

AMERICAN UNIVERSITY OF BEIRUT

ALGORITHMS FOR ATOM  
RECONFIGURATION IN QUANTUM  
SIMULATORS

by

REMY ALAA EL SABEH

A thesis

submitted in partial fulfillment of the requirements  
for the degree of Master of Science  
to the Department of Computer Science  
of the Faculty of Arts and Sciences  
at the American University of Beirut

Beirut, Lebanon  
August 2022

AMERICAN UNIVERSITY OF BEIRUT

ALGORITHMS FOR ATOM  
RECONFIGURATION IN QUANTUM  
SIMULATORS

by  
REMY ALAA EL SABEH

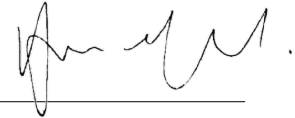
Approved by:

---

Dr. Amer E. Mouawad, Assistant Professor

Computer Science

Advisor

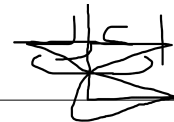


---

Dr. Izzat El Hajj, Assistant Professor

Computer Science

Member of Committee



---

Dr. Wassim El Hajj, Associate Professor

Computer Science

Member of Committee



Date of thesis defense: August 8, 2022

# AMERICAN UNIVERSITY OF BEIRUT

## THESIS RELEASE FORM

Student Name: El Sabeh Remy Alaa  
Last First Middle

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of my thesis; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes

- As of the date of submission of my thesis  
 After 1 year from the date of submission of my thesis .  
 After 2 years from the date of submission of my thesis .  
 After 3 years from the date of submission of my thesis .

Remy Sabeh 19/08/2022  
Signature Date

This form is dated and signed when asked to submit the final document to ScholarWorks.

# ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Professor Amer E. Mouawad for giving me the opportunity to work on this research project and for being involved in every step along the way. His resourcefulness, guidance, patience and support immensely aided not only in the realization of this thesis, but also in my development as a researcher.

I am also grateful for the involvement of Professor Izzat El Hajj and Professor Wassim El Hajj in this process, your input and your feedback have been invaluable.

Last but not least, I would like to thank my parents for being my pillar of support through my academic journey. I will be eternally grateful for all the sacrifices that you made.

# ABSTRACT OF THE THESIS OF

Remy Alaa El Sabeh for Master of Science  
Major: Computer Science

Title: Algorithms for Atom Reconfiguration in Quantum Simulators

The reconfiguration framework is an emerging framework that concerns finding a step-by-step transformation between two feasible solutions of a problem such that the transformation steps preserve a feasible solution. The complexity (parameterized or otherwise) of the reconfiguration counterpart of several classical NP-hard problems has been studied in the literature over the last few years. In this thesis, we study algorithms revolving around atom reconfiguration. Viewed as a problem on graphs, the atom reconfiguration problem is about finding the most efficient sequence of moves that allows us to go from one atom arrangement to another, atoms being marked vertices. In this context, a move is the movement of an atom through a sequence of edges from some source to some destination such that the selected atom does not collide with any other atom along the way. We define the efficiency of a reconfiguration sequence in terms of number of atom displacements, number of atom extractions/implantations, or a combination of both. We focus on the latter variant. This variant is hard because it is a general case of the atom reconfiguration problem restricted to extraction/implantation minimization, which is known to be NP-complete on general graphs and even when restricted to grid graphs. No matter the variant we are dealing with, our objective function is atom loss, which we aim to minimize. Atom loss is a function of two move operation types applied on atoms, extractions/implantations and displacements, among other variables. We empirically analyze multiple atom reconfiguration solvers, and we prove theorems that aid in orienting our algorithmic work and that may be of independent interest.

# TABLE OF CONTENTS

|                                                                                          |           |
|------------------------------------------------------------------------------------------|-----------|
| <b>ACKNOWLEDGEMENTS</b>                                                                  | <b>1</b>  |
| <b>ABSTRACT</b>                                                                          | <b>2</b>  |
| <b>1 Introduction</b>                                                                    | <b>10</b> |
| 1.1 Reconfiguration problems: an overview . . . . .                                      | 10        |
| 1.2 Problem definition . . . . .                                                         | 11        |
| 1.3 Goals and contributions . . . . .                                                    | 14        |
| 1.4 Thesis structure . . . . .                                                           | 18        |
| <b>2 Related Work</b>                                                                    | <b>19</b> |
| <b>3 Preliminaries</b>                                                                   | <b>21</b> |
| <b>4 Token Moving on Paths on the CPU</b>                                                | <b>24</b> |
| 4.1 The token moving procedure . . . . .                                                 | 25        |
| 4.2 The token-vertex matching procedure . . . . .                                        | 30        |
| 4.2.1 <i>The Hungarian-based token-vertex matcher</i> . . . . .                          | 31        |
| 4.2.2 <i>The Greedy token-vertex matcher</i> . . . . .                                   | 35        |
| 4.2.3 <i>The Linear Exact token-vertex matcher</i> . . . . .                             | 37        |
| 4.2.4 <i>The Bruteforce token-vertex matcher</i> . . . . .                               | 37        |
| <b>5 Token Moving on Grids on the CPU</b>                                                | <b>39</b> |
| 5.1 REDREC v2.0 . . . . .                                                                | 39        |
| 5.2 The Hungarian-based grid solvers . . . . .                                           | 44        |
| 5.2.1 <i>The modules</i> . . . . .                                                       | 47        |
| 5.2.1.1 All-pairs shortest path (APSP) . . . . .                                         | 47        |
| 5.2.1.2 Minimum weight perfect matching with path re-<br>retrieval (MWPM + PR) . . . . . | 47        |
| 5.2.1.3 Collision avoidance (COLAV) . . . . .                                            | 47        |
| 5.2.1.3.1 Bidirectional COLAV . . . . .                                                  | 48        |
| 5.2.1.3.2 Bounded COLAV . . . . .                                                        | 49        |
| 5.2.1.3.3 Derivation-averse COLAV . . . . .                                              | 51        |

|             |                                                               |            |
|-------------|---------------------------------------------------------------|------------|
| 5.2.1.4     | Ordering . . . . .                                            | 51         |
| 5.2.1.4.1   | Path merging . . . . .                                        | 51         |
| 5.2.1.4.2   | Path unwrapping . . . . .                                     | 52         |
| 5.2.1.4.3   | Cycle breaking . . . . .                                      | 52         |
| 5.2.1.4.3.1 | <i>Computing edge frequencies</i> . . .                       | 53         |
| 5.2.1.4.3.2 | <i>Cycle detection</i> . . . . .                              | 53         |
| 5.2.1.4.3.3 | <i>Cycle identification</i> . . . . .                         | 55         |
| 5.2.1.4.3.4 | <i>Cycle breaking</i> . . . . .                               | 56         |
| 5.2.1.4.3.5 | <i>Preserving the properties of the path system</i> . . . . . | 58         |
| 5.2.1.4.3.6 | <i>Cycle breaking: the conclusion</i> . .                     | 58         |
| 5.2.1.5     | Output generation . . . . .                                   | 59         |
| 5.2.1.5.1   | Exact extraction/implantation forest solver                   | 59         |
| 5.2.1.5.2   | Greedy solver . . . . .                                       | 59         |
| 5.2.1.6     | Greedy token isolation . . . . .                              | 61         |
| 5.2.2       | <b>HUNGARIAN-REDIST</b> . . . . .                             | 62         |
| 5.2.3       | <b>HUNGARIAN-COLAV</b> . . . . .                              | 63         |
| <b>6</b>    | <b>Token Moving on Paths on the GPU</b>                       | <b>64</b>  |
| 6.1         | The token moving procedure . . . . .                          | 64         |
| 6.2         | The token-vertex matching procedure . . . . .                 | 69         |
| 6.2.1       | <i>The Hungarian-based token-vertex matcher</i> . . . . .     | 69         |
| 6.2.2       | <i>The Greedy token-vertex matcher</i> . . . . .              | 72         |
| 6.2.3       | <i>The Linear Exact token-vertex matcher</i> . . . . .        | 72         |
| 6.2.4       | <i>The Bruteforce token-vertex matcher</i> . . . . .          | 74         |
| <b>7</b>    | <b>Token Moving on Grids on the GPU</b>                       | <b>76</b>  |
| 7.1         | REDREC v2.0 . . . . .                                         | 76         |
| 7.2         | REDREC v2.1 . . . . .                                         | 77         |
| <b>8</b>    | <b>Proofs</b>                                                 | <b>78</b>  |
| <b>9</b>    | <b>Experimental Setup and Experimental Results</b>            | <b>95</b>  |
| 9.1         | Experiments on path solvers on the CPU . . . . .              | 95         |
| 9.2         | Experiments on path solvers on the GPU . . . . .              | 98         |
| 9.3         | Experiments on grid solvers on the CPU . . . . .              | 101        |
| 9.4         | Experiments on grid solvers on the GPU . . . . .              | 106        |
| <b>10</b>   | <b>Conclusion</b>                                             | <b>109</b> |
| 10.1        | Future work . . . . .                                         | 109        |

|          |                                                                                       |            |
|----------|---------------------------------------------------------------------------------------|------------|
| <b>A</b> | <b>Figures</b>                                                                        | <b>111</b> |
| A.1      | Experiments on path solvers on the CPU . . . . .                                      | 111        |
| A.2      | Experiments on path solvers on the GPU . . . . .                                      | 112        |
| A.3      | Experiments on grid solvers on the CPU . . . . .                                      | 113        |
| A.3.1    | <i>Distribution of extraction/implantation operations<br/>across tokens</i> . . . . . | 113        |
| A.3.2    | <i>Distribution of displacement operations across tokens</i>                          | 114        |



# ILLUSTRATIONS

|     |                                                                                                                                                 |    |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1.1 | An atom array (left), a static trap array (middle) and a dynamic trap array (right) . . . . .                                                   | 11 |
| 1.2 | A visualization of the modules in the RTFC . . . . .                                                                                            | 13 |
| 4.1 | Degenerate instance where minimizing displacement incurs a large extraction/implantation overhead. . . . .                                      | 25 |
| 4.2 | Instance where greedy token-vertex matching does not minimize overall displacement . . . . .                                                    | 35 |
| 5.1 | Instance where overall displacement operations and overall extraction/implantation operations cannot be minimized at once. . . . .              | 45 |
| 5.2 | Relation between overall displacements and overall extractions/implantations in pareto-optimal solutions . . . . .                              | 46 |
| 5.3 | Example of a path system that induces a cycle that cannot be broken via computing a MST without increasing total path system distance . . . . . | 54 |
| 5.4 | Leveraging source-target corners to reduce the number of paths that induce the special cycle . . . . .                                          | 56 |
| 5.5 | Nontrivial cycle breaking base case: no source-target corners found                                                                             | 57 |
| 5.6 | Instance where a greedy solution extracts/implants a token more than once. . . . .                                                              | 59 |
| 5.7 | Instance where a greedy solution displaces a different number of tokens depending on the ordering of the paths. . . . .                         | 61 |
| 8.1 | Instance where a token-vertex matching with the minimum total distance is not executable. . . . .                                               | 79 |
| 8.2 | Instance where a token-vertex matching with no crossings is not executable. . . . .                                                             | 80 |
| 8.3 | Cases where the innermost path between $s_0$ and $s_2$ is between $end_{s_0}$ and a vertex of $s_2$ . . . . .                                   | 89 |
| 8.4 | Cases where the innermost path between $s_0$ and $s_2$ is between $start_{s_0}$ and a vertex of $s_2$ . . . . .                                 | 89 |

|      |                                                                                                                                                                                                                                                                                                    |     |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 9.1  | CPU running times for path solvers on centered targets with no surplus (batched token moving). . . . .                                                                                                                                                                                             | 96  |
| 9.2  | CPU running times for path solvers on centered targets with a surplus ratio of 0.1 (batched token moving). . . . .                                                                                                                                                                                 | 96  |
| 9.3  | CPU running times for path solvers on centered targets with no surplus (block batched token moving). . . . .                                                                                                                                                                                       | 97  |
| 9.4  | CPU running times for path solvers on centered targets with a surplus ratio of 0.1 (block batched token moving). . . . .                                                                                                                                                                           | 97  |
| 9.5  | GPU running times for path solvers on centered targets with no surplus (batched token moving). . . . .                                                                                                                                                                                             | 99  |
| 9.6  | GPU running times for path solvers on centered targets with a surplus ratio of 0.1 (batched token moving). . . . .                                                                                                                                                                                 | 99  |
| 9.7  | GPU running times for path solvers on centered targets with no surplus (block batched token moving). . . . .                                                                                                                                                                                       | 100 |
| 9.8  | GPU running times for path solvers on centered targets with a surplus ratio of 0.1 (block batched token moving). . . . .                                                                                                                                                                           | 100 |
| 9.9  | Average GPU running time per block batch for path solvers on centered targets with no surplus (left) and with a surplus ratio of 0.1 (right). . . . .                                                                                                                                              | 102 |
| 9.10 | Mean Success Probability of token moving on grids of size $64 \times 32$ and targets of size $32 \times 32$ ( $p_{load} = 0.6$ ) with greedy token isolation disabled for the different grid solvers, averaged over 500 instances.                                                                 | 103 |
| 9.11 | Operations executed in token moving on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) with greedy token isolation disabled for the different grid solvers, averaged over 500 instances.                                                                              | 104 |
| 9.12 | Distribution of extraction/implantation operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) with greedy token isolation disabled for greedy HUNGARIAN-NOCOLAV (left) and greedy HUNGARIAN-COLAV (right), averaged over 500 instances. . . . . | 105 |
| 9.13 | Distribution of extraction/implantation operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) with greedy token isolation disabled for HUNGARIAN-NOCOLAV (left) and HUNGARIAN-COLAV (right), averaged over 500 instances. . . . .               | 105 |
| 9.14 | Distribution of extraction/implantation operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for the 3-approximation algorithm for extraction/implantation minimization, averaged over 500 instances. . . . .                                  | 106 |
| A.1  | CPU running times for path solvers on centered targets with no surplus (unbatched token moving). . . . .                                                                                                                                                                                           | 111 |

|      |                                                                                                                                                                                                                                                                                         |     |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| A.2  | CPU running times for path solvers on centered targets with a surplus ratio of 0.1 (unbatched token moving). . . . .                                                                                                                                                                    | 112 |
| A.3  | GPU running times for path solvers on centered targets with no surplus (unbatched token moving). . . . .                                                                                                                                                                                | 112 |
| A.4  | GPU running times for path solvers on centered targets with a surplus ratio of 0.1 (unbatched token moving). . . . .                                                                                                                                                                    | 113 |
| A.5  | Distribution of extraction/implantation operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for REDREC v1, averaged over 500 instances. . . . .                                                                                    | 113 |
| A.6  | Distribution of extraction/implantation operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for REDREC v2.0, averaged over 500 instances. . . . .                                                                                  | 114 |
| A.7  | Distribution of extraction/implantation operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for REDREC v2.1, averaged over 500 instances. . . . .                                                                                  | 114 |
| A.8  | Distribution of displacement operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) with greedy token isolation disabled for greedy HUNGARIAN-NOCOLAV (left) and greedy HUNGARIAN-COLAV (right), averaged over 500 instances. . . . . | 114 |
| A.9  | Distribution of displacement operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) with greedy token isolation disabled for HUNGARIAN-NOCOLAV (left) and HUNGARIAN-COLAV (right), averaged over 500 instances. . . . .               | 115 |
| A.10 | Distribution of displacement operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for the 3-approximation algorithm for extraction/implantation minimization, averaged over 500 instances. . . . .                                  | 115 |
| A.11 | Distribution of displacement operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for REDREC v1.0, averaged over 500 instances. . . . .                                                                                             | 115 |
| A.12 | Distribution of displacement operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for REDREC v2.0, averaged over 500 instances. . . . .                                                                                             | 116 |
| A.13 | Distribution of displacement operations across tokens on grids of size $64 \times 32$ and targets of size $32 \times 32$ (no surplus) for REDREC v2.1, averaged over 500 instances. . . . .                                                                                             | 116 |

# TABLES

|     |                                                                                                                |     |
|-----|----------------------------------------------------------------------------------------------------------------|-----|
| 9.1 | REDREC v2.0 execution time on the GPU in milliseconds, averaged over 100 instances . . . . .                   | 107 |
| 9.2 | REDREC v2.0 average execution time per batch on the GPU in milliseconds, averaged over 100 instances . . . . . | 108 |

# CHAPTER 1

## INTRODUCTION

### 1.1 Reconfiguration problems: an overview

Reconfiguration problems are a class of problems where we define feasibility conditions, two feasible solutions, and a set of possible transformations: those transformations define adjacency relations between feasible solutions, which we call configurations. The question of interest is whether we can apply a sequence of transformations that get us from the source feasible solution to the target feasible solution, such that all intermediary configurations are feasible. The concept of reconfiguration can be found in a plethora of puzzles, such as the 15-puzzle game (with research dating back to 1879 [1]) and Rubik's cubes. While a variety of problems lend themselves to being treated as reconfiguration problems, it was not until recently that the reconfiguration framework was formally defined [2]. We present standard terminology used in reconfiguration problems. When we are dealing with a reconfiguration problem, the solution space is defined as the set of all possible configurations, or feasible solutions that can be reached from the source feasible solution using the transformations at our disposition. The solution space can be modeled by what we call a reconfiguration graph that encompasses the different configurations. A configuration is said to be adjacent to another configuration if there exists a single step transformation from one to the other: the vertices representing those configurations are adjacent in the reconfiguration graph. Given a reconfiguration graph, we have a yes-instance for the reconfiguration problem if and only if the vertex corresponding to the target configuration is reachable from the vertex corresponding to the source configuration. Since the reachability problem can be solved in polynomial time in terms of the size of the input and knowing that adjacency can be checked in polynomial time, it should be easy to see that the exponential size of the reconfiguration graph is at the root of the hardness of some reconfiguration problems.

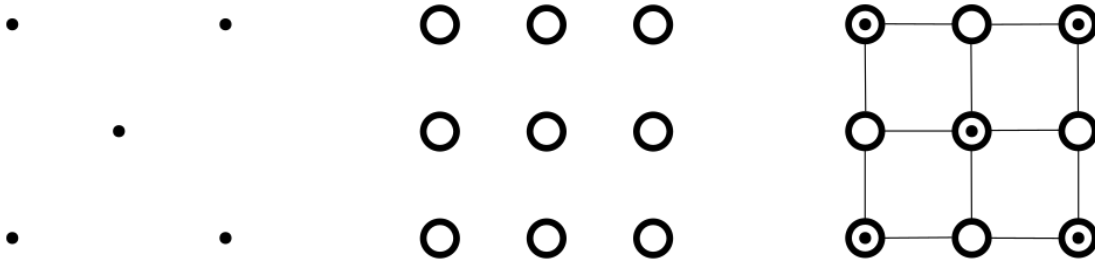


Figure 1.1: An atom array (left), a static trap array (middle) and a dynamic trap array (right)

## 1.2 Problem definition

Controllable quantum many-body systems are at the heart of quantum information processing. Such systems can be leveraged in quantum simulations to solve problems that are intractable in the classical sense [3], and they can also be employed in quantum computing in the implementation of quantum algorithms that provide more efficient solutions to classical problems [4, 5]. The quantum many-body systems that we are interested in consist of ultracold atoms in arrays of optical traps [6]. The optical traps are generated using focused light beams and make it possible to manipulate the atomic particles without coming in contact with them by exploiting their polarizability, as the beams can be used to generate electromagnetic fields. The combination of optical beams and particles has made it possible to deterministically assemble atomic systems without having to rely on interactions between the particles. The system is made up of three parts: the atom array, the static trap array and a generator vector, which defines the permissible movements of atoms (Figure 1.1). The combination of the three parts forms the dynamic trap array. Formally,  $Q$  is the set of traps, and the generator vector defines a set  $R$  of tuples  $(q_i, q_j)$  ( $q_i \in Q, q_j \in Q$ ) which we call permissible atom movements: if  $(q_i, q_j) \in R$ , then an atom occupying  $q_i$  can be moved to  $q_j$ , or vice versa, provided that  $q_j$  is vacant. A trap is said to be occupied if it is endowed with an atom, and a trap is said to be vacant if it is not occupied.

In this thesis, we focus on the design and implementation of atom assembly algorithms. We start by describing the most general formulation of the assembly process. In the preparation of the system,  $K$  atoms are randomly and uniformly dispersed over the traps: the occupied traps,  $S$ , constitute the initial state. By abuse of notation, we refer to a state, or a configuration, by its set of occupied traps. The operator of the system then specifies a subset of the traps,  $T$ , consisting of  $K' \leq K$  traps, which constitutes the desired state. We say that we have a surplus of  $K - K'$  atoms. Starting from the initial state, assembling the atoms entails generating a sequence of permissible atom movements. We execute the sequence of permissible atom movement in order on the initial state. An execution of a permissible atom movement  $(tr_i, tr_j)$  is its application on a given

state  $S_k$ , which yields a state  $S_l$  such that  $S_k \Delta S_l = \{tr_i, tr_j\}$ ; we say that the permissible atom movement transforms  $S_k$  into  $S_l$ . Note that the result of the application of a permissible atom movement  $(tr_i, tr_j)$  is defined if and only if  $tr_i$  is occupied and  $tr_j$  is empty in the state  $S_k$  it is being applied on. An assembly is said to be successful if the execution of the generated sequence of permissible atom movements transforms  $S$  into  $T'$ , where  $T' \supseteq T$ .

In reality, atom assembly involves a process of atom corruption, leading to loss. The execution of permissible atom movements corrupts atoms. The degree of corruption of an atom is proportional to the number of permissible atom movements involving it in the atom assembly, among other factors. The generation of a solution for the atom assembly will therefore also have to account for atom loss. We do not yet have the tools to describe what atom loss is, so we describe the pipeline that we make use of in our experiment next.

The atom assembly algorithms that we implement are integrated in a real-time feedback control system (RTFC) that aims to achieve the desired state for the atoms using a low-latency feedback loop [7]. We describe the hardware components of the RTFC.

The feedback control system consists of a frame grabber card (FGC), a graphics processing unit (GPU), and an arbitrary waveform generator (AWG), all of which are mounted on a central processing unit (CPU). The control protocol underlying the assembly process consists of five interdependent modules:

1. The **image acquisition module** gradually receives raw data from the EM-CCD camera into the buffer of the FGC. As soon as the data transfer to the FGC buffer terminates, the FGC transfers the data to a memory location accessible by the second module.
2. The **image processing and data analysis module** reads in the raw data from the camera as input and outputs a bit vector representing the state of the system. That is, this module detects the presence of atoms in traps, and it does so using a convolution, a commonly used technique in image processing, with a predefined threshold.
3. The **problem solving module** takes in the current state of the system and the desired state of the system as input and outputs a sequence of control instructions that assemble the atoms into the desired state.
4. The **waveform synthesis module** translates control instructions into control operations using a prepopulated table mapping control instructions to elementary waveforms.
5. The **waveform streaming module** executes the generated control operations.

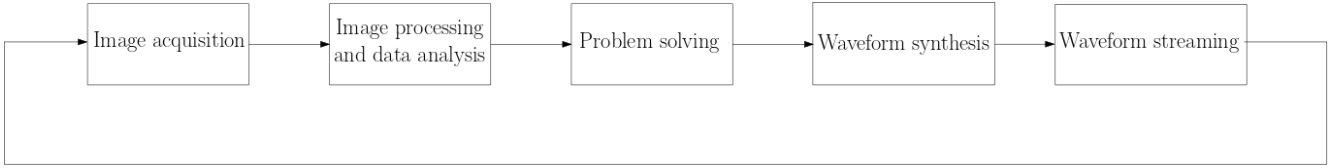


Figure 1.2: A visualization of the modules in the RTFC

The two modules of interest to us are the problem solving module and the waveform synthesis module. The problem solving module encapsulates the atom assemblers we designed, whereas the waveform synthesis module dictates part of the implementation of the algorithms, as the output of the problem solving module has to match the format of the expected input of the waveform synthesis module.

We now elucidate the functioning of the RTFC described above, specifically with regards to atom loss. A run of the RTFC is the execution of the five modules above in the order that they appear in. Atom loss is realized when the waveform streaming module terminates. When an atom is lost, it is removed from the set of atoms. The waveform streaming module transforms  $S$  into  $T'$ , as dictated by the problem solving module and the waveform synthesis module. Subsequently, the atoms in a set of traps  $T'' \subseteq T'$  are lost due to corruption. At this point, we are in one of three possible cases, the first two of which halt the experiment:

- $T' \setminus T'' \supseteq T$ : this indicates a successful atom assembly.
- $|T' \setminus T''| < |T|$ : in this case, the number of atoms lost exceeds initial atom surplus: atom assembly fails.
- Otherwise,  $|T' \setminus T''| \geq |T|$  and  $T' \setminus T'' \not\supseteq T$ . While atom assembly was not successful, it is still possible to assemble the atom array starting from this state.

If we are in the third case, we proceed to another run of the RTFC, and we specify  $T' \setminus T''$  to be the initial state, and  $T$  to be the final state. The process keeps getting repeated until termination (i.e. until one of the first two cases happens). This process is guaranteed to terminate ( Lemma 1).

In preparation for a formal definition of the problem, we describe the operations on atoms in more detail. So far, we have been using the notion of permissible atom movements. In practice, the movement of atoms is determined by a combination of three control operations: displacements, extractions, and implantations. Moving an atom from the trap  $tr_i$  it occupies to a vacant trap  $tr_j$  requires us to extract it from the optical trap it is lodged in, displacing it using a sequence of permissible atom movements  $(tr_1, tr_2), (tr_2, tr_3), \dots, (tr_{j-1}, tr_j)$ , and implanting it in a vacant optical trap. This is the broadest description of an atom move: an



atom move may require multiple extractions/implantations, and whether this is the case is dependent on the specifics of the hardware. Unless stated otherwise, we will be using this definition. As far as loss probability is concerned, it is established that extractions and implantations are equally inductive to loss, and are more inductive to loss than displacement operations. Loss probability can be calculated per particle and is a function of the number of operations executed on the particle, a parameter for displacement operations, a parameter for extraction/implantation operations (those parameters need not be the same for all atoms), as well as the time elapsed since the start of the experiment.

Given the definition of atom assembly with loss and the definition of loss as a function of the primitive atom operations, namely displacements, extractions and implantations, we focus on maximizing atom assembly success probability, where success is defined in relation to the proposed RTFC. Based on what has been mentioned so far, increasing success probability can be done in one of two ways: we can either increase the initial number of atoms in the system while curbing the repercussions this may have on the system's stability, or we can look at smarter ways to assemble the array of optical traps. In this thesis, we do the latter.

### 1.3 Goals and contributions

We formulate the problem of generating a sequence of control instructions that maximizes success probability as a graph theoretic problem (and, more specifically, a graph reconfiguration problem) as the problem defined in quantum terms presents all the characteristics of a graph reconfiguration problem. The atom array, the static array and the generator vector define tokens (a staple of problems in the reconfiguration framework), vertices and edges respectively.

MAXIMUM SUCCESS TOKEN MOVING (MSTM)

**Input:** A graph  $G$ , a set of vertices  $S$  and  $T$  such that there are tokens on the vertices in  $S$  ( $|S| \geq |T|$ ), a per-token loss probability associated with moving a token, and a per-token loss probability associated with sliding the token across an edge.

**Output:** A sequence of token moves that covers a set of vertices  $T' \supseteq T$  with the maximum success probability.

Token loss and token moving success are defined analogously to atom loss and atom assembly success respectively, in that they are artifacts of the RTFC in which the solvers are integrated. Also, we define operations on tokens analogously to control operations on atoms. That is, a token move consists of a token extraction, a sequence of token displacements, and a token implantation. Furthermore, the output should optimize for the expected success probability, and

a solution for the problem pertains to a single run. We therefore solve the same problem in every run.

Given the stochastic nature of token loss, and the interaction between movement decisions made during a run and their consequence in terms of token loss at the end of the run, the only means to assess the efficacy of a given algorithm in terms of success probability we currently have are simulations. We define more primitive problems that we have the tools to solve and that will constitute the building blocks of heuristic algorithms for MSTM. The problems and algorithms we define provide local guarantees. That is, they ignore the feedback loop they are integrated in. Therefore, the notion of loss will not figure in the description and design of the algorithms, and discussions involving loss will be limited to the experimental section, where we analyze the operational performance of the algorithms in terms of success probability, among other metrics, within the RTFC. However, we maintain the notion of token surplus ( $|S| \geq |T|$ ), and all our algorithms work with surplus.

We restrict ourselves to two graph classes: path graphs and grid graphs. While there may be some generalizations to be made in some of our work, specifically when it comes to theory, the reason why those graph classes are peculiar is because they are representative of predominant optical trap geometries, namely chain optical trap geometries and grid optical trap geometries. On path graphs, we design exact algorithms that solves the following problem:

MINIMUM DISPLACEMENT TOKEN MOVING (MDTM)

**Input:** A graph  $G$ , a set of vertices  $S$  and  $T$  such that there are tokens on the vertices in  $S$  ( $|S| \geq |T|$ ).

**Output:** A sequence of token moves that covers a set of vertices  $T' \supseteq T$  with the minimum number of displacements.

We propose three algorithms (henceforth interchangeably called solvers), the first two of which are exact, that solve the problem:

1. The **Hungarian-based path solver**, which relies on a modification of an algorithm that solves the assignment problem, the Hungarian method [8].
2. The **Linear Exact path solver**, which relies on an algorithm that solves a variant of the assignment problem, the linear assignment problem, conceived by Karp [9].
3. The **Greedy path solver**, which relies on a greedy algorithm for the assignment problem that solves the problem more efficiently than the Hungarian-based path solver at the cost of exactness.

We also design an algorithm, the **Bruteforce path solver**, that solves a variant of MDTM on paths where  $T$  induces a path.

Keeping in mind that our solvers are part of a low-latency feedback loop that may require the problem to be solved on the CPU or on the GPU, pending experimental results, we design and implement algorithms for the four solvers above on both the CPU and the GPU. The algorithms on the CPU are implemented in C, whereas the algorithms on the GPU are implemented in CUDA, a general purpose parallel computing platform [10].

To motivate our grid algorithms, we now go back to the MSTM problem. We define one correlated problem:

MINIMUM EXTRACTION/IMPLANTATION TOKEN MOVING (MEITM)

**Input:** A graph  $G$ , a set of vertices  $S$  and  $T$  such that there are tokens on the vertices in  $S$  ( $|S| \geq |T|$ ).

**Output:** A sequence of token moves that covers a set of vertices  $T' \supseteq T$  with tokens with the minimum number of token extractions/implantations.

MDTM admits a polynomial-time solution on general graphs, while MEITM (even with no surplus) is NP-hard and even APX-hard on general graphs, and remains NP-hard when restricted to grid graphs, though it does admit a 3-approximation algorithm for general graphs, and therefore for grid graphs as well [11]. The reason why we introduce MEITM and, previously, MDTM, is because those two problems form the pillars of the heuristics that our algorithm design relies on.

We now define the problem that we aim to solve on grid graphs:

MINIMUM LOSS TOKEN MOVING (MLTM)

**Input:** A graph  $G$ , a set of vertices  $S$  and  $T$  such that there are tokens on the vertices in  $S$  ( $|S| \geq |T|$ ), a loss function, a per-token loss parameter associated with moving the token, and a per-token loss parameter associated with sliding the token across an edge.

**Output:** A sequence of token moves that covers a set of vertices  $T' \supseteq T$  with the minimum loss.

The loss function is an operator-defined function that is meant to be a local measure (i.e. a measure within a RTFC run) of token moving success. As such, reducing the MSTM problem to the MLTM problem is heuristic in itself. The loss function is a function that aggregates per-token loss functions, which means that it is a function of per-token displacement operations and per-token extraction-implantation operations. Clearly, MLTM is at least as hard as MEITM, as MEITM is a special case of MLTM. We resort to heuristics to solve the problem.

The heuristics we design serve two purposes. On the one hand, we should be able to express how displacement-averse or how extraction/implantation-averse the solver should be (that is, how willing we are to incur extra overall displacements for the sake of reducing overall extraction/implantation operations, or vice versa). On the other hand, we should be able to indicate whether we prefer spreading operations across a large set of tokens versus restricting them to a small set of tokens. Preferences are tied to the loss function, and given that the loss function is part of the input, we make sure our algorithms are able to express preferences.

For grid graphs, we propose four algorithms. Two of our algorithms build on an unpublished grid solver, REDREC (short for redistribution-reconfiguration) [12] and the two others are parameterized in a way that makes it possible for them to express the aforementioned considerations in terms of overall operations and dispersion of operations. All those algorithms were designed with the aim of ameliorating the current results for MSTM on grid graphs. We briefly describe those algorithms below:

1. **REDREC v2.0**: REDREC v2.0 is a modification of the original REDREC algorithm that is more modular and uses path solvers as subroutines.
2. **REDREC v2.1**: REDREC v2.1 is a variant of REDREC v2.0 whose purpose is to offer speedups in terms of running time on the GPU, speedups that are made possible through decreasing the inherent sequentiality of REDREC v2.0.
3. **HUNGARIAN-REDIST**: HUNGARIAN-REDIST is a grid solver inspired by the procedures in REDREC that uses the Hungarian method as a subroutine and minimizes overall displacement operations, but has no considerations when it comes to overall extraction/implantation operations. Given that displacement minimization with surplus admits a polynomial-time solution, HUNGARIAN-REDIST is meant to serve as a baseline for atom assembly success probability.
4. **HUNGARIAN-COLAV**: HUNGARIAN-COLAV is an extraction/implantation aware grid solver. While HUNGARIAN-REDIST outputs a solution that minimizes overall displacement, HUNGARIAN-COLAV attempts to mitigate overall extractions/implantations, even if this comes at the cost of extra displacements. The algorithm’s parameters allow us to specify the extent to which we are willing to deviate from overall displacement minimization for the sake of avoiding extra extraction/implantation operations.

The **REDREC** algorithms assume that  $T$  is centered in the grid, whereas the Hungarian-based algorithms do not.  $T$  is said to be centered if it induces a

grid graph on  $G$  and if  $V_G \setminus T$  induce two disjoint grid graphs on  $G$ . Given that those algorithms are heuristic, our evaluation will involve software simulations of the RTFC on those four algorithms, the main metric of interest being mean atom assembly success probability across multiple instances. The four grid solvers are implemented serially in C, and the REDREC-based solvers are implemented in CUDA [10] as well in order to evaluate the feasibility (in terms of running time) of running such algorithms as part of the RTFC.

## 1.4 Thesis structure

In the first stage, we will be surveying related research in the literature. Section 2 will go over related works in theoretical computer science and in quantum physics, so as to give the required context for what follows. After we cover definitions, we proceed to detail the algorithms we used for the purpose of solving the different variants of the token moving problem we alluded to in the introduction. Sections 4 and 5 cover implementation details and pseudocode for path solvers and grid solvers respectively on the CPU, whereas Sections 6 and 7 cover implementation details and pseudocode of the parallel counterparts of the serial solvers covered in the previous two sections. In most cases, there are a set of statements that underlie the correctness of our algorithms: for the sake of readability, we separate proofs from implementation and algorithm design details; proofs for lemmas and theorems that we made use of in our algorithms can be found in Section 8. Eventually, while it may be the case that our work gives rise to related problems that are purely of theoretical interest, our short-term goal involves running the atom assembly pipeline on a physical array of optical traps. To that end, we turn to empirical data. The purpose of this empirical data is twofold: on the one hand, we would like to assess the feasibility of integrating our solvers into the pipeline, and on the other hand, we would like to benchmark how well our solvers perform in terms of mean atom assembly success probability. Section 9 compiles data concerning the solvers that we have implemented. In the conclusion (Section 10), we highlight possible improvements, algorithmic or otherwise, that will be tackled in our future work.

# CHAPTER 2

## RELATED WORK

The problem we handle in this thesis possesses multiple facets. We start by reviewing related works in theoretical computer science.

The MSTM problem falls under the framework of reconfiguration problems. The term “reconfiguration” was coined by Ito et al [2]. Ito et al. prove the PSPACE-completeness for reconfiguration versions of the most common graph problems, namely Independent Set Reconfiguration, Clique Reconfiguration and Vertex Cover Reconfiguration. Given that the idea of reconfiguration naturally occurs within games, it is unsurprising that the idea of reconfiguration found its way into them, particularly through Demaine’s nondeterministic constraint logic (NCL) model of computation [13], which predates the reconfiguration framework: a reduction from NCL was used to prove the PSPACE-completeness of sliding-block puzzles and Rush Hour. The expressiveness of reconfiguration in such games contributed to the introduction of reconfiguration in “game-like” problems as a tool to prove hardness, and that explains why the atom problem we are interested in is presented as a reconfiguration problem. There are several published results concerning token moving that are theoretical in nature. Călinescu et al. proved that MEITM with no surplus is both NP-complete and APX-hard on general graphs and is solvable in polynomial time on trees [11], where APX is the class of optimization problems for which we cannot hope to achieve a  $(1 + \epsilon)$ -approximation in polynomial time for every  $\epsilon > 0$ . Cooper et al. extended the hardness analysis of that same variant to include parameterized complexity [14]: MEITM admits an FPT algorithm when parameterized by the number of atoms  $k$ , the number of moves  $l$ , or the sum of both of those parameters, where FPT is the complexity class that contains the fixed parameter tractable problems, i.e. the problems that can be solved in time  $f(l) \cdot |x|^{O(1)}$  (where  $|x|$  is the input size) for some computable function  $f$  and some parameter  $l$ .

On the quantum end of things, we are interested in a class of quantum simulators that involve cold atoms in arrays of optical tweezers [15, 16]. The possibility of accelerating particles by radiation pressure from a continuous laser was first documented by Ashkin, a physicist, in 1970 [17]. Trapping came as a result of

further developments by Ashkin et al., and optical trapping was demonstrated for the first time in 1986 on particles ranging in size between 25 nm and 10  $\mu\text{m}$ , albeit with limited particle lifetimes [18]. As was predicted in this work, optical tweezers proved to be usable for trapping atoms and biological particles. In the same year, atomic optical trapping was demonstrated for the first time by Chu et al. [19], Chu being one of the collaborators in the previous work. In biology, optical tweezers make it possible to manipulate viruses and bacteria without causing any damage to the particles, and this was demonstrated by Ashkin et al. [20] via an experiment on tobacco mosaic virus particles suspended in a water chamber. The usage of optical tweezers for quantum simulation is recent. There is extensive work that has been done in the literature pertaining to improving the assembly process, particularly on lattice arrays, or grid-shaped arrays of atoms: Schymik et al.'s work focuses on minimizing assembly time through minimizing the number of moves [21], as assembly time is correlated to the number of moves, where a move ignores the notion of extraction and implantation. Ebadi et al. demonstrated a programmable quantum simulator based on deterministically prepared two-dimensional arrays of up to 256 neutral atoms [16], and the atom rearrangement algorithm proposed in the paper was further improved by REDREC [12].

In our work, we also use GPUs to accelerate our proposed reconfiguration algorithms. GPUs have been used to accelerate a wide variety of graph algorithms [22, 23, 24, 25, 26, 27]. To the best of our knowledge, our work is the first to use GPUs for accelerating graph reconfiguration.

# CHAPTER 3

## PRELIMINARIES

The general graph notation we use in this thesis is the standard notation found in graph theory books [28].

A grid graph is a graph whose embedding in  $\mathbb{R}^2$  forms a regular tiling, where each tile is a square delimited by 4 vertices and 4 edges. Initially, tokens occupy a subset  $S \subseteq V$  of the vertices of the graph, and in token moving, we aim to move the tokens along edges of the graph, so as to have them occupy a superset of  $T \subseteq V$ : we call  $T$  the target region. In reconfiguration terms, a configuration is identifiable by the occupied vertices of the graph; a vertex is said to be occupied if it is endowed with a token, and it is said to be vacant otherwise. The adjacency relation between configurations is dependent on the output mode used. A vertex in  $S$  is a source vertex and a vertex in  $T$  is a target vertex.

In both path solvers and grid solvers, we make use of the planarity of the graphs to simplify notation. Both path graphs and grid graphs can be embedded in  $\mathbb{R}^2$ . We embed a path  $P_k$  along the x-axis of the Cartesian plane, starting at  $(0, 0)$ . That is, the path's vertices are at coordinates  $(0, 0), (1, 0), \dots, (k - 1, 0)$ . We make use of the coordinates to reference the vertices of the path: vertex  $v_i$  ( $0 \leq i < k$ ) is the vertex at  $(i, 0)$ , and we call  $i$  the vertex's index. As for grid graphs, we embed a grid in the first quadrant of the Cartesian plane along the x-axis and the y-axis. That is, a grid graph will have a vertex at  $(0, 0)$  whose neighbors are at  $(0, 1)$  and  $(1, 0)$ . The height of a grid graph  $G$ , which we call  $H_G$ , is 1 plus the largest y-coordinate among its vertices, whereas the width of a grid graph  $G$ , which we call  $W_G$ , is 1 plus the largest x-coordinate among its vertices. The subscripts are dropped if the graph the dimensions refer to is clear from the context. As for indexing, vertex  $v_{i,j}$  ( $0 \leq i < G_W, 0 \leq j < G_H$ ) is at coordinates  $(i, j)$ . A displacement  $(v_i, v_j)$  (resp.  $(v_{i,j}, v_{k,l})$ ) is said to be permissible on a path (resp. on a grid) if  $|i - j| = 1$  (resp.  $|i - k| = 1$  and  $j = l$ , or  $|j - l| = 1$  and  $i = k$ ) and if it is applied on a configuration where  $v_i$  (resp.  $v_{i,j}$ ) is occupied and  $v_j$  (resp.  $v_{k,l}$ ) is vacant.

In path solvers, we define entities related to token-vertex matching.  $TK = \{t_0, t_1, \dots, t_{k-1}\}$  is the set of tokens on the vertices of the path. A token-vertex



matching is a one-to-one function  $M : T \mapsto TK$ . We say that the cardinality of  $M$  is  $|T|$ . The tokens that are part of the token-vertex matching are  $M(T)$ .  $M^{-1}(t_i)$  is called the target vertex of token  $t_i$  (defined if and only if  $t_i \in M(T)$ ), and  $M(v_j)$  is the token matched or assigned to  $v_j$ . We use the embedding to introduce the notion of “left” and “right”.  $v_i$  is to the left (resp. right) of  $v_j$  if and only if  $i < j$  (resp.  $i > j$ ), and  $v_i$  is to the immediate left (resp. immediate right) of  $v_j$  if and only if  $i + 1 = j$  (resp.  $i - 1 = j$ ). We define order on tokens similarly. We also use the embedding to define distances.  $d(a, b)$  is the absolute value of the difference in coordinates between two vertices or tokens  $a$  and  $b$  (the coordinate of a token being the coordinate of the vertex it occupies). We also use the same notation for token-vertex matchings:  $d(M)$  is the total distance of a matching  $M$ , and it is equal to  $\sum_{v \in T} d(v, M(v))$ . A token-vertex matching is executable if output generators are able to get the tokens in the matching to their designated target vertices using a sequence of control instructions. We use the notation  $t_i$  to refer to token  $i$ , where token  $i$  has  $i$  occupied vertices to its left. For all  $t_i \in M(T)$ , we use the notation  $v_{t_i}$  to refer to the vertex that contains the token  $t_i$  in a specific configuration. Since  $t_i$  and  $v_{t_i}$  have no notion of coordinates, we define the function  $idx(\cdot)$  that takes in a token (resp. a vertex) and returns the index of the vertex it occupies (resp. its index). If  $t_j \in M(T)$ , we say that  $t_i$  obstructs  $t_j$  if  $idx(M^{-1}(t_j)) \leq idx(t_i) < idx(t_j)$  or, symmetrically, if  $idx(M^{-1}(t_j)) \geq idx(t_i) > idx(t_j)$ . A token-vertex matching is said to have a crossing if there exists two tokens  $t_i, t_j$  in the matching such that  $idx(t_i) < idx(t_j)$  and  $idx(M^{-1}(t_i)) > idx(M^{-1}(t_j))$ . In a given configuration, a token  $t_i \in M(T)$  is said to be left-moving (resp. right-moving) if  $idx(t_i) > idx(M^{-1}(t_i))$  (resp.  $idx(t_i) < idx(M^{-1}(t_i))$ ). A token  $t_i$  that is not in  $M(T)$  or that is in  $M(T)$  with  $idx(t_i) = idx(M^{-1}(t_j))$  is non-moving. We also use the term “direction” to characterize whether a token is left-moving, right-moving or non-moving. The direction of a token is bound to a specific configuration. A token block is a set of tokens  $t_i, t_{i+1}, \dots, t_{i+k}$  that have the same direction such that  $idx(v_{t_{i+j}}) + 1 = idx(v_{t_{i+j+1}})$  for all  $j$  between 0 and  $k - 1$ . The direction of a token block is dictated by the direction of the tokens within it. The head of a right-moving (resp. left-moving) block is the leftmost (resp. rightmost) token within the block. The tail of a right-moving (resp. left-moving) block is the rightmost (resp. leftmost) token within the block.

In grid solvers, we use rows and columns to refer to vertices. In a grid, row  $i$  consists of vertices  $v_{0,i}, v_{1,i}, \dots, v_{W-1,i}$ , and column  $i$  consists of vertices  $v_{i,0}, v_{i,1}, \dots, v_{i,H-1}$ . The token surplus in a column  $i$  is equal to the number of tokens on vertices in column  $i$  minus the number of vertices in column  $i$  that are in  $T$ . A horizontal path is a path consisting of vertices in the same row, and a vertical path is a path consisting of vertices in the same column. Horizontal paths and vertical paths are rectilinear paths. A path is said to have a derivation if its sequence of vertices comprises a vertex preceded by a vertex on the same row (resp. column) and succeeded by a vertex on the same column (resp. row).

The number of such vertices is the number of derivations in a path. Paths have directions that are defined in relation to the edges in the path, which can have one of four directions. If the pair  $v_{i,k}, v_{i,k+1}$  (resp.  $v_{i,k}, v_{i,k-1}$ ) is present contiguously in the vertex sequence of the path (for some  $i, k$ ), we call this an upward (resp. downward) edge. If the pair  $v_{i,k}, v_{i+1,k}$  (resp.  $v_{i,k}, v_{i-1,k}$ ) is present contiguously in the vertex sequence of the path (for some  $i, k$ ), we call this a rightward (resp. leftward) edge. A bidirectional path is a path such that its edges have two distinct directions.

A path system in a graph  $G$  is a set  $\mathcal{P} = \{P_0, P_1, \dots, P_{k-1}\}$  of paths, where every path  $P_i$  is a sequence  $\langle v_1, v_2, \dots, v_l \rangle$  of vertices;  $v_1$  is called the source vertex,  $v_l$  is called the target vertex, and consecutive vertices are adjacent in  $G$ . Rerouting a path involves changing its vertex sequence while preserving its source and target vertices. A stationary token is a token on a vertex that is either the source vertex and the target vertex of the same path, or the source vertex of a path and the target vertex of another path. An isolated token is a token whose vertex is the source and target vertex of a path and is not in the vertex sequence of any other path. A merged path system is a path system such that no pair of paths within the system intersects more than once. An intersection between two paths is a nonempty, maximal sequence of vertices that appears in their vertex sequence representation contiguously either in the same order or in reverse order. An unwrapped path system is a path system such that no path within the system contains another path that is not itself within it. A path  $P_i$  is said to contain a path  $P_j$  if the intersection between  $P_i$  and  $P_j$  is  $P_j$ . If path  $P_i$  contains path  $P_j$ ,  $P_j$  is said to be a subpath of  $P_i$ , and  $P_i$  is said to be a superpath of  $P_j$ . It follows that any path is both a subpath and a superpath of itself. A cycle-free path system is a path system such that the graph induced on its paths is a forest. In a path system, every path represents a token move. We say that we execute a path  $\langle v_1, v_2, \dots, v_l \rangle$  when we displace the token on  $v_1$  from  $v_1$  to  $v_2$ ,  $v_2$  to  $v_3$ ,  $\dots$ ,  $v_{l-1}$  to  $v_l$ . A cycle is either represented by a sequence of vertices  $\langle v_1, v_2, \dots, v_k \rangle$  or a sequence of edges  $\langle e_1, e_2, \dots, e_k \rangle$ . Given a set of paths  $\mathcal{P} = \{P_0, P_1, \dots, P_{k-1}\}$  that induces a cycle characterized by edge set  $\mathcal{E}$ , we define edge coloring as the function  $col : \mathcal{E} \mapsto \{0, 1, \dots, |\mathcal{P}| - 1\}$ , i.e. color  $i$  is associated with  $P_i$ . If  $col(e_i) = j$ , we say that the edge  $e_i$  is  $j$ -colored. A cycle is contiguously colored if there does not exist a pair of edges  $e_i$  and  $e_j$  in its edge representation such that  $col(e_i) = col(e_j)$  with the two edges not being contiguous in the cycle. A cycle that is not contiguously colored is discontinuously colored. A color in a cycle is discontinuous if there exists two non-contiguous edges  $e_i, e_j$  in the cycle such that  $col(e_i) = col(e_j)$ .

# CHAPTER 4

## TOKEN MOVING ON PATHS ON THE CPU

Working on paths is pertinent not only because grid instances can be partially solved and reduced to a set of path instances, but also because they allow us to leverage the capabilities of the current hardware via batching. We first start by describing the subroutines that make up every path solver. Path solvers are made up of two parts:

1. The token-vertex matching procedure
2. The token moving procedure

We briefly describe the two procedures. The token-vertex matching procedure involves assigning a token to every target vertex. Formally, the token-vertex matching procedure outputs a one-to-one mapping  $M : T \mapsto TK$  such that this matching is executable. The reason why a token-vertex matching has cardinality  $|T|$  is because no more than  $|T|$  tokens will be moved in a solution for MDTM (see Lemma 2). In the case of the **Greedy path solver**, the token-vertex matching's cardinality is equal to  $|S|$  for reasons that will be explained in Section 4.2.2.

The token moving procedure takes in the output of the token-vertex matching procedure, that is, an executable token-vertex matching, and outputs the corresponding control instructions. Before delving into the details of the representation of the control instructions, we take a quick detour to contextualize the relevance of the MDTM problem on paths.

Saying that MDTM is a good enough heuristic for MSTM is flawed, as it makes no mention of extractions/implantations. In fact, we can construct instances where MDTM does a very poor job if we abide by the definitions that we have been working with up until this point, because it extracts/implants a large number of tokens, though it could have been able to avoid that had we been more relaxed when it comes to displacement. Such instances can occur frequently even if we assume the random and uniform distribution of tokens initially. In Figure



Figure 4.1: Degenerate instance where minimizing displacement incurs a large extraction/implantation overhead.

4.1, there exists a solution for the instance that uses 3 displacements: move  $t_1$  to  $v_4$ , move  $t_2$  to  $v_5$ , move  $t_3$  to  $v_6$ . We incur 3 extraction/implantation operations spread over a total of 3 tokens. However, if we no longer only look at solutions that minimize displacement, one possible solution would be to move  $t_0$  to  $v_4$ . This alternative solution incurs 4 displacements and 1 extraction/implantation. Seeing that one of our prevalent assumptions is that extractions/implantations are more costly than displacements, it is reasonable to think that there exists reasonable per-token loss functions such that the alternative solution is superior in terms of token moving success probability.

The reason why we are strictly dealing with MDTM for paths is because we are working with underlying assumptions that concern atom assembly. The quantum simulator we are working with is capable of extracting aligned atoms in a single operation prior to any displacements taking place. As soon as all displacements are executed, all aligned atoms are implanted in their respective traps. Under this assumption, there is no need to take into consideration extractions/implantations as all atoms are assumed to be extracted once, then implanted once, irrespective of whether they were displaced or not. We work with this assumption for anything that concerns path solvers.

## 4.1 The token moving procedure

We now describe the representation of the control instructions that are outputted by the token moving procedure. We make use of three distinct output modes, which are detailed below.

1. Unbatched output mode: This mode describes a solution for the token moving problem as a sequence of displacements, where each displacement consists of a source vertex and a target vertex. The displacements are executed in the order they appear in in the outputted sequence. Naturally, every displacement has to be permissible vis-à-vis the configuration that results from applying all the preceding displacements in the order they occur in on the starting configuration.
2. Batched output mode: As the name suggests, the batched output mode displaces tokens in batches. In reconfiguration terms, in the case of batched output, a batch replaces a displacement as the smallest transformation step. A batch can be defined as a maximal set of displacements. The displacements need not be in the same direction. As was the case for the unbatched

output mode, the set of displacements making up a batch has to be permissible vis-à-vis the configuration that results from applying all the preceding batches in the order they occur in on the starting configuration.

3. Block batched output mode: The block batched output mode was designed with the specifics of the hardware of the quantum simulator in mind. In reality, while it is true that optical trapping-based systems are capable of displacing multiple atoms at once, the system we are working with can only do so in a single direction. It follows that the control instructions we expect to get from the usage of the batched output mode cannot be directly translated into control operations to be transmitted to the waveform streamer, because the hardware lacks the capability to execute a multidirectional batch in a single time step. We therefore introduce the notion of block batching. A block batch is a batch of token blocks. To indicate the token blocks within a batch, as well as their movement, we include the token head, the direction of the token head, as well as the size of the block, which is the number of tokens in the block, as part of the output. Permissibility of a block batch is defined in a way that is similar to that of a regular batch.

While the first two modes are intuitive and make sense with no added context, the implementation of block batching stems from a hardware consideration. We briefly explain why we opted for this specific output format. Our decision is related to waveform synthesis and streaming. An **elementary waveform** allows us to move a single token block in one direction or the other. Therefore, for a path of size  $N$ , there are a total of  $\frac{2N(N+1)}{2} = N(N+1)$  elementary waveforms ( $\binom{N}{2}$  elementary waveforms in every direction). Such waveforms can be executed by the waveform streaming module to displace atoms that make up a single block. The system allows for more complex control operations to be generated, as we can in fact execute multiple block operations at once, so long as the operations move the respective blocks in the same direction. The procedure that makes this possible consists of retrieving the corresponding elementary waveforms for the desired block operations, then generating a compound elementary waveform. A **compound waveform** is constructed out of a set of elementary waveforms using a commutative operation.

The three output modes we covered above are common to all four path solvers and do not make any assumption regarding how the token-vertex matching is generated. They expect an executable token-vertex matching as input and they output control instructions in the form that is proper to them. We cover the implementation details of the serial version of the three output modes before looking at the details of token-vertex matching generation. The token-vertex matching is inputted in the form of two lists, *matching\_src* and *matching\_dst*,

where  $matching\_src$  contains the index of the vertices containing the traps in the matching in the initial configuration, and  $matching\_dst$  contains the index of the vertices in  $T$ , such that  $matching\_src[i]$  is matched with  $matching\_dst[i]$ .

---

**Algorithm 1** Unbatched token mover

---

**Input:** An executable token-vertex matching,  $matching\_src$  and  $matching\_dst$

**Output:** A sequence of displacements,  $disp\_src$  and  $disp\_dst$

```

1:  $matching\_size \leftarrow matching\_src.length$ 
2:  $num\_displacements \leftarrow 0$ 
3: for  $i \leftarrow matching\_size - 1$  to 0 do
4:   while  $matching\_src[i] < matching\_dst[i]$  do
5:      $disp\_src[num\_displacements] \leftarrow matching\_src[i]$ 
6:      $disp\_dst[num\_displacements] \leftarrow matching\_src[i] + 1$ 
7:      $num\_displacements \leftarrow num\_displacements + 1$ 
8:      $matching\_src[i] \leftarrow matching\_src[i] + 1$ 
9:   end while
10: end for
11: for  $i \leftarrow 0$  to  $matching\_size - 1$  do
12:   while  $matching\_src[i] > matching\_dst[i]$  do
13:      $disp\_src[num\_displacements] \leftarrow matching\_src[i]$ 
14:      $disp\_dst[num\_displacements] \leftarrow matching\_src[i] - 1$ 
15:      $num\_displacements \leftarrow num\_displacements + 1$ 
16:      $matching\_src[i] \leftarrow matching\_src[i] - 1$ 
17:   end while
18: end for

```

---

The unbatched token mover (Algorithm 1) works over two phases, the former of which we will be describing (the latter is symmetric). The first phase of the unbatched token mover loops over the right-moving tokens from right to left, and every right-moving token found in this order is displaced towards the vertex it was matched to via a sequence of displacements. Since we have established in Lemma 3 that the total number of displacements required to execute an executable matching is equal to its total distance, we allocate  $O(N^2)$  memory for both  $disp\_src$  and  $disp\_dst$ , and we use the  $num\_displacements$  variable to keep track of the total number of displacements.

---

**Algorithm 2** Batched token mover

---

**Input:** An executable token-vertex matching,  $matching\_src$  and  $matching\_dst$

**Output:** A sequence of displacements,  $disp\_src$  and  $disp\_dst$ , batch pointers,  $batchPtr$

```
1:  $matching\_size \leftarrow matching\_src.length$ 
2:  $distances\_to\_target \leftarrow get\_distances\_to\_target(matching\_src,$ 
    $matching\_dst, matching\_size)$ 
3:  $num\_batches \leftarrow get\_max\_distance\_to\_target(distances\_to\_target,$ 
    $matching\_size)$ 
4:  $num\_displacements \leftarrow 0$ 
5: for  $batch\_index \leftarrow 0$  to  $num\_batches - 1$  do
6:    $batchPtr[batch\_index] \leftarrow num\_displacements$ 
7:   for  $i \leftarrow 0$  to  $matching\_size - 1$  do
8:     if  $matching\_src[i] < matching\_dst[i]$  then
9:        $disp\_src[num\_displacements] \leftarrow matching\_src[i]$ 
10:       $disp\_dst[num\_displacements] \leftarrow matching\_src[i] + 1$ 
11:       $num\_displacements \leftarrow num\_displacements + 1$ 
12:       $matching\_src[i] \leftarrow matching\_src[i] + 1$ 
13:     else
14:        $disp\_src[num\_displacements] \leftarrow matching\_src[i]$ 
15:        $disp\_dst[num\_displacements] \leftarrow matching\_src[i] - 1$ 
16:        $num\_displacements \leftarrow num\_displacements + 1$ 
17:        $matching\_src[i] \leftarrow matching\_src[i] - 1$ 
18:     end if
19:   end for
20: end for
```

---

As explained earlier, the batched token mover (Algorithm 2) displaces all tokens that have not reached their target vertex yet in every batch. For the sake of brevity, we omit the implementation details of functions that serve a specific purpose that is easy to describe without pseudocode. We first start by computing the number of batches. The number of batches is equal to the maximum distance of a token to its target vertex in the matching, so we start by computing those distances. In a matching  $M$ , the distance of a token  $t_i$  to its target vertex  $M^{-1}(t_i)$  is given by the absolute value of  $idx(M^{-1}(t_i)) - idx(t_i)$ . After obtaining the number of batches, the next step is to generate the batches. A token keeps getting included in batches until it reaches its target vertex. We keep track of the number of executed displacements and we write the starting index of every batch in the displacement arrays to the  $batchPtr$  array. We also make sure to update the locations of the tokens after every batch. The  $batchPtr$  array stores up to  $N - 1$  integers, as that is the maximum number of batches

required to solve a MDTM instance.

---

**Algorithm 3** *single\_direction\_block\_output*

---

**Input:** An executable, single-direction token-vertex matching, *matching\_src* and *matching\_dst*

**Output:** A sequence of token blocks to displace represented by token block head displacements, *head\_src* and *head\_dst*, the sizes of the blocks to displace, *size\_block*, block batch pointers, *batchPtr*

```

1: is_head, block_to_head  $\leftarrow$  detect_heads(matching_src, matching_dst)
2: block_to_tail  $\leftarrow$  detect_tails(matching_src, matching_dst)
3: index_to_block  $\leftarrow$  map_index_to_block(is_head)
4: num_batches  $\leftarrow$  compute_max_distance(matching_src, matching_dst)
5: num_block_displacements  $\leftarrow$  0
6: for i  $\leftarrow$  0 to num_batches - 1 do
7:   batchPtr[i]  $\leftarrow$  num_block_displacements
8:   execute_block_moves(matching_src, matching_dst, block_to_head,
     block_to_tail, head_src, head_dst,
     size_block, num_block_displacements)
9:   update_heads(matching_src, matching_dst, index_to_block,
     block_to_head)
10: end for

```

---



---

**Algorithm 4** *Block batched token mover*

---

**Input:** An executable token-vertex matching, *matching\_src* and *matching\_dst*

**Output:** A sequence of token blocks to displace represented by token block head displacements, *head\_src* and *head\_dst*, the sizes of the blocks to displace, *size\_block*, block batch pointers, *batchPtr*

```

1: matching_r_src, matching_r_dst, matching_l_src,
   matching_l_dst  $\leftarrow$  split_matching(matching_src, matching_dst)
2: single_direction_block_output(matching_r_src, matching_r_dst)
3: single_direction_block_output(matching_l_src, matching_l_dst)

```

---

The block batched token mover algorithm (Algorithm 4) is split into two phases: the first phase displaces right-moving blocks whereas the second phase displaces left-moving blocks. This arrangement aligns with what has been previously mentioned regarding block batching handling blocks in each direction differently. We start by filtering the inputted matching depending on the direction of the tokens, and we make sure to discard the tokens that do not move from the matching. The explanation for the two phases is the same, minus the



details related to the computation of block heads and block tails, as their definitions vary depending on the direction of the block. Since the inputted matching is executable, any subset of it will be executable as well, so the split generates two executable token-vertex matchings, a matching containing left-moving tokens and a matching containing right-moving tokens, which are sorted by their source vertex.

The explanation for the `single_direction_block_output` function will assume that we are dealing with right-moving blocks. The function first detects block heads, and computes a mapping from block indices to heads. A token is a block head if there is no token in the filtered matching occupying the vertex on its left. Since the tokens are sorted in order of the vertices they occupy, it is sufficient to check the previous index of `matching_src`. Clearly, the token on the leftmost vertex is always a head. The number of blocks is equal to the number of heads, so in our detection of heads, we also make sure to keep track of every block's head. We do a similar process for block tails. A token is a block tail if there is no token in the filtered matching occupying the vertex on its right. Clearly, the token on the rightmost vertex is always a tail. For every pair in the matching, we would like to compute the index of the block the token belongs to. This is achievable by a prefix sum on the `is_head` array. Before output generation begins, we would like to figure out what the number of block batches required to execute the token-vertex matching is. We do this identically to lines 2 and 3 of the batched token mover algorithm. We now have all the components that go into output generation. At every batch, we will be adding a pointer that indicates the index at which the corresponding block batch starts in the `head_src` and `head_dst` arrays. We then look for block heads: block heads are in charge of outputting the block displacement. Once a block head is found, it outputs its source and destination (in this case, the destination will be to the immediate right of the source), as well as the size of the block that has this token as its block head. This makes it possible to make use of the format of the output to retrieve elementary waveforms in the next module. Block heads are dynamic: as soon as a block head occupies its target, it is no longer a head, and the head of the block gets updated accordingly to be the leftmost token in the block that has not reached its target vertex yet.

We cover the proof of correctness of the three algorithms in Lemma 3.

## 4.2 The token-vertex matching procedure

We now describe the token-vertex matching procedure. As the name suggests, this procedure matches tokens to vertices. For our purposes, the matching in question needs to have no crossings and be distance-minimizing, as it will be used as input for the token moving procedure, and those two conditions are sufficient to guarantee that the matching is executable. In this chapter, we will

be covering four algorithms that compute an executable matching, with the differences between them to be highlighted in their respective sections. All four algorithms share the same input: two bit arrays, *source* and *target*, of length  $N$  equal to the length of the path.  $source[i]$  is equal to 1 if there is a token on  $v_i$  initially, and is equal to 0 otherwise.  $target[i]$  is equal to 1 if  $v_i \in T$ , and is equal to 0 otherwise. We also include the number of tokens and the cardinality of  $T$ , which we call  $K$  and  $K'$  respectively, as part of the input.

#### 4.2.1 *The Hungarian-based token-vertex matcher*

To motivate the Hungarian-based token-vertex matcher, we describe a related problem: the assignment problem. In the assignment problem, we are given two finite sets  $A$  and  $B$  of equal cardinality and a cost function  $C : A \times B \mapsto \mathbb{R}$ , and we are asked to find a cost-minimizing bijection  $f : A \mapsto B$ , where we define cost as  $\sum_{a \in A} C(a, f(a))$ . This problem has a polynomial-time solution. The first documented solution for it, the Hungarian method, is attributed to Kuhn [8]. The original algorithm has an asymptotic running time of  $O(n^4)$ , where  $n = |A|$ , though it was later shown by Edmonds and Karp that the same algorithm can be improved to achieve an  $O(n^3)$  running time [29]. We show how the problem of computing a distance-minimizing token-vertex matching can be reduced to the assignment problem.

In the absence of token surplus, we set  $A = TK$ ,  $B = T$ , and for any pair  $a \in A$ ,  $b \in B$ , we set  $C(a, b)$  to be the distance between token  $a$  and target vertex  $b$ . Clearly, the matching we get out of running the Hungarian algorithm minimizes total distance. In fact, if we have  $|TK| = |T|$ , this is unnecessary, as there is a single distance-minimizing token-vertex matching that also has no crossings (see Lemma 5), and the matching can be found in time linear in the length of the path. Since we would like the Hungarian-based path solver to solve instances with surplus, we make a small modification in the reduction. We define a set  $U$  ( $|U| = |TK| - |T|$ ), which comprises what we call bogus vertices, and we set  $A = TK$  and  $B = T \cup U$ . As for the cost function, we define it for any tuple in  $A \times B$  as follows:

$$C(a, b) = \begin{cases} d(a, b) & b \in T \\ W & \textit{otherwise} \end{cases}$$

$W$  is a large number (we set  $W$  to be equal to  $|T| \cdot N$  to ensure that it is larger than the largest  $|T|$  distances). Running the Hungarian algorithm on the constructed instance will yield an assignment such that a subset of  $A$  will be assigned to non-bogus vertices in  $B$ ; this subset of the matching is a distance-minimizing matching, and we prove this in Lemma 6.

---

**Algorithm 5** Hungarian-based token-vertex matcher

---

**Input:** Two bit arrays representing the starting and ending configurations, *source* and *target*,  $K$  and  $K'$ , the number of tokens and the cardinality of  $T$  respectively

**Output:** A distance-minimizing token-vertex matching with no crossings, *matching\_src* and *matching\_dst*

- 1: *hungarian\_to\_source\_idx*[ $K$ ], *hungarian\_to\_target\_idx*[ $K$ ]
  - 2: *hungarian\_mat*[ $K$ ][ $K$ ]
  - 3: *compress\_input*(*source*, *target*, *hungarian\_to\_source\_idx*,  
*hungarian\_to\_target\_idx*)
  - 4: *construct\_hungarian\_matrix*(*source*, *target*, *hungarian\_mat*)
  - 5: *hungarian*(*hungarian\_mat*, *matching\_src*, *matching\_dst*,  
*hungarian\_to\_source\_idx*, *hungarian\_to\_target\_idx*,  $K$ ,  $K'$ )
  - 6: *sort*(*matching\_src*)
- 

For the Hungarian-based token-vertex matcher (Algorithm 5), we start by computing mappings. For *source*, we compute the mapping between token indices and the indices of the vertices they occupy. For *target*, we compute the mapping between the order of the vertices in  $T$  and their index in the path. We call this process “compression”. The reason why this step is needed is because the Hungarian cost matrix, which is populated next, contains cost information that uses token indices and order of vertices in  $T$ , with no information about the indices of the source vertices and the target vertices. The Hungarian algorithm will therefore return a matching consisting of token indices and target region indices, which we map back to vertex indices that are written into *matching\_src* and *matching\_dst* in the same function. The Hungarian method implementation we use, which is an optimization of the original implementation of the standard Hungarian method, is an  $O(K^3)$  implementation by Maxim Ivanov based on an algorithm designed by Andrey Lopatin [30]. This implementation writes the vertices in  $T$  in the *matching\_dst* array in order. Since the only guarantee that the Hungarian gives us is distance minimization, we still need to ensure that the outputted matching is executable, so we sort *matching\_src*. We prove that this preserves the total distance while eliminating all crossings in Lemma 4. The running time of this token-vertex matcher is  $O(K^3)$ .

Part of what we aim to achieve when it comes to the MSTM problem is to make the algorithms as flexible as possible. Specifically, even if we aim to minimize displacement, the Hungarian method may output one of many distance-minimizing matchings, and we may get very different success probabilities depending on how the displacement operations are distributed across the tokens in the selected matching. Given that assembly time is of the essence because of the limited lifetime of the atoms, we introduce a batch-limiting mechanism. If we want to ensure that among the distance-minimizing matchings, we pick the one

that also minimizes the maximum distance from a token to its target (i.e. the number of batches), a modification has to be made.

Let  $opt\_bounded\_distance$  be an array, and let  $opt\_bounded\_distance[i]$  be the minimum total distance of a token-vertex matching we can obtain if we bound the maximum distance between a token and its target vertex in the matching by  $i$ . In practice, bounding distances by  $i$  entails changing  $C(a, b)$  to  $W$  for any  $a \in S, b \in T$  such that  $d(a, b) > i$ . Clearly, if the Hungarian method is executed with a distance bound of  $i$ , and the obtained token-vertex matching has a total distance that is greater than or equal to  $(|TK| - |T| + 1) \cdot W$ , this means that there are no valid token-vertex matchings with a maximum distance of  $i$ .

**Observation 1.**  *$opt\_bounded\_distance$  is monotone non-increasing in  $i$ .*

The minimum distance of a token-vertex matching as computed by Algorithm 5,  $min\_total\_distance$ , is equal to  $opt\_bounded\_distance[N - 1]$ , as no assignments are restricted by the latter given that  $d(a, b) \leq N - 1$  for all  $a \in TK, b \in T$ . Minimizing the maximum distance between a token and its vertex in the matching or, equivalently, the total number of batches, therefore means finding the smallest  $i$  such that  $opt\_bounded\_distance[i] = min\_total\_distance$ , and this can be done using a binary search because  $opt\_bounded\_distance$  is monotone non-increasing.

A valid token-vertex matching is a matching that has a total distance that is equal to  $min\_total\_distance$ . The correctness of the algorithm follows from the correctness of binary search combined with Observation 1. With very little modification, Algorithm 6 can be used to choose a token-vertex matching with the desired degree of displacement operation spread across tokens. Rather than using the matching bounded by a distance of  $l$ , where  $l$  is the smallest index such that  $opt\_bounded\_distance[l] = min\_total\_distance$ , we can use any matching bounded by distance  $j$  ( $l \leq j < N$ ), and the larger  $j$  is, the less spread out displacement operations will be, as by increasing the maximum allowed distance per token, we are allowing displacements to concentrate on one or more tokens. The running time of this amended Hungarian-based token-vertex matcher is  $O(K^3 \log N)$ .

---

**Algorithm 6** Hungarian-based token-vertex matcher with batch number minimization

---

**Input:** Two bit arrays representing the starting and ending configurations, *source* and *target*,  $K$  and  $K'$ , the number of source tokens and the cardinality of  $T$  respectively

**Output:** A distance-minimizing, batch-minimizing token-vertex matching with no crossings, *matching\_src* and *matching\_dst*

```

1: hungarian_to_source_idx[ $K$ ], hungarian_to_target_idx[ $K$ ]
2: hungarian_mat[ $K$ ][ $K$ ]
3: compress_input(source, target, hungarian_to_source_idx,
   hungarian_to_target_idx)
4: construct_hungarian_matrix(source, target, hungarian_mat)
5: hungarian(hungarian_mat, matching_src, matching_dst,
   hungarian_to_source_idx, hungarian_to_target_idx,  $K$ ,  $K'$ )
6: sort(matching_src)
7: min_total_distance  $\leftarrow$  get_matching_distance(matching_src,
   matching_dst,  $K'$ )
8:  $l \leftarrow 1$ 
9:  $r \leftarrow N - 1$ 
10: while  $l < r$  do
11:    $m \leftarrow (l + r) // 2$ 
12:   construct_bounded_hungarian_matrix(source, target, hungarian_mat,  $m$ )
13:   hungarian(hungarian_mat, matching_src, matching_dst,
   hungarian_to_source_idx, hungarian_to_target_idx,  $K$ ,  $K'$ )
14:   sort(matching_src)
15:   if is_valid_matching(matching_src, matching_dst,  $m$ ,
   min_total_distance) then
16:      $r \leftarrow m$ 
17:   else
18:      $l \leftarrow m + 1$ 
19:   end if
20: end while
21: construct_bounded_hungarian_matrix(source, target, hungarian_mat,  $l$ )
22: hungarian(hungarian_mat, matching_src, matching_dst,
   hungarian_to_source_idx, hungarian_to_target_idx,  $K$ ,  $K'$ )
23: sort(matching_src)

```

---

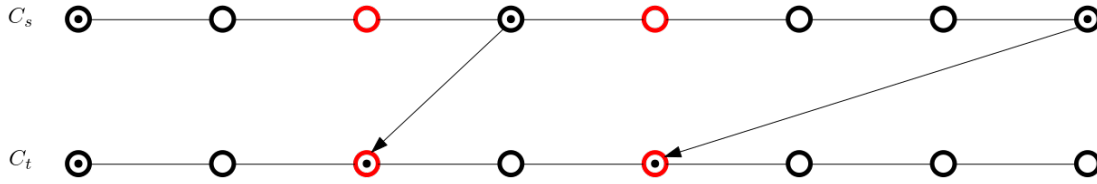


Figure 4.2: Instance where greedy token-vertex matching does not minimize overall displacement

### 4.2.2 *The Greedy token-vertex matcher*

The Hungarian-based path solver achieves its intended purpose of solving the MDTM problem, and can also be augmented to control the number of batches. Its only drawback is that we anticipate its running time to be problematic when it comes to its integration within the RTFC. We are interested in atom assembly on chain geometries consisting of up to 1024 traps, and even for such a relatively small problem size, a running time that is cubic in the number of atoms, which is proportional to the length of the chain, is poor.

We design a greedy path solver for the assignment problem. We loop through the target vertices in order, and, for each target vertex, we pick the closest token that has not been added to the matching and we match it to the selected target vertex. We forego exactness in terms of displacement minimization (Figure 4.2). In the referenced figure, matching  $t_0$  to  $v_2$  and  $t_1$  to  $v_4$  yields a matching with a smaller total distance. Using the greedy path solver, the matching we obtain may have crossings and is not displacement-minimizing, therefore, it is not a valid input for any of the token moving algorithms, and it cannot be transformed into an executable matching using any of the information we have so far. We resolve this issue by expanding the matching to include non-moving tokens, which we add to the matching as tokens that are mapped to the vertex that they initially occupy. The expanded matching is useful because it involves all tokens and sorting tokens and target vertices rids it of crossings: though the expanded matching need not be distance-minimizing, the absence of crossings and the inclusion of all the tokens in the matching is sufficient for us to be able to use a simplified version of the inductive argument in Lemma 3 to prove that the extended matching is an executable one.

The algorithm (Algorithm 8) is very similar to the algorithm for the Hungarian-based token-vertex matcher, the only difference being in the matching function (i.e. `greedy_matching`). This function loops over the columns (i.e. target vertices) of the cost matrix, and finds an available row (i.e. token) with the minimum cost. The pair is added to the matching and the row is marked as used, so that no other column will be able to pick that same row again. Once we are done iterating over all columns, some tokens may have not been matched, so we match those tokens to the vertex they are currently occupying, which is what the for

---

**Algorithm 7** greedy\_matching

---

**Input:** *hungarian\_mat*, *source*, *matching\_src*, *matching\_dst*,  
*hungarian\_to\_source\_index*, *hungarian\_to\_target\_index*,  $K$ ,  $K'$ ,  $N$

**Output:** An expanded token-vertex matching, *matching\_src* and  
*matching\_dst*

```
1: row_taken[K]
2: is_in_matching[N]
3: matching_idx ← 0
4: for col ← 0 to K' - 1 do
5:   chosen_row ← get_min_cost_available_row(hungarian_mat, row_taken)
6:   row_taken[chosen_row] ← True
7:   src_vertex ← hungarian_to_source_index[chosen_row]
8:   trg_vertex ← hungarian_to_target_index[col]
9:   matching_src[matching_idx] ← src_vertex
10:  matching_dst[matching_idx] ← trg_vertex
11:  matching_idx ← matching_idx + 1
12:  is_in_matching[src_vertex] ← True
13: end for
14: for i ← 0 to N - 1 do
15:   if source[i] == 1 and !is_in_matching[i] then
16:     matching_src[matching_idx] ← i
17:     matching_dst[matching_idx] ← i
18:     matching_idx ← matching_idx + 1
19:   end if
20: end for
```

---

---

**Algorithm 8** Greedy token-vertex matcher

---

**Input:** Two bit arrays representing the starting and ending configurations,  
*source* and *target*,  $K$  and  $K'$ , the number of source tokens and the cardinality  
of  $T$  respectively

**Output:** An executable token-vertex matching, *matching\_src* and  
*matching\_dst*

```
1: hungarian_to_source_idx[K], hungarian_to_target_idx[K]
2: hungarian_mat[K][K]
3: compress_input(source, target, hungarian_to_source_idx,
   hungarian_to_target_idx)
4: construct_hungarian_matrix(source, target, hungarian_mat)
5: greedy_matching(hungarian_mat, source, matching_src, matching_dst,
   hungarian_to_source_index, hungarian_to_target_index)
6: sort(matching_src)
7: sort(matching_dst)
```

---

loop at line 14 does. The running time of this token-vertex matcher is  $O(K^2)$ .

### 4.2.3 *The Linear Exact token-vertex matcher*

The usage of the Hungarian method to compute a distance-minimizing token-vertex matching does not make use of all the information that is at our disposition. More specifically, the cost matrix that is constructed out of the MDTM path instances has a specific property: the values in every column are bitonic, that is, they are a circular shift of a monotone sequence, so it should not come as a surprise that a better solver exists. This same exact problem was solved by Karp et al. in 1975 [9]. Karp presents a formulation of the solution that allows us to compute the matching in  $O(K^2)$  time. This same formulation is then broken down and then modified to be computable in time linear in  $K$ . Our implementation is a direct translation of the  $O(K)$  solution that Karp proposed into C code. Since the Linear Exact path solver is able to compute a displacement-minimizing token-vertex matching with a quadratic speedup over the Hungarian-based path solver, is there any drawback to relying on the Linear Exact path solver in all cases? The augmented Hungarian-based token-vertex matcher (Algorithm 6) provides us with the means to indicate the spread of displacement operations across tokens via changing the cost function. Karp's linear assignment algorithm cannot be used as a subroutine in the way the Hungarian method was, therefore we are unable to specify preferences when it comes to the number of batches, and the matching that is computed using the Linear Exact token-vertex matcher is any matching that minimizes total displacement.

### 4.2.4 *The Bruteforce token-vertex matcher*

Bruteforce path solving solves a restricted instance of MDTM on paths, as it assumes that the target vertices induce a path (or, equivalently, it assumes that the target region is contiguous).

The Bruteforce token-vertex matcher (Algorithm 9) relies on two statements: the tokens that are in a displacement-minimizing token-vertex matching are contiguous tokens, and the tokens that are in a displacement-minimizing token-vertex matching have to include all the tokens that are initially within the target region (Lemmas 7 and 8 respectively); any matching that does not conform to those two properties will not be displacement-minimizing and therefore does not need to be considered.

Bruteforce token matching compresses the source array. Afterwards, as the name of the algorithm suggests, we will be trying all possible token-vertex matchings that obey the two aforementioned properties: there are  $O(K')$  such matchings, and those matchings are called splits, owing to the fact that those matchings are constructed by splitting the deficit across the the left and the right of the target region, where the deficit is equal to the number of tokens that the target



---

**Algorithm 9** Bruteforce token-vertex matcher

---

**Input:** Two bit arrays representing the starting and ending configurations,  $source$  and  $target$ ,  $K$  and  $K'$ , the number of source tokens and the cardinality of  $T$  respectively,  $K_1$  and  $K_2$ , the number of vertices to the left and to the right of the target region

**Output:** A distance-minimizing token-vertex matching with no crossings,  $matching\_src$  and  $matching\_dst$

```
1:  $distance\_of\_splits[K' + 1]$ 
2:  $compressed\_source[K]$ 
3:  $compress\_source(source, compressed\_source)$ 
4:  $deficit \leftarrow compute\_target\_deficit(source, target)$ 
5:  $compute\_distance\_of\_splits(source, target, distance\_of\_splits)$ 
6:  $selected\_split \leftarrow find\_min\_distance\_split(distance\_of\_splits)$ 
7:  $cur\_target \leftarrow K_1$ 
8:  $cur\_source \leftarrow selected\_split$ 
9: for  $i \leftarrow 0$  to  $K' - 1$  do
10:    $matching\_src[i] \leftarrow compressed\_src[cur\_source]$ 
11:    $matching\_dst[i] \leftarrow cur\_target$ 
12:    $cur\_source \leftarrow cur\_source + 1$ 
13:    $cur\_target \leftarrow cur\_target + 1$ 
14: end for
```

---

region is missing. We compute the total distance of every split, then we select the index of a split with the minimum total distance, where the index of a split is the index of the leftmost vertex in  $source\_compressed$  included in it. The possible number of splits is  $O(K')$ , and computing the total distance of every split is  $O(K')$ , so the running time of the Bruteforce token-vertex matcher is  $O(K'^2)$ .

## CHAPTER 5

# TOKEN MOVING ON GRIDS ON THE CPU

We now move to the study of solvers for MLTM on grids. Our coverage of path solvers will allow us to establish local guarantees when dealing with grids, and we will be recurrently alluding to path solvers through our work with grid solvers.

In our discussion about path solvers, given that the problem we were tackling aligned in its nature with the capabilities of the hardware, mentioning possible output modes as well as their formats made sense. When it comes to grid solvers, we put less emphasis on output modes, and the reason why that is the case is that the selected output mode does not affect the metrics we are targeting (namely success probability). With that being said, it is worth mentioning that the output of all the algorithms that we implement in this chapter and in the parallel counterpart of this chapter is batched, except for HUNGARIAN-COLAV, even when the algorithms themselves are not designed with batching in mind.

We start by describing REDREC v2.0, which is a grid solver that solves instances of token moving where the target region is centered in the grid. A variant of REDREC v2.0, REDREC v2.1, is designed specifically for parallel solving, and will therefore be omitted from this chapter.

### 5.1 REDREC v2.0

As the name suggests, REDREC v2.0 (Algorithm 10) is an improved version of REDREC, a grid solver that was designed with loss minimization in mind [12]. The improvements come in the form of added maneuverability and provable termination guarantees.

We define terms we use in the description of the algorithm. In a grid graph with a centered target region  $T$ , the reservoir region is the subgraph induced on  $T$ . The top (resp. bottom) reservoir is the grid induced on the vertices that are above (resp. below) the target region in the embedding of the grid graph.

---

**Algorithm 10** REDREC v2.0

---

**Input:** A grid graph in the form of a bit matrix, the height of the top and the bottom reservoir region (denoted  $R_1$  and  $R_2$  respectively)

**Output:** A sequence of displacements,  $disp\_src$  and  $disp\_dst$ , batch pointers,  $batchPtr$

```
1: if number of tokens is smaller than the size of the target region then
2:   return "Cannot be reconfigured"
3: end if
4: Computation of surplus
5: Preemptive partial path solve
6: Solve columns with a surplus of 0
7: while Column with negative surplus exists do
8:   Donor-receiver pair selection
9:   Redistribute-reconfigure-shuffle between donor column and receiver column
10:  if donor surplus  $\geq$  receiver deficit then
11:    Path solve and delete receiver column
12:    if donor surplus = receiver deficit then
13:      Delete donor column
14:    end if
15:  else
16:    Delete donor column
17:    Preemptive partial path solve on receiver
18:  end if
19: end while
20: Path solve unsolved columns
```

---

The steps in REDREC v2.0 are described in detail below:

1. **Computation of surplus:** We are given the starting configuration as a bit matrix. For every column, we compute the surplus: the surplus of a column is the difference between the number of tokens on vertices in the column and the number of target vertices in the column. Since REDREC v2.0 only deals with centered targets by design, the number of target vertices is the same across all columns and is equal to  $H - R_1 - R_2$ . If the surplus is negative, its absolute value is called the deficit.
2. **Preemptive partial path solve:** The premise that underlies the REDREC algorithms is a process we call shuffling. Shuffling consists of displacing tokens from a donor column (a column with positive surplus) to a receiver column (a column with negative surplus). The algorithm restricts such displacements to happen only in the reservoir region (the top reservoir, the bottom reservoir, or a combination of both). As such, we need to ensure that there is always sufficient room in the reservoir to store incoming tokens. Assume some donor column  $d$  would like to shuffle  $m$  tokens to some receiver column  $r$ , which has a deficit that is bigger than or equal to  $m$ , and assume that the receiver column has less than  $m$  vacant vertices in its reservoir region: if shuffling is limited to the reservoir region, we will not be able to shuffle all  $m$  tokens at once. We preemptively handle this case by partially solving such columns (i.e. columns whose deficit is larger than their number of vacant vertices in their reservoir region). Partially sorting a column entails displacing the tokens on the vertices towards the center of the target region, and the intuition behind that is that the tokens we will subsequently shuffle to this column are expected to be evenly distributed across the top and the bottom reservoir regions, therefore minimizing the tokens we would need to move again after preemptively moving them towards the center.
3. **Solving columns with a surplus of 0:** REDREC v2.0 aims to gradually delete columns. Columns that have a positive surplus are potential donor columns, columns that have a negative surplus will definitively be receiver columns at some point, and columns that have a surplus of 0 can be independently solved using a path solver and then deleted. Since solving a column involves displacing all the tokens in the column towards the center, any of our exact path solvers can be used here. We elaborate on the process of deleting a column. A column is deleted if and only if it is solved and it has a surplus of 0. The purpose of deleting a column makes sense in the context of defining adjacency relations between columns: two columns  $i$  and  $j$  ( $i < j$ ) are said to be adjacent if and only if columns  $i + 1, i + 2, \dots, j - 1$  are deleted. Before we start the search for donor-receiver pairs, we solve and delete all columns with a surplus of 0.

4. **Donor-receiver pair selection:** The core mechanism of REDREC v2.0 (as well as its predecessor) is the selection of a pair of columns such that one of the columns acts as a donor column and the other acts as a receiver column. The columns that we select need to be adjacent, and the reason why that is the case is that their adjacency, coupled with restricting the shuffling to the reservoir region and the assurance that there will always be a sufficient number of vacant vertices in the reservoir region of the receiver makes it so that there are no restrictions on the donor's end, i.e. the donor is capable of eliminating the receiver's deficit completely if it possesses a sufficient surplus itself. The choice of a donor-receiver pair is critical, as the redistribution-reconfiguration-shuffling procedure is locally optimal (vis-à-vis displacement). We choose a donor-receiver pair in a way that either minimizes or maximizes some function, and the function can be made to favor reducing extraction/implantation operations at the cost of extra displacement operations or vice versa. The basic function we use is a function that is related to the number of tokens to shuffle between two adjacent columns  $i$  and  $j$ , which is equal to  $\min(|surplus[i]|, |surplus[j]|)$  (if the surplus of the donor is larger than the deficit of the receiver, shuffle just enough to cover the receiver's deficit, otherwise, shuffle all surplus tokens), i.e. we choose a donor-receiver pair  $(i, j)$  that maximizes this value (eligible donor-receiver pairs we consider are pairs of columns such that one column has a nonzero surplus, and the other has a nonzero deficit, and there must exist at least one such pair). This function is used not only to rank pairs of columns, but also to compute the number of tokens to be shuffled. The logic behind maximizing the number of tokens to shuffle is that this should minimize the number of iterations required until all columns have a surplus of zero, though this is heuristic thinking, and there may very well be other, more adequate (or more fit-for-purpose) functions that can supplant this function.
  
5. **Redistribute-reconfigure-shuffle between donor and receiver:** Once a donor-receiver pair is selected, we need to figure out the displacement of tokens in batches required for us to be able to shuffle the tokens from the donor to the receiver. We know the number of tokens to shuffle, but we do not know which tokens will be shuffled from the donor column. To that end, we solve multiple MDTM instances. The logic here is very similar to the logic of the Bruteforce token-vertex matching procedure. Assume we would like to shuffle a total of  $m$  tokens from the donor to the receiver. There must exist at least one pair  $(n, m - n)$  ( $0 \leq n \leq m$ ) such that the receiver has at least  $n$  vacant vertices in the top reservoir and  $m - n$  vacant vertices in the bottom reservoir (this is guaranteed due to the preemptive partial solving procedure): we call such pairs valid reservoir pairs, and each valid reservoir pair constitutes a reservoir split. We fix the source array

(i.e. the initial configuration) to be equal to the donor column's bits. As for the receiver, for each valid reservoir pair  $(n, m - n)$ , the target will consist of 1s in the target region, and we also set the closest  $n$  vacant vertices to the target region in the top reservoir and the closest  $m - n$  vacant vertices to the target region in the bottom reservoir as target vertices. We end up with a total of at most  $m + 1$  possible reservoir splits, and after solving any one of those splits using the Linear Exact path solver, we guarantee (by construction) that each of the  $m$  tokens in the reservoir region of the donor has an unobstructed path towards a vacant vertex in the reservoir region of the receiver (otherwise, the donor column and the receiver column are not adjacent and would not have been selected). The reason why bruteforcing  $m + 1$  reservoir splits is necessary is because we would like to pick the reservoir split that minimizes overall displacement operations: this is a local exactness guarantee that was incorporated as part of our heuristic work. For a given valid reservoir split, the choice of which vertices to select as target vertices seems arbitrary, because it may be the case that farther vacant vertices in the receiver column lead us to incur fewer overall displacement operations. We stress that the calculation of incurred displacement operations is not limited to the number outputted by the path solver; we also anticipate that the tokens that were picked for shuffling will be displaced further towards the target region of the receiver, so this distance is also taken into account. This extra distance is calculated as follows: for every shuffled token, we add its distance to the vacant vertex in the target region of the same order. In other words, for the  $i^{th}$  highest token among the shuffled tokens, we take into consideration its distance to the  $i^{th}$  highest vacant vertex in the target region. This justifies our choice of setting the closest vacant vertices to the receiver as target vertices. As soon as we obtain the reservoir split that minimizes the modified displacement metric, the tokens are displaced based on the output of the path solver. Afterwards, the tokens destined for shuffling are shuffled all at once in batches towards the receiver column, with the number of batches required to shuffle those tokens being equal to the distance in columns between the donor column and the receiver column, including deleted columns, if any.

*To summarize:* redistribution-reconfiguration-shuffling consists of two steps: redistributing the tokens on the donor column's end, which inherently leads to its reconfiguration (i.e. its target region being covered with tokens), and shuffling, which displaces tokens from the donor to the receiver. Note that we do not execute any displacements within the receiver column at this point, we just ensure that the tokens we intended to shuffle are now on vertices in the receiver column.

**Column deletion:** After redistribution-reconfiguration-shuffling, we have one of three cases, and the case we are looking at is determined by whether

the initial surplus of the selected donor column exceeds the initial deficit of the selected receiver column. If that is the case, after shuffling, the receiver has a surplus of 0, which means that we can use the standard path solver on it and delete it (as explained in step 3). If the surplus of the donor column is equal to the deficit of the receiver column, the same process is executed, and we also delete the donor column; the donor column would have already been solved in the redistribution-reconfiguration-shuffling phase, so we do not need to handle its solving separately. If the surplus of the donor column is less than the deficit of the receiver column, the receiver still has a negative surplus, even after reshuffling, so we are not able to solve the receiver and delete it. The donor, however, has been solved in the redistribution-reconfiguration-shuffling phase, and is deleted. It may be the case that shuffling violated the invariant related to step 2: if it did, the receiver is partially solved.

**Path solving unsolved columns:** Once we ensure that no column has any deficit, we know that all columns have a surplus that is bigger than or equal to 0. Some of those columns are already reconfigured and marked as deleted, some of those columns are already reconfigured but are not marked as deleted (this applies to any column that had redistribution-reconfiguration applied to it at least once while still having a positive surplus), and some columns have not been reconfigured yet, despite having sufficient tokens. This final step loops through all columns that have not been deleted yet and solves them. At the end of this step, we ensure that all columns have been individually reconfigured, meaning that the instance has been reconfigured as well.

The proof of termination and correctness of REDREC v2.0 can be found in Lemma 9.

## 5.2 The Hungarian-based grid solvers

We now introduce two algorithms that heuristically solve the MLTM problem on grids.

We start by describing the motivation behind the inception of our two algorithms. In the literature, overall displacement minimization and overall extraction/implantation minimization have been treated as two separate problems. For overall displacement minimization, we have positive results: given any graph, a set  $S$  of vertices and a set  $T$  of vertices ( $|S| = |T|$ ), such that the vertices in  $S$  are occupied, MDTM has a polynomial-time solution. The algorithm is described in the work of Călinescu et al., and unsurprisingly involves computing a distance-minimizing token-vertex matching, which can be done using the

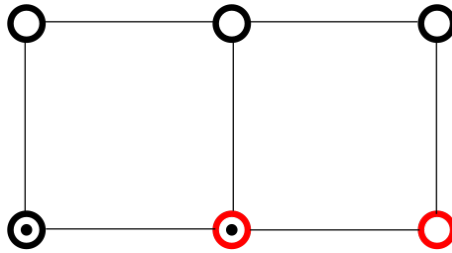


Figure 5.1: Instance where overall displacement operations and overall extraction/implantation operations cannot be minimized at once.

Hungarian method [11]. The described algorithm can be extended to work with surplus using the construction of Subsection 4.2.1.

In the same paper, some results concerning MEITM on general graphs and grid graphs are also covered. In particular, we know that MEITM does not admit a polynomial time solution on general graphs, and this remains the case even when we restrict the problem to grid graphs. We resort to approximation algorithms. MEITM is APX-hard on general graphs. Fortunately, we have positive results; the problem admits a 3-approximation on general graphs, and therefore, on grid graphs as well. The 3-approximation uses an exact extraction/implantations solver for tree instances which can be found in the same paper as a subroutine.

The 3-approximation algorithm for MEITM is displacement-unaware, and the displacement-minimizing algorithm is extraction/implantation-unaware. In other words, we do not have verifiable bounds on the other metric in each of the two algorithms. In an ideal world, we would want to minimize both overall displacement operations and overall extraction/implantations. Unfortunately, this is not possible for all grid instances. In Figure 5.1, a displacement-minimizing solution consists of two displacements and two extraction/implantations: one such solution (and in fact, the only one for this instance) consists of displacing the middle token to the right, followed by displacing the left token to the right. When it comes to extraction/implantation minimization, an extraction/implantation minimizing solution consists of extracting the leftmost token, displacing it once upward, twice to the right, then once downward, and then implanting it, for a total of 1 extraction/implantation and 4 displacements. Clearly, we cannot come up with a solution involving a single extraction/implantation (i.e. that displaces a single token) and that has two displacements.

The pattern seems to be that the interaction between overall displacement operations and overall extraction/implantation operations is such that a decrease in one is possible when the other is increased, and vice versa. However, this is only true for a finite set of solutions; if we consider a solution that consists of an unnecessary sequence of extraction/implantations or displacements, those can be reduced without the implication of an increase in the other metric of interest. Let a pareto-optimal solution be a solution such that we cannot de-



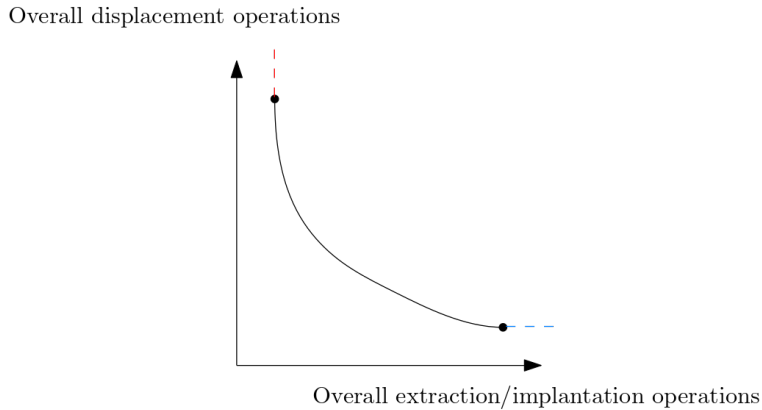


Figure 5.2: Relation between overall displacements and overall extractions/implantations in pareto-optimal solutions

crease overall displacement (resp. overall extractions/implantations) without increasing overall extractions/implantations (resp. overall displacement). Figure 5.2 models the shape that the pareto-optimal solutions for a fixed instance take on in a plot that compares overall displacement operations to overall extraction/implantation operations. The leftmost point represents pareto-optimal solutions among the displacement-minimizing solutions, whereas the rightmost point represents pareto-optimal solutions among the extraction/implantation-minimizing solutions. The remaining pareto-optimal solutions can be found along the curve between those two points. The dotted red line corresponds to solutions for the MDTM problem, whereas the dotted blue line corresponds to solutions for the MEITM problem. We have already seen that none of the solutions along the blue line can be computed in polynomial time unless  $P = NP$ . It follows that computing pareto-optimal solutions is hard, because a solver that can compute the solutions along the curve in polynomial time would also solve the MEITM problem in polynomial time, and we know that this is not possible.

Given a MLTM instance, and knowing that we have a number of solutions that is exponential in the size of the input, which solution do we go for? It should not be too surprising that we favor pareto-optimal solutions over the solutions that fall on the right of the curve. Among all the possible pareto-optimal solutions in Figure 5.2, what is the best solution to go for? The answer to this question will entirely depend on the per-token loss functions, as well as the aggregate loss function. Penalizing extractions/implantations more than displacements will make options on the curve towards the left more attractive, whereas penalizing displacements more than extractions/implantations makes solutions that fall on the curve towards the right better choices. However, we have established that there does not exist an algorithm that can generate solutions along the curve, so we focus our attention on heuristics: our aim is to design algorithms that output solutions that fall to the right of the referenced figure and that take on its shape,

and this is exactly what our algorithms do.

The two algorithms in question present a lot of similarities when it comes to the procedures they rely on. As such, we start with a general presentation of the modules that go into them.

## 5.2.1 *The modules*

### 5.2.1.1 All-pairs shortest path (APSP)

Computing a distance-minimizing token-vertex matching involves knowing the shortest path between any token and any target vertex. On paths, this is trivial, as the path between a token and a target vertex is unique. In fact, even on grids, this remains easy assuming that the edges of the grids are unweighted: the length of the shortest path between two vertices  $v_{c_1,r_1}$  and  $v_{c_2,r_2}$  is  $abs(r_1-r_2)+abs(c_1-c_2)$  (i.e. the Manhattan distance between the two vertices in the embedding of the grid graph) and we arbitrarily select the shortest path to be the rectilinear path from  $v_{c_1,r_1}$  to  $v_{c_2,r_1}$  followed by the rectilinear path from  $v_{c_2,r_1}$  to  $v_{c_2,r_2}$ . In some cases, the input of this module is an edge-weighted grid graph: in this scenario, we make use of the Floyd-Warshall algorithm to compute all-pairs shortest path [31]. We augment the Floyd-Warshall algorithm to make it store a shortest path between any source vertex and any target vertex. If surplus exists, we add bogus target vertices which are assigned a distance of  $+\infty$  to all source vertices.

### 5.2.1.2 Minimum weight perfect matching with path retrieval (MWPM + PR)

The minimum weight perfect matching problem is equivalent to the assignment problem covered in Section 4.2.1. This module takes in a matrix containing the length of the shortest path between every source vertex-target vertex pair (including bogus target vertices) and outputs a distance-minimizing source vertex-target vertex matching. The source vertices that were matched to bogus target vertices are added to the matching as vertices mapped to themselves. We therefore end up with a source vertex-target vertex matching of size equal to the number of tokens. After the matching is generated, the matching is turned into a path system: for every pair in the matching, the corresponding shortest path is retrieved from the output of APSP, and we end up with a set of paths which form our path system. The sum of the edge lengths of the paths in the obtained path system  $\mathcal{P}$  is equal to the distance of the source vertex-target vertex matching.

### 5.2.1.3 Collision avoidance (COLAV)

The path system that was outputted by MWPM + PR is distance-minimizing. Therefore, if we were to compute a solution that has an equal number of displacement operations, the solution will fall on the dotted red line, i.e will involve a lot

of extractions/implantations. We look into means of mitigating overall extraction/implantation operations, even if that comes at the cost of extra displacement operations. To that end, we introduce three versions of collision avoidance, or COLAV.

### 5.2.1.3.1 Bidirectional COLAV

Bidirectional collision avoidance is the only version of COLAV that does not increase overall path system distance. The purpose of bidirectional COLAV is to maximize token isolation, as isolated tokens are not displaced. It does so by looping over every path, and, for every path, fixing the rest of the path system, and rerouting the current path in a way that maximizes token isolation while preserving the length of the path. If the original path is rectilinear, there is nothing to do, as the path cannot be rerouted without increasing its length. Otherwise, suppose that the source vertex of the path is  $v_{c_1, r_1}$  and the target vertex of the path is  $v_{c_2, r_2}$ . Let  $dp_W = \text{abs}(c_1 - c_2)$ ,  $dp_H = \text{abs}(r_1 - r_2)$ ; between vertex  $v_{c_1, r_1}$  and  $v_{c_2, r_2}$ , there are a total of  $\frac{(dp_W + dp_H)!}{dp_H! dp_W!}$  distance-minimizing paths. Using a brute-force approach for every path is therefore not feasible, as the number of rerouted paths to consider for every path is exponential in the Manhattan distance between the source vertex and the target vertex of the path.

For the rest of this section, we assume that we are working with a path system  $\mathcal{P}$  and a path  $P_{cur}$  with a source vertex  $v_{c_1, r_1}$  and a target vertex  $v_{c_2, r_2}$  such that  $r_1 < r_2$  and  $c_1 < c_2$ . The other three cases entail changing some signs. We also define  $dp_W$  and  $dp_H$  as we did above.

The problem we are trying to solve presents a substructure that we can make use of to design a polynomial time dynamic programming solution. Before we describe the details of the dynamic programming solution, for each path  $P_{cur}$  considered, we populate a  $dp_W \times dp_H$  matrix,  $is\_isolated\_token$ , as follows:  $is\_isolated\_token[i][j]$  is set to 1 if  $v_{c_1+i, r_1+j}$  contains an isolated token in  $\mathcal{P} \setminus \{P_{cur}\}$  and is set to 0 otherwise.

For the path  $P_{cur}$ , we introduce a  $dp_W \times dp_H$  matrix,  $dp$ , such that  $dp[i][j]$  is the smallest number of isolated tokens on any bidirectional path between  $v_{c_1, r_1}$  and  $v_{c_1+i, r_1+j}$  in  $\mathcal{P} \setminus \{P_{cur}\}$ .  $dp[i][j]$  is computed as follows:

$$dp[i][j] = \begin{cases} 0 & i = 0, j = 0 \\ is\_isolated\_token[i][j] + dp[i-1][j] & j = 0 \\ is\_isolated\_token[i][j] + dp[i][j-1] & i = 0 \\ is\_isolated\_token[i][j] + \min(dp[i-1][j], dp[i][j-1]) & otherwise \end{cases}$$

The value we are interested in is  $dp[dp_W - 1][dp_H - 1]$ , which can be computed in  $O(dp_H dp_W)$  for one path. The proof of correctness of this algorithm can be found in Lemma 10.

Once we obtain  $dp[dp_W - 1][dp_H - 1]$ , if its value is smaller than the number of isolated tokens in  $\mathcal{P} \setminus \{P_{cur}\}$  the original path  $P_{cur}$  was going through, then  $P_{cur}$  has to be rerouted, otherwise,  $P_{cur}$  is unchanged. Since path reconstruction is required, we need to store the decisions that were made by the dynamic programming procedure, and the easiest way to do that is by introducing a  $dp_W \times dp_H$  matrix,  $prev$ , such that  $prev[i][j]$  indicates whether  $dp[i][j]$ 's value was obtained by reaching vertex  $v_{c_1+i, r_1+j}$  from the bottom ( $prev[i][j] = 0$ ) or from the left ( $prev[i][j] = 1$ ). Using the  $prev$  matrix, we can therefore reconstruct the rerouted path and substitute the initial path with it.

We now describe how bidirectional COLAV is run. Bidirectional COLAV loops over all paths in the path system inputted to it, and attempts to reroute each path. If at least one path was rerouted, once all paths have been iterated over, the process is repeated. The process keeps getting repeated until the algorithm goes through all paths without modifying any path. The algorithm terminates in polynomial time, and its running time is  $O((HW)^3)$  (Lemma 11).

### 5.2.1.3.2 Bounded COLAV

Bidirectional COLAV is a best-effort approach that increases token isolation at no extra displacement cost. In other words, bidirectional COLAV gets us closer to the rightmost point on the curve in Figure 5.2. Bidirectional COLAV will not allow us to achieve a movement along the direction of the curve, as we are not foregoing displacement minimization. Therefore, we would like to introduce a mechanism that allows us to increase token isolation, even if that comes at the cost of extra displacement operations. We also want to make it possible to control how much leeway is given to this algorithm when it comes to deviating from the minimization of overall displacement operations.

We introduce the concept of margin. In bidirectional COLAV, we were dealing with bidirectional paths. In bounded COLAV, we define a parameter  $m$  for margin, which will define the range of the paths we consider. For a margin  $m$ , a source vertex  $v_{c_1, r_1}$  and a target vertex  $v_{c_2, r_2}$  (WLOG,  $r_1 < r_2$  and  $c_1 < c_2$ ), the rerouted path can include any of the vertices that are within the subgrid bounded by the vertices  $(\max(c_1 - m, 0), \max(r_1 - m, 0))$  (bottom left corner) and  $(\min(c_2 + m, G_W - 1), \min(r_2 + m, G_H - 1))$  (top right corner) and those paths can involve edges in any of the four directions.

As was the case for the analysis of bidirectional COLAV, for the rest of this section, we assume that we are working with a path system  $\mathcal{P}$  and a path  $P_{cur}$  with a source vertex  $v_{c_1, r_1}$  and a target vertex  $v_{c_2, r_2}$  such that  $r_1 < r_2$  and  $c_1 < c_2$ . The other three cases entail changing some signs.  $dp_W$  and  $dp_H$  are defined in the same way they were in bidirectional COLAV.

Evidently, there is no point in attempting to brute-force the paths here either, as their number is exponential in  $dp_H + dp_W + m$ . In fact, even for  $m = 0$ ,

the possible reroutings are a superset of the possible reroutings in bidirectional COLAV, as we removed the restriction on bidirectionality.

We start by describing the rerouting selected by bounded COLAV. Bounded COLAV selects the rerouting of  $P_{cur}$  that maximizes the number of isolated tokens. If there are multiple such reroutings, bounded COLAV selects the rerouting that has the shortest path length. If there are multiple such reroutings, bounded COLAV selects the rerouting that has the smallest number of derivations. If there are multiple such reroutings, we arbitrarily select one of them.

Again, we make use of dynamic programming to solve the problem. Bounded COLAV, just like bidirectional COLAV, will make use of the *is\_isolated\_token* matrix; however, in the case of bounded COLAV, we have to cover all vertices that are part of the subgrid defined above, so the *is\_isolated\_token* matrix is of size  $(dp_W + 2m) \times (dp_H + 2m)$ . We introduce a  $([dp_H + 2m] \times [dp_W + 2m]) \times (dp_W + 2m) \times (dp_H + 2m)$  matrix,  $dp$ , such that  $dp[i][j][k]$  is the smallest number of isolated tokens on any path of length  $i$  between  $v_{c_1, r_1}$  and  $v_{c_1+j-m, r_1+k-m}$  in  $\mathcal{P} \setminus \{P_{cur}\}$ .  $dp[i][j][k]$  is computed as follows:

$$dp[i][j][k] = \begin{cases} 0 & i = 0, j = m, k = m \\ -1 & i = 0, j \neq m \text{ or } k \neq m \\ -1 & dp[i-1] = -1 \text{ for} \\ & \text{all neighbors} \\ is\_isolated\_token[r_1 + j - m][c_1 + k - m] \\ + \min(dp[i-1][j-1][k], dp[i-1][j+1][k], \\ dp[i-1][j][k-1], dp[i-1][j][k+1]) & otherwise \end{cases}$$

By “neighbors” of a pair  $(j, k)$  we are referring to the subset of the set  $\{(c_1 + j - m - 1, r_1 + k - m), (c_1 + j - m, r_1 + k - m - 1), (c_1 + j - m + 1, r_1 + k - m), (c_1 + j - m, r_1 + k - m + 1)\}$  that corresponds to row and column coordinates of vertices that are within the grid. Also, we have previously defined the dimensions of  $dp$ ; by convention, the value of  $dp[i][j][k]$  where at least one of  $i, j$  or  $k$  is out of bounds is equal to  $+\infty$ .

The value we are interested in is  $\min(dp[dp_H + dp_W][m + dp_W - 1][m + dp_H - 1], dp[dp_H + dp_W + 1][m + dp_W - 1][m + dp_H - 1], \dots, dp[(dp_H + 2m) \times (dp_W + 2m) - 1][m + dp_W - 1][m + dp_H - 1])$ . That is, we consider the maximum number of isolated tokens for every path length and we pick the maximum across all path lengths. The proof of correctness of this algorithm can be found in Lemma 12.

Just like bidirectional COLAV, bounded COLAV requires keeping track of paths, their lengths, as well as their number of derivations;  $dp$  can be easily augmented to accommodate for that. Bounded COLAV is run in a similar fashion

to bidirectional COLAV, that is, we loop over all paths while there are some paths that were rerouted in the last pass.

Bounded COLAV has one notable advantage over bidirectional COLAV: the same definition of  $dp$  holds on general graphs, provided we omit derivations from our analysis and we modify the definition of neighboring vertices accordingly. The proof of termination of the algorithm can be found in Lemma 13.

### 5.2.1.3.3 Derivation-averse COLAV

In some cases, the hardware is not capable of executing a multidirectional path with a single extraction/implantation operation. Though this is not the case of the quantum simulator that we are working with, derivation-averse COLAV was designed with the knowledge of the existence of such quantum simulators in mind. For a selected path, derivation-averse COLAV reroutes it in a way that minimizes the sum of the derivations and the number of isolated tokens along the path in the path system that excludes it. The intuition is that we are counting every derivation as an extra extraction/implantation of the token.

The implementation details of this COLAV technique will be omitted, as its relevance is limited. With that being said, it presents a lot of similarities to bounded COLAV, and it is also applicable to general graphs.

### 5.2.1.4 Ordering

This module is the core module of both algorithms. The purpose of ordering is to ensure that no token has to be extracted/implanted more than once while guaranteeing that we do not increase the total distance of the path system. The ordering module takes in an arbitrary path system and turns it into a cycle-free path system without increasing total distance. We now describe the different parts that go into ordering.

**5.2.1.4.1 Path merging** The path merging module takes in an arbitrary path system and turns it into a merged path system without increasing the overall distance of the path system or the number of distinct edges used in the path system. For a path  $P_i$  that intersects path  $P_j$ , let  $v_{i,s}$  and  $v_{i,e}$  be the first and the last vertex in  $P_i$  that are in  $P_j$ : the subsequence  $v_{i,s} \dots v_{i,e}$  induces a subpath denoted by  $P_{i,s_i \rightarrow e_i}$  in  $P_i$ , and a subpath denoted by  $P_{j,s_i \rightarrow e_i}$  in  $P_j$ . An edge in a pair of intersecting paths  $P_i, P_j$  is said to be unique if it belongs to one of  $P_{i,s_i \rightarrow e_i}, P_{j,s_j \rightarrow e_j}$  but not the other.

With the above in mind, we describe the merging process. While there exists an edge in the path system that is unique in some pair of intersecting paths, we look for the edge that is unique in the smallest number of intersecting path pairs. If such an edge does not exist, this implies that the path system is merged. Once we have selected an edge, we pick an arbitrary pair of intersecting paths

where the edge is unique, and we proceed to merge this pair. Let the selected pairs be  $P_i$  and  $P_j$ , and we attempt to reroute  $P_{i,s_i \rightarrow e_i}$  through  $P_{j,s_i \rightarrow e_i}$ , or  $P_{j,s_j \rightarrow e_j}$  through  $P_{j,s_i \rightarrow e_i}$ : if one of the reroutings shortens the length of the rerouted path, the rerouting is preserved. Otherwise, we make both paths go through whichever of  $P_{j,s_i \rightarrow e_i}$  and  $P_{i,s_j \rightarrow e_j}$  maximizes token isolation. If those subpaths isolate the same number of tokens, we make both paths go through the subpath that does not contain the edge we selected initially.

By definition of a merged path system, termination implies correctness. Therefore, it remains to show that the algorithm terminates. We prove this in Lemma 14.

**5.2.1.4.2 Path unwrapping** The path unwrapping module takes in a merged path system and turns it into a merged and unwrapped path system without increasing the overall distance of the path system or the number of distinct edges used in the path system. We go through the paths in arbitrary order, and for every path, we unwrap all the paths that it wraps. To do so, we go through the path system and we detect all paths whose source vertices and target vertices are contained within the selected path. We then sort the source vertices and the target vertices by their order of appearance within the selected path. The final step assigns source  $i$  to target  $i$  in the ordering; the path with source  $i$  as source vertex is then rerouted to target vertex  $i$  via the selected path.

For a given selected path, this process destroys all wrappings within it. Assume it does not, that is, the assignment of sources to targets in order of appearance in the initial selected path made a wrapping persist. The concerned vertices indexed by their order,  $s_i, s_j, t_i, t_j$ , have had to appear in one of four orders. Without loss of generality, we will assume that the path with  $s_i$  as its source vertex wraps the other path. Two of those four orders will be covered, as the analysis for the other two is symmetrical. If the sequence of vertices in the initial selected path takes on the form  $\dots s_i, \dots s_j, \dots t_j, \dots t_i, \dots$ , we have a contradiction, as this implies that  $i < j$  and  $j < i$ . The same is true if we have the form  $\dots s_i, \dots t_j, \dots s_j, \dots t_i, \dots$ . The last two forms which are identical to those with the order of  $s_i$  and  $t_i$  flipped yields the same contradiction.

We still have to show that path unwrapping does not “unmerge” a path system, and that there are no wrappings left when the algorithm terminates. We do that in Lemmas 15 and 16 respectively.

**5.2.1.4.3 Cycle breaking** The cycle breaking module takes in a merged, unwrapped path system and outputs a merged, unwrapped, cycle-free path system without increasing the overall distance of the path system or the number of distinct edges used in the path system.

Removing cycles in a graph can be done in polynomial time by computing a minimum spanning tree (MST). In our case, after the minimum spanning tree is computed, we would want to recompute a source vertex-target vertex matching,

---

**Algorithm 11** Cycle breaking

---

**Input:** A merged, unwrapped path system with total distance  $d$

**Output:** A merged, unwrapped, cycle-free path system with total distance  $d' \leq d$

- 1: Populate edge frequencies
  - 2: **while** a cycle exists **do**
  - 3:     Identify cycle
  - 4:     Break cycle
  - 5:     Merge path system
  - 6:     Unwrap path system
  - 7:     Populate edge frequencies
  - 8: **end while**
- 

as well as a path system out of the matching. The example in Figure 5.3 shows that using minimum spanning trees to break cycles may increase the total distance of the path system, no matter the recomputed matching; the graph the figure deals with is a weighted graph, but the argument can be modified to work with unweighted graphs by replacing weighted edges with paths of the same length.

Cycle breaking consists of a sequence of functions that are executed as long as cycles in the path system exist. Those functions will be detailed separately below:

**5.2.1.4.3.1 Computing edge frequencies** This is a fairly straightforward function: given a path system, the frequency of an edge is the number of paths containing it. The running time of this function is linear in the total distance of the path system.

**5.2.1.4.3.2 Cycle detection** Cycle detection consists of finding whether there is a cycle in the graph induced on the paths in the path system, which we will call the path system graph. This can be achieved using any graph traversal algorithm; either a breath-first search (BFS) or a depth-first search (DFS) is sufficient. However, this function is more elaborate: if a cycle is found, we want it to return the least frequent edge among all edges in cycles, where edge frequency is defined with respect to the entire path system as highlighted in the previous step. We therefore sort the edges with nonzero frequencies in non-decreasing order of frequency, and then, in this ordering, we look for the earliest edge that is part of a cycle. Observation 2 explains how we can check whether an edge is part of some cycle using either BFS or DFS.

**Observation 2.** *Let  $u$  and  $v$  be the endpoint of some edge  $e$ .  $e$  is part of a cycle in an undirected graph  $G$  if and only if  $v$  is reachable from  $u$  in  $G \setminus \{e\}$ .*



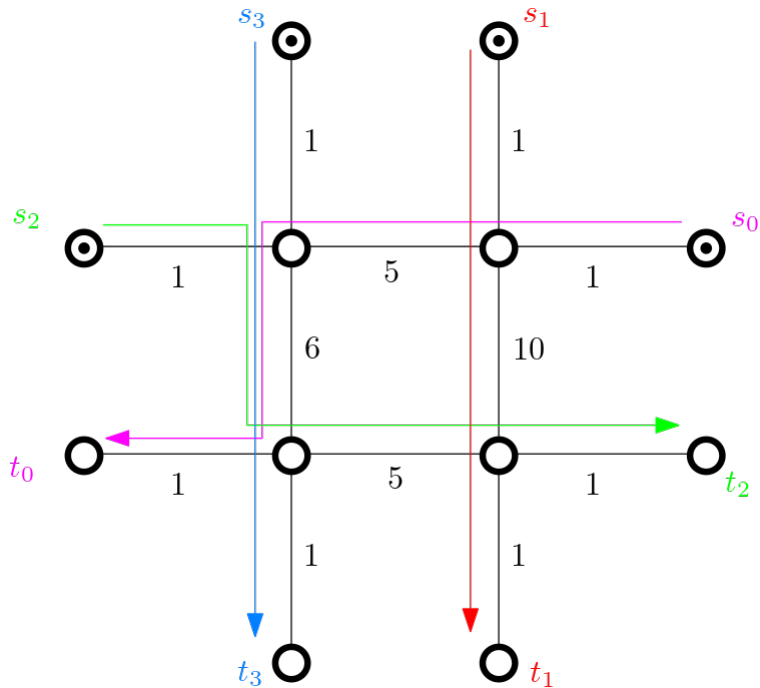


Figure 5.3: Example of a path system that induces a cycle that cannot be broken via computing a MST without increasing total path system distance. The initial path system has a total distance equal to 46. The MST of the graph the path system induces includes all the edges except the edge of weight 10. Computing all-pairs shortest path on the MST followed by the computation of a distance-minimizing source vertex-target vertex matching forms a path system whose total distance is equal to 52 (irrespective of the matching, as the edges of weight 5 will each be in two paths and the edge of weight 6 will be in 4 paths regardless of the outputted matching, which can be any of  $4! = 24$  possible matchings).

**5.2.1.4.3.3 Cycle identification** Given the edge outputted by the previous step, which we call the cycle edge, we now look for a cycle that contains it. More specifically, we are interested in retrieving paths that induce a cycle containing the cycle edge. Before we describe the procedure that allows us to retrieve the desired set of paths, we cover a few definitions. We define two types of paths: cycle edge-containing paths which are paths that contain the cycle edge, and non-cycle edge-containing paths which are paths that do not contain the cycle edge. If there is a cycle in the path system graph, there must exist a set of paths that contains at most two cycle edge-containing paths, such that the graph induced on the paths in the set contains a single cycle which includes the cycle edge. Moreover, the cycle is contiguously colorable. The resulting cycle is called a special cycle. This is a result of Lemmas 17, 18 and 19; we apply the algorithms from each lemma sequentially, and we end up with a special cycle, as well as the set of paths that induce it.

Let  $r$  be the number of cycle edge-containing paths in the path system. For each cycle edge-containing path, we construct  $2(r - 1) + 1$  different graphs we call path intersection graphs, for a total of  $r(2(r - 1) + 1)$  path intersection graphs. Every path intersection graph is built out of a selection of one cycle edge-containing path as a base path, and at most one other cycle edge-containing path as a support path. In every path intersection graph, we add a vertex for every non-cycle edge-containing path, and we add an edge between two such vertices if the corresponding paths intersect. Moreover, in every path intersection graph with  $P_i$  as its base path, we add two vertices  $l_{P_i}$  and  $r_{P_i}$ . The vertex  $l_{P_i}$  (resp.  $r_{P_i}$ ) is associated with all the vertices in the base path that occur before (resp. after) the cycle edge in the vertex sequence of the path (including one of the endpoints of the cycle edge in both cases). If some non cycle edge-containing path intersects  $P_i$ , we add an edge from its vertex to either  $l_{P_i}$  or  $r_{P_i}$  depending on whether it intersects  $P_i$  in the subpath associated with  $l_{P_i}$  or  $r_{P_i}$  (clearly, a non cycle edge-containing path cannot intersect both subpaths without going through the cycle edge because the path system is merged). We still have to explain what a support path is. A support path is a cycle edge-containing path that is needed as part of the cycle we are looking for. Such paths may either be treated as left-intersecting or right-intersecting paths, and that explains why we construct  $2(r - 1) + 1$  different path intersection graphs for every base path.

We claim that the problem of obtaining a set of paths that induces a special cycle reduces to running  $(l_{P_i}, r_{P_i})$  reachability queries on the constructed  $r(2(r - 1) + 1)$  path intersection graphs (Lemma 20). More specifically, we use BFS to check whether  $r_{P_i}$  is reachable from  $l_{P_i}$ , and if we find any yes-instance, we reconstruct the set of paths by using the BFS tree: this set of paths induces a special cycle.

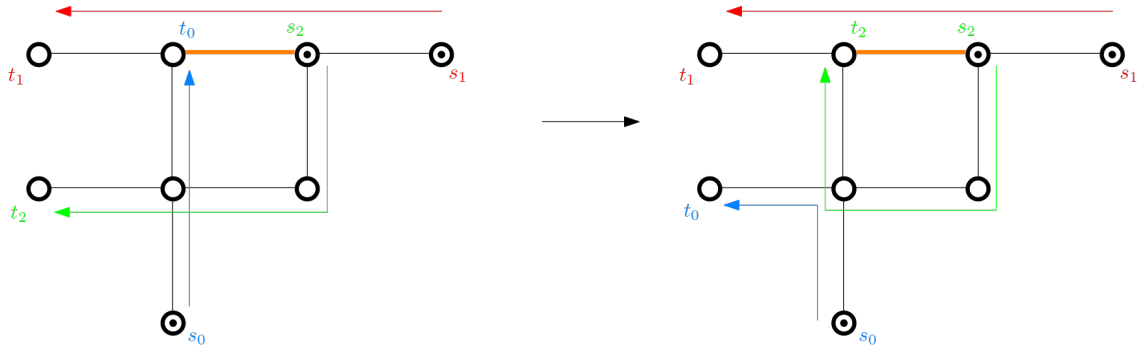


Figure 5.4: Leveraging source-target corners to reduce the number of paths that induce the special cycle

**5.2.1.4.3.4 Cycle breaking** This procedure takes in a sequence of paths (the selected paths) that induce a special cycle and works on breaking said cycle by modifying the path system. As a consequence of the cycle being special, the only intersections that exist within the sequence of paths is between two contiguous paths (the first and the last path in the sequence of paths are contiguous as well).

This procedure aims to reduce the number of paths that induce the cycle in question by looking for what we call a source-target corner. For every pair of contiguous paths, we swap target vertices, and we look at the graph induced on the updated selected paths: if one of the two updated paths can be eliminated from the selected paths without destroying the cycle, we say that we found a source-target corner, and we end up with a smaller set of paths that induces the same cycle. In Figure 5.4, 3 paths induce the special cycle. A source-target corner exists between path 2 and path 0, because swapping targets and removing one of the two paths preserves the cycle, so targets are swapped and path 0 is removed from the selected paths. This process does not increase the total distance of the path system.

**Observation 3.** *If the number of remaining selected paths is odd, there must exist a source-target corner.*

Eventually, we end up with a cycle with no source-target corners; if the cycle is made up of two paths, we use a logic that is similar to the logic we presented for path merging: we attempt to reduce total path length, if this fails, we attempt to isolate tokens, if this also fails, we reroute the path containing the cycle edge through the other path. If we end up with a cycle formed out of four or more paths (Figure 5.5), we refer to the remaining paths as a reduced set of paths. Let  $m$  be the cardinality of the reduced set of paths. We can generate two path systems induced on the reduced set of paths, which we call reduced path systems, such that they break the cycle. Assume we use the pair  $(s_i, t_i)$  to refer to the source and the target of path  $P_i$ :

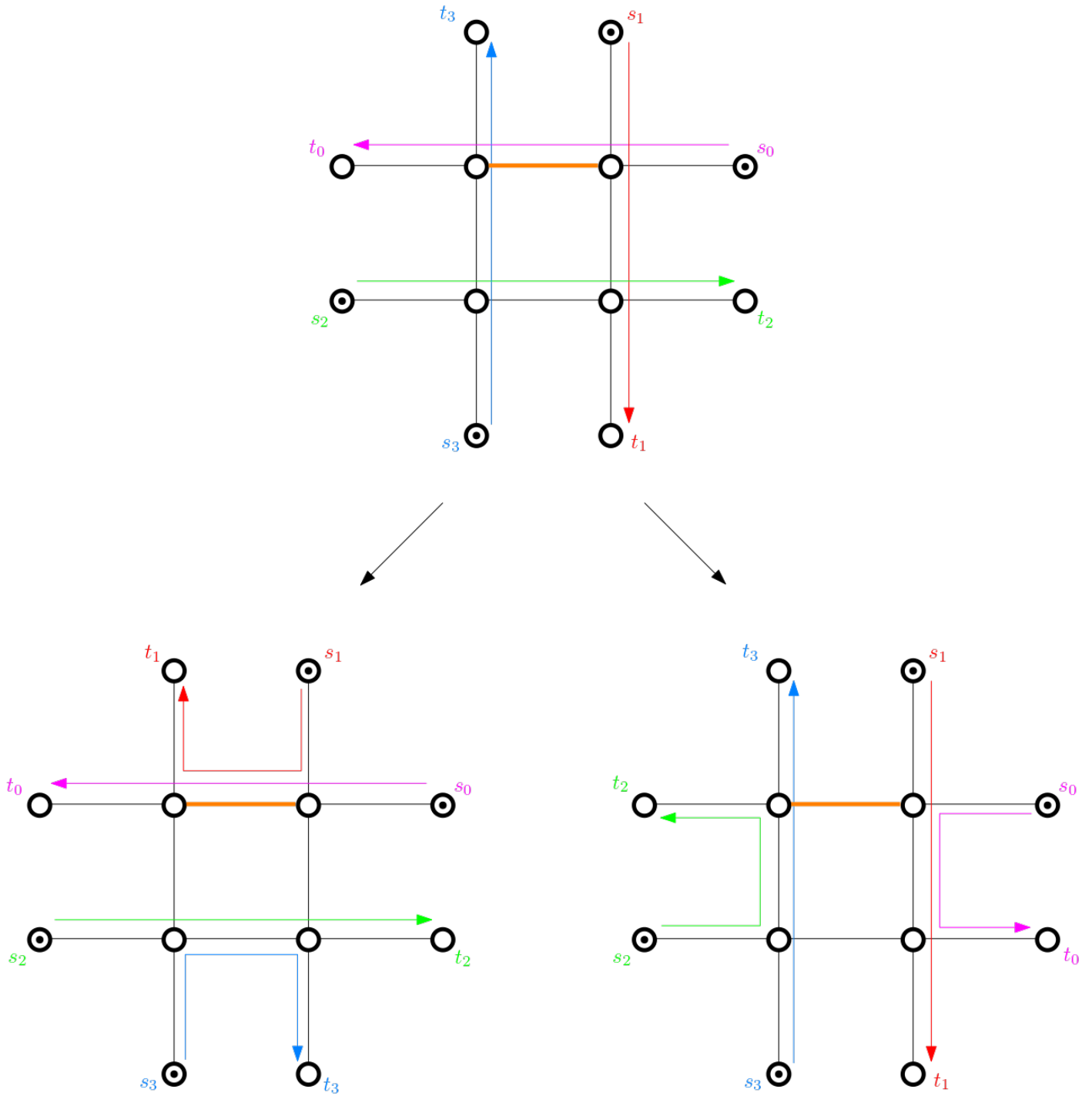


Figure 5.5: Nontrivial cycle breaking base case: no source-target corners found

1. Even reduced path system: match  $s_i$  with  $t_i$  if  $i$  is even,  $s_i$  with  $t_{(i+2)\%m}$  if  $i$  is odd (without using the edges unique to  $P_i$ , which are forbidden edges),  $0 \leq i \leq m - 1$ .
2. Odd reduced path system: match  $s_i$  with  $t_i$  if  $i$  is odd,  $s_i$  with  $t_{(i+2)\%m}$  if  $i$  is even (without using the edges unique to  $P_i$ , which are forbidden edges),  $0 \leq i \leq m - 1$ .

The fact that both reduced path systems break the cycle follows from the fact that the initial set of paths induces a single cycle, and from the existence of a unique edge in the cycle for every path: a subset of those unique edges are no longer part of any path in the reduced path systems. Also, when creating the new matching, we are also interested in the corresponding paths: those paths that avoid the forbidden edges are unique.

We would now like to update the selected paths using one of the two reduced path systems we constructed.

**Observation 4.** *The total distance of one of the two generated reduced path systems is less than or equal to that of the path system involving the selected paths.*

Observation 4 follows from the fact that the sum of the total distances of the reduced path systems is equal to twice the total distance of the original set of paths. If the total distance of the two reduced path systems are different, we pick the reduced path system with the smallest total distance and we update the paths accordingly. Otherwise, we pick the reduced path system that does not include the cycle edge. In the case of the example in Figure 5.5, the reduced path system on the right gets picked.

**5.2.1.4.3.5 Preserving the properties of the path system** It may be the case that the cycle breaking procedure gives rise to pairs of paths that are not merged, or paths that are wrapped: those are two possible consequences of the elimination of source-target corners. Therefore, every time a cycle is broken, we run the path merging and the path unwrapping procedures to reinstate the invariant, followed by repopulating the edge frequencies of the new path system. The path system being merged and unwrapped is what ensures the correctness of the theoretical work.

**5.2.1.4.3.6 Cycle breaking: the conclusion** We have yet to show that the procedure terminates: the proof can be found in Lemma 21. Once cycle breaking terminates, the path system graph is a forest.

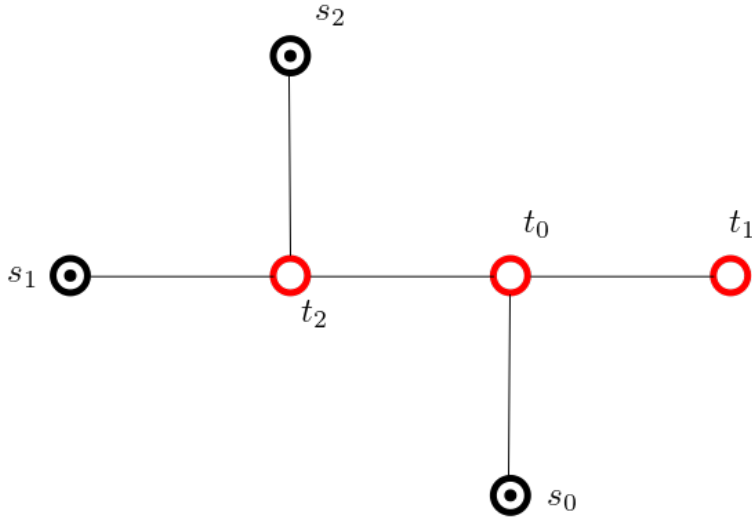


Figure 5.6: Instance where a greedy solution extracts/implants a token more than once.

### 5.2.1.5 Output generation

**5.2.1.5.1 Exact extraction/implantation forest solver** The exact extraction-implantation forest solver takes in a path system whose paths induce a forest and outputs a sequence of displacements that solve the token moving instance. It goes without saying that this solver can only be used if the ordering module is activated, as the ordering module is what ensures that the path system conforms to the expected input format of the exact extraction-implantation forest solver.

Before the solver is run, we attempt to reduce the total distance of the inputted path system by running APSP + MWPM + PR on the graph it induces. This is used as a safety net, and eliminates all pairs of paths that cross an edge in opposite directions (Lemma 22) without increasing (and while possibly decreasing) the total distance of the path system. Also, this does not increase the number of tokens we intended to displace, as this number is equal to the total number of tokens minus the isolated tokens, and the isolated tokens remain isolated, though we may isolate extra tokens by recomputing the matching. The path system graph, which is still a forest, is then passed, one tree at a time, to Călinescu's exact tree solver. The exact tree solver displaces all the tokens that were intended to be displaced once, and the total number of displacements it outputs is equal to the total distance of the path system that was inputted to it (Lemma 23).

**5.2.1.5.2 Greedy solver** The greedy solver takes in an arbitrary path system (it need not be unwrapped, merged, or cycle-free) and outputs a sequence of displacements that solve the token moving instance. This greedy solver is what we have been describing as the solver for MDTM covered in the Călinescu paper.

The algorithm looks at the graph induced on the provided path system and computes the shortest path between any source vertex and any target vertex; it also populates a matching, which initially consists of the source vertex-target vertex mapping in the provided path system. It then goes over paths in an arbitrary order and, for every path, it attempts to execute it. If the path can be executed, that is, if the token is not obstructed, then it is moved. Otherwise, there is some obstructing token: we retrieve the target of the first obstructing token, we modify the matching by setting the target of the current token to be that and the target of the obstructing token to be the old target of the current token, and we recurse on the obstructing token. The performance of greedy solving largely depends on the ordering of the paths to move, which is arbitrary. The ordering affects the number of tokens that end up being displaced, as well as the overall number of extraction/implantation operations. An example of the latter can be seen in Figure 5.6. The indices of the source vertices and the target vertices indicate the index of the path they are associated with, and since the graph is a tree, the corresponding paths can be inferred from the figure. There is a solution to this instance that extracts/implants every token once, for a total of 3 extraction/implantation operations: move token on  $s_1$  to  $t_1$ , move token on  $s_2$  to  $t_2$ , move token on  $s_0$  to  $t_0$ . Another ordering of the moves would lead to redundant extraction/implantation operations: move token on  $s_2$  to  $t_2$ , move token on  $s_0$  to  $t_0$ , attempt to move token on  $s_1$ : the first obstructing token is on  $t_2$ , and its target vertex is  $t_2$ : now, the target vertex of the token on  $s_1$  becomes  $t_2$ , and the target vertex of the token on  $t_2$ , which we attempt to move next, becomes  $t_1$ . When this recursive process is over, we end up moving the token on  $t_0$  to  $t_1$ , the token on  $t_2$  to  $t_0$ , and the token on  $s_1$  to  $t_2$ , for a total of 5 extraction/implantation operations. Those arbitrary choices made here are avoided when ordering then using the forest solver.

We also cover an example where the ordering of the paths affects the number of tokens the greedy solver displaces. In Figure 5.7, if the execution sequence of the paths picked by the greedy solver is  $P_0, P_2, P_1$ , the solution displaces all 3 tokens. Otherwise, if the path execution sequence is  $P_2, P_1, P_0$ , greedy solver displaces two tokens:  $P_2$  is not obstructed, so it is executed. As for  $P_1$ , its target vertex is obstructed by the token at  $s_0$ : after target swapping, the target of the token at  $s_1$  becomes  $t_0$ , and that of the token at  $s_0$  becomes  $t_1$ . We then attempt to move the token on  $s_0$ : it already occupies its target, so no displacements are executed. We now attempt to move the token on  $s_1$  to  $t_0$ . Assume that the shortest path between  $s_1$  and  $t_0$  goes through the green path in the figure. This move is obstructed by the token at  $t_2$ : the rest of the execution of the greedy solver will move the token on  $t_2$  to  $t_0$  and the token on  $s_1$  to  $t_2$ , in that order. In this scenario, the greedy solver moved 2 out of the 3 tokens.

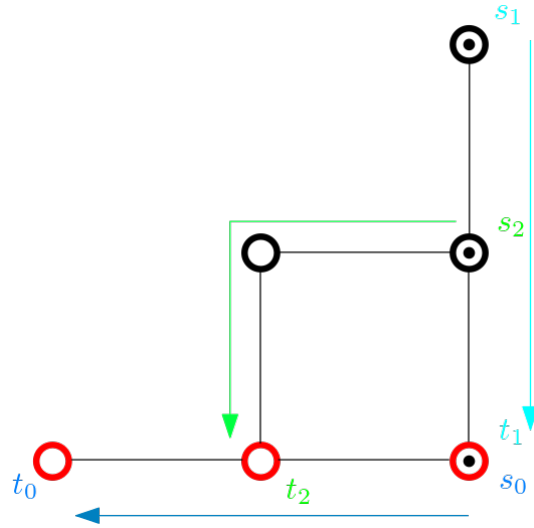


Figure 5.7: Instance where a greedy solution displaces a different number of tokens depending on the ordering of the paths.

### 5.2.1.6 Greedy token isolation

We want our algorithms to provide absolute advantages over the usage of solutions involving the greedy solver. Isolating the maximum number of tokens without increasing the total distance of the path system is a hard problem, because the path system may induce a grid, and we have already seen that MEITM does not have a polynomial time solution on grids. The token isolation that we integrate into the procedures described above (namely collision avoidance, path merging and cycle breaking) merely do locally optimal token isolation choices. Since greedy solving attempts to move tokens in an arbitrary order, in some cases, it may end up displacing fewer tokens than a solution generated by the exact extraction/implantation forest solver. We aim to eliminate this possibility, and we do that by greedily isolating tokens.

Greedy token isolation takes in a path system and greedily attempts to turn stationary tokens into isolated atoms. The removal attempt consists of removing the vertex from the induced graph, then attempting to compute a matching. If a matching that does not increase the total distance of the path system can be found, the token is isolated and eliminated from the path system, otherwise, we move on to the next token.

When greedy atom isolation is done, we guarantee that, in the resulting path system, all non-isolated tokens have to move. Therefore, this puts the greedy solver and the exact extraction/implantation forest solver on equal footing, with the latter having the extra advantage of being able to isolate even more tokens as the path system changes as a result of the ordering procedure.

Greedy atom isolation is run every time the path system is updated. It



will therefore not be added to the pseudocode of the algorithms for the sake of readability.

### 5.2.2 *HUNGARIAN-REDIST*

---

**Algorithm 12** HUNGARIAN-REDIST

---

**Input:** A 2D matrix representing a source configuration and a 2D matrix representing a target configuration

**Output:** A sequence of batches that solves the token moving instance

- 1: Unweighted APSP
  - 2: MWPM + PR
  - 3: Remove independent vertical paths
  - 4: **if** Moves are to be ordered **then**
  - 5:     Merge path system
  - 6:     Unwrap path system
  - 7:     Break cycles in the path system
  - 8:     Use exact extraction/implantation forest solver
  - 9: **else**
  - 10:     Greedy solve
  - 11: **end if**
  - 12: Solve unsolved columns, if any
- 

HUNGARIAN-REDIST was designed with the intent of being a generalized version of REDREC v2.0. That is, HUNGARIAN-REDIST follows the same logic of ensuring that all columns can be solved independently of each other by moving tokens in a way that gives all columns a positive surplus, before proceeding to use exact path solvers to solve columns independently.

In its pseudocode, everything has previously been covered as part of the modules, with the exception of the removal of independent vertical paths. An independent vertical path is a vertical path that does not share its source vertex or its target vertex with any other path in the path system. Since the point of the algorithm is to modify surpluses before independently solving columns, executing an independent path does not affect values of surpluses, therefore, such paths are ignored, and moving the concerned tokens to their target in the same column is handled in the last step of the algorithm.

We indicate that the output is in the form of batches because of the usage of path solvers as a subroutine, which are able to batch the output. All batches outputted before path solving, which takes place at the end of the algorithm, consist of a single token.

---

**Algorithm 13** HUNGARIAN-COLAV

---

**Input:** A 2D matrix representing a source configuration and a 2D matrix representing a target configuration

**Output:** A sequence of displacements that solves the token moving instance

```
1: if Use weights is set to true then
2:   Weighted APSP
3: else
4:   Unweighted APSP
5: end if
6: MWPM + PR
7: if Use weights is not set to true then
8:   Apply any of the three COLAV techniques
9: end if
10: if Moves are to be ordered then
11:   Merge path system
12:   Unwrap path system
13:   Break cycles in the path system
14:   Use exact extraction/implantation forest solver
15: else
16:   Greedy solve
17: end if
```

---

### 5.2.3 *HUNGARIAN-COLAV*

Unlike HUNGARIAN-REDIST, HUNGARIAN-COLAV was not inspired by the REDREC algorithms, but rather, by the overall displacement operations versus overall extraction/implantation operations tradeoff that we have recurrently alluded to in this thesis. The instance will be directly solved as a result of using either the greedy solver or the exact extraction/implantation forest solver.

In the pseudocode of HUNGARIAN-COLAV, everything has been covered except the usage of a weighted grid graph. The intuition behind making the grid graph weighted is to simulate what the COLAV techniques are doing. In the original configuration, edges with no tokens on their endpoints, edges with a single token on their endpoints and edges with two tokens on their endpoints are given three different weights, with the weight of an edge increasing with the number of tokens on its endpoints. This makes it so that, in the computation of the all-pairs shortest paths, the shortest paths are inclined to bypass tokens, which should in theory reduce the number of tokens to displace at the cost of extra displacement operations.

In HUNGARIAN-COLAV, batching is not brought up because every token is moved separately by design, so the output is a sequence of displacements.

# CHAPTER 6

## TOKEN MOVING ON PATHS ON THE GPU

The structure of this chapter is akin to that of Chapter 4. The first section will cover parallelization techniques for the token moving algorithms, whereas the second section will cover parallelization techniques for the token-vertex matching algorithms.

We include information that is relevant to all sections that cover parallel code. We use syntax that is identical to the syntax that is used in CUDA programming. In particular, angular brackets (“<<< >>>”) is a decorator used in the function call of a kernel to indicate its execution parameters, namely the number of thread blocks and the number of threads per block, in that order. Function calls that do not use angular brackets are device function calls. We omit declaring intermediate variables for the sake of readability: variables that are first mentioned within a kernel are assumed to be allocated in shared memory (unless stated otherwise), and all other undeclared variables are assumed to be allocated in device memory. Finally, variables that are computable from the input, notably  $K$  and  $K'$ , are precomputed and are used directly in the pseudocode despite not explicitly being part of the input to reduce clutter.

### 6.1 The token moving procedure

We parallelize the three token moving modes: unbatched token moving, batched token moving, and block batched token moving. Given the relatively small size of the instances we are dealing with, the parallel implementations of the token moving modes make use of a single thread block, as the extra parallelism we can leverage by using multiple blocks does not compensate for the added cost of synchronization across blocks. Therefore, in pseudocode, the kernel that will be shown is a single-block kernel.

In the kernel for parallel unbatched token moving presented in Algorithm 14,

---

**Algorithm 14** Parallel unbatched token mover

---

**Input:** An executable token-vertex matching,  $matching\_src$  and  $matching\_dst$

**Output:** A sequence of displacements,  $disp\_src$  and  $disp\_dst$

- 1: compute\_left\_moving\_distances( $matching\_src, matching\_dst,$   
     $left\_distances$ )
  - 2: compute\_right\_moving\_distances( $matching\_src, matching\_dst,$   
     $right\_distances$ )
  - 3: **synchronize threads**
  - 4: exclusive\_sum( $left\_distances, excl\_sum\_left\_distances$ )
  - 5: exclusive\_sum( $right\_distances, excl\_sum\_right\_distances$ )
  - 6: **if**  $threadIdx.x == 0$  **then**
  - 7:      $num\_displacements\_left \leftarrow excl\_sum\_left\_distances[K' - 1]$   
        $+left\_distances[K' - 1]$
  - 8:      $num\_displacements\_right \leftarrow excl\_sum\_right\_distances[K' - 1]$   
        $+right\_distances[K' - 1]$
  - 9: **end if**
  - 10: **synchronize threads**
  - 11: generate\_unbatched\_output( $matching\_src, matching\_dst, left\_distance,$   
     $excl\_sum\_left\_distances, offset = 0$ )
  - 12: generate\_unbatched\_output( $matching\_src, matching\_dst, right\_distance,$   
     $excl\_sum\_right\_distances, offset = num\_displacements\_left$ )
-

we assign one thread per token-vertex pair in the matching, and every thread will be in charge of outputting the displacements of its token-vertex pair. Every thread computes the distance between its token and its target vertex, and the output is written either to *left\_distances* or *right\_distances*, depending on whether the token the thread is assigned to is left-moving or right-moving. We make this distinction based on direction because left-moving tokens need to be displaced separately from right-moving tokens. After we are done computing distances, we run an exclusive sum on *left\_distances* and *right\_distances* separately. The purpose of the exclusive sum is for every thread to figure out the memory location it should write to in the output. Given that the exclusive sum is computed for every direction separately, this allows every thread to know its output offset within the portion of the output that moves tokens in the same direction. For the implementation of exclusive sum, we use the Kogge-Stone scan algorithm [32]. Now that we have the output offsets for both directions, we are ready to generate the output: every thread generates the displacements of its own token-vertex pair at the memory location designated by the exclusive sum we obtained. Since left displacements take place before right displacements, pre-computing the total number of left displacements, which is the offset to be added to right displacements, makes it possible for us to omit synchronizing threads between output generation for left-moving tokens and that for right-moving tokens. In output generation, every thread iterates *dist* times, where *dist* is the distance between the token and the target vertex in the pair assigned to the thread, and outputs a single displacement at every iteration.

In the kernel for parallel batched token moving presented in Algorithm 15, we assign one thread per token-vertex pair, as was the case for unbatched token moving. Since batches need not involve displacements in a unique direction, we can compute distances for all token-vertex pairs without having to worry about directions. Once the distances have been computed, we are interested in computing the number of batches, which is the maximum distance between a token and its target vertex in the matching. We do that using a standard reduction [32]. Using that same reduction, we can also compute the total displacement, which is equal to the sum of the aforementioned distances. The final step is generating the output: given that we do not care about order of displacements within a batch, every thread iterates *dist* times (up to a maximum of *num\_batches* times) and, at every iteration, it atomically increments a variable that indicates the index where the output should be written, with all the threads being synchronized at the end of every batch/iteration. At the beginning of every iteration, the number of outputted displacements so far is written to the *batchPtr* array by the thread at index 0.

---

**Algorithm 15** Parallel batched token mover

---

**Input:** An executable token-vertex matching,  $matching\_src$  and  $matching\_dst$

**Output:** A sequence of displacements,  $disp\_src$  and  $disp\_dst$ , batch pointers,  $batchPtr$

- 1:  $compute\_distances(matching\_src, matching\_dst, distances)$
  - 2: **synchronize threads**
  - 3:  $reduce(distances, num\_batches, num\_displacements)$
  - 4: **synchronize threads**
  - 5:  $generate\_batched\_output(matching\_src, matching\_dst, distances, num\_batches)$
- 

As is the case for the previous kernels, the kernel presented in Algorithm 16 assigns one thread per token-vertex pair. We start by computing the direction of every token-vertex pair, as well as representatives for each direction.  $directions[i]$  may take on 3 values: 1 if the token in the  $i^{th}$  token-vertex pair is right-moving, 0 if the token in the  $i^{th}$  token-vertex pair is non-moving, and  $-1$  if the token in the  $i^{th}$  token-vertex pair is left-moving. We also compute direction representatives: a direction representative is the thread index of the thread used to access the earliest token-vertex pair corresponding to that direction. We may therefore have up to two direction representatives. We then detect heads, using the same logic as the one we used in the serial version of this algorithm. By convention, we explicitly specify that the leftmost token is not a block head in order to make the next step of the algorithm work. Once all heads are detected and threads synchronized, we run a pair of prefix sums; for the right-moving blocks where the heads are the leftmost tokens of blocks, we run an inclusive prefix sum (the previous statement makes this work). For the left-moving blocks where the heads are the rightmost tokens, we run an exclusive prefix sum, and we end up with the block index of every token. With this information, we can compute the head and the tail of every block in a way that is very similar to the computation of the values in  $is\_head$ . Before output generation, we want to make it possible for us to compute the output for left-moving token blocks and right-moving token blocks at the same time, so we compute the number of right-moving token block head displacements and the corresponding number of batches, which we use as offset for left-moving blocks in the  $head\_src/head\_dst$  and the  $batchPtr$  arrays respectively. We also compute the number of batches for left-moving token blocks, as this is something we need for output generation. The number of right-moving token block head displacements is equal to the sum of the distances between a right-moving tail and its target vertex, whereas corresponding number of batches is equal to the maximum distance between a right-moving tail and its target vertex. The total number of batches for left-moving blocks is defined similarly.

---

**Algorithm 16** Parallel block batched token mover

---

**Input:** An executable token-vertex matching,  $matching\_src$  and  $matching\_dst$

**Output:** A sequence of token blocks to displace represented by token block head displacements,  $head\_src$  and  $head\_dst$ , the sizes of the blocks to displace,  $size\_block$ , block batch pointers,  $batchPtr$

- 1:  $populate\_directions\_and\_representatives(matching\_src, matching\_dst, directions, representatives)$
  - 2: **synchronize threads**
  - 3:  $detect\_heads(matching\_src, matching\_dst, is\_head\_left, is\_head\_right)$
  - 4: **synchronize threads**
  - 5:  $inclusive\_sum(is\_head\_right, index\_to\_block\_right)$
  - 6:  $exclusive\_sum(is\_head\_left, index\_to\_block\_left)$
  - 7: **synchronize threads**
  - 8:  $map\_block\_index\_to\_head\_and\_tail(direction, index\_to\_block\_right, index\_to\_block\_left, block\_index\_to\_head\_right, block\_index\_to\_head\_left, block\_index\_to\_tail\_right, block\_index\_to\_tail\_left, is\_tail\_right)$
  - 9: **synchronize threads**
  - 10:  $compute\_offsets(index\_to\_block\_right, is\_tail\_right, direction, matching\_src, matching\_dst, num\_displacements\_right, num\_batches\_right, num\_batches\_left)$
  - 11: **synchronize threads**
  - 12:  $generate\_block\_output(matching\_src, matching\_dst, direction, index\_to\_block\_right, block\_index\_to\_head\_right, block\_index\_to\_tail\_right, index\_to\_block\_left, block\_index\_to\_head\_left, block\_index\_to\_tail\_left, num\_displacements\_right, num\_batches\_right)$
- 

In the output generation function, every thread loops a total of  $max(num\_batches\_left, num\_batches\_right)$  times. In every iteration, the threads acting as direction representatives write the batch pointers (i.e. the number of outputted block head displacements in their direction so far) for their respective directions in the  $batchPtr$  array (the left direction representative is offset by  $num\_batches\_right$ ). Subsequently, every thread gets the direction and the block index of the token-vertex pair it is in charge of and checks whether the token in question is a head. If it is, it writes the token block displacement to the output, along with the size of the block (which we know because we know both the head and the tail of the block). All threads then update the positions of the tokens in their token-vertex pair. The final step in the iteration entails updating the heads of every block.

## 6.2 The token-vertex matching procedure

We now look at the parallel counterparts of the serial token-vertex matching algorithms from Chapter 4.

### 6.2.1 *The Hungarian-based token-vertex matcher*

The code for the parallel Hungarian-based path solver (Algorithm 17) consists of 3 distinct kernel functions. The Hungarian kernel we are using has been developed by Lopes et al. [33] and is loosely based on Munkres' cubic-time serial algorithm for assignment [34]. It requires the input array to be of size a power of two; to that end, we introduce *hungarian\_N* and we set it to be equal to the smallest power of 2 that is larger than or equal to  $K$ . The function *construct\_hungarian\_matrix* computes the cost matrix to be used as input for the Hungarian, and it starts by compressing. After compression, every thread is assigned a target vertex and is in charge of computing the distances between all tokens and the target vertex is it assigned to: every threads loops over the values in the compressed source. Some of the threads will be assigned to bogus vertices; those threads populate the matrix with a cost of  $N$ , and some threads are in charge of the expanded columns, where the expanded columns are columns that had to be added to the Hungarian cost matrix to make it conform to the expected input of the Hungarian solver kernel; such columns are filled with infinities except at the diagonal, which are filled with 0s. Once the cost matrix is populated, the next step consists of running the parallel Hungarian solver. The Hungarian solver makes use of dynamic parallelism, which explains why it is configured to use one thread block with a single thread. As was the case in the serial implementation, the matching returned by the Hungarian solver has sorted targets, so to ensure the matching is executable, we sort *matching\_src* to get rid of crossings, and we make use of the standard parallel implementation of radix sort to do that [35].



---

**Algorithm 17** Parallel Hungarian-based token-vertex matcher

---

**Input:** Two bit arrays representing the starting and ending configurations, *source* and *target*,  $K$  and  $K'$ , the number of source tokens and the cardinality of  $T$  respectively

**Output:** A distance-minimizing token-vertex matching with no crossings, *matching\_src* and *matching\_dst*

- 1: `construct_hungarian_matrix` $\lll 1, \text{hungarian\_}N \ggg$  (*source*, *target*,  
*hungarian\_to\_source\_idx*, *hungarian\_to\_target\_idx*,  
*hungarian\_mat*)
  - 2: **synchronize device**
  - 3: `hungarian` $\lll 1, 1 \ggg$  (*hungarian\_mat*, *matching\_src*, *matching\_dst*,  
*hungarian\_to\_source\_idx*, *hungarian\_to\_target\_idx*,  $K$ ,  $K'$ )
  - 4: **synchronize device**
  - 5: `radix_sort` $\lll 1, \text{matching\_src.size} \ggg$  (*matching\_src*)
- 

Now, we would like to introduce the mechanism we used binary search for in Algorithm 6: controlling the number of batches. Since binary search is inherently serial and would require a total of  $\log_2(N)$  calls to the Hungarian solver in the worst case, we would need to look for an alternative that would allow us to make use of parallelism, especially knowing that the streaming multiprocessors are severely underutilized even if we account for the dynamic parallelism that is occurring within the Hungarian solver. We make use of a variant of binary search that is better suited for parallel computing:  $k$ -ary search.  $k$ -ary search reduces the number of serial calls to the Hungarian solver from  $\log_2(N)$  in the worst case to  $\log_k(N)$  in the worst case.

$k$ -ary search allows us to hone in on the minimum number of batches we can use by launching  $k$  Hungarian solver kernels at once, where every solver takes in a cost matrix with a different bound on the maximum distance. The values of the bounds on distance we check are initially separated by the length of the search space (initially equal to  $N$ ) divided by  $k$ : we call this value the gap. At every iteration, we look for the largest value of  $i$  such that `opt_bounded_distance`[ $i$ ] is larger than `min_total_distance`: our search space is now bounded between  $i$  and  $i + m$  where  $m$  is the size of the old search space divided by  $k$ . As the search space gets reduced by a factor of  $k$ , so does the gap. In code, we use the term *granularity* to refer to the value of  $k$ .

We now clarify what the code in Algorithm 18 is doing line by line. We start by running the standard parallel Hungarian-based token-vertex matcher, which allows us to compute `min_total_distance` as well as how `hungarian_mat` looks like (we omit the latter from the arguments of the function for conciseness). We then compute the  $k$  values of  $i$  to be used as bounds on the maximum distance. Once we have those values, we duplicate the original Hungarian cost matrix  $k$  times into the `hungarian_mats` array and we update the cost values within each

---

**Algorithm 18** Parallel Hungarian-based token-vertex matcher with batch number minimization

---

**Input:** Two bit arrays representing the starting and ending configurations, *source* and *target*,  $K$  and  $K'$ , the number of source tokens and the cardinality of  $T$  respectively

**Output:** A distance-minimizing, batch-minimizing token-vertex matching with no crossings, *matching\_src* and *matching\_dst*.

```

1:  $l \leftarrow 1$ 
2:  $r \leftarrow N$ 
3:  $sol\_space\_size \leftarrow N$ 
4:  $offset \leftarrow 0$ 
5:  $min\_total\_distance \leftarrow$ 
    parallel_hungarian_based_t-v_matcher(source, target,  $K$ ,  $K'$ )
6: while  $sol\_space\_size > 0$  do
7:   values_of_i_to_check<<< 1, granularity >>> (i_to_check,
    offset,  $sol\_space\_size$ )
8:   synchronize device
9:   duplicate_hungarian_matrix<<< granularity,
    hungarian_N >>> (hungarian_mat, hungarian_mats)
10:  synchronize device
11:  bound_hungarian_mats<<< granularity,
    hungarian_N >>> (hungarian_mats, i_to_check)
12:  synchronize device
13:  hungarian<<< granularity,
    1 >>> (hungarian_mats, matchings_src, matchings_dst,
    hungarian_to_source_idx, hungarian_to_target_idx,  $K$ ,  $K'$ )
14:  synchronize device
15:  check_valid_matchings_and_update_offset<<< granularity,  $K'$  >>>
    (hungarian_mats, matchings_src, matchings_dst,
     $min\_total\_distance$ , i_to_check, offset)
16:  synchronize device
17:   $sol\_space\_size \leftarrow sol\_space\_size/granularity$ 
18: end while
19: increment_offset<<< 1, 1 >>> (offset)
20: synchronize device
21: generate_matching<<< 1,  $K'$  >>> (matchings_src, matchings_dst, offset,
    matching_src, matching_dst)
22: synchronize device
23: radix_sort<<< 1, matching_src.size >>> (matching_src)
24: synchronize device

```

---

one of those  $k$  copies to account for the presence of a distance bound. Every copy and update is handled by one thread block, and copying/updating works the same way as constructing a Hungarian cost matrix from scratch, i.e. every thread takes care of computing costs for a single target vertex. The next step is launching  $k$  kernels, each of which computes a matching for one of the  $k$  generated cost matrices. The final step is figuring out the largest value of  $i$  that does not yield a valid matching, and setting *offset* to be equal to that value. Once we are out of the for loop, we increment *offset* by 1, which makes it equal to the smallest value of  $i$  such that  $opt\_bounded\_distance[i]$  is equal to  $min\_total\_distance$ , and we write the corresponding token-vertex matching to the output, and we remove the crossings from it by sorting the source vertices.

The correctness of this algorithm follows from that of  $k$ -ary search.

### 6.2.2 *The Greedy token-vertex matcher*

The parallel version of greedy token-vertex matching is relatively simple. In the serial version of that same algorithm (Algorithm 8), the input is compressed, the cost matrix is constructed, the matching is greedily generated, and the outputs of the greedy matching function are sorted. The parallel version of the algorithm preserves this structure and turns every one of those functions into a kernel call on a single block. We have already covered compression, cost matrix construction, and radix sorting in parallel, so we still have to describe the implementation of `parallel_greedy_matching`, the parallel implementation of `greedy_matching`. The implementation is similar to Algorithm 7, so a description of the changes should be sufficient for the reconstruction of the algorithm. We assign one thread per target vertex, and every thread chooses its row iteratively. Once all threads are done choosing their row, we synchronize the threads. Naturally, it may be the case that two columns chose the same row, since the choices are being made in parallel, so we use a lock for every row; whichever thread atomically increments the value of the lock first gets matched to the row and gets to add the pair to the matching, whereas the other columns that were competing for the same row will have to try to find a match again in the next iteration. Eventually, all columns get matched, though some rows may remain unmatched (since  $|TK| \geq |T|$ ); the unmatched tokens are matched to the vertex they occupy.

### 6.2.3 *The Linear Exact token-vertex matcher*

In the serial version of the Linear Exact path solver, we implemented the linear-time algorithm that is described in the Karp paper. However, this algorithm is not amenable to parallelization, as it is inherently sequential; it requires the computation of more than 10 arrays that have sequential dependencies. Therefore, for the parallel version, we implement the quadratic version of the algorithm

present in the same paper, which makes it possible for us to leverage well-known parallelism paradigms.

---

**Algorithm 19** Parallel Linear Exact token-vertex matcher

---

**Input:** Two bit arrays representing the starting and ending configurations, *source* and *target*,  $K$  and  $K'$ , the number of source tokens and the cardinality of  $T$  respectively

**Output:** A distance-minimizing token-vertex matching with no crossings, *matching\_src* and *matching\_dst*.

```

1: compress_input(source, source_compressed)
2: compress_input(target, target_compressed)
3: synchronize threads
4: inclusive_sum(source, num_sources_inclusive)
5: inclusive_sum(target, num_targets_inclusive)
6: synchronize threads
7: compute_heights(num_sources_inclusive, num_targets_inclusive,  $H$ )
8: synchronize threads
9: compute_profits(source_compressed,  $H$ ,  $e$ ,  $P$ )
10: synchronize threads
11: compute_max_profits_per_level(source_compressed,  $H$ ,  $P$ ,  $e$ ,  $pi$ )
12: synchronize threads
13: compute_to_be_removed(source_compressed,  $H$ ,  $P$ ,  $e$ ,  $pi$ ,
    to_be_removed_at_level, to_be_removed)
14: synchronize threads
15: compute_matching(source_compressed, target_compressed,
    to_be_removed, matching_src, matching_dst)
16: synchronize threads
17: radix_sort(matching_src)
18: synchronize threads

```

---

In the case of Parallel Linear Exact token-vertex matching, the matching can be computed within a single kernel, so the code presented in Algorithm 19 is kernel code. The terms used in the pseudocode are mentioned as is in the Karp paper, so the corresponding formulas can be found there, and our focus will be on thread assignment and a high-level description of the work of every thread at every device function call.

For this kernel, we use a total of  $N$  threads. We start by doing a standard compression of the input. Next, we compute heights at  $j$  ( $0 \leq j < N$ ) which is equal to the number of sources up to and including  $j$  minus the number of targets up to and excluding  $j$ , so  $H[j]$  is computed by thread index  $j$  which does a linear amount of work. The next step consists of computing profits; profits are computed for source vertices only, so we assign 1 thread per source vertex (which means that  $N - K$  of the threads are idle at this point). The profit of source  $i$  is a function

of its height and the heights  $H[j]$  ( $j > i$ ), so each thread loops at most  $N$  times to compute the profit of its associated source vertex. The Karp quadratic linear assignment algorithm determines which source vertices are to be omitted from the eventual matching, and the way it does so is by computing the highest profit per positive level, and then, for every level, finding the leftmost source vertex with a profit equal to the highest profit in the corresponding level, where a level consists of all the source vertices with a given height. Given the way the height is computed, there must exist a source vertex having height  $i$  for every value of  $i$  between 1 and  $K - K'$  (both inclusive). We therefore compute the maximum profits per level for all positive levels, and we do that by assigning one thread per source vertex to retrieve height. The maximum profit at the corresponding level is then updated atomically. The last step is selecting a single source vertex from every level to eliminate, and this is done in a similar fashion to the earlier step: every thread retrieves the height of its source vertex and compares its profit to the maximum profit at the corresponding level, if it is equal to it, we atomically minimize the index of the source vertex to be eliminated at that level. Once the work of all threads is done, we copy the results over to the *to\_be\_removed* array, which is a boolean array (*to\_be\_removed*[ $i$ ] being 1 means that vertex  $i$  should be excluded from the token-vertex matching). The final two device functions are straightforward.

#### 6.2.4 *The Bruteforce token-vertex matcher*

Our parallelization technique for the Bruteforce token-vertex matcher (Algorithm 20) entails computing the total distance of every split in parallel (i.e. every block takes care of computing the total distance of one split). As such, the code had to be split into multiple kernels. Clearly, the first two kernels and the last two kernels can be combined, though the pseudocode decouples them into multiple kernels for the sake of readability. The algorithm starts with a regular compression, followed by the computation of the number of splits. Calculating the deficit (which can be done using  $K'$  threads, if every thread gets assigned a single target vertex) is not sufficient to figure out the number of splits, as the number of splits will also depend on the distribution of the tokens outside the target region, and that explains why we are using  $N$  threads. Once we have the number of splits, we launch that many blocks to compute total distances. The index of the block will make it possible to determine the tokens involved in the split using the *source\_compressed* array, and every thread will take care of computing the distance of a single token-vertex pair in the split. The total distance of a given split is computed using atomic adds. Once the total distances of all possible splits are computed, the next step is figuring out what split to go for, which we do using a standard argmin reduction. The final step is computing the matching, which can easily be done since we already have the selected split, i.e. the index of the leftmost source vertex in the *source\_compressed* array.

---

**Algorithm 20** Parallel Bruteforce token-vertex matcher

---

**Input:** Two bit arrays representing the starting and ending configurations, source and target,  $K$  and  $K'$ , the number of source tokens and the cardinality of  $T$  respectively,  $K_1$  and  $K_2$ , the number of vacant vertices to the left and to the right of the target region

**Output:** A distance-minimizing token-vertex matching with no crossings,  $matching\_src$  and  $matching\_dst$

- 1:  $distance\_of\_splits[K' + 1]$
  - 2:  $compressed\_source[K]$
  - 3:  $compress\_input\langle\langle\langle 1, N \rangle\rangle\rangle (source, source\_compressed)$
  - 4: **synchronize device**
  - 5:  $compute\_target\_deficit\_and\_num\_splits\langle\langle\langle 1,$   
 $N \rangle\rangle\rangle (source, target, deficit, num\_splits)$
  - 6: **synchronize device**
  - 7:  $compute\_distances\langle\langle\langle num\_splits,$   
 $K' \rangle\rangle\rangle (source\_compressed, distance\_of\_splits)$
  - 8: **synchronize device**
  - 9:  $find\_min\_distance\_split\langle\langle\langle 1,$   
 $num\_splits \rangle\rangle\rangle (distances\_of\_splits, selected\_split)$
  - 10: **synchronize device**
  - 11:  $compute\_matching\langle\langle\langle 1, K' \rangle\rangle\rangle (source\_compressed, matching\_src,$   
 $matching\_dst, selected\_split)$
-

## CHAPTER 7

# TOKEN MOVING ON GRIDS ON THE GPU

In this chapter, we discuss naive parallelization techniques for REDREC v2.0, and we introduce a variation of REDREC v2.0 that is more amenable to parallelization: REDREC v2.1.

### 7.1 REDREC v2.0

REDREC v2.0 as described in Section 5.1 is heavily serial by design. The naive parallelization approach for REDREC v2.0 involves parallelizing the presented functions separately, and synchronizing the device between them. For most functions, the kernel launch overhead makes parallelization a bad choice as parallel computation is negligible, and execution ends up slowing down. However, parallelization is a good idea when it comes to solving multiple token moving instances on paths at once. This is the case for partial path solving, solving columns with a surplus of 0 at the beginning of the algorithm, and calculating the total distance of every reservoir split: each of these three scenarios involve solving multiple token moving instances on paths: we assign a thread block for every instance. Since the grid instances are relatively small in size, the number of blocks and the number of threads per block is limited, which means that SM occupancy is not something we should worry about yet.

Practically, parallelizing the computation of the solution of multiple token moving instances on paths yields local improvements, though such improvements are negligible and are unable to compensate for the slowdown caused by the synchronization overhead, which is considerable, considering the amount of serialization that still exists, specifically between the iterations of the main loop: even with parallelization, the number of iterations to be executed is linear in the width of the grid in the worst case.

## 7.2 REDREC v2.1

REDREC v2.1 tackles the issue of excessive serialization that REDREC v2.0 suffers from. Without any major changes in the logic of the algorithm, we are looking to reduce the number of iterations of the main loop. The most obvious modification involves running the redistribution-reconfiguration-shuffling procedure for multiple donor-receiver pairs at once (up to  $k$ , where  $k$  is a parameter). This reduces the number of iterations of the main loop by a factor of  $k$ , and, if  $k$  is chosen carefully, a speedup with a comparable factor should be achieved. More specifically, given the function we aim to optimize for in the donor-receiver pair selection procedure, we compute its value for all pairs of adjacent donor-receiver columns, and we sort those pairs of columns accordingly. We then proceed to pick the top  $k$  mutually exclusive pairs of columns with regards to the metric of interest. If there are less than  $k$  mutually exclusive adjacent donor-receiver columns, we select all of them. We then proceed with the rest of the iteration and we do the work for all pairs at once.

Naturally, minor modifications beyond what was explained above need to be implemented. Every pair needs to know where to write its output to; this is handled with a scan every time displacements need to be written to the output.

The downside of REDREC v2.1 is that we do not have the means to anticipate the effect it will have on loss minimization (and subsequently, success maximization) in comparison to REDREC v2.0. This will be covered in more detail in the experimental section.



# CHAPTER 8

## PROOFS

This chapter covers the proofs of all the claims that were made in the previous sections and that went into the design of our algorithms.

**Lemma 1.** *The atom assembly experiment that relies on the RTFC eventually terminates.*

*Proof.* Assume that the experiment does not terminate. That is, assume that we run into case 3 an infinite number of times. Given this assumption, loss had to stop at some point, otherwise, case 2 would have been triggered. Consider the last run  $i$  with loss, and consider the first run  $i + 1$  after that: run  $i + 1$  does not have any loss, meaning that  $T''_{i+1} = \emptyset$ , and since we know that, before loss,  $T'_{i+1} \supseteq T_{i+1}$  by design of the problem solving module,  $T'_{i+1} \setminus T''_{i+1} \supseteq T_{i+1}$  too, which corresponds to case 1. The experiment should have therefore terminated at the end of run  $i + 1$ , contradicting our original assumption.  $\square$

**Lemma 2.** *A solution for the MDTM problem on a path moves at most  $|T|$  tokens, where  $T$  is the target region.*

*Proof.* Consider a solution for MDTM on a path where more than  $|T|$  tokens are moved. There must exist a token  $t_i$  that is moved and that does not end up on a trap in  $T$ . We have two cases: if the token does not obstruct any other token, then we can keep it in place and the solution is unaffected, therefore contradicting the fact that the solution minimizes displacement. Otherwise,  $t_i$  is moved to make the move of token  $t_{i+1}$  possible. Consider the sequence of sequentially dependent tokens  $t_i, t_{i+1}, \dots, t_{i+k}$  where  $t_{i+1}$  is said to be dependent on  $t_i$  if  $t_i$  obstructs it. Those tokens have to be moving in the same direction. Without loss of generality, assume that those tokens are left-moving; a symmetrical argument can be made for when the tokens are right-moving. This sequence is clearly finite. In the original mapping  $M$ , those tokens are mapped to vertices  $M^{-1}(t_i), M^{-1}(t_{i+1}), \dots, M^{-1}(t_{i+k})$ . We can reconstruct an alternative mapping  $M'$  as follows:  $t_i$  is mapped to  $M^{-1}(t_{i+1})$ ,

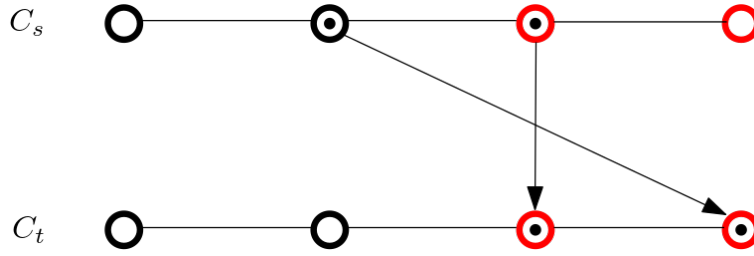


Figure 8.1: Instance where a token-vertex matching with the minimum total distance is not executable.

$t_{i+1}$  is mapped to  $M^{-1}(t_{i+2})$ ,  $\dots$   $t_{i+k-1}$  is mapped to  $M^{-1}(t_{i+1})$ ; the mapping of all the other tokens in  $M$  that were not mentioned are preserved.  $t_{i+k}$  is dropped out of the mapping. We now prove that this does not increase the distance of the mapping, and that it is still executable. No token changes direction, because if that were the case, this would contradict the fact that the tokens were sequentially dependent. Also, since the initial mapping was part of an executable matching, we have:  $idx(t_i) < idx(t_{i+1}) < \dots < idx(t_{i+k})$  and  $idx(M^{-1}(t_i)) < idx(M^{-1}(t_{i+1})) < \dots < idx(M^{-1}(t_{i+k}))$  (otherwise, the matching would have a crossing). Given there are no changes in direction, we also have:  $idx(M^{-1}(t_i)) < idx(M^{-1}(t_{i+1})) \leq idx(t_i)$ ,  $idx(M^{-1}(t_{i+1})) < idx(M^{-1}(t_{i+2})) \leq idx(t_{i+1})$ ,  $\dots$   $idx(M^{-1}(t_{i+k-1})) < idx(M^{-1}(t_{i+k})) \leq idx(t_{i+k-1})$ . From that, we get:  $d(t_i, M^{-1}(t_{i+1})) < d(t_i, M^{-1}(t_i))$ ,  $d(t_{i+1}, M^{-1}(t_{i+2})) < d(t_{i+1}, M^{-1}(t_{i+1}))$ ,  $\dots$   $d(t_{i+k-1}, M^{-1}(t_{i+k})) < d(t_{i+k-1}, M^{-1}(t_{i+k-1}))$ . Every vertex in  $T$  maps to some token in  $TK$ , no crossings were added, and we decreased the distance of at least one token, and therefore, for the matching itself. We have a contradiction. This concludes the proof.  $\square$

**Lemma 3.** *If a token-vertex matching in a path has the minimum total distance across all possible token-vertex matchings and has no crossings, it is executable by all three output modes and the total displacement required for its execution is equal to the total distance of the matching.*

*Proof.* We show that minimizing total distance is not sufficient.

The token-vertex matching depicted in Figure 8.1 minimizes total distance. However,  $t_0$  will not be able to occupy  $v_3$ , which is the vertex it is matched to, since  $idx(t_0) < idx(t_1)$  in any given configuration. Given that  $idx(t_1)$  is at most 4, and that we are aiming for  $idx(t_1) = 4$ , it follows that the matching is not executable.

Similarly, ensuring that the token-vertex matching has no crossings is not sufficient either.

The token-vertex matching depicted in Figure 8.2 does not have any crossings. However,  $idx(t_0) < idx(t_1)$  in any given configuration, and since  $t_1$  is not part of the token-vertex matching, we do not intend to displace it. Given that  $idx(t_1)$  is

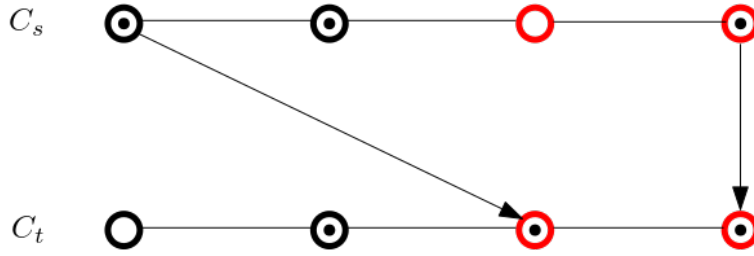


Figure 8.2: Instance where a token-vertex matching with no crossings is not executable.

equal to 1 and is constant and that we are aiming for  $idx(t_0) = 2$ , it follows that the matching is not executable.

We proceed to show that distance minimization and the absence of crossings combined make it possible to execute the matching, and that the total displacement operations needed to execute the matching will be equal to its total distance. The argument we use is very similar for all three output modes: we will proceed by induction on the total distance of the matching. Clearly, if the total distance of the matching is equal to 0, then the statement is true, as the matching is executable (in fact, the instance is already solved), and there are no displacements to be done, irrespective of the output mode we are looking at.

- For the unbatched output mode, it is sufficient to find a single executable displacement. Consider the rightmost right-moving atom, and displace it to the right. Clearly, this displacement is permissible, as the only case where it is not is if the matching has crossings. If there are no right-moving atoms, consider the leftmost left-moving atoms and displace it to the left. Without loss of generality, assume that a right-moving atom  $t_i$  was found, and that it was displaced to the right. The total distance of that same matching was decreased by 1, it is still distance-minimizing on the resulting configuration, and no crossings were created, so we can apply the inductive hypothesis.
- For the batched output mode, it is sufficient to find a single executable batch. Consider all moving tokens, we need to prove that a batch comprising all those atoms is executable. Assume it is not, that is, assume that the execution of such a batch would lead to the displacement of a token  $t_i$  into an occupied trap. Without loss of generality, assume that  $t_i$  is right-moving. The token  $t_{i+1}$  in the occupied trap is either part of the token-vertex matching, or it is not. If it is not, the matching we are working with is not distance-minimizing: we can match  $t_{i+1}$  with  $M^{-1}(t_i)$  and remove  $t_i$  from the matching. The resulting matching has no crossings and has a total distance that is smaller than that of the original matching. If  $t_{i+1}$  is part of the matching, given the way the batch greedily selects tokens to move, the only reason why  $t_{i+1}$  is not moving is because it is matched to the trap

it is occupying. However, if that were the case, this contradicts what was assumed about the matching having no crossings. We have therefore proved that a batch is always executable, the result we are looking for follows from applying the inductive hypothesis on the configuration that results from the execution of the batch, as the considered matching in this configuration has no crossings and is distance-minimizing.

- For the block batched output mode, a similar argument as the one that was used for proving a batch is executable if we assume that we start with a matching that is distance minimizing and that has no crossings can be made.

□

**Lemma 4.** *Any distance-minimizing token-vertex matching  $M$  in a path can be turned into a distance-minimizing token-vertex matching  $M'$  with no crossings.*

*Proof.* If  $M$  has no crossings to begin with, we are done. Otherwise, the crossings in  $M$  take on a very specific form. If  $t_i$  and  $t_j$  ( $i < j$ ) are involved in a crossing in  $M$ ,  $idx(M^{-1}(t_j)) - idx(t_j)$  and  $idx(M^{-1}(t_i)) - idx(t_i)$  both have the same sign (i.e. the tokens cannot be moving in opposite directions, 0 is both positive and negative), because if they do not,  $t_i$ 's and  $t_j$ 's targets in the matching can be switched, and the resulting matching will have a smaller total distance. Such target switches also do not create any new crossings. After a finite number of target switches, each of which reduces the total number of crossings by 1, we get a distance-minimizing token-vertex matching  $M'$  with no crossings. □

**Lemma 5.** *In a path with token set  $TK$  and target region  $T$ , if  $|TK| = |T| = K$ , there is a unique distance-minimizing token-vertex matching with no crossings.*

*Proof.* We map the leftmost target vertex to  $t_0$ , the second leftmost target vertex to  $t_1$ , ... the  $i^{th}$  leftmost target vertex to  $t_{i-1}$ , ... the  $K^{th}$  leftmost target vertex to  $t_{K-1}$ . Clearly, this matching has no crossings and is the only matching that has no crossings, as any other permutation in the mapping will yield at least one pair of tokens  $t_i$  and  $t_j$  ( $i < j$ ) such that  $idx(M^{-1}(t_i)) > idx(M^{-1}(t_j))$ . It remains to show that the proposed matching is distance-minimizing: this follows from Lemma 4. □

**Lemma 6.** *In a path,  $M$  is a distance-minimizing token-vertex matching if and only if  $M$  is a subset of the matching  $M'$  outputted by the Hungarian procedure.*

*Proof.* ( $\rightarrow$ ) Let  $d(M)$  be the total distance of matching  $M$ . We expect the total cost of  $M'$  be  $d(M) + (|TK| - |T|)K$ . Assume that this was not the case, i.e. assume that  $d(M') < d(M) + (|TK| - |T|)K$ , this would mean that a subset of  $M'$  of size  $|T|$  has a total cost that is smaller than  $d(M)$ . Given the construction

of the Hungarian instance, this implies that  $M$  is not distance-minimizing.

( $\leftarrow$ ) Let  $d(M') = m + (|TK| - |T|)K$  be the total cost of matching  $M'$  outputted by the Hungarian procedure. Clearly, this implies the existence of a token-vertex matching with a total distance of  $m$ . It remains to show that this token-vertex matching is distance-minimizing. Assume that it is not, and that there exists a distance-minimizing token-vertex matching that has a total distance equal to  $m' < m$ . Given the construction of the Hungarian instance, this implies that  $d(M') \leq m' + (|TK| - |T|)K$ , and we have a contradiction.  $\square$

**Lemma 7.** *In paths, a distance-minimizing token-vertex matching involves contiguous tokens when the target vertices are contiguous.*

*Proof.* Assume that we have a distance-minimizing token-vertex matching  $M$  that does not involve contiguous tokens. Consider the leftmost token  $t_i$  that is not in the matching such that  $t_{i-1}$  is. Let  $t_j$  be the leftmost token to the right of  $t_i$  that is in the matching. Clearly,  $idx(M^{-1}(t_{i-1})) + 1 = idx(M^{-1}(t_j))$  (i.e. there is no vertex  $v_j$  such that  $idx(M^{-1}(t_{i-1})) < idx(v_j) < idx(M^{-1}(t_j))$ ). We also have:  $idx(t_{i-1}) < idx(t_i) < idx(t_j)$ . It follows that  $t_i$  obstructs one of  $t_{i-1}$  and  $t_j$ . Without loss of generality, assume it obstructs  $t_{i-1}$ , which means that  $idx(t_{i-1}) < idx(t_i) \leq idx(M^{-1}(t_{i-1}))$ . If we set the target vertex of  $t_i$  to be  $M^{-1}(t_{i-1})$  and if we remove  $t_{i-1}$  from the matching, we end up with a matching  $M'$  such that  $d(M') < d(M)$ , which contradicts the assumption that  $M$  is distance-minimizing. This concludes the proof.  $\square$

**Lemma 8.** *In paths, a distance-minimizing token-vertex matching involves all the tokens that are initially in the target region when the target vertices are contiguous.*

*Proof.* From Lemma 7, we know that the tokens in a distance-minimizing token-vertex matching  $M$  are contiguous. Therefore, if there are tokens in the target region that are not part of the matching, they will either be to the left of the matched tokens or to their right. Without loss of generality, assume that the tokens in the target region that are not in the matching are to the left of the matched tokens. Let  $t_i$  be the leftmost matched token. We look at  $t_{i-1}$ , the immediate unmatched token to its left. Clearly,  $idx(M^{-1}(t_i)) \leq idx(t_{i-1}) < idx(t_i)$ , therefore we can modify the matching  $M$  by mapping  $t_{i-1}$  to  $M^{-1}(t_i)$  and by removing  $t_i$  from the matching to obtain a matching  $M'$  such that  $d(M') < d(M)$ , which contradicts what we assumed about matching  $M$  being distance-minimizing.  $\square$

**Lemma 9.** *The algorithm for REDREC v2.0 executed on a grid graph with height  $H$ , width  $W$  and target height  $H_T$  terminates in time  $O(W(H_T H + HW))$  and is correct.*

*Proof.* The termination criterion we can use is either the sum of the surpluses across the columns with negative surpluses or the number of columns that have not been deleted yet. We will work with the latter. As a reminder, a column is deleted if and only if it has a surplus of 0 and it has been reconfigured.

At iteration  $k$ , we know that there exists a column that has not been deleted yet, otherwise there would be no columns with a negative surplus. We also know that there exists a column with a negative surplus that is adjacent to a column with a positive surplus, otherwise, if the surpluses of all non-deleted columns are negative, we would be dealing with an instance we cannot reconfigure (which is a case that gets detected at the beginning of the algorithm). The donor-receiver pair selection picks one such pair. There are three possible scenarios:

- The donor's surplus is bigger than the receiver's deficit, in which case the receiver is deleted.
- The donor's surplus is smaller than the receiver's deficit, in which case the donor is deleted.
- The donor's surplus is equal to the receiver's deficit, in which case both the donor and the receiver are deleted.

In all cases, at every iteration, one column gets deleted. Since the input is finite, the number of columns is finite. It follows that we have a finite number of iterations. Once we break out of the while loop, we know that the non-deleted columns have a positive surplus, and that the deleted columns have been reconfigured; the correctness of the algorithm follows from the correctness of the path solvers which are run on the non-deleted columns.

In the worst case,  $O(W)$  iterations are needed to eliminate deficits from all columns. In a redistribution-reconfiguration-shuffling call, we run  $O(H_T)$  linear exact solvers, each of which runs in  $O(H)$ . Once the displacement-minimizing linear exact solution is selected, we shuffle  $O(H)$  tokens as far as  $W - 1$  columns. The running time of  $O(W(H_T H + HW))$  follows from this analysis. □

**Lemma 10.** *In bidirectional COLAV applied on grid graphs, for a given path with source vertex  $v_{c_1, r_1}$  and target vertex  $v_{c_2, r_2}$  ( $r_1 < r_2$ ,  $c_1 < c_2$ ),  $dp[i][j]$  is the smallest number of isolated tokens in the path system excluding the current path on any bidirectional path between the source vertex and vertex  $v_{c_1+i, r_1+j}$ .*

*Proof.* The statement and the proof assume that  $r_1 < r_2$ ,  $c_1 < c_2$ . However, even if that were not the case, the proof can be altered to work for the other three possible cases.

We prove this using a double induction on the two indices  $i$  and  $j$ . For the path going from the source vertex  $v_{c_1, r_1}$  to itself, there are no tokens on the

paths since the current path is excluded from the path system, so  $dp[0][0] = 0$  is correct. Similarly, for the vertices on the same column and the same row as  $v_{r_1, c_1}$ , there is a single path to them, so  $is\_isolated\_token[i][j] + dp[i-1][j]$  and  $is\_isolated\_token[i][j] + dp[i][j-1]$  respectively compute the number of isolated tokens on the path to those vertices correctly. Now, assume that  $dp[i][j]$  takes on the correct value for  $0 \leq i \leq k$ ,  $0 \leq j \leq k'$  (excluding the pair  $(k, k')$ ). That is,  $dp[i][j]$  is the smallest number of isolated tokens on any bidirectional path between the source vertex and vertex  $v_{c_1+i, r_1+j}$  in the specified intervals (excluding the pair  $(k, k')$ ). We would like to prove that  $dp[k][k']$  takes on the correct value.

In any path from  $v_{c_1, r_1}$  to  $v_{c_1+k, r_1+k'}$ , vertex  $v_{c_1+k, r_1+k'}$  can be reached from either vertex  $v_{c_1+k-1, r_1+k'}$  or vertex  $v_{c_1+k, r_1+k'-1}$ ; by the inductive hypothesis, we already know the smallest number of isolated tokens from  $v_{c_1, r_1}$  to either of those two vertices; the smallest number of isolated tokens from  $v_{c_1, r_1}$  to  $v_{c_1+k, r_1+k'}$  will therefore be the minimum of those two values, to which we add 1 in case there is an isolated token on vertex  $v_{c_1+k, r_1+k'}$ .

This concludes the proof.  $\square$

**Lemma 11.** *The bidirectional COLAV procedure applied on grid graphs terminates in time  $O((HW)^3)$ .*

*Proof.* In bidirectional COLAV, a path is modified if and only if we were able to find a rerouting that is able to isolate at least one more token compared to the original path. Since the number of tokens is linear in  $HW$ , so is the number of paths that will be rerouted. Also, the algorithm terminates when it loops over all paths without rerouting any path, which is bound to happen. The proof of termination is complete.

The running time of a single path rerouting is  $O(HW)$ . There can be at most  $O(HW)$  reroutings, because the number of tokens is  $O(HW)$ , and every token rerouting may require looping over  $O(HW)$  paths. The running time of bidirectional COLAV is therefore  $O((HW)^3)$ .  $\square$

**Lemma 12.** *In bounded COLAV applied on grid graphs, for a given path with source vertex  $v_{c_1, r_1}$  and target vertex  $v_{c_2, r_2}$  ( $r_1 < r_2$ ,  $c_1 < c_2$ ),  $\min(dp[dp_H + dp_W][m + dp_W - 1][m + dp_H - 1], dp[dp_H + dp_W + 1][m + dp_W - 1][m + dp_H - 1], \dots, dp[(dp_H + 2m) \times (dp_W + 2m) - 1][m + dp_W - 1][m + dp_H - 1])$  is the smallest number of isolated tokens in the path system excluding the current path on any path between the source vertex and the target vertex.*

*Proof.* As was the case for the proof of correctness of bidirectional COLAV, the statement and the proof assume that  $r_1 < r_2$ ,  $c_1 < c_2$ . However, even if that were not the case, the proof can be altered to work for the other three possible cases.

We start by proving the correctness of the dynamic programming approach, as the correctness of the statement we are trying to prove follows directly from

that. That is, we start by proving that  $dp[i][j][k]$  is the smallest number of isolated tokens on any path of length  $i$  between  $v_{c_1, r_1}$  and  $v_{c_1+j-m, r_1+k-m}$  in the path system that excludes the current path.

When computing  $dp[i][j][k]$ , we use the value  $-1$  to signify that  $v_{c_1+j-m, r_1+k-m}$  cannot be reached from  $v_{c_1, r_1}$  within  $i$  displacements.

We will induce on the first dimension only (i.e. the displacement dimension).

With no displacements, starting from  $v_{c_1, r_1}$ , we can only reach  $v_{c_1, r_1}$ . Since the current path is excluded from the path system, there are no tokens from  $v_{c_1, r_1}$  to itself, therefore  $dp[0][m][m] = 0$ . No other vertex is reachable for  $i = 0$ , so it should be the case that  $dp[0][i][j]$  is equal to  $-1$  ( $0 \leq i < dp_W + 2m$ ,  $0 \leq j < dp_H + 2m$ , except  $i = j = m$ ), which is true.

Now, assume that  $dp[l][j][k]$  takes on the correct values ( $0 \leq j < dp_W + 2m$ ,  $0 \leq k < dp_H + 2m$ ). We would like to prove that  $dp[l+1][j][k]$  takes on the correct values ( $0 \leq j < dp_H + 2m$ ,  $0 \leq k < dp_W + 2m$ ).

There are two cases to consider here: for a fixed value of the pair  $(j, k)$ . If none of  $v_{c_1+j-m-1, r_1+k-m}$ ,  $v_{c_1+j-m+1, r_1+k-m}$ ,  $v_{c_1+j-m, r_1+k-m-1}$  and  $v_{c_1+j-m, r_1+k-m+1}$  is reachable within  $l$  displacements, then  $v_{c_1+j-m, r_1+k-m}$  should not be reachable within  $l+1$  displacements. By the IH, we would have  $dp[l][j-1][k] = dp[l][j+1][k] = dp[l][j][k-1] = dp[l][j][k+1] = -1$ , which gets the value of  $dp[l+1][j][k]$  set to  $-1$  as well, which is correct.

The second case is when at least one of the neighbors of  $v_{c_1+j-m, r_1+k-m}$  has been reached in  $l$  steps. By the inductive hypothesis, we have the smallest number of isolated tokens from  $v_{c_1, r_1}$  to the reached neighbors of  $v_{c_1+j-m, r_1+k-m}$  in  $l$  steps.  $v_{c_1+j-m, r_1+k-m}$  can only be reached from any of its neighbors that were reached in  $l$  steps, so the smallest number of isolated tokens on any path of length  $l+1$  from  $v_{c_1, r_1}$  to  $v_{c_1+j-m, r_1+k-m}$  is equal to the minimum of the values obtained for the neighbors reached in  $l$  steps, to which we add 1 in case there is an isolated token on vertex  $v_{c_1+j-m, r_1+k-m}$ .

The proof is complete.  $\square$

**Lemma 13.** *The bounded COLAV procedure applied on general graphs terminates.*

*Proof.* In bounded COLAV on grid graphs, a path change has one of three consequences:

1. The isolation of one or more tokens, with an indeterminate effect on the overall distance of the path system and the overall number of derivations in the path system.
2. The decrease of the overall distance of the path system, with an indeterminate effect on the overall number of derivations in the path system.
3. The decrease of the overall number of derivations in the path system.



The three affected measures are all polynomial in the size of the grid. The first scenario can occur polynomially many times, and lead to a polynomial increase in the other two metrics. Similarly, the second scenario can occur polynomially many times, and lead to a polynomial increase in the third metric, which means that the third scenario can occur polynomially many times as well.

Hence, the procedure applied on grid graphs terminates after rerouting polynomially many paths. As for general graphs, we drop the mention of derivations, and the same termination argument still holds with small modifications.  $\square$

**Lemma 14.** *The path merging procedure applied on general graphs terminates.*

*Proof.* In path merging, merging two paths has one of three consequences:

1. The decrease of the overall distance of the path system, with an indeterminate effect on the number of isolated tokens in the path system and the frequency of unique edges in unmerged path pairs.
2. The increase of the overall number of isolated tokens in the path system, with an indeterminate effect on the frequency of unique edges in unmerged path pairs.
3. The decrease of the frequency of the edge that is unique in the smallest number of unmerged path pairs, with an indeterminate effect on the frequency of unique edges in other unmerged path pairs.

The first two consequences are evident. Clearly, the first two consequences occur polynomially many times. Subsequently, we keep selecting the same edge, until it no longer occurs uniquely in any unmerged path pair. We still have to show that, once an edge is no longer unique in any unmerged path pair, that its frequency as a unique edge in an unmerge pair of paths can no longer increase.

If some merge increased the frequency of the edge in question, since merging involves reusing edges that are already part of the path system, this implies that the edge already occurred uniquely in some unmerged path pair involving the path that the merging rerouted through, which contradicts what we said about the edge no longer being unique in any unmerged path pair.  $\square$

**Lemma 15.** *Applying path unwrapping on a merged path system defined over a general graph does not undo merging.*

*Proof.* It is sufficient to show that unwrapping paths within an arbitrary path in a merged path system does not give rise to a pair of paths that have more than 1 intersection. When unwrapping paths within a path, some paths are shortened and some paths are extended. Shortening a path does not create an additional intersection between it and some other path, so we only have to worry about the paths that get extended as a result of the unwrapping. If the extension of some

path makes it intersect some other path more than once, the selected arbitrary path which initially wrapped the now extended path intersects said path more than once, because the extended path is a subpath of the selected arbitrary path, which contradicts the assumption that we started with a merged path system.  $\square$

**Lemma 16.** *No unwrapped path remain after path unwrapping applied on general graphs terminates.*

*Proof.* Assume that path  $P_i$  remains wrapped in path  $P_j$  after termination. We know that the algorithm should have looped over  $P_j$  and its superpaths in its execution. In Section 5.2.1.4, in the part that covers path unwrapping, we prove that unwrapping paths within any superpath of  $P_j$  would eliminate the wrapping of  $P_i$  within  $P_j$ . Since the wrapping persisted, it has to be the case that the unwrapping of paths within another path that is not a superpath of  $P_j$  caused it. This is not possible, as the only paths that modify  $P_i$  and  $P_j$  are the superpaths of  $P_j$ .  $\square$

**Lemma 17.** *In cycle breaking applied on general graphs, if there is a cycle, then there is a contiguously colored cycle.*

*Proof.* Consider any cycle: if the cycle is contiguously colored, there is nothing to prove. Otherwise, we describe how this cycle can be transformed into a contiguously colored cycle. We work with the assumption that the edges that separate edges of the same color on the cycle cannot all be colored in that same color, otherwise we can eliminate a color discontinuity.

Let  $i$  be the color of the cycle edge  $(u, w)$ . We handle making all other colors contiguous first.

Consider the vertex representation  $v_1, v_2, \dots, v_k$  of the cycle (where  $v_1 = u$  and  $v_k = w$  are the endpoints of the cycle edge). Consider a discontinuous color  $j$ : let  $v_l$  and  $v_m$  be the earliest and the latest vertex respectively in the vertex representation of the cycle belonging to path  $P_j$  ( $v_l$  and  $v_m$  may be  $v_1$  and  $v_k$  respectively). We know that there is a contiguously colored path from  $v_l$  to  $v_m$ : this path (the edges of which are  $j$ -colored) can replace the  $v_l \dots v_m$  subpath in the vertex representation of the cycle. The new vertex sequence represents a cycle that contains the edge  $(v_1, v_k)$ . The original, cycle-edge containing cycle was therefore transformed into a new cycle edge-containing cycle, and the transformation reduced the number of discontinuous colors by 1: we say that we merged color  $j$ .

The above only works if the subpath from  $v_l$  to  $v_m$  in path  $P_j$  does not go through the cycle edge. If it does, we split the subpath from  $v_l$  to  $v_m$  in path  $P_j$  in two. Without loss of generality, assume that, within path  $P_j$ , the vertices are in the following order:  $v_m, u, w, v_l$  (the other four permutations,  $v_m, w, u, v_l$ ,  $v_l, u, w, v_m$  and  $v_l, w, u, v_m$  are handled similarly). The first subpath is from  $v_m$  to  $u$  and the second subpath is from  $w$  to  $v_l$ . We first make the cycle edge  $j$ -colored.

Those two subpaths then replace the  $v_m \dots u$  subpath and the  $u \dots v_l$  subpath respectively in the vertex representation of the cycle. Note that color  $j$  need not be contiguous after this step, and that changing the color of the cycle edge may have caused a discontinuity in color  $i$  if none already existed; if it did, that is not an issue because the  $i$ -colored discontinuous edges are eliminated from the cycle.

We exhaustively apply the procedure above, and we end up with a cycle where the only discontinuous color, if any, is the color of the cycle edge, which we call  $i$ . We now describe how color  $i$  can be made contiguous.

In the vertex representation of said cycle, we are interested in segments that are  $i$ -colored, where an  $i$ -colored segment is a maximal contiguous sequence of  $i$ -colored edges in the edge representation of the cycle. If there is only one  $i$ -colored segment, the cycle is contiguously colored and we are done. Otherwise, we have two or more  $i$ -colored segments. We sort those segments by their order of appearance within path  $P_i$ , starting from path  $P_i$ 's source vertex. After sorting, we end up with a (possibly empty) set of segments ( $S_{before}$ ) occurring before the segment that contains the cycle edge, followed by the segment that contains the cycle edge, followed by a (possibly empty) set of segments ( $S_{after}$ ) occurring after the segment that contains the cycle edge. The nonempty sets of segments can be separately and sequentially turned into a single segment by using the same method as the one we previously described for the other colors; empty sets of segments are ignored. It may be the case that some of the segments in  $S_{after}$  are no longer part of the cycle after we merge segments in  $S_{before}$ : after  $S_{before}$ 's segments are merged,  $S_{after}$  is updated to include only segments that are still in the cycle, and the same process is repeated for  $S_{after}$ .

Afterwards, we end up with at most three  $i$ -colored segments, such that all vertices of  $P_i$  on the cycle are part of some segment. We now show how to transform a cycle containing three  $i$ -colored segments into one containing two  $i$ -colored segments, which we then transform into a cycle containing a single  $i$ -colored segment.

Let the cycle edge-containing segment be  $s_0$ , and let the latest and the earliest segment in the vertex representation of the cycle be  $s_1$  and  $s_2$  respectively. We call  $start_{s_1}$  and  $end_{s_1}$  the earliest and the latest vertex respectively in  $s_1$  in the vertex representation of the cycle, and we use a similar notation for  $s_2$  and  $s_0$ . Now, we sort those segments by their order of appearance within path  $P_i$ , as we did in the previous step. No matter how the order looks like, there must exist a path from  $s_0$  to one of the two other segments that does not go through the third segment. Without loss of generality, assume that there is a path from  $s_0$  to  $s_2$ . There are four possible cases involving the four vertices  $start_{s_0}$ ,  $end_{s_0}$ ,  $start_{s_2}$  and  $end_{s_2}$ : there is a path from  $start_{s_0}$  to  $end_{s_2}$  along  $P_i$  not going through the other two vertices, there is a path from  $start_{s_0}$  to  $start_{s_2}$  along  $P_i$  not going through the other two vertices, there is a path from  $end_{s_0}$  to  $end_{s_2}$  along  $P_i$  not going through the other two vertices, there is a path from  $end_{s_0}$  to  $start_{s_2}$  along  $P_i$  not going through the other two vertices.

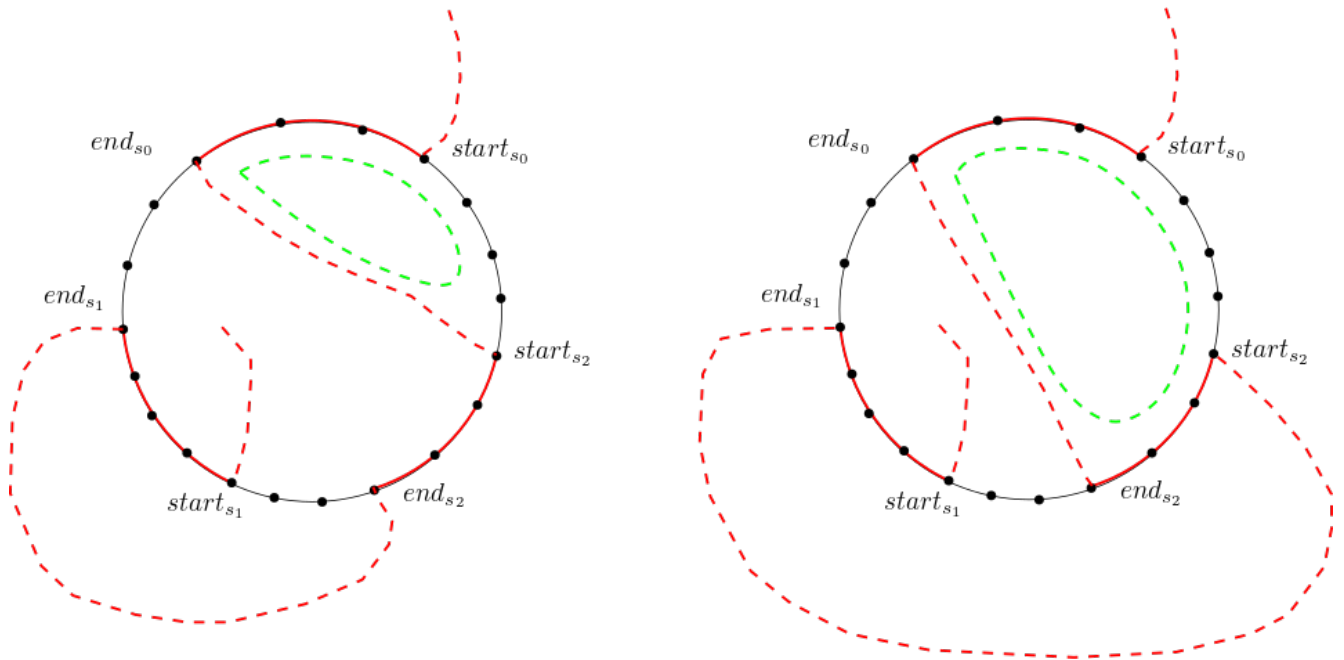


Figure 8.3: Cases where the innermost path between  $s_0$  and  $s_2$  is between  $end_{s_0}$  and a vertex of  $s_2$

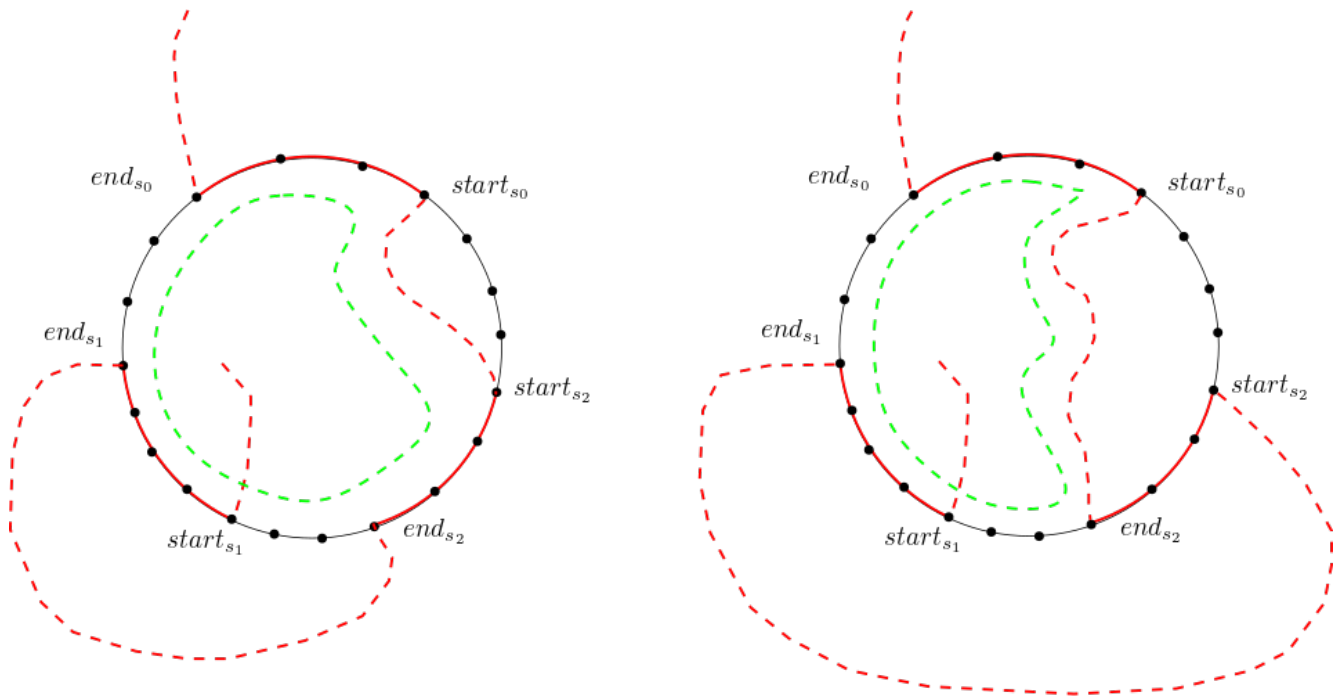


Figure 8.4: Cases where the innermost path between  $s_0$  and  $s_2$  is between  $start_{s_0}$  and a vertex of  $s_2$

In the first two cases (depicted in Figure 8.3), a cycle can be formed by replacing the  $start_{s_2} \dots end_{s_0}$  subpath in the first case (or  $start_{s_2} \dots end_{s_0}$  subpath in the second case) in the vertex representation of the original cycle by the subpath of  $P_i$  bounded by those two vertices. In all cases, we end up with a cycle where color  $i$  is contiguous (depicted in green), and we are done.

In the second two cases (depicted in Figure 8.4), the subpath of path  $i$  bounded by  $start_{s_0}$  and  $start_{s_2}$  in the first case (or by  $start_{s_0}$  and  $end_{s_2}$  in the second case) can be used to merge segments  $s_0$  and  $s_2$ . After the merge, we end up with a cycle with two  $i$ -colored segments. The same process is then repeated again to merge those two segments.  $\square$

**Lemma 18.** *In cycle breaking applied on general graphs, if there is a cycle, then there exists a cycle going through the cycle edge that is induced by two or fewer cycle edge-containing paths.*

*Proof.* We can assume that we are starting with a cycle where the edges of the same color are contiguous along the cycle, thanks to Lemma 17. We show how to transform this cycle into a cycle that uses two or fewer cycle edge-containing paths.

If the cycle already has this property, there is nothing to prove.

Otherwise, the cycle uses three or more cycle edge-containing paths. we describe how this cycle can be transformed.

Assume we are dealing with a total of  $k$  different cycle edge-containing paths in the cycle, with color 1 being the color of the cycle edge.

**Observation 5.** *Changing the color of the cycle edge can give rise to at most 2 color discontinuities.*

We now describe a process that yields a cycle with the desired properties. We cycle through the  $k$  possible colors for the cycle edge in order, and, for every color, we merge the cycle edge with the other segment of the same color. The first color change may cause two discontinuities: one discontinuity in color 1 if the cycle edge is not at the extremity of segment 1, and one discontinuity in color 2. We fix the discontinuity in color 2 using the method from the previous lemma that deals with merging the color of the cycle edge. Note that this may eliminate some colors; if a color is eliminated, it is skipped in the color cycling of the cycle edge. Next, we color the cycle edge using color 3: note that this does not cause a discontinuity in color 2 because the cycle edge is at the extremity of color 2's segment; we merge color 3, and we proceed. Any coloring + merging can eliminate a color, but can never cause a discontinuity in any of the other colors. We then cycle back to color 1. At this point, there are no discontinuities in the cycle, and no color change applied on the cycle edge can cause discontinuities, based on what was mentioned before. We claim that the cycle resulting from this process conforms to the property we are looking for. That is, it uses two or fewer cycle edge-containing paths.

Assume it does not, that is, assume that it uses three or more cycle edge-containing paths. If any of those cycle edge-containing paths is not incident to the cycle edge, the color of the cycle edge can be changed to cause a discontinuity. Otherwise, the only case where we have three or more contiguous segments incident to the cycle edge is when  $(v_{k-1}, v_k)$ ,  $(v_k, v_1)$  and  $(v_1, v_2)$  have three different colors. However, in that case,  $(v_k, v_1)$  can be colored using one of the other two colors and the path corresponding to the original color of  $(v_k, v_1)$  can be discarded.

This concludes the proof.  $\square$

**Lemma 19.** *In cycle breaking applied on general graphs, If an inclusion-minimal set of paths  $\mathcal{P} = \{P_0, P_1, \dots, P_{k-1}\}$  induces a cycle then it induces a single cycle.*

*Proof.* We claim that any minimal set of paths that induces cycles induces exactly a single cycle. Assume the set of paths  $P_0, P_1, \dots, P_{k-1}$  induces multiple cycles and is inclusion-minimal. We can also assume that the colors in the cycle edge-containing cycle they induce are contiguous.

Consider the ordering of the segments in the cycles. Two segment orderings are different if they are not a cyclical permutation (possibly reversed) of each other. If there exists two cycles that have different segment orderings, there exists a segment  $s_i$  that is adjacent to at least three different segments  $s_j, s_l, s_m$  (meaning that the corresponding paths intersect) across the two cycles. Pick any cycle: the segment  $s_i$  cannot be adjacent to  $s_j, s_l$  and  $s_m$  within the segment ordering of the cycle at once. Without loss of generality, assume that, in the selected cycle,  $s_i$  is not adjacent to  $s_j$ : starting from the segment ordering of the selected cycle, we can construct a sequence of segments that omits the segments between  $s_i$  and  $s_j$  because paths  $P_i$  and  $P_j$  intersect. The constructed sequence of segments corresponds to a set of paths with cardinality  $< k$  that induces a cycle, thus contradicting the fact that the set of paths  $P_0, P_1, \dots, P_{k-1}$  was assumed to be inclusion-minimal.

We now handle the case where all the cycles induced by the set of paths have the same ordering of segments. It is sufficient to prove that two cycles that have the same segment ordering are the same cycle. Consider two cycles  $C_i$  and  $C_j$  that have the same segment ordering. For each of the two cycles considered separately, every path contains at least one edge that it does not share with any other path, which we call a path-unique edge, otherwise, a path can be omitted from the set of paths and the remaining paths would still induce a cycle. Let  $s_0, s_1, \dots, s_{k-1}$  be the segment ordering that  $C_i$  and  $C_j$  share. Consider the two paths  $P_0$  and  $P_1$  whose segments are consecutive in the segment ordering: every cycle has to go through at least one path-unique edge in  $P_0$  and one path-unique edge in  $P_1$ , and since the segments are contiguous, the cycle goes through the path-unique edges in question via the intersection of the two paths. Since the pair of paths whose segments are consecutive in the segment ordering were picked arbitrarily, it follows that both  $C_i$  and  $C_j$  contain the edges in all the path intersections. A

similar argument can be used to show that the inclusion of the edges in the path intersections in both cycles implies the inclusion of the edges in the subpaths between the path intersections in both  $C_i$  and  $C_j$ . It follows that the two cycles are identical, since they both include the edges in the path intersections and the edges between the path intersections.

□

**Lemma 20.** *In cycle identification applied on general graphs, paths  $P_0, P_1, \dots, P_{k-1}$  induce a special cycle if and only if  $r_{P_0}$  is reachable from  $l_{P_0}$  through vertices  $v_{P_1}, v_{P_2}, \dots, v_{P_{k-1}}$  in one of the generated  $(l_{P_0}, r_{P_0})$  reachability instances.*

*Proof.* ( $\rightarrow$ ) If we have a special cycle, we know that it is contiguously colored, that the paths that induce it induce no other cycles and that it uses at most two cycle edge-containing paths. We are able to extract a cyclical sequence of paths  $P_0, P_1, \dots, P_{k-1}$  such that every path intersects the one before and after it with wraparound for  $P_0$  and  $P_{k-1}$  (and those are the only path intersections, otherwise we end up with more than one induced cycle). Also, at least one of the paths has to be cycle edge-containing. Without loss of generality, let  $P_0$  be cycle edge-containing. There are two cases we need to handle. If  $P_0$  is the only cycle edge-containing path in the special cycle, there is a path from  $v_{P_1}$  to  $v_{P_{k-1}}$  in all of the path intersection graphs that have  $P_0$  as the base path. We are interested in the path intersection graph with no support paths.  $P_1$  and  $P_{k-1}$  each intersect a different side of  $P_1$ , because if that were not the case, the resulting cycle, which is the only cycle that the set of path induces, would not contain the cycle edge. Therefore, we either have an edge from  $v_{P_1}$  to  $l_{P_0}$  and from  $v_{P_{k-1}}$  to  $r_{P_0}$ , or edges from  $v_{P_1}$  to  $l_{P_0}$  and from  $v_{P_{k-1}}$  to  $r_{P_0}$ . In either of the two cases,  $r_{P_0}$  is reachable from  $l_{P_0}$ . A similar argument can be made in the case where the special cycle has two cycle edge-containing paths.

( $\leftarrow$ ) Assume  $r_{P_0}$  is reachable from  $l_{P_0}$ , and let  $l_{P_0}, v_{P_1}, v_{P_2}, \dots, v_{P_{k-1}}, r_{P_0}$  be the corresponding path that we obtained from the BFS tree. We can easily verify that the existence of such a path implies the existence of a set of paths that induces cycles, this set of paths being  $P_0, P_1, \dots, P_{k-1}$ . We still have to show that they induce a special cycle.

The easiest criterion to verify is that no more than two cycle edge-containing paths are involved, as this follows by construction of the path intersection graphs, since all of them involve either one base path or one base path and one support path.

We now show that the path from  $l_{P_0}$  to  $r_{P_0}$  implies the corresponding set of paths induce a single cycle. Assume that a proper subset of the paths induces a cycle, and pick the smallest such subset. Let  $P_i$  be the path in said subset such that  $v_{P_i}$  is the earliest occurring vertex in the path from  $l_{P_0}$  to  $r_{P_0}$ . There exists two paths it intersects within the subset such that they occur later in the path from  $l_{P_0}$  to  $r_{P_0}$ ; let those paths be  $P_j$  and  $P_k$ . Since  $v_{P_i}$  is the earliest occurring vertex, BFS visited it earlier than  $v_{P_j}$  and  $v_{P_k}$ . Therefore, in the BFS tree,  $v_{P_j}$

and  $v_{P_k}$  are children of  $v_{P_i}$ : the path from a leaf ( $r_{P_0}$  specifically) to the root of the BFS tree cannot therefore include both  $v_{P_j}$  and  $v_{P_k}$ . It follows that the set of paths  $P_0, P_1 \dots P_{k-1}$  induces a single cycle.

The color contiguity criterion does not require proving, as we have already proven that any cycle can be turned into a contiguously colored cycle without adding new paths. With that being said, we will prove something even stronger: **all** possible colorings of the cycle that the set of path  $P_0, P_1 \dots P_{k-1}$  induces are contiguous.

Assume some coloring of the cycle creates a discontinuity. Pick the discontinuous color that corresponds to the path whose vertex is the earliest in the path from  $l_{P_0}$  to  $r_{P_0}$ . If the color is adjacent to three or more different colors then we can use the same argument that we used to prove the paths induce a single cycle to show that the color cannot be adjacent to 3 other colors. Otherwise, if we have a discontinuous color that is adjacent to two or fewer different colors, then we have an unmerged pair of paths. This is not possible, because the input of the cycle breaking procedure is a merged path system.  $\square$

**Lemma 21.** *The cycle breaking procedure applied on general graphs terminates in time polynomial in the input size.*

*Proof.* The cycle breaking procedure was designed in a way that ensures it terminates in polynomial time. If we limited it to arbitrarily detecting and breaking cycles, it would have been harder to prove that the procedure terminates, and even harder to prove that it terminates in time polynomial in the input size. Breaking a single cycle has one of three consequences:

1. The decrease of the overall distance of the path system, with an indeterminate effect on the number of isolated tokens in the path system and edge frequency.
2. The increase of the overall number of isolated tokens in the path system, with an indeterminate effect on edge frequency.
3. The decrease of the frequency of the least frequent edge that is part of a cycle, with an indeterminate effect on the frequency of the other edges.

The consequences and their hierarchy are given by construction of the algorithm. The first consequence can occur polynomially many times and decreases the sum of the edge frequencies; it may also increase the number of isolated tokens (but not decrease them, as cycle breaking reroutes paths through edges that are already in paths in the path system). Irrespective of the first consequence, the second consequence can occur polynomially many times, because there are polynomially many tokens, and the third consequence can occur polynomially many times, since edges that are taken out of cycles are not brought back into cycles because of how cycle breaking is designed.



Given all the above, as well as the fact that cycle breaking is done in polynomial time, it follows that the cycle breaking procedure terminates in time polynomial in the size of the input.  $\square$

**Lemma 22.** *In a path system on general graphs, two paths that cross some edge in opposite directions cannot be part of a distance-minimizing path system.*

*Proof.* Let  $P_i = \{v_0, v_1, \dots, v_{i-1}, v_i, v_{i+1}, v_{i+2}, \dots, v_{k-1}\}$  and  $P_j = \{u_0, u_1, \dots, u_{j-1}, u_j, u_{j+1}, u_{j+2}, \dots, u_{l-1}\}$  be two paths that cross at least one edge in opposite directions ( $u_j = v_{i+1}$  and  $u_{j+1} = v_i$ ). The paths can be updated to  $P'_i = \{v_0, v_1, \dots, v_{i-1}, u_{j+1}, u_{j+2}, \dots, u_{l-1}\}$  and  $P'_j = \{u_0, u_1, \dots, u_{j-1}, v_{j+1}, v_{j+2}, \dots, v_{k-1}\}$  respectively. Clearly, the updated path system is still valid, as we still have the same source vertices and target vertices, and we have reduced the total distance of the path system by 2. Note that the absence of paths crossing the same edge in opposite is a necessary but not a sufficient condition for a distance-minimizing path system on a grid.  $\square$

**Lemma 23.** *The number of displacements outputted by the exact extraction/implantation forest solver is equal to the total distance of the corresponding path system.*

*Proof.* It is sufficient to prove the statement for trees. The path system that the exact extraction/implantation forest solver works with is distance-minimizing. By Lemma 22, for any tree, we know that none of its paths will be crossing edges in opposite directions. Moreover, the frequency of every edge has a distinctive significance. In any tree  $T$ , consider an edge  $e = (u, v)$ , and let  $T_1$  and  $T_2$  be the two trees obtained after removing edge  $e$ . The source vertices of all paths that have vertices in both  $T_1$  and  $T_2$  are either all in  $T_1$  or all in  $T_2$ . Without loss of generality, assume that those paths go from  $T_1$  to  $T_2$ , this means that the edge frequency indicates the token deficit of  $T_1$ , or the number of target vertices in  $T_1$  minus the number of source vertices in  $T_1$ . Let the frequency of edge  $e$  be  $k$ . Clearly, if less than  $k$  tokens traversed the edge from  $T_1$  to  $T_2$ , we would not be able to clear the token deficit of  $T_1$ . We now show that the exact extraction/implantation forest solver cannot move  $k+m$  tokens from  $T_1$  to  $T_2$  and  $m$  tokens from  $T_2$  to  $T_1$ , which still clears the token deficit of  $T_1$  ( $m > 0$ ). This follows from the proof of correctness of Călinescu's exact extraction/implantation tree solver, which arbitrarily roots the tree and solves it in a bottom-up fashion, with each subtree being considered either a donor subtree (if its number of tokens exceeds its number of target vertices), a receiver subtree, (if its number of target vertices exceeds its number of tokens), or a solved subtree (if its number of tokens is equal to its number of target vertices). A donor subtree cannot turn into a receiver subtree (or vice versa), so the edge connecting the root of every subtree to the rest of the tree is only used in one direction, and this applies to every subtree, and therefore every edge.  $\square$

# CHAPTER 9

## EXPERIMENTAL SETUP AND EXPERIMENTAL RESULTS

In this section, we cover myriad experimental results related to the path solvers and the grid solvers. The integrality of the experiments was run on a machine with a AMD EPYC 7551 CPU with 8 cores and with a total of 32GB of memory. The experiments that were run on the GPU were run on a Tesla V100 GPU with 32GB of device memory.

Since the number of experiments that we can run is considerably large, we limit the plots included in this chapter to representative results, along with their interpretation. The plots for the rest of the experiments can be found in Appendix A.

### 9.1 Experiments on path solvers on the CPU

When it comes to path solvers, our main focus is running time. The algorithms were designed to work with token surplus, so we include multiple experiments with different surplus ratios, where a surplus ratio of  $x$  means  $K = (1 + x)K'$  (the value of  $K$  is rounded up to the nearest integer). The surplus ratios we look at are  $x_1 = 0$  and  $x_2 = 0.1$ . A surplus ratio of 0.1 with the number of target vertices being equal to half the number of traps is the standard scenario for both path instances and grid instances. In this section, we include scatter plots for the running times of our path solver algorithms using the batched token mover and the block batched token mover. The plotted data is averaged over 100 randomly generated token moving instances with randomly generated token locations and centered target regions to make comparisons between the bruteforce path solver and the rest of the solvers possible.

The figures included in this section each consist of two plots. The left plot contains all data points, whereas the right plot has a cutoff at 0.02s on the y-axis which makes it possible to see the relationship between datapoints that would

have otherwise overlapped.

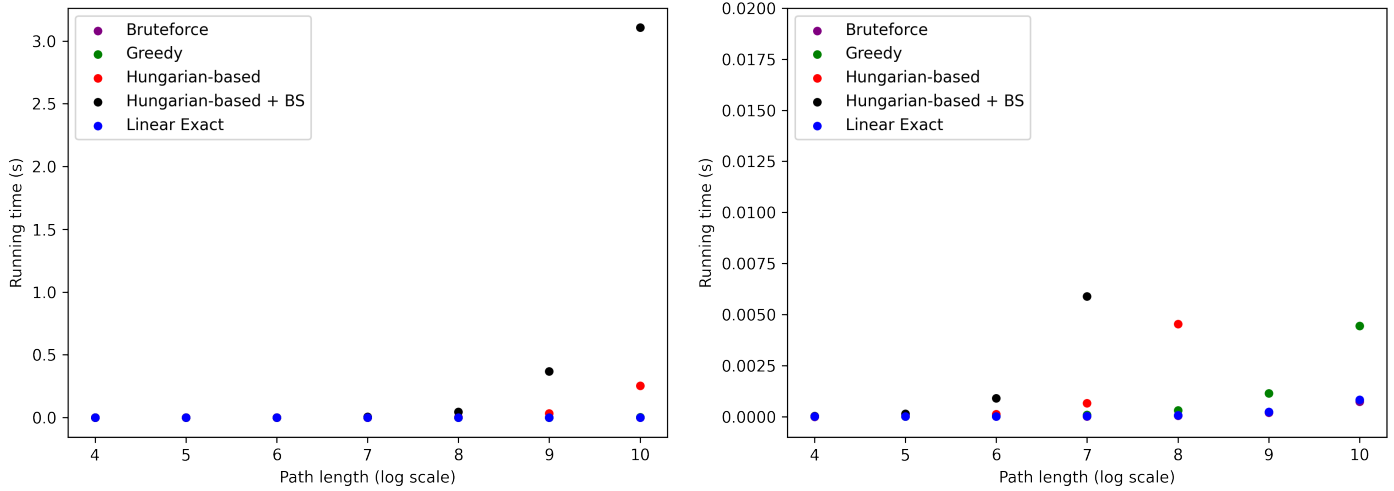


Figure 9.1: CPU running times for path solvers on centered targets with no surplus (batched token moving).

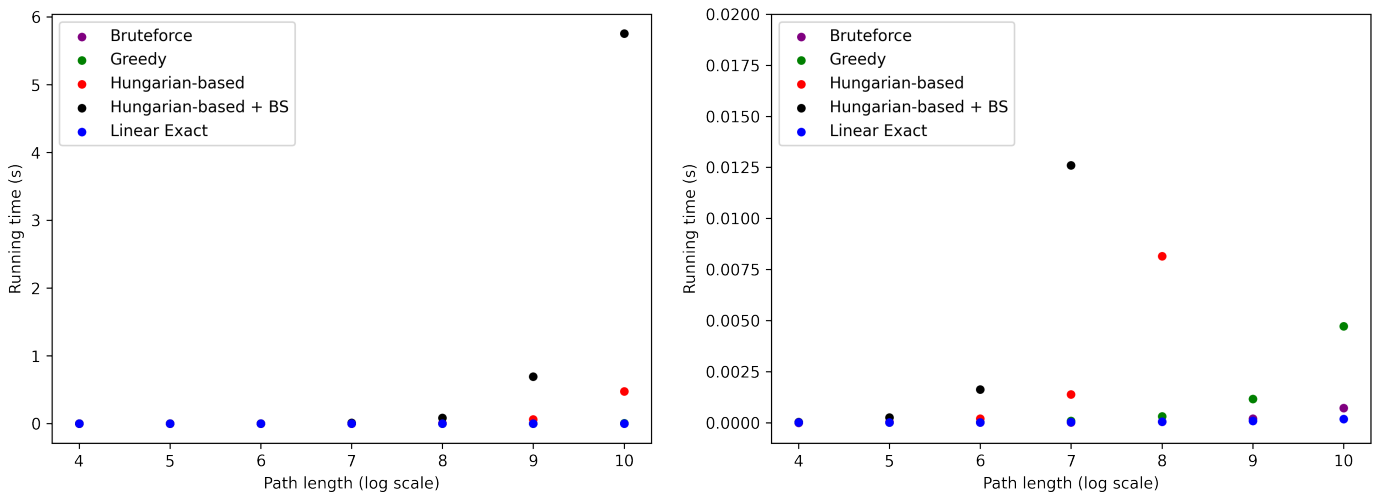


Figure 9.2: CPU running times for path solvers on centered targets with a surplus ratio of 0.1 (batched token moving).

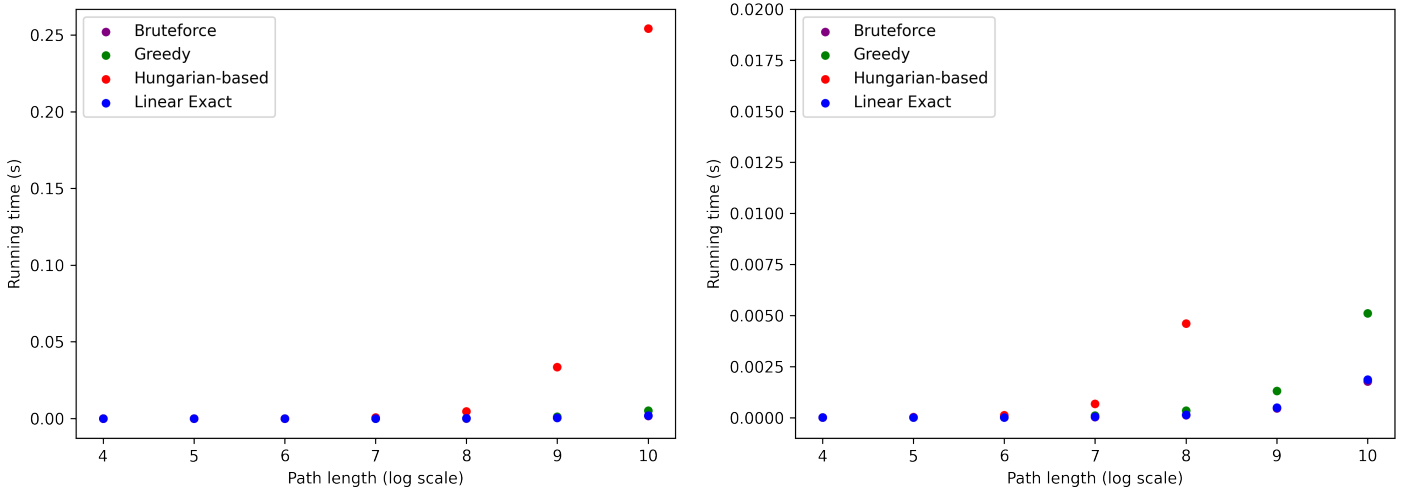


Figure 9.3: CPU running times for path solvers on centered targets with no surplus (block batched token moving).

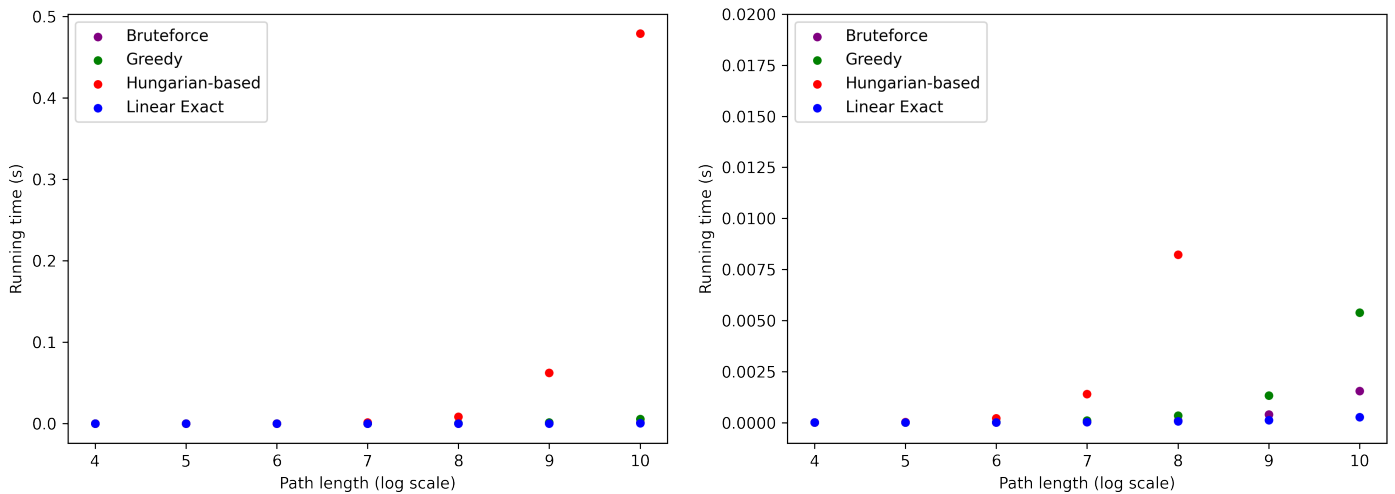


Figure 9.4: CPU running times for path solvers on centered targets with a surplus ratio of 0.1 (block batched token moving).

The results we obtain within every plot are unremarkable: the Hungarian-based path solver has the slowest running time out of all the path solvers we discussed, and this can be traced back to the running time of the Hungarian solver that is cubic in the number of tokens. The Hungarian-based path solver augmented with binary search is slower than the regular Hungarian-based path

solver by a factor that is logarithmic in the length of the path, as expected. There are three key takeaways from the plots: an increase in  $\frac{K}{K'}$  slows down all path solvers on the CPU except the linear exact path solver, and that can be attributed to its running time being linear in  $N$  rather than in  $K$ . Moreover, on the CPU, the greedy path solver is outperformed by both the brute-force path solver and the linear exact path solver in terms of running time. It follows that the greedy path solver has no practical use on the CPU, since the linear exact path solver minimizes displacement operations and has a better running time. The final note to be made about those results is that comparing the same algorithm across multiple token movers allows us to make inferences regarding the effect of the selected token mover on the running time. The running time does not vary when different token movers are used, which means that the running times are dominated by the token-vertex matching procedure.

## 9.2 Experiments on path solvers on the GPU

On the GPU, the experiments we run on path solvers are very similar to those on the CPU. However, the setup on the GPU also needs to take into account the interaction between the last three modules in the RTFC (Figure 1.2). In the parallel version of the RTFC, the problem solving module writes batches or block batches into a buffer, and the waveform synthesis module reads from the buffer as it is being written into, synthesizes the waveforms, then writes them to the buffer of the waveform streaming module. We rely on simplifying working assumptions to be able to approximately assess whether our solvers are of any use in practice. We assume that the waveform streaming module is capable of streaming waveforms at a rate of 1 waveform every  $10\ \mu\text{s}$ . We are therefore targeting generating batches at a faster rate, otherwise, the problem solving module becomes the bottleneck of the RTFC.

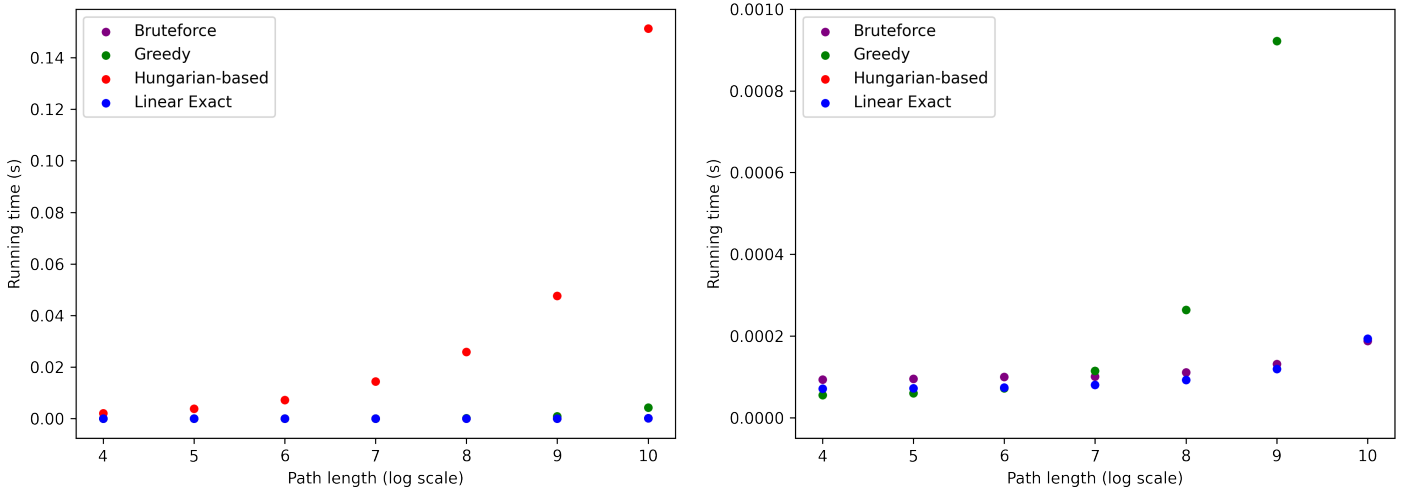


Figure 9.5: GPU running times for path solvers on centered targets with no surplus (batched token moving).

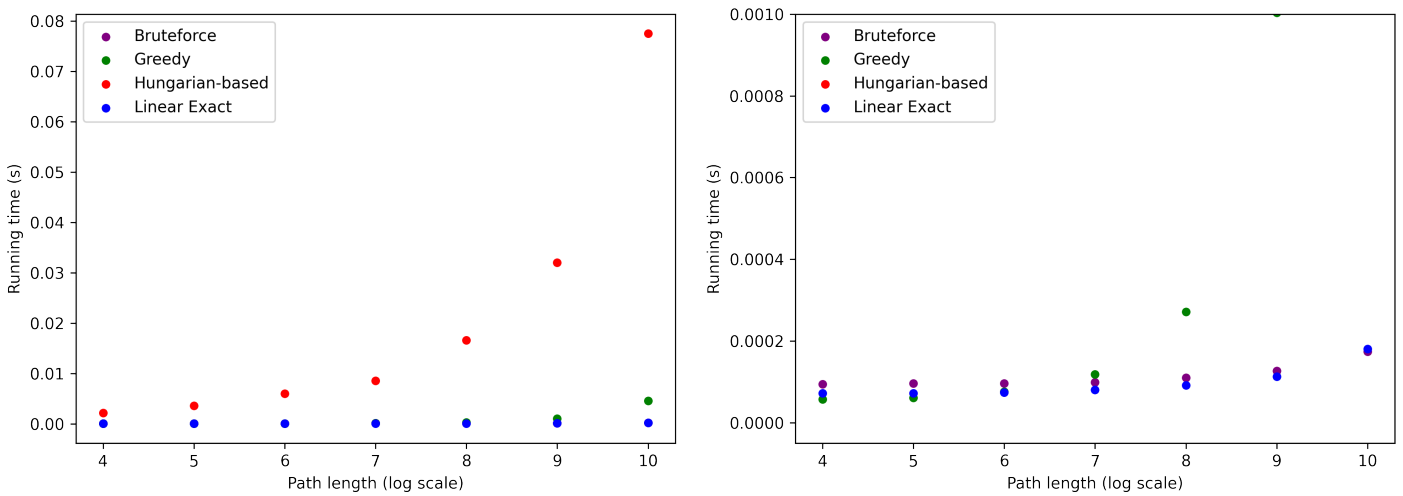


Figure 9.6: GPU running times for path solvers on centered targets with a surplus ratio of 0.1 (batched token moving).

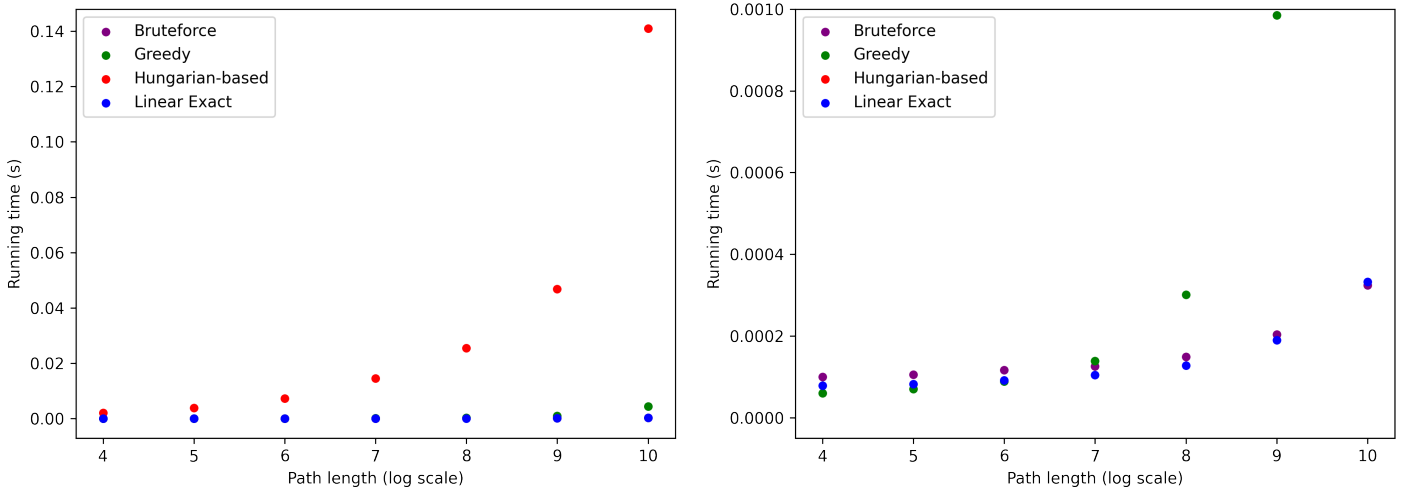


Figure 9.7: GPU running times for path solvers on centered targets with no surplus (block batched token moving).

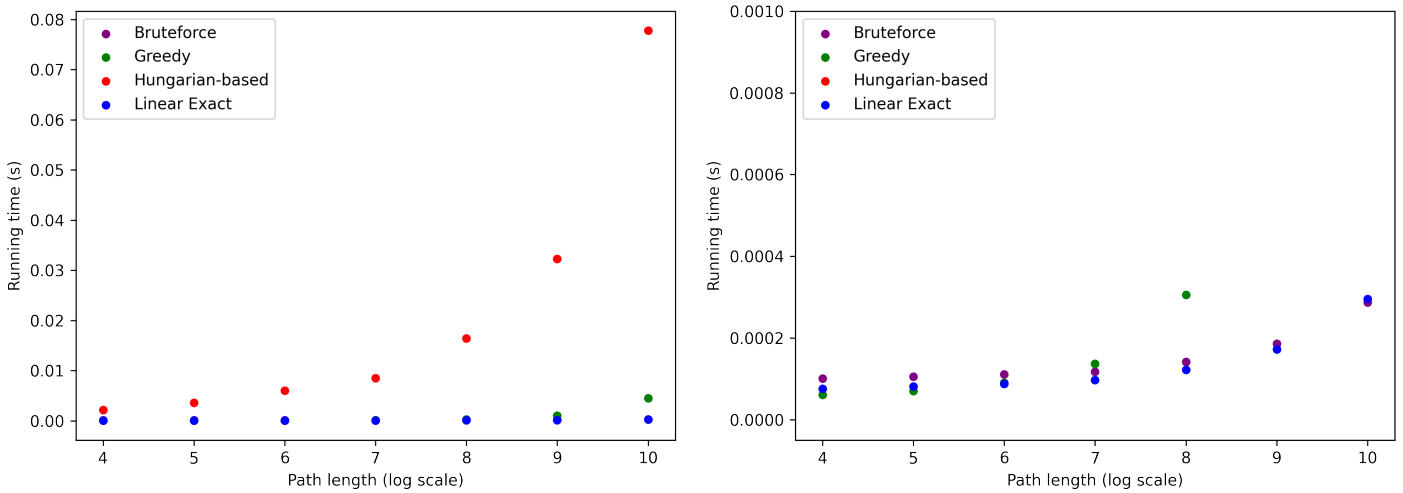


Figure 9.8: GPU running times for path solvers on centered targets with a surplus ratio of 0.1 (block batched token moving).

The analysis will mainly revolve around the experiments with batched token moving, though similar inferences can be made from the experiments with block batched token moving. The kernel launch overhead makes the parallel algorithms run slower than their serial counterparts; the average running time across non-Hungarian solvers on the CPU for  $N = 8$  is  $3\mu\text{s}$  compared to  $75\mu\text{s}$  on the

GPU. With that being said, the GPU implementations outperform all the CPU implementations as of some  $N \leq 1024$  that is different for every algorithm, except in the case of the greedy solver, which suffers from contention due to atomic operations: with no surplus and for  $N = 1024$ , the linear exact path solver on the GPU runs in  $100 \mu\text{s}$  on average, as compared to  $200 \mu\text{s}$  on the CPU, whereas the bruteforce path solver on the GPU runs in  $200 \mu\text{s}$  on average, as compared to  $750 \mu\text{s}$  on the CPU. The results we have obtained for the Hungarian-based path solver are aligned with the empirical results from the paper we reference [33] and a speedup is only perceived for values of  $N \gtrsim 750$ . What is unusual is the speedup we recorded in the running time of the Hungarian-based solver after surplus was introduced: the serial logic within the parallel implementation of the parallel Hungarian solver is tightly bound to the values and the shape of the input matrix, and any perturbation in that regard may lead to vastly different running times, which explains the results we obtained.

We exclude experiments dealing with Hungarian-based path solving with  $k$ -ary search because Hungarian-based path solvers, even with speedup, are very slow and will therefore not be included in the GPU implementation of the RTFC.

We now take a look at how the three path solvers of interest perform in terms of average running time per batch. It is worth noting that the average running time per batch is not an accurate measurement of how well the algorithms will integrate into the RTFC because we are omitting a core concern, which is how the generation of batches is distributed over the running time of the algorithm. Ideally, we would want the batches to be outputted at even intervals. However, in practice, this is not the case, as token-vertex matching has to take place first and matching makes up the majority of the running time of the solvers.

Adding tokens to the source configuration has two consequences. On the one hand, it is expected to reduce the number of block batches required to solve the instance, and on the other hand, it is expected to increase the running time. In Figure 9.9, the relevant path solvers, namely bruteforce path solver and linear exact path solver, beat the targeted  $10 \mu\text{s}$  per batch starting  $N = 32$ , and the inclusion of surplus is inconsequential.

### 9.3 Experiments on grid solvers on the CPU

This section is dedicated to comparing our grid solvers to grid solvers in the literature. In this section, we cover mean success probability (MSP), our main metric of interest, across multiple algorithms. We also highlight the performance of the algorithms in terms of total operations, as well as the distribution of the operations across tokens.

The simulations in this section were devised in a way that makes them as representative of the execution of the experiments on the RTFC, so we introduce the notion of loading efficiency. The hardware specifies a probability  $p_{load}$  of



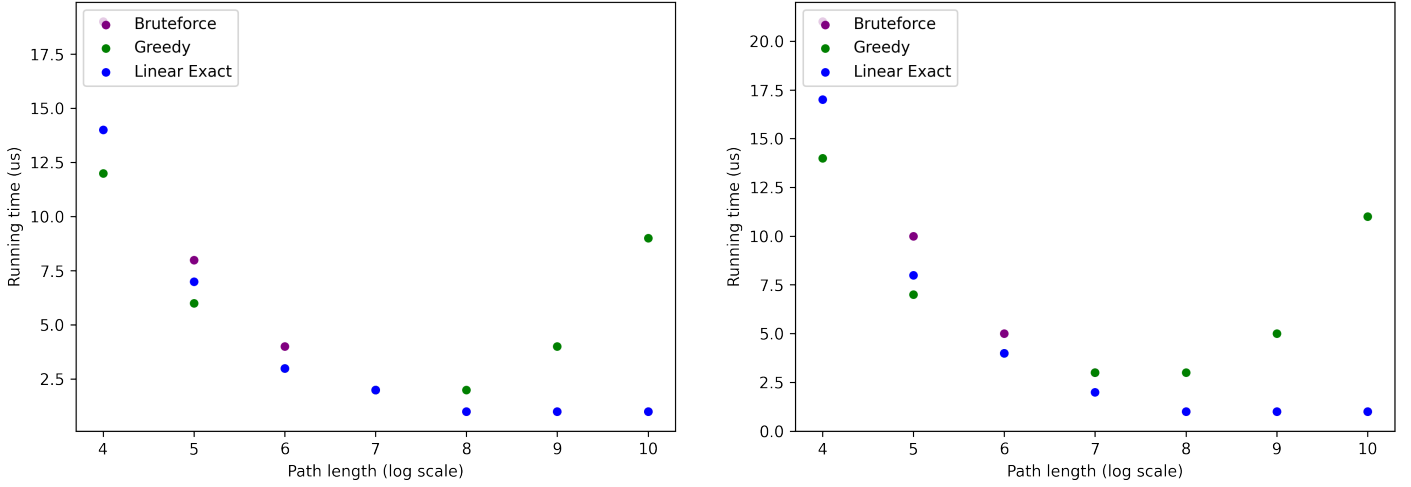


Figure 9.9: Average GPU running time per block batch for path solvers on centered targets with no surplus (left) and with a surplus ratio of 0.1 (right).

a trap being loaded with an atom, and that is how the initial atom array is assembled. This probability is shared across all traps, and the loading of the traps are independent events. If the number of atoms loaded initially is smaller than the size of the target region, the instance is discarded in the simulation.

When the concept of success probability and token loss was introduced, we oversimplified the token loss function (and, consequently, the aggregate loss function). In reality, the per-token loss function depends on a parameter for displacements ( $p_\nu$ ), a parameter for extractions/implantations ( $p_\alpha$ ), the per-token number of displacements ( $N_\nu$ ), the per-token number of extractions/implantations ( $N_\alpha$ ), the elapsed token moving time ( $t$ ), and the lifetime of the token ( $T_\nu$ ), which is given as a constant and takes on the same value across all tokens. The per-token loss function we use in our experiments is:

$$p_i(N_{\alpha_i}, N_{\nu_i}, t) = 1 - p_\alpha^{N_{\alpha_i}} p_\nu^{N_{\nu_i}} e^{-\frac{t}{T_\nu}}$$

We now introduce the naming conventions for the algorithms we refer to in our experiments. In Chapter 5, we introduced the HUNGARIAN-COLAV algorithm, which does both rerouting using COLAV, and ordering of moves using cycle breaking and the exact extraction/implantation forest solver. There is nothing that prohibits us from using one without the other: if ordering is not used, we default to the usage of the greedy solver. We therefore have a total of four different algorithms: greedy HUNGARIAN-NOCOLAV (which is our baseline), greedy HUNGARIAN-COLAV, HUNGARIAN-NOCOLAV and HUNGARIAN-COLAV. When COLAV is active, bidirectional COLAV is the version that is

used.

Figure 9.10 highlights the MSP of token moving on a grid of size  $64 \times 32$  with a centered target of size  $32 \times 32$ , with no greedy token isolation and with  $p_{load} = 0.6$ . For REDRED v2.1, we set  $k$  to 2. The REDREC algorithms are similar in terms of running time: token moving takes around 0.1s per instance on the CPU for all 3 of them. The running times of greedy HUNGARIAN-NOCOLAV and HUNGARIAN-COLAV are drastically worse and take 0.7s to run on average. The slowdown is however made up for in terms of improvement in operational performance: for the problem size of interest, the performance of greedy HUNGARIAN-NOCOLAV is comparable to that of REDREC v1, while the improved REDREC algorithms more than double the MSP: REDREC v2.0 has a MSP of 0.281 over the 500 tested instances, compared to a MSP of 0.286 for REDREC v2.1. The full HUNGARIAN-COLAV algorithm outperforms the REDREC-based algorithms, despite the slowdown in running time, and comes with the extra benefit of being able to solve instances where the target region is not centered.

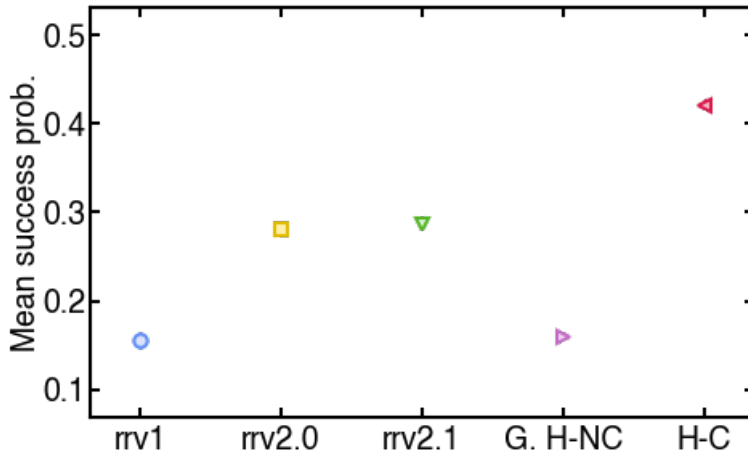


Figure 9.10: Mean Success Probability of token moving on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  ( $p_{load} = 0.6$ ) with greedy token isolation disabled for the different grid solvers, averaged over 500 instances.

We now look at the number of operations in solutions of individual instances in 8 different algorithms (Figure 9.11). On the one hand, the 3-approximation algorithm for extraction/implantation minimization outperforms the rest of the algorithms when it comes to extractions/implantations, though it suffers from the absence of a mechanism that can control displacements. On the other hand, the greedy HUNGARIAN-NOCOLAV algorithm minimizes total displacement operations but does poorly in terms of extractions/implantations. This data motivated the design of the original REDREC algorithm. REDREC v1 outperforms the greedy grid solvers in terms of extractions/implantations, though this

comes at the cost of extra displacements, with REDREC v1 incurring 10% more displacement operations on average. The improved REDREC algorithms fare slightly worse than the Hungarian-based grid solvers with ordering, though the differences are very small. With that being said, the Hungarian-based algorithms with ordering outperform the upgraded REDREC algorithms by a non-negligible margin when it comes to success probability, as can be seen in Figure 9.10, so the inferences that can be made by looking at overall operations are limited. What our algorithms did is that they bridged the gap between greedy HUNGARIAN-NOCOLAV with no greedy token isolation, our baseline, and the pareto-optimal solutions that have the minimum displacement possible.

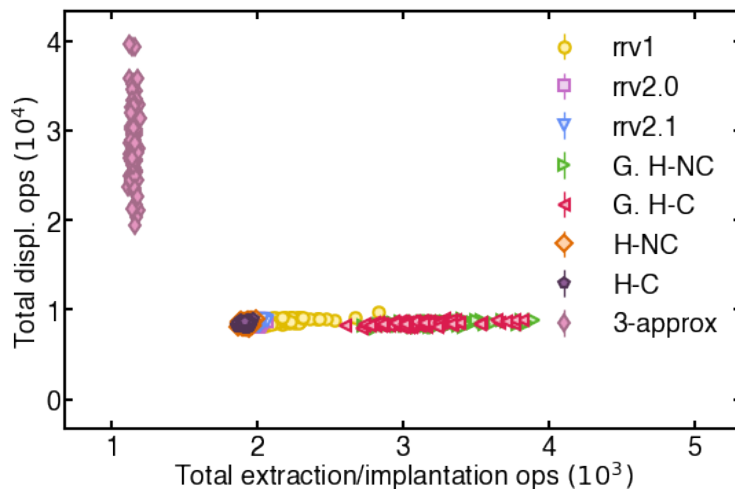


Figure 9.11: Operations executed in token moving on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) with greedy token isolation disabled for the different grid solvers, averaged over 500 instances.

The last discussion point for grid solvers on the CPU pertains to our heuristics and theoretical work. The data supports our claims regarding extraction/implantation operations: our greedy grid solvers may extract/implant the same token multiple times (Figure 9.12), but that is not the case for grid solvers with ordering (Figure 9.13). The experiments show that grid solvers with ordering perform very poorly compared to the 3-approximation algorithm for extraction/implantation minimization when it comes to extractions/implantations (Figure 9.14), and that is an artifact of the experiments being run using bidirectional COLAV, which minimizes overall displacement operations

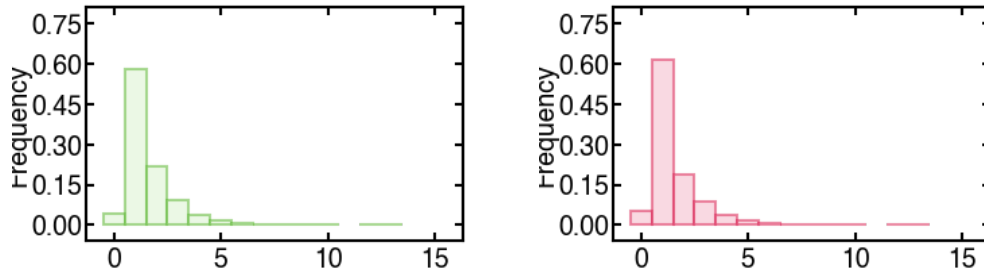


Figure 9.12: Distribution of extraction/implantation operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) with greedy token isolation disabled for greedy HUNGARIAN-NOCOLAV (left) and greedy HUNGARIAN-COLAV (right), averaged over 500 instances.

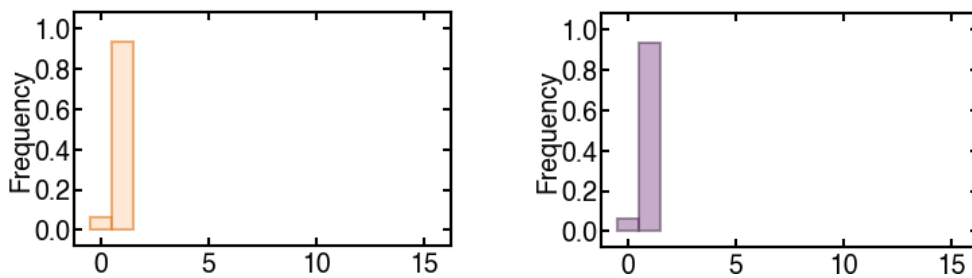


Figure 9.13: Distribution of extraction/implantation operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) with greedy token isolation disabled for HUNGARIAN-NOCOLAV (left) and HUNGARIAN-COLAV (right), averaged over 500 instances.

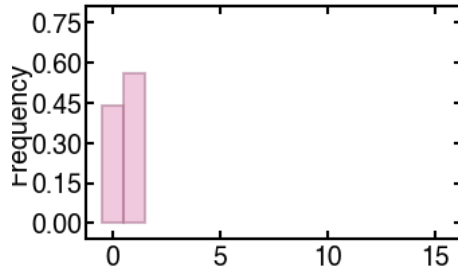


Figure 9.14: Distribution of extraction/implantation operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for the 3-approximation algorithm for extraction/implantation minimization, averaged over 500 instances.

Refined plots for the data presented in this section, in addition to extra plots that do not present information that is pertinent enough to include in this section, can be found in Appendix A.

## 9.4 Experiments on grid solvers on the GPU

Given that the previous section covered a detailed comparison of the grid solvers, we dedicate this section to assessing the running time of REDREC v2.0 on the GPU, and we focus on the average running time per batch, which we compare to the waveform streaming interval mentioned in Section 9.2. We run our experiments on grids of variable dimensions in increments of 4, from grids of size  $4 \times 4$  up to grids of size  $64 \times 64$ . In all the instances, the target region is centered and has a height that is equal to half that of the grid. Experiments showed that the surplus ratio does not change the results by much, so the presented table is for instances with a surplus ratio of 0.1. The experiments are limited to grid instances where  $G_H \geq G_W$ .

Table 9.1: REDREC v2.0 execution time on the GPU in milliseconds, averaged over 100 instances

| $H \backslash W$ | 8    | 12   | 16    | 20    | 24   | 28   | 32   |
|------------------|------|------|-------|-------|------|------|------|
| 8                | 1.84 |      |       |       |      |      |      |
| 12               | 2.13 | 2.72 |       |       |      |      |      |
| 16               | 1.72 | 2.47 | 3.12  |       |      |      |      |
| 20               | 1.97 | 2.18 | 3.065 | 3.4   |      |      |      |
| 24               | 1.91 | 2.41 | 2.88  | 3.668 | 3.68 |      |      |
| 28               | 1.71 | 2.61 | 2.97  | 3.39  | 3.9  | 4.58 |      |
| 32               | 1.88 | 2.12 | 2.84  | 3.15  | 4.00 | 4.31 | 4.64 |
| 36               | 1.76 | 2.17 | 2.92  | 3.61  | 3.95 | 4.21 | 4.88 |
| 40               | 1.76 | 2.56 | 2.86  | 3.24  | 4.40 | 3.95 | 5.06 |
| 44               | 1.99 | 2.56 | 2.54  | 3.39  | 3.89 | 4.57 | 3.99 |
| 48               | 1.84 | 2.23 | 2.78  | 3.51  | 3.33 | 3.72 | 4.10 |
| 52               | 1.96 | 2.35 | 2.49  | 3.59  | 3.39 | 4.17 | 4.19 |
| 56               | 1.78 | 2.26 | 2.70  | 3.18  | 3.26 | 3.85 | 3.99 |
| 60               | 1.77 | 2.49 | 2.67  | 3.32  | 3.79 | 4.06 | 3.86 |
| 64               | 1.73 | 2.27 | 2.65  | 3.11  | 3.91 | 3.42 | 4.26 |

| $G_H \backslash G_W$ | 36   | 40   | 44   | 48   | 52   | 56   | 60   | 64   |
|----------------------|------|------|------|------|------|------|------|------|
| 36                   | 5.21 |      |      |      |      |      |      |      |
| 40                   | 5.43 | 5.89 |      |      |      |      |      |      |
| 44                   | 4.65 | 5.38 | 5.54 |      |      |      |      |      |
| 48                   | 4.82 | 5.64 | 5.34 | 5.47 |      |      |      |      |
| 52                   | 4.87 | 5.01 | 5.6  | 6.07 | 6.06 |      |      |      |
| 56                   | 4.5  | 5.26 | 5.34 | 5.44 | 5.82 | 6.52 |      |      |
| 60                   | 4.42 | 5.67 | 5.80 | 5.48 | 5.87 | 6.61 | 7.23 |      |
| 64                   | 4.71 | 5.20 | 5.52 | 5.78 | 5.93 | 6.66 | 6.31 | 7.22 |

Table 9.2: REDREC v2.0 average execution time per batch on the GPU in milliseconds, averaged over 100 instances

| $H \backslash W$ | 8     | 12    | 16    | 20    | 24    | 28    | 32    |
|------------------|-------|-------|-------|-------|-------|-------|-------|
| 8                | 0.129 |       |       |       |       |       |       |
| 12               | 0.101 | 0.084 |       |       |       |       |       |
| 16               | 0.067 | 0.063 | 0.054 |       |       |       |       |
| 20               | 0.057 | 0.044 | 0.046 | 0.042 |       |       |       |
| 24               | 0.050 | 0.041 | 0.037 | 0.035 | 0.032 |       |       |
| 28               | 0.037 | 0.037 | 0.033 | 0.029 | 0.028 | 0.029 |       |
| 32               | 0.037 | 0.027 | 0.027 | 0.025 | 0.025 | 0.024 | 0.022 |
| 36               | 0.031 | 0.025 | 0.025 | 0.025 | 0.022 | 0.021 | 0.021 |
| 40               | 0.029 | 0.027 | 0.022 | 0.020 | 0.022 | 0.018 | 0.019 |
| 44               | 0.028 | 0.024 | 0.018 | 0.019 | 0.018 | 0.018 | 0.015 |
| 48               | 0.023 | 0.018 | 0.018 | 0.018 | 0.015 | 0.014 | 0.013 |
| 52               | 0.024 | 0.019 | 0.015 | 0.017 | 0.014 | 0.014 | 0.014 |
| 56               | 0.021 | 0.017 | 0.015 | 0.015 | 0.013 | 0.012 | 0.011 |
| 60               | 0.019 | 0.018 | 0.014 | 0.013 | 0.013 | 0.012 | 0.01  |
| 64               | 0.017 | 0.014 | 0.013 | 0.012 | 0.013 | 0.01  | 0.01  |

| $H \backslash W$ | 36    | 40    | 44    | 48    | 52    | 56    | 60    | 64    |
|------------------|-------|-------|-------|-------|-------|-------|-------|-------|
| 36               | 0.020 |       |       |       |       |       |       |       |
| 40               | 0.019 | 0.018 |       |       |       |       |       |       |
| 44               | 0.015 | 0.015 | 0.015 |       |       |       |       |       |
| 48               | 0.013 | 0.014 | 0.014 | 0.012 |       |       |       |       |
| 52               | 0.012 | 0.012 | 0.012 | 0.012 | 0.011 |       |       |       |
| 56               | 0.011 | 0.012 | 0.011 | 0.01  | 0.01  | 0.011 |       |       |
| 60               | 0.011 | 0.012 | 0.011 | 0.01  | 0.01  | 0.01  | 0.01  |       |
| 64               | 0.01  | 0.01  | 0.01  | 0.009 | 0.009 | 0.009 | 0.008 | 0.009 |

In terms of running time, parallel REDREC v2.0 is 10 times slower on average on the GPU than it is on the CPU (values not shown) across all grid dimension. This is to be expected, because we were not able to extract a lot of parallelism from the algorithm due to its serial logic. With that being said, the average running time per block batch matches the target interval of  $10\mu\text{s}$  for large grid dimensions ( $G_W \geq 52$ ,  $G_H \geq 52$ ). The usage of the current REDREC v2.0 parallel implementation is therefore only problematic for small grid sizes.

# CHAPTER 10

## CONCLUSION

In this thesis, we formally defined atom assembly with atom loss as a graph reconfiguration problem, which we call the MAXIMUM SUCCESS TOKEN MOVING problem. We worked on aggregating what we know about related problems into concrete, polynomial-time algorithms on paths and grids. On paths, we relied on displacement minimization as a heuristic for MSTM, and we introduced techniques that can be used to express the distribution of displacements across tokens. On grids, our focus was on the design of algorithms that are able to express the tradeoff between overall displacement operations and overall extraction/implantation operations, in an attempt to generalize the relevance of the algorithms for as large a class of quantum simulators as possible, given that loss functions are artifacts of the experimental setup. Our work is novel in that it is the first that encompasses the notion of the aforementioned tradeoff. We supplement our theoretical results with serial implementations and parallel implementations, as well as empirical data that backs the expected results behind our algorithm design.

### 10.1 Future work

In our theoretical work, we established that there cannot be a polynomial time solution for MLTM, as that would imply a polynomial time solution for MEITM. However, whether there exists a polynomial time algorithm that can generate a pareto-optimal solution that minimizes displacements remains an open problem. The possibility of generating any other pareto-optimal solution (except pareto-optimal solutions that minimize extractions/implantations) in isolation also remains an open problem, though we postulate that those two problems are hard.

Since implementation is an integral part of the work that went into this thesis, we propose improvements that can be made on this end. The current HUNGARIAN-COLAV and HUNGARIAN-REDIST algorithms are not useful



in practice, as they are not able to compete with the REDREC algorithms in terms of running time. The slowdown can be attributed to the frequent calls to the Floyd-Warshall algorithm. As such, looking into ways to specialize those two algorithms for centered target regions would be valuable, and may allow us to achieve an acceptable running time. If this fails, heuristics for APSP could be put to use as well.

Furthermore, the theory that underlies our Hungarian-based grid implementations works for general graphs. As such, the implementation of HUNGARIAN-COLAV can be modified to work with any graph, and not just grid graphs, as is the case now. This may be a venture that is worth looking into pending further experimental results.

In terms of gathering insights about the performance of our solvers, we were not able to include the results of the HUNGARIAN-COLAV variant that uses bounded COLAV due to time constraints. This variant is supposed to trace a curve in Figure 9.11 that takes on the shape of the curve of pareto-optimal solutions, as seen in Figure 5.2.

Finally, the work on parallelizing grid solvers is primitive: REDREC v2.1 was hastily designed to remedy REDREC v2.0's serialization; we postulate that REDREC v2.0's parallel implementation could be modified to precompute dependencies between column pairs, which would consequently make it possible for us to extract some parallelism by solving mutually exclusive column pairs in parallel, meaning that serialization would be reserved for column pairs that share a donor or a receiver. Furthermore, the experimental results show negligible differences in terms of success probability between REDREC v2.0 and REDREC v2.1: implementing REDREC v2.1 is therefore worthwhile.

# APPENDIX A

## FIGURES

### A.1 Experiments on path solvers on the CPU

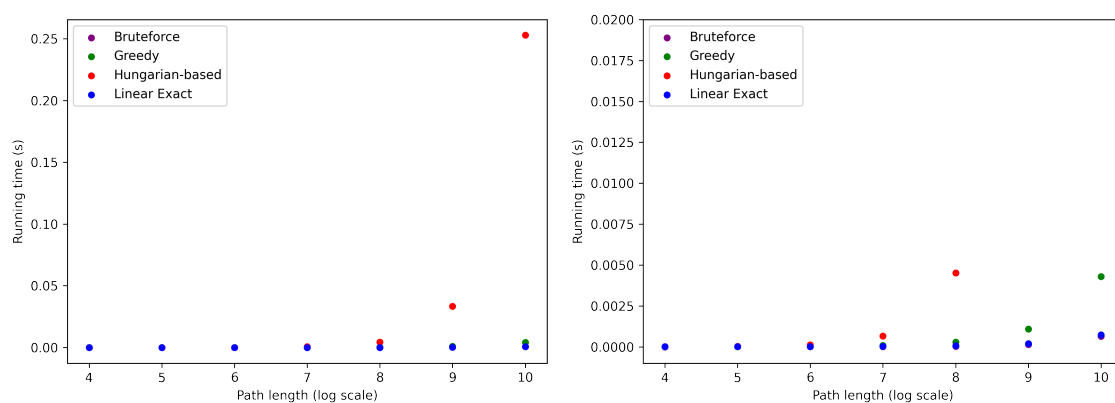


Figure A.1: CPU running times for path solvers on centered targets with no surplus (unbatched token moving).

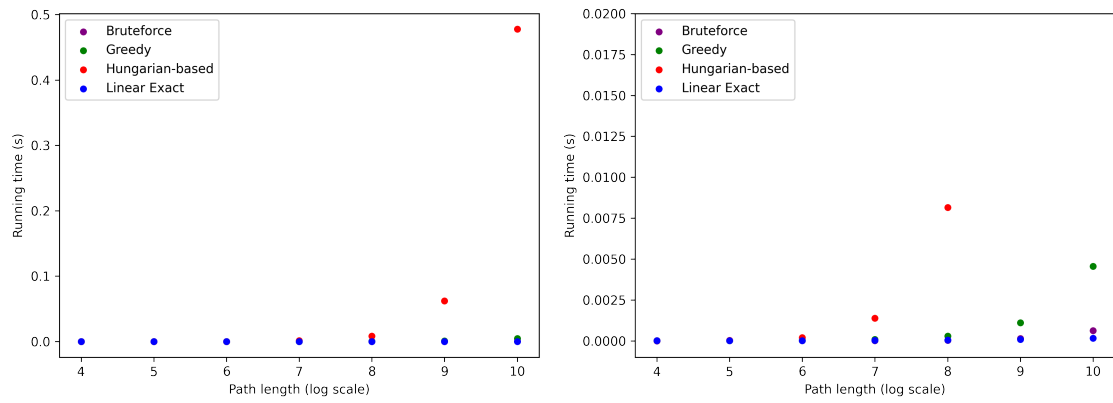


Figure A.2: CPU running times for path solvers on centered targets with a surplus ratio of 0.1 (unbatched token moving).

## A.2 Experiments on path solvers on the GPU

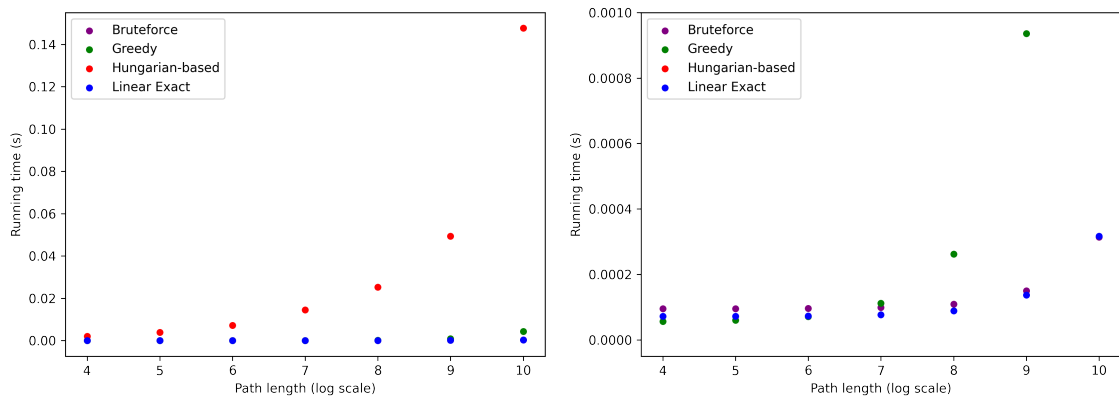


Figure A.3: GPU running times for path solvers on centered targets with no surplus (unbatched token moving).

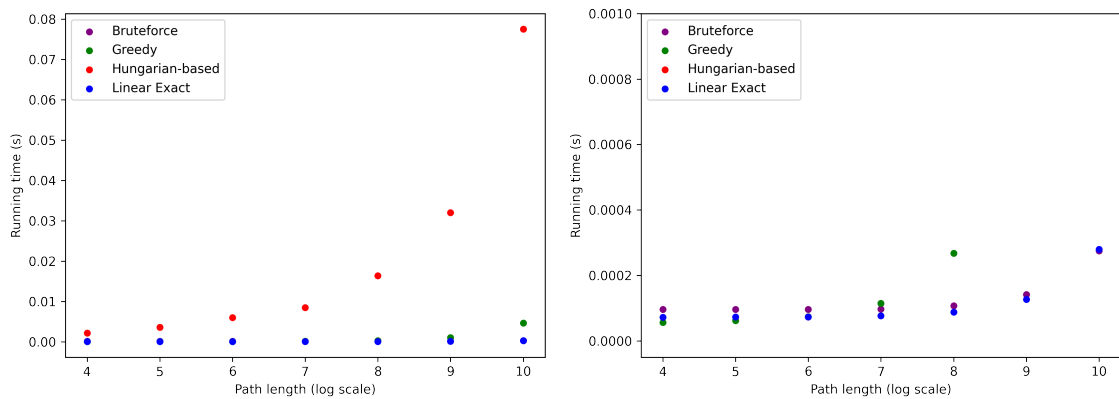


Figure A.4: GPU running times for path solvers on centered targets with a surplus ratio of 0.1 (unbatched token moving).

## A.3 Experiments on grid solvers on the CPU

### A.3.1 *Distribution of extraction/implantation operations across tokens*

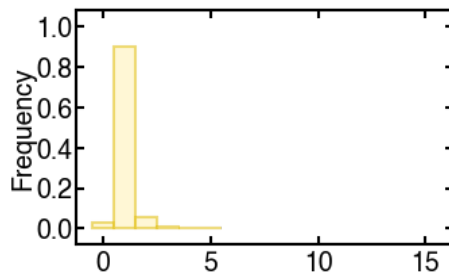


Figure A.5: Distribution of extraction/implantation operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for REDREC v1, averaged over 500 instances.

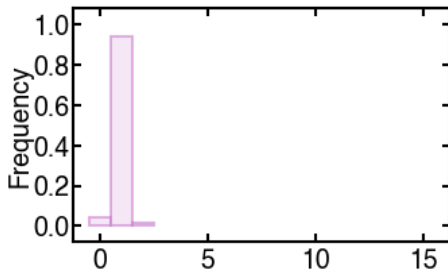


Figure A.6: Distribution of extraction/implantation operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for REDREC v2.0, averaged over 500 instances.

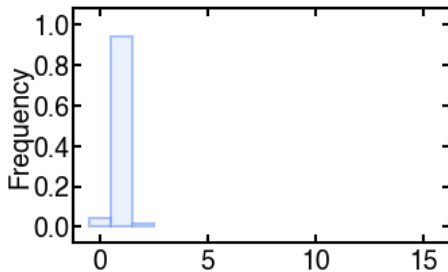


Figure A.7: Distribution of extraction/implantation operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for REDREC v2.1, averaged over 500 instances.

### A.3.2 *Distribution of displacement operations across tokens*

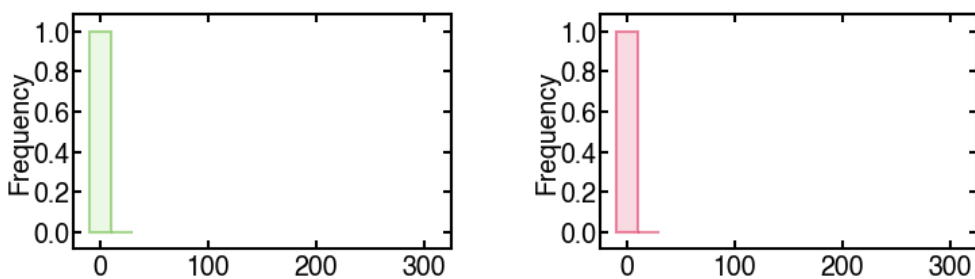


Figure A.8: Distribution of displacement operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) with greedy token isolation disabled for greedy HUNGARIAN-NOCOLAV (left) and greedy HUNGARIAN-COLAV (right), averaged over 500 instances.

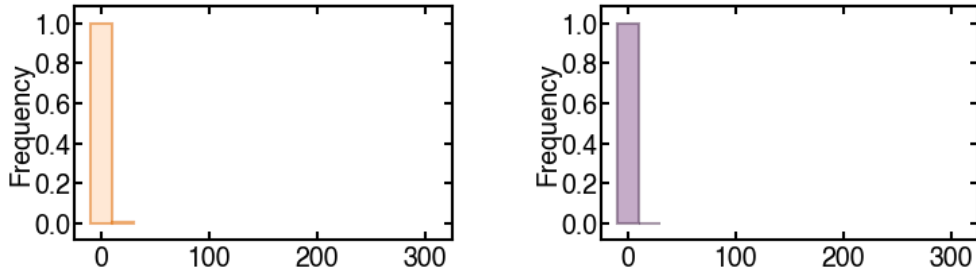


Figure A.9: Distribution of displacement operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) with greedy token isolation disabled for HUNGARIAN-NOCOLAV (left) and HUNGARIAN-COLAV (right), averaged over 500 instances.

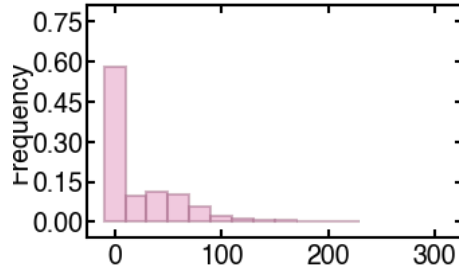


Figure A.10: Distribution of displacement operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for the 3-approximation algorithm for extraction/implantation minimization, averaged over 500 instances.

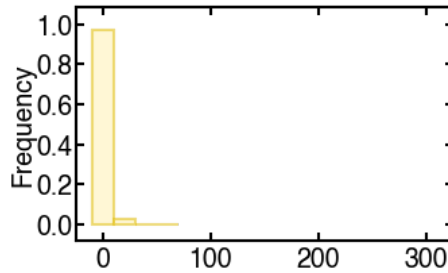


Figure A.11: Distribution of displacement operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for REDREC v1.0, averaged over 500 instances.

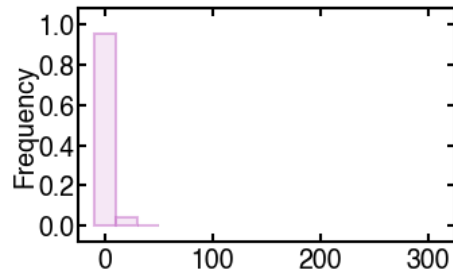


Figure A.12: Distribution of displacement operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for REDREC v2.0, averaged over 500 instances.

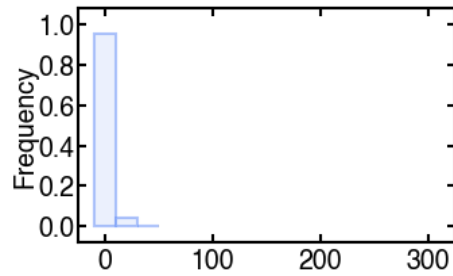


Figure A.13: Distribution of displacement operations across tokens on grids of size  $64 \times 32$  and targets of size  $32 \times 32$  (no surplus) for REDREC v2.1, averaged over 500 instances.

# BIBLIOGRAPHY

- [1] W. Woolsey Johnson and W. Story, “Notes on the “15” puzzle,” *Amer. J. Math.*, vol. 2, pp. 393–404, 1879.
- [2] T. Ito, E. D. Demaine, N. J. Harvey, C. H. Papadimitriou, M. Sideri, R. Uehara, and Y. Uno, “On the complexity of reconfiguration problems,” *Theoretical Computer Science*, vol. 412, no. 12-14, pp. 1054–1065, 2011.
- [3] I. Bloch, J. Dalibard, and S. Nascimbene, “Quantum simulations with ultracold quantum gases,” *Nature Physics*, vol. 8, no. 4, pp. 267–276, 2012.
- [4] L. Henriot, L. Beguin, A. Signoles, T. Lahaye, A. Browaeys, G.-O. Reymond, and C. Jurczak, “Quantum computing with neutral atoms,” *Quantum*, vol. 4, p. 327, 2020.
- [5] M. Khazali and K. Mølmer, “Fast multiqubit gates by adiabatic evolution in interacting excited-state manifolds of rydberg atoms and superconducting circuits,” *Physical Review X*, vol. 10, no. 2, p. 021054, 2020.
- [6] C. Gross and I. Bloch, “Quantum simulations with ultracold atoms in optical lattices,” *Science*, vol. 357, no. 6355, pp. 995–1001, 2017.
- [7] A. Cooper. private communication, 2022.
- [8] H. W. Kuhn, “The hungarian method for the assignment problem,” *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.
- [9] R. M. Karp and S.-Y. R. Li, “Two special cases of the assignment problem,” *Discrete Mathematics*, vol. 13, no. 2, pp. 129–142, 1975.
- [10] NVIDIA, P. Vingelmann, and F. H. Fitzek, “CUDA, release: 10.2.89,” 2020.
- [11] G. Călinescu, A. Dumitrescu, and J. Pach, “Reconfigurations in graphs and grids,” *SIAM Journal on Discrete Mathematics*, vol. 22, no. 1, pp. 124–138, 2008.
- [12] B. Cimring and A. Cooper. private communication, 2022.



- [13] R. A. Hearn and E. D. Demaine, “Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation,” *Theoretical Computer Science*, vol. 343, no. 1-2, pp. 72–96, 2005.
- [14] A. Cooper, S. Maaz, A. E. Mouawad, and N. Nishimura, “Parameterized complexity of reconfiguration of atoms,” in *International Conference and Workshops on Algorithms and Computation*, pp. 263–274, Springer, 2022.
- [15] A. Cooper, J. P. Covey, I. S. Madjarov, S. G. Porsev, M. S. Safronova, and M. Endres, “Alkaline-earth atoms in optical tweezers,” *Physical Review X*, vol. 8, no. 4, p. 041055, 2018.
- [16] S. Ebadi, T. T. Wang, H. Levine, A. Keesling, G. Semeghini, A. Omran, D. Bluvstein, R. Samajdar, H. Pichler, W. W. Ho, *et al.*, “Quantum phases of matter on a 256-atom programmable quantum simulator,” *Nature*, vol. 595, no. 7866, pp. 227–232, 2021.
- [17] A. Ashkin, “Acceleration and trapping of particles by radiation pressure,” *Physical review letters*, vol. 24, no. 4, p. 156, 1970.
- [18] A. Ashkin, J. M. Dziedzic, J. E. Bjorkholm, and S. Chu, “Observation of a single-beam gradient force optical trap for dielectric particles,” *Optics letters*, vol. 11, no. 5, pp. 288–290, 1986.
- [19] S. Chu, J. Bjorkholm, A. Ashkin, and A. Cable, “Experimental observation of optically trapped atoms,” *Physical review letters*, vol. 57, no. 3, p. 314, 1986.
- [20] A. Ashkin and J. M. Dziedzic, “Optical trapping and manipulation of viruses and bacteria,” *Science*, vol. 235, no. 4795, pp. 1517–1520, 1987.
- [21] K.-N. Schymik, V. Lienhard, D. Barredo, P. Scholl, H. Williams, A. Browaeys, and T. Lahaye, “Enhanced atom-by-atom assembly of arbitrary tweezer arrays,” *Physical Review A*, vol. 102, no. 6, p. 063107, 2020.
- [22] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *International conference on high-performance computing*, pp. 197–208, Springer, 2007.
- [23] D. Merrill, M. Garland, and A. Grimshaw, “High-performance and scalable gpu graph traversal,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 1, no. 2, pp. 1–30, 2015.
- [24] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: A high-performance graph processing library on the gpu,” in *Proceedings of*

*the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 1–12, 2016.

- [25] S. Diab, M. G. Olabi, and I. El Hajj, “Ktrusseexplorer: exploring the design space of k-truss decomposition optimizations on gpus,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2020.
- [26] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, “Parallel k-clique counting on gpus,” in *Proceedings of the 36th ACM International Conference on Supercomputing*, pp. 1–14, 2022.
- [27] P. Yamout, K. Barada, A. Jaljuli, A. E. Mouawad, and I. El Hajj, “Parallel vertex cover algorithms on gpus,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 201–211, IEEE, 2022.
- [28] R. Diestel, *Graph Theory*. Springer Publishing Company, Incorporated, 5th ed., 2017.
- [29] J. Edmonds and R. M. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems,” *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.
- [30] M. Ivanov, “АВенгерский алгоритм решения задачи о назначениях.” [http://e-maxx.ru/algo/assignment\\_hungary](http://e-maxx.ru/algo/assignment_hungary), Aug 2012.
- [31] R. W. Floyd, “Algorithm 97: shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.
- [32] W.-m. W. Hwu, D. B. Kirk, and I. El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.
- [33] P. A. Lopes, S. S. Yadav, A. Ilic, and S. K. Patra, “Fast block distributed cuda implementation of the hungarian algorithm,” *Journal of Parallel and Distributed Computing*, vol. 130, pp. 50–62, 2019.
- [34] J. Munkres, “Algorithms for the assignment and transportation problems,” *Journal of the society for industrial and applied mathematics*, vol. 5, no. 1, pp. 32–38, 1957.
- [35] D. Merrill, “CUB.” <https://dx.doi.org/10.5281/zenodo.6868125>, 2022.