)
            pickle.dump(self._conflictInPast, f1)

    def read(self, fileDir, filename):
        """
        :param fileDir:
        :param filename: '0_1', 'initial', 'solution'
        :return:
        """
        with open('{0}/{1}.pickle'.format(fileDir, filename), 'rb') as
f1:
            self._pathList = pickle.load(f1)
            self._problemList = pickle.load(f1)
            self._conflictInPast = pickle.load(f1)
            # fill solver() cat and usedtable in self.solve()

    """ ============================
    """
    def populatePath(self, problemInstance):
        """
        clear solver() tables, populate problemList, pathList
        """
        # from random import shuffle
        # ===== clear solver() tables ===
        self.clearSolver()
        # ===== clear ProblemList and shuffle ===
        self._problemList[:] = []
        for agent in self._initialProblem.getAgents():
            # problemInstance requires _singleAgents a list!

self._problemList.append(ProblemInstance(self._initialProblem.getGraph
(), [agent]))
        # ===== clear PathList =====
        self._pathList[:] = []
        if not self.solveInitialProblem():
            return False
        self.clearSolver()

        # self._conflictInPast = [[False for i in
range(len(self.paths()))] for j in range(len(self.paths()))]
        self._conflictInPast = [[False for i in
range(len(problemInstance.getAgents()))]
                                for j in range(len(self.paths()))]
        return True

    def solveInitialProblem(self):
        # first iteration
        self.solver().setIgnore(False)
        for ith, problem in enumerate(self._problemList):
            if not self.solver().solve(problem, self._dir, 10000,
True):
                return False
            self.solver().getCAT().addPath(self.solver().getPath(),
ith)

self.solver().getUsedTable().addPath(self.solver().getPath(), ith)
```

```
            self._pathList.append(self.solver().getPath())
        # second iteration
        for ith, problem in enumerate(self._problemList):
            self.solver().getCAT().deletePath(self._pathList[ith],
ith)

self.solver().getUsedTable().deletePath(self._pathList[ith], ith)
            if not self.solver().solve(problem, self._dir, 10000,
True):
                return False

            self.solver().getCAT().addPath(self.solver().getPath(),
ith)

self.solver().getUsedTable().addPath(self.solver().getPath(), ith)
            self._pathList[ith] = self.solver().getPath()
            # ==== for fix bug ===
            self.solver().visualizePath(problem)
            print("total Conflict:
{0}".format(self.solver().getPath()[-1].conflictViolations()))
            # ==== end  ===
        return True


    def enhanced_resolveConflict(self, conflict, swallowed):
        """
        resolve conflict
        :param conflict: new conflict include ids
        :return:
        """
        print("\n==== Resolve Conflict {0} {1}
====".format(conflict.getGroup1(), conflict.getGroup2()))
        # ===== debug: print conflictInPast ===
        # for line in self._conflictInPast:
        #     print(line)
        # ======== end  =====
        faster = conflict.getGroup1()
        slower = conflict.getGroup2()
        fastera = conflict.getAgent1()
        slowera = conflict.getAgent2()

        if sum(self._conflictInPast[faster]) >
sum(self._conflictInPast[slower]):
            faster, slower = slower, faster
            fastera, slowera = slowera, fastera
        elif sum(self._conflictInPast[faster]) ==
sum(self._conflictInPast[slower]):
            if Util().pathLength(self.paths()[faster]) >
Util().pathLength(self.paths()[slower]):
```

```
                faster, slower = slower, faster
                fastera, slowera = slowera, fastera

        totalSize = len(self._problemList[faster].getAgents()) +
len(self._problemList[slower].getAgents())
        oversize = totalSize > self._mgs
        newPath = None

        # === replan group[faster] ====
        print("\n==Replan faster {0}==".format(faster))
        if not self._conflictInPast[faster][slowera]:
            self.solver().getCAT().deletePath(self._pathList[faster],
faster)

self.solver().getUsedTable().deletePath(self._pathList[faster],
faster)

self.solver().getReservation().reservePath(self._pathList[slower])
            newpath = self.findAlternativePath(faster, slower,
oversize)
            self.solver().getReservation().clear()
            if newpath is not None:
                # success
                print("@@Find new path for group {0}".format(faster))
                self._conflictInPast[faster][slowera] = True
                self._pathList[faster] = newpath
                self.solver().getCAT().addPath(newpath, faster)
                self.solver().getUsedTable().addPath(newpath, faster)
                return True
            else:
                # not success
                print("@@Failed replan faster {0}".format(faster))
                self.solver().getCAT().addPath(self._pathList[faster],
faster)

self.solver().getUsedTable().addPath(self._pathList[faster], faster)

        # === replan group[slower] ====
        print("\n==Replan slower {0}==".format(slower))
        if not self._conflictInPast[slower][fastera]:
            self.solver().getCAT().deletePath(self._pathList[slower],
slower)

self.solver().getUsedTable().deletePath(self._pathList[slower],
slower)

self.solver().getReservation().reservePath(self._pathList[faster])
            newpath = self.findAlternativePath(slower, faster,
oversize)
```

```
            self.solver().getReservation().clear()
            if newpath is not None:
                # success
                print("@@Find new path for group {0}".format(slower))
                self._conflictInPast[slower][fastera] = True
                self._pathList[slower] = newpath
                self.solver().getCAT().addPath(newpath, slower)
                self.solver().getUsedTable().addPath(newpath, slower)
                return True
            else:
                # not success
                print("@@Failed replan slower {0}".format(slower))
                self.solver().getCAT().addPath(self._pathList[slower],
slower)

self.solver().getUsedTable().addPath(self._pathList[slower], slower)

        # ==== swallow ===
        print("\n==Plan group together {0}, {1}==".format(faster,
slower))
        swallowed = True
        if totalSize > 4:
            print("Exceed MGS (4), fail to find path.")
            return False
        # does not exceed 4 droplets
        if oversize:
            self.solver().setIgnore(True)
        else:
            self.solver().setIgnore(False)
        # update conflictInPast
        for i in range(len(self._conflictInPast[faster])):
            self._conflictInPast[faster][i] = False
        return self.resolveConflict(faster, slower)

    def findAlternativePath(self, pathId1, pathId2, oversize):
        """
        find alternative for path1
        :param pathId:
        :param oversize:
        :return: path
        """
        # initialize costlimit->infinity large
        costLimit = 10000
        if oversize:
            self.solver().setIgnore(True)
        else:
            self.solver().setIgnore(False)
            costLimit = self.paths()[pathId1][-1].gValue()
        if self.solver().solve(self.problems()[pathId1], self._dir,
self._cost, self._total):
            newpath1 = self.solver().getPath()
            # ====== confirm newpath is no conflict ===========
            if self.haveConflict(newpath1, self.paths()[pathId2]):
                print("Replan failed conflict check.")
                self.solver().getReservation().clear()
                return None
            # ====== end ==========
        else:
            # failed to find alternative path
            return None

        newCost = newpath1[-1].gValue()
        if not oversize:
            if newCost <= costLimit:
                print("=== find new path, new cost: {0}
===".format(newCost))
                self.solver().printPath()
                return newpath1
            else:
                print("=== Failed to find same cost alternative path,
new cost {0} ===".format(newCost))
                return None
        else:
            print("=== find new path, new cost: {0}
===".format(newCost))
            self.solver().printPath()
            return newpath1

    def haveConflict(self, newpath1, path2):
        # ====== confirm newpath is no conflict ===========
        haveConflict = False
        tempPaths = [newpath1, path2]
        tempConflict = Util().conflict(0, 0, tempPaths)
        if tempConflict is not None:
            print(tempConflict)
            haveConflict = True
        assert tempConflict is None or tempConflict.getTimeStep() == 1
        tempConflict = Util().conflict(1, 0, tempPaths)
        if tempConflict is not None:
            print(tempConflict)
            haveConflict = True
        assert tempConflict is None or tempConflict.getTimeStep() == 1
        return haveConflict

    def resolveConflict(self, id1, id2):
        # update _problemList[id1]
        self._problemList[id1].join(self._problemList[id2])
        # exclude paths for id1 and id2 for cat and UsedTable
        self.solver().getUsedTable().deletePath(self.paths()[id1],
id1)
        self.solver().getUsedTable().deletePath(self.paths()[id2],
id2)
        self.solver().getCAT().deletePath(self.paths()[id1], id1)
        self.solver().getCAT().deletePath(self.paths()[id2], id2)

        if not self.solver().solve(self._problemList[id1], self._dir,
self._cost, self._total):
            return False

        print("\nFind path for new group.")
        self.solver().printPath()

        self._pathList[id1] = self._solver.getPath()
        self._problemList[id2] = None
        self._pathList[id2] = None

        # update new paths for cat and usedtable
        self.solver().getCAT().addPath(self.paths()[id1], id1)
        self.solver().getUsedTable().addPath(self.paths()[id1], id1)
        return True


    def updateConflictPast(self, id):
        # set all index from id and to id to false
        self._conflictInPast[id] = [False for i in
range(len(self.paths()))]
        for i in range(len(self._conflictInPast)):
            self._conflictInPast[i][id] = False

    def clearSolver(self):
        self.solver().getReservation().clear()
        self.solver().getCAT().clear()
        self.solver().getUsedTable().clear()


def generateProblem(filename):
    from SingleAgent.Utilities.Agent import Agent
    from SingleAgent.Utilities.Graph import Graph
    from SingleAgent.Utilities.ProblemMap import ProblemMap

    size, block, agentNum, agentList = Util2().readTestFile(filename)
    graph = Graph(ProblemMap(size, block))
    agent = [Agent(x[0], x[1], x[2], 1) for x in agentList]
    problem = ProblemInstance(graph, agent)
    return problem
```

```
def main():
    import time
    from SingleAgent.Solver.AStar.ODAStar import ODAStar
    from SingleAgent.Utilities.ProblemInstance import ProblemInstance
    from SingleAgent.Utilities.Agent import Agent
    from SingleAgent.Utilities.Graph import Graph
    from SingleAgent.Utilities.ProblemMap import ProblemMap

    sys.setrecursionlimit(30000)


    print("============= test case ===============")
    graph1 = Graph(ProblemMap(14, {(2, 5): (5, 2),
                                  (0, 10): (5, 16),
                                  (7, 1): (2, 5),
                                  (8, 6): (4, 2)
                                  }))

    for i in range(1, 101, 20):
        agent1 = [Agent(0, (0, 4), (0, 9), 1),
                  Agent(1, (0, 6), (3, 0), 1),
                  Agent(2, (0, 2), (9, 4), 1),
                  Agent(3, (13, 6), (4, 2), 1),
                  Agent(4, (13, 0), (1, 3), 1),
                  Agent(5, (6, 9), (12, 7), i)
                  ]
        testProblem1 = ProblemInstance(graph1, agent1)
        filename = 'cost_test_agent6_{0}'.format(i)
        saveRoot =
'/Users/chengpeng/Documents/MTSL/ElectrodeDesgin/Result2/{0}/'.format(
filename)
        if not os.path.exists(os.path.dirname(saveRoot)):
            os.makedirs(os.path.dirname(saveRoot))

        for mgs in range(1, 4):
            solver1 = EnhandcedID(ODAStar(), mgs)
            if not solver1.solve2(testProblem1, saveRoot, 3, True):
                print("Failed to solve problem {0} with cost
{1}".format(filename, 3))



if __name__ == '__main__':

     main()
```

…/SingleAgent/Solver/IDSolver/independentDetection.py

```
import copy
import pickle
import sys
from SingleAgent.Solver.AStar.ODAStar import ODAStar
from SingleAgent.Solver.ConstraintSolver import ConstraintSolver
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Solver.Utils import Util
from SingleAgent.Utilities.Util2 import Util2


class IDSolver(ConstraintSolver):
    def __init__(self, solver):
        """ _reservation, _cat
            _pathList: n
            _problemList: n
        """
        super(IDSolver, self).__init__()
        self._pathList = []
        self._problemList = []
        self._solver = solver
        self._initialProblem = None

    def solve(self, problemInstance, fileDir):
        """
        :param problemInstance:
        :param fileDir: save direction for intermediate result
        :return:
        """
        # initial problemInstance
        self._initialProblem = problemInstance
        # self.save(fileDir, 'initialProblem')
        # fill pathList
        if len(self._pathList) == 0:
            if not self.populatePath(self._initialProblem):
                return False
        self.save(fileDir, 'initial')

        for i in range(len(self.paths())):
            if self.paths()[i] is not None:
                self.solver().getCAT().addPath(self.paths()[i], i)
                self.solver().getUsedTable().addPath(self.paths()[i],
i)

        index = 0
        while index < len(self._pathList):
            conflict = Util().conflict(index, 0, self._pathList)
            if conflict is None:
                index += 1
            else:
                if not self.resolveConflict(conflict.getGroup1(),
conflict.getGroup2()):
                    print("fail to find new path")
                    return False
            if conflict is not None:
                self.save(fileDir,
'{0}_{1}'.format(conflict.getGroup1(), conflict.getGroup2()))
            else:
                self.save(fileDir, 'solution')
        return True

    def resolveConflict(self, id1, id2):
        # print("resolve conflict for group: {0}, {1}".format(id1,
id2))
        # update _problemList[id1]
        self._problemList[id1].join(self._problemList[id2])
        # exclude paths for id1 and id2 for cat and UsedTable
        self.solver().getUsedTable().deletePath(self.paths()[id1],
id1)
        self.solver().getUsedTable().deletePath(self.paths()[id2],
id2)
        self.solver().getCAT().deletePath(self.paths()[id1], id1)
        self.solver().getCAT().deletePath(self.paths()[id2], id2)

        # otherPathList = self._pathList[:]
        # otherPathList[id1] = None
        # otherPathList[id2] = None
        if not self.solver().solve(self._problemList[id1]):
            return False

        self.solver().printPath()

        self._pathList[id1] = self._solver.getPath()
        self._problemList[id2] = None
        self._pathList[id2] = None

        # update new paths for cat and usedtable
        self.solver().getCAT().addPath(self.paths()[id1], id1)
        self.solver().getUsedTable().addPath(self.paths()[id1], id1)
        return True

    def populatePath(self, problemInstance):
        for agent in problemInstance.getAgents():
            # problemInstance requires _singleAgents a list!!

self._problemList.append(ProblemInstance(problemInstance.getGraph(),
[agent]))

        for problem in self._problemList:
            # use solver without initializing cat and used table
            if not self.solver().solve(problem):
                return False
            self._pathList.append(self._solver.getPath())
        return True

    def solver(self):
        return self._solver

    def paths(self):
        return self._pathList

    def problems(self):
        return self._problemList

    def getPath(self):
        """ get list of states as path"""
        _, _, finalPath = Util().mergePaths(self._pathList)
        return finalPath

    """ ================== Auxiliary functions
==========================
    """
    def printPath(self):
        """print paths"""
        for t, singleAgents in enumerate(self.getPath()):
            strout = "TimeStep: {0}".format(t)
            gValue = 0
            hValue = 0

            for singleAgent in singleAgents:
                gValue += singleAgent.gValue()
                hValue += singleAgent.hValue()

                strout +='; ' + str(singleAgent)
            strout += '; gValue: {0}; '.format(gValue) + 'hValue:
{0}.'.format(hValue)
            print(strout)

    def visualizePath(self, problemInstance):
        """ print path in map
        :param problemInstance:
        :return:
        """
        # deep copy to prevent changing of map content
        mapContent =
copy.deepcopy(problemInstance.getMap().getContent())
        pathList = self.getPath()
        for singleAgents in pathList:
            for singleState in singleAgents:
                x = singleState.getCoord().getNode().getPosition()[0]
                y = singleState.getCoord().getNode().getPosition()[1]
                mapContent[y][x] = str(singleState.getAgentId())
        for i in mapContent:
            print(' '.join(i))

    def correctcheck(self, pathList):
        """ Check if fluid constraints (static and dynamic) are
violated
        :param pathList:
        :return:
        """
        for ith, state in enumerate(pathList):
            if ith != 0:
                prestate = pathList[ith - 1]
            else:
                prestate = None
            if ith == len(pathList) - 1:
                poststate = None
            else:
                poststate = pathList[ith + 1]

            for i in range(len(state)):
                agent1 = state[i]
                for j in range(0, len(state)):
                    if j != i:
                        agent2 = state[j]
                        if
Util2().withinDis(agent1.getCoord().getNode(),
agent2.getCoord().getNode()):
                            print("=== solution is WRONG ===")
                            return False
                if prestate is not None:
                    for j in range(0, len(prestate)):
                        if j != i:
                            agent2 = prestate[j]
                            if
Util2().withinDis(agent1.getCoord().getNode(),
agent2.getCoord().getNode()):
                                print("=== solution is WRONG ===")
                                return False
                if poststate is not None:
                    for j in range(0, len(poststate)):
                        if j != i:
                            agent2 = poststate[j]
                            if
Util2().withinDis(agent1.getCoord().getNode(),
agent2.getCoord().getNode()):
```

```
                            print("=== solution is WRONG ===")
                            return False
        print("== solution is CORRECT ==")
        return True

    def __str__(self):
        return "IDSolver"

    """ ================== pickle/IO ==========================
    """
    def save(self, fileDir, filename):
        sys.setrecursionlimit(20000)

        with open('{0}/{1}.pickle'.format(fileDir, filename), 'wb') as
f1:
            pickle.dump(self._pathList, f1)
            pickle.dump(self._problemList, f1)

    def read(self, fileDir, filename):
        """
        :param fileDir:
        :param filename: '0_1', 'initial', 'solution'
        :return:
        """
        with open('{0}/{1}.pickle'.format(fileDir, filename), 'rb') as
f1:
            self._pathList = pickle.load(f1)
            self._problemList = pickle.load(f1)
            # fill solver() cat and usedtable in self.solve()
```

…/SingleAgent/Solver/constraintSolver.py

```
import abc
from SingleAgent.Solver.UsedTable import UsedTable
from SingleAgent.Solver.ConflictAvoidanceTable import
ConflictAvoidanceTable
from SingleAgent.Solver.Reservation import Reservation


class ConstraintSolver(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self):
        """
        _reservation: {coord, previous coord}
        _visitTable: {coord, [group]}

        """
        self._reservation = Reservation()
        self._cat = ConflictAvoidanceTable()
        self._usedTable = UsedTable()

    def getReservation(self):
        return self._reservation

    def setReservation(self, reservation):
        self._reservation = reservation

    def getCAT(self):
        return self._cat

    def setCAT(self, catTable):
        self._cat = catTable

    def getUsedTable(self):
        return self._usedTable

    def setUsedTable(self, usedTable):
        self._usedTable = usedTable

    @abc.abstractmethod
    def getPath(self):
        """get list of states as path"""

    @abc.abstractmethod
    def printPath(self):
        """print found paths"""
```

…/SingleAgent/Solver/reservation.py

```
from SingleAgent.Utilities.Coordinate import Coordinate


class Reservation(object):
    """
    No implementation of deletePath, only reserve one path
    """
    def __init__(self):
        """
        _reservedCoordinates: set(coord)
        _agentDestinations: set(pos)
        """
        self._reservedCoordinates = set([])
        self._agentDestinations = set([])
        self._lastTimeStep = 0

    def isEmpty(self):
        return len(self._reservedCoordinates) == 0 and
len(self._agentDestinations) == 0

    def reservedCoordinates(self):
        return self._reservedCoordinates

    def agentDestinations(self):
        return self._agentDestinations

    def reservePath(self, path):
        """
        Can only reserve one path for this implementation
        :param path:
        :return:
        """
        if not self.isEmpty():
            self.clear()
        for i in range(len(path)):
            state = path[i]
            for s in state.getSingleAgents():
                self.reserveSingleAgent(s)
        # add destination state
        last = path[-1]
        for s in last.getSingleAgents():
            self.reserveDestination(s.getCoord())

    def reserveSingleAgent(self, s):
        """
        reserve coord for singleAgent and neighbor coordinates
        :param s:
```

```
        :return:
        """
        # add itself
        coord = s.getCoord()
        self._reservedCoordinates.add(coord)
        if coord.getTimeStep() > 0:
            coord2 = Coordinate(coord.getTimeStep() - 1,
coord.getNode())
            self._reservedCoordinates.add(coord2)
        coord2 = Coordinate(coord.getTimeStep() + 1, coord.getNode())
        self._reservedCoordinates.add(coord2)
        # add neighbor nodes
        for node in coord.getNode().get_neighbor():
            if node is not None:
                neighborCoord = Coordinate(coord.getTimeStep(), node)
                self._reservedCoordinates.add(neighborCoord)
                if coord.getTimeStep() > 0:
                    neighborCoord = Coordinate(coord.getTimeStep() -
1, node)
                    self._reservedCoordinates.add(neighborCoord)
                neighborCoord = Coordinate(coord.getTimeStep() + 1,
node)
                self._reservedCoordinates.add(neighborCoord)

    def reserveDestination(self, coord):
        """
        Aadd coord and neighboring coord to self._agentDestinations
        :param coord:
        :return:
        """
        self._lastTimeStep = coord.getTimeStep()
        self._agentDestinations.add(coord.getNode().getPosition())
        for node in coord.getNode().get_neighbor():
            if node is not None:
                self._agentDestinations.add(node.getPosition())

    def isValid(self, state):
        """
        Check if ODState not-violates reservations
        :param state:
        :return:
        """
        # check state violates reservedCoordinates
        if self.isEmpty():
            return True
        for s in state.getSingleAgents():
            thisCoord = s.getCoord()
            if thisCoord in self._reservedCoordinates:
                return False
            # check state not violates destination
            thisPos = thisCoord.getNode().getPosition()
            thisTimeStep = thisCoord.getTimeStep()
            if thisPos in self._agentDestinations:
                if self._lastTimeStep <= thisTimeStep:
                    return False
        return True

    def freeReservation(self, path):
        """not implemented """
        return

    def getLastTimeStep(self):
        return self._lastTimeStep

    def clear(self):
        self._reservedCoordinates.clear()
        self._agentDestinations.clear()
        self._lastTimeStep = 0
```

…/SingleAgent/Solver/conflictAvoidanceTable.py

```
from SingleAgent.Utilities.Coordinate import Coordinate


class ConflictAvoidanceTable(object):
    def __init__(self):
        """
        _agentDestination: {position: {id, timeStep}}
        """
        self._groupOccupantTable = {}
        self._agentDestination = {}
        self._groupToAgentIndex = {}
        self._initial = False
        # self._earlistConflictWhileAdding = None

    def getSize(self):
        return len(self._groupOccupantTable), 
len(self._agentDestination)

    def groupOccupantTable(self):
        return self._groupOccupantTable

    def agentDestination(self):
        return self._agentDestination

    def isInitialized(self):
        return self._initial

    def isEmpty(self):
        return len(self._groupOccupantTable) == 0 and 
len(self._agentDestination) == 0

    def violation(self, state):
        """ return num of violation in state
        :param state:
        :return:
        """
        movenext = state.getMoveNext()
        if movenext == 0:
            index = len(state.getSingleAgents())
        else:
            index = movenext
        res = 0
        for s in state.getSingleAgents()[0: index]:
            res += self._violationSingleState(s)
        return res
    def _violationSingleState(self, singleAgentState):
        """
        return set of violation groups
        :param singleAgentState:
        :return:
        """
        coordList = []
        coord = singleAgentState.getCoord()
        coordList.append(coord)
        coordList.append(Coordinate(coord.getTimeStep() + 1,
coord.getNode()))
        coordList.append(Coordinate(coord.getTimeStep() - 1,
coord.getNode()))
        # violation in groupOccupantTable
        conflictGroup = set([])
        for coord in coordList:
            if coord in self._groupOccupantTable:
                conflictGroup |= self._groupOccupantTable[coord]

        # violation in destination
        pos = coord.getNode().getPosition()
        if pos in self._agentDestination:
            preDic = self._agentDestination[pos]
            for id, t in preDic.items():
                if coord.getTimeStep() >= t - 1:
                    conflictGroup |= set([id])

        conflictGroup = list(conflictGroup)
        totalconflict = 0
        for group in conflictGroup:
            totalconflict += len(self._groupToAgentIndex[group])
        return totalconflict

    def addPath(self, path, id):
        """
        :param path:
        :return:
        """
        # print("cat add path {0}".format(id))
        self._initial = True
        self._groupToAgentIndex[id] = [x.getAgentId() for x in
path[0].getSingleAgents()]
        # paths include only OD states
        for i in range(len(path)):
            state = path[i]
            # newPositions = [None for i in
range(len(state.getSingleAgents()))]
            for s in state.getSingleAgents():
                self._addSingleAgentState(s, id)
        # add destination state
        last = path[-1]
        for s in last.getSingleAgents():
            self._addDestination(s.getCoord(), id)

    def _addSingleAgentState(self, s, id):
        # add itself
        coord = s.getCoord()
        self._addCoordinate(coord, id)
        # add neighbor nodes
        for node in coord.getNode().get_neighbor():
            if node is not None:
                neighborCoord = Coordinate(coord.getTimeStep(), node)
                self._addCoordinate(neighborCoord, id)

    def _addCoordinate(self, coord, id):
        if coord not in self._groupOccupantTable:
            self._groupOccupantTable[coord] = set([id])
        else:
            self._groupOccupantTable[coord].add(id)

    def _addDestination(self, coord, id):
        self._addDestinationCoord(coord, id)
        for node in coord.getNode().get_neighbor():
            if node is not None:
                newCoord = Coordinate(coord.getTimeStep(), node)
                self._addDestinationCoord(newCoord, id)

    def _addDestinationCoord(self, coordinate, id):
        thisPos = coordinate.getNode().getPosition()
        thisTimeStep = coordinate.getTimeStep()
        if thisPos not in self._agentDestination:
            self._agentDestination[thisPos] = {id: thisTimeStep}
        else:
            preDic = self._agentDestination[thisPos]
            if id not in preDic:
                preDic[id] = thisTimeStep
            elif preDic[id] > thisTimeStep:
                preDic[id] = thisTimeStep

    def deletePath(self, path, id):
        """
        :param path:
        :return:
        """
        # print("cat delete path {0}".format(id))
        del self._groupToAgentIndex[id]
        # delete states in paths
        for i in range(len(path)):
            state = path[i]
            for s in state.getSingleAgents():
                self._deleteSingleAgentState(s, id)
        # delete destination state
        last = path[-1]
        for s in last.getSingleAgents():
            self._deleteDestination(s.getCoord(), id)

    def _deleteSingleAgentState(self, s, id):
        coord = s.getCoord()
        self._deleteCoordinate(coord, id)
        for node in coord.getNode().get_neighbor():
            neighborCoord = Coordinate(coord.getTimeStep(), node)
            self._deleteCoordinate(neighborCoord, id)

    def _deleteCoordinate(self, coord, id):
        if coord in self._groupOccupantTable:
            if id in self._groupOccupantTable[coord]:
                self._groupOccupantTable[coord].remove(id)
                if len(self._groupOccupantTable[coord]) == 0:
                    del self._groupOccupantTable[coord]

    def _deleteDestination(self, coord, id):
        self._deleteDestinationCoord(coord, id)
        for node in coord.getNode().get_neighbor():
            if node is not None:
                newCoord = Coordinate(coord.getTimeStep(), node)
                self._deleteDestinationCoord(newCoord, id)

    def _deleteDestinationCoord(self, coordinate, id):
        thisPos = coordinate.getNode().getPosition()
        if thisPos in self._agentDestination:
            preDic = self._agentDestination[thisPos]
            if id in preDic:
                del preDic[id]
                if len(preDic) == 0:
                    del self._agentDestination[thisPos]

    def clear(self):
        self._groupOccupantTable.clear()
        self._agentDestination.clear()
        self._groupToAgentIndex.clear()
        self._initial = False
```

…/SingleAgent/Solver/UsedTable.py

```
class UsedTable(object):
    def __init__(self):
        """
        cellTable: {
        """
        self._cellTable = {}
        self._initial = False

        def isInitialized(self):
        return self._initial

    def isEmpty(self):
        return len(self._cellTable) == 0

    def addPath(self, path, id):
        # print("usedTable add path {0}".format(id))
        self._initial = True
        for i in range(len(path)):
            thisState = path[i]
            for s in thisState.getSingleAgents():
                pos = s.getCoord().getNode().getPosition()
                if pos not in self._cellTable:
                    self._cellTable[pos] = set([id])
                else:
                    self._cellTable[pos].add(id)

    def deletePath(self, path, id):
        # print("usedTable delete path {0}".format(id))
        if path is None:
            return
        for i in range(len(path)):
            thisState = path[i]
            for s in thisState.getSingleAgents():
                pos = s.getCoord().getNode().getPosition()
                if pos in self._cellTable and id in
self._cellTable[pos]:
                    self._cellTable[pos].remove(id)
                    if len(self._cellTable[pos]) == 0:
                        del self._cellTable[pos]

    def toList(self, size):
        cellList = [0 for i in range(size*size)]
        for pos, _ in self._cellTable.items():
            cellList[pos[0] * size + pos[1]] = 1
        return cellList


    def getSize(self):
        return len(self._cellTable)

    def cellTable(self):
        return self._cellTable

    def clear(self):
        self._cellTable.clear()
        self._initial = False

    def __str__(self):
        return 'a cellTable dictionary'
```

…/SingleAgent/Solver/Utils.py

```
from SingleAgent.Utilities.Conflict import Conflict


class Util(object):
    def conflict(self, index, startIndex, pathList):
        """ return conflict if exists
        :param index: index of path to be checked
        :param startIndex:
        :param pathList: list of paths
        :return:
        """
        thisPath = pathList[index]
        if thisPath is None:
            return None
        for i in range(startIndex, len(pathList)):
            if i != index and pathList[i] is not None:
                path = pathList[i]
                # iterate over each time step
                for t in range(len(thisPath)):
                    isGoalState = False
                    thisState = thisPath[t]
                    if t > len(path) - 1:
                        compareState = path[-1]
                        isGoalState = True
                    else:
                        compareState = path[t]

                    conflict = Conflict(t, index, i, None, None)
                    if self.conflictState2(thisState, compareState,
isGoalState, conflict):
                        # return earliest conflict of thisPath and
pathList
                        return conflict
        return None

    def conflictState2(self, thisState, compareState, isGoal,
conflict):
        """
        return first conflict found include agentId if conflict
exists, more efficient should include
        all conflicts between these two OD states
        :param thisState:
        :param compareState:
        :param isGoal: if compare state is goal state
        :param conflict: changed conflict
        :return:
        """
        for thisS in thisState.getSingleAgents():
            thisP = thisS.getCoord().getNode().getPosition()
            for compareS in compareState.getSingleAgents():
                compareP = set([])
                # current position and neighbor

compareP.add(compareS.getCoord().getNode().getPosition())
                for node in
compareS.getCoord().getNode().get_neighbor():
                    if node is not None:
                        compareP.add(node.getPosition())
                # previous position and neighbor
                if not isGoal and compareS.predecessor() is not None:

compareP.add(compareS.predecessor().getCoord().getNode().getPosition()
)
                    for node in
compareS.predecessor().getCoord().getNode().get_neighbor():
                        if node is not None:
                            compareP.add(node.getPosition())
                if thisP in compareP:
                    conflict.setAgent(thisS.getAgentId(),
compareS.getAgentId())
                    return True

        for thisS in compareState.getSingleAgents():
            thisP = thisS.getCoord().getNode().getPosition()
            for compareS in thisState.getSingleAgents():
                compareP = set([])
                # current position and neighbor

compareP.add(compareS.getCoord().getNode().getPosition())
                for node in
compareS.getCoord().getNode().get_neighbor():
                    if node is not None:
                        compareP.add(node.getPosition())
                # previous position and neighbor
                if compareS.predecessor() is not None:
```

```
compareP.add(compareS.predecessor().getCoord().getNode().getPosition()
)
                    for node in
compareS.predecessor().getCoord().getNode().get_neighbor():
                        if node is not None:
                            compareP.add(node.getPosition())
                if thisP in compareP:
                    conflict.setAgent(compareS.getAgentId(),
thisS.getAgentId())
                    return True
        return False

    def conflictState(self, thisState, compareState, isGoal):
        """ check if two states have time conflict
        :param thisState:
        :param compareState:
        :return: bool
        """
        from SingleAgent.States import ODState
        assert isinstance(thisState, ODState.ODState)
        assert isinstance(compareState, ODState.ODState)

        thisSingleAgents = thisState.getSingleAgents()
        compareSingleAgents = compareState.getSingleAgents()
        thisPos = [state.getCoord().getNode() for state in
thisSingleAgents]
        comparePos = [state.getCoord().getNode() for state in
compareSingleAgents]

        if self._conflictStateHelper(thisSingleAgents, comparePos,
isGoal=False) or \
                self._conflictStateHelper(compareSingleAgents,
thisPos, isGoal):
            return True
        else:
            return False

    def _conflictStateHelper(self, singleAgents, comparePos, isGoal):
        for s in singleAgents:
            staticCons = s.getCoord().getNode().get_neighbor()
            if s.isRoot() or isGoal is True:
                prohibit = set(staticCons)
            else:
                dynamicCons =
s.predecessor().getCoord().getNode().get_neighbor()
                prohibit = set(staticCons) | set(dynamicCons)
            prohibit.add(s)
            if not prohibit.isdisjoint(set(comparePos)):
                # print(singleAgents[0])
                # print(singleAgents[0].getCoord())
                return True
        return False

    def mergePaths(self, pathList, problemInstance):
        """ generate paths for IDSolver
        :param pathList:
        :return: list of list of singleAgents
        """
        paths = filter(lambda x: x is not None, pathList)
        longestLength = self.getLongestPath(paths)
        mergedList = [[] for i in range(longestLength)]
        usedElectrode = set([])
        startOrGoal = problemInstance.startandGoalPosition()
        for t in range(longestLength):
            for path in paths:
                if t < len(path):
                    state = path[t]
                else:
                    state = path[-1]
                for singleAgent in state.getSingleAgents():
                    mergedList[t].append(singleAgent)
                    if singleAgent.getCoord().getNode().getPosition()
not in startOrGoal:

usedElectrode.add(singleAgent.getCoord().getNode().getPosition())
        totalCost = 0
        for path in pathList:
            if path is not None:
                state = path[-1]
                totalCost += state.gValue()
        return totalCost, len(usedElectrode), mergedList

    def getLongestPath(self, pathList):
        r = 0
        for path in pathList:
            if path is not None and len(path) > r:
                r = len(path)
        return r

    def pathLength(self, path):
        reversed = path[::-1]
        i = 0
        while i < len(reversed) - 1:
            if reversed[i] == reversed[i+1]:
                i += 1
            else:
                break
        return len(path)-i-1
```

…/SingleAgent/States/__init__.py

```
__all__ = ["MultiAgentState","ODState","SingleAgentState", "State"]
```

…/SingleAgent/States/State.py

```
import abc


class State(object):
    __metaclass__ = abc.ABCMeta

    def __init__(self, backPointer):
        """
        :param backPointer:
        """
        self._gValue = None
        self._hValue = None
        self._backPointer = backPointer
        self._conflictViolations = 0
        self._usedElectrode = 0

    def predecessor(self):
        return self._backPointer

    def gValue(self):
        return self._gValue

    def hValue(self):
        return self._hValue

    def fValue(self):
        return self.gValue() + self.hValue()

    def usedElectrode(self):
        return self._usedElectrode

    def conflictViolations(self):
        return self._conflictViolations

    def isRoot(self):
        return self._backPointer is None

    """================== astar functions =================
    """
    @abc.abstractmethod
```

```
def expand(self, problemInstance):
    """
    expand current state according to problemInstance
    :param problemInstance:
    :return: a list if states
    """

@abc.abstractmethod
def goalTest(self, problemInstance):
    """
    test if state is goal state
    :param problemInstance:
    :return:
    """
@abc.abstractmethod
def timeStep(self):
    """return timestep of this state"""

"""=========== functions to update member variables =========
"""
@abc.abstractmethod
def setHeuristic(self, mode, input):
    """
    Set hValue
    :param problemInstance:
    :return:
    """
@abc.abstractmethod
def calculateCost(self, problemInstance):
    """
    Calcualte gValue
    :return:
    """
@abc.abstractmethod
def updateCATViolations(self, cat):
    """
    update _conflictViolations
    :param cat:
    :return:
    """

@abc.abstractmethod
def updateUsedElectrode(self, table, nsize):
    """
    udpate _usedElectrode
    :param table:
    :return:
    """
"""================= functions to compare ===============
    """
    @abc.abstractmethod
    def __eq__(self, other):
        """
        for comparable object
        :param other:
        :return: bool
        """
    def __ne__(self, other):
        return not self.__eq__(other)

    @abc.abstractmethod
    def __hash__(self):
        """
        for hash value
        :return:
        """

    def __lt__(self, other):
        """Compare two state for priority queue
        Breaking tie considers smaller hValue
        :param other: same candidate state
        :return:
        """
        """
        TODO: break tie considering _usedElectrode (visitTable)
            ; and _conflictViolations (CAT)
        """
        assert other is not None, "State can not compare with None
type"
        assert not other.hValue() is None or self.hValue() is None,
'set hvalue first.'

        dif = self.gValue() - other.gValue() + self.hValue() -
other.hValue()
        # breaking tie considering hValue
        if dif == 0:
            dif2 = self.conflictViolations() -
other.conflictViolations()
            if dif2 == 0:
                return self.usedElectrode() - other.usedElectrode() <
0
            else:
                return dif2 < 0
            # return self.hValue() - other.hValue() < 0
        else:
            return dif < 0
```

…/SingleAgent/States/SingleAgentState.py

```
from State import State
from SingleAgent.Constants import *
from SingleAgent.Utilities.Coordinate import Coordinate
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Utilities.Node import Node
from SingleAgent.Utilities.Util2 import Util2



class SingleAgentState(State):
    def __init__(self, agentId, currentNode, backPointer,
problemInstance):
        """
        :param agentId:
        :param currentNode: Agent position
        :param backPointer: Agent predecessor
        :param problemInstance:
        """
        assert isinstance(backPointer, State) or backPointer is None,
"para 2 class is not State"
        assert isinstance(problemInstance, ProblemInstance), "para 3
class is not ProblemInstance"
        assert isinstance(currentNode, Node), "para 1 class is not
Node. \n{0}".format(currentNode)

        super(SingleAgentState, self).__init__(backPointer)
        self._agentId = agentId
        self._coord = None
        self._stepCost = problemInstance.findAgent(agentId).getCost()
        self._initializeCoord(currentNode)
        self.calculateCost(problemInstance)  # update gValue

    @classmethod
    def fromProblemIns(cls, agentId, problemInstance):
        """ construct single state based on agentId
        :param problemInstance:
        """
        # if only have one agent, assign agentId to that agent's id
        if len(problemInstance.getAgents()) == 1:
            targetAgent = problemInstance.getAgents()[0]
            agentId = targetAgent.getId()
        else:
            targetAgent = problemInstance.findAgent(agentId)
        startPosition = targetAgent.getStart()
        startNode =
problemInstance.getGraph().getNode()[startPosition[0]][startPosition[1
]]
```

```
        return cls(agentId, startNode, None, problemInstance)

    def _initializeCoord(self, node):
        if self.predecessor() is not None:
            self._coord =
Coordinate(self.predecessor().getCoord().getTimeStep() + 1, node)
        else:
            self._coord = Coordinate(0, node)

    def getCoord(self):
        return self._coord

    def getAgentId(self):
        return self._agentId

    """ ============  functions to update member variable ==========
    """
    def calculateCost(self, problemInstance):
        if self.predecessor() is None:
            self._gValue = 0
            return
        if self == self.predecessor():
            if self.goalTest(problemInstance):
                self._gValue = self.predecessor().gValue()
            else:
                self._gValue = self.predecessor().gValue() +
self._stepCost  # change stay cost
        else:
            self._gValue = self.predecessor().gValue() +
self._stepCost    # cost is changed

    def setHeuristic(self, mode, input):
        """
        :param mode: manhatten or trueDistance
        :param problemInstanceOrHeuristic:
        :return:
        """
        if mode == "manhatten":
            assert isinstance(input, ProblemInstance), "Manhatten
require problemInstance"
            problemInstance = input
            goalPos = problemInstance.getGoals()[self._agentId]
            self._hValue =
self._mDistance(self._coord.getNode().getPosition(), goalPos) *
self._stepCost
        elif mode == "trueDistance":
            # assert isinstance(input, TDHeuristic), "trueDis require
TDHeuristic, {0}, {1}".format(type(input), input)
            heuristic = input
            self._hValue = heuristic.trueDistance(self.getAgentId(),
self.getCoord().getNode().getPosition())
            # print("set true distance: {0}".format(self._hValue))
        elif mode == "constant":
            assert isinstance(input, int)
            self._hValue = input
        else:
            assert False, "unknown heuristic"

    def updateCATViolations(self, cat):
        """ not implemented"""
        # newViolation = cat.violation(self)
        # self._conflictViolations =
self.predecessor().ConflictViolations() + newViolation
        return

    def updateUsedElectrode(self, usedTable, nsize):
        """ not implemented"""
        # tempTable = usedTable.toList(nsize)
        # nsize = usedTable.getSize()
        #
        # current = self
        # while current is not None:
        #     index =
Util2().posToIndex(current.getCoord().getNode().getPosition(), nsize)
        #     tempTable[index] = 1
        #     current = current.predecessor()
        # self._usedElectrode = sum(tempTable)
        return

    def goalSingleAgent(self, problemInstance):
        return SingleAgentState(self._agentId, self._coord.getNode(),
self, problemInstance)

    def getStepCost(self):
        return self._stepCost


    """============  functions for astar ==========
    """
    def expand(self, problemInstance):
        """
        TODO: a list of singleAgentStates corresponding to its
neighbors(valid neighbors)
        :param problemInstance:
        :return:
        """
        stateList = []
        stateList.append(self.waitState(problemInstance))
        for posNode in self._coord.getNode().get_Four():
            if posNode is not None:
                assert isinstance(posNode, Node), "Expanding: neighbor
is not Node type"
                newState = SingleAgentState(self._agentId, posNode,
self, problemInstance)
                stateList.append(newState)
        return stateList

    def _mDistance(self, a, b):
        return abs(a[0] - b[0]) + abs(a[1] - b[1])

    def waitState(self, problemInstance):
        """
        add wait state to state
        :param problemInstance:
        :return: singleAgentState corresponding to wait
        """
        return SingleAgentState(self._agentId, self._coord.getNode(),
self, problemInstance)

    def goalTest(self, problemInstance):
        # problemInstance can be multiple/single
        return self._coord.getNode().getPosition() ==
problemInstance.getGoals()[self._agentId]

    def timeStep(self):
        return self.getCoord().getTimeStep()

    """============  functions for compare ==========
    """
    def __eq__(self, other):
        """
        AgentId, Coordinate.Node (position and type)
        :param other:
        :return:
        """
        if other is None:
            if self is not None:
                return False
        if type(self) != type(other):
            return False
        if self.getAgentId() != other.getAgentId():
            return False
        if other.getCoord() is None:
            if self.getCoord() is not None:
                return False
            else:
                return True
        if other.getCoord().getNode() != self.getCoord().getNode():
            return False
        return True

    def __hash__(self):
        prime = 31
        res = 1
        if self._coord is None:
            return 0
        else:
            res = prime * res + hash(self._coord.getNode()) * prime +
hash(self._agentId)
            return res

    def __str__(self):
        return "{0}: ".format(self._agentId) + "{0}
".format(self._coord.getNode().getPosition()) \
               + "g: {0}".format(self.gValue())
        # return "ID{0}: ".format(self._agentId) + "pins: {0},
".format(self._extraPins) \
        #        + "{0}".format(self._coord.getNode().getPosition())
```

…/SingleAgent/States/MultiAgentState.py

```
import math
from State import State
from SingleAgentState import SingleAgentState
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Utilities.Agent import Agent
from SingleAgent.Utilities.Graph import Graph
from SingleAgent.Utilities.ProblemMap import ProblemMap
from SingleAgent.Utilities.Util2 import Util2


class MultiAgentState(State):
    def __init__(self, backPointer, singleAgents, problemInstance):
        """
        :param backPointer:
        :param singleAgents: List of singleAgentStates
        :param problemInstance:
        """
        # :param costFunction:
        assert isinstance(backPointer, State) or backPointer is None
        assert isinstance(problemInstance, ProblemInstance)
        assert isinstance(singleAgents, list)
        super(MultiAgentState, self).__init__(backPointer)
        self._singleAgents = singleAgents
        self.calculateCost(problemInstance)

    @classmethod
    def fromProblemIns(cls, problemInstance):
        numAgents = len(problemInstance.getAgents())
        singleAgents = []
        for i in range(numAgents):
            agentId = problemInstance.getAgents()[i].getId()

singleAgents.append(SingleAgentState.fromProblemIns(agentId,
problemInstance))
        return cls(None, singleAgents, problemInstance)

    def getSingleAgents(self):
        return self._singleAgents

    """ ============  functions to update member variable ==========
    """
    def calculateCost(self, problemInstance):
        self._gValue = 0
        if self.predecessor() is None:
            return
        for singleState in self._singleAgents:
            self._gValue += singleState.gValue()

    def setHeuristic(self, mode, input):
        self._hValue = 0
        for singleState in self._singleAgents:
            singleState.setHeuristic(mode, input) # first set
heuristic for each singleAgentState
            self._hValue += singleState.hValue()

    def updateCATViolations(self, cat):
        # if cat.violation(self):
        #     newViolation = 1
        # else:
        #     newViolation = 0
        if self.predecessor() is None:
            self._conflictViolations = cat.violation(self)
        else:
            self._conflictViolations =
self.predecessor().conflictViolations() + cat.violation(self)

    def updateUsedElectrode(self, table, nsize):
        tempList = table.toList(nsize)
        current = self
        while current is not None:
            for singleAgent in current.getSingleAgents():
                index = 
Util2().posToIndex(singleAgent.getCoord().getNode().getPosition(), 
nsize)
                tempList[index] = 1
            current = current.predecessor()
        self._usedElectrode = sum(tempList)

    """ ============  functions for astar ==========
    """
    def expand(self, problemInstance):
        """ return valid next states

        :param problemInstance:
        :return:
        """
        candidateList = []
        for singleState in self._singleAgents:
            singleStateNeighborList = 
singleState.expand(problemInstance)
            if len(singleStateNeighborList) == 0:
                # print("No candiate singleState for state: 
{0}".format(singleState))
                return []
            candidateList.append(singleStateNeighborList) # every 
candidate list is none empty
        candidateStateList = self.listProduct(candidateList)

        validStates = []
        for multiState in candidateStateList:
            if self.isValid(multiState):  # a shallow copy to prevent 
change of multistate
                validStates.append(MultiAgentState(self, multiState, 
problemInstance))
        return validStates

    def isValid(self, mstate, StaticOnly = False):
        """check if multiagent states is valid (no collision 
static/dynamic)
        :param mstate: a list of singleAgents states
        :return:
        """
        if len(mstate) == 1:
            return True  # only one agent
        for s in mstate:
            assert isinstance(s, SingleAgentState)
            left = [item.getCoord().getNode() for item in mstate]  #
```

```
copy to prevent changing of mstate
            left.remove(s.getCoord().getNode())
            staticCons = s.getCoord().getNode().get_neighbor()

            if s.isRoot() or StaticOnly:
                prohibit = set(staticCons)
            else:
                dynamicCons =
s.predecessor().getCoord().getNode().get_neighbor()
                prohibit = set(staticCons) | set(dynamicCons)
            prohibit.add(s)

            if not prohibit.isdisjoint(set(left)):
                # print("invalid state: {0} ".format(mstate))
                return False
        return True

    def listProduct(self, alist):
        """
        [[s1 s2 s3], [m1 m2]] every element is None-empty list
        [[si s2], []] => []
        [[]] => [[]]
        catesian product of list elements
        :param alist: list of list of elements [[s1 s2 s3] [m1 m2]]
        :return: list of lists of elements
        """
        assert len(alist) >= 1, "Cannot product None (list)"

        if len(alist) == 1:
            res = []
            for element in alist[0]:
                res.append([element])
            return res

        res = []
        for comb in self.listProduct(alist[0:-1]):
            for element in alist[-1]:
                res.append(comb + [element])
        return res

    def goalTest(self, problemInstance):
        for singleState in self._singleAgents:
            if not singleState.goalTest(problemInstance):
                return False
        return True

    def timeStep(self):
        return self._singleAgents[0].timeStep()
    """===========  functions for compare =========
    """
    def __eq__(self, other):
        """
        each single agent: ID, getCoord.getNode (type and p)
        :param other:
        :return:
        """
        if (other is None) or (not isinstance(other,
MultiAgentState)):
            return False
        if len(self._singleAgents) != len(other.getSingleAgents()):
            return False
        for i in range(len(self._singleAgents)):
            if self._singleAgents[i] != other.getSingleAgents()[i]:
                return False
        return True

    def __hash__(self):
        prime = 31
        res = 1
        if self._singleAgents is None or len(self._singleAgents) == 0:
            return 0
        for singleAgent in self._singleAgents:
            res = prime * res + hash(singleAgent) * prime
        return res

    def __str__(self):
        res = "T: {0}, gValue: {1}, hValue: {2}, violations: {3},
electrode: {4}, ".format(self.timeStep(), self._gValue, self._hValue,

self._conflictViolations, self._usedElectrode)
        for singleState in self._singleAgents:
            res += str(singleState)
            res += '; '
        return res
```

…/SingleAgent/States/ODState.py

```
from MultiAgentState import MultiAgentState
from SingleAgentState import SingleAgentState
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Utilities.Agent import Agent
from SingleAgent.Utilities.Graph import Graph
from SingleAgent.Utilities.ProblemMap import ProblemMap
# from SingleAgent.Utilities.Util2 import Util2
```

```
class ODState(MultiAgentState):
    def __init__(self, backPointer, singleAgents, problemInstance,
moveNext, preState):
        """
        _restrictDir: to-move droplets cannot move to spots occupied
by other droplets in last
              step
        :param backPointer:
        :param singleAgents: list of singleAgents
        :param problemInstance:
        :param moveNext: next agent to move
        :param preState: previous intermediate state
        :param direction: so far assigned directions
        """
        super(ODState, self).__init__(backPointer, singleAgents,
problemInstance)
        self._moveNext = moveNext
        self._preState = preState

        # no restriction by previous
        self._restrictDir = [[] for i in range(0,
len(self._singleAgents))]
        self._updateRestrictDir()

    @classmethod
    def fromProblemIns(cls, problemInstance):
        numAgents = len(problemInstance.getAgents())
        singleAgents = []
        for i in range(numAgents):
            agentId = problemInstance.getAgents()[i].getId()

singleAgents.append(SingleAgentState.fromProblemIns(agentId,
problemInstance))
        return cls(None, singleAgents, problemInstance, 0, None)

    def _updateRestrictDir(self):
        """ update self.allowDir using predecessor
        """
        if self.predecessor() is None or self.isStandard():
            return
        for i in range(self._moveNext, len(self._singleAgents)):
            restrict = [x.getCoord().getNode() for x in
self.predecessor().getSingleAgents()]
            del restrict[i]  # do not count itself
            possibleNodes =
self._singleAgents[i].getCoord().getNode().get_Four()[:]
            for direction, nextNode in enumerate(possibleNodes):
                if (nextNode is not None) and (not
set(nextNode.get_neighbor()).isdisjoint(restrict)):
```

```
                    self._restrictDir[i].append(direction)

    """ ============  functions to update member variable ==========
    """

    """ ============  functions for astar ==========
    """
    def expand(self, problemInstance):
        """ return valid next states (intermediate/standard)
        :param problemInstance:
        :return: newODStates
        """
        currentAgent = self._singleAgents[self._moveNext]
        newSingleStates = currentAgent.expand(problemInstance)
        if newSingleStates is None or len(newSingleStates) == 0:
            return []

        newODStates = []
        for s in newSingleStates:
            mAgents = self._singleAgents[:]
            mAgents[self._moveNext] = s

            if self._moveNext == 0:
                newODStates.append(ODState(self, mAgents,
problemInstance, self.getNewMoveNext(), self))
            else:
                newODStates.append(ODState(self.predecessor(),
mAgents, problemInstance, self.getNewMoveNext(), self))
        validStates = filter(lambda x: self.isValid(x), newODStates)
        return validStates

    def isValid(self, s, StaticOnly = False):
        """ check dynamic and static fluid constraint
        :param s: OD state
        :param StaticOnly:
        :return:
        """
        assert isinstance(s, ODState)
        # standard state
        if s.isStandard():
            return super(ODState, self).isValid(s.getSingleAgents())
        # intermediate state
        movedAgents = s.getSingleAgents()[0:self._moveNext]
        if not super(ODState, self).isValid(movedAgents):
            return False
        if not super(ODState, self).isValid(s.getSingleAgents(),
True):
            return False
        return True
    def goalTest(self, problemInstance):
        if not self.isStandard():
            return False
        return super(ODState, self).goalTest(problemInstance)

    def generateNextGoal(self, problemInstance):
        newAgents = [s.goalSingleAgent(problemInstance) for s in
self._singleAgents]
        return ODState(self, newAgents, problemInstance, 0, self)

    """ ============  auxillary ==========
    """
    def isStandard(self):
        return self._moveNext == 0

    def getNewMoveNext(self):
        if self._moveNext == len(self._singleAgents) - 1:
            newMoveNext = 0
        else:
            newMoveNext = self._moveNext + 1
        return newMoveNext

    def getMoveNext(self):
        return self._moveNext

    def getPreState(self):
        return self._preState

    def getRestricDir(self):
        return self._restrictDir

    def isStay(self, compareState):
        if compareState is None or not isinstance(compareState,
ODState):
            return False
        if not super(ODState, self).__eq__(compareState):
            return False
        return True

    """ ============  functions for compare ==========
    """
    def __eq__(self, other):
        """Compare: each single agent: ID, getCoord.getNode (type and
position)
                moveNext and all possible moves of unassigned agents
        :param other:
        :return:
        """
        if other is None or not isinstance(other, ODState):
            return False
        if not super(ODState, self).__eq__(other):
            return False
        if self._moveNext != other.getMoveNext():
            return False
        return str(self._restrictDir) == str(other.getRestricDir())

    def __hash__(self):
        return super(ODState, self).__hash__() + 23 *
hash(self._moveNext) + hash(str(self._restrictDir))

    def __str__(self):
        res = super(ODState, self).__str__()
        res += '; moveNext: {0}.'.format(self._moveNext)
        return res
```

…/SingleAgent/States/ODState2.py

```
from ODState import ODState
from SingleAgentState import SingleAgentState


class ODState_2(ODState):
    def __init__(self, backPointer, singleAgents, problemInstance,
moveNext, preState):
        super(ODState_2, self).__init__(backPointer, singleAgents,
problemInstance, moveNext, preState)

    @classmethod
    def fromProblemIns(cls, problemInstance):
        numAgents = len(problemInstance.getAgents())
        singleAgents = []
        for i in range(numAgents):
            agentId = problemInstance.getAgents()[i].getId()

singleAgents.append(SingleAgentState.fromProblemIns(agentId,
problemInstance))
        return cls(None, singleAgents, problemInstance, 0, None)

    def expand(self, problemInstance):
        """ return valid next states (intermediate/standard)
        :param problemInstance:
        :return: newODStates
        """
        currentAgent = self._singleAgents[self._moveNext]
        newSingleStates = currentAgent.expand(problemInstance)
        if newSingleStates is None or len(newSingleStates) == 0:
            return []

        newODStates = []
        for s in newSingleStates:
            mAgents = self._singleAgents[:]
            mAgents[self._moveNext] = s

            if self._moveNext == 0:
                newODStates.append(ODState_2(self, mAgents,
problemInstance, self.getNewMoveNext(), self))
            else:
                newODStates.append(ODState_2(self.predecessor(),
mAgents, problemInstance, self.getNewMoveNext(), self))
        validStates = filter(lambda x: self.isValid(x), newODStates)
        return validStates

    def generateNextGoal(self, problemInstance):
        newAgents = [s.goalSingleAgent(problemInstance) for s in
self._singleAgents]
        return ODState_2(self, newAgents, problemInstance, 0, self)

    def setConflict(self, num):
        """helper function to debug"""
        self._conflictViolations = num

    def setgvalue(self, num):
        """helper function to debug"""
        self._gValue = num

    def __lt__(self, other):
        """Compare two state for priority queue
        Breaking tie considers smaller hValue
        :param other: same candidate state
        :return:
        """
        """
        TODO: break tie considering _usedElectrode (visitTable)
            ; and _conflictViolations (CAT)
        """
        assert other is not None, "State can not compare with None
type"
        assert isinstance(other, ODState_2), 'type of other state:
{0}'.format(type(other))
        if other.hValue() is None or self.hValue() is None:
            print("set Heuristic Value first")
            return None
        dif = self.conflictViolations() - other.conflictViolations()
        dif2 = self.gValue() - other.gValue() + self.hValue() -
other.hValue()
```

```
        dif3 = self.usedElectrode() - other.usedElectrode()
        # breaking tie considering hValue
        if dif == 0:
            if dif2 == 0:
                return dif3 < 0
            else:
                return dif2 < 0
            # return self.hValue() - other.hValue() < 0
        else:
            return dif < 0
        # return dif < 0
```

…/SingleAgent/Utilities/__init__.py

…/SingleAgent/Utilities/Agent.py

```
class Agent(object):
    def __init__(self, id, start, goal, step_cost):
        """ start and goal are (x,y) in the graph"""
        self._id = id
        self._start = start
        self._goal = goal
        self._stepCost = step_cost

    def getId(self):
        return self._id

    def getStart(self):
        return self._start

    def getGoal(self):
        return self._goal

    def getCost(self):
        return self._stepCost

    def __lt__(self, other):
        return self._id < other.getId()
```

…/SingleAgent/Utilities/Conflict.py

```
class Conflict(object):
    def __init__(self, timeStep, group1, group2, agent1, agent2):
        self._timeStep = timeStep
        self._group1 = group1
        self._group2 = group2
        self._agent1 = agent1
        self._agent2 = agent2

    def getTimeStep(self):
        return self._timeStep

    def getGroup1(self):
        return self._group1

    def getGroup2(self):
        return self._group2

    def getAgent1(self):
        return self._agent1

    def getAgent2(self):
        return self._agent2

    def setAgent(self, agent1, agent2):
        self._agent1 = agent1
        self._agent2 = agent2

    def setGroup(self, group1, group2):
        self._group1 = group1
        self._group2 = group2

    def isEmpty(self):
        if self._agent1 is None:
            return True
        else:
            return False

    def __str__(self):
        return "g{0}_{1}t{2}".format(self._group1, self._group2,
self._timeStep)

    def __eq__(self, other):
        """ Compares timeStep, group1 and group2
        :param other:
        :return:
        """
        if other is None or type(self) != type(other):
            return False
        if self._timeStep != other.getTimeStep():
            return False
        if self._group1 != other.getGroup1() or self._group2 !=
other.getGroup2():
            return False
        if self._agent1 != other.getAgent1() or  self._agent2 !=
other.getAgent2():
            return False
        return True
```

…/SingleAgent/Utilities/Coordinate.py

```
from Node import Node


class Coordinate(object):
    def __init__(self, timeStep, node):
        self.__timeStep = timeStep
        self.__node = node
    def getTimeStep(self):
        return self.__timeStep

    def getNode(self):
        return self.__node

    def setTimeStep(self, timeStep):
        self.__timeStep = timeStep

    def __eq__(self, other):
        """
        Node + timeStep
        :param other:
        :return:
        """
        if other is None:
            if self.__node is not None:
                return False
            else:
                return True
        if type(self) != type(other):
            return False
        if other.getNode() is None:
            return False
        if other.getNode() != self.__node:
            return False
        if other.getTimeStep() != self.__timeStep:
            return False
        return True

    def __hash__(self):
        """
        Node + timeStep
        :return:
```

```
        """
        prime = 31
        result = 1
        if self.__node is None:
            result = prime * result
        else:
            result = prime * result + hash(self.__node)
        result = prime * result + hash(self.__timeStep)
        return result

    def __str__(self):
        s1 = "timeStep: {0}, ".format(self.__timeStep)
        s2 = "Node: ({0})".format(str(self.__node))
        return "Coordinate: " + s1 + ' ' + s2
```

…/SingleAgent/Utilities/Graph.py

```
from Node import Node
from ProblemMap import ProblemMap
from SingleAgent.Constants import *


class Graph:
    def __init__(self, problemMap):
        """ nodes' neighbor generated in graph
        __nodes: (n-1) x (n-1), (x,y) for position (x,y) in the map,
None of (x,y) is block/out
        :param problemMap: 2D matrix of string
        """
        self.__size = problemMap.getSize()
        self.__map = problemMap
        self.__nodes = [[None for i in range(self.__size)] for j in
range(self.__size)]
#         note: nodes[0][1]._position = (0,1)
        self.__generateGraph()

    def __generateGraph(self):
        """ populate nodes list with nodes, populate nodes neighbors
        """
        for r in range(self.__size):
            for c in range(self.__size):
                if self.__map.getContent()[r][c] != '#':
                    self.__nodes[c][r] = Node((c, r))  # does not tell
border electrode
        #  populate neighbor nodes
        for i in range(self.__size):
            for j in range(self.__size):
                currentNode = self.__nodes[i][j]
                if currentNode is not None:
                    self.__populateNeighbor(currentNode)

    #  populate node neighbor4 and neighbor8
    def __populateNeighbor(self, node):
        """
        :param node: center Node
        :return:
        """
        if node is None:
            return
        current = node.getPosition()
        if self.__map.isNodeValid((current[0], current[1] - 1)):
            up = self.__nodes[current[0]][current[1] - 1]
            node.add_Four(Position.UP, up)
        if self.__map.isNodeValid((current[0] + 1, current[1])):
            right = self.__nodes[current[0] + 1][current[1]]
            node.add_Four(Position.RIGHT, right)
        if self.__map.isNodeValid((current[0], current[1] + 1)):
            down = self.__nodes[current[0]][current[1] + 1]
            node.add_Four(Position.DOWN, down)
        if self.__map.isNodeValid((current[0] - 1, current[1])):
            left = self.__nodes[current[0] - 1][current[1]]
            node.add_Four(Position.LEFT, left)
        if self.__map.isNodeValid((current[0] + 1, current[1] - 1)):
            node.add_Eight(Position.UPRIGHT, self.__nodes[current[0] +
1][current[1] - 1])
        if self.__map.isNodeValid((current[0] + 1, current[1] + 1)):
            node.add_Eight(Position.DOWNRIGHT, self.__nodes[current[0]
+ 1][current[1] + 1])
        if self.__map.isNodeValid((current[0] - 1, current[1] + 1)):
            node.add_Eight(Position.DOWNLEFT, self.__nodes[current[0]
- 1][current[1] + 1])
        if self.__map.isNodeValid((current[0] - 1, current[1] - 1)):
            node.add_Eight(Position.UPLEFT, self.__nodes[current[0] -
1][current[1] - 1])

    def getSize(self):
        return self.__size

    def getNode(self):
        return self.__nodes

    def getMap(self):
        return self.__map

    def plotGraph(self):
        self.__map.plotMap()
```

…/SingleAgent/Utilities/IClosedList.py

```
import abc

class ICLosedList(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def contains(self, state):
        """
        return if state is in the list
        :return:
        """

    def add(self, state):
        """
        add state to list
        :param state:
        :return:
        """

    def clear(self):
        """
        clear list
        :return:
        """
```

…/SingleAgent/Utilities/Node.py

```
class Node(object):
    def __init__(self, position):
        """
        :param type: string for wall/block/. etc.
        :param position: tuple for the (x,y)
        neighborFour: Valid 4 neighbors
        neighborEight: valid 8 neighbors
        """
        self.__position = position
        self.__neighborFour = [None for i in range(4)]
        self.__neighborEight = [None for i in range(4)]

    def add_Four(self, index, newNode):
        self.__neighborFour[index] = newNode
    def add_Eight(self, index, newNode):
        self.__neighborEight[index] = newNode

    def get_Four(self):
        return self.__neighborFour

    def get_Eight(self):
        return self.__neighborEight

    def get_neighbor(self):
        return self.__neighborFour + self.__neighborEight

    def getPosition(self):
        return self.__position

    def __hash__(self):
        return hash(self.__position)

    def __eq__(self, other):
        if other is None or type(self) != type(other):
            return False
        if self.__position != other.getPosition():
            return False
        return True

    def __ne__(self, other):
        return not self.__eq__(other)

    def __str__(self):
        return "Pos: {0}".format(self.__position)

    def __getstate__(self):
        return self.__dict__

    def __setstate__(self, d):
        self.__dict__ = d
```

…/SingleAgent/Utilities/ProblemInstance.py

```
import copy
from Graph import Graph
from Agent import Agent
from ProblemMap import ProblemMap
from Util2 import Util2


class ProblemInstance(object):
    def __init__(self, graph, agents):
        """
        _goals: {agentId: (x, y)}
        :param graph:
        :param agents: list of Agents, example [Agent(0,
(1,1),(10,10), 1)]
        """
        self.__graph = graph
        self.__agents = agents
        self.__goals = {}
        self.__initiateGoals()
        assert self.__duplicateGoalsOrStarts() == False, "Agent
initial or goal positions duplicates."

    def __initiateGoals(self):
        self.__goals.clear()
        for agent in self.__agents:
            self.__goals[agent.getId()] = agent.getGoal()

    def __duplicateGoalsOrStarts(self):
        """ check if have duplicates
        :return: True or False
        """
        """ Merge operation? """
        for i in range(len(self.__agents)):
            for j in range(i+1, len(self.__agents)):
                agent1 = self.__agents[i]
                agent2 = self.__agents[j]
                if Util2().withinDis(agent1.getStart(),
agent2.getStart()) or \
                        Util2().withinDis(agent1.getGoal(),
agent2.getGoal()):
                    return True
        return False

    def join(self, otherInstance):
        """ merge self with otherInstance
        :param otherInstance:
        :return: new problem instance
        """
        for agent in otherInstance.getAgents():
            self.__agents.append(agent)
            self.__goals[agent.getId()] = agent.getGoal()
        assert self.__duplicateGoalsOrStarts() == False, "Agent
initial or goal positions duplicates."

    def addAgent(self, agent):
        if isinstance(agent, list):
            self.__agents += agent
            for s in agent:
                self.__goals[s.getId()] = s.getGoal()
        elif isinstance(agent, Agent):
            self.__agents += [agent]
            self.__goals[agent.getId()] = agent.getGoal()
        assert self.__duplicateGoalsOrStarts() == False, "Agent
initial or goal positions duplicates."

    def removeAgent(self, agentId):
        """TODO: """
        if isinstance(agentId, int):
            pass

    def findAgent(self, agentId):
        """return agent = agentId"""
        for agent in self.__agents:
            if agent.getId() == agentId:
                return agent

    def getAgents(self):
        return self.__agents

    def getGraph(self):
        return self.__graph

    def getMap(self):
        return self.getGraph().getMap()

    def getGoals(self):
        return self.__goals

    def startandGoalPosition(self):
        """
        :return: set of start and goal positions
        """
        pos = set([])
        for agent in self.__agents:
            pos.add(agent.getStart())
            pos.add(agent.getGoal())
        return pos

    def plotProblem(self):
        """
        graph + agents for visualization
        :return:
        """
        mapContent = copy.deepcopy(self.getMap().getContent())
        for agent in self.__agents:
            mapContent[agent.getStart()[1]][agent.getStart()[0]] =
str(agent.getId())
```

```
            mapContent[agent.getGoal()[1]][agent.getGoal()[0]] = 
str(agent.getId())
        for i in mapContent:
            print(' '.join(i))
            # print('\n')
```

…/SingleAgent/Utilities/ProblemMap.py

```
from SingleAgent.Constants import *


class ProblemMap(object):
    # constructor for known content (a 2D matrix of string)
    def __init__(self, nsize, block):
        """
        __content: (n+1) x (n+1) of chars
        :param height:
        :param width:
        :param block: dictionary {(x,y), (m,n)}
        """
        self.__size = nsize + 1
        self.__content = []
        assert isinstance(block, dict)
        self._generateMap(block)

    def _generateMap(self, block):
        res = [[Symbols.BLANK for i in range(self.__size)] for j in 
range(self.__size)]
        # wall *
        for i in range(self.__size):
            res[i][0] = Symbols.WALL
            res[i][self.__size - 1] = Symbols.WALL
        for j in range(self.__size):
            res[0][j] = Symbols.WALL
            res[self.__size - 1][j] = Symbols.WALL
        # add blocks #
        for pos, size in block.items():
            for i in range(size[1]):
                for j in range(size[0]):
                    x = pos[0] + i
                    y = pos[1] + j
                    if self._existCell(pos) and not 
self.__outOfBorder((x, y)):
                        res[y][x] = Symbols.BLOCK
        self.__content = res
    def _existCell(self, pos):
        if pos[0] == self.__size - 1 or pos[1] == self.__size - 1:
            return False
        else:
            return True

    def __outOfBorder(self, position):
        return position[0] > self.__size - 1 or position[1] >
self.__size - 1 or position[0] < 0 or position[1] < 0

    # visualize the generated map
    def plotMap(self):
        for i in self.__content:
            print(' '.join(i))

    def getSize(self):
        return self.__size - 1

    def isNodeValid(self, position):
        """
        check if the position is occupied or out of bound
        :param position: (x,y)
        :return: bool
        """
        if position[0] < 0 or position[1] < 0:
            return False
        if position[0] >= self.__size - 1 or position[1] >=
self.__size - 1:
            return False
        if self.__content[position[1]][position[0]] == '#':
            return False
        return True

    def getContent(self):
        return self.__content
```

…/SingleAgent/Utilities/StateClosedList.py

```
from IClosedList import ICLosedList
from SingleAgent.Utilities.ProblemInstance import ProblemInstance
from SingleAgent.Utilities.Node import Node
from SingleAgent.Utilities.Agent import Agent
from SingleAgent.Utilities.Graph import Graph
from SingleAgent.Utilities.ProblemMap import ProblemMap


class StateClosedList(ICLosedList):
    def __init__(self):
        self._closeSet = dict()

    def contains(self, state):
        from SingleAgent.States.State import State
        assert isinstance(state, State), "ClosedList contains requires
state class"
        if state not in self._closeSet:
            return False
        preState = self._closeSet[state]
        if state < preState:
            del self._closeSet[preState]
            return False
                return True

    def add(self, state):
        """ rewrite state
        """
        from SingleAgent.States.State import State
        assert isinstance(state, State), "ClosedList add requires
state class"
        self._closeSet[state] = state

    def delete(self, state):
        assert state in self._closeSet, 'state not in StateClosedList,
\n{0}'.format(state)
        del self._closeSet[state]

    def clear(self):
        self._closeSet.clear()

    def getClosedList(self):
        return self._closeSet

    def empty(self):
        return len(self._closeSet) == 0

    def size(self):
        return len(self._closeSet)

    def __str__(self):
        res = ''
        for key, value in self._closeSet.items():
            res += str(key) + '\n'
        return res
```

…/SingleAgent/Utilities/Util2.py

```
class Util2(object):
    def oppositeDir(self, dir):
        if dir == 0:
            return 2
        elif dir == 2:
            return 0
        elif dir == 1:
            return 3
        elif dir == 3:
            return 1

    def moveDir(self, preNode, nextNode):
        """find moving direction of preNode -> nextNode"""
        pos1 = preNode.getPosition()
        pos2 = nextNode.getPosition()
        if pos1[0] == pos2[0] and pos1[1] == pos2[1]:
            return 0
        elif pos1[1] - pos2[1] == 1:
            return 1
        elif pos1[0] - pos2[0] == -1:
            return 2
        elif pos1[1] - pos2[1] == -1:
            return 3
        elif pos1[0] - pos2[0] == 1:
            return 4
        return None

    def withinDis(self, node1, node2):
        if isinstance(node1, tuple):
            return abs(node1[0] - node2[0]) < 2 and abs(node1[1] -
node2[1]) < 2
        else:
            return abs(node1.getPosition()[0] -
node2.getPosition()[0]) < 2 \
               and abs(node1.getPosition()[1] -
node2.getPosition()[1]) < 2

    def posToIndex(self, pos, nsize):
        return pos[0] * nsize + pos[1]

    def indexToPos(self, index, nsize):
        x = index // nsize
        y = index % nsize
        return (x,y)

    # ================ Functions IO ================
    def readTestFile(self, filename):
        """
        extract probleminstance parameters from file
        :param filename:
        :return: size, block, agentlist
        """
        size = 0
        agentNum = 0
        block = {}
        agentList = []
        f = open(filename, 'r')
        for line in f:
            if line[0] != '#':
                c = line.split(' ')
                if c[0] == 'grid':
                    size = int(line[5:7])
                elif c[0] =='block':
                    block[(int(c[2]), int(c[1]))] = (int(c[3]) -
int(c[1]) + 1, int(c[4]) - int(c[2]) + 1)
                elif c[0] == 'nets':
                    agentNum = int(c[1])
                elif c[0] == 'net' or c[0] == 'xet':
                    print(c)
                    agentList.append([int(c[1]), (int(c[3]),
int(c[2])), (int(c[6]), int(c[5]))])
        f.close()
        print(size)
        print(block)
        print(agentNum)
        print(agentList)
        return size, block, agentNum, agentList
```

…/SingleAgent/Constants/__init__.py

```
__all__ = ["Block","Symbols","costs", "Position"]
```

…/SingleAgent/Constants/Block.py

```
SMALL = 1
MEDIUM = 2
LARGE = 4
```

…/SingleAgent/Constants/costs.py

```
STAY = 1
```

…/SingleAgent/Constants/Position.py

```
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3
UPRIGHT = 0
DOWNRIGHT = 1
DOWNLEFT = 2
UPLEFT = 3
STAY = 4
```

…/SingleAgent/Constants/Symbols.py

```
WALL = '*'
BLOCK = '#'
BLANK = '.'
AGENTSTART = '@'
AGENTGOAL = '&'
```

Chapter 5 Summary and Recommendations

This work demonstrates finger-powered digital microfluidics based on EWOD and EPD for portable applications. The voltage output of single/multiple piezoelectric elements in series connection were characterized under different bending angles for powering digital microfluidic devices. EWOD devices of various thicknesses were designed and fabricated. The basic modes of droplet manipulation such as droplet transport, merging, and splitting using finger-powered EWOD DMF were confirmed. The basic assay steps of glucose detection and immunoassay were also implemented. To overcome the pinning and surface contamination of EWOD, the same energy conversion scheme was applied to an alternative fluidic manipulation paradigm: electrophoretic transport of discrete droplets (EPD). To establish the design criteria for finger-powered EPD, numerical models for predicting the induced droplet charges and subsequent electrophoretic forces for various droplet sizes and electrode pitches were developed and experimentally validated. The transport of aqueous droplets, as well as the direct manipulation of body fluids, were demonstrated using the finger-powered EPD. Further, a mechanical hand-rotated drum system and an efficient pin assignment method was integrated into the final system to demonstrate pre-programmed functional droplet actuation.

In addition, an OD+ID based droplet router was implemented to solve one of the practical problems in microfluidic chip design: droplet routing problem. The routing results on selected hard benchmarks show that our algorithm achieves better timing result (latest arrival time), fault tolerance (number of used cells), as well as total routing cost, compared with the previous best known results. The algorithm also provides a flexible approach to routing droplets of different routing costs (due to different values of viscosity, pH, etc.) while minimizing the total routing cost.

The following recommendations are made for future research on finger-powered digital microfluidic devices.

1. The basic functions of transport and merging in EPD are established in this work. However, for implementation of full bioassays, other fluidic functions such as droplet splitting are necessary. A reliable splitting mechanism needs to be developed and integrated into the current EPD. Alternatively, EPD can function as the transportation unit in an integrated digital microfluidic solution, which may combine multiple droplet actuation mechanisms such as EWOD and surface acoustic wave. In the latter scenario, a proper intermediate component for connecting the different functional units needs to be developed.
2. To further improve the reliability of EPD and reduce the accidental sticking on electrodes, shorter touching time, smaller touching area, as well as a hydrophobic solid surface are desirable. Therefore, more research can be done on suitable material selection and fabrication methods of EPD electrodes.
3. With a few additional functions added to our current droplet routing algorithm based on ID, 3-net problems (including droplet merging and splitting) may be solved directly within the algorithm.

Reference

[1] L. Gervais, N. De Rooij, and E. Delamarche, “Microfluidic chips for point-of-care immunodiagnostics,” *Adv. Mater.*, vol. 23, no. 24, 2011.

[2] E. K. Sackmann, A. L. Fulton, and D. J. Beebe, “The present and future role of microfluidics in biomedical research.,” *Nature*, vol. 507, no. 7491, pp. 181–9, 2014.

[3] M. Rohan, “Point-of-Care Diagnostics Market worth 36.96 Billion USD by 2021,” *Markets and Markets*. [Online]. Available: http://www.marketsandmarkets.com/PressReleases/point-of-care-diagnostic.asp. [Accessed: 05-Feb-2017].

[4] Y. S. Zhang *et al.*, “Google Glass-Directed Monitoring and Control of Microfluidic Biosensors and Actuators.,” *Sci. Rep.*, vol. 6, p. 22237, 2016.

[5] N. Meredith, C. Quinn, D. Cate, T. Reilly, J. Volckens, and C. Henry, “Paper-Based Analytical Devices for Environmental Analysis,” *Analyst*, vol. 141, pp. 1874–1887, 2016.

[6] B. Bruijns, A. van Asten, R. Tiggelaar, and H. Gardeniers, “Microfluidic devices for forensic DNA analysis: A review,” *Biosensors*, vol. 6, no. 3, pp. 1–35, 2016.

[7] S. Jovanovich *et al.*, “Developmental validation of a fully integrated sample-to-profile rapid human identification system for processing single-source reference buccal samples,” *Forensic Sci. Int. Genet.*, vol. 16, pp. 181–194, 2015.

[8] S. Choi, “Powering point-of-care diagnostic devices,” *Biotechnol. Adv.*, vol. 34, no. 3, pp. 321–330, 2016.

[9] P. Zwanenburg, X. Li, and X. Y. Liu, “Magnetic valves with programmable timing capability for fluid control in paper-based microfluidics,” *Proc. IEEE Int. Conf. Micro Electro Mech. Syst.*, pp. 253–256, 2013.

[10] D.-H. Kim *et al.*, “Dissolvable films of silk fibroin for ultrathin conformal bio-integrated electronics.,” *Nat. Mater.*, vol. 9, no. 6, pp. 511–7, 2010.

[11] S. A. Bhakta, R. Borba, M. Taba, C. D. Garcia, and E. Carrilho, “Determination of nitrite in saliva using microfluidic paper-based analytical devices,” *Anal. Chim. Acta*, vol. 809, pp. 117–122, 2014.

[12] J. H. Son *et al.*, “Hemolysis-free blood plasma separation.,” *Lab Chip*, vol. 14, no. 13, pp. 2287–92, 2014.

[13] L. Gervais and E. Delamarche, “Toward one-step point-of-care immunodiagnostics using capillary-driven microfluidics and PDMS substrates.,” *Lab Chip*, vol. 9, no. 23, pp. 3330–3337, 2009.

[14] D. Juncker *et al.*, “Autonomous Microfluidic Capillary System,” vol. 74, no. 24, pp. 1–6, 2003.

[15] K. K. Lee and C. H. Ahn, “A new on-chip whole blood/plasma separator driven by asymmetric capillary forces.,” *Lab Chip*, vol. 13, no. 16, pp. 3261–7, 2013.

[16] I. K. Dimov *et al.*, “Stand-alone self-powered integrated microfluidic blood analysis system (SIMBAS),” *Lab Chip*, vol. 11, no. 5, pp. 845–850, 2011.

[17] I. K. Dimov *et al.*, “Stand-alone self-powered integrated microfluidic blood analysis system (SIMBAS),” *Lab Chip*, vol. 11, no. 5, pp. 845–850, 2011.

[18] F. B. Myers, R. H. Henrikson, J. Bone, and L. P. Lee, “A Handheld Point-of-Care Genomic Diagnostic System,” *PLoS One*, vol. 8, no. 8, 2013.

[19] A. W. Martinez, S. T. Phillips, M. J. Butte, and G. M. Whitesides, “Patterned paper as a platform for inexpensive, low-volume, portable bioassays,” *Angew. Chemie - Int. Ed.*, vol. 46, no. 8, pp. 1318–1320, 2007.

[20] K. Miyamoto, R. Yamamoto, K. Kawai, and S. Shoji, “Stand-alone microfluidic system using partly disposable PDMS microwell array for high throughput cell analysis,” in *Sensors and Actuators, A: Physical*, vol. 188, pp. 133–140, 2012.

[21] K. A. Addae-Mensah, Y. K. Cheung, V. Fekete, M. S. Rendely, and S. K. Sia, “Actuation of elastomeric microvalves in point-of-care settings using handheld, battery-powered instrumentation.,” *Lab Chip*, vol. 10, pp. 1618–1622, 2010.

[22] Y. Lan *et al.*, “Polyoxometalate-based metal-organic framework-derived hybrid electrocatalysts for highly efficient hydrogen evolution reaction,” *J. Mater. Chem. A Mater. energy Sustain.*, vol. 0, pp. 1–6, 2013.

[23] C.-H. Shih, H.-C. Wu, C.-Y. Chang, W.-H. Huang, and Y.-F. Yang, “An enzyme-linked immunosorbent assay on a centrifugal platform using magnetic beads.,” *Biomicrofluidics*, vol. 8, no. 5, p. 52110, 2014.

[24] S. Z. Andreasen *et al.*, “Integrating Electrochemical Detection with Centrifugal Microfluidics for Real-Time and Fully Automated Sample Testing,” *RSC Adv*, vol. 5, no. 22, pp. 17187–17193, 2015.

[25] R. J. M. Vullers, R. van Schaijk, I. Doms, C. Van Hoof, and R. Mertens, “Micropower energy harvesting,” *Solid. State. Electron.*, vol. 53, no. 7, pp. 684–693, 2009.

[26] J. Olivo, S. Carrara, G. De Micheli, and G. De Micheli, “Energy Harvesting and Remote Powering for Implantable Biosensors,” *IEEE Sens. J.*, vol. 11, no. 7, pp. 1573–1586, 2011.

[27] H. Zhang *et al.*, “Triboelectric nanogenerator as self-powered active sensors for detecting liquid/gaseous water/ethanol,” *Nano Energy*, vol. 2, no. 5, pp. 693–701, 2013.

[28] H. Zhang, Y. Yang, T. C. Hou, Y. Su, C. Hu, and Z. L. Wang, “Triboelectric nanogenerator built inside clothes for self-powered glucose biosensors,” *Nano Energy*, vol. 2, no. 5, pp. 1019–1024, 2013.

[29] L. Jiang, M. Mancuso, and D. Erickson, “Light-Driven Microfluidics Towards Solar-Powered Point-of-Care Diagnostics,” in *16th International Conference on Miniaturized Systems for Chemistry and Life Sciences*, pp. 1333–1335, 2012

[30] K. Iwai, R. D. Sochol, L. P. Lee, and L. Lin, “Finger-Powered Bead-in-Droplet Microfluidic System for Point-of-Care Diagnostics,” *Proc. IEEE 25th Int. Conf. Micro Electro Mech. Syst.*, pp. 949–952, 2012.

[31] K. Iwai, R. D. Sochol, and L. Lin, “Finger-powered, pressure-driven microfluidic pump,” *Proc. IEEE Int. Conf. Micro Electro Mech. Syst.*, pp. 1131–1134, 2011.

[32] K. Iwai, A. T. Higa, R. D. Sochol, and L. Lin, “Finger-powered microdroplet generator,” *Proc. 16th Int. Solid-State Sensors, Actuators Microsystems Conf.*, pp. 230–233, 2011.

[33] C. Liu *et al.*, “ATimer-Actuated, Immunoassay Cassette for Detecting Molecular Markers in Oral Fluids,” *Lab Chip*, vol. 36, no. 3, pp. 490–499, 2010.

[34] G. Korir and M. Prakash, “Punch Card Programmable Microfluidics,” *PLoS One*, vol. 10, no. 3, pp. e0115993, Mar. 2015.

[35] E. Samiei *et al.*, “A review of digital microfluidics as portable platforms for lab-on a-chip applications,” *Lab Chip*, vol. 16, no. 13, pp. 2376–2396, 2016.

[36] M. H. Shamsi, K. Choi, A. H. C. Ng, and A. R. Wheeler, “A digital microfluidic electrochemical immunoassay,” *Lab Chip*, vol. 14, no. 3, pp. 547–54, 2014.

[37] A. H. C. Ng *et al.*, “Digital Micro fluidic Magnetic Separation for Particle-Based Immunoassays,” *Anal. Chem.*, vol. 84, pp. 8805–12, 2012.

[38] E. M. Miller, A. H. C. Ng, U. Uddayasankar, and A. R. Wheeler, “A digital microfluidic approach to heterogeneous immunoassays,” *Anal. Bioanal. Chem.*, vol. 399, no. 1, pp. 337–345, 2011.

[39] A. H. C. Ng, U. Uddayasankar, and A. R. Wheeler, “Immunoassays in microfluidic systems,” *Anal. Bioanal. Chem.*, vol. 397, no. 3, pp. 991–1007, 2010.

[40] Y.-J. Liu, D.-J. Yao, H.-C. Lin, W.-Y. Chang, and H.-Y. Chang, “DNA ligation of ultramicro volume using an EWOD microfluidic system with coplanar electrodes,” *J. Micromechanics Microengineering*, vol. 18, no. 4, p. 45017, 2008.

[41] Y.-H. Chang, G.-B. Lee, F.-C. Huang, Y.-Y. Chen, and J.-L. Lin, “Integrated polymerase chain reaction chips utilizing digital microfluidics.,” *Biomed. Microdevices*, vol. 8, no. 3, pp. 215–25, Sep. 2006.

[42] I. Barbulovic-Nad, H. Yang, P. S. Park, and A. R. Wheeler, “Digital microfluidics for cell-based assays,” *Lab Chip*, vol. 8, no. 4, pp. 519–26, 2008.

[43] I. A. Eydelnant *et al.*, “Virtual microwells for digital microfluidic reagent dispensing and cell culture,” *Lab Chip*, vol. 12, no. 4, pp. 750–757, 2012.

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

[45] M. J. Jebrail *et al.*, “A digital microfluidic method for dried blood spot analysis,” *Lab Chip*, vol. 11, no. 19, pp. 3218–3224, 2011.

[46] V. Srinivasan, V. K. Pamula, and R. B. Fair, “Droplet-based microfluidic lab-on-a-chip for glucose detection,” *Anal. Chim. Acta*, vol. 507, no. 1, pp. 145–150, 2004.

[47] V. K. Bhutani, M. Kaplan, B. Glader, M. Cotten, J. Kleinert, and V. Pamula, “Point-of-Care Quantitative Measure of Glucose-6-Phosphate Dehydrogenase Enzyme Deficiency.,” *Pediatrics*, vol. 136, no. 5, pp. 2015-2122, 2015.

[48] R. Sista *et al.*, “Development of a digital microfluidic platform for point of care testing,” *Lab Chip*, vol. 8, no. 12, p. 2091, 2008.

[49] S.-K. Fan, T.-H. Hsieh, and D.-Y. Lin, “General digital microfluidic platform manipulating dielectric and conductive droplets by dielectrophoresis and electrowetting.,” *Lab Chip*, vol. 9, no. 9, pp. 1236–1242, 2009.

[50] Z. Guttenberg *et al.*, “Planar chip device for PCR and hybridization with surface acoustic wave pump.,” *Lab Chip*, vol. 5, no. 3, pp. 308–17, 2005.

[51] A. A. García *et al.*, “Magnetic movement of biological fluid droplets,” *J. Magn. Magn. Mater.*, vol. 311, no. 1, pp. 238–243, 2007.

[52] E. Bormashenko, R. Pogreb, Y. Bormashenko, A. Musin, and T. Stein, “New investigations on ferrofluidics: Ferrofluidic marbles and magnetic-field-driven drops on superhydrophobic surfaces,” *Langmuir*, vol. 24, no. 21, pp. 12119–12122, 2008.

[53] V. Miralles, A. Huerre, H. Williams, B. Fournié, and C. Jullien, “A versatile technology for droplet-based microfluidics: thermomechanical actuation,” *Lab Chip*, vol. 15, no. 9, pp. 2133–2139, 2015.

[54] L. Malic *et al.*, “Integration and detection of biochemical assays in digital microfluidic LOC devices,” *Lab Chip*, vol. 10, no. 4, pp. 418–431, 2010.

[55] T.-Y. Ho, “Design automation for digital microfluidic biochips,” *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 7, pp. 16–26, 2014.

[56] Tsung-Yi Ho, K. Chakrabarty, and P. Pop, “Digital microfluidic biochips: Recent research and emerging challenges,” *Proc. 9th Int. Conf. Hardware/Software Codesign Syst. Synth. (CODES+ISSS)*, pp. 335–343, 2011.

[57] Fei Su, W. Hwang, and K. Chakrabarty, “Droplet Routing in the Synthesis of Digital Microfluidic Biochips,” *Proc. Des. Autom. Test Eur. Conf.*, vol. 1, pp. 1–6, 2006.

[58] K. F. Böhringer and S. Member, “Modeling and Controlling Parallel Tasks in Droplet-Based Microfluidic Systems,” *Proc. Computer-Aided Design*, 2006, vol. 25, no. 2, pp. 334–344.

[59] M. Cho and D. Z. Pan, “A High-Performance Droplet Routing Algorithm for Digital Microfluidic Biochips,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits a nd Systems*, 2008, vol. 27, no. 10, pp. 216–223.

[60] T. W. Huang and T. Y. Ho, “A fast routability-and performance-driven droplet routing algorithm for digital microfluidic biochips,” in *ICCD’09 Proceedings of the 2009 IEEE international conference on Computer design*, 2010, pp. 445–450.

[61] P. Roy, H. Rahaman, and P. Dasgupta, “A novel droplet routing algorithm for digital microfluidic biochips,” in *Proceedings of the 20th symposium on Great lakes symposium on VLSI - GLSVLSI ’10*, 2010, p. 441.

[62] S. Chakraborty, S. Chakraborty, C. Das, and P. Dasgupta, "Efficient two phase heuristic routing technique for digital microfluidic biochip," *IET Comput. Digit. Tech.*, vol. 10, no. 5, pp. 233–242, 2016.

[63] P. Yuh, S. Sapatnekar, C. Yang, and Y. Chang, "A Progressive-ILP Based Routing Algorithm for Cross-Referencing Biochips," *in DAC*, pp. 284–289, 2008.

[64] J. W. Chang, S. H. Yeh, T. W. Huang, and T. Y. Ho, "An ilp-based routing algorithm for pin-constrained EWOD chips with obstacle avoidance," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 32, no. 11, pp. 1655–1667, 2013.

[65] Y. Zhao and K. Chakrabarty, "Co-optimization of droplet routing and pin assignment in disposable digital microfluidic biochips," *Proc. 2011 Int. Symp. Phys. Des. - ISPD '11*, p. 69, 2011.

[66] O. Keszocze, R. Wille, T.-Y. Ho, and R. Drechsler, "Exact One-pass Synthesis of Digital Microfluidic Biochips," in *DAC*, 2014.

[67] P. H. Yuh, C. C. Y. Lin, T. W. Huang, T. Y. Ho, C. L. Yang, and Y. W. Chang, "A SAT-based routing algorithm for cross-referencing biochips," in *International Workshop on System Level Interconnect Prediction, SLIP*, 2011, pp. 1–7.

[68] R. Bhattacharya, H. Rahaman, and P. Roy, "A new heterogeneous droplet routing technique and its simulator to improve route performance in digital microfluidic biochips," in *International Conference on Microelectronics, Computing and Communication*, 2016.

[69] P. Roy, P. Howlada, R. Bhattacharjee, and H. Rahaman, “A new cross contamination aware routing method with intelligent path exploration in digital microfludics biochips,” *Proc. of IEEE DTIS*, 2013.

[70] Y. Zhao and K. Chakrabarty, “Cross-contamination Avoidance Technique for Droplet Routing in Digital Microfluidic Biochip,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 6, pp. 817–830, 2012.

[71] T.-W. Huang, C.-H. Lin, and T.-Y. Ho, “A Contamination Aware Droplet Routing Algorithm for Digital Microfluidic Biochips,” in *Proc. of the ICCAD*, 2009, pp. 151–156.

[72] T. Xu and K. Chakrabarty, “Broadcast electrode-addressing for pin-constrained multi-functional digital microfluidic biochips,” in *Proceedings of the 45th Design Automation Conference*, 2008, p. 991.

[73] T. Xu and K. Chakrabarty, “Droplet-trace-based array partitioning and a pin assignment algorithm for the automated design of digital microfluidic biochips,” *Proc. Codes+Isss*, pp. 112–117, 2006.

[74] T. Xu and K. Chakrabarty, “A cross-referencing-based droplet manipulation method for high-throughput and pin-constrained digital microfluidic arrays,” *Proc. -Design, Autom. Test Eur.*, pp. 552–557, 2007.

[75] T. Xu, S. Member, and K. Chakrabarty, “A Droplet-Manipulation Method for Achieving Digital Microfluidic Biochips,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 27, no. 11, pp. 1905–1917, 2008.

[76] Y. Luo and K. Chakrabarty, “Design of Pin-Constrained General-Purpose Digital Microfluidic Biochips,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2013, vol. 32, no. 9, pp. 1307–1320.

[77] Y. C. Lei, C. S. Hsu, J. D. Huang, and J. Y. Jou, “Chain-based pin count minimization for general-purpose digital microfluidic biochips,” *Proc. Asia South Pacific Des. Autom. Conf. ASP-DAC*, vol. 25–28–Janu, pp. 599–604, 2016.

[78] S. Chatterjee, H. Rahaman, and T. Samanta, “Multi-objective optimization algorithm for efficient pin-constrained droplet routing technique in digital microfluidic biochip,” *Proc. - Int. Symp. Qual. Electron. Des. ISQED*, pp. 252–257, 2013.

[79] R. Mukherjee, H. Rahaman, I. Banerjee, T. Samanta, and P. Dasgupta, “A heuristic method for co-optimization of pin assignment and droplet routing in digital microfluidic biochip,” *Proc. IEEE Int. Conf. VLSI Des.*, pp. 227–232, 2012.

[80] Y. Zhao and K. Chakrabarty, “Simultaneous optimization of droplet routing and control-pin mapping to electrodes in digital microfluidic biochips,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 31, no. 2, pp. 242–254, 2012.

[81] C. C. Y. Lin and Y. W. Chang, “ILP-based pin-count aware design methodology for microfluidic biochips,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 29, no. 9, pp. 1315–1327, 2010.

[82] S. K. Cho, H. Moon, and C.-J. J. Kim, “Creating, Transporting, Cutting, and Merging Liquid Droplets by Electrowetting-Based Actuation for Digital Microfluidic Circuits,” *J. Microelectromechanical Syst.*, vol. 12, no. 1, pp. 70–80, Feb. 2003.

[83] M. J. Schertzer, R. Ben-Mrad, and P. E. Sullivan, “Using capacitance measurements in EWOD devices to identify fluid composition and control droplet mixing,” *Sensors Actuators, B Chem.*, vol. 145, no. 1, pp. 340–347, 2010.

[84] E. Samiei and M. Hoorfar, “Systematic analysis of geometrical based unequal droplet splitting in digital microfluidics,” *J. Micromechanics Microengineering*, vol. 25, no. 5, p. 55008, 2015.

[85] J. Berthier, P. Dubois, P. Clementz, P. Claustre, C. Peponnet, and Y. Fouillet, “Actuation potentials and capillary forces in electrowetting based microsystems,” *Sensors Actuators, A Phys.*, vol. 134, no. 2, pp. 471–479, 2007.

[86] S. W. Walker, S. W. Walker, B. Shapiro, and B. Shapiro, “Modeling the fluid dynamics of electrowetting on dielectric(EWOD),” *J. microelectromechanical Syst.*, vol. 15, no. 4, pp. 986–1000, 2006.

[87] W. C. Nelson and C. C. J. Kim, “Journal of Adhesion Science and Droplet Actuation by (EWOD): A Review,” *J. Adhes. Sci. Technol.*, vol. 26, no. August 2012, pp. 1747–1771, 2012.

[88] P. Lalieux, “PIEZOELECTRICITY,” 2013.

[89] W. H. Liew, M. S. Mirshekarloo, S. Chen, K. Yao, and F. E. H. Tay, “Nanoconfinement induced crystal orientation and large piezoelectric coefficient in vertically aligned P(VDF-TrFE) nanotube array,” *Sci. Rep.*, vol. 5, p. 9790, 2015.

[90] N. G. Elvin and a. a. Elvin, “Large deflection effects in flexible energy harvesters,” *J. Intell. Mater. Syst. Struct.*, vol. 23, no. 13, pp. 1475–1484, 2012.

[91] K. Kanda, T. Saito, Y. Iga, K. Higuchi, and K. Maenaka, “Influence of parasitic capacitance on output voltage for series-connected thin-film piezoelectric devices,” *Sensors (Switzerland)*, vol. 12, no. 12, pp. 16673–16684, 2012.

[92] H. Chen, J. Cogswell, C. Anagnostopoulos, and M. Faghri, “A fluidic diode, valves, and a sequential-loading circuit fabricated on layered paper,” *Lab Chip*, vol. 12, no. 16, p. 2909, 2012.

[93] D. J. Im, M. M. Ahn, B. S. Yoo, D. Moon, D. W. Lee, and I. S. Kang, “Discrete electrostatic charge transfer by the electrophoresis of a charged droplet in a dielectric liquid,” *Langmuir*, vol. 28, no. 32, pp. 11656–11661, 2012.

[94] D. J. Im, J. Noh, D. Moon, and I. S. Kang, “Electrophoresis of a charged droplet in a dielectric liquid for droplet actuation,” *Anal. Chem.*, vol. 83, no. 13, pp. 5168–5174, 2011.

[95] M. Hase, S. N. Watanabe, and K. Yoshikawa, “Rhythmic motion of a droplet under a dc electric field,” *Phys. Rev. E*, vol. 74, no. 4, p. 46301, 2006.

[96] S. Mhatre and R. M. Thaokar, “Drop motion, deformation, and cyclic motion in a non-uniform electric field in the viscous limit,” *Phys. Fluids*, vol. 25, no. 7, p. 72105, 2013.

[97] A. M. Drews, M. Kowalik, and K. J. M. Bishop, “Charge and force on a conductive sphere between two parallel electrodes: A Stokesian dynamics approach,” *J. Appl. Phys.*, vol. 116, no. 7, p. 74903, 2014.

[98] C. P. Lee, H. C. Chang, and Z. H. Wei, “Charged droplet transportation under direct current electric fields as a cell carrier,” *Appl. Phys. Lett.*, vol. 101, no. 1, p. 14103, 2012.

[99] D. J. Im, B. S. Yoo, M. M. Ahn, D. Moon, and I. S. Kang, “Digital Electrophoresis of Charged Droplets,” *Anal. Chem.*, vol. 85, no. 8, pp. 4038–4044, 2013.

[100] M. Specialties, “Piezo Film Sensors Technical Manual,” *Measurement Specialties, Inc.*, 2006, online: https://www.sparkfun.com/datasheets/Sensors/Flex/MSI-techman.pdf.

[101] Hooker and M. W., Lockheed Martin Engineering and Sciences Co., 1998, “Properties of PZT-Based Piezoelectric Ceramics Between -150 and 250 C,” rechived from: https://ntrs.nasa.gov/search.jsp?R=19980236888.

[102] M. E. O’Neill and K. Stewartson, “On the slow motion of a sphere parallel to a nearby plane wall,” *J. Fluid Mech.*, vol. 27, no. 4, p. 705, 1967.

[103] A. J. Goldman, R. G. Cox, and H. Brenner, “Slow viscous motion of a sphere parallel to a plane wall—I Motion through a quiescent fluid,” *Chem. Eng. Sci.*, vol. 22, no. 4, pp. 637–651, 1967.

[104] M. Hase, S. N. Watanabe, and K. Yoshikawa, “Rhythmic motion of a droplet under a dc electric field,” *Phys. Rev. E*, vol. 74, no. 4, p. 46301, 2006.

[105] Y.-M. M. Jung, H.-C. C. Oh, and I. S. Kang, “Electrical charging of a conducting water droplet in a dielectric fluid on the electrode surface,” *J. Colloid Interface Sci.*, vol. 322, no. 2, pp. 617–623, 2008.

[106] T. WARD and G. M. HOMSY, “Chaotic streamlines in a translating drop with a uniform electric field,” *J. Fluid Mech.*, vol. 547, no. 1, p. 215, 2006.

[107] P.-H. Y. P.-H. Yuh, C.-L. Y. C.-L. Yang, and Y.-W. C. Y.-W. Chang, “BioRoute: a network-flow based routing algorithm for digital microfluidic biochips,” *2007 IEEE/ACM Int. Conf. Comput. Des.*, pp. 752–757, 2007.

[108] P. Roy, H. Rahaman, R. Bhattacharya, and P. Dasgupta, “A best path selection based parallel router for DMFBs,” *Proc. - 2011 Int. Symp. Electron. Syst. Des. ISED 2011*, pp. 176–181, 2011.

[109] P. Roy, H. Rahaman, and P. Dasgupta, “Two-level clustering-based techniques for intelligent droplet routing in digital microfluidic biochips,” *Integr. VLSI J.*, vol. 45, no. 3, pp. 316–330, 2012.

[110] O. Keszocze, R. Wille, K. Chakrabarty, and R. Drechsler, “A general and exact routing methodology for Digital Microfluidic Biochips,” *2015 IEEE/ACM Int. Conf. Comput. Des. ICCAD 2015*, pp. 874–881, 2016.

[111] T. W. Huang and T. Y. Ho, “A two-stage integer linear programming-based droplet routing algorithm for pin-constrained digital microfluidic biochips,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 30, no. 2, pp. 215–228, 2011.

[112] D. Silver, 2005, “Cooperative Pathfinding,”, In Young, R. M., and Laird, J. E., eds., AIIDE, 117–122. AAAI Press.

[113] K. C. Wang and A. Botea, “Fast and Memory-Efficient Multi-Agent Path finding,” *in Proc. ICAPS 2008*, pp. 380–387, 2008.

[114] T. Standley, “Finding Optimal Solutions to Cooperative Pathfinding Problems,” in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10),* 2010, pp. 173–178.

[115] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, “The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding,” *Artif. Intell.*, vol. 2, no. i, pp. 662–667, 2010.

[116] G. Sharon, R. Stern, A. Felner, and N. Sturtevant, “Conflict-Based Search For Optimal Multi-Agent Path Finding,” *Proc. Fifth Annu. Symp. Comb. Search*, pp. 563–569, 2012.

[117] E. Boyarski, A. Felner, G. Sharon, and R. Stern, “Don’t Split , Try To Work It Out : Bypassing Conflicts in Multi-Agent Pathfinding,” *Int. Conf. Autom. Plan. Sched.*, pp. 47–51, 2015.