Multi-Agent Path Finding for Reversible Conveyors

Master's Thesis
submitted to the
Software Modeling and Verification Group
Computer Science Department
RWTH Aachen University

by **Dominic Sebastian Meiser**

Supervisor:
M.Sc. Roy Hermanns

First examiner:
apl. Prof. Dr. Thomas Noll

Second examiner:
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Registration Date:
30th April 2025

Submission Date:
30th October 2025

Communicated By:
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Abstract

Quantum computers contain qubits that need to be close to each other for certain operations. The SpinBus architecture is a quantum computer architecture that allows shuttling these qubits around on reversible conveyors. The connectivity, i.e. the number of qubits that can be close to each other, is limited to 2. This means that qubits need to be moved frequently. Also, a conveyor can have more than one qubit on it, which means that qubits cannot move independently. A compiler for this architecture needs to plan moving qubits with the specifics of the architecture in mind.

This thesis investigates Q-MAPF, an extension of Multi-Agent Path Finding (MAPF) for application in qubit routing. We show poor performance in prior work, in terms of both time and memory usage, and explore ways to improve the performance. First, we approach the problem by exploiting symmetries, which turns out to not improve the performance. Then, we work on a more optimal constraint generation for CBS, an algorithm that can solve MAPF problems, which improves performance: The execution time is improved up to 2.5 times, and we can solve more problems than previously due to lower memory usage. Finally, we extend the problem to Q-TAPF. Combined Target and Path Finding (TAPF) is an extension of MAPF where the destination of an agent is part of the optimisation problem. This has the potential to improve the quality of the result whilst reducing the runtime. While Q-TAPF can show improvements of halving the execution time, it does not do so consistently.

Contents

CHAPTER 1

Introduction

Quantum computers promise to solve some problems more efficiently than the wide-spread classical computers we all use on a daily basis today. Therefore, researchers are busy improving the existing technology and exploring new ideas.

One of the challenges that arrise when scaling quantum computers to higher numbers of qubits is the amount of control wires needed to control the quantum computer. To combat this, quantum architectures with shared controls are being developed. One of those is the Crossbar architecture [16]. These quantum architectures with shared controls introduce challenges for compilers, due to the fact that changing one wire likely influences more than one qubit. For the Crossbar architecture, a compiler has been developed based on heuristics [20, 19].

An experimental new quantum architecture called SpinBus [10] is currently under development, and hopes to scale to hundreds of qubits. Like Crossbar, it also is a quantum architecture that allows physically moving qubits. However, the controls work quite differently compared to those of Crossbar: Where qubits are placed into a grid structure in the Crossbar architecture, in which they are moved up to a barrier, in SpinBus, qubits are placed into unit cells linked by busses and there are no barriers on those busses. This introduces new challenges for quantum compilers.

Unit cells and busses form the buildin blocks of the SpinBus architecture. Each unit cell allows the execution of either two single-qubit or one two-qubit quantum gate. The busses connect unit cells together, so we can move qubits from one unit cell to another. However, there is only one control signal per bus, and two control signals per unit cell. We call these sections controlled by a single control signal a *conveyor*. Each conveyor can be moved in either direction, forward and reverse, or idle. But, this decision always influences all qubits on the conveyor, making it harder to coordinate the qubit movement. Also, each qubit need to maintain a minimum distance from any other qubit. We will explain this further in chapter 3.

Already without these conveyors, routing is a hard problem. Multi-Agent Path Finding (MAPF) is a well-studied problem [30] that was shown to be NP-hard [36]. MAPF is very versatile and has applications in several fields, including video games [27], airport planning [18] and warehouse automation [11]. Several algorithms have been developed that solve this problem, including variations of A* and an algorithm

called Conflict-Based Search (CBS) [23]. We discuss MAPF and the algorithms that can solve it in more detail in chapter 2.

Prior work was done to create a compiler for the SpinBus architecture [8]. The main problem when compiling a quantum circuit for SpinBus is qubit routing. An extension of MAPF was proposed in [8], alongside an extension for the Conflict-Based Search (CBS) algorithm [23] called QCBS, which, alongside the SpinBus architecture, is the topic of chapter 3.

In its current form, the QCBS algorithm is very slow and memory intensive. This was already observed by the authors of [8] and reproduced by benchmarks in this thesis. We explore ways that have the potential to improve the performance of qubit routing for SpinBus. First, we extend A* with Operator Decomposition [29] for the extended MAPF problem. Then, we explore two ideas that improved CBS and extend them to QCBS: Corridor symmetries [15] in chapter 4 and disjoint splitting [12] in chapter 5.

In chapter 6, we will improve on the target assignment for SpinBus. This was already studied for the Crossbar architecture [32], but in a different way: They introduced SWAP operations, and showed that an optimal assignment can reduce the necessary overhead in SWAP operations by an order of magnitude compared to a random assignment. We previously used a basic algorithm instead of a random one. Still, it can be improved. We merge target assignment and path finding into a combined algorithm called TAPF, whose extension for SpinBus we call Q-TAPF, and provide a modification of CBS to solve Q-TAPF.

CHAPTER 2

Multi-Agent Path Finding

Multi-Agent Path Finding (MAPF) describes the problem of finding an (optimal) path on a graph for multiple agents, without causing any conflicts between these agents. This problem was shown to be NP-hard [36]. It can be applied to a wide variety of problems, from NPCs[1] in video games [27] to robots in large warehouses [11]. We will use robots in warehouses as an example.

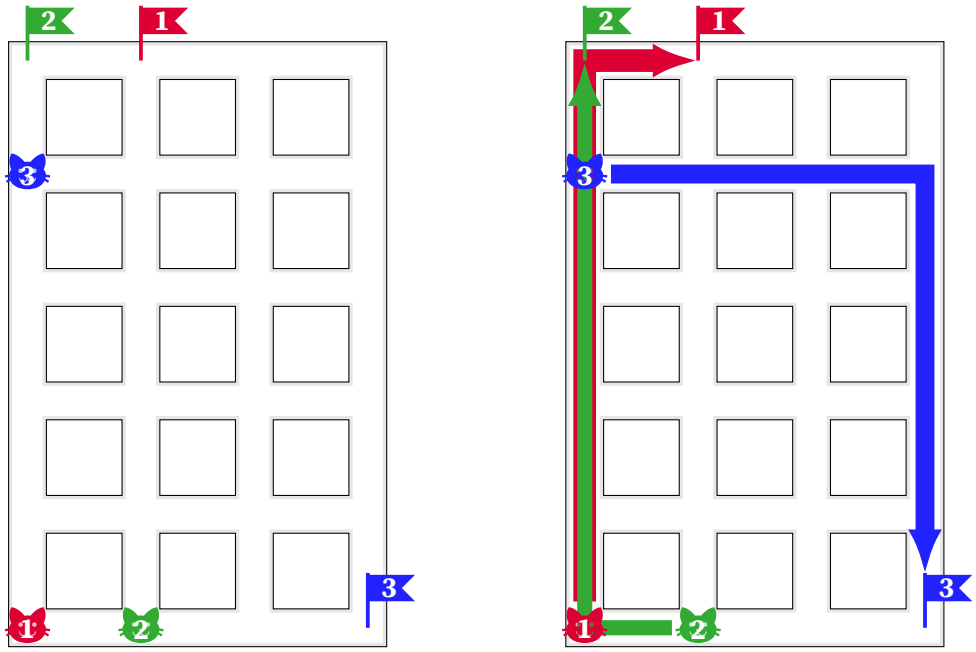
Imagine a warehouse built such that there are square sections filled with goods separated by "streets" for the robots to drive on. This forms a grid which we can interpret as a graph, where each intersection is represented via a vertex and each street is represented via an edge. Each street, i.e. each edge, has unit length, so it takes a robot the same amount of time to traverse each street. Our streets and intersections are narrow, meaning that two robots travelling in opposite directions along the same street at the same time will inevitably crash head-on into each other. On an intersection, in addition to head-on collisions, they get the opportunity to crash into the side of another robot.

Figure 2.1a shows an example with 3 robots on a 6×4 grid. Each robot is assigned a number between 1 and 3. The start locations are marked with cat faces and the destinations are marked with flags, each containing the agent's number. The MAPF problem now is to find a path for all robots such that they do not collide and reach their destinations with minimal cost. A measurement for the cost could be, for example, the total time it takes for the robots to finish their movement. There are multiple solutions for this example, one of which is shown in Figure 2.1b.

There exist several variations of the MAPF problem. An overview over MAPF and its variations can be found in [31]. For this thesis, we will use the variation described below. This variation extends the one used in [8].

A MAPF problem has $k \geqslant 1$ agents, which we number from 1 through k. The problem operates on an undirected weighted graph $G = (V, E)$ with weight function $w : V \cup E \to \mathbb{N}$. Each agent has a start location that is assigned to it by the function $s : \{1, \dots, k\} \to V$, and similarly $d : \{1, \dots, k\} \to V$ assigns destinations to the agents.

[1] A non-player character (NPC) is a character in video games whose behaviour and movement is controlled by software



(a) A MAPF problem **(b)** An optimal solution

Figure 2.1: Example MAPF problem

Each agent wants to travel from their start location to their destination, minimising the cost of their respective path. The cost of each agent's individual path is computed using the weight function w. However, interactions between agents may cause conflicts that must not occur. Our MAPF variation has two types of conflicts, vertex and edge conflicts, which we define later. These conflicts might prevent two or more agents from choosing a path with minimal cost. We will later introduce different objective functions to judge paths for all agents.

Since G is undirected, we treat (v, u) and (u, v) as equal. If $(v, u) \in E$, then $(u, v) \in E$ must also be true. We also require G to be a simple graph. That means that self-loops, i.e. edges (v, v) that start and end at the same vertex, are not allowed. We will later see that this case is handled by wait actions.

Further, the function $N : V \to 2^V$ returns the neighbors of the vertex in the graph G. If we just need the number, the function $\text{degree} : V \to \mathbb{N}_0$ returns the *degree*, i.e. the number of neighbors, of a vertex.

The plan for an agent consists of actions. An *action* is a partial function $a : V \dashrightarrow V$. There are two permitted actions: Move and wait. A *move action* means that an agent moves along an edge $(u, v) \in E$, so the action is $a : u \mapsto v$. Contrary, a *wait action* does not move, so $a : v \mapsto v$ for all $v \in V$. A *plan* of length ℓ is a tuple $(a_1, \dots, a_\ell)$ of consecutive actions. If that plan, starting at the start location of an agent, ends at the destination of that agent, it forms a *single-agent plan*.

Definition 2.1. *An **action** is a partial function $a : V \dashrightarrow V$, either a **move action** $a : u \mapsto v$ for $(u, v) \in E$, or a **wait action** $a : v \mapsto v$ for all $v \in V$.*

Definition 2.2. *Let $\pi = (a_1, \dots, a_\ell)$ be a sequence of actions for agent i. If for each $1 \leqslant t \leqslant \ell$, $(a_t \circ \dots \circ a_1)(s(i))$ is defined, and $(a_\ell \circ \dots \circ a_1)(s(i)) = d(i)$, we call π a **single-agent plan** for agent i. We write $\pi[0] := s(i)$ and $\pi[t] := (a_t \circ \dots \circ a_1)(s(i))$ for $1 \leqslant t \leqslant \ell$ and $|\pi| := \ell$. For the destination of a plan, we write $d(\pi) := \pi[|\pi|]$.*

Once an agent has reached its destination, we need to decide what happens to that agent. There are two common ways to deal with finished agents, i.e. agents that have reached their destination. The first, *disappear-at-target*, assumes that the agent vanishes once it reaches its destination and does not incur any further cost or conflict. The second, *stay-at-target*, assumes that the agent stays at its destination at least until all other agents finished, without incurring any further cost. The agent can however still cause conflicts with other agents. As a consequence, it is possible that the optimal solution has an agent, after it reached its destination, temporarily move away from it again for another agent to pass through.

The terms *disappear-at-target* and *stay-at-target* are well-known and coined in literature [31]. However, the term *stay-at-target* has also been used to refer to a third option in [8]. In that third option, agents incur normal waiting costs at their destination until all agents have reached their respective destinations, in addition to causing conflicts. In order to distinguish it from the *stay-at-target* option we already introduced, we refer to this option as *wait-at-target*.

Definition 2.3. *For a single-agent plan, we define $\pi[t]$ for $t > |\pi|$ assuming*

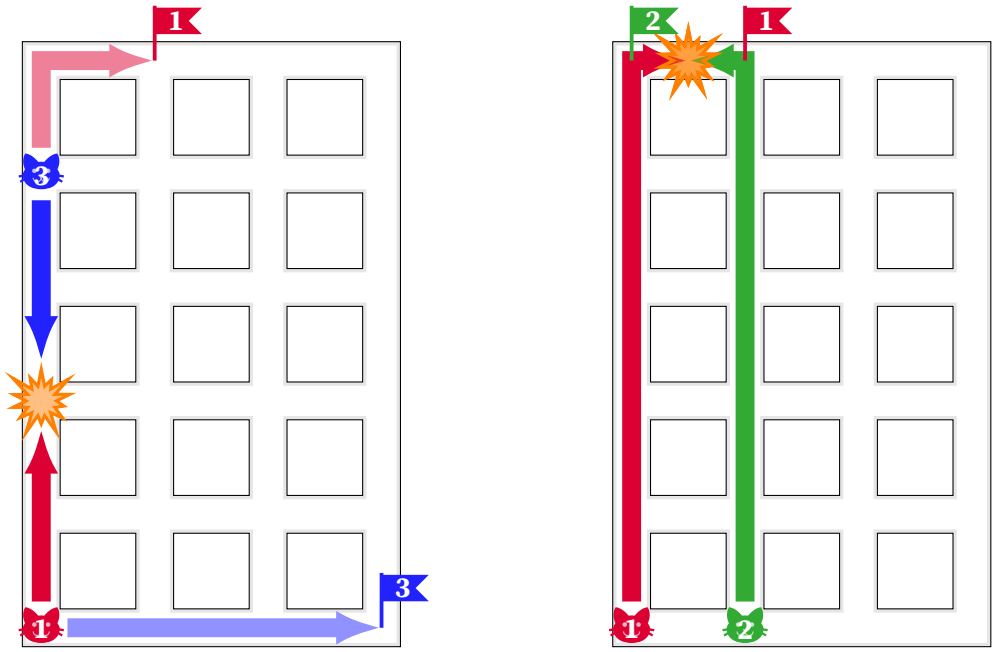
- ***disappear-at-target**: $\pi[t] := \text{👻}$ where $\text{👻} \notin V$ is a marker with $\text{👻} \neq \text{👻}$, marking the agent as disappeared.*
- ***stay-at-target** or **wait-at-target**: $\pi[t] := d(\pi)$, which is equivalent to the agent's destination*

A *vertex conflict* occurs when two agents want to occupy the same vertex in the graph at the same time. If you think about our robots again, this conflict occurs whenever two robots enter the same intersection at the same time. Since the streets are too narrow for two lanes of robots, this means the robots are not travelling in the same direction, and since the intersections are also narrow, they will inevitably crash. You can see an example of a vertex conflict in Figure 2.2a: The agents 1 and 3 are planned using two shortest paths that collide at an intersection, i.e. at a vertex.

Definition 2.4. *Let π and π' be two single-agent plans. If there exists $0 \leqslant t \leqslant \max\{|\pi|, |\pi'|\}$ s.t. $\pi[t] = \pi'[t]$, there is a **vertex conflict** between the two plans.*

An *edge conflict*[2] occurs when two agents want to traverse the same edge at the same time in opposing directions. Looking at our robots with our narrow streets, an edge conflict means that the two robots involved in the conflict crash head-on into

[2]While *edge conflict* is a common name for this conflict, [31] uses the name *swapping conflict*. Since *swapping* is a term already coined in the context of quantum computing, we will use the name *edge conflict* exclusively.



(a) A vertex conflict **(b)** An edge conflict

Figure 2.2: Example MAPF problem with conflicts

each other. You can see an example of an edge conflict in Figure 2.2b: The agents 1 and 2 are planned using two shortest paths that collide between two intersections, i.e. on an edge.

Definition 2.5. *Let π and π' be two single-agent plans. If there exists $0 \leqslant t < \max\{|\pi|, |\pi'|\}$ s.t. $\pi[t] = \pi'[t+1] \wedge \pi[t+1] = \pi'[t]$, there is an **edge conflict** between the two plans.*

Now that we have talked about single agent plans as well as the conflicts between them, we can start to talk about multi-agent plans, which we call *joint plans*. A joint plan itself is just a tuple of all the single-agent plans. What we want is a *valid* joint plan. That requires no conflicts to occur between any two single-agent plans. In the context of MAPF, "conflicts" refers to vertex and edge conflicts. In subsequent chapters, we will encounter other conflicts, so the exact meaning of *valid* is always dependent on the context.

Definition 2.6. *Let $\pi_1, \dots, \pi_k$ be single-agent plans. The tuple $\Pi = (\pi_1, \dots, \pi_k)$ is called a **joint plan**.*

Definition 2.7. *A joint plan is called **valid** w.r.t. some conflict types if there are no conflicts of the specified types between any two different single-agent plans.*

If we go back to the example problem in Figure 2.1a, we can try to find an optimal solution. We can solve the vertex conflict from Figure 2.2a by choosing a different path for agent 3: Instead of going down and then right, agent 3 can instead go right first and then down. The cost of the solution hasn't changed, but now it is no longer in

the way of agent 1. The edge conflict between the agents 1 and 2 (Figure 2.2b) can be solved in a similar fashion: We can choose a different path for agent 2 so that it doesn't conflict with the plan of agent 1 anymore. This solution is visualised in Figure 2.1b.

MAPF searches for an optimal solution while avoiding conflicts. For a single agent, the cost of its single-agent plan is the sum of the weight of the vertices that the agent waits at and the edges that agent travels along. For multiple agents, we need to make a choice on how to compute the cost of the joint plan, i.e. we need an objective function $\oplus$. The most straight forward way would be to build the sum of all the single-agent plan costs. Another option is to use the *makespan*, for which we compare the costs of each single-agent plan and take the maximum as the cost for the joint plan.

Definition 2.8. *The **cost/weight of a single-agent plan** $\pi = (a_1, \dots, a_\ell)$ is the sum of the cost of its actions*

$$w(\pi) := \sum_{1 \leqslant t \leqslant \ell} \begin{cases} w(\pi[t]) & \text{if } \pi[t] = \pi[t-1] \\ w((\pi[t-1], \pi[t])) & \text{otherwise} \end{cases}$$

Definition 2.9. *Let $\mathbb{N} \oplus \mathbb{N} \to \mathbb{N}$ be an objective function. The **cost/weight of a joint plan** $\Pi = (\pi_1, \dots, \pi_k)$ is, assuming*

- ***disappear-at-target** or **stay-at-target**: $w(\Pi) := w(\pi_1) \oplus \dots \oplus w(\pi_k)$*
- ***wait-at-target**: Let $\ell_{max} := \max\{|\pi_1|, \dots, |\pi_k|\}$. Then,*

$$w(\Pi) := \bigoplus_{1 \leqslant i \leqslant k} (w(\pi_i) + (\ell_{max} - |\pi_i|) \cdot w(d(\pi_i)))$$

Definition 2.10. *Let $G = (V, E)$ be an undirected graph with weights $w : V \cup E \to \mathbb{N}$, and $\mathbb{N} \oplus \mathbb{N} \to \mathbb{N}$ an objective function. Further, let there be k agents with start locations $s : \{1, \dots, k\} \to V$ and destinations $d : \{1, \dots, k\} \to V$. Make a choice whether you assume disappear-at-target, stay-at-target or wait-at-target. Then, **MAPF** is the optimisation problem: Find a valid joint plan Π (w.r.t. vertex and edge conflicts) s.t. there is no valid joint plan Π' with $w(\Pi') < w(\Pi)$.*

2.1 Solve Single Agents

Let us start solving MAPF by talking about single agents. The well-known A* algorithm [6] is perfectly suited to plan a single agent. A* computes a shortest path along a graph from a start vertex to a destination using a heuristic. The destination can either be a single vertex, or a list of vertices, in which case A* will pick the vertex with the shortest path from the list of destinations. For now, we can ignore the second case as our agents will only ever have one single destination.

A* works by maintaining two lists of vertices: An *open* and a *closed* list. The open list contains nodes in order of some evaluation function $\hat{f}$. Initially, the start node is in the open list, while the closed list starts off empty. In each step, we choose the

smallest vertex (ranked by $\hat{f}$) from the open list, and add it to the closed list. If that vertex is our destination, we found a shortest path. Otherwise, we *expand* the vertex: For each neighbor of the vertex, compute $\hat{f}$. If the neighbor is not in the closed list, or the value of $\hat{f}$ has decreased since it was put in the closed list, mark that neighbor as open. If the open list is empty before the destination is reached, there is no path. To ensure that we can reconstruct the path, A* also keeps a list of parents, which get updated whenever we put a vertex into the closed list.

The evaluation function $\hat{f}$ for a vertex v combines the cost of the shortest path currently known to the algorithm from the start to v, $\hat{g}(v)$, and the heuristic cost, $h(v)$, of the path from v to the destination: $\hat{f}(v) = \hat{g}(v) + h(v)$. Note that A* refines the value of $\hat{g}(v)$ by itself, while it never adjusts the heuristic. For the admissibility of A*, it is important to never over-estimate the heuristic costs.

2.2 Solve MAPF using A*

We can extend A* to solve path-finding not only for the single- but also for the multi-agent case. For this, we use a simple approach which has been suggested multiple times [29, 30, 8]: Use k-tuples of vertices, which are also called *configurations*, as the search space for A*. Each configuration describes the location of all agents. By executing one action per agent, we can advance from one configuration to the next. This forms an exponential state space which we call the *joint state space*.

Definition 2.11. *The tuple $(v_1, \dots, v_k) \in V^k$ is called a **configuration**. The i-th element of the tuple corresponds to the location of the i-th agent.*

2.2.1 Construction of the joint state space

To ensure that the solution of A* corresponds to a valid joint plan, we need to restrict the search space slightly. Avoiding vertex conflicts is simple: We do not allow any configuration where two vertices are identical. Since each configuration represents all agent's location at a timestep, this removes exactly those states that would cause a vertex conflict. For edge conflicts, we instead need to restrict the neighbors of each state. No two states may be neighbors where two agents swap locations.

The initial state is derived from the initial vertices of the agents. Thus, we get $(s(1), \dots, s(k))$ as the initial state. Similarly, there is one target state, $(d(1), \dots, d(k))$.

For the costs, a naïve approach might be to combine the costs of each single-agent action using the objective function. However, this will quickly lead to an over-approximation: Say there are two agents, i and $i\!\!\ddot{}$. In the first step, agent i takes an action with cost 2, and agent $i\!\!\ddot{}$ an action with cost 1. In the next step, agent i takes an action with cost 1 while agent $i\!\!\ddot{}$ takes an action with cost 2. The cost of the joint plan is thus 3 when using the makespan objective function. But, if we applied makespan to each step, both steps would have cost 2, which adds up to 4. This is clearly an over-approximation and must thus be avoided.

Instead, we keep track of each agent's cost. A* will then temporarily combine the single agent costs using the objective function to determine the state with lowest costs to explore next. Formally, $(w_1, \dots, w_k) < (w'_1, \dots, w'_k)$ iff $w_1 \oplus \dots \oplus w_k < w'_1 \oplus \dots \oplus w'_k$.

Definition 2.12. *The **joint state space** for MAPF on a graph $G = (V, E)$ with k agents is an undirected, weighted graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ with weight function $w_\mathcal{G}$ where*

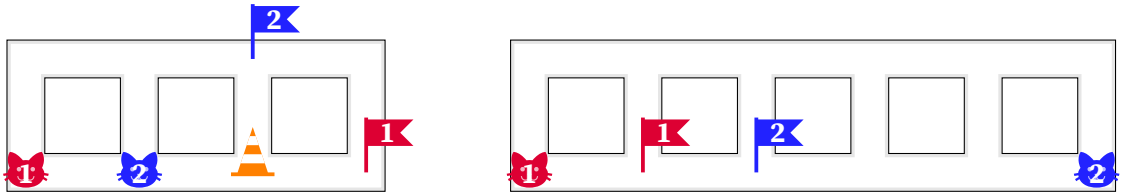
- $V_\mathcal{G} := \{(v_1, \dots, v_k) \in V^k \mid \forall 1 \leqslant i \neq i \leqslant k : v_i \neq v_i\}$
- $((u_1, \dots, u_k), (v_1, \dots, v_k)) \in E_\mathcal{G}$ *iff*
  1. $(u_i, v_i) \in E$ *for all* $1 \leqslant i \leqslant k$
  2. *The joint plan* $\Pi = (\pi_1, \dots, \pi_k)$*, made up of the single-agent plans* $\pi_i = (u_i \mapsto v_i)$ *of length* 1*, is valid, i.e. there are no edge conflicts*
- $w_\mathcal{G}((u_1, \dots, u_k), (v_1, \dots, v_k)) := (w_1, \dots, w_k)$ *where* $w_i := \begin{cases} w(u_i) & \text{if } u_i = v_i \\ w((u_i, v_i)) & \text{otherwise} \end{cases}$

We can reuse the heuristic from the single agent version using the objective function. Again, the heuristic has a heuristic entry for each agent, just like the costs. $\hat{f}((v_1, \dots, v_k)) := (\hat{f}(v_1), \dots, \hat{f}(v_k))$.

2.2.2 Problems with the joint state space

Using A* on the joint state space works well when assuming wait-at-target. When assuming disappear-at-target, however, there is an obvious problem: Agents don't disappear from the state. Consider the problem in Figure 2.3a. There are two agents on a graph that that can be represented as a 2×4 grid with one vertex being unavailable. This creates a single point of failure – which also happens to be the destination of agent 2. But since we are assuming disappear-at-target, this should not be a problem, as agent 2 will simply disappear once it reaches its destination and not cause any further conflicts.

Well, the joint state sees it as a problem though. It will find the "optimal" solution to let agent 2 wait for at least 2 timesteps (or move up and then down again), then start moving agent 2 towards its destination. This solution would be optimal when using the makespan objective function with a cost model that makes agent 2's wait actions



(a) First example (b) Second example

Figure 2.3: Two simple MAPF problems with 2 robots

at least as cheap as agent 1's move actions. But it is certainly not optimal when using the sum-of-cost objective function.

Theorem 2.13. *A* with the joint state space from Definition 2.12 does not guarantee optimal results when assuming disappear-at-target and using the sum-of-cost objective function.*

Proof. Let the input graph be the one from Figure 2.3a, where each action has unit costs of 1.

When ignoring any conflicts, the minimal cost for agent 1 is 5 and the minimal cost for agent 2 is 2. Assuming disappear-at-target, no additional costs are imposed on agents once they reached their destination and disappear there. Since there are no conflicts between the shortest paths and we are using the sum-of-cost objective function, the minimal cost for the joint plan is thus $5 \oplus 2 = 7$.

In order for agent 1 to reach its destination, it needs to pass through the destination of agent 2. Since agent 2 must have a location in each state of A*, it needs to not be at its destination at the time where agent 1 passes through its destination. The earliest timestep where agent 1 can be at the destination of agent 2 is 3. Thus, agent 2 can only reach and stay at its destination after timestep 3. Its minimal cost as found by A* with the joint state space is thus 4. Adding that to the minimal cost of agent 1, we reach $5 \oplus 4 = 9$. The minimal solution that A* using the joint state space found is thus not optimal. □

But the makespan objective function also does not help us in all cases. While the counter-example described above does work with makespan, one simple modification changes this: If we assign different costs to different actions, A* fails to find an optimal solution with makespan also.

Theorem 2.14. *A* with the joint state space from Definition 2.12 does not guarantee optimal results when assuming disappear-at-target and using the makespan objective function.*

Proof. Let the input graph be the one from Figure 2.3a. All wait actions have cost 2. The action to move from the start location of agent 2 to any vertex next to it has costs 3. All other move actions have cost 1.

This brings the minimal cost to $5 \oplus 4 = 5$. However, as before, A* cannot move agent 2 to its destination at or before timestep 3. There are several options to achieve this: First option would be to let the agent wait for two timesteps at its start before moving. This would have a cost of 8. Second option would be to move it to one of its neighbors, back to its start, and then towards the destination. This has a cost of 10. The third and best option is to move the agent left and then follow agent 1, not returning to the starting location at all. This has costs of 6. So, the best solution that can be found by A* has costs of $5 \oplus 6 = 6$, thus not optimal. □

The joint state space also has problems when assuming stay-at-target. Consider the problem in Figure 2.3b together with a weight function that assigns higher costs to wait actions than to move actions. The optimal solution would be to move both agents straight to their destination, where agent 1 will wait for 2 timesteps at no cost.

But A* on the joint state space doesn't treat waiting with cost different than waiting with no cost, so it will assume that this solution has higher costs than it actually has. Over-estimating costs is known to make A* suboptimal, and this case is no different. The "optimal" solution A* on the joint state space finds will involve moving agent 1 for all three timesteps, e.g. by going up, right and then down. With the sum-of-cost objective function, this however will have higher costs than the optimal solution.

Theorem 2.15. *A* with the joint state space from Definition 2.12 does not guarantee optimal results when assuming stay-at-target and using the sum-of-cost objective function.*

Proof. Let the input graph be the one from Figure 2.3b, where each move action has cost 1 and each wait action has cost 2.

When ignoring any conflicts, the minimal cost for agent 1 is 1 and for agent 2 is 3. Assuming stay-at-target, no additional costs are imposed on agents once they reached their destination and stay there. Since there are no conflicts between the shortest paths and we are using the sum-of-cost objective function, the minimal cost for the joint plan is thus $1 \oplus 3 = 4$.

This optimal solution can be traced through the joint state space. Let v_1 through v_6 be the vertices in the lower row in Figure 2.3b from left to right. We will encounter the following states:

- (v_1, v_6) (initial state)
- (v_2, v_5) (edge weight: $(1, 1)$)
- (v_2, v_4) (edge weight: $(2, 1)$)
- (v_2, v_3) (edge weight: $(2, 1)$)

Adding up the edge weights of the optimal solution gives us $(5, 3)$, so the total cost was over-approximated to $5 \oplus 3 = 8$.

There is an alternate path through the joint state space:

- (v_1, v_6) (initial state)
- (v_2, v_5) (edge weight: $(1, 1)$)
- (v_1, v_4) (edge weight: $(1, 1)$)
- (v_2, v_3) (edge weight: $(1, 1)$)

Adding these edge weights up gives $(3, 3)$, and the total cost computes as $3 \oplus 3 = 6$, which in this case is the correct value. A* on the joint state space will thus prefer this solution over the optimal solution. □

We could, again, modify the cost model of the example in Figure 2.3b to make it a counter-example for makespan as well. In the above counter-example, the increased

costs for the 1st agent would be "hidden" if makespan was used, since they never exceeded those of the 2nd agent. If however the movement costs for agent 1 are always higher than those of agent 2, we would see the cost increase with the makespan objective function as well.

Theorem 2.16. *A* with the joint state space from Definition 2.12 does not guarantee optimal results when assuming stay-at-target and using the makespan objective function.*

Proof. Let the input graph be the one from Figure 2.3b. The move actions in the first two columns have cost 2, while the others have cost 1. Each wait action has cost 3. This makes the minimum cost $2 \oplus 3 = 3$, which A* over-approximates as $8 \oplus 3 = 8$. The alternate path described in the proof of Theorem 2.15, combined with the changed cost model, has costs of $6 \oplus 3 = 6$, which are correctly computed (and thus not over-approximated) by A*. A* on the joint state space will thus prefer this solution over the optimal solution. □

Luckily for us, the problems we introduce later will be assuming wait-at-target and use the makespan objective function. While certainly possible, we therefore do not need to develop an improved version of A* on the joint state space that behaves correctly when assuming stay- or disappear-at-target. We will, however, discuss an improved version regarding its performance: *Operator Decomposition*.

2.2.3 Operator Decomposition

Operator Decomposition is an approach proposed in [29] that reduces the degree, i.e. the number of neighboring states for each state, in A*. The idea is to only assign an action to one agent per step in the search space. Unfortunately, [29] does not give us a formal definition for A* with Operator Decomposition (A*/OD), so we develop our own based on the work of [29].

In order for this to work, we need to record the current state of the algorithm. For this, we replace the state, which previously was just the configuration, with a tuple containing

- The current configuration $(v_1, \dots, v_k)$ that has the current locations of all agents. For the agents that haven't moved yet, these will be equivalent to those locations of the parent node.
- A non-empty set of agents $A \subseteq \{1, \dots, k\}$ that have not moved yet.
- A set of edges $\varrho \subseteq E$ that were traversed by any agent.

We classify these states based on the agents that need to move. If all agents have to move, i.e. $A = \{1, \dots, k\}$, the state is called a *standard* state. Otherwise it is called an *intermediate* state. You can think of intermediate states as a partial time step, and of standard states as a completed time step. A standard state, where all agents have yet to move, implies that no edges were traversed yet, so ϱ will be empty.

Initial and target states need to be standard states. The initial state thus becomes $((s(1), \dots, s(k)), \{1, \dots, k\}, \emptyset)$, and the target state is $((d(1), \dots, d(k)), \{1, \dots, k\}, \emptyset)$.

In each step, we choose an agent from A to move next. This choice needs to be based on some arbitrary but consistent ordering. Since we use the numbers 1 through k for the agents, we will use the order on natural numbers. Therefore, we consider the agent $i = \min A$ in each step. That agent can then either do a wait action, in which case we don't need to modify the position, or a move action, in which case we need to update the position and the list of traversed edges. Of course an action is only allowed if it does not cause a conflict between another agent that already moved. We explicitly do not check agents that are yet to move for occupancy conflicts. This is because if we now rejected a move for an alleged vertex conflict, but then moved the agent later before the next standard state, we rejected a valid move. Lastly, we need to remove the agent from the list of agents that have not moved yet.

Definition 2.17. *The **operator decomposition state space** for MAPF on a graph $G = (V, E)$ with k agents is a directed, weighted graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ with weight function $w_\mathcal{G}$ where*

- $V_\mathcal{G} := V^k \times (2^{\{1,\dots,k\}} \setminus \emptyset) \times 2^E$
- $(((u_1, \dots, u_k), A, \varrho), ((v_1, \dots, v_k), A', \varrho')) \in E_\mathcal{G}$ *iff*
  1. *If $A = \{i\}$, then $A' = \{1, \dots, k\}$ and $\varrho' = \emptyset$*
  2. *If $|A| \neq 1$, then $A = A' \uplus \{i\}$ for some $i \in \{1, \dots, k\} \setminus A'$*
  3. *$u_{\ddot{\imath}} = v_{\ddot{\imath}}$ for all $\ddot{\imath} \in \{1, \dots, k\} \setminus \{i\}$, i.e. only one agent moved*
  4. *$v_{\ddot{\imath}} \neq v_i$ for all $\ddot{\imath} \in \{1, \dots, k\} \setminus A$, i.e. no vertex conflict*
  5. *If $u_i \neq v_i$, then $(v_i, u_i) \notin \varrho$, i.e. no edge conflict*
  6. *If $u_i \neq v_i$ and $|A| \neq 1$, then $\varrho' = \varrho \cup \{(u_i, v_i)\}$*
  7. *If $u_i = v_i$ and $|A| \neq 1$, then $\varrho' = \varrho$*
- $w_\mathcal{G}(((u_1, \dots, u_k), A, \varrho), ((v_1, \dots, v_k), A', \varrho')) := (w_1, \dots, w_k)$ *where w_i is defined as in Definition 2.12*

The heuristic for Operator Decomposition A* (A*/OD) works exactly the same as the one for the joint state space.

When we assume that MAPF operates in a warehouse with a grid layout, i.e. each intersection has at most 4 edges, there are at most 5 actions an agent can take: 4 move actions and one wait action. The simple approach using the exponential state space had a branching factor of up to 5^k and a depth of m, where m is the maximum time of the solution. Using Operator Decomposition, we have a branching factor of 5, but the depth only increases to $k \cdot m$.

2.3 Solve MAPF using CBS

Another algorithm for solving MAPF is *Conflict-Based Search* (CBS), which was specifically created for MAPF. CBS, first introduced in [23], relies on a low-level algorithm to do the single-agent path finding. It then uses some high-level that guides the low-level algorithm until a conflict-free solution is found. It starts off by planning all agents individually and identifying the conflicts between the single agent paths. One of these conflicts is then used to produce constraints for the low level to avoid that conflict in the next iteration.

A conflict always involves two agents, i and $\ddot{\imath}$, and occurs at a certain timestep t. We will resolve the conflict by producing two constraints: One constraint that "prioritises" the first agent, and one that "prioritises" the second agent. A *vertex constraint* $\langle i, v, t\rangle$ restricts agent i from occupying vertex v at time t. To resolve a vertex conflict at vertex v, we thus produce the constraints $\langle i, v, t\rangle$ and $\langle \ddot{\imath}, v, t\rangle$. An *edge constraint* $\langle i, u, v, t\rangle$ similarly restricts agent i from traversing the edge (u, v) in the direction from u to v at time t. So, to resolve an edge conflict involving edge (u, v), where i is the agent travelling from u to v, we produce the constraints $\langle i, u, v, t\rangle$ and $\langle \ddot{\imath}, v, u, t\rangle$.

Definition 2.18. *A **vertex constraint** is a tuple $\langle i, v, t\rangle$ that restricts agent i from occupying vertex v at time t.*

Definition 2.19. *An **edge constraint** is a tuple $\langle i, u, v, t\rangle$ that restricts agent i from traversing the edge (u, v) at time t in the direction from u to v.*

These constraints are used to build a *Conflict Tree* (CT). Each node contains

1. A set of constraints. The root node has an empty set. Each child node adds one (or more[3]) constraints.
2. The single-agent plans. These plans must be consistent with the set of constraints. They are found using the low-level algorithm.
3. The cost of the node. This is equivalent to the cost of the joint plan for the single-agents, which depends on the objective function. Note that the joint plan does not need to be valid.

Similarly to A*, CBS maintains an open list. The open list initially contains the root node, which has no constraints. In each iteration, CBS chooses a node with minimal cost from the open list and *expands* that node. First, it determines the conflicts between the single agent paths of the node. If there are none, we return the paths as an optimal solution to the MAPF problem we are solving. Otherwise, we choose a conflict and generate constraints as described above. For each (of the two[4]) constraints,

[3] While standard CBS as introduced in [23] always adds exactly one constraint, there are variations of CBS that introduce multiple constraints [14].

[4] While we always split using two constraints, CBS can produce k constraints if the same conflict occurs between k agents [23].

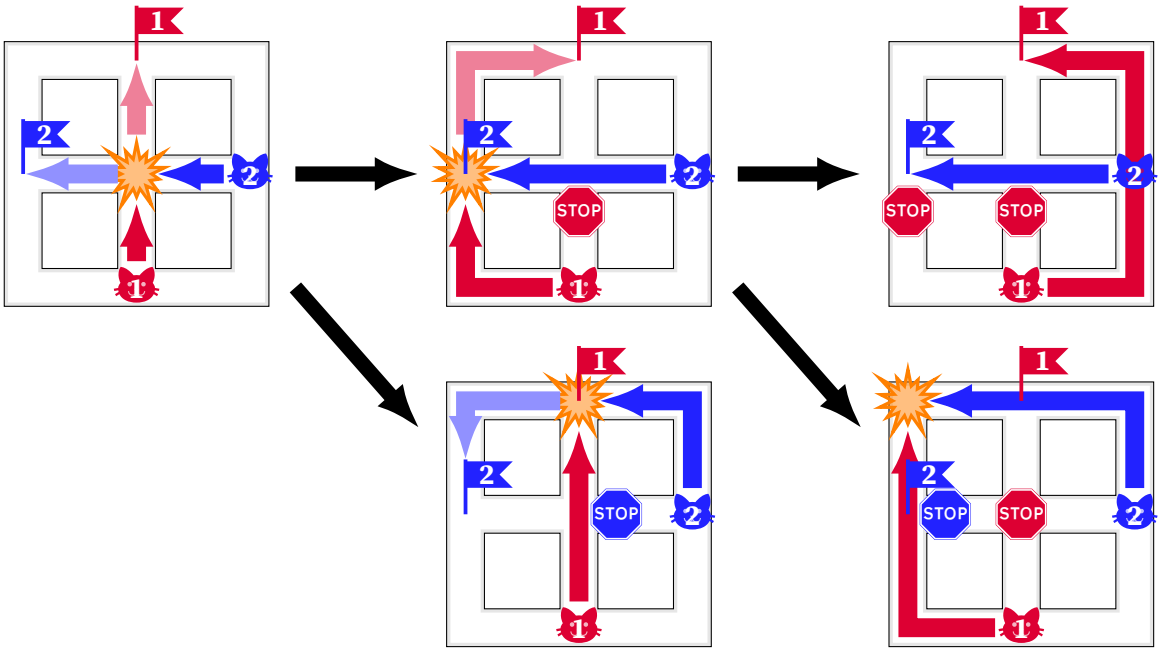


Figure 2.4: An example demonstrating how CBS works

we ask the low-level algorithm to produce new single agent paths. If the low-level algorithm succeeds in planning all agents, we produce a new node and add it to the open list.

An example for CBS can be seen in Figure 2.4. We operate on a 3×3 grid with two agents, and assume stay-at-target and that movement costs the same on each edge and is always cheaper than waiting. The root node of the CT is displayed on the left. Both agents have planned their respective shortest paths, leading to a vertex conflict at the center vertex at time $t = 1$. The CBS algorithm expands this node, generating a constraint where agent 1 must not occupy that vertex at time $t = 1$, and another one for agent 2 with the same restriction. For both of these generated constraints, a CT node is created. These can be seen in the center of Figure 2.4. The constraint is indicated with a STOP-sign in the agent's colour. Since waiting is more expensive than taking a detour, the low-level planned a detour for the restricted agent in each node.

While we could have been lucky and obtained conflict-free plans from either constraint, this is not guaranteed. In the case depicted in Figure 2.4, we were unlucky and got two new CT nodes with conflicts. We pick the first one that has a vertex conflict at the destination of agent 2 at time $t = 2$ and expand it. This generates two constraints, one for either agent, blocking the destination of agent 2 at time $t = 2$. Again, we create two CT nodes, which can be seen on the right hand side. And this time, the low level for the first node found plans for both agents that are conflict-free. Thus, CBS has found a solution. The remaining CT nodes are not expanded.

Notice that, if there is no solution to MAPF, there is no guarantee for CBS to terminate. In some cases, there might be a point where the low-level algorithm always fails to find paths, which in turn will deplete the open list of CBS and force it to ter-

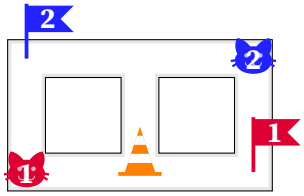


Figure 2.5: A MAPF Problem with no solution, where CBS does not terminate

minate. However, in other cases like the one depicted in Figure 2.5, the low-level always finds a path for each agent. These will however never be conflict free, so CBS will keep generating new constraints forever, and never terminate.

2.3.1 Space-Time A*

For the low level algorithm, we need a search algorithm that can honor the constraints placed by the CBS algorithm. With basic A*, the fact that our constraints are time-based becomes a problem, as the algorithm does not keep track of time by itself. Therefore, we use Space-Time A* [28, 15]. The search space for Space-Time A* are vertex-timestep pairs. If v is the start of the single agent, we start Space-Time A* at $(v, 0)$.

Starting with the initial case where we have no constraints, we say that a neighbor to (u, t) is a pair $(v, t+1)$ where v is a neighbor to u. Now, for each constraint, we remove neighbors that would violate the constraint: For any vertex constraint $\langle a, v, t\rangle$ where a is the agent we are planning for, we disallow any vertex-timestep pair (v, t), i.e. remove (v, t) as a neighbor from any other pair so that Space-Time A* can never visit the restricted vertex v at time t. Similarly, for any edge constraint $\langle a, u, v, t\rangle$ where a is the agent we are planning for, we remove $(v, t+1)$ as a neighbor of (u, t).

Definition 2.20. *The **space-time state space** for a graph $G = (V, E)$ is a directed, weighted graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ with weight function $w_\mathcal{G}$ where*

- $(v, t) \in V_\mathcal{G}$ *iff*
  1. $v \in V, t \in \mathbb{N}_0$
  2. *there is no vertex constraint* $\langle a, v, t\rangle$
- $((u, t), (v, t+1)) \in E_\mathcal{G}$ *iff*
  1. $(u, v) \in E$
  2. *there is no edge constraint* $\langle a, u, v, t\rangle$
- $w_\mathcal{G}((u, t), (v, t+1)) := w((u, v))$

2.3.2 Optimality of CBS

Unlike A* on the joint state space, which worked best with the wait-at-target assumption, CBS has a problem with exactly this assumption. Given that the low level of CBS never "extends" the length of the single-agent plans, it never considers the costs of waiting at the agent's destination. Since the conflict checking will, unless we'd assume disappear-at-target, keep the agent at its destination, other literature simply extends the single-agent plan with wait actions [8]. This can, however, become a problem.

Consider the example in Figure 2.3b again. As before, we assume that waiting costs more than moving. Now, a single-agent plan for agent 1 will move it one timestep and reach its destination, resulting in a single-agent plan of length 1. At the same time, the single-agent plan for agent 2 will be of length 3. If we simply extended agent 1's plan with two wait actions, we would increase its cost more than if we extended it with two move actions. Therefore, CBS found a suboptimal solution.

Theorem 2.21. *CBS does not guarantee optimal results when assuming wait-at-target.*

Proof. Let the input graph be the one from Figure 2.3b, where each move action has cost 1 and each wait action has cost 2.

The optimal solution has cost $3 \oplus 3$: Agent 1 will move right, then left, and then right again, counting 3 move actions at cost 1, and agent 2 will move straight from its start location to its destination, which also costs 3 move actions. If using sum-of-costs, this is equal to 6, whereas makespan computes the joint costs as 3.

CBS will compute the optimal solution for both agents as direct move to their respective destinations. This has a plan of length 1 for agent 1 and of length 3 for agent 2. To construct a joint plan, we must therefore fill the plan for agent 1 with 2 move actions at cost 2, which brings the cost of the joint plan to $5 \oplus 3$. Regardless of the objective function, this is suboptimal: Sum-of-costs computes this as 8 while makespan says this is 5. □

Timing Conflicts

We can combat this by introducing another type of conflict: *Timing conflicts*. A timing conflict at time t means that some agent's single-agent plan finished at time t, while another agent's plan finished at another timestep. To resolve these, we introduce *timing constraints*: The timing constraint $\langle \leqslant t \rangle$ means that all agent's single-agent plans must finish at or before timestep t, while the timing constraint $\langle > t \rangle$ means that all agent's single-agent plans must finish after timestep t.

Definition 2.22. *Let π and π' be two single-agent plans. If $|\pi| \neq |\pi'|$, there is a **timing conflict** between the two plans.*

Definition 2.23. *A **timing constraint** is a pair $\langle \otimes t \rangle$ where $\otimes \in \{\leqslant, >\}$ which restricts the length of each agent i's plan π_i to $|\pi_i| \otimes t$.*

We use timing conflicts and constraints every time we use CBS with the wait-at-target assumption to guarantee optimality.

Theorem 2.24. *CBS, when using timing conflicts, does guarantee optimal results when assuming wait-at-target.*

Proof. CBS is known to produce optimal results when assuming stay-at-target. More importantly, CBS always computes costs for its node as a lower bound to the actual costs, and correctly computes them for the solution when assuming stay-at-target. This was shown in [23].

Since the solution of CBS is conflict free, the introduction of timing conflicts ensures that all single-agent plans in the solution of CBS have the same length. Therefore, there is no longer an observable difference between the cost computation of wait-at-target and that of stay-at-target. Thus, CBS returned an optimal solution and correctly computed the costs of its solution. □

Mutually Disjunctive Constraints

In subsequent chapters, we will introduce new constraints for CBS. It is therefore important to know which properties of a constraint are important to guarantee optimal results from CBS. We use a property called *mutually disjunctive* [14]. The idea behind this is to ensure that any valid (but not necessarily optimal) solution is not prohibited by both constraints.

Definition 2.25. *Two constraints are mutually disjunctive iff any pair of conflict-free single-agent plans for the two agents involved in the constraints satisfy at least one of the constraints.*

Using only mutually disjunctive constraints leads to CBS giving us an optimal solution. We will not prove this here, but refer to [14] instead. We will, however, show that the two types of constraints generated by CBS, namely vertex and edge constraints, as well as the timing constraints we introduced earlier, are mutually disjunctive.

Lemma 2.26. *The vertex constraints $\langle i, v, t\rangle$ and $\langle \ddot{\imath}, v, t\rangle$ for two different agents $i \neq \ddot{\imath}$ are mutually disjunctive.*

Proof. Assume this wasn't true. That means there are conflict-free plans violating both constraints. Let there be two single-agent plans π and π' for agents i and $\ddot{\imath}$ respectively that violate both constraints. That means that $\pi[t] = v$ so that π violates the first constraint, and $\pi'[t] = v$ so that π' violates the second constraint. Then, there is a vertex conflict between the two plans. Which means the plans weren't conflict-free, contradicting the assumption. □

Lemma 2.27. *The edge constraints $\langle i, v, w, t\rangle$ and $\langle \ddot{\imath}, w, v, t\rangle$ for two different agents $i \neq \ddot{\imath}$ are mutually disjunctive.*

Proof. Again, assume this wouldn't be true. Let there be two conflict-free single-agent plans π and π' for agents i and $\ddot{\imath}$ respectively that violate both constraints. That means that $\pi[t] = v$, $\pi[t+1] = w$, $\pi'[t] = w$ and $\pi'[t+1] = v$. Then, there is an edge conflict between the two plans. Again, this contradicts the assumption. □

Lemma 2.28. *The timing constraints $\langle \leqslant t \rangle$ and $\langle > t \rangle$ are mutually disjunctive.*

Proof. If this wasn't true, there would be two conflict-free single-agent plans π and π' that violate both constraints. Then, $|\pi| > t$ so that it violates the first constraint, and $|\pi'| \leqslant t$ so that it violates the second constraint. This means there is a timing conflict between the two conflict-free plans, which contradicts the assumption. □

CHAPTER 3

SpinBus: A Quantum Architecture with Reversible Conveyors

The SpinBus architecture, introduced in [10], is an experimental quantum architecture which uses qubit shuttling conveyors in an attempt to build a quantum computer that eventually scales to hundreds of qubits. These conveyors can shuttle multiple qubits at the same time while being controlled by just one control signal, which we call a *wire*. Each conveyor can be idled, moved forwards or moved in reverse by its control wire. That movement happens throughout the entire length of the conveyor, i.e. it is impossible to limit movement to a certain area. This global movement introduces challenges for the routing of qubits. Where in classical MAPF, all conflicts were local to one vertex or one edge, in this architecture, we encounter conflicts that are no longer local. We study this problem in this thesis and try to find ways to speed up the computation of routing of qubits. We will not go into details on how the physics works.

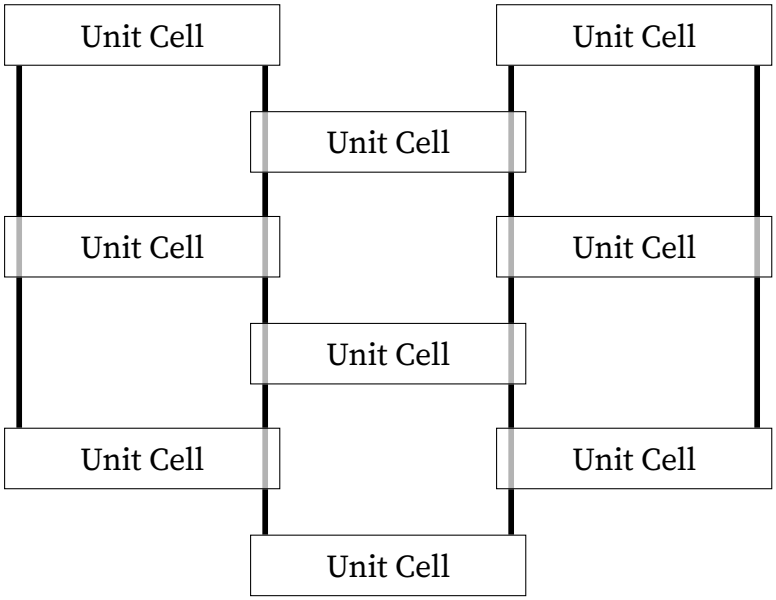


Figure 3.1: SpinBus: A macro view of the architecture

We can divide the architecture into two parts: Busses and Unit Cells. A *bus* is a long conveyor covering the entire height of the quantum computer. Each bus is controlled by a single wire. Unit cells are placed between two busses, connecting to one bus on their left and to another on their right hand side. They are arranged staggered, so there are only ever T-intersections on the busses (and also for the rest of the conveyors), but never 4-way intersections. Please refer to Figure 3.1 for a visualisation.

A *unit cell* is a short section inbetween two conveyors. It contains a *manipulation zone* that can execute single or two-qubit gates, and an *initialisation and readout zone* (*irz*) to initialise new qubits or read values of existing qubits. The conveyors inside the unit cell are controlled by two wires: The *split* wire and the *continuous* wire. We can visualise this as a graph (see Figure 3.2). The labeling is as follows: *lj*/*rj* refers to the left/right junction, which is also part of the busses. *lm*/*rm* are on the left/right hand side of the manipulation zone, which are the two positions where qubits need to be for gates to be executed. The *h* vertex marks the handover zone, where the control wire changes from the split to the continuous wire. Finally, the *irz* is connected to the center junction *cj*. In Figure 3.2, the connection from the unit cell to the busses is indicated with a dotted green wire, while for the unit cell, the split wire is coloured red (○——○) and the continuous wire is coloured blue (□——□) .

We expand the graph from Figure 3.2 with intermediate vertices to make each edge an edge of unit length. In our model, the length of an edge is discrete and maps linearly to the time it takes a qubit to travel on the edge. And MAPF requires that it takes unit time to traverse an edge.

To make it easier to visualise qubits and their movement, we introduce a slightly more abstract way to draw a unit cell. In Figure 3.3, you can see a 3×3 SpinBus device. There are 9 unit cells, arranged in a staggered way as described before. Those parts of the unit cell controlled by only the split wire are drawn with a red background, while those controlled only by the continuous wire are drawn with a blue background. The busses have a green background. Those edges controlled by more than one wire have two background colours corresponding to both wires.

Figure 3.3 is not correctly scaled by design. The part between the center and right junction, for instance, should be longer than the one connecting the center junction to the right manipulation zone as described in Figure 3.2. Instead, this way of drawing a SpinBus device is designed to show qubit routes in a concise way. You will encounter these again in section 3.2.

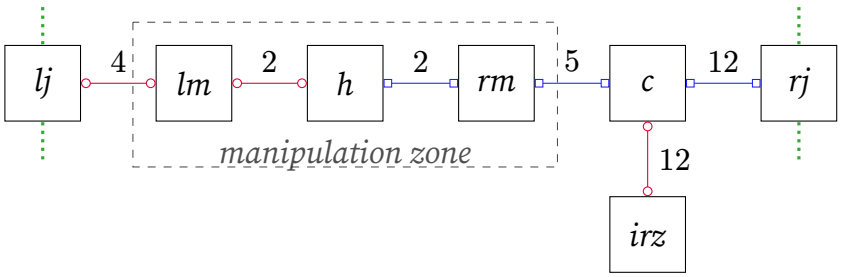


Figure 3.2: A unit cell as a graph

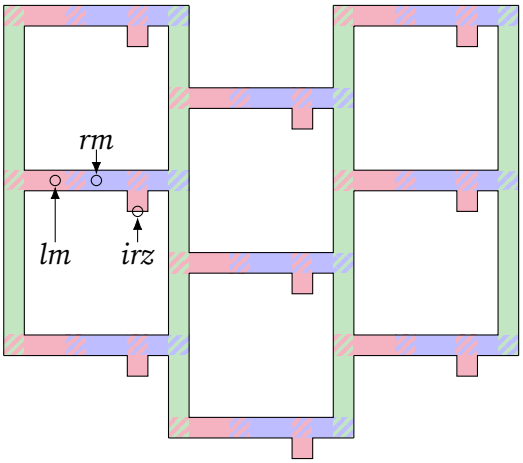


Figure 3.3: A SpinBus device with 3×3 unit cells

While we envision qubits as little dots in a graph sitting between edges, the reality is a little less simple. Qubits really like each other and want to talk whenever they get close enough. Unfortunately, this qubit cross-talk influences the states of the qubits, which are essential to the algorithm we run on the quantum computer. To combat the unwanted cross-talk, we introduce a *minimum distance* $\mathfrak{md} \geqslant 1$ between qubits. This minimum distance is a property of the quantum architecture and requires that there are at least $\mathfrak{md}$ edges between any two qubits at any time.[5]

But qubits don't only lose information when cross-talking with other qubits. Their information also degrades over time. This phenomenon is called *decoherence*. While we can decrease the loss of information, we can never stop it completely. This is why it is important to compile a quantum circuit in a way that makes sure it executes as fast as possible.[6]

Speaking of executing quantum circuits, this involves executing gates. The SpinBus architecture is designed in a way that during gate execution, the control wires of the unit cells (the red and blue wires in Figure 3.3) cannot be used for qubit shuttling. We simplify this requirement to say that no qubit should be moving while other qubits execute gates. Thus, we will split each quantum circuit into *layers*. Each layer has a start configuration, which records the qubit positions at the start of the layer, and a target configuration, i.e. the location of each qubit where it needs to be for the gates to be executed, and finally the gate instructions. In [8], a more generic graph structure for the architecture is defined that also contains information about possible gate executions. However, since we are focusing on the routing part, we will ignore the specific gates and just assume that we can execute each gate at each manipulation zone.

[5] This will be formalised into a conflict in Definition 3.2.

[6] What we really want is to minimise the loss of information, and reducing the time it takes to finish the circuit is a good heuristic for that.

3.1 Qubit Routing

Let us summarise what we learnt about the SpinBus architecture. First, qubits (which are the agents in the MAPF problem) may only move in the same direction when on the same conveyor. Second, qubits must maintain the minimum distance $\mathfrak{md}$ to reduce cross-talk. Third, qubits need to reach their destination as fast as possible to reduce decoherence, and we need to wait for all qubits to finish moving.

3.1.1 Q-MAPF

We are not the first to consider MAPF for Qubit Routing. It was already considered in [8], where MAPF was generalised into Q-MAPF. Q-MAPF comes with occupancy and direction conflicts, which generalise the vertex and edge conflicts from MAPF. Also, it assumes wait-at-target and the makespan objective function.

As an input to the Q-MAPF problem, we need the following:

1. The input we had for MAPF problems (Definition 2.10): An (undirected) graph $G = (V, E)$ with weight function w, and k agents with start locations s and destinations d.
2. The minimum distance $\mathfrak{md}$, which we already mentioned in the previous section.
3. Some information that maps edges in the graph to control wires of the quantum device. For this, we introduce the function $\mathfrak{A}_{\mathfrak{W}}$ that maps edges to the control wires alongside the direction of that control wire. We denote the set of all control wires as $\mathcal{V}_{\mathfrak{W}}$.

   $$\mathfrak{A}_{\mathfrak{W}} : E \to \left(2^{\{\text{Forward},\text{Idle},\text{Reverse}\}\times\mathcal{V}_{\mathfrak{W}}}\right)$$

   Note that eventhough the graph G is undirected, we treat the edges (u, v) and (v, u) as different for this function unless $u = v$.

   Each edge $e \in E$ must be feasible, so for each wire $\mathfrak{w} \in \mathcal{V}_{\mathfrak{W}}$, it must be assigned at most one direction, i.e. $|\{\mathfrak{d} \mid (\mathfrak{d}, \mathfrak{w}) \in \mathfrak{A}_{\mathfrak{W}}(e)\}| \leqslant 1$. Additionally we require that, for each pair of vertices $u \neq v \in V$ that are connected via an edge, there must be a control wire, and its direction has to be forward/reverse depending on the order of the vertices. Formally, there must exist a wire $\mathfrak{w} \in \{\mathfrak{w}' \mid (\mathfrak{d}, \mathfrak{w}') \in \mathfrak{A}_{\mathfrak{W}}((u, v)) \cup \mathfrak{A}_{\mathfrak{W}}((v, u))\}$ s.t.

   $$\{\mathfrak{d} \mid (\mathfrak{d}, \mathfrak{w}) \in \mathfrak{A}_{\mathfrak{W}}((u, v)) \cup \mathfrak{A}_{\mathfrak{W}}((v, u))\} = \{\text{Forward}, \text{Reverse}\}$$

   holds for this wire $\mathfrak{w}$.

With the input to Q-MAPF sorted, we can move on to define the conflicts.

An *occupancy conflict* occurs whenever two qubits violate the minimum distance. If two qubits i and i' are at vertices v and v' respectively, and the distance $\|v, v'\|$ between these two vertices is less than $\mathfrak{md}$, we have an occupancy conflict. Note that in case of a vertex conflict, $v = v'$, and thus $\|v, v'\| = 0$. Since $\mathfrak{md} \geqslant 1$, each vertex conflict is also an occupancy conflict.

Definition 3.1. *Let $v, v' \in V$. $\|v, v'\|$ denotes the minimum amount of edges that connect v to v'.*

Definition 3.2. *Let π and π' be two single-agent plans. If there exists $0 \leqslant t \leqslant \max\{|\pi|, |\pi|'\}$ s.t. $\|\pi[t], \pi'[t]\| < \mathfrak{md}$, there is an **occupancy conflict** between the two plans.*

Corollary 3.3. *Each vertex conflict is an occupancy conflict.*

A *direction conflict* occurs whenever two qubits want to assign different values to the same control wire. Let i and i' be two qubits that want to travel along edges e and e', respectively. If there exists a control wire $\mathfrak{w} \in \mathcal{V}_{\mathfrak{W}}$ s.t. $|\{\mathfrak{d} \mid (\mathfrak{d}, \mathfrak{w}) \in \mathfrak{A}_{\mathfrak{W}}(e) \cup \mathfrak{A}_{\mathfrak{W}}(e')\}| > 1$, i.e. there is more than one assignment to any control wire, we have a direction conflict. Note that in the case of an edge conflict, we travel different directions across the same wire, so we must get two different assignments (one $\mathrm{Forward}$ and one $\mathrm{Reverse}$).

Definition 3.4. *Let π and π' be two single-agent plans. If there exists $\mathfrak{w} \in \mathcal{V}_{\mathfrak{W}}$ and $0 \leqslant t < \max\{|\pi|, |\pi'|\}$ s.t. $|\{\mathfrak{d} \mid (\mathfrak{d}, \mathfrak{w}) \in \mathfrak{A}_{\mathfrak{W}}((\pi[t], \pi[t+1])) \cup \mathfrak{A}_{\mathfrak{W}}((\pi'[t], \pi'[t+1]))\}| > 1$, there is a **direction conflict** between the two plans.*

Corollary 3.5. *Each edge conflict is a direction conflict.*

Now we can formulate the Q-MAPF problem.

Definition 3.6. *Let $G = (V, E)$ be an undirected graph with weights $w : V \cup E \to \mathbb{N}$, control wires $\mathcal{V}_{\mathfrak{W}}$ with a function $\mathfrak{A}_{\mathfrak{W}} : E \to (2^{\{\mathrm{Forward}, \mathrm{Idle}, \mathrm{Reverse}\} \times \mathcal{V}_{\mathfrak{W}}})$, and $\mathbb{N} \oplus \mathbb{N} \to \mathbb{N}$ an objective function. Further, let there be k agents with start locations $s : \{1, \dots, k\} \to V$ and destinations $d : \{1, \dots, k\} \to V$. We assume wait-at-target. Then, **Q-MAPF** is the optimisation problem: Find a valid joint plan Π (w.r.t. occupancy and direction conflicts) s.t. there is no valid joint plan Π' with $w(\Pi') < w(\Pi)$.*

3.1.2 Qubit Routing using A*

Now that we have formalised the Q-MAPF problem, we need an algorithm to solve it. For that, we extend the Operator Decomposition A* (A*/OD) algorithm from subsection 2.2.3.

The vanilla OD (Definition 2.17) used $V^k \times 2^{\{1,\dots,k\}} \times 2^E$ as its state space. This already contains enough information to use it for Q-MAPF: Occupancy conflicts can be checked based on the known minimum distance $\mathfrak{md}$ and the current positions

stored in the state, and direction conflicts can be checked by looking up the control wires and their assignments for each edge using the known function $\mathfrak{A}_{\mathfrak{W}}$. However, we don't actually need to store the edges themselves in the state, so we are going to replace them with the control wires and their assignments directly.

With this change, we adopt the initial and target states by replacing the empty set with a function that is undefined for all wires. Thus, we define

$$\varrho_\perp : \mathcal{V}_{\mathfrak{W}} \dashrightarrow \{\text{Forward}, \text{Reverse}, \text{Idle}\}, \mathfrak{w} \mapsto \perp$$

Then, the initial state becomes $((s(1), \dots, s(k)), \{1, \dots, k\}, \varrho_\perp)$, and the target state becomes $((d(1), \dots, d(k)), \{1, \dots, k\}, \varrho_\perp)$. The heuristic remains unchanged.

Definition 3.7. *The **operator decomposition state space for Q-MAPF** on a graph $G = (V, E)$ with k agents, control wires $\mathcal{V}_{\mathfrak{W}}$, wire assignments $\mathfrak{A}_{\mathfrak{W}}$ and minimum distance $\mathfrak{md}$, is a directed, weighted graph $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$ with weight function $w_{\mathcal{G}}$ where*

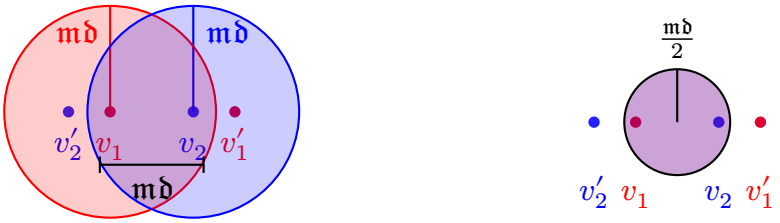
- $V_{\mathcal{G}} = V^k \times (2^{\{1,\dots,k\}} \setminus \emptyset) \times \{\varrho \mid \varrho : \mathcal{V}_{\mathfrak{W}} \dashrightarrow \{\text{Forward}, \text{Reverse}, \text{Idle}\}\}$
- $(((u_1, \dots, u_k), A, \varrho), ((v_1, \dots, v_k), A', \varrho')) \in E_{\mathcal{G}}$ *iff*
  1. *If $A = \{i\}$, then $A' = \{1, \dots, k\}$ and $\varrho'(\mathfrak{w}) = \perp$ for all $\mathfrak{w} \in \mathcal{V}_{\mathfrak{W}}$*
  2. *If $|A| \neq 1$, then $A = A' \uplus \{i\}$ for some $i \in \{1, \dots, k\} \setminus A'$*
  3. *$u_i = v_i$ for all $i \in \{1, \dots, k\} \setminus \{i\}$, i.e. only one agent moved*
  4. *$\|v_i, v_i\| \geqslant \mathfrak{md}$ for all $i \in \{1, \dots, k\} \setminus A$, i.e. no occupancy conflict*
  5. *For all $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}((u_i, v_i))$, $\varrho(\mathfrak{w}) = \mathfrak{d}$ or $\varrho(\mathfrak{w}) = \perp$, i.e. no direction conflict*
  6. *If $|A| \neq 1$, then $\varrho'(\mathfrak{w}) = \begin{cases} \mathfrak{d} & \text{if } (\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}((u_i, v_i)) \\ \varrho(\mathfrak{w}) & \text{otherwise} \end{cases}$*
- *The weight function $w_{\mathcal{G}}$ is equivalent to the one in Definition 2.17*

3.1.3 Qubit Routing using CBS

The authors of [8] define an extension of CBS for qubit routing called *QCBS*. QCBS solves Q-MAPF, so it replaces vertex with occupancy and edge with direction conflicts. However, QCBS goes further: It replaces the input graph, as defined in the MAPF and Q-MAPF problems, with device graphs. While these device graphs are generally better suited for an entire quantum compiler, for the purpose of qubit routing, we will stick with weighted undirected graphs.

We therefore create an extension for CBS without using device graphs which we call *CBS for Q-MAPF*. It takes the same inputs as the Q-MAPF problem (see Definition 3.6). Since we assume wait-at-target, we use timing conflicts alongside occupancy and direction conflicts. The constraint generation works as follows.

To resolve an occupancy conflict, we need to know the set of vertices that we have to block. This set should at least cover the location of one of the agents involved in



(a) Naïve approach (b) Valid approach

Figure 3.4: Visualisation of the minimum distance for occupancy constraints (inspired by [8, Figure 3.5])

the conflict, so that the constraint actually resolves the conflict. But we can do better: Remember that the constraints we want to generate will always pick one agent to constrain, hindering it to cause a problem with the other unconstrained agent. Since occupying any vertex within the minimum distance $\mathfrak{md}$ will cause an occupancy conflict, we try to include more of these vertices in the constraint. But at the same time, we need to ensure that the constraint does not prevent a conflict-free solution.

A naïve approach might be to restrict an agent from occupying the conflicting vertex alongside any vertex within a radius of $\mathfrak{md}$. This approach however leads to constraints that are not mutually disjunctive. Assume two agents are at two vertices, v_1 and v_2, and the distance between these two vertices is smaller than $\mathfrak{md}$, as shown in Figure 3.4a. Additionally, assume there are two more vertices, v_1' and v_2', whose distance between each other is greater than $\mathfrak{md}$, but where v_1' is within $\mathfrak{md}$ of v_2 and vice versa. The first constraint for the agent at v_1 would then restrict it from occupying v_1', and the second constraint would block the other agent from occupying v_2', therefore potentially prohibiting a conflict-free solution.

To obtain a valid constraint, we first determine the center between the two vertices v_1 and v_2 involved in the conflict. Since $\|v_1, v_2\| < \mathfrak{md}$, there must be a path from v_1 to v_2 that is shorter than $\mathfrak{md}$. We choose one with minimal length[7]. If its length[8] is even, there is a vertex in its center. Otherwise, the center is in the middle of an edge between two vertices u and w on the path. In this case, we will call the edge (u, w) the center. The occupancy constraint then restricts an agent to occupy any vertex within (but not on the boundary of) the circle around the center with radius $\frac{\mathfrak{md}}{2}$ as shown in Figure 3.4b.

We can now define occupancy constraints. They are defined in [8] in a generic way that allows for different minimum distances, which we have no use for. Hence, we skip this parameter and always assume the global minimum distance $\mathfrak{md}$.

Definition 3.8. *An **occupancy constraint** is a tuple $\langle i, c, t\rangle$ that restricts agent i from occupying a vertex $v \in V$ at time t if its distance to the center $\|v, c\|$ is less than $\frac{\mathfrak{md}}{2}$. c is either a vertex or an edge. If c is an edge, we compute $\|v, (u, w)\| = \min\{\|v, v'\| \mid v' \in \{u, w\}\} + 0.5$.*

[7] If this path is not unique, there could be different constraints based on which path we choose. However, each of these constraints would resolve the occupancy conflict between v_1 and v_2.

[8] Remember from Definition 3.1 that this is the number of edges.

Now we can move on to direction conflicts. These occur when two agents "compete" for a wire. To resolve this conflict, we forbid one of the conflicting wire assignment. A *direction constraint* $\langle i, \mathfrak{w}, \mathfrak{d}, t\rangle$ restricts the agent i from assigning direction $\mathfrak{d}$ to wire $\mathfrak{w}$ at time t. This means that the agent cannot traverse any edges that would require this assignment. Similarly to the definition for edge conflicts (Definition 2.5), the time t refers to the start of the movement. To resolve a direction conflict with agents i and $\ddot{\imath}$ and directions $\mathfrak{d}$ and $\mathfrak{d}'$ respectively, we generate the direction constraints $\langle i, \mathfrak{w}, \mathfrak{d}, t\rangle$ and $\langle \ddot{\imath}, \mathfrak{w}, \mathfrak{d}', t\rangle$.

Definition 3.9. *A **direction constraint** is a tuple $\langle i, \mathfrak{w}, \mathfrak{d}, t\rangle$ that restricts agent i from assigning direction $\mathfrak{d} \in \{\text{Forward}, \text{Idle}, \text{Reverse}\}$ to wire $\mathfrak{w} \in \mathcal{V}_{\mathfrak{M}}$ at time t.*

The author of [8] argues that QCBS is optimal due to the mutual disjunctiveness of the constraints, even though without calling them mutually disjunctive. We therefore skip proving the mutual disjunctiveness of the generated occupancy and direction conflicts in this thesis.

Corollary 3.10. *Let $i \neq \ddot{\imath}$ be two agents, $\mathfrak{w} \in \mathcal{V}_{\mathfrak{M}}$ a control wire and $\mathfrak{d} \neq \mathfrak{d}'$ be two directions $\mathfrak{d}, \mathfrak{d}' \in \{\text{Forward}, \text{Reverse}, \text{Idle}\}$. Then,*

1. *The occupancy constraints $\langle i, c, t\rangle$ and $\langle \ddot{\imath}, c, t\rangle$ are mutually disjunctive.*
2. *The direction constraints $\langle i, \mathfrak{w}, \mathfrak{d}, t\rangle$ and $\langle \ddot{\imath}, \mathfrak{w}, \mathfrak{d}', t\rangle$ are mutually disjunctive.*

3.2 Quantum Circuits

To determine the performance of the qubit routing algorithms, we test it on real quantum circuits. Each quantum circuit is manually split into layers. A layer has a routing and an execution phase. The routing phase is the one that is important for us, as this is where the qubits move from their current positions to their respective destinations. This involves manually specifying the destinations for the qubits, which means that if we use a basic algorithm, there is a chance that there might be a "better" way to position the qubits to reduce the timesteps necessary to complete the entire circuit. We will come back to this later in chapter 6, but for now, we ignore the suboptimal positioning and focus on qubit routing with these fixed positions. We also ignore the execution phase itself, but of course the gates, especially the two-qubit gates, place hard requirements on the positions for the qubits: Two qubits executing a two-qubit gate have to be in the same manipulation zone in the same unit cell.

We also use a small number of qubits and accordingly, small SpinBus devices that match the number of qubits. While most real-world applications require many more qubits, it is not practical for performance observation and analysis to use that many qubits.

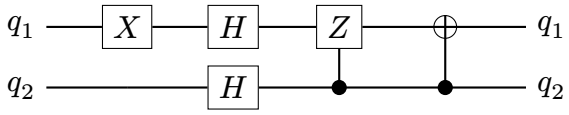


Figure 3.5: A very simple quantum circuit

3.2.1 Notation

A quantum circuit consists of two building blocks: Qubits and (quantum) gates. Qubits are displayed as horizontal lines while gates are displayed as blocks on these lines. Gates operate on one or more qubits, which in the context of the SpinBus architecture is limited to one or two qubits. A two-qubit gate is indicated by a gate on one qubit with a line connecting it to the second qubit.

The example in Figure 3.5 has two qubits and four layers. The first layer executes an X gate on the first qubits while the second qubit idles. The second layer executes an H gate on both qubits. In the third layer, a two-qubit gate called controlled Z gate is executed. The fourth layer shows another controlled gate, this time a $CNOT$ gate.

There are other ways to write down this very same circuit. For example, we could execute the H gate on the second qubit in the first layer, and idle it in the second layer instead. We could also move it to its own layer. In fact, the idea of layers is not necessary to write down a circuit. While we always write gates executed in one layer below each other, other literature might not use the concept of layers or display them differently.

3.2.2 Quantum Fourier Transform

One of the typical examples for a quantum algorithm is Shor's algorithm. Shor's algorithm, developed in 1994, can find prime factors of integers efficiently [24, 7]. Since the asymmetric encryption algorithms most commonly used today rely on the difficulty of prime factorisation, this quantum algorithm could disrupt the trust in common asymmetric encryption schemes widely used today if we had quantum computers with sufficiently many qubits [26, 21, 9].

While we won't go into much detail about Shor's algorithm in this thesis, we will take a look at the Discrete Fourier Transformation (DFT) used by Shor's algorithm. It can be implemented as a quantum circuit (which we call QFT) as such ([7]): Given n qubits $(q_1, \dots, q_n)$, for each $1 \leqslant i \leqslant n$ compute first $H\ q_i$ and then $R_{j-i+1}\ q_i, q_j$ for each $i < j \leqslant n$. The end result is in the wrong order, so usually you would add swap gates at the end, but our architecture can just physically move the qubits so we skip those swap gates.

We will use two versions of the QFT algorithm for the purpose of benchmarking our qubit routing algorithms. The first one, QFT_3, with $n = 3$ on a 2×2 SpinBus device, the circuit for which is shown in Figure 3.6. If we set $n = 4$, we get the QFT_4 algorithm, whose circuit is shown in Figure 3.7. We use a 3×3 SpinBus device for it.

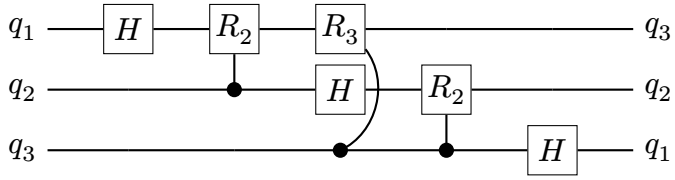


Figure 3.6: The quantum circuit for QFT_3

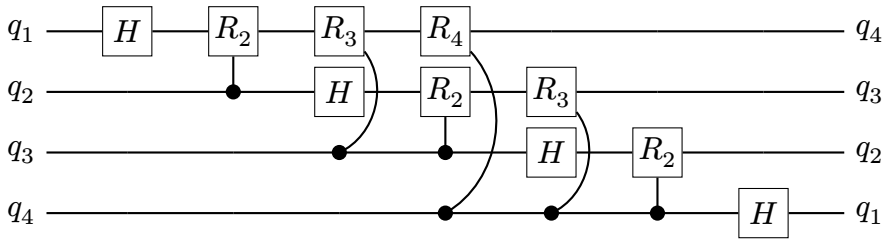
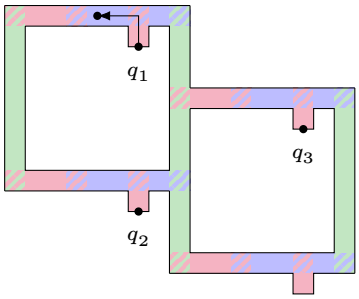
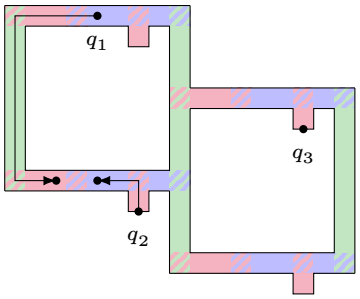


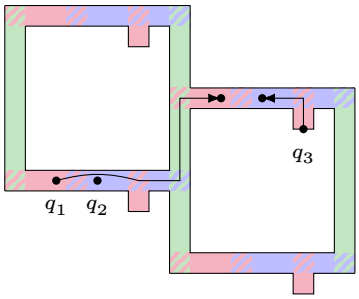
Figure 3.7: The quantum circuit for QFT_4



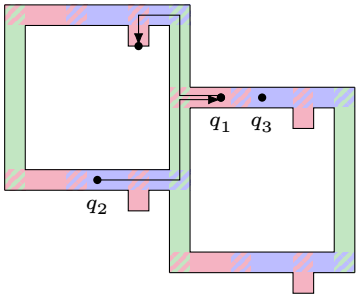
(a) 1st layer: Compute H q_1



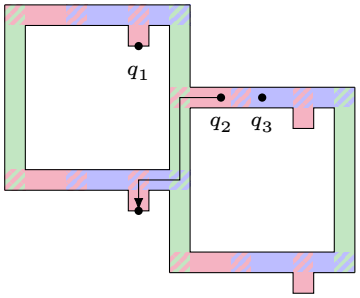
(b) 2nd layer: Compute R_2 q_1, q_2



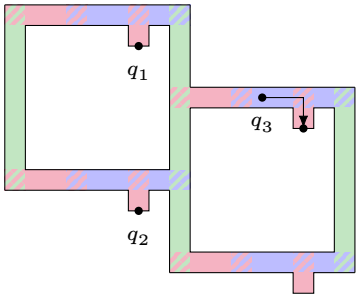
(c) 3rd layer: Compute R_3 q_1, q_3 and H q_2



(d) 4th layer: Compute R_2 q_2, q_3 and return q_1 to the IRZ



(e) 5th layer: Compute H q_3 and return q_2 to the IRZ

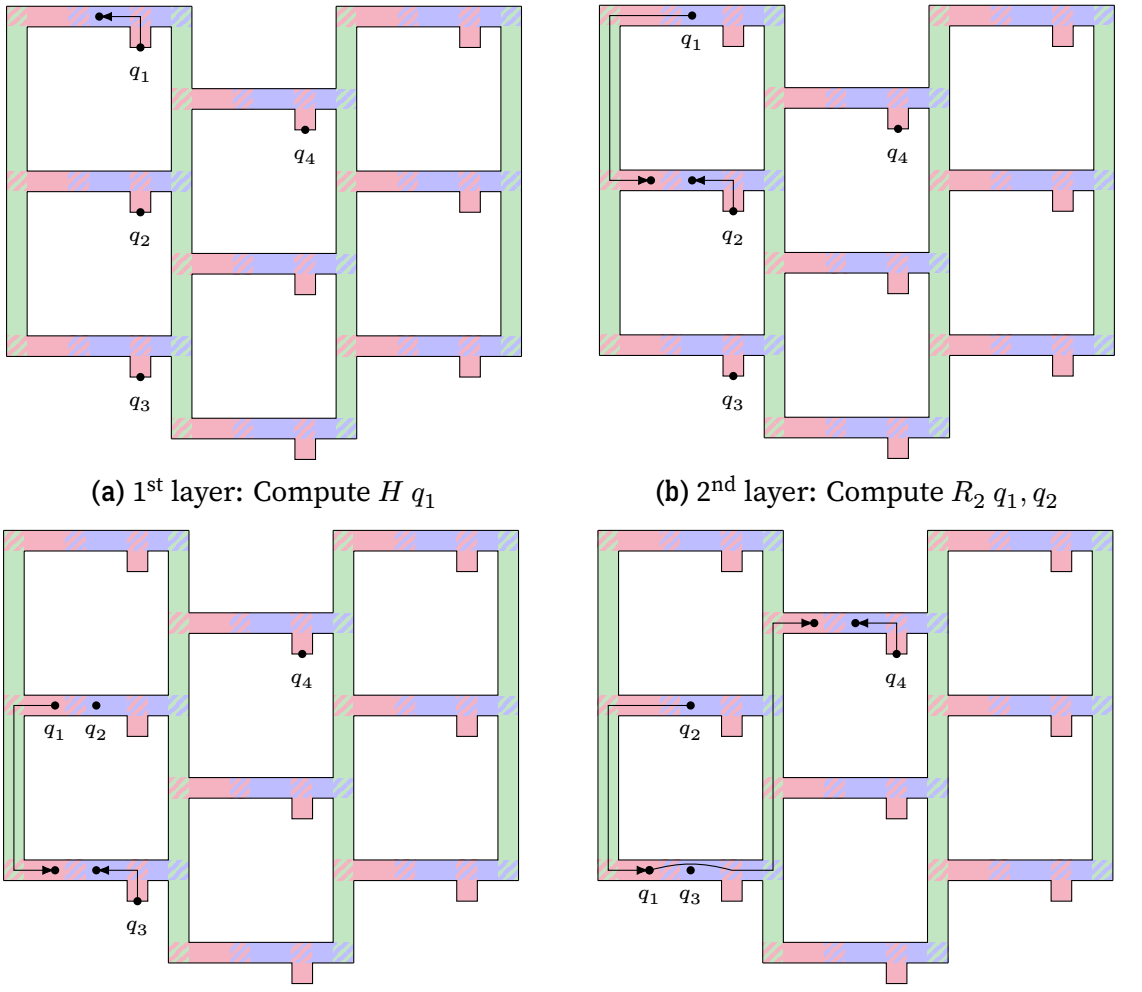


(f) 6th layer: Return q_3 to the IRZ

Figure 3.8: The layers of QFT_3

In order to compute these circuits on the Spinbus architecture, we need to define the locations for the qubits for each layer. Since optimal target assignment is not a focus for now, we use a simple algorithm: We distribute the qubits in numerical order into the *irz*'s of the spinbus device. Then, the first time each qubit is involved in a gate, it moves to the right manipulation zone in its unit cell. For the two-qubit gates, the qubit with the lower index moves to the left manipulation zone of the other qubit's unit cell. Once a qubit is no longer involved in any gates, it moves back to its original *irz* to be out of the way[9] of the other qubits. You can see all layers for QFT_3 in Figure 3.8, and for QFT_4 in figure 3.9.

[9] "Out of the way" refers to not blocking a manipulation zone. A qubit sitting in the *irz* of a unit cell might of course cause direction conflicts with other qubits moving through that unit cell.



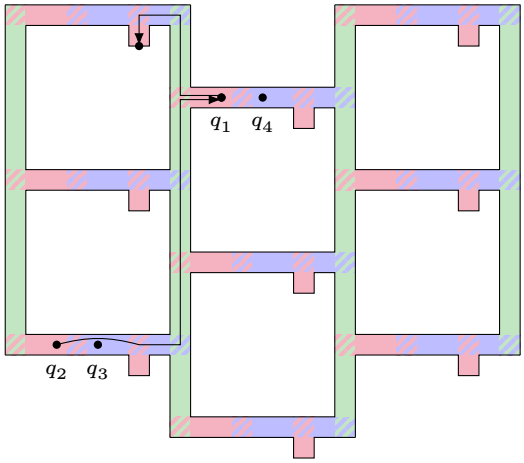
(a) 1st layer: Compute $H\ q_1$

(b) 2nd layer: Compute $R_2\ q_1, q_2$

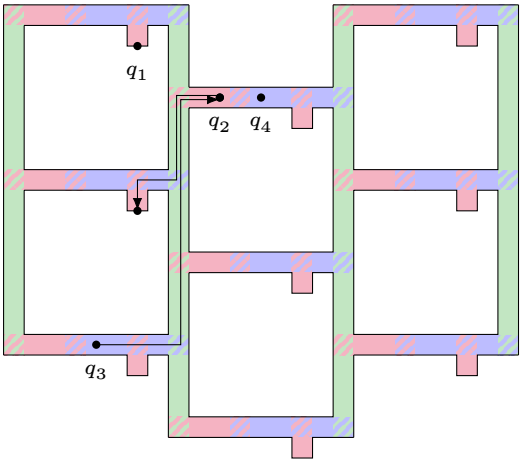
(c) 3rd layer: Compute $R_3\ q_1, q_3$ and $H\ q_2$

(d) 4th layer: Compute $R_4\ q_1, q_4$ and $R_2\ q_2, q_3$

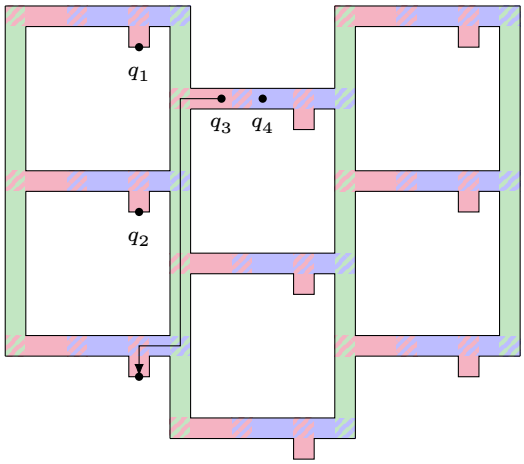
Figure 3.9: The layers of QFT_4



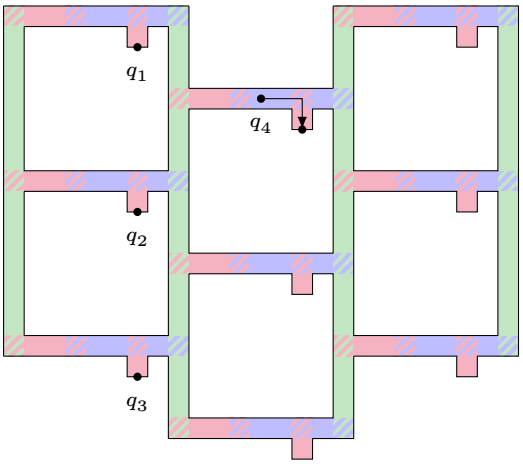
(e) 5th layer: Compute R_3 q_2, q_4 and H q_3 and return q_1 to the IRZ



(f) 6th layer: Compute R_2 q_3, q_4 and return q_2 to the IRZ



(g) 7th layer: Compute H q_4 and return q_3 to the IRZ



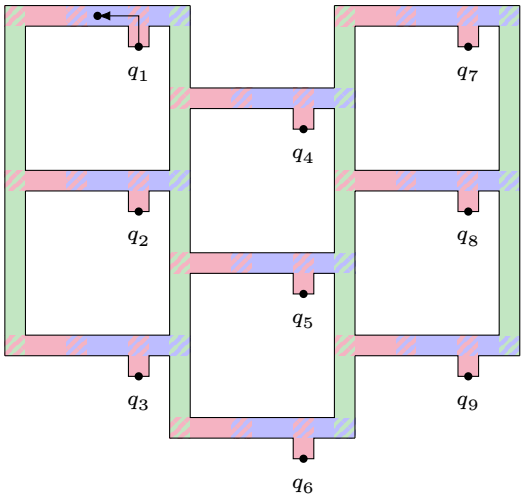
(h) 8th layer: Return q_4 to the IRZ

Figure 3.9: The layers of QFT_4

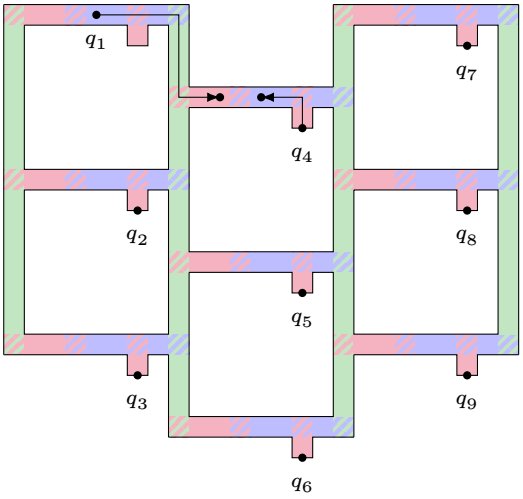
3.2.3 Shor Code: Encoding

Another quantum algorithm we take a look at is the Shor Code. It is a Quantum Error Correction Code introduced in 1995 by Shor [25]. It encodes one logical qubit using 9 physical qubits and can correct certain errors when decoding back to one qubit. The circuit for the encoding algorithm (SHORENC) can be seen in Figure 3.11.

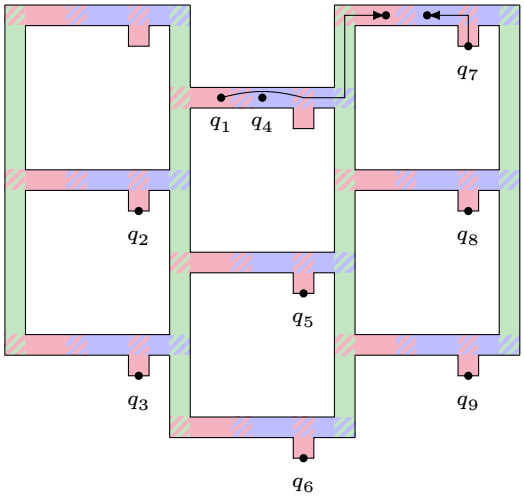
We employ the same strategy that we used for the QFT circuits to produce the layers for SHORENC. The result is shown in Figure 3.10.



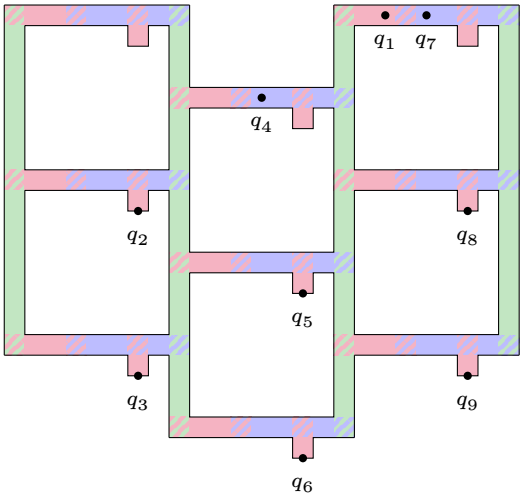
(a) 1st layer: Compute $X\ q_1$



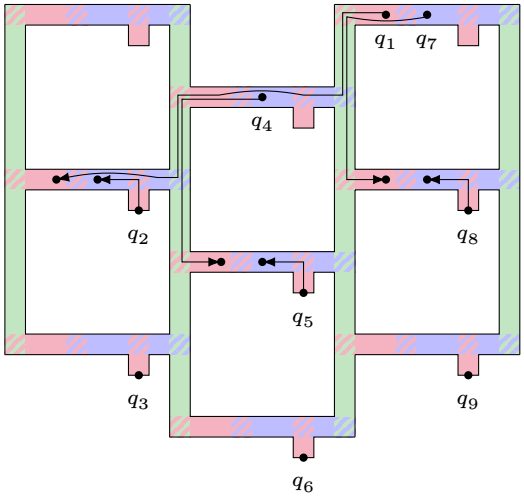
(b) 2nd layer: Compute $CNOT\ q_4, q_1$



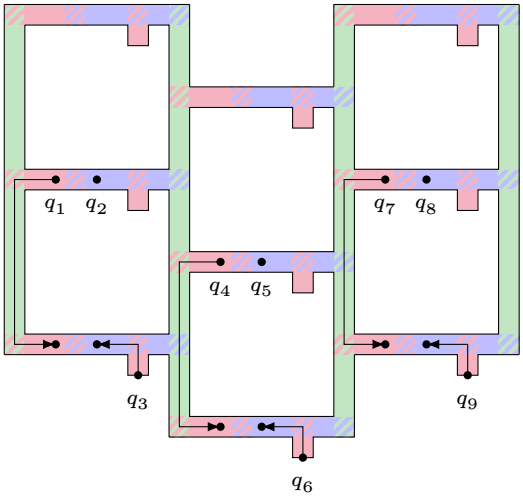
(c) 3rd layer: Compute $CNOT\ q_7, q_1$



(d) 4th layer: Compute $H\ q_1, q_4, q_7$



(e) 5th layer: Compute $CNOT\ q_2, q_1$, $CNOT\ q_5, q_4$ and $CNOT\ q_8, q_7$



(f) 6th layer: Compute $CNOT\ q_3, q_1$, $CNOT\ q_6, q_4$ and $CNOT\ q_9, q_7$

Figure 3.10: The layers of ShorEnc

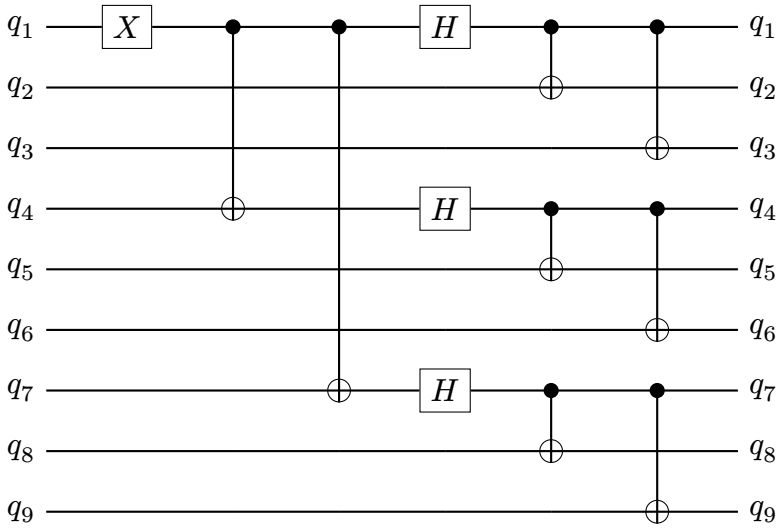


Figure 3.11: The quantum circuit for SHORENC (from [33])

3.3 Qubit Routing Performance

To analyse performance, we run both A* with Operator Decomposition (A*/OD) and CBS on a Framework 16 with an AMD Ryzen 7 7840HS running an up-to-date[10] ArchLinux. We limit the memory of each run to 38 GiB. Each benchmark is run 50 times, except for those that fail to produce a result, where we stop after the first failure. A run is considered a failure if it does not finish (DNF), which in all cases happened due to the memory limit.

The source code[11] that we use for this benchmark is written in Rust[12]. A basic proof-of-concept implementation for both A*/OD and CBS was provided by the author of [8]. We improved and optimised the implementations before using them for benchmarking.

We use a *scale* for the SpinBus device which works as follows: At a scale factor $f = 1$, we use the exact number of edges ℓ as in Figure 3.2. For other factors $f \neq 1$, we use the function in Equation 3.1. Remember from Figure 3.1 that the inner busses are exactly half as long as the outer busses. The scale function ensures that we can compute the inner busses as exactly half the length of the outer busses.

$$\text{scale}(\ell, f) = \begin{cases} \max\{\lfloor \ell \cdot f \rfloor, 1\} & \text{if } \ell \equiv \max\{\lfloor \ell \cdot f \rfloor, 1\} \pmod 2 \\ \max\{\lfloor \ell \cdot f \rfloor, 1\} + 1 & \text{otherwise} \end{cases} \tag{3.1}$$

The code that we use computes the minimum distance for a scaled version of SpinBus as $\text{scale}(\mathfrak{md}, f)$. However, since all problems set $\mathfrak{md} = 2$, the scale function will

[10] At the time of writing, Linux kernel 6.16.3 was current

[11] The source code lives at https://git.rwth-aachen.de/moves/quantum/shuttling-compiler. At the time of writing, access is only granted on request.

[12] https://www.rust-lang.org/

		A*/OD		CBS	
Circuit	Scale	Avg	SD	Avg	SD
QFT_3	0.1	0.010	0.0004	0.040	0.0078
	0.2	0.012	0.0005	0.054	0.0099
	0.3	0.023	0.0008	0.119	0.0138
	0.4	0.056	0.0011	0.191	0.0363
	0.5	0.093	0.0013	0.330	0.0513
	0.6	0.155	0.0023	0.497	0.0680
	0.7	0.166	0.0022	0.610	0.0867
	0.8	0.268	0.0031	0.979	0.1648
	0.9	0.287	0.0039	1.176	0.1685
	1.0	0.444	0.0069	1.558	0.1983
QFT_4	0.1	5.772	0.0309	0.888	0.3784
	0.2	44.649	0.2734	99.828	47.9865
	0.3	174.280	0.6339	DNF	
	0.4	DNF			
ShorEnc	0.1	DNF		DNF	

Table 3.12: Benchmark between A*/OD and CBS (in seconds)

not change the minimum distance. Therefore, for all problems and all scale factors, we know that $\mathfrak{mb} = 2$.

We run the qubit routing algorithms on QFT_3, QFT_4 and ShorEnc. The first one, QFT_3, tests the routing algorithms on an almost conflict free problem, so we can see how well the algorithm scales with the travel distance when we increase the scale factor of the SpinBus device. The other two, QFT_4 and ShorEnc, use a bigger SpinBus device and contain non-trivial conflicts that need to be resolved, with different amounts of agents.

You may find the result of the benchmark in Tables 3.12 and B.3. We show the average runtime (Avg) and the standard deviation (SD) in Table 3.12, and the minimum (Min) and maximum (Max) values in Table B.3. Additionally, we display the results in a plot in Figure 3.13. You can see a box per scale and algorithm that extends from the minimum to the maximum recorded time, with a dot marking the average time. In later plots such as Figure 5.3, where the minimum or maximum values are at least 1.5 times smaller/larger than the next value, we draw a line from the box to the outlier.

Sadly, QFT_3 is the only routing problem that either algorithm could solve at a scale factor of 1. Looking at the plot of the runtime of QFT_3 in Figure 3.13a, we see that A* with Operator Decomposition outperformed CBS. Its runtime was also more consistent. This is to be expected, as our CBS implementation uses hashmaps[13], which introduce non-determinism since their hash is seeded with a random value.

[13]Our code uses the hashmap implementation of hashbrown in version 0.15.1: https://docs.rs/hashbrown/0.15.1/hashbrown/struct.HashMap.html

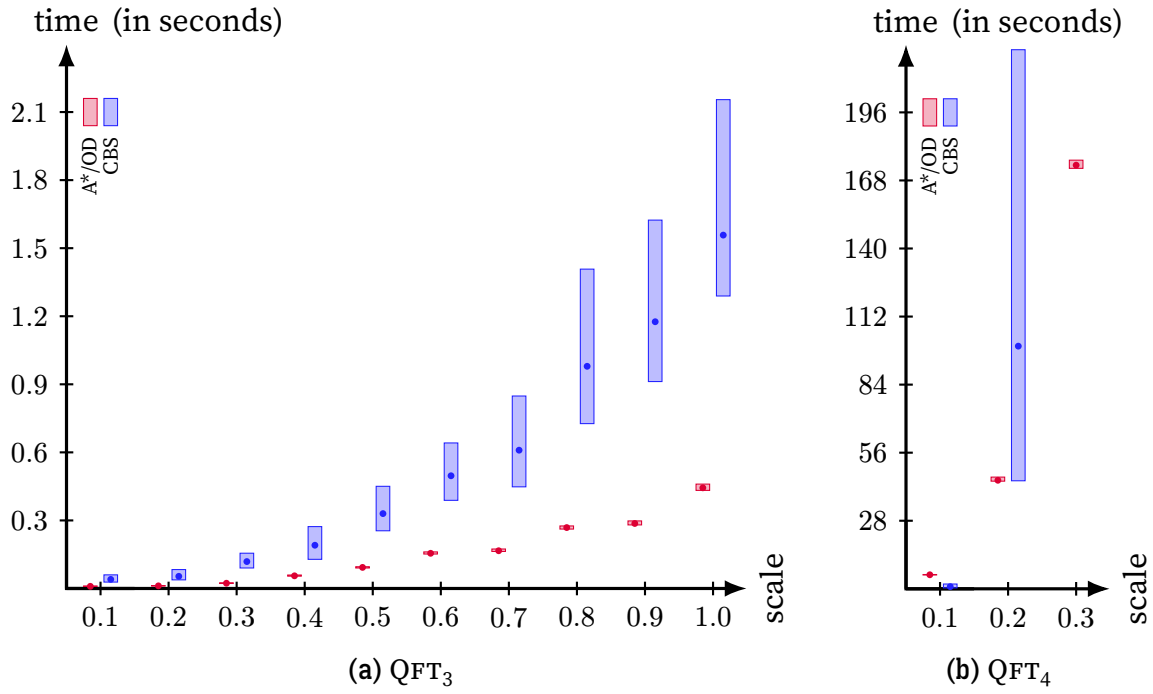


Figure 3.13: Benchmark between A*/OD and CBS (QFT_3 and QFT_4)

This behaviour continues in QFT_4. While CBS starts off strong at a scale factor of 0.1, A*/OD outperformes CBS for scale factors above 0.1. Also, the runtime of CBS at scale factor 0.2 was very inconsistent, which can be clearly seen in Figure 3.13b. For a scale factor of 0.3, CBS managed to finish 6 runs with the fastest done after 1310.230 and the longest after 2670.092 seconds. In the 7^{th} run, CBS ran out of memory, at which point we stopped the benchmark. Both algorithms ultimately failed at layer 6 (Figure 3.8f). Since both failed at the same layer, we also cannot switch algorithms based on which is better suited for the layer at hand.

The picture with ShorEnc is even more grim: Neither routing algorithm finishes at scale 0.1.[14] Compared to QFT_4, ShorEnc has more qubits to move, and thereby also more qubits that can conflict per layer. Clearly, neither algorithm scaled well with the increase in qubits compared to QFT_4. Interestingly, A*/OD already fails at layer 3 (Figure 3.10c) whereas CBS only fails at layer 5 (Figure 3.10e).

Overall, there is no winning algorithm so far. While A*/OD was faster and more consistent in its runtime, CBS actually solved layer 3 of ShorEnc which A*/OD was unable to. If we want to use either algorithm, we need to improve them first.

[14] There is a more optimal target assignment that allows CBS to finish routing for ShorEnc at small scales. We will discuss this later in chapter 6.

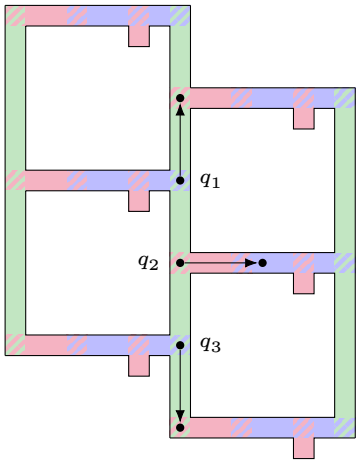


Figure 3.14: Three qubits with a direction conflict between each pair

3.4 Global Direction Constraints

After seeing these benchmark results, it is obvious that we need improvements. We start with a simple idea that could improve CBS: Apply direction constraints to all qubits, not just the one involved in the conflict.

Consider the example in Figure 3.14. At the first timestep, all three qubits are on the center bus, and all three qubits want to travel in different directions. We assume that for all qubits, the distance between their start location and their destination is the same. The individual shortest paths are shown in Figure 3.14, but they are not conflict free. So, the optimal solution involves moving q_2 in the first step, as well as either q_1 or q_3, to the side. Let's say we pick q_1. Then, we let q_3 travel to its destination, and afterwards also move it to the side. Once q_3 has left the bus, q_1 can move back onto the bus and travel to its destination. Finally, we move q_3, which moved to the side earlier in order to not block the movement of q_1 on the bus, back to its destination.

If we were to use the direction constraints we introduced earlier, and solve the earliest conflict with the lowest[15] qubit first, we get two constraints: The first one restricts q_1 from moving downwards, and the second one restricts q_2 from idling the bus (which is required for it to move right). Following either constraint, there is going to be another direction conflict at the same timestep with q_3.

To resolve this situation with just one step in the CBS algorithm, we introduce a global direction constraint. By *global* constraint, we mean a constraint that is not specific to one agent, but applies equally to all agents. We already saw global constraints before: Timing constraints, which restrict the time of the plans of all agents equally, are global constraints.

Definition 3.11. *A **global direction constraint** is a tuple $\langle \mathfrak{w}, \mathfrak{D}, t \rangle$ that restricts all agents from traversing an edge $e \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{M}}(e)$ with $\mathfrak{d} \in \mathfrak{D}$ at time t.*

[15] Remember that agents in MAPF are just numbers from 1 through k, so "lowest" refers to the order on these natural numbers.

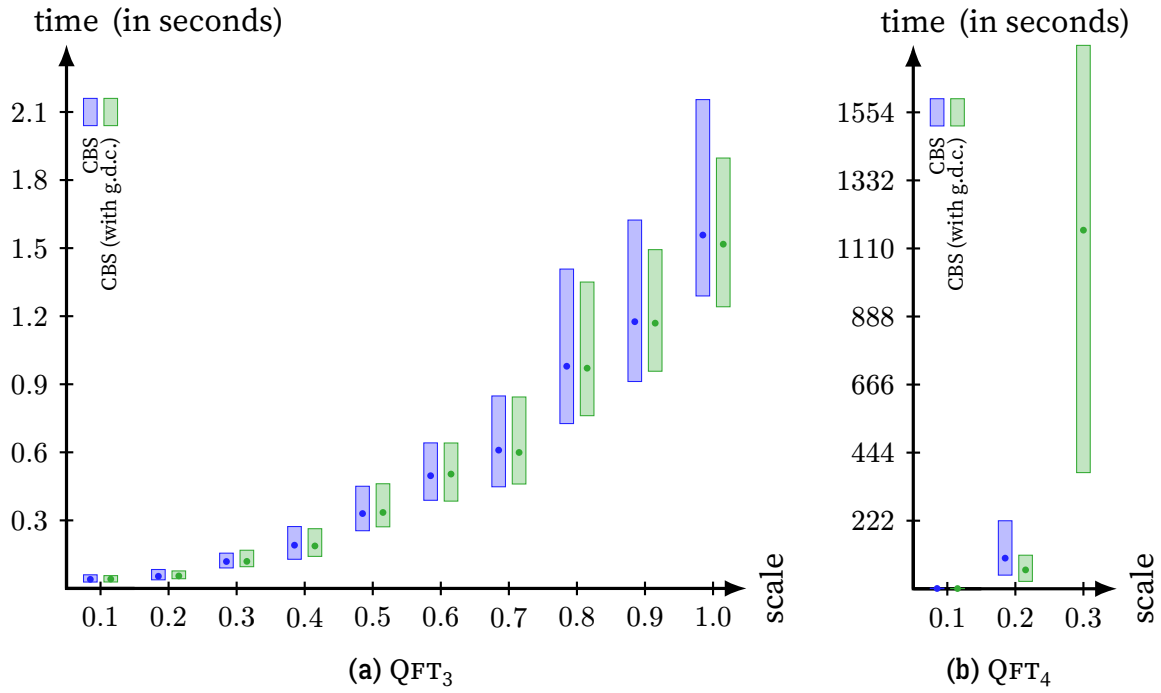
We generate these constraints as follows. If we encounter a direction conflict, where we previously would have generated the (non-global) direction constraints $\langle i, \mathfrak{w}, \mathfrak{d}, t\rangle$ and $\langle \ddot{\imath}, \mathfrak{w}, \mathfrak{d}', t\rangle$, we now generate the global direction constraints $\langle \mathfrak{w}, \{\mathfrak{d}\}, t\rangle$ and $\langle \mathfrak{w}, \{\mathfrak{d}'\}, t\rangle$. Using a set of directions instead of a single direction will become relevant later in subsection 5.2.1.

Lemma 3.12. *The constraints $\langle \mathfrak{w}, \{\mathfrak{d}\}, t\rangle$ and $\langle \mathfrak{w}, \{\mathfrak{d}'\}, t\rangle$ with $\mathfrak{d} \neq \mathfrak{d}'$ are mutually disjunctive.*

Proof. Assume this wasn't true. Then, there are conflict-free plans violating both constraints. In order to violate the first constraint, the plan π_i for some agent i has to traverse an edge $e \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{M}}(e)$ at time t. For the second constraint to be violated, the plan $\pi_{\ddot{\imath}}$ for some agent $\ddot{\imath}$ has to traverse an edge $e' \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}') \in \mathfrak{A}_{\mathfrak{M}}(e)$ at time t. However, since $\mathfrak{d} \neq \mathfrak{d}'$, there is a direction conflict between the agents i and $\ddot{\imath}$. This contradicts the assumption. □

3.5 Qubit Routing Performance using Global Direction Constraints

We run the benchmark again, this time using CBS with global direction constraints (g.d.c.). The result can be found in Tables B.2 and B.3 in the appendix, alongside the result from Table 3.12. We visualise the time it took for QFT_3 in Figure 3.15a and for



(a) QFT_3 (b) QFT_4

Figure 3.15: Benchmark between CBS with and without global direction constraints (QFT_3 and QFT_4)

QFT_4 in Figure 3.15b. In the latter plot, the results for scale 0.1 are hardly visible as they lay on top of the x axis.

The results for QFT_3 show that global direction constraints were able to improve the consistency of the runtime slightly, while the average runtime did not change much. We see a bigger improvement in the QFT_4 benchmark. Not only did it take less time to finish routing at scale factors 0.1 and 0.2, but CBS with global direction constraints also managed to finish scale 0.3, where it previously ran out of memory. Alltogether, we can conclude that using global direction constraints is an improvement.

CHAPTER 4

Corridor Symmetry

We saw in the previous chapter that optimising the handling of direction conflicts can have a positive impact on performance. In this chapter, we attempt to improve this further.

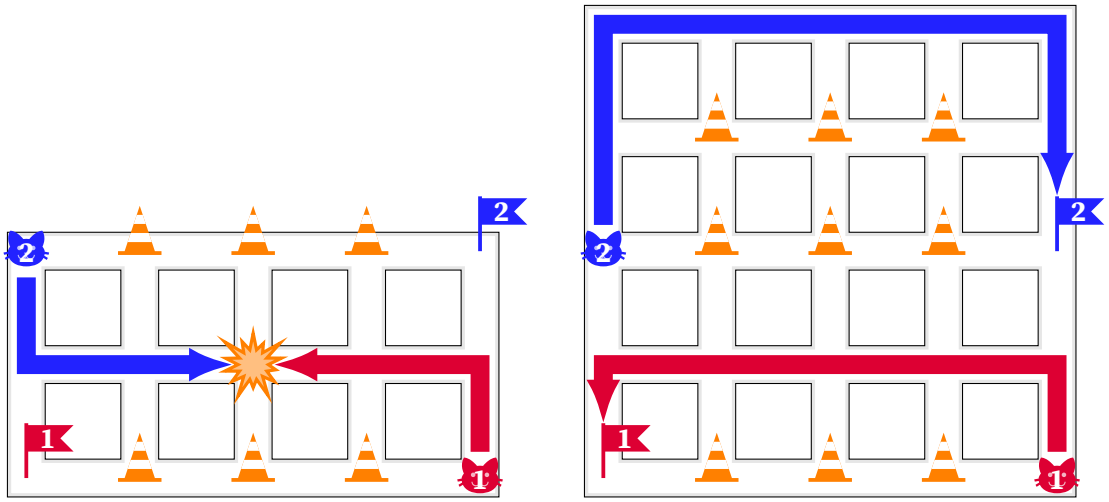
Direction conflicts usually don't come alone, especially if we were to consider scale factors closer to 1, where the distance between intersections increases: If two qubits compete for the same bus, the qubit that "won" the first constraint will usually want to continue travelling on the bus for another timestep, and the qubit that "lost" still wants to start its travel in the opposite direction.

We can see an example of this when routing the 3rd layer of QFT_3 from Figure 3.8c. A step-by-step invocation of CBS, showing conflicts and chosen constraints, is shown in the appendix in section A.1. We can see that the conflicts in Figure A.2e through Figure A.2v look "similar": We first get a direction conflict between q_1 and q_2 where q_1 wants to move right/forward and q_2 wants to move left/reverse, and then another one where q_2 wants to idle instead. And then again for the next timestep. In total, this pattern repeats 9 times. Apart from Figure A.2h, the constraint that leads to the optimal solution is also similar between all conflicts.

This leads us to a question: Can we exploit these symmetries to improve runtime? It turns out, in vanilla CBS, we can. This was already explored in terms of so called *corridor conflicts* in [15]. We visit the existing work and try to apply it to CBS for Q-MAPF.

Sections of the graph inbetween two intersections are called *corridors*. The vertices of these sections (besides the two intersections) must be of degree 2. This means that qubits inside of a corridor cannot leave the corridor until they are at one of the two exits. A corridor can thus never accomodate two agents travelling in opposite directions. Since we clearly saw in the last chapter that our algorithm needs an improvement, we would like to exploit this fact by identifying symmetric cases in corridors and resolve them more efficiently. We define corridors the same as [15].

Definition 4.1. *A **corridor** $C = C' \uplus \{v_1, v_2\}$ is a chain of connected vertices $C' \subseteq V$, each of degree 2, together with two endpoints $v_1 \neq v_2 \in V$ connected to C'. Its **length** is the distance between its two endpoints, i.e. $|C'| + 1$.*



(a) A corridor conflict between agents

(b) A detour for agent 2 around the corridor conflict

Figure 4.1: An example of a corridor conflict (adopted from [15, Fig. 13])

4.1 Corridor Conflicts in vanilla CBS

As we said before, symmetries, including corridors, have been studied for vanilla CBS in [15]. We start by summarising their work in this section.

A *corridor conflict*, unlike the conflicts we looked at previously, is no longer bound to one single timestep. It extends vertex or edge conflicts that occur in a corridor. We consider any vertex or edge conflict that occurs inside of a corridor to be a corridor conflict if both agents enter the corridor in opposite directions. An example of a corridor conflict is depicted in Figure 4.1a. The center row forms a corridor, where the two exits are the leftmost and rightmost vertices. There is a conflict, because both agents travel through the corridor in opposite directions at the same time. In the example, this leads to a vertex conflict, but if the corridor had an even instead of an odd number of vertices, it would be a direction conflict instead.

Definition 4.2. *Let π and π' be two single-agent plans. If there is a vertex or an edge conflict between both plans, both agents are inside of the same corridor at the time of the conflict, and they enter the corridor in opposite directions, there is a* ***corridor conflict*** *between the two plans.*

To resolve a corridor conflict, we generate two range constraints. A *range constraint* is basically a vertex contraint with a time interval instead of a single timestep. Each constraint prioritises one of the two agents by placing a constraint at the exit of the corridor for the other agent. This constraint forces the agent to wait until the prioritised agent has exited their corridor.

This way of generating constraints imposes a restriction on the type of problems we can solve. Assume there are two agents involved in a corridor conflict, both of

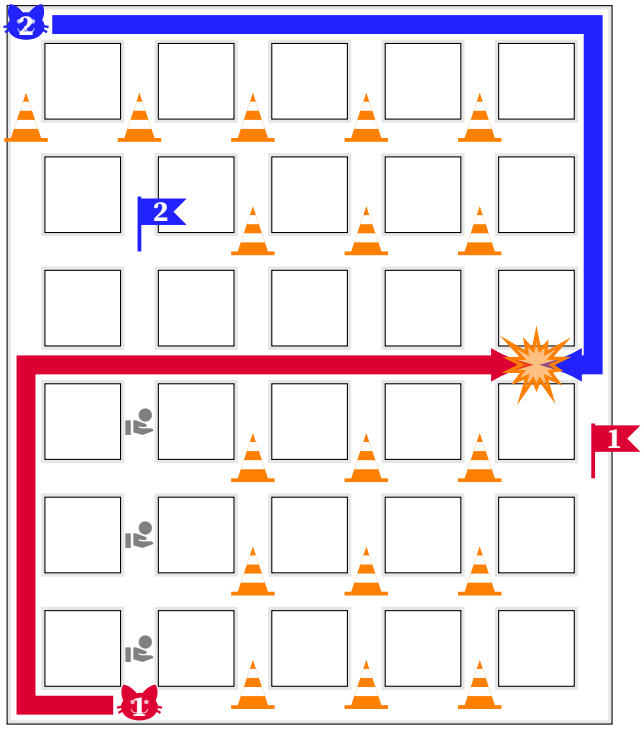


Figure 4.2: An example where the shortest path is not the one with minimal costs

which arrive at the corridor with a path with minimum cost, but there exists a shorter path (i.e. one that takes less time) with higher cost. An example for this situation is depicted in Figure 4.2: The shortest path for agent 1 would make it go straight up from its start location, so that it reaches its destination just before agent 2 enters the corridor. Thus, the shortest path does not cause a conflict. The paths that are depicted in Figure 4.2 are those with minimal cost for each agent. This path thus avoids edges with higher cost, which are marked by the symbol. The detour that agent 1 takes to minimise its cost however means that the agent goes through the corridor later, and causes an edge conflict with agent 2 just before it leaves the corridor. If we generated two range constraints for this conflict, we would restrict the agents, one per constraint, from occupying their respective corridor exit before a certain timestep. But, this prevents the valid, conflict-free solution of each agent taking their shortest path. This is in direct conflict with the mutual disjunctive property we discussed earlier. Thus:

Definition 4.3. *For this entire chapter, we define cost equal to time. Since each move action along an edge and each wait action at a vertex require exactly one timestep, we define*

- $w(e) := 1$ *for all* $e \in E$
- $w(v) := 1$ *for all* $v \in V$

Before we can generate the constraints, we need to analyse the shortest path that lead to the corridor. We also need to consider detours. A *detour* is a path for either

agent that leads to the exit of the corridor without going through the corridor. You can see a detour in Figure 4.1b. We define the following helper functions:

Definition 4.4. *Let $G = (V, E)$ be the graph of a MAPF problem, and $C \subseteq V$ a corridor. Then,*

- $\bar{\mathcal{O}}_i(v)$ *denotes the earliest timestep at which agent i can be at vertex v*
- $\bar{\mathcal{O}}_{\not i}(v)$ *denotes the earliest timestep at which agent i can be at vertex v without going through the corridor C*

Using these helper functions, we are now able to generate range constraint. As said before, a range constraint is placed on the exit of the corridor and restricts the agent from being at the exit before a certain timestep. Assume agents i and $\ddot{\imath}$ are involved in a corridor conflict inside of corridor C with length ℓ. Let v_1 and v_2 be the two exit vertices of the corridor, s.t. agent i traverses the corridor from v_2 to v_1 and $\ddot{\imath}$ in the opposite direction. It thus takes agent i at least until $\bar{\mathcal{O}}_i(v_1) + 1$ to be clear of the corridor, and agent $\ddot{\imath}$ at least another ℓ timesteps to reach its corridor exit, unless agent $\ddot{\imath}$ takes a detour. With this information, we generate the constraint that prioritises agent i, restricting $\ddot{\imath}$ from being at v_2 until either $\bar{\mathcal{O}}_i(v_1) + \ell + 1$ or $\bar{\mathcal{O}}_{\not\ddot{\imath}}(v_2)$. The other constraint is generated symmetrically.

Definition 4.5. *A **range constraint** is a tuple $\langle i, v, [t_{\min}, t_{\max}]\rangle$ that restricts agent i from occupying vertex v at any timestep in the interval from $t_{\min}$ to $t_{\max}$ (inclusive).*

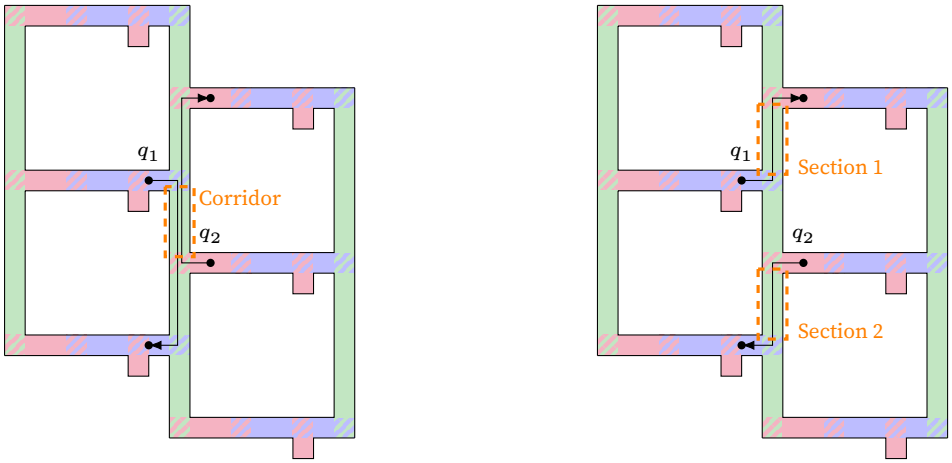
In order to use these generated range constraints for CBS, we need to proove that they still guarantee optimality. We established before that showing that the constraints are mutually disjunctive is sufficient to show optimality. Unlike the previous constraints, the context the range constraint was generated within is important for mutual disjunctiveness.

Lemma 4.6. *Let $i \neq \ddot{\imath}$ be two agents involved in a vertex or edge conflict inside of a corridor C, and let v_1 and v_2 be the two exit vertices of the corridor s.t. agent i traverses the corridor from v_2 to v_1 and agent $\ddot{\imath}$ traverses the corridor in the opposite direction. Then, the range constraints $\langle i, v_1, [0, \min\{\bar{\mathcal{O}}_{\ddot{\imath}}(v_2) + \ell, \bar{\mathcal{O}}_{\not i}(v_1) - 1\}]\rangle$ and $\langle \ddot{\imath}, v_2, [0, \min\{\bar{\mathcal{O}}_i(v_1) + \ell, \bar{\mathcal{O}}_{\not\ddot{\imath}}(v_2) - 1\}]\rangle$ are mutually disjunctive.*

Lemma 4.6 was proven in [15].

4.2 Conveyor Conflicts in CBS for Q-MAPF

In CBS for Q-MAPF, the range constraints from vanilla CBS, such as the one in Figure 4.3a, remain valid. But this is of little use, at least in our examples. In fact, neither QFT_3 (Figure 3.8), nor QFT_4 (Figure 3.9), nor ShorEnc (Figure 3.10) contain a single corridor conflict in the shortest paths as drawn in the mentioned figures.



(a) A direction conflict inside a corridor (b) A direction conflict on the same bus

Figure 4.3: Examples of direction conflicts in Q-MAPF

Luckily, we have a lot more cases that are similar to a corridor. This is due to the fact that conveyors are longer than a single corridor. Consider the conflict in Figure 4.3b. There is no corridor conflict according to vanilla CBS, as there is no edge conflict and also no single corridor that the two qubits traverse in opposite directions. But the two marked sections of the bus are corridors, and the direction conflict happens while each qubit is in a corridor, eventhough not the same corridor. So what we are looking for is a corridor symmetry that takes into account agents in different corridors that conflict due to both corridors being on the same conveyor.

We call these conflicts *conveyor conflicts*. A conveyor conflict occurs when there is a direction conflict between two agents, and each agent is in a corridor (but not necessarily in the same corridor as the other agent). Unlike with corridor conflicts, we do not consider vertex or occupancy conflicts. At least on the busses, where most of the conflicts occur, the conveyor does not change on either end of the corridor. Therefore, where the first conflict in vanilla CBS that could occur might be a vertex conflict, in CBS for Q-MAPF the first conflict will be a direction conflict at the time where both agents are inside the corridor.

Definition 4.7. *Let π and π' be two single-agent plans. If there is a direction conflict between both plans, and both agents are inside of a corridor at the time of the conflict, there is a* ***conveyor conflict*** *between the two plans.*

If either corridor is of length less than 2, i.e. it takes one agent only a single timestep to leave the corridor, we resolve the direction conflict directly. We do the same if one of the agents is idle. With that, we specifically mean that it assigns Idle to all wires that can control its movement, i.e. if the agent does not travel at all. This is because we generally require a "direction" in which a qubit travels through the corridor, and we cannot observe such a direction when the qubit doesn't move along the conflicting wire. Note that for SpinBus, there generally are no corridors with a length of at least 2 where the conflicting wire is set to Idle. Otherwise, if none of the

corridors are too short and none of the agents idle, we generate range constraints as follows.

We start by finding the section of the conveyor belt we treat as the corridor. For this, we define the partial function S which works as follows. Let π be the plan for an agent involved in the direction conflict, t the timestep at which the conflict occurs, $\mathfrak{w}$ the wire involved in the conflict, and $\mathfrak{d}$ the direction that the agent's plan wants to assign to $\mathfrak{w}$. We start by setting the start vertex of the corridor to $\pi[t]$ and the exit vertex to $\pi[t+1]$. If these are identical, or $\mathfrak{d} = \text{Idle}$, we stop. Otherwise, we look at the current exit vertex. If it is of degree 2, one of the neighbors has to be part of the corridor already, and we call the other one u. This vertex u will become the new exit vertex if it is part of the immediate path of the agent, and requires the wire $\mathfrak{w}$ to be assigned the direction $\mathfrak{d}$. This is repeated until the new exit vertex that was found no longer matches the criteria. Note that it is absolutely possible for an agent to wait or change direction inside a corridor, in which case this condition will stop the search.

Definition 4.8. *The partial function S is defined as*

$$S(\pi, t, \mathfrak{w}, \mathfrak{d}) \coloneqq S'(\pi, t+1, \mathfrak{w}, \mathfrak{d}, \{\pi[t], \pi[t+1]\}, \pi[t], \pi[t+1])$$

$$S'(\pi, t, \mathfrak{w}, \mathfrak{d}, C, v_1, v_2) \coloneqq \begin{cases} \bot & \text{if } v_1 = v_2 \text{ or } \mathfrak{d} = \text{Idle} \\ (C, v_1, v_2) & \text{if } \text{degree}(v_2) \neq 2 \\ & \text{or } (\mathfrak{w}, \mathfrak{d}) \notin \mathfrak{A}_{\mathfrak{W}}(v_2, u) \\ & \text{or } \pi[t+1] \neq u \\ S'(\pi, t+1, \mathfrak{w}, \mathfrak{d}, C \cup \{u\}, v_1, u) & \text{otherwise} \end{cases}$$

where $\{u\} = N(v_2) \setminus C$.

Since S only advances the corridor in the direction of the exit, and only if the exit is of degree two, all vertices but the start and exit must have degree two. This satisfies the constraints for a corridor from Definition 4.1.

Corollary 4.9. *The set of vertices returned from the partial function S form a corridor.*

Using this function, we can now resolve a conveyor conflict as follows. If the function S returns a corridor for both agents involved in the conflict, we generate two range conflicts. The range conflict prevents the non-prioritised agent to occupy the exit node v_2 of its corridor of length ℓ, where ℓ' is the length of the other agent's corridor, for the range $[t+\ell, \min\{t+\ell+\ell', \bar{\mathfrak{G}}(v_2)\}]$.

You see some slight differences here as compared to the range conflicts generated by vanilla CBS. In particular, we start the range at $t+\ell$ instead of 0, and compute the timestep where the other agent finishes its corridor as $t+\ell'$ instead of $\bar{\mathfrak{O}}(v_2)$. This deviation is unnecessary if this is the first conflict CBS resolves. However, there might be a list of constraints already placed before CBS resolves the conveyor conflict at hand. And these constraints might result in an agent travelling to the start of the corridor, away from it, and back to it. In vanilla CBS, this wouldn't really happen, as the agent could just as well wait. But idling at a position in CBS for SpinBus is more likely to cause a conflict than in vanilla CBS, hence the change.

We cannot do the same for the detour though. Ideally, we'd compute the detour as something like $t + \bar{\mathfrak{C}}'(v_2)$ for some function $\bar{\mathfrak{C}}'$ that computes the minimum time it takes to travel from start to exit of the corridor without going through it. However, since this detour might involve "undoing" the last few steps of the path, the time that the low level finds might be shorter since it won't take these unneccesarry steps, thereby never travelling to the start of the corridor. Therefore, we stick with $\bar{\mathfrak{C}}(v_2)$.

Using these constraints, we can solve the examples in Figure 4.1 more efficiently than without. To be able to use them for any Q-MAPF problem and not just the two examples, we need them to retain optimality of CBS. This leads us to the following hypothesis, which we will end up disproving later.

Hypothesis 4.10. *Let $i \neq \ddot{\imath}$ be two agents and $\mathfrak{w}$ a wire involved in a conveyor conflict at time t, where i wants to assign $\mathfrak{d}_i$ and $\ddot{\imath}$ wants to assign $\mathfrak{d}_{\ddot{\imath}}$ to $\mathfrak{w}$. Further, let π_i be the plan of i and $\pi_{\ddot{\imath}}$ the plan of $\ddot{\imath}$. If there are two corridors $(C_i, v_{1,i}, v_{2,i}) = S(\pi_i, t, \mathfrak{w}, \mathfrak{d}_i)$ of length ℓ_i and $(C_{\ddot{\imath}}, v_{1,\ddot{\imath}}, v_{2,\ddot{\imath}}) = S(\pi_{\ddot{\imath}}, t, \mathfrak{w}, \mathfrak{d}_{\ddot{\imath}})$ of length $\ell_{\ddot{\imath}}$, then the range constraints $\langle i, v_{2,i}, [t + \ell_i, \min\{t+\ell_i+\ell_{\ddot{\imath}}, \bar{\mathfrak{C}}_i(v_{2,i})\}]\rangle$ and $\langle \ddot{\imath}, v_{2,\ddot{\imath}}, [t+\ell_{\ddot{\imath}}, \min\{t+\ell_i+\ell_{\ddot{\imath}}, \bar{\mathfrak{C}}_{\ddot{\imath}}(v_{2,\ddot{\imath}})\}]\rangle$ are mutually disjunctive.*

4.3 Using Conveyor Conflicts for QFT_3

Let us try using conveyor conflicts on the example we used before: The 3rd layer of QFT_3. The result can be found in the appendix in section A.2. Right off the bat, we can see some symmetrical conflicts: From Figure A.3h through to Figure A.3z, we see a plethora of direction conflicts with slowly increasing timesteps. These follow a pattern where there are one or more direction/conveyor conflict between q_1 and q_2 where both want to move in opposite directions, and a direction conflict where one moves and one wants to idle, at the same timestep, for multiple timesteps in a row. This is exactly what the constraint generation based on conveyor conflicts was supposed to prevent. Even more troubling, in Figure A.3j through Figure A.3l and in Figure A.3q and Figure A.3r, we see that the exact same direction conflict reappeared. The conveyor conflicts based on these direction conflicts are not the same, as the corridors are different, but the fact that the conveyor conflict did not resolve its underlying direction conflict is not a sign of efficient conflict resolution.

But how about optimality? Let us look at the optimal solution we obtained without conveyor conflicts, Figure A.2x. This involves visiting the center junction c at time $t = 14$. But, in Figure A.3q, we get a range constraint $\langle q_2, c, [10, 15]\rangle$, preventing this optimal solution. The other constraint generated alongside that range constraint would delay q_1, thus increasing the overall cost and leading to a suboptimal solution. So while this ended up not preventing all optimal solutions, it prevented one optimal solution. This is enough to disprove Hypothesis 4.10, but not yet enough to show that using constraint generation based on conveyor conflicts can lead to suboptimal results.

(a) Direction constraint $\langle q_2, \text{cont}, \text{Idle}, 2\rangle$

(b) Range constraint $\langle q_1, c, [7, 7]\rangle$

(c) Range constraint $\langle q_2, rm, [3, 6]\rangle$

(d) Direction constraint $\langle q_2, \text{cont}, \text{Idle}, 3\rangle$

(e) Range constraint $\langle q_2, rm.1.c, [4, 6]\rangle$

(f) Occupancy constraint $\langle q_2, \{h, h.1.rm\}, 2\rangle$

(g) Range constraint $\langle q_2, h, [4, 7]\rangle$

(h) Direction constraint $\langle q_1, \text{cont}, \text{Forward}, 4\rangle$

(i) Range constraint $\langle q_2, rm, [8, 10]\rangle$

(j) Range constraint $\langle q_2, rm.1.c, [7, 9]\rangle$

(k) Range constraint $\langle q_2, rm.2.c, [6, 8]\rangle$

(l) Direction constraint $\langle q_2, \text{cont}, \text{Idle}, 5\rangle$

(m) Direction constraint $\langle q_2, \text{cont}, \text{Reverse}, 4\rangle$

(n) Direction constraint $\langle q_2, \text{cont}, \text{Idle}, 6\rangle$

(o) Range constraint $\langle q_2, c, [7, 8]\rangle$

(p) Occupancy constraint $\langle q_2, \{c, c.1.irz\}, 8\rangle$

(q) Direction constraint $\langle q_2, \text{cont}, \text{Idle}, 7\rangle$

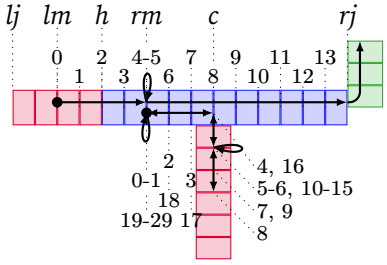
(r) Range constraint $\langle q_2, c, [9, 9]\rangle$

(s) Range constraint $\langle q_2, c.1.rj, [8, 8]\rangle$

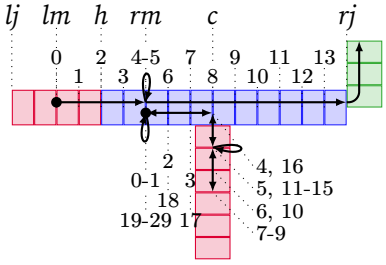
(t) Range constraint $\langle q_2, c, [10, 15]\rangle$

(u) Range constraint $\langle q_2, c.2.irz, [8, 8]\rangle$

Table 4.4: List of constraints while solving the 3rd layer of QFT_3



(a) The low-level paths following the constraints



(b) A possible optimal solution

Figure 4.5: The state after constraints from Table 4.4

And it gets worse. Consider the constraints in Table 4.4. We obtained these constraints from the implementation that we use for benchmarking after adding some debugging aids. The low level follows "hasmap insertion order", which is basically non-determinism, instead of the rules we used in Appendix A. Also, the implementation has to resolve timing conflicts as they appear, which we chose to skip for the example. It is thus not surprising that the constraints from the implementation differ from those we obtained manually. Of course, the constraint generation should be done in a way that regardless of the low level, conflict resolution order etc., the solution remains optimal.

The constraints in Table 4.4 do not yet prevent us from finding an optimal solution: A possible optimal solution that is consistent with the constraints is shown in Figure 4.5b. So far, this just looks like a normal constraint set in some CT node that CBS has not yet expanded. It therefore represents an intermediate state within the CBS algorithm. Alongside the constraints, CBS also stores the current plans for the agents in the CT node. These are shown in Figure 4.5a.

The paths in Figure 4.5a are not conflict free. This is not a problem since they represent an intermediate state of CBS, and not a solution. We must therefore continue the algorithm. The next conflict is a direction conflict between q_1 and q_2 at $t = 7$. q_1 would like the split wire to idle, but q_2 would like to move downwards. Both qubits are part of a single-element corridor, q_1's corridor exit is c and q_2's corridor exit is $c.3.irz$. The corresponding range constraints would block that exit for $t \in [8, 8]$.

The constraint for q_1 would delay it, thus increasing overall costs. This does not lead to an optimal solution. However, the constraint for q_2, in conjunction with the existing constraint (u), now prevent q_2 from idling at $t = 7$: The only remaining options for q_2 to idle at $t = 7$ and $t = 8$ as required for the optimal solution are $c.4.irz$ and irz. However, q_2 will not reach these vertices in time, thus still moving towards them at $t = 7$.

We have thus just created two constraints that each block all possible optimal solutions which were possible without either constraint. This not only disproves Hypothesis 4.10, which claims mutual disjunctiveness, but destroys any hope of constraint generation based on conveyor conflicts in the way we defined them from being optimal.

Theorem 4.11. *CBS using constraint generation based on conveyor conflicts, such as described in Hypothesis 4.10, does not always yield an optimal solution.*

4.4 Qubit Routing Performance using Conveyor Conflicts

Despite knowing that CBS using constraint generation based on conveyor conflicts can lead to suboptimal solutions, we are still interested in its performance. If the performance gains are significant, it could still be a worthwhile improvement. Therefore, we implemented the corridor detection described in Definition 4.8, and the constraint generation, into our benchmark code base.

		CBS (with g.d.c.)		CBS (with corridors)	
Circuit	Scale	Avg	SD	Avg	SD
QFT_3	0.1	0.041	0.0060	0.423	0.2086
	0.2	0.055	0.0082	0.490	0.1967
	0.3	0.119	0.0135	0.929	0.4084
	0.4	0.187	0.0296	3.872	1.6944
	0.5	0.335	0.0431	11.880	6.0069
	0.6	0.504	0.0593	34.396	25.7553
	0.7	0.599	0.0821	39.961	27.7749
	0.8	0.971	0.1303	147.571	72.9804
	0.9	1.169	0.1174	147.705	47.3405
	1.0	1.517	0.1700	DNF	
QFT_4	0.1	0.688	0.2101	DNF	
	0.2	61.505	25.1929		
	0.3	1169.648	439.1450		
	0.4	DNF			
ShorEnc	0.1	DNF		DNF	

Table 4.6: Benchmark between CBS with global direction constraints and CBS with conveyor conflicts and range constraints (Average and standard deviation; in seconds)

We then run the same benchmark from section 3.3 for CBS for Q-MAPF with conveyor constraint resolution. The result can be found in Tables 4.6 and B.1, alongside the previous best results of CBS from Table B.2. Unfortunately, it is quite obvious that this is not the desired result. The performance is worse in every single test. Besides that, CBS with corridors also fails to compute even more circuits. Not only did it not finish QFT_4 even at scale 0.1, it also failed QFT_3 at scale 1.0.

4.5 Alternative Approaches

Since we were unable to keep optimality as proven in Theorem 4.11, we tried multiple similar yet different approaches that we expected to not retain optimality beforehand. The results that we saw were similar among all of them: Using corridors sometimes reduced the number of CT nodes that CBS expanded with costs less than or equal to those of the optimal solution. But ultimately, none performed better than CBS without corridors: Since the solution it finds is suboptimal, it ends up expanding more CT nodes than before.

The first alternative we considered was to "enlarge" the time that a corridor exit is blocked. In the constraint generation we described above in detail, we used the length of the other agent's corridor to determine the time a corridor exit is blocked

for. However, an agent may travel for more than one corridor on a bus, i.e. continue its travel on the bus past an intersection. We tried an approach that computes the total timesteps that an agent "blocks" the bus and uses that to compute the range constraint. We did see an improvement in performance over the "shorter" corridor approach, but not significant enough to make a huge difference.

The second alternative we considered was to block not only the exit of a corridor but also its entrance. This approach, by design, prevents the repetitive constraints we saw in section A.2. Unfortunately, it ended up blocking a lot more valid solutions as well. In the end, we saw worse performance alongside more suboptimal solutions.

Performance wise, the problems remains that, if the optimal solution was excluded by the constraints that were generated, CBS had to continue searching. This meant that in total, a lot more CT nodes had to be expanded before a solution was found (or we ran out of memory). Unfortunately, we must conclude that this approach has not yielded the desired result.

While it might be possible to derive a working version of the CBS constraint generation that exploits corridor symmetry in CBS for Q-MAPF, we failed to find one. It seems that, while conveyors provide some domain specific knowledge for SpinBus, it is hard to place these into hard constraints. A better approach, which was not explored at all in this thesis, might be to use the knowledge to prioritise the expansion of some CT nodes, as part of an heuristic approach, that are less likely to cause cascading direction conflicts.

CHAPTER 5

Disjoint Constraints

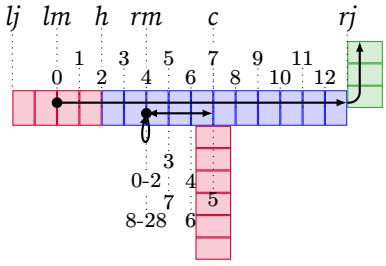
Our previous attempt from chapter 4, i.e. changing constraint generation to exploit corridor symmetries, ended up not giving us a desired result. It neither retained optimality, nor improved runtime.

Another promising idea is *disjoint splitting*. Introduced in [12], it changes constraint generation to prevent "duplicates". Take a look at Figure 5.1, a conflict from Figure A.2 of the step-by-step execution of CBS on the 3rd layer of QFT_3. We chose the constraint $\langle q_2, \text{cont}, \text{Reverse}, 5\rangle$, which lead to an optimal solution. The other constraint generated in this step, i.e. the "sibling" in the CT, is $\langle q_1, \text{cont}, \text{Forward}, 5\rangle$. This constraint looks familiar: The next constraint we pick in Figure A.2h is exactly that constraint. In other word, both constraints generated for this conflict lead to exactly the same optimal solution (which means the cont wire is set to Idle at $t = 5$).

And this is a problem for the performance of CBS. Unlike A*, CBS does not maintain a closed list. So, if there are two paths that end up generating the same constraints, just in a different order, CBS will continue expanding both CT subtrees. This is duplicate work that should be avoided.

Disjoint splitting [12] is an approach to constraint generation to prevent this very situation. The idea is simple: When generating constraints, we want every plan, conflict-free or not, to only be permitted by one of the constraints.

Definition 5.1. *Two or more constraints are called* ***disjoint*** *if every plan is permitted by at most one constraint.*



Conflict 7
Direction conflict at $t = 5$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 7
$\langle q_2, \text{cont}, \text{Reverse}, 5\rangle$

Figure 5.1: A conflict from the 3rd layer of QFT_3 (equivalent to Figure A.2g)

Corollary 5.2. *If two or more constraints are mutually disjunctive and disjoint, every conflict-free plan is permitted by exactly one constraint.*

All constraints we introduced previously were *negative constraints*: They prevented a single-agent plan from doing something specific, like occupying a vertex at a certain time. In order to create disjoint constraints, we introduce positive counterparts to these constraints. A *positive constraint* forces a single-agent plan to do something specific.

Definition 5.3. *A **positive constraint** forces an agent to take a specific action. It may restrict other agents from taking the same action. A **negative constraint** restricts agents from taking certain actions, but never forces agents to take a specific action.*

The vertex, edge, occupancy and direction constraints we introduced in earlier chapters were all negative constraints. We used the tuple-syntax $\langle \dots \rangle$ to refer to them. To indicate a positive constraint, we add the word *positive* to its name, and use the tuple-syntax $\langle\!\langle \dots \rangle\!\rangle$.

5.1 Disjoint Splitting in Vanilla CBS

For vanilla CBS, [12] gives us disjoint constraint generation for vertex conflicts. A vertex conflict between agents i and $\ddot{\imath}$ over vertex v at time t can be resolved using a positive constraint forcing i to occupy v at time t, alongside a negative constraint preventing i to occupy v at time t. The positive constraint implicitly restricts all other agents from doing the same thing.

Definition 5.4. *A **positive vertex constraint** is a tuple $\langle\!\langle i, v, t \rangle\!\rangle$ that forces agent i to occupy vertex v at time t, and restricts all other agents from occupying vertex v at time t.*

Lemma 5.5. *The constraints $\langle\!\langle i, v, t \rangle\!\rangle$ and $\langle i, v, t \rangle$ are mutually disjunctive.*

Lemma 5.6. *The constraints $\langle\!\langle i, v, t \rangle\!\rangle$ and $\langle i, v, t \rangle$ are disjoint.*

The proof for Lemma 5.5 and Lemma 5.6 can be found in [12].

While [12] does not give a positive constraint for edge conflicts, we can create one using the same idea. For an edge conflict, we force one agent to use the edge in question, implicitly preventing all other agents from traversing the edge in opposite direction, or restrict the agent from using the edge.

Definition 5.7. *A **positive edge constraint** is a tuple $\langle\!\langle i, u, v, t \rangle\!\rangle$ that forces agent i to traverse the edge (u, v) at time t in the direction from u to v, and restricts all other agents from traversing the edge (u, v) at time t in the direction from v to u.*

Lemma 5.8. *The constraints $\langle\!\langle i, u, v, t \rangle\!\rangle$ and $\langle i, u, v, t \rangle$ are mutually disjunctive.*

Proof. Assume this wasn't true. Then there are conflict-free plans violating both constraints. In order to violate the second (negative) constraint, the plan π_i for agent i must traverse the edge (u, v) in direction from u to v at time t. Then, π_i does not violate the first constraint. So some other agent $\ddot{\imath}$ with plan $\pi_{\ddot{\imath}}$ must violate the first constraint. That means that $\pi_{\ddot{\imath}}$ must traverse the edge (u, v) in direction from v to u at time t. But then there is an edge conflict between π_i and $\pi_{\ddot{\imath}}$, which means they were not conflict free, contradicting the assumption. □

Lemma 5.9. *The constraints $\langle\!\langle i, u, v, t\rangle\!\rangle$ and $\langle i, u, v, t\rangle$ are disjoint.*

Proof. Let $\Pi = (\pi_1, \dots, \pi_k)$ be a (not necessarily conflict-free) joint plan with plan π_i for agent i. If i traverses the edge (u, v) at time t in direction from u to v, i.e. $\pi_i[t] = u$ and $\pi_i[t+1] = v$, π_i satisfies the first but not the second constraint. Otherwise, i does not traverse the edge, so the first constraint is violated while the second constraint is satisfied. □

The third type of constraint we introduced for vanilla CBS, timing constraints, are already disjoint: For a positive integer t, any single-agent plan π either satisfies $|\pi| \leqslant t$ or $|\pi| > t$, but never both.

Corollary 5.10. *The timing constraints $\langle \leqslant t\rangle$ and $\langle > t\rangle$ are disjoint.*

5.2 Disjoint Splitting in CBS for Q-MAPF

We now develop variants of the constraints we used for CBS for Q-MAPF with the goal of making them disjoint. We design the positive version of the occupancy constraint analogous to the positive vertex conflict:

Definition 5.11. *A **positive occupancy constraint** is a tuple $\langle\!\langle i, c, t\rangle\!\rangle$ that forces the agent i to occupy a vertex $v \in V$ at time t whose distance to the center $\|v, c\|$ is less than $\frac{\mathfrak{md}}{2}$, and restricts all other agents from occupying any such vertex v at time t.*

Lemma 5.12. *The constraints $\langle\!\langle i, c, t\rangle\!\rangle$ and $\langle i, c, t\rangle$ are mutually disjunctive.*

Proof. Assume this wasn't true. Then there are conflict-free plans violating both constraints. In order to violate the second (negative) constraint, the plan π_i for agent i must occupy a vertex $v \in V$ at time t that is too close to the center vertex c: $\|v, c\| < \frac{\mathfrak{md}}{2}$. Then, π_i does not violate the first constraint. So some other agent $\ddot{\imath}$ with plan $\pi_{\ddot{\imath}}$ must violate the first constraint. That means that $\pi_{\ddot{\imath}}$ must occupy a vertex $u \in V$ at time t s.t. $\|u, c\| < \frac{\mathfrak{md}}{2}$. Then, $\|v, u\| \leqslant \|v, c\| + \|c, u\| < \frac{\mathfrak{md}}{2} + \frac{\mathfrak{md}}{2} = \mathfrak{md}$, i.e. the vertices v and u are closer than the minimum distance $\mathfrak{md}$. This means there is an occupancy conflict between π_i and $\pi_{\ddot{\imath}}$, so they were not conflict free, contradicting the assumption. □

Lemma 5.13. *The constraints $\langle\!\langle i, c, t\rangle\!\rangle$ and $\langle i, c, t\rangle$ are disjoint.*

Proof. Let $\Pi = (\pi_1, \dots, \pi_k)$ be a (not necessarily conflict-free) joint plan with plan π_i for agent i. If i occupies a vertex $v \in V$ at time t that is within $\frac{\mathfrak{md}}{2}$ of the vertex c, π_i

satisfies the first but not the second constraint. Otherwise, i does not occupy such a vertex v, so the first constraint is violated while the second constraint is satisfied. □

We can follow the same approach for direction constraints:

Definition 5.14. *A **positive direction constraint** is a tuple $\langle\!\langle i, \mathfrak{w}, \mathfrak{d}, t \rangle\!\rangle$ that forces the agent i to traverse an edge $e \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}(e)$ at time t, and restricts all other agents from traversing any edge e' s.t. $(\mathfrak{w}, \mathfrak{d}') \in \mathfrak{A}_{\mathfrak{W}}(e')$ for some $\mathfrak{d}' \neq \mathfrak{d}$ at time t.*

Lemma 5.15. *The constraints $\langle\!\langle i, \mathfrak{w}, \mathfrak{d}, t \rangle\!\rangle$ and $\langle i, \mathfrak{w}, \mathfrak{d}, t \rangle$ are mutually disjunctive.*

Proof. Assume this wasn't true. Then there are conflict-free plans violating both constraints. In order to violate the second (negative) constraint, the plan π_i for agent i must use an edge $e \in E$ at time t that uses the wire assignment, i.e. $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}(e)$. Then, π_i does not violate the first constraint. So some other agent i with plan π_i must violate the first constraint. That means that π_i must use an edge $e' \in E$ at time t s.t. $(\mathfrak{w}, \mathfrak{d}') \in \mathfrak{A}_{\mathfrak{W}}(e')$ for some $\mathfrak{d}' \neq \mathfrak{d}$. However, then there is a direction conflict between π_i and π_i, so they were not conflict free, contradicting the assumption. □

Lemma 5.16. *The constraints $\langle\!\langle i, \mathfrak{w}, \mathfrak{d}, t \rangle\!\rangle$ and $\langle i, \mathfrak{w}, \mathfrak{d}, t \rangle$ are disjoint.*

Proof. Let $\Pi = (\pi_1, \dots, \pi_k)$ be a (not necessarily conflict-free) joint plan with plan π_i for agent i. If i traverses an edge $e \in E$ with $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}(e)$ at time t, π_i satisfies the first but not the second constraint. Otherwise, i does not occupy such an edge e, so the first constraint is violated while the second constraint is satisfied. □

5.2.1 Disjoint Global Direction Constraints

So far we have implemented a disjoint version of the non-global direction constraints. However, as we saw earlier, global direction constraints were an improvement over their non-global counterpart. Hence, we would like to develop a disjoint version of global direction constraints as well.

Since the (negative) global direction constraints introduced in Definition 3.11 are not disjoint on their own, we need a positive version of the constraint to go alongside it. This would then, instead of restricting all agents from assigning a specific direction to a specific wire, force at least one agent to assign a specific direction to a specific wire. At the same time, it would restrict all agents from assigning a different direction to the wire. Since any conflict-free joint plan will have at most one wire direction assignment for a specific wire and timestep, this means that joint plans that assign the specific direction are permitted only by the positive, and all other plans, whether they assign a different direction or no direction at all, are permitted only by the negative constraint.

Draft 5.17. *A **positive global direction constraint** is a tuple $\langle\!\langle \mathfrak{w}, \mathfrak{d}, t \rangle\!\rangle$ that forces any but at least one agent to traverse an edge $e \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}(e)$ at time t, and restricts all agents from traversing an edge $e' \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}') \in \mathfrak{A}_{\mathfrak{W}}(e')$ with $\mathfrak{d}' \neq \mathfrak{d}$ at time t.*

Unfortunately, this is impossible to implement into the CBS algorithm. So far, the high level of CBS resolved a conflict using constraints in each step, and then the low level, which we implemented using A*, found a path consistent with the constraints provided. Now, we are asking for "at least one agent" to take a certain action. It is impossible for each low-level algorithm to know whether it needs to be this one agent, and impossible for the CBS algorithm to then know which of its agent to force to be that one agent.

This means that we will not be able to produce a disjoint version of global direction constraints that is usable with CBS. We can, however, take half of the idea and still improve upon the global direction constraints we had before. We will still use two (negative) constraints, but change the constraint generation.

Let i and $\ddot{\imath}$ be two agents involved in a direction conflict at time t, where agent i would like to assign $\mathfrak{d}$ to the wire $\mathfrak{w}$. So from now on, when using disjoint splitting together with global direction constraints, we generate the constraints $\langle \mathfrak{w}, \{\mathfrak{d}\}, t\rangle$ and $\langle \mathfrak{w}, \{\text{Forward}, \text{Reverse}, \text{Idle}\} \setminus \{\mathfrak{d}\}, t\rangle$. As said before, these two constraints are not disjoint, but reduce the amount of plans permitted by both constraints compared to the constraint generation from section 3.4.

Lemma 5.18. *The constraints* $\langle \mathfrak{w}, \{\mathfrak{d}\}, t\rangle$ *and* $\langle \mathfrak{w}, \{\text{Forward}, \text{Reverse}, \text{Idle}\} \setminus \{\mathfrak{d}\}, t\rangle$ *are mutually disjunctive.*

Proof. Assume this wasn't true. Then, there are conflict-free plans violating both constraints. In order to violate the first constraint, the plan π_i for some agent i has to traverse an edge $e \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}) \in \mathfrak{A}_{\mathfrak{W}}(e)$ at time t. For the second constraint to be violated, the plan $\pi_{\ddot{\imath}}$ for some agent $\ddot{\imath}$ has to traverse an edge $e' \in E$ s.t. $(\mathfrak{w}, \mathfrak{d}') \in \mathfrak{A}_{\mathfrak{W}}(e)$ at time t for some $\mathfrak{d}' \neq \mathfrak{d}$. However, since $\mathfrak{d} \neq \mathfrak{d}'$, there is a direction conflict between the agents i and $\ddot{\imath}$. This contradicts the assumption. □

5.3 Qubit Routing Performance with Disjoint Splitting

We run the benchmark described in section 3.3 for CBS for Q-MAPF with disjoint splitting. The results can be found in Tables B.4 and B.5 in the appendix, alongside the previous best CBS results. We ran two versions of disjoint splitting: One with all disjoint constraints, and one where we used disjoint splitting except for direction conflicts, where we used the modified global direction constraints described earlier in this chapter. We plot the results in Figures 5.2 and 5.3. Just like in section 3.5, the results at scale factor 0.1 in Figure 5.3a are hardly visible as they sit on top of the x axis.

Overall, disjoint splitting proved to be an improvement. It had better and more consistent runtimes compared to CBS with just global direction constraints, and was able to route ShorEnc up to a scale factor of 0.2. Also, it managed to outperform A*/OD in QFT_4 up to a scale factor of 0.2, where CBS previously could only compete with A*/OD at scale 0.1.

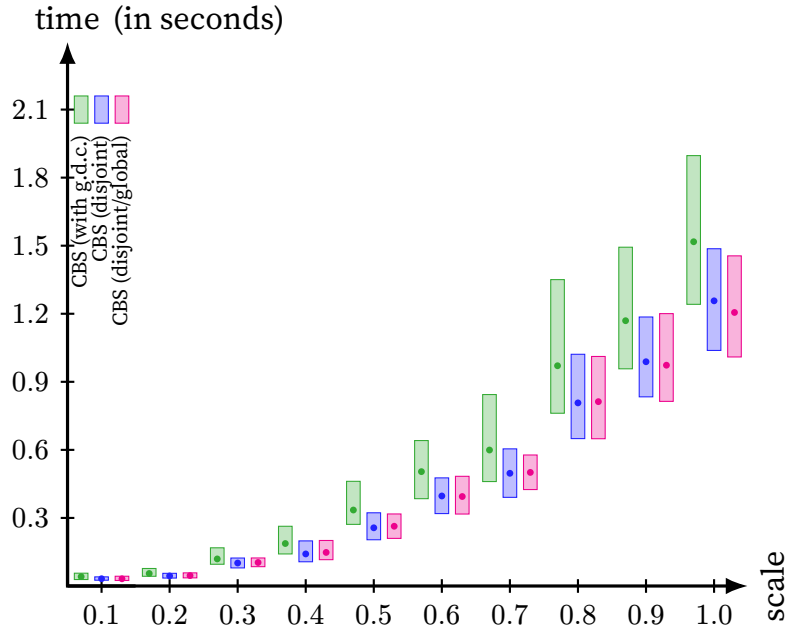
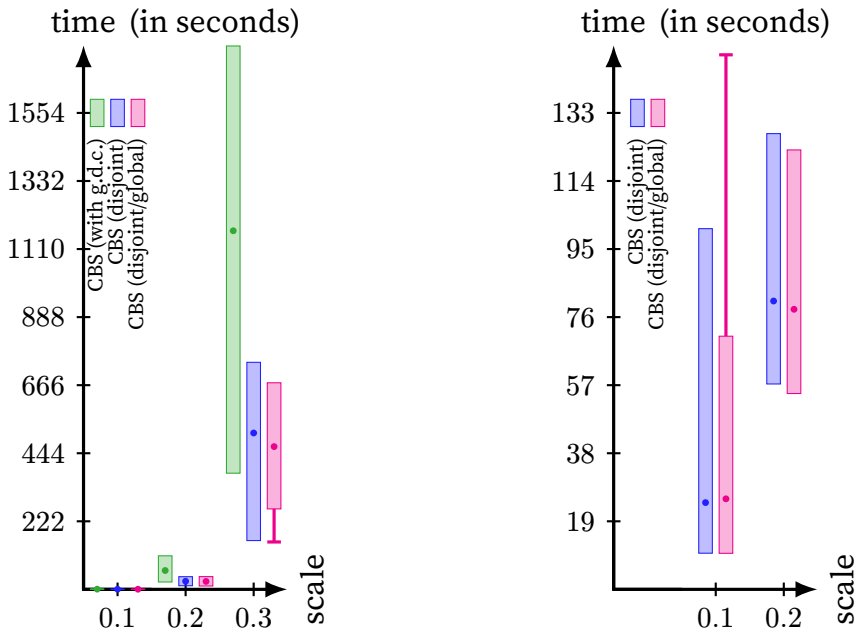


Figure 5.2: Benchmark between CBS with global direction constraints and CBS with disjoint splitting, with and without (modified) global direction constraints (QFT_3)



(a) QFT_4 (b) ShorEnc

Figure 5.3: Benchmark between CBS with global direction constraints and CBS with disjoint splitting, with and without (modified) global direction constraints (QFT_4 and ShorEnc)

The difference between CBS with all disjoint constraints and CBS with modified global direction constraints and otherwise disjoint constraints, however, is hardly visible. If we compare the average runtimes from Table B.4, then there actually is a tie: Each of the two tested algorithms has an ever so slightly better average than the other in exactly 7 cases. For now, we cannot tell which version is better. In the next chapter, we will see that the version with all disjoint constraints has an advantage.

While these are great improvements, they are still not enough. We would like to route all circuits we use for benchmarking at scale 1. And, ideally, also circuits with many more qubits. Using disjoint splitting is thus still not enough to make these algorithms usable for implementing a real-world quantum compiler.

CHAPTER 6

Target Assignment and Path Finding

In chapter 4, we have tried to improve the runtime by exploiting symmetries. This turned out to no longer produce optimal solutions in all cases. Unfortunately, this also ended up not improving runtime. Quite the opposite: As CBS needs to expand all CT nodes with a lower costs before looking at CT nodes with higher costs, the suboptimality has lead to CBS expanding a lot more nodes.

We saw in chapter 5 that we could improve the runtime. But while an improvement, it does not cut the runtime down as much as we want. We need to go further.

The idea is: If we decrease the time it takes for qubits to travel, we hopefully get an improvement in runtime. And there is room for improvement: In section 3.2, we said that when splitting a quantum circuit into layers, we manually pick the qubit destinations. Later, in subsection 3.2.2, we described an algorithm for doing so, which did not consider optimality, and was purely designed to be simple. We used this for the QFT and the SHORENC circuits.

So now, we explore a way to assign optimal qubit destinations for the current layer. This problem is referred to as *combined target assignment and path finding (TAPF)*. We will create our own version suitable for use with Q-MAPF.

To be clear, TAPF still does not guarantee optimal qubit positions for the entire quantum circuit. TAPF runs one layer at a time, so it guarantees that, using the target assignment of the previous layer for the qubit start positions, the target positions will be optimal w.r.t. the costs of MAPF on the current layer. It is possible that a target assignment, which is suboptimal for the current layer, could reduce the costs of the next layer more than the increased cost of the current layer. But, TAPF will turn out to be an improvement over the basic assignment we used before.

TAPF has already been studied outside the context of qubit routing. In vanilla MAPF, however, agents are considered a lot more interchangeable. In [17], they consider k destinations for k agents, which can be grouped, and TAPF should decide which agent goes to which destination. In our case, each target (i.e. manipulation zone of a qubit) can accept each qubit, so we have only one single team. This special case of TAPF is also called *anonymous MAPF*, and can even be solved in polynomial time [35]. But, one important piece of information is missing, which is important to

the application of qubit routing: Two-qubit gates, i.e. agents that need to be routed to two "related" targets.

6.1 Requirements for Q-TAPF

Since our requirements on TAPF differ from the existing TAPF variants, we create our own definition and algorithms for our version of TAPF which we are going to call *Q-TAPF*. Let us start by collecting our requirements.

The broad targets are our unit cells. Each unit cell has three target vertices that could be interesting: The two sides of the manipulation zone, *lm* and *rm*, and the *irz*. The *lm* and *rm* vertices can act as two targets of two unrelated qubits, or together for a pair of qubits that have to execute a two-qubit gate. We ignore the *irz* for now. As long as a quantum circuit does not require a readout inbetween quantum gates, we can just assume that the qubits get routed to an *irz* and read there after the circuit finishes. It is of course possible to account for the *irz*, or different kinds of targets, but this will make the algorithm more complicated.

Our agents are the qubits. However, not all qubits have a target in the current layer. In many layers, only some qubits execute gates. We previously set the destination of qubits which don't need to execute any gates equivalent to their start location. However, this might not be optimal. So for Q-TAPF, while some qubits have targets, others might not have a target. They still need to be considered, and a destination must be assigned so that the next layer has a start location for them.

6.2 Definition of Q-TAPF

Now that we know the requirements, we can define Q-TAPF. For this, we need to change the input of Q-MAPF.

First, we add a set of *destinations* $D \subseteq V$. These destinations are partitioned into *destination groups* $\mathfrak{O} \subseteq 2^D \setminus \{\emptyset\}$. Since the destination groups form a partition, we know that $\biguplus \mathfrak{O} = D$. For SpinBus, the destinations would be the *lm* and *rm* vertices of each unit cell, and the groups would be the set $\{lm, rm\}$ for each unit cell. While not supported by the SpinBus architecture, these groups allow for quantum gates operating on more than two qubits.

Second, we change the destinations of the agents. These are now given as a set $d \subseteq \{1, \dots, k\}$, describing all agents that need a destination, and $\mathfrak{A} \subseteq \{\mathfrak{a} \mid \mathfrak{a} \in 2^d \wedge |\mathfrak{a}| \geqslant 2\}$, describing all agents that are grouped. This means that each agent in a group needs to be assigned a destination s.t. the destinations are also in a group. We do not need to require disjointness here, as two non-disjoint sets $\mathfrak{a}, \mathfrak{a}'$ with $i \in \mathfrak{a}$, $i \in \mathfrak{a}'$ basically behave as if they were one big group $\mathfrak{a} \cup \mathfrak{a}'$. Creating non-disjoint agent groups will however reduce the effectiveness of the disjoint constraints we will develop later, so it should be avoided.

For MAPF, we defined single-agent plans in Definition 2.2 to end at a predetermined destination. We therefore have to give a new definition suitable for use with TAPF.

Definition 6.1. *Let $\pi = (a_1, \dots, a_\ell)$ be a sequence of actions for agent i. If for each $1 \leqslant t \leqslant \ell$, $(a_t \circ \dots \circ a_1)(s(i))$ is defined, and, if $i \in d$, the plan ends at a valid destination $(a_\ell \circ \dots \circ a_1)(s(i)) \in D$, we call π a **single-agent plan** for agent i.*

Additionally, we need to add another type of conflict. This new type of conflict needs to ensure that the conflict-free plans adhere to the new requirement that agents of a group end up at destinations that are part of a group.

Definition 6.2. *Let π_i and $\pi_{i'}$ be two single-agent plans for agents i and i'. If there is an agent group $\mathfrak{a} \in \mathfrak{A}$ that contains both agents, $i \in \mathfrak{a}$ and $i' \in \mathfrak{a}$, and there is no destination group $\mathfrak{o} \in \mathfrak{O}$ that contains both destinations, $d(\pi_i) \in \mathfrak{o}$ and $d(\pi_{i'}) \in \mathfrak{o}$, there is a **target conflict** between the two plans.*

A target conflict thus tells us that at least one of the agents needs to reconsider its choice for the destination. Now we can give a formal definition of Q-TAPF.

Definition 6.3. *Let $G = (V, E)$ be an undirected graph with weights $w : V \cup E \to \mathbb{N}$, control wires $\mathcal{V}_{\mathfrak{W}}$ with a function $\mathfrak{A}_{\mathfrak{W}} : E \to (2^{\{\text{Forward},\text{Idle},\text{Reverse}\} \times \mathcal{V}_{\mathfrak{W}}})$, $\mathbb{N} \oplus \mathbb{N} \to \mathbb{N}$ an objective function, and $D \subseteq V$ a set of destinations partitioned into groups $\mathfrak{O} \subseteq 2^D$. Further, let there be k agents with start locations $s : \{1, \dots, k\} \to V$, of which the agents in $d \subseteq \{1, \dots, k\}$ need destinations and are grouped by $\mathfrak{A} \subseteq 2^d$. We assume wait-at-target. Then, **Q-TAPF** is the optimisation problem: Find a valid joint plan Π (w.r.t. occupancy, direction and target conflicts) s.t. there is no valid joint plan Π' with $w(\Pi') < w(\Pi)$.*

6.3 Solve Q-TAPF using A* with Operator Decomposition

We can adopt the Q-MAPF version of the A*/OD algorithm to work for Q-TAPF. The difference is that instead of having one single target state, we now have several. A state $((v_1, \dots, v_k), \{1, \dots, k\}, \varrho_\perp)$ is a target state if

1. all agents that need a destination have one: $\forall i \in d : v_i \in D$
2. all agents that are grouped have destinations that are part of a group:

$$\forall \mathfrak{a} \in \mathfrak{A} : \exists \mathfrak{o} \in \mathfrak{O} : \forall i \in \mathfrak{a} : v_i \in \mathfrak{o}$$

We also need to update the heuristic. Since SpinBus supports only single and two-qubit gates, we assume that each agent group $\mathfrak{a} \in \mathfrak{A}$ has size 2. However, this approach could easily be extended to larger agent groups when using this algorithm for a different quantum architecture.

As before, the heuristic is a k-tuple. For qubits that don't need a destination, the heuristic returns 0. When a qubit needs a destination, but is not part of a group, we use a single-agent heuristic. Any heuristic suitable for vanilla A* will do, so long it takes into consideration that there are several possible destinations for the agent. If we have a qubit group $(i, \ddot{\imath}) \in \mathfrak{A}$, it would be possible to ignore the grouping and use independent single-agent heuristics. But, we can provide a better heuristic which directly improves the performance of A*. For that, we find a destination group where the costs combined with the objective function for either agent's current position to either destination in the group are minimal. These costs are of course computed by a heuristic, as the actual costs might depend on interactions between the two agents and/or other agents along the way. The minimal heuristic cost was thus computed as some $h_i \oplus h_{\ddot{\imath}}$. We use these two values, h_i and $h_{\ddot{\imath}}$, as the heuristic costs for either agent.

6.4 Solve Q-TAPF using CBS

The CBS algorithm can also be modified to solve Q-TAPF. We first adjust the low-level A* algorithm. Like in the multi-agent OD version of A*, a state $v \in V$ is a target for an agent i if either $i \notin d$ or $v \in D$. The heuristic returns 0 if $i \notin d$, or the minimal heuristic costs to any target that is not restricted. We discuss the constraints that restrict targets next.

6.4.1 Target Constraints

With the updated low level, we can move on to the high-level of CBS. All mutually disjunctive constraints that we introduced for CBS for Q-MAPF remain valid, and we continue to use them. This is true not only for the negative constraints, but also their positive counterparts. However, to solve Q-TAPF, we need to solve the new target conflicts.

We can resolve target conflicts similar to occupancy conflicts: One of the agents is prohibited from occupying a vertex from its current destination group at the end of its plan.

Definition 6.4. *A **target constraint** is a tuple $\langle i, \mathfrak{o} \rangle$ that restricts agent i from occupying any destination vertex $v \in \mathfrak{o}$ at the end of its plan.*

As we can see from the definition, target constraints are negative constraints. We can use these to guarantees optimality, which we prove using mutual disjuctiveness. Unfortunately, they are not disjoint.

Lemma 6.5. *Let i and $\ddot{\imath}$ be two agents with plans π_i and $\pi_{\ddot{\imath}}$ with a target conflict between these plans. Further, let $v_i = d(\pi_i)$ and $v_{\ddot{\imath}} = d(\pi_{\ddot{\imath}})$ be the destination vertices, and $\mathfrak{o}_i, \mathfrak{o}_{\ddot{\imath}} \in \mathfrak{O}$ with $v_i \in \mathfrak{o}_i$, $v_{\ddot{\imath}} \in \mathfrak{o}_{\ddot{\imath}}$ the destination groups of the agents. Then, the target constraints $\langle i, \mathfrak{o}_i \rangle$ and $\langle \ddot{\imath}, \mathfrak{o}_{\ddot{\imath}} \rangle$ are mutually disjuctive.*

Proof. Assume that there are conflict-free plans for i and $\ddot{\imath}$ that violate both constraints. That means that the plan for i ends at a destination $v_i' \in \mathfrak{o}_i$, and the plan for $\ddot{\imath}$ ends at a destination $v_{\ddot{\imath}}' \in \mathfrak{o}_{\ddot{\imath}}$. Since there was a target conflict between the initial plans π_i and $\pi_{\ddot{\imath}}$, the agents must be in a group, and the two destination groups must have been different, i.e. $\mathfrak{o}_i \neq \mathfrak{o}_{\ddot{\imath}}$. Also, the destination groups form a partition, which requires them to be disjoint. Thus, $v_i' \notin \mathfrak{o}_{\ddot{\imath}}$ and $v_{\ddot{\imath}}' \notin \mathfrak{o}_i$. But then there still is a target conflict between the plans, so the plans were not conflict-free, contradicting the assumption. □

6.4.2 Disjoint Target Constraints

We can create a disjoint version of these constraints, however, we cannot just follow the idea behind disjoint splitting for vertex conflicts from [12]. The problem with that is quite simple: Say two agents, i and $\ddot{\imath}$, are part of a group and route to two destinations, v_i and $v_{\ddot{\imath}}$, that are not part of a destination group. Say we forced agent i to stay at that vertex (or at a vertex in the same destination group), and found some restriction for the other agent. Then, the other agent may now find a different destination group, which doesn't resolve the conflict. Thus, we need both agents at the same group.

So instead, the positive constraint will force all agents of a group to go to destinations of the same group.

Definition 6.6. *A **positive target group constraint** is a tuple $\langle\!\langle \mathfrak{a}, \mathfrak{o} \rangle\!\rangle$ that forces all agents $i \in \mathfrak{a}$ to occupy a destination vertex $v \in \mathfrak{o}$ at the end of their plans.*

There is no reason to add a restriction for other agents to this positive constraint: All agents involved in the target conflict are already covered by the positive constraint. Other agents, assuming there are more vertices in $\mathfrak{o}$ than there are agents in $\mathfrak{a}$, can use vertices of the same destination group as their target without causing any conflicts.

We could pair this positive constraint with the negative constraint from Definition 6.4. This would be disjoint: There is no plan that assigns only vertices from one destination group to the agents of that group, and at the same time assigns a different destination to one of the agents from that group.

However, we can do even better: There is no conflict-free solution that will assign a vertex $v \in \mathfrak{o}$ as a destination to an agent $\ddot{\imath} \in \mathfrak{a}$ while an agent $i \in \mathfrak{a}$ is restricted from using all vertices in $\mathfrak{o}$ as its destination. So, the negative counterpart can just restrict all agents in $\mathfrak{a}$ from using all vertices in $\mathfrak{o}$ as their destination.

Definition 6.7. *A **target group constraint** is a tuple $\langle \mathfrak{a}, \mathfrak{o} \rangle$ that restricts all agents $i \in \mathfrak{a}$ from occupying any destination vertex $v \in \mathfrak{o}$ at the end of their plans.*

Pairing these two constraints yields mutually disjunctive and disjoint constraints.

Lemma 6.8. *For a group of agents $\mathfrak{a} \in \mathfrak{A}$ and a group of destinations $\mathfrak{o} \in \mathfrak{O}$, the constraints $\langle\!\langle \mathfrak{a}, \mathfrak{o} \rangle\!\rangle$ and $\langle \mathfrak{a}, \mathfrak{o} \rangle$ are mutually disjuctive.*

Proof. Assume this wasn't true. Then there are conflict-free plans violating both constraints. In order to violate the first (positive) constraint, an agent $i \in \mathfrak{a}$ must occupy a destination vertex $v \notin \mathfrak{o}$ at the end of their plan π_i. In order to violate the second (negative) constraint, an agent $i' \in \mathfrak{a}$ must occupy a destination vertex $v' \in \mathfrak{o}$ at the end of their plan $\pi_{i'}$. Since all destination groups are disjoint, there cannot be another destination group $\mathfrak{o}' \neq \mathfrak{o} \in \mathfrak{O}$ s.t. $v' \in \mathfrak{o}'$. But, $i, i' \in \mathfrak{a} \in \mathfrak{A}$, and π_i and $\pi_{i'}$ are conflict-free, so there must be a destination group containing both v and v' so that there is no target conflict, which is clearly not the case, contradicting the assumption. □

Lemma 6.9. *For a group of agents $\mathfrak{a} \in \mathfrak{A}$ and a group of destinations $\mathfrak{o} \in \mathfrak{O}$, the constraints $\langle\!\langle \mathfrak{a}, \mathfrak{o} \rangle\!\rangle$ and $\langle \mathfrak{a}, \mathfrak{o} \rangle$ are disjoint.*

Proof. Let $\Pi = (\pi_1, \ldots, \pi_k)$ be a (not necessarily conflict-free) joint plan. Pick an agent $i \in \mathfrak{a}$ with plan π_i. If $d(\pi_i) \in \mathfrak{o}$, the second (negative) constraint is violated. Otherwise, the first (positive) constraint is violated. Therefore, the plan is permitted by at most one of the two constraints. □

6.5 Qubit Routing Performance for Q-TAPF

We use the QFT_3, QFT_4 and ShorEnc quantum circuits again to benchmark our qubit routing performance, this time using Q-TAPF. Most notably, this means that the fixed qubit destinations we used previously no longer apply, and instead, the destina-

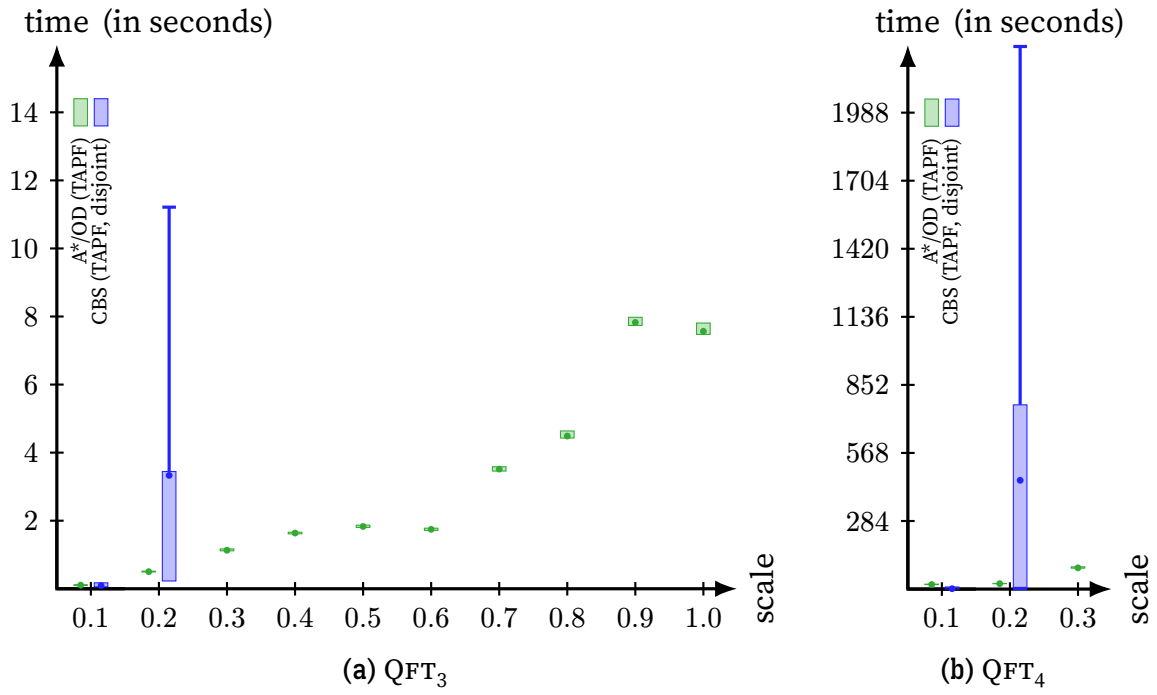


Figure 6.1: Benchmark for TAPF between A*/OD and CBS with disjoint splitting (QFT_3 and QFT_4)

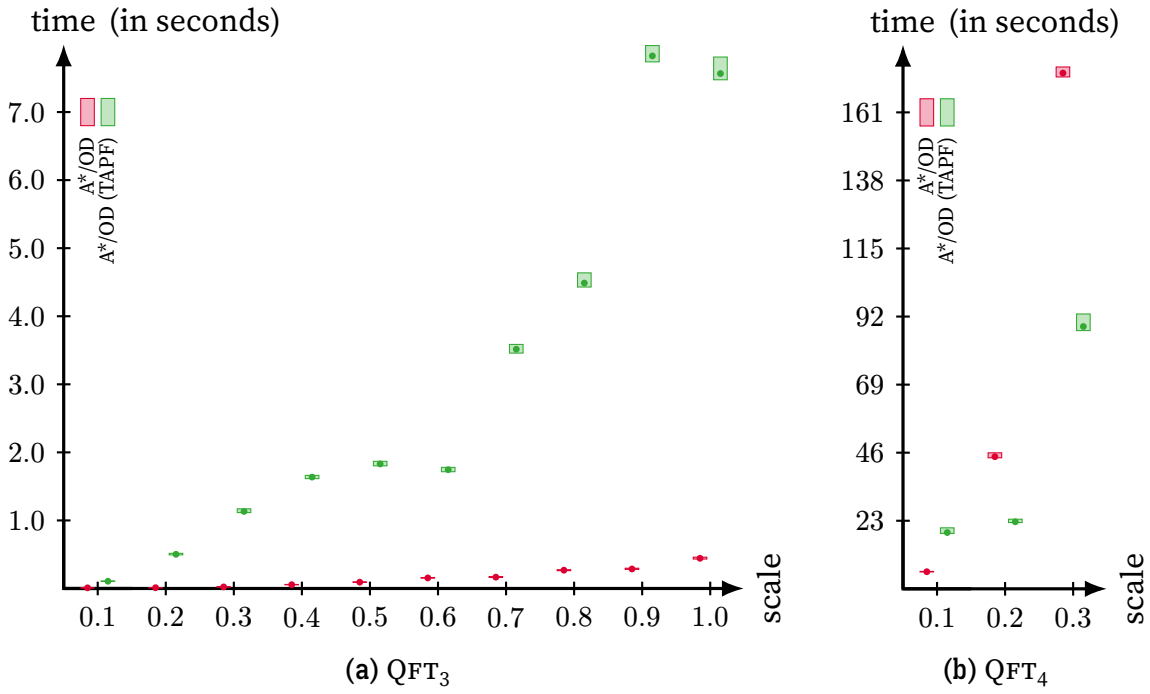
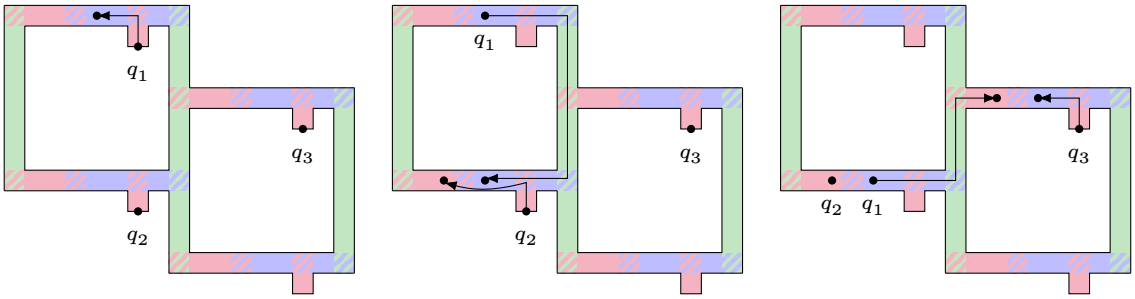


Figure 6.2: Benchmark between A*/OD for MAPF and for TAPF (QFT_3 and QFT_4)

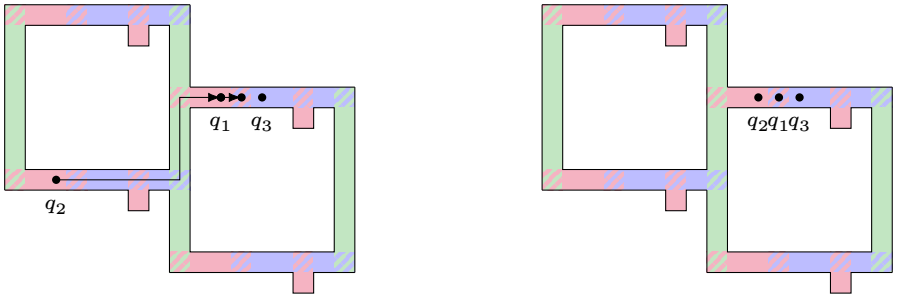
tions assigned by the previous layer get used for the next layer. We benchmark A*/OD vs CBS for Q-TAPF with global direction constraints and using disjoint splitting, both with and without global direction constraints. The results can be found in Tables B.6 and B.7 in the appendix. Additionally, we plot the performance of A*/OD and the best of the CBS variants, disjoint splitting without global direction constraints, in Figure 6.1.

CBS has performed poorly and inconsistently throughout the benchmark: While it was able to solve ShorEnc at scale 0.1 in just over 1.5 seconds in one benchmark run, in another run in ran out of memory. For this reason, we continued the benchmark at a higher scale eventhough it had already failed before. At scale 0.2, we got a result in just over 30 seconds. Both of these would be an improvement over the MAPF results in Table B.4 if they were consistent. At 0.3, we were not able to complete the benchmark. We will take a closer look at the CBS results in subsection 6.5.1.

A*/OD however has improved. While the time it took to route QFT_3 has increased from Q-MAPF to Q-TAPF, which can be seen in Figure 6.2a, this is expected. The destinations we manually assigned to the qubits for QFT_3 were almost optimal, so the additional time it took the algorithm to find the optimal targets is noticable in the benchmark. When we analyse the solution later (Table 6.5), we will see that only the last two layers have reduced their costs. However, when we look at QFT_4 in Figure 6.2b, we clearly see an improvement. Here, the destinations we manually assigned that Q-MAPF used were less optimal. And while A*/OD unfortunately still couldn't solve the problem at scale 0.4 and above, the time it took to complete the scale factors 0.2 and 0.3 has improved significantly.

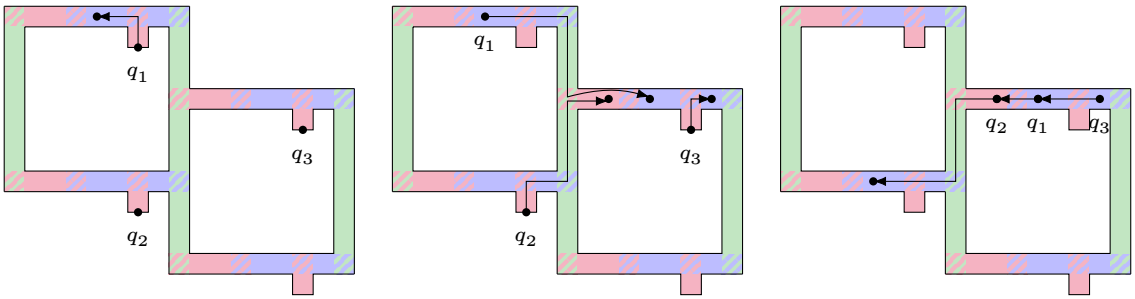


(a) 1st layer: Compute $H\ q_1$ (b) 2nd layer: Compute $R_2\ q_1, q_2$ (c) 3rd layer: Compute $R_3\ q_1, q_3$ and $H\ q_2$



(d) 4th layer: Compute $R_2\ q_2, q_3$ (e) 5th layer: Compute $H\ q_3$

Figure 6.3: The layers of QFT_3 in the faster runs of CBS for Q-TAPF



(a) 1st layer: Compute $H\ q_1$ (b) 2nd layer: Compute $R_2\ q_1, q_2$ (c) 3rd layer: Compute $R_3\ q_1, q_3$ and $H\ q_2$

Figure 6.4: The first layers of QFT_3 in the slower runs of CBS for Q-TAPF

6.5.1 Analysis of the inconsistent performance of CBS

Let us take a closer look at the results of CBS for Q-TAPF. For QFT_3 at scale 0.2, we have 46 out of 50 runs finish in under 1.5 seconds, and then 4 runs taking 34 seconds or more. This large "gap" in the performance is visible in Figure 6.1, and is also present in QFT_4. While CBS previously had worse consistency than A*/OD, we never observed such large gaps. We explained the previous variance with the randomised order of hash maps used within CBS, but there is clearly something else at play here.

The strategy that CBS uses in the faster runs for QFT_3, shown in Figure 6.3, is quite similar to what we manually planned in Figure 3.8. Besides the fact that qubits are not forced to return to their respective *irz*, CBS for Q-TAPF decided here to switch the

positions of the qubits in the 2nd layer. It is also the 2nd layer that takes the majority of time here.

However, when we visualise one of the slower runs in Figure 6.4, we see they are quite different for the 2nd and 3rd layer. The last two layers are equivalent to the ones in Figure 6.3, so we omitted them. While some time was still spent on the 2nd layer, the majority of time was spent on layer 3. Notably, the 2nd layer found a different result this time. Both results have the same costs of 16, so from the perspective of the routing algorithm working on just that layer, both solutions are equally good. But it turns out that the next layer, which receives the positions of the qubits from the previous layer, now becomes harder.

When we look at Figure 6.4c, the problem doesn't seem to hard at first. However, CBS doesn't start out with the knowledge that q_2 will go to a different unit cell. It first tries a lot of possible wire assignments to allow q_1 to stay at its location and q_3 to reach that same location, as both are viable targets. Obviously, both qubits cannot occupy the same location, but the occupancy conflict is never expanded as there are earlier direction conflicts. Then, CBS tries to move q_1 to the location of q_2 instead of letting it stay at its current location. Only after exhausting all possible combinations there, it finally moves q_2 to a different unit cell.

6.5.2 Analysis of the solutions of Q-TAPF

As we discussed before, the solutions that Q-TAPF can find differ in their costs from their MAPF counterpart, and can also be different between subsequent runs of Q-TAPF. If we look at A*/OD, at the maximum scale for each circuit at which A*/OD finds a solution, we find an improvement. The data is displayed in Table 6.5. At least for this test, the results were consistent.

We do the same again for CBS at scale 0.2 for the QFT circuits and 0.1 for the SHORENC circuit. The results may be found in Table 6.6. This time, the results were not consistent: CBS can find different paths, thus, there are different total costs. In the previous subsection, we analysed CBS for QFT_3 and found two different paths, one being significantly slower than the other. The faster one is the one with total costs of 43, while the slower one comes to a total of just 39. So here, the slower one

			Cost per Layer							
Circuit	Scale	Algo	1	2	3	4	5	6	7	Total
QFT_3	1.0	Q-MAPF	17	62	51	54	53			237
		Q-TAPF	17	62	51	46	0			176
QFT_4	0.3	Q-MAPF	5	24	20	36	37	37	34	193
		Q-TAPF	5	24	26	24	15	19	0	113

Table 6.5: Comparison of the solution cost of A*/OD between Q-MAPF and Q-TAPF

			Cost per Layer							
Circuit	Scale	Algo	1	2	3	4	5	6	7	Total
QFT_3	0.2	Q-MAPF	3	18	15	12	11			59
		Q-TAPF	3	16	10	14	0			43
			3	16	10	10	0			39
QFT_4	0.2	Q-MAPF	3	18	14	25	25	25	21	131
		Q-TAPF	3	16	17	14	16	15	0	81
			3	16	17	14	16	10	0	76
			3	16	16	17	10	14	0	76
			3	16	16	16	14	10	0	75
			3	16	16	16	11	10	0	72
			3	16	16	17	10	10	0	72
ShorEnc	0.1	Q-MAPF	3	8	15	0	14	15		55
		Q-TAPF	3	8	13	4	13	13		54
			3	8	13	4	13	12		53
			3	8	13	3	13	13		53
			3	8	13	3	13	12		52

Table 6.6: Comparison of the solution cost of CBS between Q-MAPF and Q-TAPF

produced the better solution. However, as said before, this is pure chance since there are different optimal solutions for the 3rd layer, which cause different costs for the remaining layers due to different start locations. For QFT_4, we found 6 different costs per layer, with four different total costs. We only found 4 different costs for ShorEnc, but since some of the runs do not finish, there is a good chance that there are more versions if we had more memory.

Regardless of which solution we pick from either table, we see an improvement in the cost of the solution when comparing Q-MAPF to Q-TAPF. It ranges from a factor of 1.3 up to an improvement by a factor of 1.8. The improvements for ShorEnc were rather small, but this could be due to the low scale. However, evaluating ShorEnc at higher scales requires too much time and memory, so we could not run it.

CHAPTER 7

Conclusion

We started this thesis by giving a definition of Multi-Agent Path Finding (MAPF) in chapter 2. MAPF comes in many variations: You can choose different conflicts, different objective functions, and different behaviour of agents once they reach their destination. We chose specific conflicts, vertex and edge conflicts (the latter are also sometimes referred to as swapping conflicts, e.g. in [31]), while we left the others generic: For the objective function, we considered both makespan and sum-of-cost, while for the behaviour at an agents destination, we considered three options: The first two, disappear- and stay-at-target, are both commonly found in literature. The last one, which we called wait-at-target, is the one relevant for SpinBus, but rarely found in other literature. In [8], this behaviour was called stay-at-target, but we renamed it to avoid confusion with other literature.

To solve MAPF, we looked at two algorithms, A* (with Operator Decomposition) and Conflict-Based Search (CBS). We proved that A* in its most basic form, which is also called the standard admissible algorithms by some literature [29], is not suitable to solve MAPF optimally when assuming disappear- or stay-at-target. We then formalised A* with Operator Decomposition (A*/OD), based on [29], for the wait-at-target assumption. We also showed that CBS, which is known to solve MAPF optimally under the disappear- and stay-at-target assumptions, may produce suboptimal results when assuming wait-at-target. Since the qubit routing problem for SpinBus assumes wait-at-target, we introduced timing constraints. These allow CBS to produce optimal results for MAPF under the wait-at-target assumption.

In chapter 3, we described the specifics of the SpinBus quantum architecture that are relevant to qubit routing. We summarised the prior work for qubit routing on SpinBus, especially the Q-MAPF extension of MAPF from [8]. To solve Q-MAPF, we adopted both algorithms we described for MAPF, A*/OD and CBS. We then defined three quantum circuits that we want to apply to the routing algorithms: QFT_3 and QFT_4, circuits implementing quantum fourier transform which is a part of Shor's algorithm [24], and ShorEnc, the encoding circuit for Shor's quantum error correction code [25]. Using these three circuits, we evaluated the performance of A*/OD and CBS. For that, we created a scale function to reduce the size of the graph representing the SpinBus device. We found A*/OD to perform better than CBS, but overall,

both algorithms were too inefficient, regarding both their execution time and their memory consumption. The QFT_3 circuit was the only one that either algorithm could compute at full scale, and neither algorithm could compute the ShorEnc circuit at any of the tested scale factors.

The first improvement we looked at was a variant of direction constraints called global direction constraints that we developed in section 3.4. This, while only applicable to CBS, was a slight improvement, but not sufficient: It still could not compute any tested scale factor of the ShorEnc circuit. However, this showed to us that improving the handling of direction conflicts in CBS is a direction worth exploring.

Continuing in this direction, we explored corridor symmetries in chapter 4. These showed improvements in vanilla CBS [15]. Unfortunately, we were not able to keep the optimality of CBS for Q-MAPF when using corridor symmetry. Despite that, we developed and tested a CBS version using conveyor conflicts and range constraints that explore corridor symmetry. The test showed that this version performed worse than all previous versions. Combined with the fact that the results were now also potentially suboptimal, we must conclude that this attempt did not yield any desirable results.

Instead, we explored another idea that improved vanilla CBS, called disjoint splitting [12], in chapter 5. We adopted it for Q-MAPF, implemented and tested it on the three quantum circuits. We saw improvements over both A*/OD and the previous best version of CBS: The execution time was lower across all three circuits compared to CBS, and this was the first version to solve the ShorEnc circuit, eventhough not at full scale.

In the last chapter, we explored optimal target assignment. For all previous tests, we had manually assigned qubit destinations for the quantum circuits, based on a simple algorithm. This was not the most optimal target assignment. Since our circuits were quite small with no more than 9 qubits, we could have manually improved the target assignment. However, as SpinBus is planning to support more than a hundred qubits in the future, manually assigning destinations is cumbersome and unlikely to be optimal. Target-Assignment and Path Finding (TAPF) is an extension of MAPF that takes this into consideration. We extended TAPF into Q-TAPF and developed versions of A*/OD and CBS that could solve it. Testing these showed inconsistent results, especially for the ShorEnc circuit: Sometimes, the test was much faster than its MAPF counterpart, while other times, it failed due to insufficient memory. These inconsistencies were caused by the existence of different optimal target assignments: While the qubit routing solution had the same cost for both target assignments, the execution time and memory consumption necessary to produce that solution varied widely. The cost of the solution however improved, sometimes slightly and sometimes by a factor of 1.8, compared to Q-MAPF.

CHAPTER 8

Outlook

The results show that there is room for optimisation. By extending optimisations known to work for CBS, we were able to improve the runtime for CBS for Q-MAPF. Since CBS is a well studied algorithm, there are more optimisations that could potentially work for Q-MAPF than what we could cover in this thesis.

One of these CBS implementations is known as Improved Conflict-Based Search (ICBS) [2]. They propose three improvements that together form ICBS: The first one proposes to merge agents into *meta agents*, which can be solved by using a MAPF solver such as A*/OD as the low-level. The second improvement changes the order in which conflicts are expanded, based on whether they are *cardinal*. In this thesis, we have always chosen the first conflict w.r.t. the timestep at which it occurs for expansion. However, we can also classify a conflict as *cardinal* if both constraint cause the cost of the solution to increase, and prioritise these conflicts. We have briefly experimented with this in the code base used for benchmarking, found it to have a slightly negative impact on performance, and not looked into it further. The last improvement proposes to not split a CT node if one of the agents could choose a different path with the same cost.

Building on top of ICBS, [4] added a heuristic to the CBS algorithm. Similar to how the heuristic for A* works, this heurisic is used to estimate the cost of a CT node. This promises to expand CT nodes in a better order, potentially reaching a solution with less CT nodes expanded. They showed that this approach could reduce the runtime. This was again improved in [13], where they propose two new heuristics. In future work, one could extend this heuristic approach to CBS for Q-MAPF and see if it improves its performance.

The heuristic approach could also be interesting when combined with domain-specific knowledge about conveyors and corridors in SpinBus. This was already mentioned briefly in section 4.5. One could build a heuristic that prioritises those constraints that are unlikely to cause further direction constraints on the same conveyor. Or, one could give a better estimation for the costs of the CT nodes, similar to how A* searches a graph: If we know that a conveyor conflict will take at least an additional ℓ timesteps to lead to a conflict-free solution, we can add these heuristic costs to the CT node. There is a lot of room for future work to explore this approach.

Another approach for future work could be to relax the optimality in favour of performance. While we tried this to some extend in chapter 4 without success, there are ideas for vanilla CBS one could explore and potentially adopt for its Q-MAPF variant. Enhanced CBS (ECBS) [1] provides a bounded sub-optimal algorithm that can outperform other variations of CBS. This was improved further into an algorithm called Flexible ECBS (FECBS) [3]. FECBS relaxes the bounded-suboptimality requirement of ECBS from each agent's plan to the joint plan. A version of qubit routing for SpinBus based on these bounded sub-optimal algorithms could be developed in a future work.

One could also use algorithms other than A* and CBS to solve the qubit routing problem for SpinBus. At least for MAPF, there are more algorithms than just these two. A list of commonly used algorithms is given in [30]. Besides CBS and variations of A* like Operator Decomposition and M* [34], it includes Increasing Cost Tree Search (ICTS) [22] and Constraint Programming (CP), which can be combined with CBS into an algorithm called Lazy CBS [5]. These algorithms, and others not mentioned here, could be evaluated if they can be extended to solve Q-MAPF, and tested for their performance.

Finally, one could go down the path of improving the solution, not focusing on the execution time. We saw when we analysed Q-TAPF that we improved the cost of the solution when adding up the costs of all the different layers. There were different solutions though, as we only optimised one layer at a time, for which multiple solutions with minimal cost existed. A better result could be obtained when one combined the problem in such a way that it considered not only the current but also the following layers.

Bibliography

[1] Max Barer et al. 'Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem'. In: *Proceedings of the International Symposium on Combinatorial Search* 5.1 (2014), pp. 19–27. DOI: 10.1609/socs.v5i1.18315. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18315.

[2] Eli Boyarski et al. 'ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding'. In: *Proceedings of the Eighth International Symposium on Combinatorial Search* (2015). DOI: 10.1609/socs.v6i1.18343. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18343.

[3] Shao-Hung Chan et al. 'ECBS with Flex Distribution for Bounded-Suboptimal Multi-Agent Path Finding'. In: *Proceedings of the International Symposium on Combinatorial Search* 12.1 (2021), pp. 159–161. DOI: 10.1609/socs.v12i1.18569. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18569.

[4] Ariel Felner et al. 'Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 28.1 (June 2018), pp. 83–87. DOI: 10.1609/icaps.v28i1.13883. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/13883.

[5] Graeme Gange, Daniel Harabor and Peter J. Stuckey. 'Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 29.1 (July 2019), pp. 155–162. DOI: 10.1609/icaps.v29i1.3471. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/3471.

[6] Peter E. Hart, Nils J. Nilsson and Bertram Raphael. 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136. URL: https://ieeexplore.ieee.org/document/4082128.

[7] Jannik Hellenkamp and Dominique Unruh. 'Shor's Algorithm'. In: *Introduction to Quantum Computing - Lecture notes for the summer term 2024*. 2024. Chap. 9. URL: https://qis.rwth-aachen.de/teaching/24ss/intro-quantum-computing/script/shorsAlgorithm.html (visited on 28th May 2025).

[8] Roy Hermanns. 'Optimized Routing for Shuttling-Based Quantum Processing Architectures'. MA thesis. RWTH Aachen University, 2024.

[9] Kyungbae Jang et al. 'New Quantum Cryptanalysis of Binary Elliptic Curves'. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025.2 (Mar. 2025), pp. 781–804. DOI: 10.46586/tches.v2025.i2.781-804. URL: https://tches.iacr.org/index.php/TCHES/article/view/12065.

[10] Matthias Künne et al. 'The SpinBus architecture for scaling spin qubits with electron shuttling'. In: *Nature Communications* 15 (2024). DOI: 10.1038/s41467-024-49182-4.

[11] Tim Tsz-Kit Lau and Biswa Sengupta. *The Multi-Agent Pickup and Delivery Problem: MAPF, MARL and Its Warehouse Applications*. 2022. DOI: 10.48550/arXiv.2203.07092. URL: https://arxiv.org/abs/2203.07092.

[12] Jiaoyang Li et al. 'Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 29.1 (July 2019), pp. 279–283. DOI: 10.1609/icaps.v29i1.3487. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/3487.

[13] Jiaoyang Li et al. 'Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search'. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 442–449. DOI: 10.24963/ijcai.2019/63. URL: https://www.ijcai.org/proceedings/2019/63.

[14] Jiaoyang Li et al. 'Multi-Agent Path Finding for Large Agents'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 7627–7634. DOI: 10.1609/aaai.v33i01.33017627. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4756.

[15] Jiaoyang Li et al. 'Pairwise symmetry reasoning for multi-agent path finding search'. In: *Artificial Intelligence* 301 (2021), p. 103574. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103574. URL: https://www.sciencedirect.com/science/article/pii/S0004370221001259.

[16] Ruoyu Li et al. 'A crossbar network for silicon quantum dot qubits'. In: *Science Advances* 4.7 (2018). DOI: 10.1126/sciadv.aar3960. URL: https://www.science.org/doi/abs/10.1126/sciadv.aar3960.

[17] Hang Ma and Sven Koenig. *Optimal Target Assignment and Path Finding for Teams of Agents*. 2016. DOI: 10.48550/arXiv.1612.05693. URL: https://arxiv.org/abs/1612.05693.

[18] Robert Morris et al. 'Planning, Scheduling and Monitoring for Airport Surface Operations.' In: *AAAI Workshop: Planning for Hybrid Systems*. 2016, pp. 608–614. URL: https://cdn.aaai.org/ocs/ws/ws0196/12611-57497-1-PB.pdf.

[19] Nikiforos Paraskevopoulos, Carmen G. Almudever and Sebastian Feld. 'BeSnake: A Routing Algorithm for Scalable Spin-Qubit Architectures'. In: *IEEE Transactions on Quantum Engineering* 5 (2024), pp. 1–22. DOI: 10.1109/TQE.2024.3429451. URL: https://ieeexplore.ieee.org/abstract/document/10601342.

[20] Nikiforos Paraskevopoulos et al. 'SpinQ: Compilation Strategies for Scalable Spin-Qubit Architectures'. In: *ACM Transactions on Quantum Computing* 5.1 (2023). DOI: 10.1145/3624484. URL: https://dl.acm.org/doi/full/10.1145/3624484.

[21] Martin Roetteler et al. *Quantum resource estimates for computing elliptic curve discrete logarithms*. 2017. DOI: 10.48550/arXiv.1706.06752. URL: https://arxiv.org/abs/1706.06752.

[22] Guni Sharon et al. 'The increasing cost tree search for optimal multi-agent pathfinding'. In: *Artificial Intelligence* 195 (2013), pp. 470–495. ISSN: 0004-3702. DOI: 10.1016/j.artint.2012.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0004370212001543.

[23] Guni Sharon et al. 'Conflict-based search for optimal multi-agent pathfinding'. In: *Artificial Intelligence* 219 (2015), pp. 40–66. ISSN: 0004-3702. DOI: 10.1016/j.artint.2014.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0004370214001386.

[24] Peter W. Shor. 'Algorithms for quantum computation: discrete logarithms and factoring'. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[25] Peter W. Shor. 'Scheme for reducing decoherence in quantum computer memory'. In: *Physical Review A* 52.4 (Oct. 1995). DOI: 10.1103/PhysRevA.52.R2493. URL: https://journals.aps.org/pra/pdf/10.1103/PhysRevA.52.R2493.

[26] Peter W. Shor. 'Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer'. In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172.

[27] Devon Sigurdson et al. 'Multi-Agent Pathfinding with Real-Time Heuristic Search'. In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. 2018, pp. 1–8. DOI: 10.1109/CIG.2018.8490436.

[28] David Silver. 'Cooperative Pathfinding'. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 1.1 (Sept. 2021), pp. 117–122. DOI: 10.1609/aiide.v1i1.18726. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18726.

[29] Trevor Standley. 'Finding Optimal Solutions to Cooperative Pathfinding Problems'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 24.1 (July 2010), pp. 173–178. DOI: 10.1609/aaai.v24i1.7564. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7564.

[30] Roni Stern. 'Multi-Agent Path Finding – An Overview'. In: *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*. Ed. by Gennady S. Osipov, Aleksandr I. Panov and Konstantin S. Yakovlev. Cham: Springer International Publishing, 2019, pp. 96–115. ISBN: 978-3-030-33274-7. DOI: 10.1007/978-3-030-33274-7_6.

[31] Roni Stern et al. 'Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks'. In: *Symposium on Combinatorial Search (SoCS)* (2019). URL: https://mapf.info/index.php/Main.SKoen19k (visited on 11th June 2025).

[32] Alejandro Morais Tejerina. 'Mapping quantum algorithms in a crossbar architecture'. MA thesis. TU Delft, 2019. URL: https://resolver.tudelft.nl/uuid:cb7194fb-7b99-4ea9-a5ad-8e3617d1f9f5.

[33] Dominique Unruh. 'Lecture 20'. Introduction to Quantum Computing. 2024. URL: https://video.fsmpi.rwth-aachen.de/24ss-qc/17706 (visited on 28th May 2025).

[34] Glenn Wagner and Howie Choset. 'Subdimensional expansion for multirobot path planning'. In: *Artificial Intelligence* 219 (2015), pp. 1–24. ISSN: 0004-3702. DOI: 10.1016/j.artint.2014.11.001. URL: https://www.sciencedirect.com/science/article/pii/S0004370214001271.

[35] Jingjin Yu and Steven M. LaValle. *Multi-agent Path Planning and Network Flow*. Jan. 2013. DOI: 10.48550/arXiv.1204.5717. URL: https://arxiv.org/abs/1204.5717.

[36] Jingjin Yu and Steven M. LaValle. 'Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 27.1 (June 2013), pp. 1443–1449. DOI: 10.1609/aaai.v27i1.8541. URL: https://ojs.aaai.org/index.php/AAAI/article/view/8541.

List of Figures and Tables

List of Mathematical Symbols

2^M		For a set M, $2^M := \{m \mid m \subseteq M\}$ is the power set of M.
a	Definition 2.1	An action, i.e. a partial function mapping vertices to vertices. A move action maps one vertex to another, while a wait action maps every vertex to itself.
$\mathfrak{A}$	Definition 6.2	The agent groups for Q-TAPF.
$\mathfrak{a}$	Definition 6.2	An element of $\mathfrak{A}$.
$\mathfrak{A}_{\mathfrak{W}}$		A function mapping edges to direction and control wires.
c	Definition 3.2	The center vertex of an occupancy conflict.
c	Figure 3.2	The center junction, a vertex of a unit cell.
C	Definition 4.1	A corridor.
C'	Definition 4.1	A corridor excluding its two exit vertices.
d		In MAPF, a function assigning destinations to agents. In TAPF, a set of agents that need a target destination assigned. Also used as a function to get the destination of a single-agent plan.
$\mathfrak{d}$		Commonly used for the direction of a control wire.
$\mathfrak{D}$	Definition 6.2	The destination groups for Q-TAPF.
$\mathfrak{d}$	Definition 6.2	An element of $\mathfrak{D}$.
degree		A function to get the degree of a vertex.
E		The edge set of a graph G. For undirected graphs, this is symmetrical, i.e. $(u, v) \in E$ iff $(v, u) \in E$.
$E_{\mathcal{G}}$		The edge set of the graph $\mathcal{G}$.

Forward		A direction of a control wire.
$\hat{f}$		Evaluation function for an A* state.
$\hat{g}$		Function to obtain the costs of the best path currently known to A*.
G		A graph, usually undirected.
$\mathcal{G}$	Definition 2.12	The joint state space.
	Definition 2.17	The operator decomposition state space.
h	Figure 3.2	The handover point, a vertex of a unit cell.
$\hat{h}$		Heuristic costs for A*.
i, $ï$		Indices commonly used for agents.
Idle		A direction of a control wire.
irz	Figure 3.2	The initialisation and readout zone, a vertex of a unit cell.
k		The number of agents in a MAPF problem.
ℓ		Commonly used as the length of a plan.
lj	Figure 3.2	The left junction, a vertex of a unit cell.
lm	Figure 3.2	The left manipulation zone, a vertex of a unit cell.
$\mathfrak{md}$		The minimum distance between two agents.
N		A function to get all neighbors, i.e. adjacent vertices, of a vertex.
$\mathbb{N}$		The set of natural numbers / positive integers.
$\mathbb{N}_0$		The set of non-negative integers.
$q_1, q_2, \ldots$		Agents, specifically Qubits, for Q-MAPF.
Reverse		A direction of a control wire.
rj	Figure 3.2	The right junction, a vertex of a unit cell.
rm	Figure 3.2	The right manipulation zone, a vertex of a unit cell.
s		A function assigning start locations to agents in MAPF.
S	Definition 4.8	A function finding a corridor around a direction conflict.

scale	Equation 3.1	A function to scale length of a SpinBus device.
t		A timestep, a non-negative number. Commonly used as the index of a plan and in conflicts and constraints.
u, v		Commonly used for vertices
V		The vertex set of a graph G.
$V_{\mathcal{G}}$		The vertex set of the graph $\mathcal{G}$.
$\mathcal{V}_{\mathfrak{W}}$		A set of control wires.
w		The weight function for a graph. Usually defined for both vertices and edges.
	Definition 2.8	The weight function for a single-agent plan π.
	Definition 2.9	The weight function for a joint plan Π.
$w_{\mathcal{G}}$		The weight function for the graph $\mathcal{G}$.
$\mathfrak{w}$		Commonly used for a wire.
π	Definition 2.2 Definition 6.1	A single-agent plan, i.e. a tuple of actions.
Π	Definition 2.6	A joint plan, i.e. a tuple of single-agent plans.
$\oplus$	Definition 2.9	The objective function, which combines the single-agent costs into the costs for the joint plan.
🕱	Definition 2.3	Special vertex, indicating that an agent has "disappeared".
$\bar{\mathfrak{O}}$	Definition 4.4	A function returning the earliest time a certain agent can reach a certain vertex.
$\bar{\mathfrak{O}}$	Definition 4.4	A function returning the earliest time a certain agent can reach a certain vertex without going through a corridor.

APPENDIX A

CBS Step by Step for layer 3 of QFT_3

This appendix contains a step by step run of CBS on the 3rd layer of QFT_3 at a scale factor of 0.5. In the first section, we show a run on a default CBS for Q-MAPF, while in the second section there is a run using conveyor conflicts as introduced in section 4.2.

The layer involves three qubits on a 2×2 SpinBus device. We can see the full layer in Figure 3.8c, and the relevant two unit cells in more detail in Figure A.1. An optimal solution can be found in Figure A.2x and has costs of 29.

While solving CBS manually, we choose to ignore timing conflicts. Instead, we immediately extend the length of the shorter paths to match the longest. We can do this as we set cost equal to time, which we must do as it is a requirement for conveyor conflicts set forth in Definition 4.3. The minimum distance $\mathfrak{md}$ between agents is set to 2. We prefer the earliest conflict to resolve in each step, and resolve direction conflicts first if there are multiple earliest conflicts.

For the constraint, we manually pick one that does not conflict with the optimal solution (if possible). This usually involves picking the constraint for q_2, except in Figure A.2h and Figure A.3h, where we pick the constraint for q_1. While $t = 5$ isn't the only choice q_1 has for idling as it could also choose 3 or 4, we make this choice to keep the example somewhat simple.

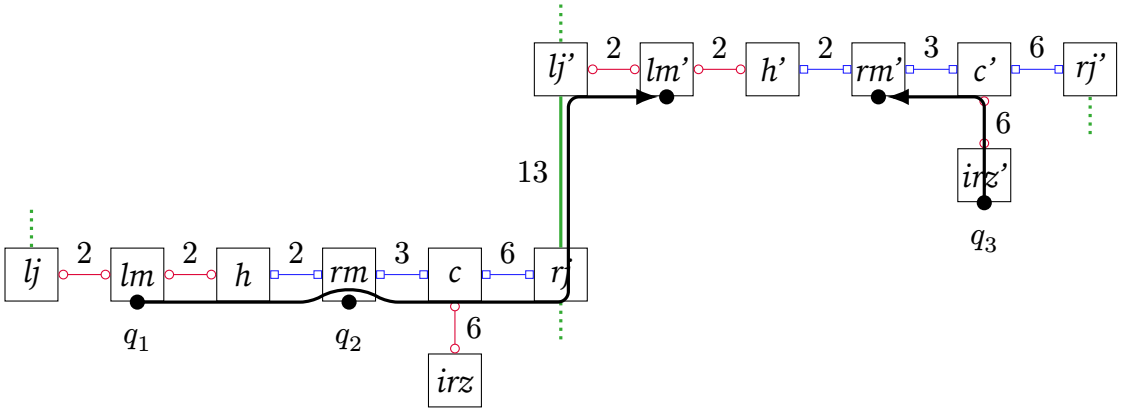


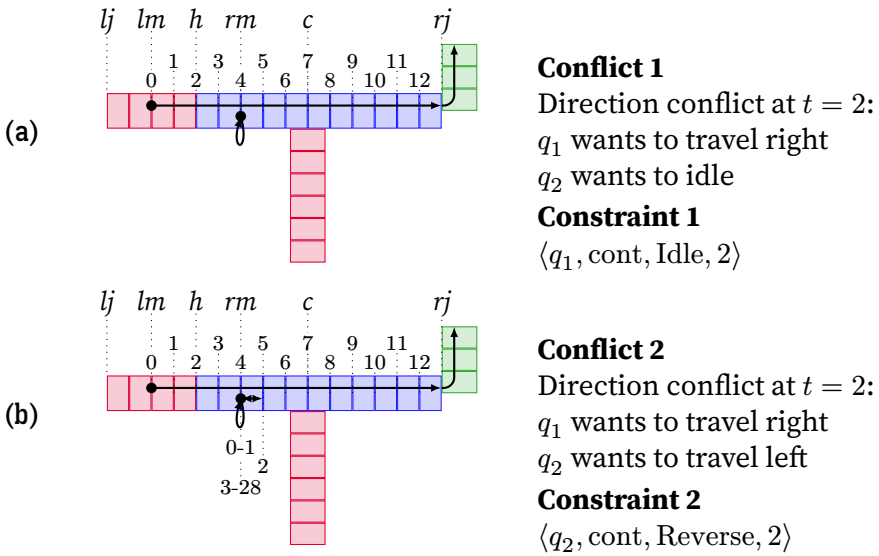
Figure A.1: The two relevant unit cells for the 3rd layer of QFT_3 at scale factor 0.5

The low-level algorithm will have a lot of choices as there usually isn't a single unique shortest path for the qubits. For consistency, we give ourselves some rules. First, we ask the low level to find the fastest way towards its destination, and prefer idling there if they arrive "too early". Second, for q_2, we prefer to move it right over left inside of the unit cell.

We visualise only the unit cell where q_1 and q_2 start, alongside a part of the center bus. There are no conflicts outside of the visualised part so the other parts of the SpinBus device are not relevant to us. Each rectangle represents one edge, i.e. the vertices are on the border of the rectangles. We write the time at which q_1 visits a vertex above and the time at which q_2 visits a vertex below the vertex. It is possible that a qubit visits a vertex multiple times, in which case there will be multiple numbers. Idling qubits are indicated via a loop and a $t_1 - t_2$ label, indicating that a qubit arrives at time t_1 and then idles until time t_2 at the vertex after which it moves away.

The constraints displayed below mention "intermediate" vertices, i.e. those hidden behind a number in Figure A.1. We use the syntax $a.1.b$ to refer to the vertex that is 1 edge from a in the direction of b. For example, the vertex one to the right of the start position of q_1 is called $lm.1.h$.

A.1 CBS for Q-MAPF

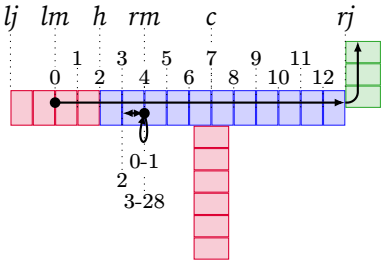


Conflict 1
Direction conflict at $t = 2$:
q_1 wants to travel right
q_2 wants to idle
Constraint 1
$\langle q_1, \text{cont}, \text{Idle}, 2 \rangle$

Conflict 2
Direction conflict at $t = 2$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 2
$\langle q_2, \text{cont}, \text{Reverse}, 2 \rangle$

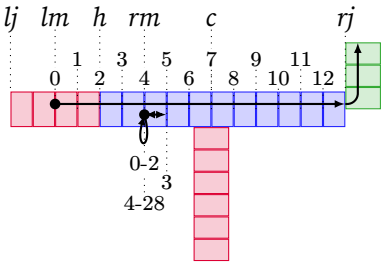
Figure A.2: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(c)



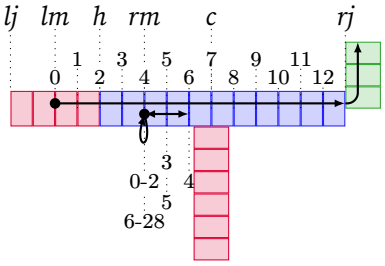
Conflict 3
Direction conflict at $t = 3$:
q_1 wants to travel right
q_2 wants to idle
Constraint 3
$\langle q_2, \text{cont}, \text{Idle}, 3 \rangle$

(d)



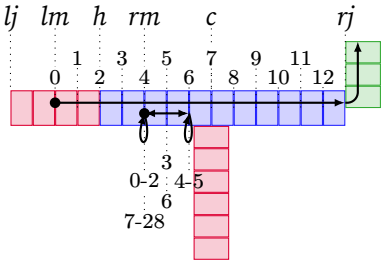
Conflict 4
Direction conflict at $t = 3$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 4
$\langle q_2, \text{cont}, \text{Reverse}, 3 \rangle$

(e)



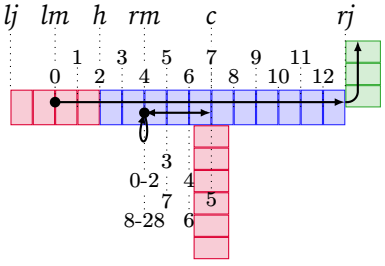
Conflict 5
Direction conflict at $t = 4$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 5
$\langle q_2, \text{cont}, \text{Reverse}, 4 \rangle$

(f)



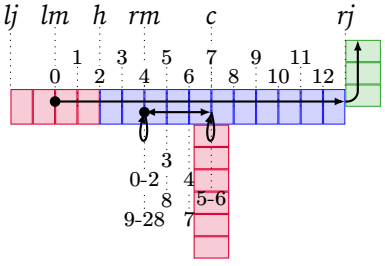
Conflict 6
Direction conflict at $t = 4$:
q_1 wants to travel right
q_2 wants to idle
Constraint 6
$\langle q_2, \text{cont}, \text{Idle}, 4 \rangle$

(g)



Conflict 7
Direction conflict at $t = 5$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 7
$\langle q_2, \text{cont}, \text{Reverse}, 5 \rangle$

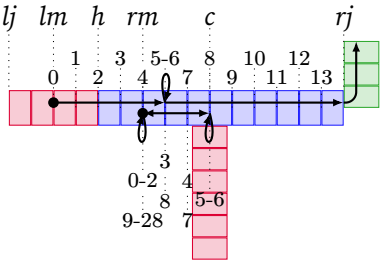
(h)



Conflict 8
Direction conflict at $t = 5$:
q_1 wants to travel right
q_2 wants to idle
Constraint 8
$\langle q_1, \text{cont}, \text{Forward}, 5 \rangle$

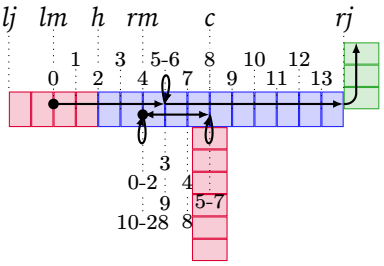
Figure A.2: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(i)



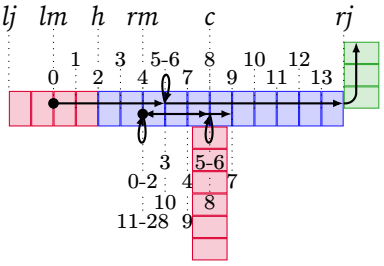
Conflict 9
Direction conflict at $t = 6$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 9
$\langle q_2, \text{cont}, \text{Reverse}, 6 \rangle$

(j)



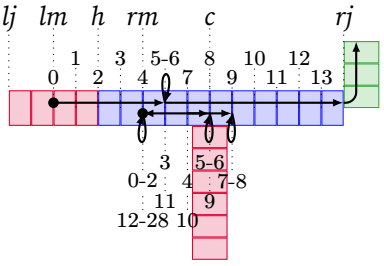
Conflict 10
Direction conflict at $t = 6$:
q_1 wants to travel right
q_2 wants to idle
Constraint 10
$\langle q_2, \text{cont}, \text{Idle}, 6 \rangle$

(k)



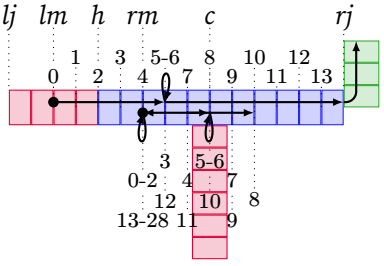
Conflict 11
Direction conflict at $t = 7$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 11
$\langle q_2, \text{cont}, \text{Reverse}, 7 \rangle$

(l)



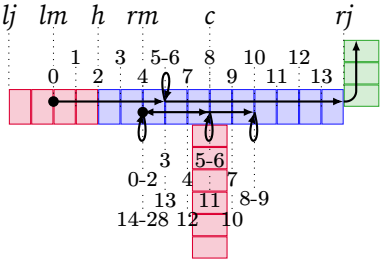
Conflict 12
Direction conflict at $t = 7$:
q_1 wants to travel right
q_2 wants to idle
Constraint 12
$\langle q_2, \text{cont}, \text{Idle}, 7 \rangle$

(m)



Conflict 13
Direction conflict at $t = 8$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 13
$\langle q_2, \text{cont}, \text{Reverse}, 8 \rangle$

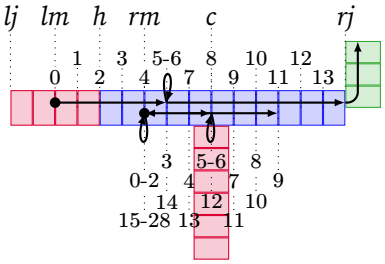
(n)



Conflict 14
Direction conflict at $t = 8$:
q_1 wants to travel right
q_2 wants to idle
Constraint 14
$\langle q_2, \text{cont}, \text{Idle}, 8 \rangle$

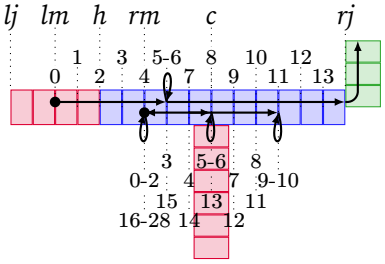
Figure A.2: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(o)



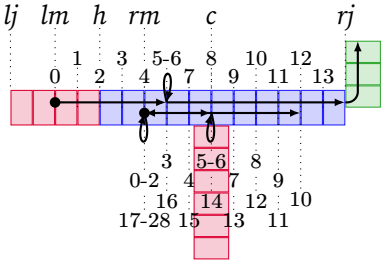
Conflict 15
Direction conflict at $t = 9$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 15
$\langle q_2, \text{cont}, \text{Reverse}, 9 \rangle$

(p)



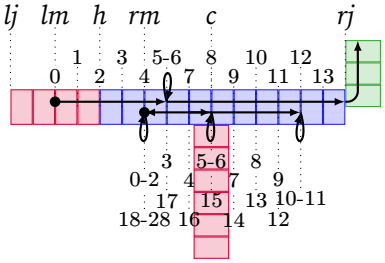
Conflict 16
Direction conflict at $t = 9$:
q_1 wants to travel right
q_2 wants to idle
Constraint 16
$\langle q_2, \text{cont}, \text{Idle}, 9 \rangle$

(q)



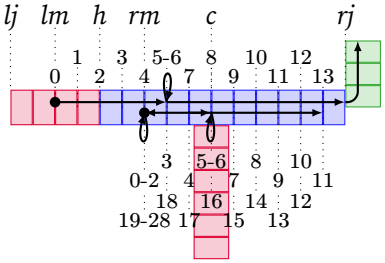
Conflict 17
Direction conflict at $t = 10$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 17
$\langle q_2, \text{cont}, \text{Reverse}, 10 \rangle$

(r)



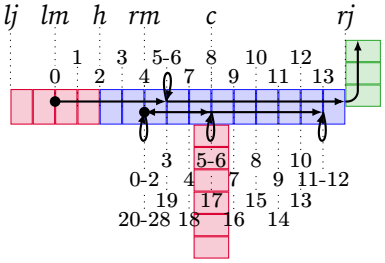
Conflict 18
Direction conflict at $t = 10$:
q_1 wants to travel right
q_2 wants to idle
Constraint 18
$\langle q_2, \text{cont}, \text{Idle}, 10 \rangle$

(s)



Conflict 19
Direction conflict at $t = 11$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 19
$\langle q_2, \text{cont}, \text{Reverse}, 11 \rangle$

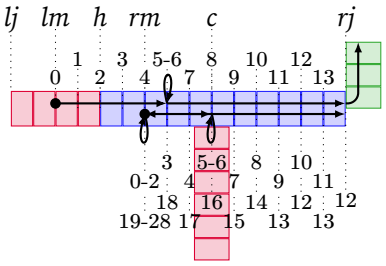
(t)



Conflict 20
Direction conflict at $t = 11$:
q_1 wants to travel right
q_2 wants to idle
Constraint 20
$\langle q_2, \text{cont}, \text{Idle}, 11 \rangle$

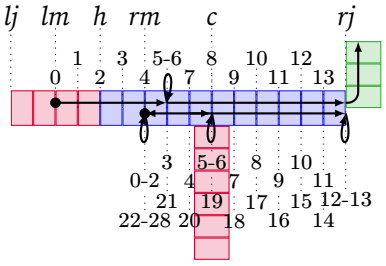
Figure A.2: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(u)



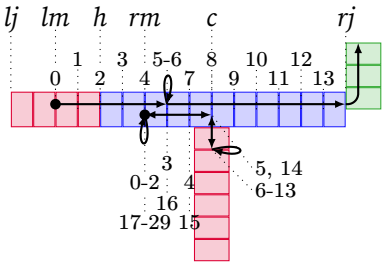
Conflict 21
Direction conflict at $t = 12$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 21
$\langle q_2, \text{cont}, \text{Reverse}, 12 \rangle$

(v)



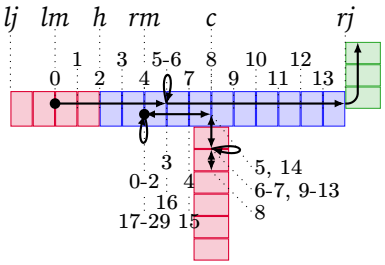
Conflict 22
Direction conflict at $t = 12$:
q_1 wants to travel right
q_2 wants to idle
Constraint 22
$\langle q_2, \text{cont}, \text{Idle}, 12 \rangle$

(w)



Conflict 23
Occupancy conflict at $t = 8$:
q_1 wants to occupy $c.1.irz$
q_2 wants to occupy c
Constraint 23
$\langle q_2, \{c, c.1.irz\}, 8 \rangle$

(x)



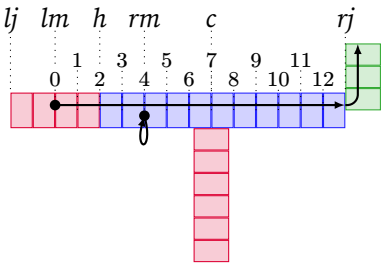
Solution

Figure A.2: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

A.2 CBS for Q-MAPF with Conveyor Conflicts

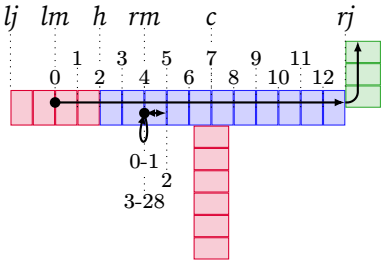
For CBS with conveyor conflicts, we deviate slightly from the low-level description we gave before: In Figure A.3t and later we prefer moving q_2 right over down towards the IRZ. This is still optimal since we have an implicit timing constraint anyways, but no longer returns q_2 to its destination as early as possible.

(a)



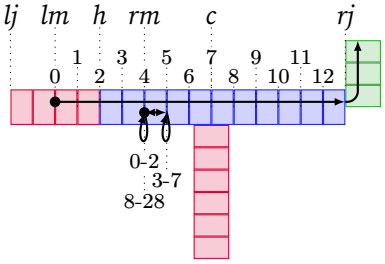
Conflict 1
Direction conflict at $t = 2$:
q_1 wants to travel right
q_2 wants to idle
Constraint 1
$\langle q_1, \text{cont}, \text{Idle}, 2 \rangle$

(b)



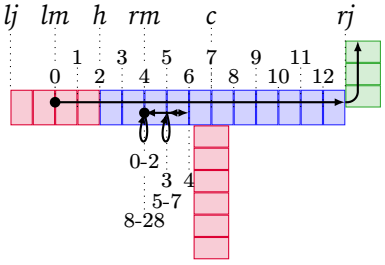
Conflict 2
Direction/Conveyor conflict at $t = 2$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 2
$\langle q_2, rm, [3, 7] \rangle$

(c)



Conflict 3
Direction conflict at $t = 3$:
q_1 wants to travel right
q_2 wants to idle
Constraint 3
$\langle q_2, \text{cont}, \text{Idle}, 3 \rangle$

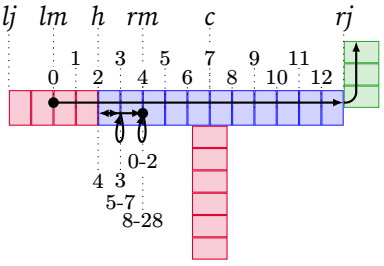
(d)



Conflict 4
Direction/Conveyor conflict at $t = 4$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 4
$\langle q_2, rm.1.c, [5, 7] \rangle$

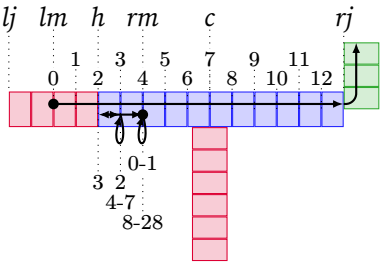
Figure A.3: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(e)



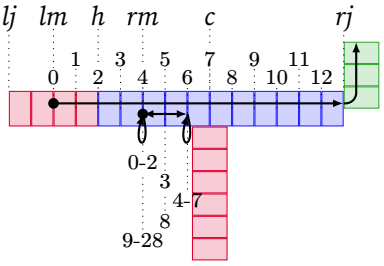
Conflict 5
Direction/Conveyor conflict at $t = 2$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 5
$\langle q_2, h, [4, 8] \rangle$

(f)



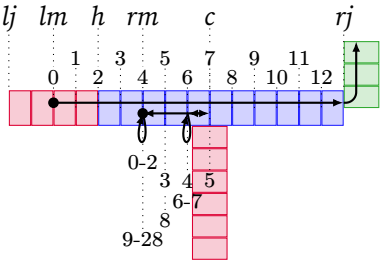
Conflict 6
Occupancy conflict at $t = 2$:
q_1 wants to occupy h
q_2 wants to occupy $h.1.rm$
Constraint 6
$\langle q_2, \{h, h.1.rm\}, 2 \rangle$

(g)



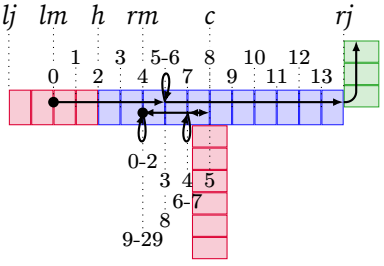
Conflict 7
Direction conflict at $t = 4$:
q_1 wants to travel right
q_2 wants to idle
Constraint 7
$\langle q_2, \text{cont}, \text{Idle}, 4 \rangle$

(h)



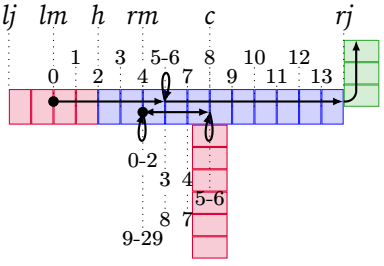
Conflict 8
Direction/Conveyor conflict at $t = 5$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 8
$\langle q_1, c, [7, 7] \rangle$

(i)



Conflict 9
Direction conflict at $t = 5$:
q_1 wants to idle
q_2 wants to travel left
Constraint 9
$\langle q_2, \text{cont}, \text{Reverse}, 5 \rangle$

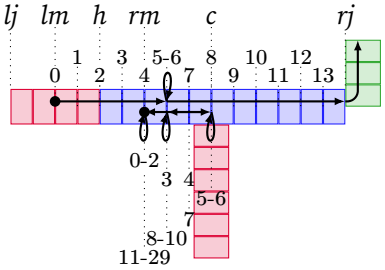
(j)



Conflict 10
Direction/Conveyor conflict at $t = 6$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 10
$\langle q_2, rm, [9, 10] \rangle$

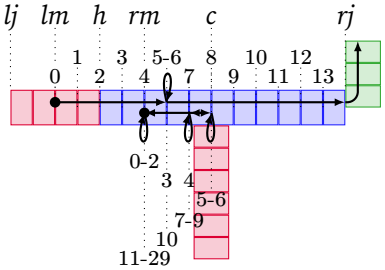
Figure A.3: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(k)



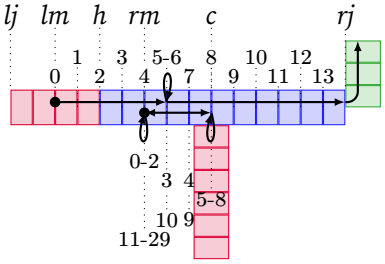
Conflict 11
Direction/Conveyor conflict at $t = 6$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 11
$\langle q_2, rm.1.c, [8, 9] \rangle$

(l)



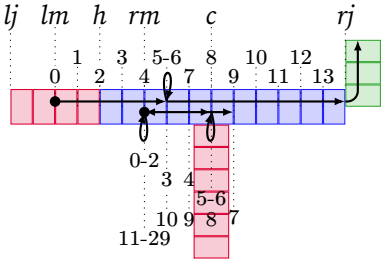
Conflict 12
Direction/Conveyor conflict at $t = 6$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 12
$\langle q_2, rm.2.c, [7, 8] \rangle$

(m)



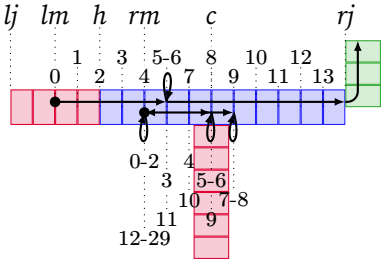
Conflict 13
Direction conflict at $t = 6$:
q_1 wants to travel right
q_2 wants to idle
Constraint 13
$\langle q_2, \text{cont}, \text{Idle}, 6 \rangle$

(n)



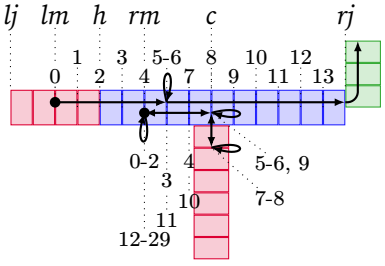
Conflict 14
Direction/Conveyor conflict at $t = 7$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 14
$\langle q_2, c, [8, 8] \rangle$

(o)



Conflict 15
Direction conflict at $t = 7$:
q_1 wants to travel right
q_2 wants to idle
Constraint 15
$\langle q_2, \text{cont}, \text{Idle}, 7 \rangle$

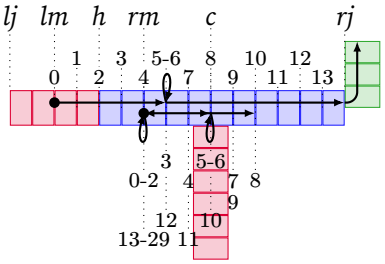
(p)



Conflict 16
Direction conflict at $t = 8$:
q_1 wants to travel right
q_2 wants to idle
Constraint 16
$\langle q_2, \text{cont}, \text{Idle}, 8 \rangle$

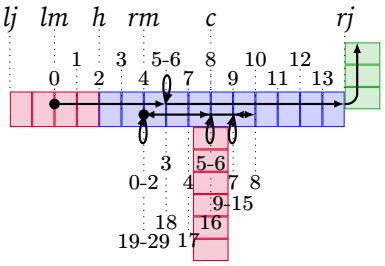
Figure A.3: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(q)



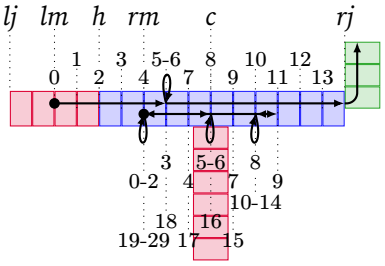
Conflict 17
Direction/Conveyor conflict at $t = 8$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 17
$\langle q_2, c, [10, 15] \rangle$

(r)



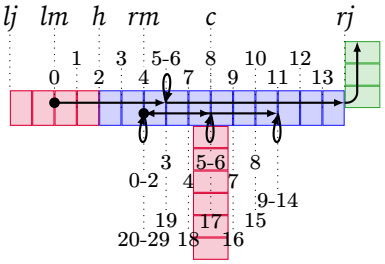
Conflict 18
Direction/Conveyor conflict at $t = 8$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 18
$\langle q_2, c.1.rj, [9, 14] \rangle$

(s)



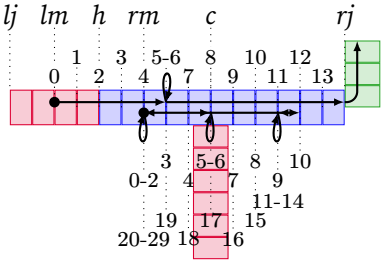
Conflict 19
Direction/Conveyor conflict at $t = 9$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 19
$\langle q_2, c.2.rj, [10, 14] \rangle$

(t)



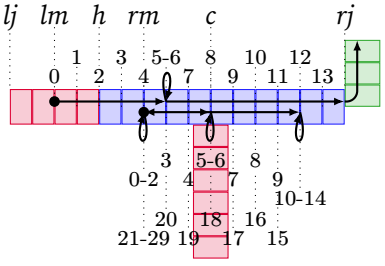
Conflict 20
Direction conflict at $t = 9$:
q_1 wants to travel right
q_2 wants to idle
Constraint 20
$\langle q_2, \text{cont}, \text{Idle}, 9 \rangle$

(u)



Conflict 21
Direction/Conveyor conflict at $t = 10$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 21
$\langle q_2, c.3.rj, [11, 14] \rangle$

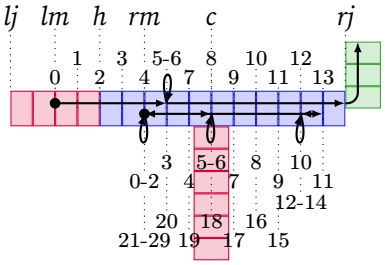
(v)



Conflict 22
Direction conflict at $t = 10$:
q_1 wants to travel right
q_2 wants to idle
Constraint 22
$\langle q_2, \text{cont}, \text{Idle}, 10 \rangle$

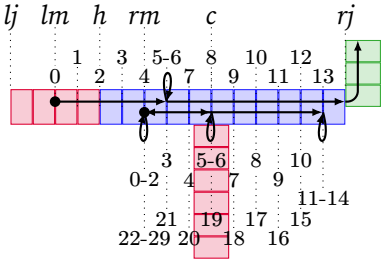
Figure A.3: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

(w)



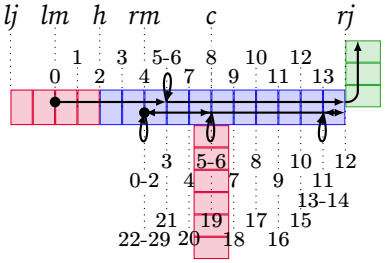
Conflict 23
Direction/Conveyor conflict at $t = 11$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 23
$\langle q_2, c.4.rj, [12, 14]\rangle$

(x)



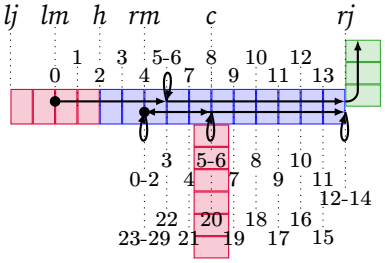
Conflict 24
Direction conflict at $t = 11$:
q_1 wants to travel right
q_2 wants to idle
Constraint 24
$\langle q_2, \text{cont}, \text{Idle}, 11\rangle$

(y)



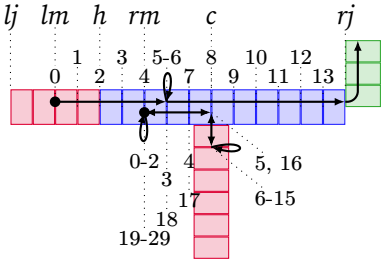
Conflict 25
Direction/Conveyor conflict at $t = 12$:
q_1 wants to travel right
q_2 wants to travel left
Constraint 25
$\langle q_2, c.5.rj, [13, 14]\rangle$

(z)



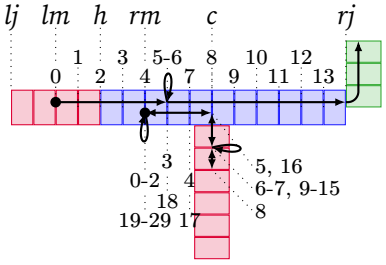
Conflict 26
Direction conflict at $t = 12$:
q_1 wants to travel right
q_2 wants to idle
Constraint 26
$\langle q_2, \text{cont}, \text{Idle}, 12\rangle$

(aa)



Conflict 27
Occupancy conflict at $t = 8$:
q_1 wants to occupy $c.1.irz$
q_2 wants to occupy c
Constraint 27
$\langle q_2, \{c, c.1.irz\}, 8\rangle$

(ab)



Solution

Figure A.3: Conflicts and Constraints for CBS while solving the 3rd layer of QFT_3

APPENDIX B

Benchmark Results

This appendix contains benchmark results as described in section 3.3.

		CBS (with g.d.c.)		CBS (with corridors)	
Circuit	Scale	Min	Max	Min	Max
QFT_3	0.1	0.028	0.056	0.151	1.078
	0.2	0.043	0.077	0.177	0.988
	0.3	0.096	0.168	0.407	2.267
	0.4	0.141	0.263	1.892	8.958
	0.5	0.272	0.461	4.972	30.506
	0.6	0.385	0.641	16.674	153.570
	0.7	0.460	0.844	17.494	128.429
	0.8	0.761	1.350	101.543	572.894
	0.9	0.957	1.493	100.214	299.893
	1.0	1.242	1.897	DNF	DNF
QFT_4	0.1	0.416	1.071	5.395	DNF
	0.2	24.043	109.402		
	0.3	378.872	1772.406		
	0.4	DNF	DNF		
ShorEnc	0.1	DNF	DNF	DNF	DNF

Table B.1: Benchmark between CBS with global direction constraints and CBS with conveyor conflicts and range constraints (Minimum and maximum; in seconds)

		A*/OD		CBS		CBS (with g.d.c.)	
Circuit	Scale	Avg	SD	Avg	SD	Avg	SD
QFT_3	0.1	0.010	0.0004	0.040	0.0078	0.041	0.0060
	0.2	0.012	0.0005	0.054	0.0099	0.055	0.0082
	0.3	0.023	0.0008	0.119	0.0138	0.119	0.0135
	0.4	0.056	0.0011	0.191	0.0363	0.187	0.0296
	0.5	0.093	0.0013	0.330	0.0513	0.335	0.0431
	0.6	0.155	0.0023	0.497	0.0680	0.504	0.0593
	0.7	0.166	0.0022	0.610	0.0867	0.599	0.0821
	0.8	0.268	0.0031	0.979	0.1648	0.971	0.1303
	0.9	0.287	0.0039	1.176	0.1685	1.169	0.1174
	1.0	0.444	0.0069	1.558	0.1983	1.517	0.1700
QFT_4	0.1	5.772	0.0309	0.888	0.3784	0.688	0.2101
	0.2	44.649	0.2734	99.828	47.9865	61.505	25.1929
	0.3	174.280	0.6339	DNF		1169.648	439.1450
	0.4	DNF				DNF	
ShorEnc	0.1	DNF		DNF		DNF	

Table B.2: Benchmark between A*/OD and CBS with and without global direction constraints (Average and standard deviation; in seconds)

		A*/OD		CBS		CBS (with g.d.c.)	
Circuit	Scale	Min	Max	Min	Max	Min	Max
QFT_3	0.1	0.009	0.011	0.028	0.060	0.028	0.056
	0.2	0.011	0.013	0.038	0.083	0.043	0.077
	0.3	0.022	0.025	0.090	0.155	0.096	0.168
	0.4	0.054	0.059	0.128	0.273	0.141	0.263
	0.5	0.090	0.096	0.254	0.450	0.272	0.461
	0.6	0.152	0.161	0.389	0.642	0.385	0.641
	0.7	0.163	0.174	0.448	0.848	0.460	0.844
	0.8	0.262	0.276	0.727	1.408	0.761	1.350
	0.9	0.280	0.298	0.912	1.624	0.957	1.493
	1.0	0.432	0.460	1.289	2.155	1.242	1.897
QFT_4	0.1	5.695	5.878	0.377	1.970	0.416	1.071
	0.2	44.278	45.957	44.443	221.784	24.043	109.402
	0.3	172.972	176.342	1310.230	DNF	378.872	1772.406
	0.4	DNF	DNF			DNF	DNF
ShorEnc	0.1	DNF	DNF	DNF	DNF	DNF	DNF

Table B.3: Benchmark between A*/OD and CBS with and without global direction constraints (Minimum and maximum; in seconds)

		CBS (with g.d.c.)		CBS (disjoint)		CBS (disjoint/global)	
Circuit	Scale	Avg	SD	Avg	SD	Avg	SD
QFT_3	0.1	0.041	0.0060	0.032	0.0034	0.032	0.0038
	0.2	0.055	0.0082	0.044	0.0047	0.046	0.0049
	0.3	0.119	0.0135	0.101	0.0097	0.104	0.0093
	0.4	0.187	0.0296	0.141	0.0182	0.148	0.0207
	0.5	0.335	0.0431	0.257	0.0301	0.263	0.0284
	0.6	0.504	0.0593	0.397	0.0387	0.394	0.0386
	0.7	0.599	0.0821	0.497	0.0558	0.501	0.0402
	0.8	0.971	0.1303	0.807	0.0897	0.813	0.0886
	0.9	1.169	0.1174	0.989	0.0817	0.973	0.0814
	1.0	1.517	0.1700	1.257	0.1196	1.206	0.1110
QFT_4	0.1	0.688	0.2101	0.422	0.1008	0.419	0.0939
	0.2	61.505	25.1929	26.476	9.3264	25.890	9.4550
	0.3	1169.648	439.1450	510.072	133.4955	465.693	134.0381
	0.4	DNF		DNF		DNF	
ShorEnc	0.1	DNF		24.221	16.7873	25.273	21.9351
	0.2			80.493	16.1427	78.184	18.0166
	0.3			DNF		DNF	

Table B.4: Benchmark between CBS with global direction constraints and CBS with disjoint splitting, with and without (modified) global direction constraints (Average and standard deviation; in seconds)

		CBS (with g.d.c.)		CBS (disjoint)		CBS (disjoint/global)	
Circuit	Scale	Min	Max	Min	Max	Min	Max
QFT_3	0.1	0.028	0.056	0.025	0.040	0.024	0.043
	0.2	0.043	0.077	0.035	0.056	0.036	0.058
	0.3	0.096	0.168	0.080	0.123	0.085	0.124
	0.4	0.141	0.263	0.107	0.199	0.116	0.201
	0.5	0.272	0.461	0.204	0.323	0.210	0.318
	0.6	0.385	0.641	0.319	0.477	0.317	0.483
	0.7	0.460	0.844	0.391	0.604	0.425	0.577
	0.8	0.761	1.350	0.650	1.021	0.649	1.012
	0.9	0.957	1.493	0.834	1.186	0.814	1.201
	1.0	1.242	1.897	1.038	1.487	1.010	1.455
QFT_4	0.1	0.416	1.071	0.279	0.653	0.268	0.657
	0.2	24.043	109.402	11.867	41.379	10.949	41.657
	0.3	378.872	1772.406	159.162	740.577	154.588	673.830
	0.4	DNF	DNF	905.381	DNF	933.215	DNF
ShorEnc	0.1	DNF	DNF	10.089	100.698	10.063	149.225
	0.2			57.333	127.226	54.666	122.672
	0.3			DNF	DNF	DNF	DNF

Table B.5: Benchmark between CBS with global direction constraints and CBS with disjoint splitting, with and without (modified) global direction constraints (Minimum and maximum; in seconds)

		A*/OD (TAPF)		CBS (TAPF with g.d.c.)		CBS (TAPF, disjoint)		CBS (TAPF, disjoint/global)	
Circuit	Scale	Avg	SD	Avg	SD	Avg	SD	Avg	SD
QFT_3	0.1	0.106	0.0014	0.142	0.0432	0.094	0.0225	0.100	0.0269
	0.2	0.502	0.0047	3.500	9.2464	3.332	3.7719	2.363	3.2457
	0.3	1.133	0.0096	DNF		DNF		DNF	
	0.4	1.637	0.0098						
	0.5	1.830	0.0143						
	0.6	1.746	0.0144						
	0.7	3.516	0.0278						
	0.8	4.488	0.0436						
	0.9	7.825	0.0548						
	1.0	7.566	0.0681						
QFT_4	0.1	19.012	0.2962	7.257	7.9869	1.542	1.6938	1.302	1.1520
	0.2	22.653	0.1953	DNF		453.815	666.6487	DNF	
	0.3	88.634	0.9532			DNF			
	0.4	DNF							
ShorEnc	0.1	DNF		DNF		DNF		DNF	

Table B.6: Benchmark for TAPF between A*/OD and CBS, with and without disjoint splitting and with and without global direction constraints (Average and standard deviation; in seconds)

		A*/OD (TAPF)		CBS (TAPF with g.d.c.)		CBS (TAPF, disjoint)		CBS (TAPF, disjoint/global)	
Circuit	Scale	Min	Max	Min	Max	Min	Max	Min	Max
QFT_3	0.1	0.104	0.110	0.083	0.246	0.057	0.175	0.056	0.213
	0.2	0.494	0.517	0.470	35.828	0.228	11.214	0.225	9.608
	0.3	1.115	1.170	5.099	DNF	DNF	DNF	DNF	DNF
	0.4	1.614	1.662						
	0.5	1.802	1.869						
	0.6	1.716	1.780						
	0.7	3.458	3.586						
	0.8	4.429	4.637						
	0.9	7.737	7.977						
	1.0	7.474	7.809						
QFT_4	0.1	18.716	20.609	1.317	44.057	0.415	8.632	0.486	6.950
	0.2	22.383	23.522	DNF	DNF	2.552	2264.009	9.042	DNF
	0.3	87.254	92.877			2042.382	DNF		
	0.4	DNF	DNF						
ShorEnc	0.1	DNF	DNF	2.273	DNF	1.564	DNF	1.913	DNF

Table B.7: Benchmark for TAPF between A*/OD and CBS, with and without disjoint splitting and with and without global direction constraints (Minimum and maximum; in seconds)