POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Matematica

Tesi di Laurea Magistrale

A multi-agent path finding instance: the Matrix Problem

Relatori
prof. Fabio Fagnani

firma del relatore

. . . .

Candidato
Andrea Di Nezza

firma del candidato

. . . .

Anno Accademico 2024-2025

Alle mie sorelle, S. e S.
† A mia nonna Marcella

Summary

The Multi-Agent Pathfinding (MAPF) problem is the fundamental challenge of planning paths for multiple agents, subject to the key constraint that agents must be able to follow their paths concurrently without colliding with each other. Applications of MAPF include automated warehouses and autonomous vehicles.

This thesis investigates a specific extension of the MAPF problem, known as the "Matrix Problem". In the Matrix Problem, all agents, initially at rest in predefined positions, must reach a common destination target set without vertex or swapping conflicts. Agents are divided into two categories: tasked agents, whose motion must pass through a target vertex before stopping, and untasked agents, which can move directly to occupy a position within the destination target.

In particular, this work focuses on the "Full Tasked Matrix Problem", where the graph is saturated with tasked agents. We model this scenario as a set of synchronized collective motions taking place along the edges of the graph. Within this framework, we define all relevant motion types and establish the necessary and sufficient topological requirements for the existence of feasible solutions.

Finally, through illustrative examples, we derive sufficient conditions for both minimal and optimal solutions. These results open perspectives for applications in network optimization and multi-agent systems, representing a first step toward addressing the complete Matrix Problem.

Contents

List of Figures

Life
can only be understood
backwards;
but it must be lived
forwards.

[Søren Kierkegaard, Journals]

Chapter 1

Introduction

Multi-Agent Systems (MAS) consist of multiple autonomous agents that interact within a shared environment to achieve individual or collective goals. These systems are inherently decentralized, with each agent acting based on its own perceptions and objectives while coordinating with others to accomplish tasks beyond the capabilities of a single agent.

MAS have found applications in smart manufacturing, where multiple robots collaborate to perform complex assembly operations efficiently Pulikottil [2023], as well as in intelligent transportation systems, where they help optimize traffic flow and autonomous vehicle routing Balbo et al. [2024]. Additionally, MAS have been applied in public safety to support coordinated responses to emergencies Ma et al. [2025]. Recent research has focused on enhancing scalability and adaptability, with approaches such as multi-agent reinforcement learning Donatus et al. [2025] and integration of large language models to enable natural language-based coordination Tran et al. [2025].

Building on MAS, **Multi-Agent Pathfinding** (MAPF) is a fundamental problem in multi-agent planning that aims to compute collision-free paths for multiple agents moving from their start positions to designated goals. The main challenge of MAPF is ensuring that all agents follow their paths concurrently without collisions, critical in environments where efficiency and safety are essential.

Discrete multi-agent path planning problems seem to have originated from Sam Loyd's 15-puzzle (Story [1879], Loyd [1959]), a well-known board-based puzzle game. The 15-puzzle can be viewed as moving 15 agents on a 16-vertex grid graph, which generalizes to the multi-agent path planning problem on a N-vertex graph with $n < N$ agents. In the most basic formulation, only one pebble moves per time step to an adjacent unoccupied vertex; this is called pebble motion on graphs (PMG).

Since agents act autonomously and can communicate, multiple agents are capable of moving in the same time step. A parallel move of agents is a synchronous move of a (non-self-intersecting) chain of agents as long as the first agent moves into a vertex unoccupied at the beginning of the time step. If multiple disjoint parallel moves per step are allowed, this variant is multi-agent path planning on graphs with parallel moves (MPPp) Ryan [2008], Surynek [2010].

Multi-Agent Path Finding generalizes MPPp by explicitly enforcing collision constraints for all agents, including vertex and edge collisions, and often introducing optimization objectives such as minimizing the makespan or the sum-of-costs. MAPF builds on PMG and MPPp while incorporating both feasibility under rigid constraints and path optimality.

MAPF has gained attention due to its practical relevance. In automated warehouses, it allows fleets of robots to transport goods efficiently without conflicts Makino and Ito [2025]. In autonomous driving, MAPF techniques plan safe coordinated trajectories for multiple vehicles navigating shared roads Reda [2024]. In robotics, multi-agent coordination supported by MAPF is crucial for collaborative tasks in factories or shared workspaces Grenouilleau et al. [2019].

Recent advances in MAPF research focus on improving scalability and adaptability in real-world applications. Methods such as finite-horizon hierarchical planning allow agents to plan in a receding-horizon fashion, reducing computational complexity in dynamic settings Li et al. [2025]. Lifelong MAPF approaches handle scenarios where agents are continuously assigned new goals by decomposing the problem into sequences of smaller planning instances Li et al. [2021]. Additionally, algorithms like EECBS offer bounded-suboptimal solutions that balance solution quality and computational efficiency.

The **Matrix Problem** (MP) is a variant of Multi-Agent Path Finding (MAPF) that presents two distinctive features compared to the traditional MAPF:

1. No target is assigned to individual agents; there exists only a common target set, and it does not matter which agent occupies which node within the set.

2. Some agents are required to pass through one and only one specific node, where they may, for example, drop off or pick up an object, introducing a coordination constraint along their path.

In this scenario, agents must move while avoiding collisions and respecting the passage constraints, which makes the problem more complex than standard MAPF, while maintaining greater flexibility in the final assignment of agents to targets.

The Matrix Problem is relevant in contexts where the sequence or transit through key points is more important than the individual assignment of targets:

- **Industrial automation:** Collaborative robots in production lines must traverse key stations for inspection, assembly, or object exchange, without constraints on each robot's final node.

- **Logistics and warehousing:** Mobile robots in automated warehouses must pass through mandatory picking or loading/unloading stations to drop off or collect materials, while the final destination can be any node within the target set.

The Matrix Problem falls within the broader context of Multi-Agent Path Finding (MAPF), a problem well-known for its computational complexity. The main challenge lies in computing collision-free paths for a group of agents moving on a shared graph. Considering known results from the literature on the two underlying problem variants:

- **Traditional MAPF:** In this version, each agent has a predefined goal and must reach it without interfering with others. Finding optimal solutions is **NP-hard**, even on simple graphs such as 2D grids Jing and LaValle [2013], Ma and Koenig [2016].
- **Anonymous MAPF:** In this variant, agents have no predefined individual goals; only a **common goal set** exists, and it does not matter which agent reaches which target. This approach introduces greater flexibility and reduces problem rigidity. It has been shown that anonymous MAPF can be solved **optimally in polynomial time** using algorithms such as maximum flow on time-expanded networks Zhong et al. [2022].

These differences in computational complexity directly impact the design and implementation of solutions for the Matrix Problem, affecting the efficiency and scalability of the algorithms employed.

The present thesis is structured around the progressive development of the theoretical and algorithmic framework required to analyze and solve the Matrix Problem (MP) and its fully tasked variant. The logical progression of the work is organized into three main parts, each building upon the previous one in order to establish a coherent and rigorous treatment.

The appendix provides an introduction to graph theory, with particular emphasis on the notions that are essential for the subsequent analysis. The aim of this chapter is twofold. On the one hand, it introduces the basic terminology and notation in a self-contained manner, ensuring that the treatment is uniform and unambiguous throughout the thesis. On the other hand, it emphasizes those structural properties of graphs that play a fundamental role in the study of the Matrix Problem.

The first part offers an overview of the MAPF problem, discussing its classical treatment in the literature and outlining the properties that must be taken into account to properly characterize a problem instance.

The second part introduces the Matrix Problem, for which a rigorous mathematical formulation is provided. This chapter carefully develops the necessary definitions and formal structures in order to establish a solid foundation for the theoretical analysis that follows. The contribution of this section is twofold. First, it characterizes the Matrix Problem in precise mathematical terms, presenting the essential definitions required for a clear and consistent understanding of the problem. Second, it introduces a general — though not yet optimized — method that guarantees the existence of a feasible solution and, in certain cases, also leads to the identification of an optimal one.

The final part of the thesis focuses on the Fully Tasked Matrix Problem, the special case in which the graph is saturated with task agents. This setting represents the most constrained and dense configuration of the Matrix Problem and requires the development of new tools for analysis. The first main result of this section is the identification of a topological condition on the graph that is necessary and sufficient for the existence of a feasible solution. This result provides a complete characterization of solvability in the fully tasked case, establishing a precise link between graph structure and the feasibility of coordinated multi-agent motion. Building on this characterization, the second main contribution is the derivation of a sufficient condition for the existence of a minimal-cost solution. This result not only guarantees feasibility but also leads to the explicit construction of a solution whose cost can be proven to be minimal. The analysis thereby bridges existence conditions with optimization aspects, producing both theoretical guarantees and constructive algorithms.

Chapter 2

Multi-Agent Pathfinding

We first describe what we refer to as a classical MAPF problem. The input to a classical MAPF problem with k agents is a tuple $(\mathcal{G}, k, s, t)$ where $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph, $s : [1, ..., k] \rightarrow \mathcal{V}$ maps an agent to a source vertex, and $t : [1, ..., k] \rightarrow \mathcal{V}$ maps an agent to a target vertex. Time is assumed to be discretized, and in every time step each agent is situated in one of the graph vertices and can perform a single action.

An action in classical MAPF is a function $a : \mathcal{V} \rightarrow \mathcal{V}$ such that $a(v) = v'$ means that if an agent is at vertex v and performs a then it will be in vertex v' in the next time step. Each agent has two types of actions: wait and move. A wait action means that the agent stays in its current vertex another time step. A move action means that the agent moves from its current vertex v to an adjacent vertex v' in the graph (i.e., $(v, v') \in \mathcal{E}$). For a sequence of actions $\pi = (a_1, \ldots, a_n)$ and an agent i, we denote by $\pi_i[x]$ the location of the agent after executing the first x actions in π, starting from the agent's source $s(i)$. Formally, $\pi_i[x] = a_x(a_{x-1}(\cdots a_1(s(i))))$. A sequence of actions π is a *single-agent plan* for agent i iff executing this sequence of actions in $s(i)$ results in reaching $t(i)$, that is, iff $\pi_i[|\pi|] = t(i)$. A *solution* is a set of k single-agent plans, one for each agent.

The overarching goal of MAPF solvers is to find a solution, i.e., a single-agent plan for each agent, that can be executed without collisions. To achieve this, MAPF solvers use the notion of *conflicts* during planning, where a MAPF solution is called valid iff there is no conflict between any two single-agent plans.

The definition of what constitutes a conflict depends on the environment, and correspondingly the literature on classical MAPF includes several different definitions of conflicts between plans.

We list common conflict definitions below. Let π_i and π_j be a pair of single-agent plans.

- **Vertex conflict.** A vertex conflict between π_i and π_j occurs iff, according to these plans, the agents are planned to occupy the same vertex at the same time step. Formally, there is a vertex conflict between π_i and π_j iff there exists a time step x such that
$$\pi_i[x] = \pi_j[x].$$
- **Swapping conflict.** A swapping conflict between π_i and π_j occurs iff the agents

are planned to swap locations in a single time step. Formally, there is a swapping conflict between π_i and π_j iff there exists a time step x such that

$$\pi_i[x+1] = \pi_j[x] \quad \text{and} \quad \pi_j[x+1] = \pi_i[x]$$

This conflict is sometimes called an *edge conflict* in the current MAPF literature.

Note that the above set of conflict definitions does not represent all possible conflicts found in the literature, but only those we will consider in this work.

In a solution to a classical MAPF problem, agents may reach their targets at different time steps. Therefore, one must define how an agent behaves in the time steps after it has reached its target and before the last agent has reached its target. Two common assumptions are:

- **Stay at target.** An agent waits at its target until all agents have reached their targets. This waiting agent may cause vertex conflicts with any plan that passes through its target afterwards. Formally, under this assumption, a pair of single-agent plans π_i and π_j have a vertex conflict if there exists a time step $t \geq |\pi_i|$ such that $\pi_j[t] = \pi_i[|\pi_i|]$.

- **Disappear at target.** Once an agent reaches its target it immediately disappears. Hence, the plan of that agent cannot cause any conflict after the time step in which it has reached its target.

In this work, we assume the *stay-at-target* behavior.

It is safe to say that in most real applications of MAPF, some MAPF solutions are better than others. To capture that, work in classical MAPF considers an objective function that is used to evaluate MAPF solutions. The two most common functions used for evaluating a solution in classical MAPF are makespan and sum of costs.

- **Makespan.** The number of time steps required for all agents to reach their targets. For a MAPF solution $\pi = \{\pi_1, \ldots, \pi_k\}$, the makespan is defined as $\max_{1 \leq i \leq k} |\pi_i|$.

- **Sum of costs.** The sum of the lengths of the individual plans. Formally, $\sum_{i=1}^{k} |\pi_i|$. This is also known as *flowtime*.

If the stay-at-target assumption is used together with the sum-of-costs objective, one must specify how waiting at a target is treated. The common assumption is that waiting counts toward the cost only if the agent intends to leave the target later. For instance, if agent i reaches its target at time step t, leaves at time t', returns at time t'', and stays until the end, then its plan contributes t'' to the sum of costs.

Makespan has been used extensively by compilation-based MAPF algorithms, while sum of costs has been used by most search-based MAPF algorithms. But, there has also been work on both objective functions by both types of MAPF algorithms (Surynek et al. [2016]). There has also been work on maximizing the number of agents reaching their targets within a given makespan (i.e., deadline) (Ma et al. [2018]).

In classical MAPF, each agent has exactly one task - to get it to its target. Several extensions have been made in the MAPF literature in which agents may be assigned more than one target.

In anonymous MAPF, the objective is to move the agents to a set of target vertices, but it does not matter which agent reaches which target (Kloder and Hutchinson [2006], Yu and LaValle [2013]). Another way to view this MAPF variant is as a MAPF problem in which every agent can be assigned to any target, but it has to be a one-to-one mapping between agents and targets.

In online MAPF, a sequence of MAPF problems is solved on the same graph. This is also known as *lifelong MAPF* (Ma et al. [2017, 2019]). Two models are:

- **Warehouse model.** A fixed set of agents repeatedly receives new targets after reaching previous ones (Ma et al. [2019]). This setting is inspired by MAPF for autonomous warehouses.
- **Intersection model.** New agents can appear, each with a single task Svancara et al. [2019], Dresner and Stone [2008]. This setting is inspired by autonomous vehicles entering and exiting intersections.

Hybrid models, combining both behaviors, are also possible. In our setting, we consider a variant of the warehouse model where future targets are known in advance, enabling global optimization from the beginning.

MAPF can be categorized into two main settings. In the *distributed setting*, each agent has its own computational resources and decision-making protocol, which may be either self-interested or cooperative Grady et al. [2011], Bhattacharya and Kumar [2011], Bnaya et al. [2013]. Various communication paradigms can be assumed, such as message passing or broadcasting. In contrast, the *centralized setting* assumes that all agents are controlled by a single decision-maker.

This paper focuses exclusively on centralized algorithms, where the aim is to minimize a global cost function.

Chapter 3

The Matrix Problem

3.1 Problem Presentation

In the *Matrix Problem*, we are given a set of agents initially positioned at fixed locations on a directed graph. The goal is for these agents to navigate the graph without colliding with each other, while ensuring that some pass through a specific vertex, known as the *origin*. Additionally, all agents must stop moving once they reach their final positions, which must lie within a designated set of vertices.

The input consists of the following components:

- A directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $n = |\mathcal{V}|$,
- An origin vertex $o \in \mathcal{V}$,
- A set of k agents $\mathcal{A} = \{1, \ldots, k\}$,
- A corresponding set of distinct starting vertices $s_1, \ldots, s_k$, where $s_a \in \mathcal{V}$ for all $a \in \mathcal{A}$,
- A set of storage vertices $\mathcal{D} \subseteq \mathcal{V}$, where $m = |\mathcal{D}| \geq k$.

The agents are divided into two categories:

- *Task agents* $\mathcal{A}^t = \left\{1, \ldots, \overline{k}\right\}$, and
- *Untasked agents* $\mathcal{A}^u = \mathcal{A} \setminus \mathcal{A}^t$.

Definition 3.1 (Matrix instance)**.** The n-tuple $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$ is referred to as a *Matrix instance.*

Example 3.1. *Consider the simple example illustrated in Figure 3.1:*

- *Let* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *be a directed graph, with*
  - *Four vertices* $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$*, and*

  - *The following directed edges: $v_1 \to v_2$, $v_2 \to v_3$, $v_3 \to v_4$, and $v_4 \to v_1$: $\mathcal{E} = \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$.*

- *The origin is $o = v_3$ (The origin vertex is highlighted with a red border in the figure).*

- *We have $k = 3$ agents (The agents are indicated with different colors: orange, green, and blue).*

- *Only agents 1 and 2 are task agents ($\bar{k} = 2$), so $\mathcal{A}^t = \{1, 2\}$ and $\mathcal{A}^u = \{3\}$.*

- *The starting positions of the agents are:*

  - *Agent 1 starts at v_1,*
  - *Agent 2 starts at v_2,*
  - *Agent 3 starts at v_4.*

- *The storage vertices are $\mathcal{D} = \{v_2, v_3, v_4\}$ (Storage vertices are highlighted with thicker borders in the figure).*

During the agents' movements, we must avoid two types of conflicts:

(a) *Vertex conflict*: Two agents occupy the same vertex at the same time.

(b) *Swapping conflict*: Two agents exchange positions between consecutive time steps.

Example 3.2. *Consider a ring graph with four vertices and two agents, each represented by a different color. We present two examples, one for each type of conflict, and for each case, we show the graph with the two agents before and after a movement:*

(a) A vertex conflict occurs in Figure 3.2a, where both agents attempt to move into the same vertex. In real-world applications, such a situation is physically implausible.

(b) A Swapping conflict is shown in Figure 3.2b. Here, the two agents attempt to swap positions: the first wants to move along an edge to the position of the second, while the second wants to use the same edge in the opposite direction. As in the previous case, this scenario is not physically feasible in real-world applications.

Observation 1. *We can observe that the vertex conflict is a static conflict, tied to the agents' positions at a specific point in time. On the other hand, the swapping conflict is a dynamic conflict: simply observing the agents' positions at a given moment does not reveal this conflict, which arises when considering their movement and the subsequent change in positions.*

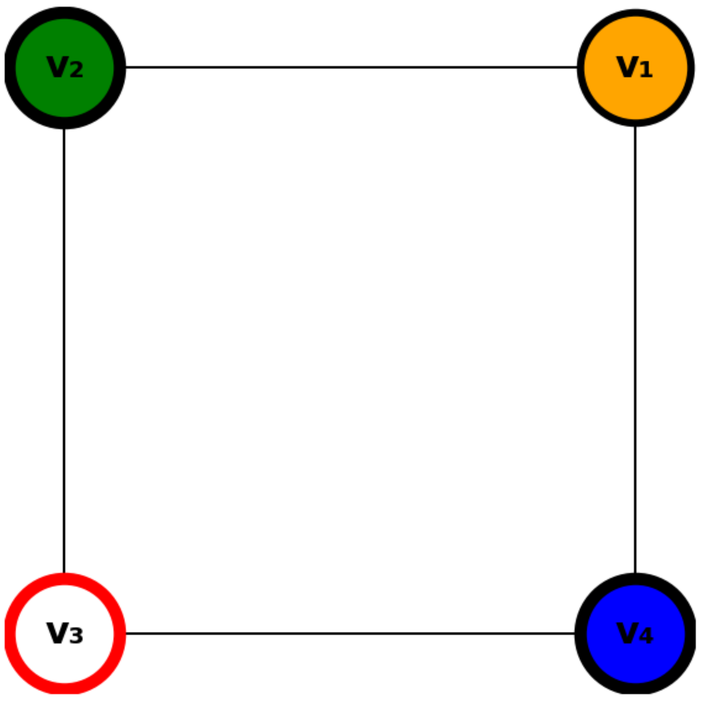
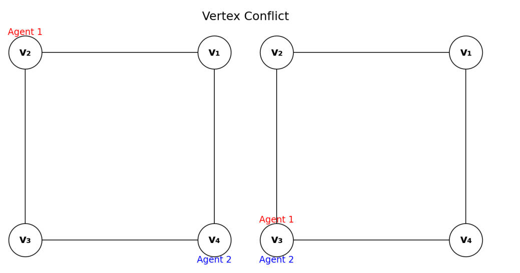
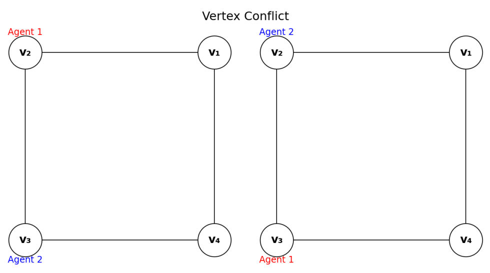


Figure 3.1: Example of a Matrix instance.



(a) Vertex conflict: two agents occupy the same vertex simultaneously.



(b) Swapping conflict: two agents swap positions in consecutive steps.

Figure 3.2: Types of conflicts to avoid.

3.2 Description of agent' motions

We begin by considering only the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and the agents $\mathcal{A} = \{1, \ldots, k\}$.

The classical approach to this problem is to assign to each agent a walk on the graph. Formally, for every agent $a \in \mathcal{A}$ we are given a starting position $s_a \in \mathcal{V}$, and we impose the requirement that the terminal position of the walk must belong to the set of storage vertices $\mathcal{D}$.

In principle, one could independently compute a walk for each agent and then collect these k walks together. However, this is not sufficient: the individual walks must be synchronized in time. Indeed, it may happen that while one agent is moving, another remains stationary. This type of behavior cannot be represented if walks are only considered as minimal vertex-to-vertex sequences without further adjustments.

If we represent each walk as a sequence of vertices, synchronizing different agents is straightforward: one can simply allow repetitions of vertices to represent "waiting steps". On the other hand, if we wish to regard a walk as a sequence of edges, then such repetitions are not directly available. To resolve this, we must introduce self-loops: edges of the form (v, v) for each $v \in \mathcal{V}$. These loops allow an agent to remain at the same vertex during one time step.

Hence, although the original problem is posed on the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we extend it to the augmented graph $\overline{\mathcal{G}} = \left(\mathcal{V}, \overline{\mathcal{E}}\right)$, where $\overline{\mathcal{E}} = \mathcal{E} \cup \{(v, v) : v \in \mathcal{V}\}$. This guarantees that agents can remain stationary, and thus walks can always be synchronized. In particular, starting from any graph $\mathcal{G}$—with all, some, or no self-loops—we may equivalently work on $\overline{\mathcal{G}}$ without changing the set of feasible solutions.

With this construction, the walks of the k agents on $\mathcal{G}$ can be transformed into synchronized walks of the same length l on $\overline{\mathcal{G}}$. The remaining issue is to avoid conflicts among the agents, both in terms of vertex occupation and edge usage. This motivates the following definition.

Definition 3.2 (Parwaise Non-conflicting Walks)**.** Given a graph $\overline{\mathcal{G}} = \left(\mathcal{V}, \overline{\mathcal{E}}\right)$, let $\boldsymbol{\omega}^a = (\omega_0^a, \omega_1^a, \ldots, \omega_\ell^a)$ be a walk of length ℓ for each agent $a \in \mathcal{A}$. The collection of walks $\{\boldsymbol{\omega}^a\}_{a \in \mathcal{A}}$ is said to be *pairwise non-conflicting* if the following conditions hold:

1. For all time steps $i \in \{0, 1, \ldots, \ell\}$ and for all pairs of distinct agents $a, b \in \mathcal{A}$, it holds that
$$\omega_i^a \neq \omega_i^b \qquad \text{(no vertex conflict at any time step).}$$

2. For all time steps $i \in \{0, 1, \ldots, \ell - 1\}$ and for all pairs of distinct agents $a, b \in \mathcal{A}$, it holds that
$$(\omega_i^a, \omega_{i+1}^a) \neq (\omega_{i+1}^b, \omega_i^b) \qquad \text{(no edge-swapping conflict between consecutive steps).}$$

Equivalently, each walk can be expressed as a sequence of edges: $\boldsymbol{\omega}^a = (\omega_0^a, \omega_1^a, \ldots, \omega_\ell^a) = (e_1^a, \ldots, e_\ell^a)$ such that each $e_h^a = (\omega_{h-1}^a, \omega_h^a) \in \overline{\mathcal{E}}$. In this representation, the edge-swapping condition can be reformulated more compactly as:

2v. For all time steps $i \in \{1, \ldots, \ell\}$ and for all pairs of distinct agents $a, b \in \mathcal{A}$, it holds that

$$e_i^a \neq e_i^b \qquad \text{(no edge-swapping conflict between consecutive steps)}.$$

This suggest to introduce several key elements that will allow us to analyze the existence of a solution.

As in Ma and Koenig [2017], the original multi-agent path-finding problem can be formulated as a shortest-path problem on a graph whose vertices correspond to tuples of cells—one for each agent—thus defining a state-space graph of the system.

We define the configuration vector $\boldsymbol{z} \in \mathcal{V}^k$ as the vector $(z_1, \ldots, z_a, \ldots, z_k)$, where $z_a \in \mathcal{V}$ represents the position of the a-th agent at time t.

Definition 3.3 (Configuration Vector)**.** Given a pair $(\mathcal{G}, k)$, we define a *configuration vector* as a vector $\boldsymbol{z} \in \mathcal{V}^k$ such that $z_a \neq z_b$ for all $a \neq b \in \mathcal{A}$, ensuring that no two agents occupy the same vertex simultaneously.

By defining configuration vectors in this way, we avoid any conflicts arising from agents occupying the same vertex (vertex conflicts).

Next, we introduce the initial configuration vector $\boldsymbol{z} = \boldsymbol{s}$, which corresponds to the agents' starting positions.

The movement of the agents through the graph $\mathcal{G}$ can be interpreted as transitions between different configuration vectors. Thus, we define a new graph, called the configuration graph, where:

- Nodes represent feasible configuration vectors;
- Edges represent conflict-free transitions between configurations.

Definition 3.4 (Configuration Graph)**.** Given a pair $(\mathcal{G}, k)$, we define the *configuration graph*, denoted as $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$, as follows:

- The set of vertices (*Configuration Space*) is:

$$\mathcal{Z} = \left\{ \boldsymbol{z} \in \mathcal{V}^k \;\middle|\; z_a \neq z_b, \forall\, a, b \in \mathcal{A},\; a \neq b \right\}.$$

- The set of edges is:

$$\mathcal{F} = \left\{ (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{Z} \times \mathcal{Z} \;\middle|\; (z_a, z'_a) \in \overline{\mathcal{E}}, \forall\, a \in \mathcal{A}, \text{ and if } \exists b : z_b = z'_a \implies z'_b \neq z_a \right\}.$$

By defining edges between configuration vectors in this way, we also avoid any conflicts arising from swaps between agents (swapping conflicts).

Example 3.3. *Consider two examples of Configuration Graphs. The first example (Figure 3.3) is based on a simple cycle graph with four vertices and four agents. Thus, we consider the pair:* $(\mathcal{G}, k) = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}, 4)$.

The second example (Figure 3.4) is the same graph with an additional edge, while keeping the same number of agents. We consider the pair:

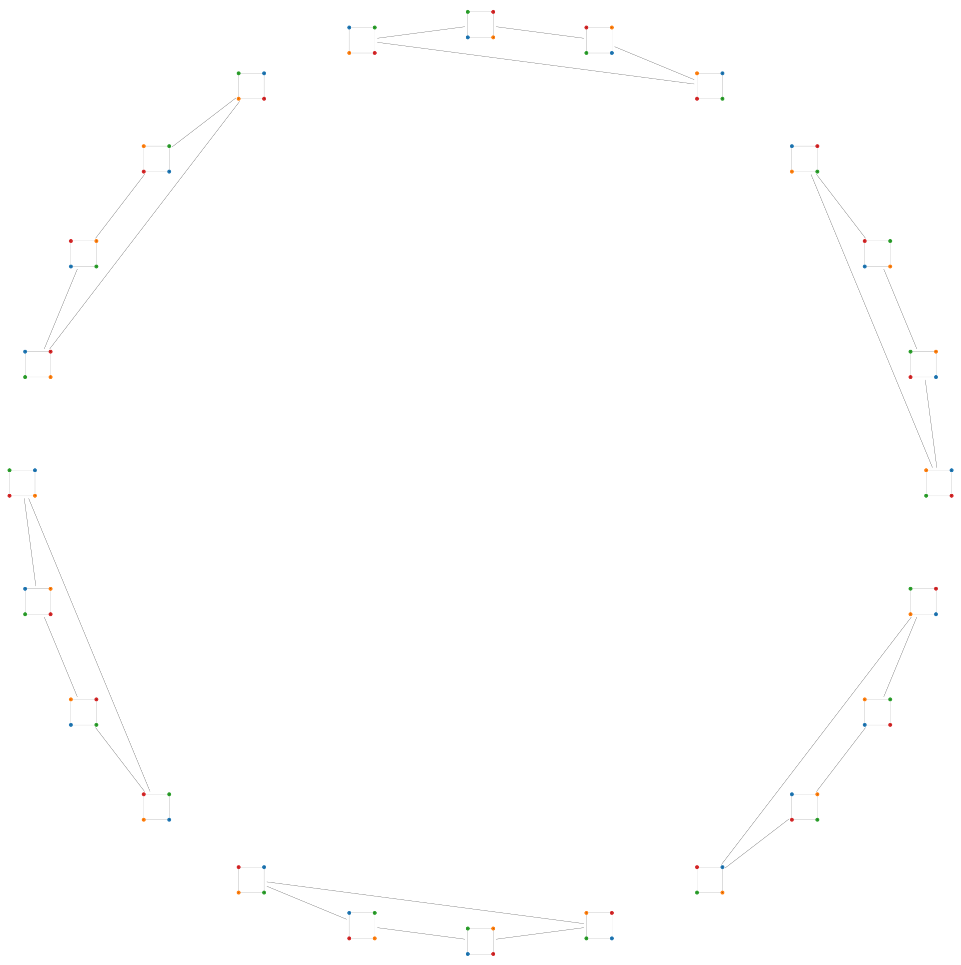


Figure 3.3: First example of a Configuration Graph.

$(\mathcal{G}, k) = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1), (v_2, v_4)\}, 4)$.

The additional edge (v_2, v_4) *increases the complexity of the Configuration Graph, generating a significantly larger number of connections. While the first Configuration Graph is relatively simple to visualize, the addition of just one edge significantly increases its complexity, creating numerous new connections between configuration vectors.*

Theorem 3.1. Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. If the graph $\mathcal{G}$ is undirected, then the configuration graph $\mathcal{H}$ is also undirected.

Proof. Consider an edge $(\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$. This implies that $(z_a, z'_a) \in \overline{\mathcal{E}}$, for all $a \in \mathcal{A}$, and if there exists b such that $z_b = z'_a$, then $z'_b \neq z_a$. We must show that $(\boldsymbol{z}', \boldsymbol{z}) \in \mathcal{F}$. Since $\mathcal{G}$ is

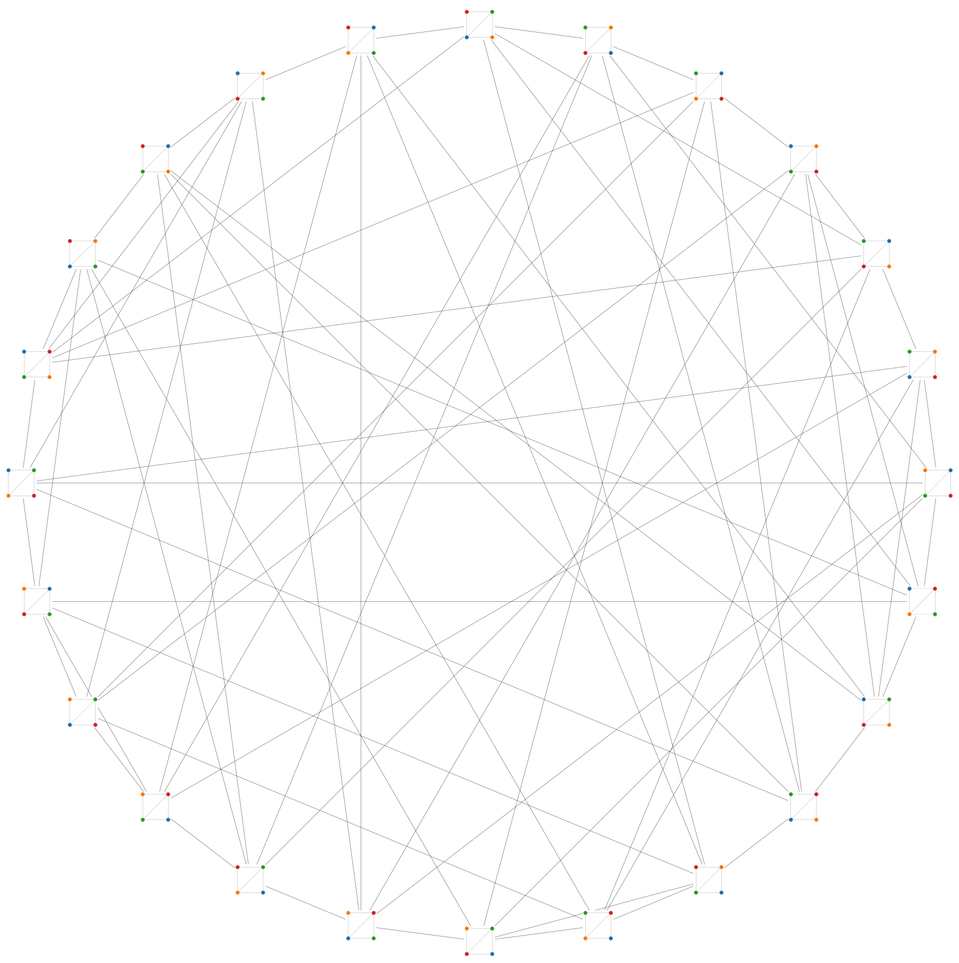


Figure 3.4: Second example of a Configuration Graph.

undirected, we have $(z'_a, z_a) \in \overline{\mathcal{E}}$ for all $a \in \mathcal{A}$, and the second condition is automatically satisfied by reversing the equality constraints. □

Consider an edge $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$ belonging to $\mathcal{F}$. We will now provide some characterizations of this edge, which will be useful in the subsequent analysis.

Definition 3.5 (Agent categories)**.** Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. Given an edge $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$, we define the following sets of agents:

- The set of *moving agents*, denoted by $M_f \subseteq \mathcal{A}$, is defined as
$$M_f = \{a \in \mathcal{A} \mid z_a \neq z'_a\} .$$
These are the agents whose positions change between configurations $\boldsymbol{z}$ and $\boldsymbol{z}'$.
- The set of *static agents*, denoted by $S_f \subseteq \mathcal{A}$, is defined as
$$S_f = \{a \in \mathcal{A} \mid z_a = z'_a\} .$$
These are the agents whose positions remain unchanged between $\boldsymbol{z}$ and $\boldsymbol{z}'$.

The sets M_f and S_f form a partition of the agent set $\mathcal{A}$, i.e., $\mathcal{A} = M_f \cup S_f$ with $M_f \cap S_f = \emptyset$.

Definition 3.6 (Edges Used)**.** Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. Given an edge $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$, we restrain to consider only the moving agents M_f and define the set of *edges used*, denoted by $E_f \subseteq \mathcal{E}$ as

$$E_f = \{(v, v') \in \mathcal{E} \mid \exists\ a \in M\ s.t.\ (v, v') = (z_a, z'_a)\}$$

The set E_f can be generae a restrain of $\mathcal{G}$: $\mathcal{G}\,[E_f]$. This graph include only the starting and ending posizion of the moving agents, with the edges useb by this agents in the mouvement $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$.

We will soon understand why the set E_f can be seen as a restrain, and what exactly it is a restriction of. However, we will need to wait until the next chapter to fully grasp why this concept is so significant.

Now we can consider the movements of the individual agents on the starting graph $\mathcal{G}$ in a synchronized manner, treating them as a sequence of configuration vectors, or equivalently, as a path on the configuration graph.

Definition 3.7 (Motion (vertices))**.** Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. An *motion* $\boldsymbol{\gamma}$ through $\mathcal{H}$ is a path on the configuration graph:

$$\boldsymbol{\gamma} = (\boldsymbol{z}^0, \boldsymbol{z}^1, \ldots, \boldsymbol{z}^i, \ldots, \boldsymbol{z}^l) \quad \text{such that} \quad (\boldsymbol{z}^i, \boldsymbol{z}^{i+1}) \in \mathcal{F} \quad \forall i \in \{1, 2, \ldots, l\} .$$

The length of an motion is the number of elements in the sequence minus 1, denoted as $|\boldsymbol{\gamma}| = l$.

For our purposes, it may be more useful to define a path in terms of a sequence of edges.

Definition 3.8 (Motion (edges))**.** Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. A *motion* $\boldsymbol{\gamma}$ through $\mathcal{H}$ is a sequence of edges in the configuration graph:

$$\boldsymbol{\gamma} = (f_1, f_2, \ldots, f_i, \ldots, f_l) \quad \text{such that} \quad f_i = (\boldsymbol{z}^i, \boldsymbol{z}^{i+1}) \in \mathcal{F} \quad \forall i \in \{1, 2, \ldots, l\} .$$

The length of a motion is the number of edges in the sequence, denoted as $|\boldsymbol{\gamma}| = l$.

An evolution can be decomposed into individual walks executed by the agents through the edges of the original graph $\mathcal{G}$, involving self-loops.

Every motion $\gamma = (f^1, \dots, f^l)$ where $f^t = (z^t, z^{t+1})$ through $\mathcal{H}$ can be naturally decomposed into k individual motions of the k agents, by considering $\gamma_a = (f_a^1, \dots, f_a^l)$ where $f_a^t = (z_a^t, z_a^{t+1})$ for $k = 1, \dots, k$.

We notice that, by construction, the k individual motions are actually walks in the augmented graph $\left(\mathcal{V}, \overline{\mathcal{E}}\right)$.

Moreover, again by construction, we noticed that the walks are pairwise non-conflicting.

We have the following result:

Theorem 3.2 (Representation Theorem of a motion)**.** Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. A sequence of $k = |\mathcal{A}|$ pairwise non-conflicting walks of length l, $\gamma_a = (f_a^1, \dots, f_a^l)$ with $a = 1, \dots, k$, on the augmented graph $\left(\mathcal{V}, \overline{\mathcal{E}}\right)$ is a motion $\boldsymbol{\gamma} = (f^1, f^2, \dots, f^i, \dots, f^l)$, where $f^t = (\boldsymbol{z}^t, \boldsymbol{z}^{t+1})$.

Proof. Consider for each $i = 1, \dots, l$ the vectors $\boldsymbol{z}^i$ defined by $z_a^i = \beta_a^i$ for all $a \in \mathcal{A}$ and $\boldsymbol{z}^0$ defined by $z_a^0 = \alpha_a^1$ for all $a \in \mathcal{A}$, where $f_a^{i+1} = (\alpha_a^i, \beta_a^i)$.

We have $\boldsymbol{z}^i \in \mathcal{Z}$ for each $i = 0, \dots, l$ because $\boldsymbol{z}^i \in \mathcal{V}^k$ and $z_a^i \neq z_b^i$ for all $a \neq b \in \mathcal{A}$, according to Property 1 of Non-conflicting Walks.

At this point, for each $i = 1, \dots, l$, we can conclude that $f^i = (\boldsymbol{z}^{i-1}, \boldsymbol{z}^i) \in \mathcal{F}$ because both $\boldsymbol{z}^{i-1}$ and $\boldsymbol{z}^i$ belong to $\mathcal{Z}$, and for all $a \in \mathcal{A}$, we have $(z_a^{i-1}, z_a^i) = (\alpha_a^i, \beta_a^i) = f_a^{i+1} \in \overline{\mathcal{E}}$. Moreover, if there exists $b \in \mathcal{A}$ such that $z_b^{i-1} = z_a^i$, then it must hold that $z_b^{i-1} \neq z_a^i$, again by Property 2 of Non-conflicting Walks. □

3.3 Successful Motion and performances

Definition 3.9 (Successful Motion)**.** Given a Matrix instance $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. A motion $\boldsymbol{\gamma} = (\boldsymbol{z}^0, \boldsymbol{z}^1, \dots, \boldsymbol{z}^l)$ of length l through $\mathcal{H}$ that satisfies the following conditions is called a *Successful Motion*:

1. $\boldsymbol{z}^0 = \boldsymbol{s}$ (the initial positions of the agents are given),
2. $z_a^l \in \mathcal{D}, \forall\ a \in \mathcal{A}$ (all agents stop in the goal set),
3. $\forall\ a \in \mathcal{A}^t$, there exists a time $\overline{s}_a$ such that $z_a^{\overline{s}_a} = o$ (task agents must pass through the origin).

A Successful Motion $\boldsymbol{\gamma}$ is a solution to the Matrix problem. We can define the *Successful Set* Γ as the collection of all Successful Motions $\boldsymbol{\gamma}$ through $\mathcal{H}$.

The first goal of this thesis is to understand when this space Γ is non-empty.

Let us introduce a generic definition:

Definition 3.10 (*D*-Time Vector)**.** Given a pair $(\mathcal{G}, k)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. Consider a motion $\boldsymbol{\gamma} = (\boldsymbol{z}^0, \boldsymbol{z}^1, \dots, \boldsymbol{z}^l)$ of length l through

$\mathcal{H}$ and a not-empty set $D \subseteq \mathcal{V}$ destination. A *D-Time Vector* $\boldsymbol{\sigma} \in \overline{\mathbb{R}}_{+}^{k}$ is a vector that collect the first time the a-th agent reaches the set D:

$$\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_k) \quad \text{such that} \quad z_a^{\sigma_a} \in D \text{ and } z_a^i \notin D, \forall\ i\ s.t.\ 0 \leq i < \sigma_a, \forall\ a \in \mathcal{A}$$

For convention, if $\exists\ a \in \mathcal{A}\ s.t.\ z_a^i \notin D, \forall\ i\ s.t.\ 0 \leq i \leq l$, we set $\sigma_a = +\infty$.

Given a motion $\boldsymbol{\gamma}$ through $\mathcal{H}$, for this definition, we can consider only the $\overline{k}$ tasked agents and define the *Origin-Time Vector* as the D-Time Vector for $D = \{o\}$, which we denote as $\overline{\boldsymbol{\sigma}} = (\overline{\sigma}_1, \ldots, \overline{\sigma}_{\overline{k}})$. We can now rephrase the three conditions of a Successful Motion:

$$\forall\ a \in \mathcal{A}^t, \overline{\sigma}_a \neq +\infty$$

When $\Gamma \neq \emptyset$, we can search for an Optimal Motion, which is a Successful Motion that satisfies additional conditions.

For example, in some applications, we may be interested in maximizing the Origin-Time Vector or minimizing the sum of the Origin-Time Vector.

Definition 3.11 (Origin-Costs)**.** Given a Matrix instance $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. Given a Successful Motion defined by $\boldsymbol{\gamma} = (\boldsymbol{z}^0, \boldsymbol{z}^1, \ldots, \boldsymbol{z}^l) \in \Gamma$ of length l through $\mathcal{H}$, we obtain an Origin-Time Vector $\overline{\boldsymbol{\sigma}}$ and can define:

- the *origin-cost* as
$$\overline{c}(\gamma) = \sum_{a \in \mathcal{A}^t} \overline{\sigma}_a,$$
- the *max-origin-cost* as
$$\overline{C}(\gamma) = \max_{a \in \mathcal{A}^t} \overline{\sigma}_a.$$

Since it is a Successful Motion, each $\overline{\sigma}_a$ is finite and strictly less or equal than l. Therefore, we note that the max-origin-cost can be at most l, i.e., $\overline{C}(\gamma) \leq l$.

Any Successful Motion $\boldsymbol{\gamma}$ that minimizes the origin-cost is a solution for the *Optimal Origin-Cost Problem*, and we define the set $\overline{\Gamma}_1 \subseteq \Gamma$ as the collection of all such *Optimal Origin-Cost Motions*.

Any Successful Motion $\boldsymbol{\gamma}$ that minimizes the max-origin-cost is a solution for the *Optimal Max-Origin-Cost Problem*, and we define the set $\overline{\Gamma}_{+\infty} \subseteq \Gamma$ as the collection of all such *Optimal Max-Origin-Cost Motions*.

Sometimes, it might be more interesting to analyze the time to reach the storage vertices $\mathcal{D}$. Given a motion $\boldsymbol{\gamma}$ through $\mathcal{H}$, according to the definition above, we can consider the k agents and define the *Stop-Time Vector* as the D-Time Vector for $\mathcal{D}$, denoted as $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_k)$. However, with the additional condition that $\rho_i > \sigma_i$ for all $i = 1, \ldots, k$, since the Task Agents must reach the origin before the storage.

Definition 3.12 (Stop-Costs)**.** Given a Matrix instance $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. Given a Successful Motion defined by $\boldsymbol{\gamma} = (\boldsymbol{z}^0, \boldsymbol{z}^1, \ldots, \boldsymbol{z}^l) \in \Gamma$ of length l through $\mathcal{H}$, we obtain a Stop-Time Vector $\boldsymbol{\rho}$ and can define:

- the *total-cost* as
$$\bar{c}(\gamma) = \sum_{a\in\mathcal{A}} \rho_a,$$
- the *max-cost* as
$$\overline{C}(\gamma) = \max_{a\in\mathcal{A}} \rho_a.$$

Since it is a Successful Motion, each ρ_a is finite and strictly strictly less or equal than l. Therefore, we note that the max-cost can be at most l, i.e., $C(\gamma) \leq l$.

Any Successful Motion $\boldsymbol{\gamma}$ that minimizes the total-cost is a solution for the *Optimal Total-Cost Problem*, and we define the set $\Gamma_1 \subseteq \Gamma$ as the collection of all such *Optimal Total-Cost Motions.*

Any Successful Motion $\boldsymbol{\gamma}$ that minimizes the max-cost is a solution for the *Optimal Max-Cost Problem*, and we define the set $\Gamma_{+\infty} \subseteq \Gamma$ as the collection of all such *Optimal Max-Cost Motions.*

The second goal is to understand when these spaces, $\overline{\Gamma}_1, \overline{\Gamma}_{+\infty}, \Gamma_1, \Gamma_{+\infty}$, are non-empty.

3.4 Complexity Considerations

After introducing the above elements, solving the problem corresponds to finding motion, a path in the Configuration Graph, starting from the initial node $\boldsymbol{s}$ and reaching a node in the set $\mathcal{O}$. The set $\mathcal{O}$ consists of all configuration vectors whose elements belong to $\mathcal{D}$ and such that at least one configuration vector is visited for each agent task starting from its origin.

This formulation raises two main issues.

Issue 1: Size of the Configuration Graph. The number of nodes of the graph (i.e., the number of distinct configuration vectors) is given by the number of permutations of n objects taken k at a time:
$$P(n,k) = \frac{n!}{(n-k)!}.$$
This quantity grows extremely fast with n and k. In particular, $P(n,n) = n!$ already explodes for small n, while for k close to n the order of magnitude is n^k. The construction of edges further exacerbates the problem, since no closed-form characterization is available: one must test pairs of configuration vectors to check whether a valid transition exists. A possible mitigation is to restrict the construction to the connected component of $\boldsymbol{s}$, generating vertices and edges on the fly through a graph search (DFS or BFS). However, no guarantee of substantial efficiency gain is provided.

Issue 2: Search for a valid path. Define sets S_i, with $0 < i \leq \overline{k}$, each containing the configuration vectors in which the i-th task agent is at its origin. To check for the existence of a solution, we employ a BFS over an extended state space using a bitmask representation. Each state is defined as:
$$\text{state} = (v, m),$$

where v is the current node, and m is a bitmask encoding which sets $S_1, \dots, S_{\overline{k}}$ have been visited. Each node v is preassigned a value $\text{bits}(v)$, the bitmask of sets it belongs to.

The initial state is

$$(\boldsymbol{s}, m_0), \qquad m_0 = \text{bits}(\boldsymbol{s}).$$

For each state (v, m): 1. visit all neighbors w of v; 2. compute the updated mask

$$m' = m \vee \text{bits}(w);$$

3. generate the new state (w, m'), enqueuing it if unseen.

A valid solution is found as soon as a state (t, m) is reached with

$$t \in \mathcal{D}, \qquad (m \,\&\, \text{required}) = \text{required},$$

where `required` denotes the mask with all mandatory tasks. Since BFS explores by increasing path length, the resulting path is also shortest, so we have the minimun max-origin-cost.

It's interessent to see the complessity of this algorithm. In a classical BFS over $G = (V, E)$, the number of visited states is $|V|$, with overall complexity

$$O(|V| + |E|).$$

In the extended formulation, a state is a pair (v, m) with

$$v \in V, \qquad m \in \{0, \dots, 2^{\overline{k}} - 1\},$$

yielding at most $|V| \cdot 2^{\overline{k}}$ states. Each state can generate at most $\deg(v)$ transitions, thus the overall time complexity is

$$O\big((|V| + |E|) \cdot 2^{\overline{k}}\big).$$

The number of nodes in the Configuration Graph is

$$|V| = P(n, k) = \frac{n!}{(n-k)!},$$

and the number of edges can be considered of the same order, i.e. $|E| \approx P(n, k)$. Hence, the BFS with bitmask explores at most

$$O\big(P(n, k) \cdot 2^{\overline{k}}\big), \qquad \overline{k} \leq k.$$

We now turn to alternative methods that may provide more efficient strategies for determining the existence of a solution to this problem.

Chapter 4

The Full Tasked Matrix Problem

4.1 Introduction

The simplest scenario to consider is the presence of task agents only.

Therefore, we begin our analysis with a special case in which no agent can move independently. Specifically, we consider the situation where every movement necessarily involves at least two agents moving in a coordinated manner. In practice, we will soon observe that the movement of a single agent within the graph often requires the simultaneous movement of at least one other agent. This observation motivates the introduction of the concept of a *rigid edge*, which formalizes the notion of synchronized movement.

Definition 4.1 (Rigid Edge)**.** Given a pair $(\mathcal{G}, k)$, let $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$ denote the associated configuration graph. An edge $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$ is called a *rigid edge* if the set of occupied vertices remains unchanged between $\boldsymbol{z}$ and $\boldsymbol{z}'$, that is,

$$\{v \in \mathcal{V} \mid \exists a \in \mathcal{A} \text{ s.t. } z_a = v\} = \{v \in \mathcal{V} \mid \exists a \in \mathcal{A} \text{ s.t. } z'_a = v\}\,.$$

In other words, a transition is said to be *rigid* if it involves the coordinated movement of multiple agents—no single agent can perform such a transition in isolation.

One way to verify this is that the set of vertices of $\mathcal{G}$ associated with $\boldsymbol{z}|_{M_f}$, namely the positions of the agents in the initial configuration, coincides with the set of vertices of $\mathcal{G}$ associated with $\boldsymbol{z}'|_{M_f}$.

Definition 4.2 (Rigid Configuration)**.** Given a pair $(\mathcal{G}, k)$, let $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$ be the associated configuration graph. A configuration vector $\boldsymbol{z} \in \mathcal{Z}$ is called a *rigid configuration* if all outgoing transitions $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$ are rigid edges.

Definition 4.3 (Rigid Configuration)**.** Given a pair $(\mathcal{G}, k)$, let $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$ be a configuration graph. A configuration $\boldsymbol{z} \in \mathcal{Z}$ is called a *rigid configuration* if

1. there exists at least one $\boldsymbol{z}' \neq \boldsymbol{z}$ such that $(\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$, and

2. every such transition $(\boldsymbol{z}, \boldsymbol{z}')$ is a rigid edge.

That is, in a rigid configuration, no agent is able to move independently; every possible transition necessarily requires synchronized movement of at least two agents. This concept will be central to our analysis of limiting behaviors in multi-agent dynamics.

Theorem 4.1. *$k = n$ if and only if every configuration is rigid.*

Proof. $\Rightarrow$ Suppose $k = n$. Then each vertex in the graph is occupied by exactly one agent. For any configuration $\boldsymbol{z}$, there are no unoccupied vertices. Thus, the set of occupied vertices is equal to the entire vertex set:

$$\{v \in \mathcal{V} \mid \exists a \in \mathcal{A} \text{ s.t. } z_a = v\} = \mathcal{V}.$$

In this case, no agent can move without displacing another. Therefore, every valid transition must involve multiple agents, and hence every transition is rigid.

$\Leftarrow$ We observe that $\mathcal{G}$ cannot consist solely of isolated vertices, since in that case no configuration would be rigid: property (1) in Definition 4.3 would always fail. Hence, there must exist at least two distinct vertices $u, v \in \mathcal{V}$ with $(u, v) \in \mathcal{E}$. Suppose, for the sake of contradiction, that $k < n$. Then there exists a configuration $\boldsymbol{z}$ in which u is occupied by some agent a, while v is unoccupied. The simple move of agent a from u to v, with all other agents remaining fixed, yields a non-rigid edge $(\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$, contradicting the assumption that $\boldsymbol{z}$ was a rigid configuration.

□

Observation 2. *If there exists a non-rigid edge, then necessarily $k \neq n$.*

Observation 3. *Consider the case where the number of agents k equals the number of vertices $n = |\mathcal{V}|$. In this situation, it is natural to assume that the set of target vertices $\mathcal{D}$ coincides with the entire vertex set, i.e.,*

$$\mathcal{D} = \mathcal{V}.$$

Consequently, the specification of the final position for each agent becomes irrelevant, since every vertex is both occupied initially and targeted eventually.

Simultaneously, the set of initial positions also coincides with the full vertex set:

$$\boldsymbol{s} = \mathcal{V}.$$

Under this assumption, if all vertices correspond to tasks, the precise ordering of agents visiting each vertex is immaterial. Alternatively, if some vertices do not correspond to tasks, one can equivalently partition the vertex set into tasked and untasked vertices, focusing only on the movement to the tasked subset. In either scenario, the problem is simplified, as we only need to ensure that each agent eventually occupies a distinct tasked vertex.

For these reasons, the natural starting point for analysis is the Fully-Agents and Purely-Tasked Case, *defined by*

$$n = k = \overline{k},$$

where $\overline{k}$ denotes the number of tasks. In this setting, every agent has a dedicated task, every vertex is both a start and a target, and the combinatorial complexity associated with partial assignments or vertex orderings is eliminated. This scenario therefore provides the simplest nontrivial instance from which more general configurations can be studied systematically.

Before proceeding further, let us clarify the relevance of rigid edges within the configuration graph framework. In particular, we show how their structure is intrinsically related to cycles in the underlying graph.

Theorem 4.2 (Allowed Movements)**.** Given a pair $(\mathcal{G}, k)$, let $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$ be the associated configuration graph. Let $f = (\boldsymbol{z}, \boldsymbol{z}') \in \mathcal{F}$ be a rigid edge, and let $E_f \subseteq \mathcal{E}$ be the set of edges in $\mathcal{G}$ traversed during the transition f.

Then the subgraph $\mathcal{G}[E_f]$ is a collection of vertex-disjoint directed cycles. More precisely, for each vertex v in $\mathcal{G}[E_f]$, there exists a unique directed cycle C_v such that $v \in C_v$.

Proof. Let $m = |M_f|$ be the number of agents involved in the transition f. Since f is rigid, each agent moves from one vertex to another without altering the overall set of occupied vertices. Hence, there are m directed edges in E_f, one for each moving agent.

For each vertex $v \in \mathcal{G}[E_f]$, there exists exactly one agent departing from v in configuration $\boldsymbol{z}$ and exactly one agent arriving at v in configuration $\boldsymbol{z}'$. Therefore, each vertex in $\mathcal{G}[E_f]$ has in-degree 1 and out-degree 1.

This structure implies that the subgraph $\mathcal{G}[E_f]$ is a union of directed cycles. Indeed, starting from any vertex and iteratively following its unique outgoing edge necessarily yields a directed cycle, and different cycles are vertex-disjoint. □

Remark. When $k = n$, that is, when all vertices of $\mathcal{G}$ are occupied by agents, every movement corresponds to a rigid transition. In this case, all transitions in the configuration graph necessarily induce disjoint directed cycles, and no single agent can move independently.

4.2 Successful Motion

The aim of this section is to prove the following result:

Theorem 4.3. Given a Matrix instance $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$, consider the associated configuration graph $\mathcal{H} = (\mathcal{Z}, \mathcal{F})$. If $\mathcal{G}$ is undirected and $n = k = \overline{k}$, then the successful set Γ is non-empty if $\mathcal{G}$ is 2-edge-connected.

The case $\mathcal{G}$ directed is more complicated...

We start to observe that in an undirected graph any path γ from o to v can always be reversed to obtain a path from v to o.

Lemma 4.1. Let $\mathcal{G}$ be a connected undirected graph such that every edge of $\mathcal{G}$ belongs to at least one cycle (equivalently, $\mathcal{G}$ is the union of cycles). Then $\mathcal{G}$ is 2-edge-connected.

Proof. Since $\mathcal{G}$ is connected by hypothesis, to prove 2-edge-connectivity it suffices to show that $\mathcal{G}$ has no bridges. Let $e = (x, y)$ be an arbitrary edge of G. By hypothesis there exists a cycle C of $\mathcal{G}$ containing e. Hence there is a closed walk in $\mathcal{G}$ that traverses e and returns from y to x along the remainder of C. In particular, there exists an x–y path in $\mathcal{G}$ that does not use the edge e (namely, the arc of C complementary to e). Therefore the removal of e does not separate x and y.

We must show that removal of e does not disconnect any two arbitrary vertices $u, v \in \mathcal{V}$. Take any $u, v \in \mathcal{V}$ and let γ be a u-v path in $\mathcal{G}$ (exists because $\mathcal{G}$ is connected). If γ does not contain e, then γ remains a u-v path in $\mathcal{G} \setminus \{e\}$ and the pair (u, v) stays connected. If γ does contain e, write $\gamma = \gamma_u + (x, y) + \gamma_v$ where γ_u is a u-x subpath and γ_v is a y-v subpath. Replace the edge $(x, y) = e$ in γ by the alternative x-y path provided by the cycle C that contains e. The resulting walk from u to v avoids e and thus is a path in $\mathcal{G} \setminus \{e\}$ (after removing possible repeated vertices along the walk). Hence any two vertices that were connected in $\mathcal{G}$ remain connected in $\mathcal{G} \setminus \{e\}$.

Because e was arbitrary, no edge of $\mathcal{G}$ is a bridge. Therefore $\mathcal{G}$ is bridgeless and, being connected, G is 2-edge-connected. □

With this final remark, we now have all the necessary ingredients to present the complete proof of the theorem stated above.

Proof. $\Rightarrow$ Assume that there exists a successful motion

$$\Gamma = (f_1, \dots, f_\ell),$$

We decompose Γ into k individual agent walks

$$w_1, \dots, w_k.$$

Every configuration is rigid, so $k = n$. Fix an arbitrary vertex $v \in \mathcal{V}$ and let a be the agent with initial position $w_a^0 = v$. Since Γ is successful, a reaches the target o at some finite time t_a, hence the walk of a contains a v-o walk. By removing repeated vertices we obtain a simple v-o path

$$\gamma_v = (v_0 = v, v_1, \dots, v_r = o).$$

By construction, every edge of γ is traversed at some time by a agent during some transition f_j. By hypothesis each transition f_j decomposes into a disjoint union of cycles; therefore for every edge e of γ_v there exists at least one cycle (arising from some transition) that contains e. Consequently, as we traverse γ from v to o, each contiguous maximal block of edges of γ that belongs to the same transition-cycle determines a cycle in the family. Grouping these blocks yields a finite sequence of cycles

$$C_1, \dots, C_m$$

such that

$$v \in V(C_1), \qquad o \in V(C_m), \qquad C_i \cap C_{i+1} \neq \varnothing \text{ for } i = 1, \dots, m-1.$$

(Indeed, consecutive edges of γ either belong to the same cycle or to two cycles that intersect at the intermediate vertex; collecting maximal contiguous segments belonging to the same cycle produces the asserted chain.)

Applying Lemma 4.1 to the chain $C_1, \ldots, C_m$ yields two edge-disjoint v-o paths in the underlying undirected graph $\mathcal{G}$. Since v was arbitrary, we obtain that for every vertex there are two edge-disjoint paths to o; hence no edge is a bridge. Therefore $\mathcal{G}$ is bridgeless and, being connected by hypothesis, it is 2-edge-connected.

$\Leftarrow$ Suppose that $\mathcal{G}$ is not 2-edge connected. Then there exists a bridge $(u, v) \in \mathcal{E}$. Removing this edge partitions the vertex set into two disjoint subsets V_u and V_v, where V_u (resp. V_v) contains all vertices connected to u (resp. v) without traversing (u, v) or (v, u). Without loss of generality, assume that $o \in V_v$.

In any successful motion γ, the edge (u, v) must eventually be traversed, otherwise all agents initially located in V_u could never reach o. Consequently, there exist two consecutive configurations $\boldsymbol{z}, \boldsymbol{z}'$ along γ such that $(\boldsymbol{z}, \boldsymbol{z}')$ involves the edge (u, v). By Theorem 3.1, the configuration $\boldsymbol{z}$ is rigid. However, this contradicts Theorem 3.2, since the bridge (u, v) does not belong to any cycle.

Therefore, bridges cannot exist in $\mathcal{G}$, and the graph must be 2-edge connected.

□

We present two complementary approaches to determine whether it is 2-edge-connected: a flow-based method and a depth-first search (DFS) based method using lowpoints.

Flow-based verification

Let us first describe a method based on maximum flow, which is applicable to both directed and undirected graphs.

Construction. Fix an arbitrary vertex $o \in \mathcal{V}$. For each vertex $v \in \mathcal{V} \setminus \{o\}$, consider the following:

1. Transform $\mathcal{G}$ into a flow network: replace each edge with two oppositely directed arcs and assigning capacity 1 to each edge.

2. Compute the maximum flow $f_{\max}(v, o)$ from v to o in this network.

Lemma 4.2 (Flow characterization of 2-edge-connectivity)**.** $\mathcal{G}$ is 2-edge-connected if and only if $f_{\max}(v, o) \geq 2$ for all $v \in \mathcal{V} \setminus \{o\}$.

Proof. ($\Rightarrow$) If $\mathcal{G}$ is 2-edge-connected, Menger's theorem guarantees at least two edge-disjoint paths from v to o. Each path can carry one unit of flow, hence $f_{\max}(v, o) \geq 2$.

($\Leftarrow$) Conversely, if $f_{\max}(v, o) \geq 2$ for all v, then no single edge separates v from o. Since this holds for all v, $\mathcal{G}$ contains no bridges and is connected, thus it is 2-edge-connected. □

Complexity. Computing $n-1$ maximum flows (one for each $v \neq o$) has complexity $O(n^2m^2)$ using Edmonds-Karp, or $O(n^{3/2}m)$ using Dinic for unit-capacity graphs. While general, this approach is comparatively costly and is mainly of theoretical interest or for small graphs.

DFS and lowpoint-based verification

For undirected graphs, a more efficient approach exploits depth-first search (DFS) and the concept of lowpoints. This method runs in linear time and directly identifies bridges.

Lowpoints. Perform a DFS of $\mathcal{G}$ starting from an arbitrary vertex, numbering vertices in order of discovery. For each vertex v, define its lowpoint as the minimum discovery number reachable from v through any descendant in the DFS tree or via a back edge.

Bridge identification. An edge (u, v) (with u parent of v in the DFS tree) is a bridge if and only if

$$\mathrm{low}(v) > \mathrm{disc}(u),$$

where $\mathrm{disc}(u)$ is the discovery number of u.

Lemma 4.3 (DFS characterization of 2-edge-connectivity)**.** *An undirected connected graph $\mathcal{G}$ is 2-edge-connected if and only if its DFS contains no bridge edges.*

Proof. If $\mathcal{G}$ contains a bridge, removing it disconnects the graph, so it cannot be 2-edge-connected. Conversely, if the DFS finds no bridges and the graph is connected, every edge belongs to a cycle or is otherwise not a cut-edge; hence $\mathcal{G}$ is 2-edge-connected. □

Complexity. The DFS-based approach requires a single DFS traversal and lowpoint computation, which can be performed in $O(|\mathcal{V}| + |\mathcal{E}|)$ time, i.e., linear in the size of the graph. This makes it optimal for large undirected graphs.

Practical recommendation. - For undirected graphs, the DFS + lowpoint method is preferable due to its simplicity and linear complexity. - The flow-based method is more general and can handle directed graphs or graphs with capacity constraints, but at a higher computational cost.

4.3 Minimal solution

The minimal solution, in the case $n = k = \overline{k}$, requires a motion of length $n-1$. Indeed, for each configuration vector $\boldsymbol{z}^0, \ldots, \boldsymbol{z}^l$ of the motion, there can be at most one agent in the origin position. Therefore, if we have $\overline{k} = n$ task agents, we need at least n distinct configuration vectors, one for each task agent, in order to satisfy the requirements of the Matrix Problem for task agents. We call this solution the *Minimal Solution*: at each step, a task agent that has not yet reached the origin moves to the origin.

If it exists, the Minimal Solution is also an *Optimal max-origin-cost motion*; at the same time, it is an *Optimal origin-cost motion.* In fact, when minimizing the sum of the arrival times to the origin, no better solution can exist than the minimum achievable for each individual agent.

Finding a necessary and sufficient condition on the graph topology that guarantees the existence of a Minimal Solution is a highly nontrivial problem. Let us analyze why.

A first observation is that if the graph admits a Hamiltonian cycle, then a Minimal Solution certainly exists.

Theorem 4.4. Given a Matrix instance $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$, if the graph $\mathcal{G}$ admits a Hamiltonian cycle, then there exists a Minimal Solution.

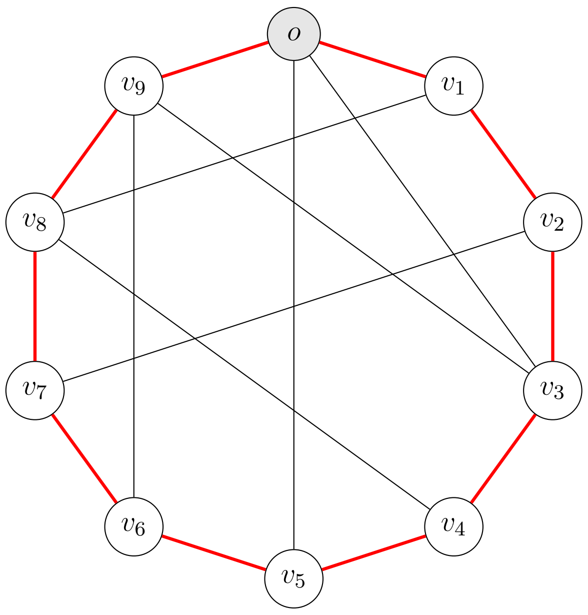
Proof. Assume that $\mathcal{G}$ contains a Hamiltonian cycle C. Fix an orientation of C (e.g. clockwise) and consider synchronous steps in which every agent moves along C by one edge at each step.

Since C is Hamiltonian, each agent visits every vertex of $\mathcal{G}$ exactly once before returning to its starting point. In particular, the distinguished vertex o (the origin) lies on C, so each agent visits o exactly once: in every step exactly one task agent that has not yet reached o arrives at o, and no agent reaches o more than once before all have reached it. Therefore, the induced motion yields a schedule in which, at each step, exactly one previously unreached task agent moves to the origin, which is—by definition—a Minimal Solution. $\square$

Example 4.1. *Consider the graph G with nodes $\{o, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$ and several connecting edges. Although the graph has many possible paths, it admits the following Hamiltonian cycle:*

$$o \to v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7 \to v_8 \to v_9 \to o.$$

By rotating agents along this Hamiltonian cycle, we obtain a Minimal Solution of length $n - 1 = 9$.



When one wishes to verify whether a Hamiltonian cycle exists in a given graph $G = (V, E)$, the most direct approach is to use a recursive search with backtracking. This method systematically explores candidate paths and checks whether they can be extended to a cycle visiting all vertices exactly once. Although the problem is NP-complete, this procedure provides a constructive way to either identify a Hamiltonian cycle or prove its absence.

Algorithm (Backtracking Approach) The procedure is as follows:

1. Start from a chosen vertex o and initialize the path $P = [o]$.
2. At each step, extend P by adding a neighbor v of the last vertex in P, provided v has not been visited before.
3. If the path reaches length $|V|$ and the last vertex is adjacent to o, then P forms a Hamiltonian cycle.
4. If no further extension is possible, backtrack and try an alternative choice.

To reduce the search space, simple pruning rules can be applied, e.g. discarding any vertex of degree smaller than 2, since it cannot belong to a Hamiltonian cycle.

Complexity Analysis The worst-case complexity of the backtracking algorithm is

$$O(|V|!),$$

since it may need to explore all permutations of vertices. In practice, however, pruning rules and structural properties of the graph (e.g. sparsity) significantly reduce the search effort. For small or medium-sized graphs this method is feasible, while for larger instances heuristic or approximation algorithms are preferred.

However, this is not a necessary condition. Indeed, even if there is no single Hamiltonian cycle but instead several cycles that share the origin node a Minimal Solution may still exist.

Theorem 4.5. Given a Matrix instance $\left(\mathcal{G}, o, k, \overline{k}, \boldsymbol{s}, \mathcal{D}\right)$, if for every node $v \neq o$ there exists a cycle containing both v and the origin o, then a Minimal Solution exists.

To proceed toward the proof of this theorem, we make the following observation, which is both valid and relevant for future developments: Once a task agent passes through the origin o, it is transformed into an untask agent and, from that moment onward, moves according to the dynamics of an untask agent.

Before proceeding to the proof, we make the following observation, which will be useful for the construction: once a task agent passes through the origin o, it is transformed into an untask agent and subsequently moves according to the dynamics of untask agents.

This means that a solution is a Minimal Solution if and only if at each step an agent moves from a task to an untask.

Example 4.2. *To complete Example A.2, the motion can be described as the sequence of nodes occupied by untask agents at each configuration vector:*

$$
\begin{aligned}
&0 : o\\
&1 : o, v_9\\
&2 : o, v_9, v_8\\
&3 : o, v_9, v_8, v_7\\
&4 : o, v_9, v_8, v_7, v_6\\
&5 : o, v_9, v_8, v_7, v_6, v_5\\
&6 : o, v_9, v_8, v_7, v_6, v_5, v_4\\
&7 : o, v_9, v_8, v_7, v_6, v_5, v_4 v_3\\
&8 : o, v_9, v_8, v_7, v_6, v_5, v_4 v_3, v_2\\
&9 : o, v_9, v_8, v_7, v_6, v_5, v_4 v_3, v_2, v_1
\end{aligned}
$$

Proof. We constructively prove the existence of a Minimal Solution.

Consider the set of all distinct cycles C_v, where C_v is the cycle containing o and v, which exists by hypothesis for each node $v \neq o$. Choose an initial cycle C_i.

Step 1. If C_i were the only cycle, the strategy would be to rotate agents along C_i in a fixed direction, as in the Hamiltonian cycle case. We consider this as a partial solution, represented by a list of configuration vectors. Each full rotation of $|C_i| - 1$ steps converts one task agent into an untask agent. We identify each configuration vector of the partial solution with the set of nodes occupied by untask agents. At each step, the positions of all task agents on C_i remain connected, and there always exists an edge linking at least one of them to the origin o.

Step 2. Consider the last element of the partial solution, excluding o; denote this set by L, and consider cycles sharing nodes with L. If no such cycle exists, select another cycle C_k and repeat Step 1, appending the new partial solution to the one obtained so far. In this case, we first rotate all agents along $C_i \setminus \{o\}$, then along $C_k \setminus \{o\}$, so that all task agents are converted into untask agents. Note that the order of choosing cycles is arbitrary: rotations along C_i followed by those on C_k, or vice versa, lead to the same outcome. Moreover, rotations along C_k can be interleaved with those on C_i without affecting the result.

Otherwise, let C_j be a cycle that shares nodes with L. Create a list of shared nodes in the order they appear along L, starting with o: $o, v_1, v_2, \dots$.

Step 3. Initialize a new partial solution as an empty list. For each $v_i = v_1, v_2, \dots$ in order:

- Append to the new partial solution the configuration vectors not yet adding of the old solution that do not include v_i as node occupied by untask agents.
- For each preceding element v_j in the list (in order starting from o), check whether C_j contains a path from v_i to v_j that non include other element of L.
- If such a path exists, construct a combined cycle as follows:
  1. In the subgraph induced by the untask nodes plus v_i, there exists a path from o to v_i (otherwise v_i could not appear as untask in the next configuration vector of the old solution).
  2. The path from v_i to v_j along C_j.
  3. The path from v_j to o consisting only of task nodes, which exists by construction.
- Perform rotations along this combined cycle until all nodes along the path from v_j to v_i (excluding v_i) are converted into untask agents. Notice that at each step, the positions of all task agents on C_i remain connected, and there always exists an edge linking at least one of them to the origin o.

Finally, append to the new partial solution the remaining configuration vectors from the old solution. Note that the rotations between two consecutive v_i (which belong to different cycles) can be rearranged freely without affecting the overall outcome.

This procedure constructs a sequence of rotations achieving a Minimal Solution. □

Example 4.3. *Let G be the graph shown in Figure 4.1. Its vertex set is*

$$V = \{o, v_1, \dots, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{16}\},$$

and its edges contain the Hamiltonian ring on $\{o, v_1, \dots, v_9\}$ together with the additional edges

$$(o, v_3),\ (o, v_5),\ (v_2, v_7),\ (v_4, v_8),\ (v_6, v_9),\ (v_1, v_8),\ (v_3, v_9)$$

and the external/internal attachments

$$(o, v_{10}),\ (v_{10}, v_7),\ (v_{11}, v_7),\ (v_{11}, v_9),\ (v_{12}, v_5),\ (v_{12}, v_9),\ (v_{13}, v_5),\ (v_{13}, o),$$

$$(v_{14}, o),\ (v_{14}, v_{15}),\ (v_{15}, v_{16}),\ (v_{16}, o)$$

We choose three cycles as building blocks for the constructive algorithm:

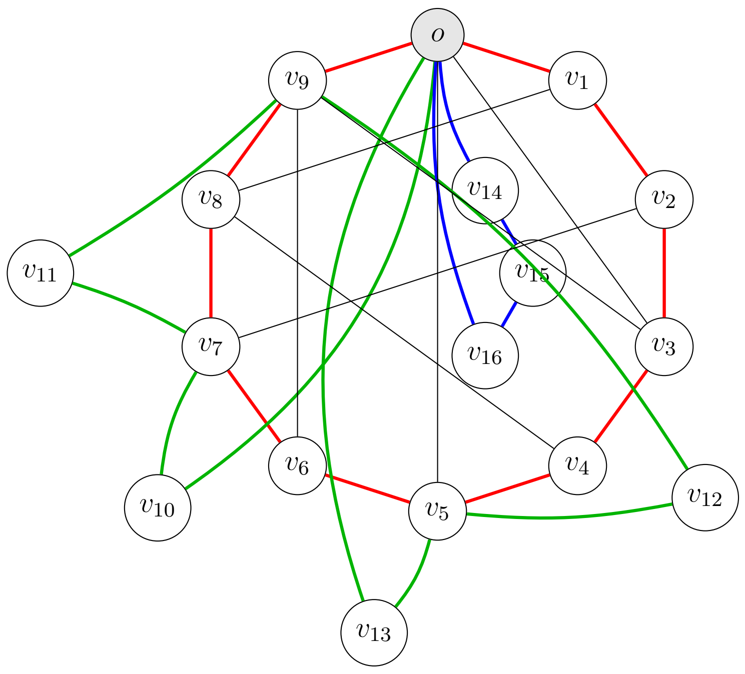


Figure 4.1: Graph with the three starting cycles.

- ***Red cycle*** $C_{red} = (o, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, o)$.
- ***Green cycle*** $C_{green} = (o, v_{10}, v_7, v_{11}, v_9, v_{12}, v_5, v_{13}, o)$.
- ***Blue cycle*** $C_{blue} = (o, v_{14}, v_{15}, v_{16}, o)$.

All nodes of G belong to at least one of these cycles (by construction), so the hypothesis of the theorem is satisfied.

At time 0 only the origin is untask:

$$0 : \{o\}.$$

Step 1: Use the Red cycle). *We start by applying Step 1 of the algorithm on the red cycle:*

$$C_{red} = (o, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, o).$$

We rotate agents along C_{red} in a fixed direction, converting one node into untask at each full rotation through the origin. The partial solution after applying red rotations step by step:

$$\begin{aligned}
&0 : \{o\}\\
&1 : \{o, v_9\}\\
&2 : \{o, v_9, v_8\}\\
&3 : \{o, v_9, v_8, v_7\}\\
&4 : \{o, v_9, v_8, v_7, v_6\}\\
&5 : \{o, v_9, v_8, v_7, v_6, v_5\}\\
&6 : \{o, v_9, v_8, v_7, v_6, v_5, v_4\}\\
&7 : \{o, v_9, v_8, v_7, v_6, v_5, v_4, v_3\}\\
&8 : \{o, v_9, v_8, v_7, v_6, v_5, v_4, v_3, v_2\}\\
&9 : \{o, v_9, v_8, v_7, v_6, v_5, v_4, v_3, v_2, v_1\}
\end{aligned}$$

Step 2: Identify cycles sharing nodes with the partial solution.
Current partial solution after Step 3 (red + green cycles):

$$last\_partial\_solution = \{o, v_9, v_8, v_7, v_6, v_5, v_4, v_3, v_2, v_1\}.$$

We examine the remaining cycles:

- *Blue cycle has no nodes in common with the current partial solution.*
- *Green cycle shares nodes with the red cycle: v_9, v_7, v_5.*

Construct the list of shared nodes in order along the current partial solution, including the origin:

$$L = \{o, v_9, v_7, v_5\}.$$

Step 3: Construct the new partial solution.
Initialize the new partial solution:

$$new\_solution = []$$

Iteration over $v_i \in L \setminus \{o\}$*:*

- $v_i = v_9$

  Add to new_solution the configuration vectors from the old solution that do not include v_9.

  $$0 : \{o\}.$$

  Check paths in C_{green} from v_9 to preceding nodes in L:

  o**.** *not exists*

- $v_i = v_7$

  Add old configuration vectors that do not include v_7.

  $$\begin{aligned}&1 : \{o, v_9\}\\ &2 : \{o, v_9, v_8\}\end{aligned}$$

  Check paths in C_{green} from v_9 to preceding nodes in L:

o. (o, v_{10}, v_7)

Construct combined cycle: $(o) + (o, v_{10}, v_7) + (v_7, v_6, v_5, v_4, v_3, v_2, v_1)$

$$3 : \{o, v_9, v_8, v_{10}\}$$

v_9. (v_9, v_{11}, v_7)

Construct combined cycle: $(o, v_9) + (v_9, v_{11}, v_7) + (v_7, v_6, v_5, v_4, v_3, v_2, v_1)$

$$4 : \{o, v_9, v_8, v_{10}, v_{11}\}$$

- $v_i = v_5$

  Add old configuration vectors that do not include v_5.

$$5 : \{o, v_9, v_8, v_{10}, v_{11}, v_7\}$$
$$6 : \{o, v_9, v_8, v_{10}, v_{11}, v_7, v_6\}$$

Check paths in C_{green} *from* v_5 *to preceding nodes in L:*

o. (o, v_{13}, v_5)

Construct combined cycle: $(o) + (o, v_{13}, v_5) + (v_5, v_4, v_3, v_2, v_1)$

$$7 : \{o, v_9, v_8, v_{10}, v_{11}, v_7, v_6, v_{13}\}$$

v_9. (v_9, v_{12}, v_5)

Construct combined cycle: $(o, v_9) + (v_9, v_{12}, v_5) + (v_5, v_4, v_3, v_2, v_1)$

$$8 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}\}$$

v_7. *not exists*

Complete with the remaing:

$$9 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5\}$$
$$10 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4\}$$
$$11 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3\}$$
$$12 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2\}$$
$$13 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1\}$$

Step 2: Identify cycles sharing nodes with the partial solution.
Current partial solution after Step 3 (red + green cycles):

$$partial\_solution = \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1\}.$$

We examine the remaining cycles:

- *Blue cycle has no nodes in common with the current partial solution.*

Step 1 (Blue cycle):
We continue by applying Step 1 of the algorithm on the blue cycle:

$$C_{blue} = (o, v_{14}, v_{15}, v_{16}, o).$$

We rotate agents along C_{blue} in a fixed direction, converting one node into untask at each full rotation through the origin. The partial solution after applying blue rotations step by step:

$0 : \{o\}$
$1 : \{o, v_9\}$
$2 : \{o, v_9, v_8\}$
$3 : \{o, v_9, v_8, v_{10}\}$
$4 : \{o, v_9, v_8, v_{10}, v_{11}\}$
$5 : \{o, v_9, v_8, v_{10}, v_{11}, v_7\}$
$6 : \{o, v_9, v_8, v_{10}, v_{11}, v_7, v_6\}$
$7 : \{o, v_9, v_8, v_{10}, v_{11}, v_7, v_6, v_{13}\}$
$8 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}\}$
$9 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5\}$
$10 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4\}$
$11 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3\}$
$12 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2\}$
$13 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1\}$
$14 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1, v_{14}\}$
$15 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1, v_{14}, v_{15}\}$
$16 : \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1, v_{14}, v_{15}, v_{16}\}$

Step 2: Identify cycles sharing nodes with the partial solution.
Current partial solution after Step 3 (red + green cycles):

$$partial\_solution = \{o, v_9, v_8, v_{10}, , v_{11}, v_7, v_6, v_{13}, v_{12}, v_5, v_4, v_3, v_2, v_1, v_{14}, v_{15}, v_{16}\}.$$

We examine the remaining cycles: there isn't, so we stop.

Algorithm for Biconnected Components Now the problem addressed here is the following: for every vertex $v \in \mathcal{V} \setminus \{o\}$ decide whether there exists a simple cycle of $\mathcal{G}$ that contains both o and v.

We present a conceptually simple and asymptotically optimal method tailored to undirected graphs, based on the decomposition of $\mathcal{G}$ into biconnected components (also called blocks).

Definition 4.4. A *biconnected component* (or *block*) of $\mathcal{G}$ is a maximal 2-vertex-connected subgraph.

Definition 4.5. A vertex $a \in \mathcal{V}$ is an *articulation point* (or cut vertex) if the removal of a increases the number of connected components of G.

Observation 4. *If $\mathcal{H}$ is a 2-vertex-connected subgraph and $x, y \in \mathcal{V}(\mathcal{H})$ are two distinct vertices, then there exists a simple cycle of $\mathcal{H}$ that contains both x and y. Consequently, two vertices of $\mathcal{G}$ belong to a common simple cycle if and only if they belong to a common biconnected component.*

Algorithmic strategy.

1. Compute the biconnected components of $\mathcal{G}$ using a single-depth first search (DFS) variant commonly known as Tarjan's algorithm for biconnected components.

2. Let $\{\mathcal{B}_i\}_{i=1}^{m}$ denote the blocks that contain the origin o. For each vertex $v \neq o$ answer true if v belongs to one block $\mathcal{B}_i$ and false otherwise.

During the DFS, two integer labels are associated with each vertex u:

- The discovery time $\mathrm{disc}[u]$ is the time step at which u is first visited. Each vertex receives a distinct discovery index that reflects the order of exploration.

- The low-link value $\mathrm{low}[u]$ is defined as the minimum among:

  – $\mathrm{disc}[u]$ itself;

  – the discovery times of all ancestors of u that are reachable from u through a back edge;

  – the low-link values of all DFS descendants of u.

  Intuitively, $\mathrm{low}[u]$ represents the earliest visited vertex that can be reached from the subtree rooted at u by traversing zero or more tree edges followed by at most one back edge.

The algorithm performs a DFS starting from an arbitrary vertex, keeping track of discovery times, low-link values, and a stack of edges.

- When an edge (u, v) is first explored, it is pushed on the stack.

- After finishing the DFS exploration of v, the value of $\mathrm{low}[u]$ is updated to be the minimum between $\mathrm{low}[u]$ and $\mathrm{low}[v]$.

- If $\mathrm{low}[v] \geq \mathrm{disc}[u]$, then u is an articulation point separating the subtree rooted at v from the rest of the graph. At this moment, all edges on the stack from the top down to (u, v) form a new biconnected component; these edges are then removed from the stack.

- Back edges (edges from a vertex u to an already discovered ancestor that is not its parent) are used to update $\mathrm{low}[u]$ by propagating smaller discovery times upward.

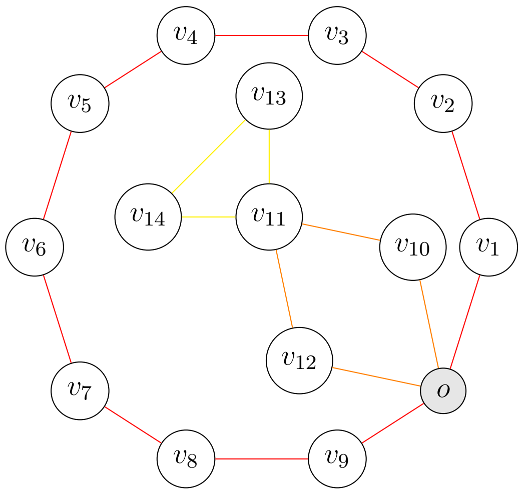
In this manner, the algorithm partitions the edge set of $\mathcal{G}$ into biconnected components. The root of the DFS tree is treated specially: it is an articulation point if and only if it has at least two DFS children.

Whenever the condition $\mathrm{low}[v] \geq \mathrm{disc}[u]$ holds, the set of edges popped from the stack forms a maximal biconnected component. By construction, every edge belongs to exactly one block. Since any two vertices contained in the same block lie on a common cycle, checking membership in the block(s) of o correctly determines whether there exists a simple cycle containing both o and v.

Complexity. Each vertex is visited exactly once and each edge is examined a constant number of times. Edges are pushed and popped from the stack at most once. Thus the algorithm runs in linear time $O(|V| + |E|)$ and uses $O(|V| + |E|)$ space in the worst case.

We are, however, still far from identifying a necessary condition. This limitation directly follows from the structure of the algorithm introduced above.

Example 4.4. *Consider the following graph construction. Start with a cycle of nine vertices, denoted by $v_1, v_2, \ldots, v_9$, together with an additional vertex o. Next, introduce a second cycle involving the vertices $o, v_{10}, v_{11}, v_{12}$. Finally, add a third cycle formed by the vertices v_{11}, v_{13}, v_{14}.*



One possibility minimal solution is

$$
\begin{aligned}
0 &: \{o\}\\
1 &: \{o, v_{10}\}\\
2 &: \{o, v_{10}, v_{11}\}\\
3 &: \{o, v_{10}, v_{14}, v_9\}\\
4 &: \{o, v_{10}, v_{14}, v_9, v_{11}\}\\
5 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8\}\\
6 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}\}\\
7 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}\}\\
8 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7\}\\
9 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7, v_6\}\\
10 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7, v_6, v_5\}\\
11 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7, v_6, v_5, v_4\}\\
12 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7, v_6, v_5, v_4, v_3\}\\
13 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7, v_6, v_5, v_4, v_3, v_2\}\\
14 &: \{o, v_{10}, v_{14}, v_9, v_{13}, v_8, v_{11}, v_{12}, v_7, v_6, v_5, v_4, v_3, v_2, v_1\}
\end{aligned}
$$

Consider moves numbered 3 and 5, highlighted in red because they muove the agent on the red cycle. They also mouve the agent on the yellow cycle simultaneously, which is feasible since the red and yellow cycles are disjoint.

Let the graph $\mathcal{G}$ consist of a red block and an orange block sharing an articulation point, and a yellow block sharing with the orange block the articulation point v_{11}. From this structure, we define the *block-cutpoint tree* of G.

Observation 5. *A* block *is a connected subgraph containing no internal articulation points.*

The block-cutpoint tree represents the interconnection of blocks and articulation points in $\mathcal{G}$.

Formally:

- Nodes of the block-cutpoint tree are of two types:
  1. Blocks: $\mathcal{B}_1, \mathcal{B}_2, \dots$
  2. Articulation points: $a_1, a_2, \dots$
- A block $\mathcal{B}_i$ is connected to an articulation point a_j if and only if $a_j \in \mathcal{B}_i$.

Theorem 4.6 (Block-Cutpoint Graph is a Bipartite Tree)**.** Let $\mathcal{G}$ be a connected graph. Its block-cutpoint graph $\mathcal{T}$ is a bipartite tree.

Proof. We prove the three necessary properties: bipartiteness, connectivity, and acyclicity.

Bipartiteness. By construction, the nodes of $\mathcal{T}$ are of two disjoint types: blocks and articulation points. Edges only connect a block to its contained articulation points, never two blocks or two articulation points. Hence $\mathcal{T}$ is bipartite.

Connectivity. Since $\mathcal{G}$ is connected, every block is connected to at least one articulation point and, through these, to all other blocks. Therefore, $\mathcal{T}$ is connected.

Acyclicity. Assume, for contradiction, that $\mathcal{T}$ contains a cycle. Then two distinct blocks in the cycle would share more than one articulation point. This contradicts the maximality of blocks (a block is a maximal 2-connected subgraph with no internal articulation points). Therefore, no cycle can exist, and $\mathcal{T}$ is acyclic.

Combining these properties, $\mathcal{T}$ is a connected, acyclic, bipartite graph, i.e., a bipartite tree. □

Observation 6. *Each leaf corresponds to a* pendant block*, i.e., a block connected to a single articulation point.*

Within our theory, the node o can always be treated as an articulation point.

After the example one can say: "Given a graph, if its block-cutpoint tree rooted at o contains at most one second-level block that is a leaf, a minimal solution exists", but it's incorrect.

However, minimality requires the ability to alternate moves on the orange block with moves on the combined red and yellow blocks. Consequently, if a second-level block (excluding its articulation point) contains fewer or equal elements than the first-level blocks from which it does not descend (and excluding articulation points), a minimal solution exists.

Even this criterion is not necessary. If two second-level blocks share the same articulation point, and the sum of their element (excluding its articulation point) are less or equal to the number of elements of the first-level blocks from which they do not descend, a minimal solution also exists.

Even this criterion is not necessary. In order to clarify this assertion, let us consider the following illustrative case.

Example 4.5. *Consider the following graph construction. Start with a cycle of nine vertices, denoted by $v_1, v_2, \ldots, v_9$, together with an additional vertex o. Next, introduce a second cycle involving the vertices v_5, v_{10}, v_{11}. The graph in Figure 4.2 admits the following interpretation.*

One may first rotate along the red cycle. Then, while the yellow cycle is rotating, either the blue or the orange cycle can be activated simultaneously.

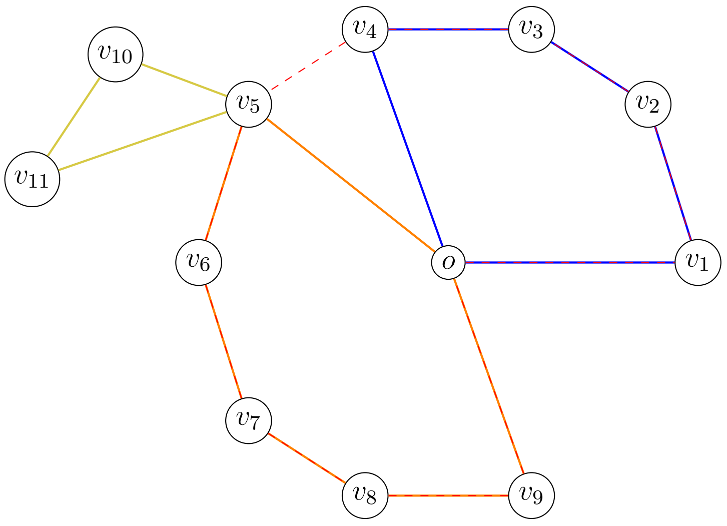


Figure 4.2: Decomposition of cycles in the illustrative graph.

$0 : \{o\}$
$1 : \{o, v_5\}$
$2 : \{o, v_{11}, v_1\}$
$3 : \{o, v_{11}, v_1, v_5\}$
$4 : \{o, v_{11}, v_1, v_{10}, v_2\}$
$5 : \{o, v_{11}, v_1, v_{10}, v_2, v_3\}$
$6 : \{o, v_{11}, v_1, v_{10}, v_2, v_3, v_4\}$
$7 : \{o, v_{11}, v_1, v_{10}, v_2, v_3, v_4, v_5, v_6\}$
$8 : \{o, v_{11}, v_1, v_{10}, v_2, v_3, v_4, v_5, v_6\}$
$9 : \{o, v_{11}, v_1, v_{10}, v_2, v_3, v_4, v_5, v_6, v_7\}$
$10 : \{o, v_{11}, v_1, v_{10}, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$
$11 : \{o, v_{11}, v_1, v_{10}, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$

To properly understand this mechanism, one should not restrict the solution of each block to the construction given in the proof of the theorem. Instead, for each block we must consider its elementary cycle graph. This graph is commonly referred to as the elementary cycle graph. Its purpose is to encode the mutual relationships among the elementary cycles contained in a given block. By definition, each node represents one elementary cycle, while an edge between two nodes indicates that the corresponding cycles are not disjoint, but rather share at least one edge of the original block. In this way, the elementary cycle graph provides a higher-level representation in which the focus shifts from individual edges and vertices to the interplay between cycles themselves.

The structural condition then becomes the following: the number of second-level elements must be strictly smaller than the sum of the elements of first-level cycles from which they do not descend plus the sum of the elements of elementary cycles from which they do not descend.

We now consider the case where there exist two second-level blocks that do not share the same articulation point. It is necessary to distinguish two subcases:

1. The two blocks descend from the same first-level block.
2. The two blocks descend from distinct first-level blocks.

We start with the second case. It is immediately apparent that the inequality stating that the sum of the elements of the second-level blocks (excluding their respective articulation points) must be smaller than the sum of the elements of the first-level blocks (excluding o) does not hold in general. This is illustrated by the following example.

Example 4.6. *Consider the graph depicted in Figure 4.3. The main cycle (red) passes through o and nodes $v_1, \dots, v_9$. An inner cycle (orange) passes through $o, v_{10}, v_{11}, v_{12}$. A yellow cycle is attached to v_{11}, consisting of nodes $v_{11}, v_{13}, v_{14}, v_{15}, v_{16}, v_{17}$. Finally, a blue cycle is attached to v_5, consisting of nodes $v_5, v_{18}, v_{19}, v_{20}, v_{21}$.*

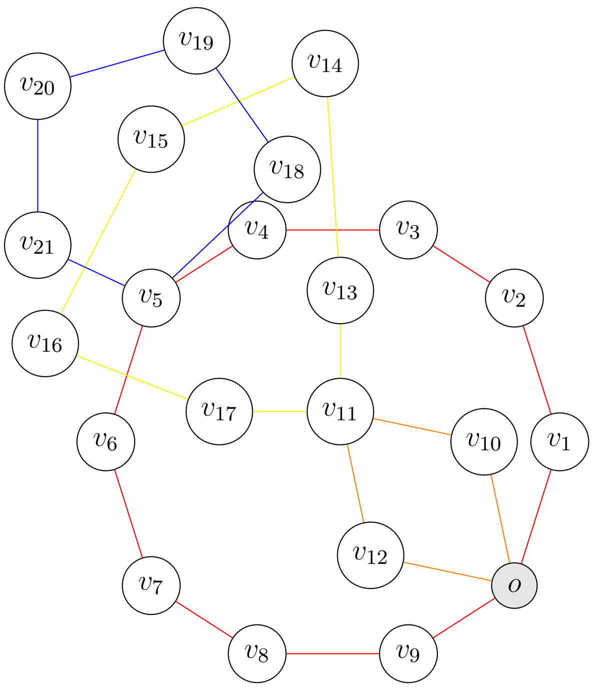


Figure 4.3: Example of a graph with three second-level blocks and an additional large cycle attached to v_5.

Despite not satisfying the element-count inequality, it is possible to alternate moves so that a different task agent passes through o at each move.

$$9 \geq 5$$
$$3 \geq 4$$

This is a minimal solution:

$0 : \{o\}$

$1 : \{o, v_9\}$

$2 : \{o, v_9, v_8\}$

$3 : \{o, v_9, v_8, v_7\}$

$4 : \{o, v_9, v_8, v_7, v_6\}$

$5 : \{o, v_9, v_8, v_7, v_6, v_{12}\}$

$6 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{11}\}$

$7 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_5\}$

$8 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{11}\}$

$9 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_5\}$

$10 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{11}\}$

$11 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_5\}$

$12 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{11}\}$

$13 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_5\}$

$14 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{11}\}$

$15 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5\}$

$16 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}\}$

$17 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}\}$

$18 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4\}$

$19 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4, v_3\}$

$20 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4, v_3, v_2\}$

$21 : \{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4, v_3, v_2, v_1\}$

The block-cutpoint tree for this configuration can be drawn as follows:

- *First-level block: red cycle and orange cycle excluding o.*
- *Second-level blocks: yellow cycle attached to v_{11} and the blue cycle attached to v_5.*
- *Articulation points: o, v_{11}, v_5.*

This example illustrates that the criterion above is insufficient to guarantee the existence of a minimal solution in this scenario, since the dimension of blocks alone does not capture the necessary structural constraints.

The reason is straightforward: when considering the sum of the sizes of non-ascending cycles, we are counting the rotations taht can be exploited to move a disjoint second-level cycle as a whole. However, now we have more rotation on the first-level blocks than their dimension...

In order to reformulate the expression, we proceed as follows. For each first–level block, we denote by e_i its *effective dimension*, that is, the size of the block minus the articulation

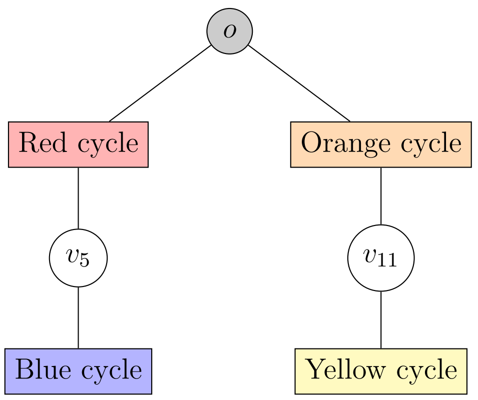


Figure 4.4: Block-cutpoint tree of the graph. Only o is colored gray; rectangles represent blocks and circles are cutpoints.

point from which it depends. We further denote by

$$s_i = \sum_{j>1} e_{i,j},$$

the sum of the effective dimensions of all descendant blocks at levels $j > 1$ that originate from the i-th block of first level.

The minimality condition can be expressed as follows: if for every j such that $s_j \neq 0$ one has

$$\sum_{i \neq j} (e_i + s_i) \;\geq\; s_j,$$

then a minimal solution exists.

The key point is that each rotation on a block of level greater than 1 (which does not directly generate or remove a task at its articulation point) must be paired with one rotation on a block of level 1, and possibly with rotations on other higher-level blocks, provided that these form disjoint cycles. In particular, the cycles that must be covered are the s_i, while the available rotations are given by the $e_i + s_i$, corresponding to the number of task agents residing on the branch of the decomposition tree. These sets are disjoint, since they belong to distinct branches that share only the articulation point (which itself is always part of a first-level cycle).

Example 4.7. *Consider a block of third level. Figure 4.5 shows the corresponding structure: a red cycle at the root, an orange cycle attached to it, and a green cycle descending from the orange one.*

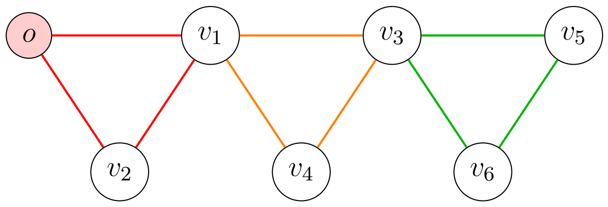


Figure 4.5: A third–level block with nested cycles: red (level 1), orange (level 2), green (level 3).

A minimal sequence of moves is:

$$\begin{aligned}
0 &: \{o\},\\
1 &: \{o, v_1\},\\
2 &: \{o, v_3\},\\
3 &: \{o, v_5, v_1\},\\
4 &: \{o, v_5, v_3\},\\
5 &: \{o, v_5, v_6, v_1\}.
\end{aligned}$$

Moves 3 and 5 involve the simultaneous rotation of the red and green cycles, which are disjoint. From this point onward, the green cycle can be ignored, and the problem reduces to the configuration with only the red and orange cycles. However, since the configuration already contains an untasked agent at node v_1, the last move corresponds to the first step of the case without the green cycle. Notice that we have performed two rotations on the red cycle (to compensate for the agent living only on the third-level block), and the rotations to be paired with a first-level rotation equal exactly the size of the third-level block.

Example 4.8. *We now consider a fourth-level block. Figure 4.6 illustrates the structure: a red cycle at the root, an orange cycle attached, then a green one, and finally a purple one descending from the green.*

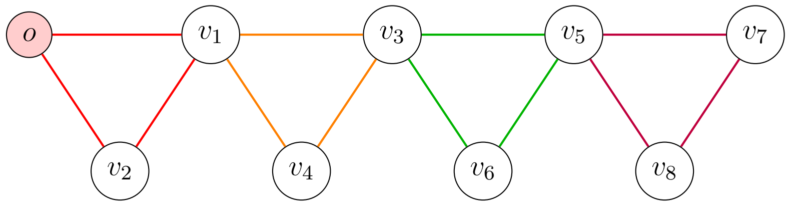


Figure 4.6: A fourth–level block with nested cycles: red (level 1), orange (level 2), green (level 3), purple (level 4).

A minimal sequence of moves is:

$$\begin{aligned}
0:&\ \{o\},\\
1:&\ \{o, v_1\},\\
2:&\ \{o, v_3\},\\
3:&\ \{o, v_5, v_1\},\\
4:&\ \{o, v_7, v_3\},\\
5:&\ \{o, v_7, v_6, v_1\},\\
6:&\ \{o, v_7, v_8, v_3\}.
\end{aligned}$$

Moves 3 and 5 involve the joint action of the red and green cycles, which are disjoint; moves 4 and 6 involve the orange and purple cycles, also disjoint. At this stage, the orange cycle can be discarded, and the dynamics proceed as in the case with only the red, orange, and green cycles. However, since the configuration already reached state 2, the last two moves simply retrace previous ones and are therefore excluded. Again, the number of rotations that must be paired with a first-level rotation equals the size of the fourth-level block.

The same reasoning extends inductively: a fifth-level block behaves analogously to the third-level one, and so on. This explains why exactly one full cycle of rotations is always required for every block from the second level onwards.

The most reliable approach to have the solution is to perform a recursive analysis of the block-cutpoint graph, solving the problem progressively from the leaves upward. The procedure works effectively also for deeper levels in the decomposition:

1. **Descent.** Starting from the root o, select any first-level block and proceed recursively into it.

2. **Leaf solution.** Upon reaching a leaf block, compute the solution for that block with the articulation point as the origin (its existence is guaranteed, and we have already described the constructive algorithm above).

3. **Backtracking.** Once all leaf blocks associated with the same articulation point have been processed, backtrack to the upper-level block. First, compute the solution disregarding the lower blocks. Then, build a table with one row for each articulation point (two in this illustrative case).

   - Insert a column for each motion in the upper block, marking with an "x" the articulation points lying on that block.
   - After the column corresponding to the move that frees a given articulation point, insert two additional columns for each necessary motion in the lower blocks: one corresponding to the already computed solution in the child block and one corresponding to a motion in the upper block.

   Note that the solution constructed in this way, like the leaf solution, can be read both left-to-right and right-to-left without loss of validity. We repeat backtracking until reach the root o.

4. **Merging.** At higher levels, when two consecutive columns correspond to independent motions (e.g., one in a first-level block B_1 and one in a third-level block B_3), they can be merged into a single simultaneous step $B_1 + B_3$: we have two consecutive column that are not escluding, for example $(B_1, x, -)$ and $(-, x, B_3)$.

5. **Completion.** Returning to the root o, process the next block in the same fashion. At this stage, we obtain two independent solutions, one for each block. Since they are independent, we only need to respect the internal order of each, while interleaving them optimally to minimize the overall number of moves merged more columns not escluding.

A possible strategy to organize the merged sequence, while preserving the internal ordering, relies on the structural decomposition of the input sets. Given the construction of the sequences, each of them can be partitioned into three distinct blocks:

A: a block not containing "$-$" in the first row,

B: a block containing "$-$" in the first row,

C: a block not containing "$-$" in the first row (after B).

If no "$-$" symbol appears in the first row, the sequence reduces to block A only. The most effective procedure is to order the columns by blocks first, independently of their content, and only afterwards assign the entries consistently with the order within each block.

Case 1: If we have A_1, A_2 but no B_1, B_2, the merging is straightforward: columns from A_1 and A_2 can be arbitrarily interleaved while preserving internal order. This results in a single merged block

$$A_{1+2}.$$

Case 2: Suppose the configuration is A_1, B_1, C_1, A_2 with no B_2. In this case, it is convenient to insert the columns of A_2 into B_1, after those not containing "$-$" in the first row, starting from the end. This allows every new column to be merged consistently with the preceding column.

- If the columns of A_2 are sufficient, we can merge the remaining part of A_2 with A_1 and then merge C_1 with C_2, yielding

$$A_{1+2}.$$

- If they are not sufficient, the result is instead

$$A_1, \quad B_{1+2}, \quad C_{1+2},$$

because in this case the merged block B_{1+2} exists, while C absorbs the surplus.

Case 3: When all blocks appear, i.e.,

$$A_1, \quad B_1, \quad C_1, \quad A_2, \quad B_2, \quad C_2,$$

the recommended strategy is to alternate as much as possible between B_1 and B_2, since both blocks include columns with and without a "$-$" in the first row.

- If the columns are sufficient, one can merge $A_1 + A_2$ and $C_1 + C_2$, obtaining a single block
$$A_{1+2}.$$
- If B_1 contains strictly more columns than B_2, then the merged block B_{1+2} still contains "$-$" in the first row. In this case, the merging procedure is analogous to Case 2:
  - If the columns of A_2 are sufficient, we can merge the remaining part of A_2 with A_1 and merge C_1 with C_2, yielding
$$A_{1+2}.$$
  - If they are not sufficient, the columns of A_2 and B_2 must be shifted forward until just after the last column of B_{1+2} that still contains "$-$" in the first row. At this stage, the first column of C_2 is appended to fill the gap. Repeat until don't finish the column of B_{1+2} that still contains "$-$" or the column of C_2:
    * If the columns of C_2 are sufficient, we can merge the remaining part of C_2 with C_1, yielding
$$A_{1+2}.$$
    * If they are not sufficient, the result is instead
$$A_1, \quad B_{1+2}, \quad C_{1+2},$$
because now B remains dominant while C absorbs the residual.

It is important to remark that blocks A and C are interchangeable. That is, one valid solution considers the sequence in the order
$$A_i,\, B_i,\, C_i,$$
while an equivalent solution may instead be
$$C_i,\, B_i,\, A_i.$$

By iterating this process for all blocks, we eventually obtain either:

- a minimal solution, if every move sequence includes an operation in the row corresponding to the root articulation point, or
- otherwise, a valid Optimal Max-Origin-Cost Motion (though not necessarily an Optimal Origin-Cost Motion).

This recursive synthesis thus ensures that the structural dependencies of the block-cutpoint decomposition are fully respected, while at the same time providing a constructive method to derive a globally consistent motion sequence.

Example 4.9. *Consider the graph in Figure 4.3, with the central articulation point o, an outer red cycle $(v_1, \dots, v_9, o)$, an inner orange cycle $(o, v_{10}, v_{11}, v_{12})$, a yellow cycle attached at v_{11}, and a blue cycle attached at v_5.*

We apply the recursive procedure described above as follows:

1. ***Descent to leaf blocks.*** *Start from the orange cycle. The yellow cycle attached is a leaf blocks in the block-cutpoint tree. We compute a solution with the corresponding articulation point as origin.*

$$0 : \{v_{11}\}$$
$$1 : \{v_{11}, v_{17}\}$$
$$2 : \{v_{11}, v_{17}, v_{16}\}$$
$$3 : \{v_{11}, v_{17}, v_{16}, v_{15}\}$$
$$4 : \{v_{11}, v_{17}, v_{16}, v_{15}, v_{14}\}$$
$$5 : \{v_{11}, v_{17}, v_{16}, v_{15}, v_{14}, v_{13}\}$$

2. ***Backtracking at articulation points.*** *Having solved the yellow cycle, we backtrack to the orange cycle. At this level, we first construct the solution ignoring the child block.*

$$0 : \{o\}$$
$$1 : \{o, v_{12}\}$$
$$2 : \{o, v_{12}, v_{11}\}$$
$$3 : \{o, v_{12}, v_{11}, v_{10}\}$$

Then, we insert in the motion table two additional columns after the move that frees v_{11}, alternating one step from the yellow cycle solution and one step from the orange cycle solution. The result is a valid interleaving, which preserves both orders.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	Y	Y	-	Y	-	Y	-	Y	-	Y	-	Y	Y
-	x	x	O	x	O	x	O	x	O	x	O	x	x
$\{o\}$	$\{o, v_{12}\}$	$\{o, v_{12}, v_{11}\}$	$\{o, v_{12}, v_{17}\}$	$\{o, v_{12}, v_{17}, v_{11}\}$	$\{o, v_{12}, v_{17}, v_{16}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{11}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}, v_{11}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}, v_{14}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}, v_{14}, v_{11}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}, v_{14}, v_{13}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}, v_{14}, v_{13}, v_{11}\}$	$\{o, v_{12}, v_{17}, v_{16}, v_{15}, v_{14}, v_{13}, v_{11}, v_{10}\}$

3. ***Descent to leaf blocks.*** *Close the orange cycle, pass to the red cycle. The blue cycle attached is a leaf blocks in the block-cutpoint tree. We compute a solution with*

the corresponding articulation point as origin.

$0 : \{v_5\}$
$1 : \{v_5, v_{18}\}$
$2 : \{v_5, v_{18}, v_{19}\}$
$3 : \{v_5, v_{18}, v_{19}, v_{20}\}$
$4 : \{v_5, v_{18}, v_{19}, v_{20}, v_{21}\}$

4. ***Backtracking at articulation points.*** *Having solved the blue cycle, we backtrack to the red cycle. At this level, we first construct the solution ignoring the child block.*

$0 : o$
$1 : o, v_9$
$2 : o, v_9, v_8$
$3 : o, v_9, v_8, v_7$
$4 : o, v_9, v_8, v_7, v_6$
$5 : o, v_9, v_8, v_7, v_6, v_5$
$6 : o, v_9, v_8, v_7, v_6, v_5, v_4$
$7 : o, v_9, v_8, v_7, v_6, v_5, v_4 v_3$
$8 : o, v_9, v_8, v_7, v_6, v_5, v_4 v_3, v_2$
$9 : o, v_9, v_8, v_7, v_6, v_5, v_4 v_3, v_2, v_1$

Then, we insert in the motion table two additional columns after the move that frees v_5, alternating one step from the blue cycle solution and one step from the red cycle solution. The result is a valid interleaving, which preserves both orders.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
-	R	R	R	R	R	-	R	-	R	-	R	-	R	R	R	R	R
-	x	x	x	x	x	B	x	B	x	B		B	x	x	x	x	x
$\{o\}$	$\{o, v_9\}$	$\{o, v_9, v_8\}$	$\{o, v_9, v_8, v_7\}$	$\{o, v_9, v_8, v_7, v_6\}$	$\{o, v_9, v_8, v_7, v_6, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_{21}\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_{21}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_{21}, v_5, v_4\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_{21}, v_5, v_4, v_3\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_{21}, v_5, v_4, v_3, v_2\}$	$\{o, v_9, v_8, v_7, v_6, v_{18}, v_{19}, v_{20}, v_{21}, v_5, v_4, v_3, v_2, v_1\}$

Merging independent moves. *One possibile way is, as upon:*

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*
-	R	R	R	R	Y	Y	R	Y	R	Y	R	Y	R	Y	R	Y	Y	R	R	R	R
-	-	-	-	-	x	x	O	x	O	x	O	x	O	x	O	x	x	-	-	-	-
-	x	x	x	x	-	-	x	B	x	B	x	B	x	B	x	-	-	x	x	x	x
$\{o\}$	$\{o, v_9\}$	$\{o, v_9, v_8\}$	$\{o, v_9, v_8, v_7\}$	$\{o, v_9, v_8, v_7, v_6\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{11}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{11}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{11}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{11}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{11}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4, v_3\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4, v_3, v_2\}$	$\{o, v_9, v_8, v_7, v_6, v_{12}, v_{13}, v_{18}, v_{14}, v_{19}, v_{15}, v_{20}, v_{16}, v_{21}, v_{17}, v_5, v_{11}, v_{10}, v_4, v_3, v_2, v_1\}$

The situation becomes significantly more involved when a block contains multiple articulation points, each of which connects to further descendant blocks. Understanding how to organize the rotations in such cases is both more challenging and more interesting. We now illustrate these difficulties with explicit examples.

Example 4.10. *Consider a graph G consisting of:*

- *a red cycle including the nodes $o, v_1, \ldots, v_9$;*
- *an orange cycle including the nodes $v_8, v_{10}, v_{11}, v_{12}$;*
- *a yellow cycle including the nodes $v_7, v_{13}, v_{14}, v_{15}, v_{16}, v_{17}$, attached to v_{11};*
- *a blue cycle including the nodes $v_5, v_{18}, v_{19}, v_{20}, v_{21}$.*

The corresponding block–cutpoint graph is displayed in Figure 4.8, where blocks are shown as squares and articulation points as circles. Each articulation point connects to multiple descendant blocks, producing several branches that must be coordinated.

For this configuration, a minimal solution exists. A possible assignment is summarized in Table 4.1.

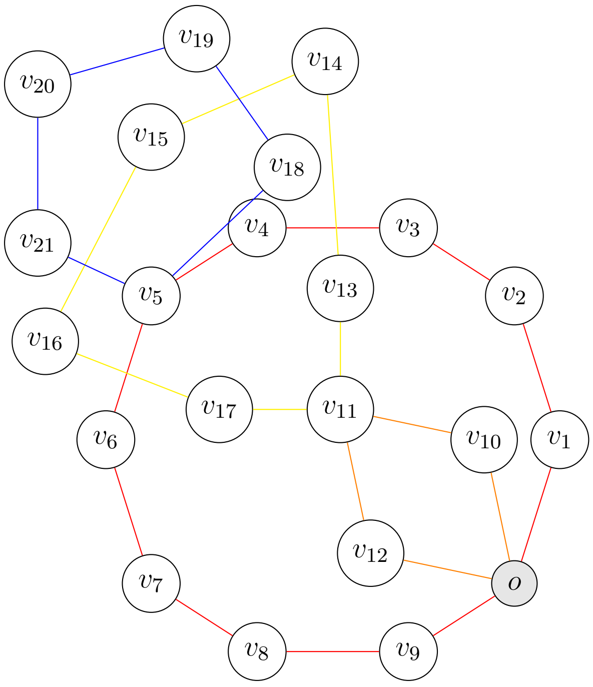


Figure 4.7: Example of a graph with three second-level blocks and an additional large cycle attached to v_5.

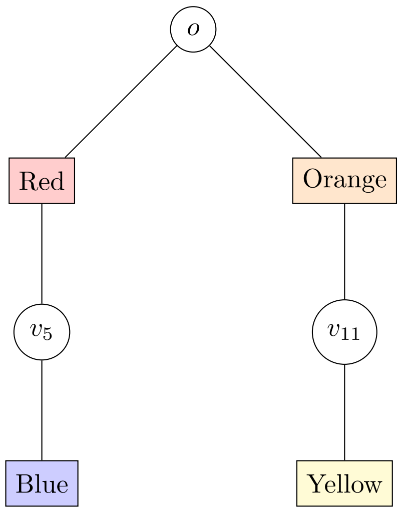


Figure 4.8: Block–cutpoint graph associated with Figure 4.7.

A naive idea would be to only compare the maximum block sizes corresponding to distinct articulation points. However, as we will see in the next example, this is not sufficient.

Example 4.11. *Now consider a graph G consisting of:*

Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Red	R	R	R	B	R	R	B	R	B	R	R	R	R	R	R	R	R	R
Orange	x	x	x	O	x	x	O	x	O	x	x	x	x	x	x	x	x	x
Yellow	x	x	x	Y	x	x	Y	x	x	x	x	x	x	x	x	x	x	x

Table 4.1: Example of a minimal assignment of rotations across the blocks.

- *a red cycle including the nodes* $o, v_1, \ldots, v_9$*;*
- *an orange cycle including the nodes* $v_8, v_{10}, v_{11}, v_{12}$*;*
- *a yellow cycle including the nodes* $v_4, v_{13}, v_{14}, v_{15}$*;*
- *a blue cycle including the nodes* $o, v_{16}, v_{17}, v_{18}$*.*

Figure 4.9: Graph with two second-level blocks that cannot yield a minimal solution.

In this case, no minimal solution exists. The issue arises from the distance between two second-level blocks that depend on different articulation points within the same parent block. The distance between them exceeds the size of each individual block. The only way to attempt a feasible rotation is to rotate both second-level blocks simultaneously, since both have size three. However, for the rotation to be valid, each must preserve its articulation node as untasked. Consequently, regardless of the direction of rotation (clockwise or counterclockwise), we end up with a block of size three that can cover at most three nodes, while four nodes would need to be covered. Thus, no minimal solution is achievable.

One might think of computing the maximum between the size of a block in one direction and, for the subsequent blocks, the maximum between the distance to the previous and their size. However, since rotations can occur in both directions, the correct quantity would already involve taking the minimum between the two maxima. Moreover, the addition of a single edge, such as (o, v_2) *(or equivalently* $(o, v_3), (o, v_4), (o, v_5), (o, v_6)$*), would immediately allow for the existence of a minimal solution.*

We now outline how to derive a general solution. For each first-level block, consider the graph of its elementary cycles. All cycles from which a second-level block descends must be taken into account. Each cycle can be rotated in two directions; ideally, we aim to maximize the number of feasible rotations. However, the total number of admissible configurations is not simply 2^n, where n denotes the number of elementary cycles associated with a second-level block.

For instance, consider Figure 4.2, where a second-level block is attached to both the blue and the orange cycle. Two scenarios may occur:

- A clockwise rotation of the blue cycle enforces a counterclockwise rotation of the red cycle, in order to keep the initial segment of the blue cycle (node v_4) free. This, in turn, enforces a clockwise rotation of the orange cycle so that the final segment of the red cycle (node v_5) remains free at the end.

- Conversely, a counterclockwise rotation of the blue cycle enforces a clockwise rotation of the red cycle, in order to keep the final segment of the blue cycle (node v_4) free at the end. This further enforces a counterclockwise rotation of the orange cycle so that the initial segment of the red cycle (node v_5) remains free at the beginning.

Hence, it is crucial to allocate a larger number of admissible moves to those first-level cycles that will later require fewer adjustments at higher levels. In this way, more moves are available to be interleaved with the cycles that demand a greater number of operations.

An additional constraint must be satisfied: one must avoid leaving gaps on the first-level cycles (as in the example above). This requirement takes precedence, since otherwise one would be forced to perform a rotation in which a task is either left incomplete or duplicated, thereby precluding the possibility of obtaining a minimal solution.

Once all articulation nodes are released, the synchronized movements of higher-level cycles are alternated with those required to re-release the articulation nodes. After an optimized solution has been obtained for each first-level block, these partial solutions are combined together as illustrated above.

Appendix A

Introduction to Graph Theory

Graph theory is a large and flexible branch of discrete mathematics, that deals with the study of structures called graphs. Fundamentally, a graph shows how things are connected in pairs: it has a collection of vertices (or nodes) and a bunch of edges that connect pairs of vertices.

From computer networks to social interactions, from transportation systems to biological processes, this basic but strong simplification lets us depict a great range of actual world systems.

The origins of graph theory trace back to 1736, when the Swiss mathematician Leonhard Euler discussed the well-known Seven Bridges of Königsberg issue. The city of Königsberg (now Kaliningrad) was built around the Pregel River and included two large islands connected to each other and to the mainland by seven bridges. Finding a walk through the city that would pass each bridge exactly once was the challenge.

Euler showed that such a walk was not possible and in the process developed the fundamentals of graph theory. He presented a fresh mathematical representation of the problem by abstracting the landmasses as nodes and the bridges as edges rather than stressing the geography arrangement. His studies not only answered the first question but also established the foundation for what would later become Eulerian trail and Eulerian circuits.

Since then, the field has grown to be a basic part of many different academic fields, such as computer science, biology, logistics, sociology, and many others.

In this chapter, we establish the fundamental definitions, ideas, and categories that guide the study of graphs. Upon these foundational elements will be built the more advanced topics explored in the subsequent chapters.

A.1 Basic Definitions and Concepts

This section seeks to give the exact mathematical definitions of graphs and their main characteristics. Furthermore, we provide illustrative instances meant to foster a natural sense of these abstract concepts.

Definition A.1 (Graph)**.** A *(directed, weighted) graph* is defined as a triple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, where:

- $\mathcal{V}$ is the set of vertices.
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges.
- w is a function that assigns a weight to each edge, i.e., $w : \mathcal{E} \to \mathbb{R}$.

Given this formal definition of a graph, it is sometimes helpful to show the structure of the graph in matrix form. One such depiction is the adjacency matrix, which captures how vertices in the graph are related to one another.

Definition A.2 (Weighted Adjacency Matrix)**.** Given a (directed, weighted) graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, the *weighted adjacency matrix* is the matrix $A \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ defined as:

$$A_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

where $w(v_i, v_j)$ denotes the weight assigned to the edge from vertex v_i to vertex v_j.

However, rather than clearly specifying the function w, the adjacency matrix A offers another view of the graph encoding the weights of the matrix form's edges as well as its topology. Within this framework, we can write $\mathcal{G} = (\mathcal{V}, \mathcal{E}, A)$, where A stands for the weighted adjacency matrix.

Example A.1. *Consider the (directed, weighted) graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, A)$ shown in Figure A.1, where:*

$$\begin{aligned} \mathcal{V} &= \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}, \\ \mathcal{E} &= \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6), (v_6, v_1), \\ &\quad (v_2, v_5), (v_3, v_6), (v_4, v_7), (v_3, v_2), (v_5, v_4), (v_7, v_2)\}, \\ A &= \begin{pmatrix} 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 5 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \end{aligned}$$

Definition A.3 (Graph Properties)**.** A graph can be categorized based on several structural features:

- It is called *unweighted* if no weights are assigned to the edges. Here it is basically stated as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.
- It is called *undirected* if edges are unordered pairs, i.e., $(u, v) = (v, u)$.

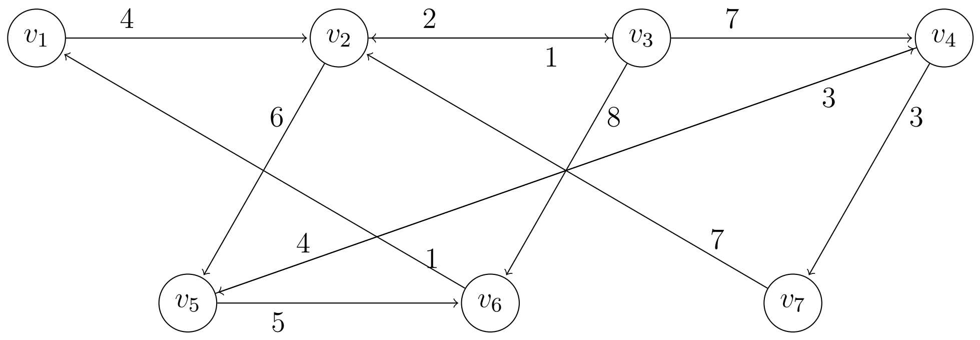


Figure A.1: Directed weighted graph $\mathcal{G}$

- A graph is said to be *simple* if it is undirected, unweighted, and contains no self-loops (edges of the form (u, u)).

Observation 7. *For an unweighted graph, the adjacency matrix A is a simpler form of the weighted adjacency matrix, with the entries A_{ij} equals 1 or 0 depending on whether the vertices v_i and v_j have an edge connecting them.*

For an undirected graph, the adjacency matrix A is symmetric, i.e., $A_{ij} = A_{ji}$ for all $i, j \in \mathcal{V}$.

For a simple graph, the adjacency matrix A is symmetric with null-diagonal (every element in the diagonal indicates the presec of the self-loops for i-esimo vertex) where the entries A_{ij} can be either 1 or 0.

In this thesis, we focus exclusively on unweighted undirected graphs, making only a few remarks on the case of directed graphs.

Remark. So, to leave, the same struct, we consider that a graph is undirected if and if only $\forall e = (u, v) \in \mathcal{E} \implies e' = (v, u) \in \mathcal{E}$.

We present now two extreme yet conceptually useful examples of graphs.

Definition A.4 (Complete Graph)**.** A *complete graph* is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which every pair of distinct vertices is connected by an edge. A complete graph with n vertices is denoted by $\mathcal{K}_n$.

Note that, depending on the context, complete graphs may or may not include self-loops. In this thesis, we allow self-loops unless otherwise specified.

Definition A.5 (Edgeless Graph)**.** An *edgeless graph* (also called *totally disconnected*) is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which no two distinct vertices are connected by an edge. In other words, $\mathcal{E}$ contains only self-loops, if any.

We condier only not edgeless graph in this thesis.

Example A.2. *Consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in Figure A.2, which is an undirected, unweighted complete graph. The vertex set is $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, and the edge set $\mathcal{E}$ includes all edges between every pair of distinct vertices. According to the previous definition, this graph is denoted by $\mathcal{K}_7$.*

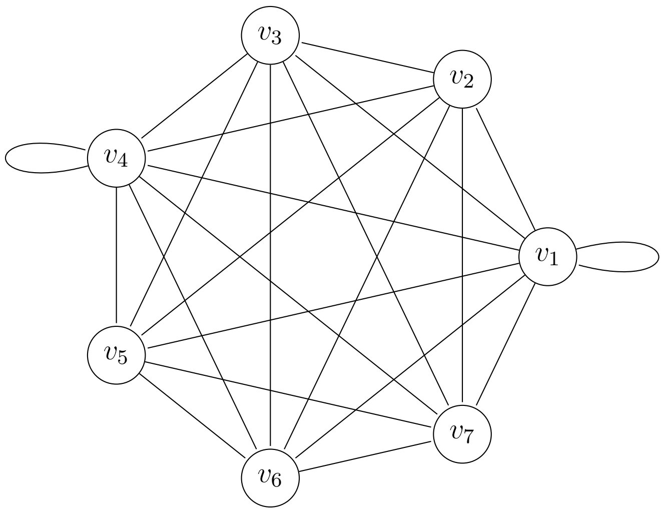


Figure A.2: Complete graph $\mathcal{K}_7$

After having discussed the general structure of graphs, we now focus on the edges and how they fit into a graph. While vertices are the fundamental entities of a graph, the edges connect them and specify their interactions.

Definition A.6 (Tail and Head Vertices)**.** Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we define two functions $\theta : \mathcal{E} \to \mathcal{V}$ and $\kappa : \mathcal{E} \to \mathcal{V}$ such that, for each edge $e \in \mathcal{E}$, $\theta(e)$ is the *tail vertex* and $\kappa(e)$ is the *head vertex.*

Definition A.7 (Boundary Vertices)**.** Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, for a subset of vertices $\mathcal{U} \subseteq \mathcal{V}$, we define its (out-)boundary $\partial_{\mathcal{U}}$ and in-boundary $\partial_{\mathcal{U}}^-$ as, respectively, $\partial_{\mathcal{U}} = \{e \in \mathcal{E} : \theta(e) \in \mathcal{U},\ \kappa(e) \notin \mathcal{U}\}$, $\partial_{\mathcal{U}}^- = \{e \in \mathcal{E} : \theta(e) \notin \mathcal{U},\ \kappa(e) \in \mathcal{U}\}$.

When $\mathcal{U}$ is a singleton, we simply write ∂_i for $\partial_{\{i\}}$ and ∂_i^- for $\partial_{\{i\}}^-$.

For a undirected graph, for every $\mathcal{U} \subseteq \mathcal{V}$ we have that $\partial_{\mathcal{U}} = \partial_{\mathcal{U}}^-$.

We can consider also a relationships between two edges:

Definition A.8 (Opposite Edges)**.** Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, two edges $e_1, e_2 \in \mathcal{E}$ are said to be *opposite* if $\theta(e_1) = \kappa(e_2) \quad \text{and} \quad \kappa(e_1) = \theta(e_2)$.

For a undirected graphs, for each edge $e_1 \in \mathcal{E}$, there exists an edge $e_2 \in \mathcal{E}$ such that e_1 and e_2 are opposite.

Definition A.9 (Vertex-Edge Incidence Matrix)**.** Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the *vertex-edge incidence matrix* is the matrix $B \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{E}|}$ defined entry-wise as:

$$B_{ie} = \begin{cases} +1 & \text{if } \theta(e) = i \neq \kappa(e), \\ -1 & \text{if } \kappa(e) = i \neq \theta(e), \\ 0 & \text{if } i \notin \{\theta(e), \kappa(e)\} \text{ or } \theta(e) = \kappa(e) = i. \end{cases}$$

In other words, the matrix B has rows indexed by vertices $i \in \mathcal{V}$ and columns indexed by edges $e \in \mathcal{E}$. Each column corresponding to a self-loop is identically zero. For any edge e that is not a self-loop, the corresponding column in B contains two nonzero entries: $+1$ in the row corresponding to the tail vertex $\theta(e)$ and -1 in the row corresponding to the head vertex $\kappa(e)$.

Note that an unweighted directed graph without self-loops is completely characterized by its incidence matrix B.

Proposition A.1. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $B \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{E}|}$ be its vertex-edge incidence matrix. Then:

- For any vector $x \in \mathbb{R}^{|\mathcal{V}|}$, the vector $B^\top x \in \mathbb{R}^{|\mathcal{E}|}$ has components
$$(B^\top x)_e = x_{\theta(e)} - x_{\kappa(e)}, \quad \forall e \in \mathcal{E};$$
- For any vector $y \in \mathbb{R}^{|\mathcal{E}|}$, the vector $By \in \mathbb{R}^{|\mathcal{V}|}$ has components
$$(By)_i = \sum_{e:\kappa(e)=i} y_e - \sum_{e:\theta(e)=i} y_e, \quad \forall i \in \mathcal{V}.$$

In particular, if $x = \mathbf{1}$ is the all-ones vector in $\mathbb{R}^{|\mathcal{V}|}$, then $B^\top \mathbf{1} = 0$.

Example A.3. *Let $\mathcal{G} = K_7$ be the complete, unweighted, undirected graph on 7 vertices, as in Example A.2.*

The incidence matrix $B \in \mathbb{R}^{7 \times 44}$... too big to be present in this thesis!

A.2 Subgraphs

This section introduces the concept of subgraphs, which are a basic tools in graph theory for evaluating and breaking graphs into more manageable, significant parts. Studying subgraphs helps us understand local structures, simplify difficult graphs, and identify particular patterns of concern.

Definition A.10 (Subgraph)**.** A *subgraph* of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph $\mathcal{H} = (\mathcal{V}', \mathcal{E}')$ such that: $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$.

Several types of subgraphs are commonly studied, depending on how they are derived from the original graph:

Definition A.11 (Induced Subgraph)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a subset $\mathcal{V}' \subseteq \mathcal{V}$, the *induced subgraph* $\mathcal{G}[\mathcal{V}']$ is the graph formed by the vertices in $\mathcal{V}'$ and all the edges in $\mathcal{E}$ that connect pairs of vertices in $\mathcal{V}'$. Formally:
$$\mathcal{G}[\mathcal{V}'] = (\mathcal{V}', \{(u,v) \in \mathcal{E} \mid u, v \in \mathcal{V}'\}).$$

Definition A.12 (Spanning Subgraph)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a *spanning subgraph* is a subgraph that includes all the vertices of $\mathcal{G}$, i.e., $\mathcal{V}' = \mathcal{V}$, but possibly only a subset of its edges. We denote the spanning subgraph obtained by removing a set of edges $\mathcal{E}' \subseteq \mathcal{E}$ as $\mathcal{G} \setminus \mathcal{E}'$.

Definition A.13 (Generated Subgraph)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a subset of edges $\mathcal{E}' \subseteq \mathcal{E}$, the *generated subgraph* $\mathcal{G}[\mathcal{E}']$ is the graph whose edge set is $\mathcal{E}'$ and whose vertex set consists of all vertices incident to at least one edge in $\mathcal{E}'$. Formally:

$$\mathcal{G}[\mathcal{E}'] = \left(\{u \in \mathcal{V} \mid \exists v \in \mathcal{V},\ (u,v) \in \mathcal{E}'\}\,,\ \mathcal{E}'\right).$$

Example A.4. *Consider $\mathcal{G} = (\mathcal{V}, \mathcal{E}) = \mathcal{K}_7$, as in Example A.2. Let*

$$\mathcal{E}' = \{(v_1, v_2), (v_1, v_6), (v_1, v_5), (v_1, v_1), (v_2, v_6), (v_2, v_4), (v_4, v_4), (v_4, v_5), (v_2, v_5), (v_5, v_6)\}$$

be a selected subset of edges, and consider the generated subgraph $\mathcal{G}[\mathcal{E}']$, shown in Figure A.3.

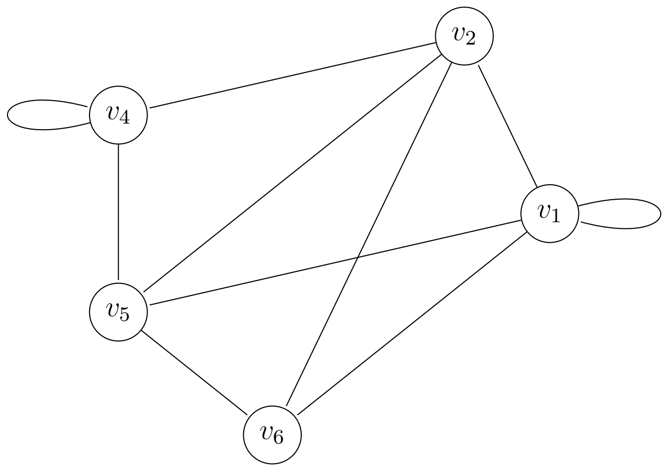


Figure A.3: Generated subgraph $\mathcal{G}[\mathcal{E}']$ from the graph $\mathcal{G} = \mathcal{K}_7$

A.3 Reachability and connectedness

In this section, we discuss graph reachability, which produces a natural equivalence relation between vertices. Through investigating the connected ideas of walks, routes, paths, and cycles, we progressively grow this idea and finally arrive at the concepts of connectivity and graph condensation.

We start with the basic definition of a walk, which is how the idea of reachability is made.

Definition A.14 (Walk (vertex-based))**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. A *walk* from vertex i to vertex j is a finite sequence of vertices $\gamma = (i_0, i_1, \ldots, i_\ell)$ such that $i_0 = i$, $i_\ell = j$, and $(i_{h-1}, i_h) \in \mathcal{E}$ for all $h = 1, \ldots, \ell$; that is, there is an edge between each pair of consecutive vertices. The integer ℓ is called the *length* of the walk. By convention, a walk of length 0 consists of a single vertex and is considered a walk from that vertex to itself.

An alternative edge-based representation of walks is as follows:

Definition A.15 (Walk (edge-based))**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. A *walk* from vertex i to vertex j may also be described as a sequence of edges $\gamma = (e_1, e_2, \dots, e_\ell)$ such that each $e_h = (i_{h-1}, i_h) \in \mathcal{E}$, with $i_0 = i$ and $i_\ell = j$.

This naturally leads to the definition of reachability:

Definition A.16 (Reachability)**.** A vertex j is said to be *reachable* from a vertex i in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if there exists a walk from i to j.

The following operation on walks will be useful in the analysis of more complex structures:

Definition A.17 (Concatenation)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Let $\gamma_1 = (i_0, i_1, \dots, i_{\ell_1})$ be a walk from i_0 to i_{ℓ_1}, and let $\gamma_2 = (j_0, j_1, \dots, j_{\ell_2})$ be a walk from j_0 to j_{ℓ_2}, with $j_0 = i_{\ell_1}$. The *concatenation* of γ_1 and γ_2 is the walk $\gamma = \gamma_1 \circ \gamma_2 = (i_0, i_1, \dots, i_{\ell_1}, j_1, \dots, j_{\ell_2})$, of length $\ell_1 + \ell_2$.

We now introduce further restrictions on walks that define important graph-theoretic objects.

Definition A.18 (Trail and Path)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph.

- A *trail* is a walk in which no edge is repeated (i.e. $e_h \neq e_k$ for all $1 \leq h, k \leq \ell$).
- A *path* is a walk in which all vertices are distinct, except possibly the first and last (i.e., $i_h \neq i_k$ for all $0 \leq h, k \leq \ell$, unless $h = 0$ and $k = \ell$).

Clearly, every path is also a trail, but not every trail is a path.

Given two distinct vertices $i \neq j$ in $\mathcal{V}$, we denote by Γ_{ij} the set of paths in $\mathcal{G}$ starting in i and ending in j, and we refer to such paths as *i-j path*. Clearly, the set Γ_{ij} is nonempty if and only if vertex j is reachable from vertex i.

Moreover, for two nonempty disjoint subsets of vertices $\mathcal{O}, \mathcal{D} \subset \mathcal{V}$ s.t. $\mathcal{O} \cap \mathcal{D} = \emptyset$, let $\Gamma_{\mathcal{O}\mathcal{D}} = \bigcup_{o \in \mathcal{O}, d \in \mathcal{D}} \Gamma_{od}$, be the set of all paths in $\mathcal{G}$ starting at a vertex in $\mathcal{O}$ and terminating at a vertex in $\mathcal{D}$. So we can consider the previus case as a particula $\mathcal{O}$-$\mathcal{D}$ paths set whit $\mathcal{O} = \{i\}$ and $\mathcal{D} = \{j\}$.

Definition A.19 (Edge-Path Incidence Matrix)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph and $\mathcal{O}, \mathcal{D} \subset \mathcal{V}$ s.t. $\mathcal{O} \cap \mathcal{D} = \emptyset$. We define the edge-path incidence matrix $G \in \{0{,}1\}^{|\mathcal{E}| \times |\Gamma_{\mathcal{O}\mathcal{D}}|}$ with entries

$$G_{e\gamma} = \begin{cases} 1 & \text{if } e \in \gamma, \\ 0 & \text{if } e \notin \gamma. \end{cases}$$

Two particularly notable classes of paths are the following:

Definition A.20 (Eulerian Trail)**.** An *Eulerian trail* in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a walk that traverses each edge in $\mathcal{E}$ exactly once, regardless of the start and end vertices.

Definition A.21 (Hamiltonian Path)**.** A *Hamiltonian path* in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a path that visits every vertex in $\mathcal{V}$ exactly once.

The following definition introduces two notions of independence in the set Γ_{ij}.

Definition A.22 (Indipendent Walk)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph, and let $i \neq j$ in $\mathcal{V}$ be two distinct vertices. Then, two *i-j* walk $\gamma = (e_1, \dots, e_\ell)$ and $\tilde{\gamma} = (\tilde{e}_1, \dots, \tilde{e}_{\tilde{\ell}})$ in Γ_{ij} are said to be:

- *vertex-independent* if they share no intermediate vertex, i.e., $\{\kappa(e_h) : 1 \leq h < \ell\} \cap \{\kappa(\tilde{e}_h) : 1 \leq h < \tilde{\ell}\} = \emptyset$;
- *edge-independent* if they share no edge, i.e., $e_h \neq \tilde{e}_k$ for every $h = 1, \dots, \ell$ and $k = 1, \dots, \tilde{\ell}$.
- *indipendent* if they are vertex-independent and edge-independent.

We now restrict attention to walks that start and end at the same vertex and define various notions of cycles.

Definition A.23 (Closed Walks and Cycles)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. A walk $\gamma = (i_0, i_1, \dots, i_\ell)$ is said to be *closed* if $i_0 = i_\ell$. Accordingly:

- A *circuit* is a closed trail of length $\ell \geq 1$;
- A *directed cycle* is a closed path of length $\ell \geq 1$;
- A *cycle* is a directed cycle of length $\ell \geq 3$.

Observation 8. *A self-loop $(i, i) \in \mathcal{E}$ constitutes a directed cycle (and hence a circuit), but is not considered a cycle, since its length is 1.*

Similarly, a bidirectional edge between distinct vertices $i \neq j$ gives rise to two directed cycles of length 2, namely (i, j, i) and (j, i, j), which are not cycles under our convention.

Thus, every cycle is a directed cycle, but the converse is not true. This distinction aligns with standard terminology in the literature.

Definition A.24 (Eulerian Circuit and Cycle)**.** An *Eulerian circuit* in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an Eulerian trail that starts and ends at the same vertex.

An *Eulerian cycle* is an Eulerian circuit that is also a path.

Definition A.25 (Hamiltonian Cycle)**.** A *Hamiltonian cycle* (or *Hamiltonian circuit*) in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a Hamiltonian path that starts and ends at the same vertex.

Although every directed cycle is a circuit, not every circuit is a directed cycle.

Lemma A.1. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Then every closed walk of length $\ell \geq 1$ contains a directed cycle.

We want now introduce some notation more approfondita sui concetti walk.

Definition A.26 (Edge-Cycle Incidence Matrix)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we denote by Δ the set of all directed cycles in $\mathcal{G}$ considered up to rotations (e.g., (e_1, e_2, e_3), (e_2, e_3, e_1), and (e_3, e_1, e_2) are identified as one element in Δ) and define the edge-cycle incidence matrix $C \in \{0,1\}^{|\mathcal{E}| \times \Delta}$ with entries

$$C_{e\delta} = \begin{cases} 1 & \text{if } e \in \delta, \\ 0 & \text{if } e \notin \delta, \end{cases}.$$

The following simple result establishes some fundamental relations among the vertex-edge incidence matrix B, the edge-path incidence matrix G, and the edge-cycle incidence matrix C introduced above.

Lemma A.2. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with vertex-edge incidence matrix B and edge-cycle incidence matrix C. Then, $BC = 0$.

Moreover, for any two subsets $\mathcal{O} \subseteq \mathcal{V}$ and $\mathcal{D} \subseteq \mathcal{V}$ such that $\mathcal{O} \cap \mathcal{D} = \emptyset$, let G be the associated edge-path incidence matrix. Then,

$$(BG)_{i\gamma} = \begin{cases} +1 & \text{if } \gamma \text{ starts at } i, \\ -1 & \text{if } \gamma \text{ ends at } i, \\ 0 & \text{if } \gamma \text{ neither starts nor ends at } i, \end{cases}$$

for every vertex $i \in \mathcal{V}$ and path $\gamma \in \Gamma_{\mathcal{O}\mathcal{D}}$.

Example A.5. *Consider the unweighted graph in Figure A.4, with node set $\mathcal{V} = \{o, a, b, c, d\}$, link set $\mathcal{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, and node-link incidence matrix*

$$B = \begin{pmatrix} +1 & +1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & +1 & 0 & -1 & 0 & 0 \\ -1 & 0 & -1 & +1 & 0 & +1 & 0 \\ 0 & 0 & 0 & -1 & +1 & 0 & +1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 \end{pmatrix}.$$

There is a single (up to rotations) directed cycle $\gamma = (e_3, e_4, e_5)$, so that $\Delta = \{\gamma\}$ and the link-cycle incidence matrix is

$$C = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}^\top.$$

On the other hand, there are four o-d paths $\gamma_1 = (e_1, e_6)$, $\gamma_2 = (e_2, e_3, e_6)$, $\gamma_3 = (e_1, e_4, e_7)$, and $\gamma_4 = (e_2, e_3, e_4, e_7))$, so that $\Gamma_{od} = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ and the link-path incidence matrix is

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}^\top.$$

We can now verify the start and end of each path whit

$$BG = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \end{pmatrix}.$$

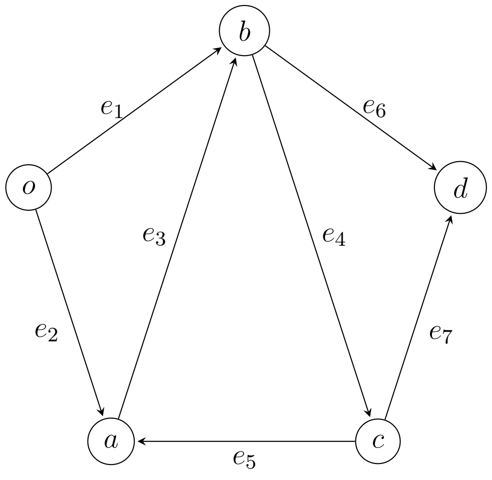


Figure A.4: Example of paths

Having formalized the notion of reachability, we now introduce a metric on the vertex set:

Definition A.27 (Distance)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the *distance* $\mathrm{dist}(i, j)$ between two vertices $i, j \in \mathcal{V}$ is defined as the length of the shortest path from i to j, with the convention that $\mathrm{dist}(i, j) = +\infty$ if no such path exists.

A shortest path between two vertices is called a *geodesic path*.

This notion leads to the concept of strong connectivity:

Definition A.28 (Strong Connectivity)**.** A graph $\mathcal{G}$ is said to be *strongly connected* if every vertex is reachable from every other vertex.

Often, the term "strongly" is omitted, and the graph is simply called *connected*.

Definition A.29 (Connected Components)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, its *connected components* are the maximal subsets $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_k \subseteq \mathcal{V}$ such that for every $i, j \in \mathcal{V}_h$, there exists a path from i to j. These sets form a partition of $\mathcal{V}$:

$$\mathcal{V} = \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_k, \quad \mathcal{V}_h \cap \mathcal{V}_l = \emptyset \text{ for } h \neq l.$$

In undirected graphs, no edge connects vertices in different components. In directed graphs, such inter-component edges may exist.

A useful abstraction for analyzing the global structure of directed graphs is the condensation graph:

Definition A.30 (Condensation Graph)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the *condensation graph* $\mathcal{H}_{\mathcal{G}}$ is obtained by collapsing each connected component into a single *super-vertex*. A directed edge is added between two super-vertices if there exists at least one edge in $\mathcal{G}$ from a vertex in the first component to a vertex in the second.

Lemma A.3. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, its condensation graph $\mathcal{H}_{\mathcal{G}}$ consists of a single vertex if and only if $\mathcal{G}$ is strongly connected.

Example A.6. *Consider the unweighted graph in Figure A.4, with node set* $\mathcal{V} = \{o, a, b, c, d\}$ *and link set* $\mathcal{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$*. The Condensation Graph is in Figure A.4*

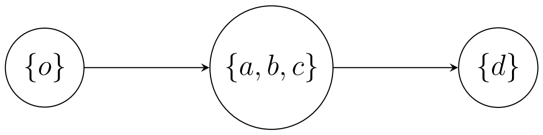


Figure A.5: Condensation graph of the graph in Figure A.4

A.4 Common examples of graphs

A natural notion in graph theory is that of graph isomorphism, which proves useful in many applications. Intuitively, two isomorphic graphs differ only by a relabeling of their vertices.

Definition A.31 (Graph Isomorphism)**.** Given two graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, we say that $\mathcal{G}_1$ and $\mathcal{G}_2$ are *isomorphic* if there exists a bijection $f : \mathcal{V}_1 \to \mathcal{V}_2$ such that $(i, j) \in \mathcal{E}_1 \Leftrightarrow (f(i), f(j)) \in \mathcal{E}_2$.

Isomorphic graphs are often considered equivalent. As a consequence, when dealing with finite graphs, it is customary to identify the vertex set $\mathcal{V}$ with the set $\{1,2,\ldots,n\}$ of the first n positive integers. For instance, complete graphs are uniquely determined up to isomorphism by their order n and are denoted by $\mathcal{K}_n$.

In the remainder of this section, we introduce several standard families of graphs frequently encountered in both theory and applications:

- The *barbell graph* is formed by connecting two disjoint complete graphs via a single edge between one vertex in each (see Figure A.6a).
- The *ring graph* $\mathcal{R}_n$ is a simple connected graph with n vertices, all of which have two edges (see Figure A.6b).
- A *tree* is a simple graph that is connected and cycle-free.
  - The *line graph* $\mathcal{L}_n = (\{1, \ldots, n\}, \mathcal{E})$, where $\mathcal{E} = \{(i, i+1), (i+1, i) \mid i = 1, \ldots, n-1\}$.
  - The *star graph* $\mathcal{S}_n = (\{0,1, \ldots, n\}, \mathcal{E})$, where $\mathcal{E} = \{(0, i), (i,0) \mid i = 1, \ldots, n\}$.

Trees possess several important properties:

Proposition A.2. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a simple connected graph with $n \geq 2$ vertices and $m = |\mathcal{E}|/2$ undirected edges. Then:

- $m \geq n - 1$;
- $\mathcal{G}$ is a tree if and only if $m = n - 1$.

A powerful tool for generating new graph families is the notion of graph product.

Definition A.32 (Graph Product)**.** Given two simple graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, their *product graph* is defined as: $\mathcal{G}_1 \times \mathcal{G}_2 = (\mathcal{V}_1 \times \mathcal{V}_2, \mathcal{E}_1 \otimes \mathcal{E}_2)$, where

$$((i_1, i_2), (j_1, j_2)) \in \mathcal{E}_1 \otimes \mathcal{E}_2 \Leftrightarrow \begin{cases} i_1 = j_1 & \text{and } (i_2, j_2) \in \mathcal{E}_2, \\ i_2 = j_2 & \text{and } (i_1, j_1) \in \mathcal{E}_1. \end{cases}$$

Proposition A.3. Given two simple graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, if both $\mathcal{G}_1$ and $\mathcal{G}_2$ are strongly connected, then their product $\mathcal{G}_1 \times \mathcal{G}_2$ is also strongly connected.

Some illustrative examples include:

- The product of two line graphs $\mathcal{L}_h$ and $\mathcal{L}_k$ yields the *grid graph* with $n = h \cdot k$ vertices (see Figure A.6c).
- The product $\mathcal{L}_2 \times \mathcal{L}_2 \times \mathcal{L}_2$ corresponds to the *cube graph* (see Figure A.6d).
- More generally, for any $k \geq 1$, the product $\mathcal{L}_2^k = \mathcal{L}_2 \times \cdots \times \mathcal{L}_2$ (k times) is called the *hypercube.*
- The product of two ring graphs, $\mathcal{R}_h \times \mathcal{R}_k$, defines a *toroidal grid* (see Figure A.6e).

A.5 Connectivity and Menger's Theorem

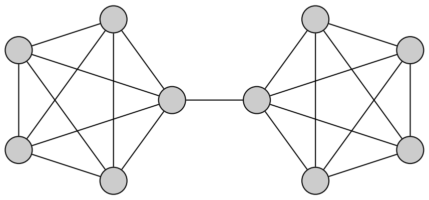
In this section, we introduce the novel concepts of *vertex-connectivity* and *edge-connectivity* of a network. Finally, we present *Menger's Theorem*, which provides an equivalent interpretation of such connectivity measures as the minimum number of vertices or edges that need to be removed from the network in order to disconnect it.

Definition A.33 (Connectivity pair (i,j))**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph, and let $i \neq j$ in V be two distinct vertices. The *vertex-connectivity* $c_{\text{vertex}}(i,j)$ and the *edge-connectivity* $c_{\text{edge}}(i,j)$ of the pair (i,j) are defined as the maximum number of vertex-independent and, respectively, edge-independent i-j paths.

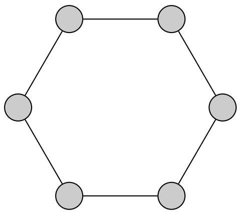
Definition A.34 (k-Edge-Connectivity Graph)**.** A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is said to be k-*edge-connected* if the spanning subgraph of $\mathcal{G}' = \mathcal{G} \setminus \mathcal{E}'$ for every $\mathcal{E}' \subset \mathcal{E}$ s.t. $|\mathcal{E}'| \leq k$ is connect.

Definition A.35 (k-Vertex-Connectivity Graph)**.** A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is said to be k-*vertex-connected* if the induced subgraph $\mathcal{G}\left[\mathcal{V} - \mathcal{V}'\right]$ for every $\mathcal{V}' \subset \mathcal{V}$ s.t. $|\mathcal{V}'| \leq k$ is connect.

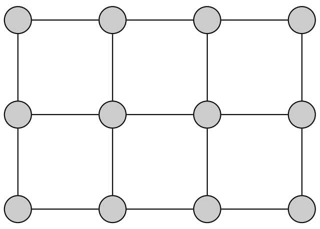
The following result, known as Menger's Theorem, relates the maximum vertex- and edge-connectivity of graph to the correspestly connectivity af all the pair (i,j).



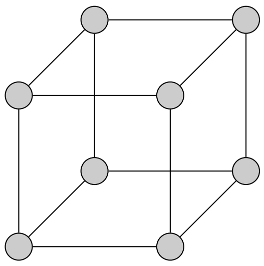
(a) Barbell Graph: two $\mathcal{K}_5$ connected



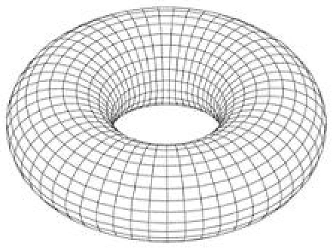
(b) Ring Graph $\mathcal{R}_6$



(c) Grid 4×3



(d) Cube Graph $\mathcal{L}_2 \times \mathcal{L}_2 \times \mathcal{L}_2$



(e) Toroidal Grid Graph

Figure A.6: Examples of standard graph families.

Theorem A.1 (Menger's Theorem)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph and let $i \neq j$ in $\mathcal{V}$ be two distinct vertices. Then:

1. The minimum number of edges that must be removed from $\mathcal{G}$ to make j unreachable from i is equal to $c_{\text{edge}}(i, j)$.

2. If there is no edge directly from i to j, the minimum number of vertices (excluding i and j) that must be removed to make j unreachable from i is equal to $c_{\text{vertex}}(i, j)$.

Menger's Theorem can be interpreted as a special case of a more general result, the *Max-Flow Min-Cut Theorem*, discussed in the following.

Lemma A.4 (Edge-Disjoint Paths)**.** Given a k-edge-connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, for any pair of distinct vertices $u, v \in \mathcal{V}$, there exist at least k edge-disjoint paths connecting them.

So, a graph $\mathcal{G}$ is k-edge-connected if and only if $k \leq \min_{i \neq j \in V} c_{\text{vertex}}(i, j)$.

Lemma A.5 (Vertex-Disjoint Paths)**.** Given a k-vertex-connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, for any two distinct vertices $u, v \in \mathcal{V}$, there exist at least k vertex-disjoint paths between u and v.

We can't say that a graph $\mathcal{G}$ is k-vertex-connected if and only if $k \leq \min_{i \neq j \in V} c_{\text{edge}}(i, j)$: this is true only if $\exists (i, j) = \arg\min_{i \neq j \in V} c_{\text{edge}}(i, j)$ such that there is no edge directly from i to j; if for all $(i, j) = \arg\min_{i \neq j \in V} c_{\text{edge}}(i, j)$ exist an edge $(i, j) \in \mathcal{E}$ the graph is $(k + 1)$-vertex-connected.

Example A.7. *Consider the unweighted graph in Figure A.4, with node set* $\mathcal{V} = \{o, a, b, c, d\}$ *and link set* $\mathcal{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$*.*

G is not connected: it's simple to view from the condensation graph in Figure A.5.

Let $\tilde{G}$ be the undirected graph obtained from G by adding for each directed edge e_i its reverse edge (denoted e_{-i}). Equivalently: treat every link as undirected. Then $\tilde{G}$ is connected (see the condensation graph when edges are made undirected).

Check what happens when deleting any single edge in $\tilde{G}$:

- *remove $e_1 : (o, b)$: path $(o, a, b) = (e_2, e_3)$ still connects o and b.*
- *remove $e_2 : (o, a)$: path $(o, b, c, a) = (e_1, e_4, e_5)$ still connects o and a.*
- *remove $e_3 : (a, b)$: path $(a, o, b) = (e_{-2}, e_{-1})$ still connects a and b.*
- *remove $e_4 : (b, c)$: path $(b, d, c) = (e_6, e_{-7})$ still connects b and c.*
- *remove $e_5 : (c, a)$: path $(c, b, a) = (e_{-4}, e_{-3})$ still connects a and c.*
- *remove $e_6 : (b, d)$: path $(b, c, d) = (e_4, e_7)$ still connects b and d.*
- *remove $e_7 : (c, d)$: path $(c, b, d) = (e_{-4}, e_6)$ still connects c and d.*
- *The same arguments apply if one removes any reverse edge e_{-i} (symmetry): for each removed reverse edge there is a corresponding path using the remaining edges.*

Hence no single edge removal disconnects $\tilde{G}$; therefore $\tilde{G}$ is at least 2-edge-connected. Moreover, a minimum edge cut of size 2 is, for instance, $\{e_6, e_7\}$, which isolates d. Therefore $\tilde{G}$ is not 3-edge-connected.

Removing any single vertex leaves $\tilde{G}$ connected:

- *remove o: path (e_5, e_3, e_6) connects all remaining vertices;*
- *remove a: path (e_1, e_4, e_7) connects all remaining vertices;*
- *remove b: path (e_2, e_{-5}, e_7) connects all remaining vertices;*
- *remove c: path (e_2, e_3, e_6) connects all remaining vertices;*
- *remove d: path (e_1, e_4, e_5) connects all remaining vertices.*

Thus no single vertex removal disconnects the graph, so $\tilde{G}$ is 2-vertex-connected. On the other hand, removing $\{b, c\}$ disconnects d from $\{a, o\}$ (only the edges $\{e_2, e_{-2}\}$ remains), so $\tilde{G}$ is not 3-vertex-connected.

A.6 Network flows

The purpose of this section is to make the reader acquainted with the notion of network flows. This is a very useful concept in many applications and will recur frequently in the following chapters of these notes.

Definition A.36 (Exogenous Net Flow)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an *exogenous net flow* on $\mathcal{G}$ is a vector $\nu \in \mathbb{R}^{|\mathcal{V}|}$ satisfying the *zero-sum constraint*:

$$\sum_{i \in \mathcal{V}} \nu_i = 0. \tag{A.1}$$

We make a number of basic considerations on the above definition:

- The positive part $[\nu_i]^+ = \max\{0, \nu_i\}$ and negative part $[\nu_i]^- = \max\{0, -\nu_i\}$ of the net flow at vertex i are to be interpreted as the exogenous inflow into and, respectively, the external outflow from i. The zero-sum constraint of then implies that the total exogenous inflow equals the total external outflow.
- We define the *throughput* as
$$\upsilon = \frac{1}{2} \sum_{i \in \mathcal{V}} |\nu_i| = \sum_{i \in \mathcal{V}} [\nu_i]^+ = \sum_{i \in \mathcal{V}} [\nu_i]^-. \tag{A.2}$$
- Depending on the application, vertices i such that $\nu_i > 0$ are called sources, origins, or generators, whereas vertices i such that $\nu_i < 0$ are called sinks, destinations, or loads. We denote by
$$\mathcal{O}_\nu = \{i \in \mathcal{V} : \nu_i > 0\}, \qquad \mathcal{D}_\nu = \{i \in \mathcal{V} : \nu_i < 0\}, \tag{A.3}$$
the sets of origins and destinations associated with the exogenous net flow vector ν.

Definition A.37 (Network Flow)**.** Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and an exogenous net flow $\nu \in \mathbb{R}^{|\mathcal{V}|}$ satisfying the zero-sum constraint, a *network flow* is a nonnegative vector $f \in \mathbb{R}_+^{|\mathcal{E}|}$ whose components f_e satisfy the *flow balance equations*:

$$\nu_i + \sum_{e \in \partial_i^-} f_e = \sum_{e \in \partial_i} f_e, \qquad \forall i \in \mathcal{V}. \tag{A.4}$$

The term f_e represents the flow along edge $e \in \mathcal{E}$. Flow balance equations states that the total inflow into vertex i - from both the exogenous inflow $[\nu_i]^+$ and the flows f_e on incoming edges - equals the total outflow from i - from both the exogenous outflow $[\nu_i]^-$ and outgoing flows.

The flow balance equations can be compactly written using the vertex-edge incidence matrix B:

$$(Bf)_i = \sum_{e \in \mathcal{E}} B_{ie} f_e = \sum_{e \in \partial_i^-} f_e - \sum_{e \in \partial_i} f_e, \ \forall i \in \mathcal{V} \tag{A.5}$$

so the costrain is equivalent to

$$Bf = \nu. \tag{A.6}$$

In the special case where there is a single origin o and a single destination d with d reachable from o, and an exogenous net flow $\nu = \upsilon(\delta(o) - \delta(d))$ for some throughput value $\upsilon \geq 0$, equation upper becomes

$$Bf = \upsilon_o - \upsilon_d = \upsilon(\delta^{(o)} - \delta^{(d)}). \tag{A.7}$$

The flows f satisfying this equation are called *o-d flows.*

Example A.8. *Consider the unweighted graph in Figure A.4, with node set* $\mathcal{V} = \{o, a, b, c, d\}$ *and link set* $\mathcal{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$.

Take o as the single origin and d as the single destination, and set the throughput $\upsilon = 3$. Thus the exogenous net flow vector is $\nu = \upsilon(\delta^{(o)} - \delta^{(d)}) = 3(\delta^{(o)} - \delta^{(d)})$.

The following network flow $f \in \mathbb{R}_+^7$, ordered as $(f_{e_1}, \ldots, f_{e_7})$, satisfies the balance constraints and is therefore a feasible $o-d$ flow of value $\upsilon = 3$:

$$f = (2,\ 1,\ 1,\ 1,\ 0,\ 2,\ 1).$$

In fact, with the ordering o, a, b, c, d, the balances at the vertices are:

- *At the origin o: outgoing flow $f_{e_1} + f_{e_2} = 2 + 1 = 3$, no inflow, hence net supply $+3 = \nu_o$.*
- *At vertex a: inflow $f_{e_2} + f_{e_5} = 1 + 0 = 1$, outflow $f_{e_3} = 1$, balance holds.*
- *At vertex b: inflow $f_{e_1} + f_{e_3} = 2 + 1 = 3$, outflow $f_{e_4} + f_{e_6} = 1 + 2 = 3$.*
- *At vertex c: inflow $f_{e_4} = 1$, outflow $f_{e_5} + f_{e_7} = 0 + 1 = 1$.*
- *At the destination d: inflow $f_{e_6} + f_{e_7} = 2 + 1 = 3$, no outflow, hence net demand $-3 = \nu_d$.*

Equivalently, if B denotes the vertex–edge incidence matrix consistent with the vertex ordering o, a, b, c, d and the above edge orientation, then $Bf = \nu$.

Flows can be constructed using the edge-path incidence matrix G.
In the single origin-destination case, given an *o-d* path γ, Lemma A.2 yields

$$f = G\delta^{(\gamma)} \Rightarrow Bf = BG\delta^{(\gamma)} = \delta^{(o)} - \delta^{(d)}. \tag{A.8}$$

Hence, $f = G\delta^{(\gamma)}$ is a unitary*o-d* flow supported on the edges of γ. This implies that $f = \upsilon G\delta^{(\gamma)}$ solves the costrain of *o-d* flows.

In the general multi-origin and multi-destination case, the hyperplane defined $\mathcal{H} = \left\{\nu \in \mathbb{R}^{|\mathcal{V}|} : \sum \nu_i = 0\right\}$ satisfies $\mathcal{H} = \operatorname{span}\left\{\delta^{(i)} - \delta^{(j)} : i, j \in \mathcal{V}\right\}$, so every exogenous net flow ν can be written as

$$\nu = \sum_{o\in\mathcal{O}_\nu}\sum_{d\in\mathcal{D}_\nu} \upsilon_{od}(\delta^{(o)} - \delta^{(d)}), \tag{A.9}$$

with $\upsilon_{od} > 0$ for all (o, d) pairs. For each such pair, choose a path $\gamma_{od} \in \Gamma_{od}$, and set

$$f = Gz, \quad z = \sum_{o\in\mathcal{O}_\nu}\sum_{d\in\mathcal{D}_\nu} \upsilon_{od}\delta^{(\gamma_{od})}. \tag{A.10}$$

Moreover, Lemma A.2 implies that the columns of C represent network flows of throughput zero. Therefore, all solutions to $Bf = \nu$ are of the form

$$f = Gz + Cy, \tag{A.11}$$

with $y \in \mathbb{R}_+^{|\Delta|}$. This decomposition yields the following fundamental result:

Theorem A.2 (Flow Decomposition Theorem)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with vertex-edge incidence matrix B and edge-cycle incidence matrix C and edge-path incidence matric G. Then, for every network flow $f \in \mathbb{R}_+^{|\mathcal{E}|}$ associated with an exogenous net flow $\nu = Bf \in \mathbb{R}^{|\mathcal{V}|}$, there exist $z \in \mathbb{R}_+^{|\Gamma_{\mathcal{O}_\nu\mathcal{D}_\nu}|}$ and $y \in \mathbb{R}_+^{|\Delta|}$ such that

$$f = Gz + Cy. \tag{A.12}$$

A.7 Capacity and Max-Flow Min-Cut Theorem

In this section, we study flows that satisfy capacity constraints on the links. Precisely, from now on, we consider (weighted) graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$, where for every link $e \in \mathcal{E}$, the value $c_e > 0$ is called the *capacity* of link e, and represents the maximum flow allowed through it. In this and the next section, we refer to such graphs as *capacitated graphs*.

A natural question is to characterize the maximum throughput ν from a given node o to another node d that can be achieved by a network flow f without violating any link capacity constraints, i.e., such that $f \leq c$.

Example A.9. *For the sake of illustration we assign specific maximum capacities to the edges, namely*

$$u_{e_1} = 2, \quad u_{e_2} = 2, \quad u_{e_3} = 1, \quad u_{e_4} = 1, \quad u_{e_5} = 1, \quad u_{e_6} = 2, \quad u_{e_7} = 1.$$

These values are not unique but are chosen so that the origin o can inject a total of three units of flow into the network, which can then be consistently routed through the intermediate vertices to reach the destination d.

We now formalize the problem of interest.

Definition A.38 (Maximum Flow Problem)**.** Given a capacitated graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ and two distinct nodes $o \neq d \in \mathcal{V}$, the *maximum flow problem* is the constrained maximization problem:

$$\nu^*_{od} = \max_{\nu \geq 0} \quad \nu \qquad \text{subject to} \quad \begin{cases} Bf = \nu(\delta^{(o)} - \delta^{(d)}), \\ 0 \leq f \leq c, \end{cases} \tag{A.13}$$

We refer to ν^*_{od} as the *maximum throughput* from o to d.

Observe that problem (A.13) is a linear program: both the objective function and the constraints are linear in the variables. The objective function corresponds to the total flow $\nu \geq 0$ entering node o and exiting node d. The constraints ensure flow conservation at all nodes and that the flow remains within capacity bounds. Network flows that satisfy the capacity constraint are called *feasible*. Notice that the set of feasible flows is never empty, as it trivially contains the null flow $f = 0$ with throughput $\nu = 0$.

How can we guarantee that a network flow achieves the maximum throughput from a source node o to a destination node d? The *Max-Flow Min-Cut Theorem* provides an answer by relating the value of the optimal throughput ν^*_{od} to certain geometric properties of the graph $\mathcal{G}$.

We begin by introducing a fundamental concept.

Definition A.39 (Cut and Cut Capacity)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ be a graph and $o \neq d \in \mathcal{V}$. Then:

- An *o-d cut* is a partition of the node set $\mathcal{V}$ into two subsets $\mathcal{U}$ and $\mathcal{U}^c = \mathcal{V} \setminus \mathcal{U}$, such that $o \in \mathcal{U}$ and $d \in \mathcal{U}^c$. We identify a cut with the subset $\mathcal{U}$ containing the source node o.

- The *capacity* of the cut $\mathcal{U}$ is defined as:

$$c_{\mathcal{U}} := \sum_{e \in \partial \mathcal{U}} c_e, \tag{A.14}$$

  where $\partial \mathcal{U}$ denotes the set of edges crossing from $\mathcal{U}$ to $\mathcal{U}^c$.

Visually, an o-d cut can be represented by a separating line in the network, with the source on one side (say, the left) and the destination on the other (say, the right). The capacity of the cut is simply the sum of the capacities of the links crossing from left to right.

The Max-Flow Min-Cut Theorem states that the maximum throughput from o to d equals the minimum capacity over all o-d cuts.

Definition A.40 (Min-Cut Capacity)**.** Fix a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ and two distinct nodes $o \neq d \in \mathcal{V}$. The *min-cut capacity* of the network is the value:

$$c^*_{od} := \min_{\substack{\mathcal{U} \subseteq \mathcal{V} \\ o \in \mathcal{U},\, d \notin \mathcal{U}}} c_{\mathcal{U}}. \tag{A.15}$$

Any cut $\mathcal{U}$ achieving $c_{\mathcal{U}} = c^*_{od}$ is called a *bottleneck.*

This leads to the main result of the section:

Theorem A.3 (Max-Flow Min-Cut Theorem)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ be a graph with capacity vector $c > 0$, and let $o \neq d \in \mathcal{V}$. Then:

$$\nu^*_{od} = c^*_{od}. \tag{A.16}$$

Moreover, if all link capacities are integers, then there exists a feasible flow achieving the maximum throughput in which the flow on every edge is an integer.

Remark. From a computational perspective, the maximum flow problem can be solved using algorithms with different complexities, depending on the adopted strategy. The running time ranges from $\mathcal{O}(|\mathcal{V}||\mathcal{E}|^2)$ to $\mathcal{O}(|\mathcal{V}||\mathcal{E}| \log |\mathcal{V}|)$, according to the specific method employed. Some algorithms provide strongly polynomial guarantees, while more sophisticated ones achieve improved efficiency through advanced optimization techniques and dynamic data structures.

Remark. Theorem A.16 admits a dual interpretation: the minimum total capacity that a hypothetical adversary must remove from the network to disconnect d from o equals c^*_{od}. Indeed, removing all edges of a minimum capacity o-d cut yields c^*_{od} and no feasible flow from o to d exists anymore. Conversely, removing a total capacity less than c^*_{od} leaves at least one feasible flow with positive throughput.

Example A.10. *Consider $o-d$ cuts that separate o from d. The cut formed by the two edges leaving o, namely $\{e_1, e_2\}$, has capacity $u_{e_1} + u_{e_2} = 2 + 2 = 4$. The cut separating $\{o, a, b, c\}$ from $\{d\}$ consists of the two edges entering d, $\{e_6, e_7\}$, whose total capacity is $u_{e_6} + u_{e_7} = 2 + 1 = 3$. By the max-flow/min-cut theorem the value of any feasible $o-d$ flow cannot exceed 3; the flow constructed above attains 3, hence it is a maximum flow and the cut $\{e_6, e_7\}$ is a minimum cut.*

We now generalize the Max-Flow Min-Cut Theorem to the case of multiple origins and destinations (still with a single commodity):

Corollary A.3.1. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, c)$ be a capacitated graph and let $\nu \in \mathbb{R}^{|\mathcal{V}|}$ be a vector such that $\sum_{i \in \mathcal{V}} \nu_i = 0$. Then, a feasible network flow f with exogenous net-flow vector ν exists if and only if:

$$\sum_{i \in \mathcal{U}} \nu_i \leq c_{\mathcal{U}}, \quad \forall \mathcal{U} \subseteq \mathcal{V}.$$

We conclude by extending this corollary to the case where some link capacities are infinite. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph and $c \in (0, +\infty]^{|\mathcal{E}|}$. Then, for every zero-sum vector

$\nu \in \mathbb{R}^{|\mathcal{V}|}$, by applying Corollary to the truncated vector $\hat{c} = \min\{c, M\}$ for large enough $M = \sum_i |\nu_i|$, we recover the necessary and sufficient condition for feasibility:

$$\sum_{i \in \mathcal{U}} \nu_i \le c_{\mathcal{U}}, \quad \forall \mathcal{U} \subseteq \mathcal{V}.$$

Corollary A.3.2. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph and let $\nu \in \mathbb{R}^{|\mathcal{V}|}$. Then, a network flow with exogenous net-flow vector ν exists if and only if:

- $\sum_{i \in \mathcal{V}} \nu_i = 0$,
- every subset of nodes $\mathcal{U} \subseteq \mathcal{V}$ such that $\sum_{i \in \mathcal{U}} \nu_i > 0$ is not trapping.

Bibliography

F. Balbo, R. Mandiau, and M. Zargayouna. Extended review of multi-agent solutions to advanced public transportation systems challenges. *Public Transport*, 16(1):159–186, 2024. doi: 10.1007/s12469-023-00332-9. URL https://link.springer.com/article/10.1007/s12469-023-00332-9. Survey of MAPF and MAS applications to public transportation.

Sourabh Bhattacharya and Vijay Kumar. Distributed optimization with pairwise constraints and its application to multi-robot path planning. In *Proceedings of Robotics: Science and Systems (RSS)*, volume 6, pages 177–184, 2011. doi: 10.15607/RSS.2011.VII.023. Distributed optimization framework for multi-robot planning.

Zohar Bnaya, Roni Stern, Ariel Felner, Roi Zivan, and Shigeo Okamoto. Multi-agent path finding for self-interested agents. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS)*, pages 47–55. AAAI Press, 2013. MAPF for self-interested agents in game-theoretic contexts.

RexCharles Donatus, Kumater Ter, Ore-Ofe Ajayi, and Daniel Udekwe. Multi-agent reinforcement learning in intelligent transportation systems: A comprehensive survey. *arXiv*, 2025. URL https://arxiv.org/abs/2508.20315. Survey of MARL for ITS applications.

Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31:591–656, 2008. ISSN 1076-9757. doi: 10.1613/jair.2502. Early application of MAPF to autonomous traffic intersections.

David K. Grady, Kostas E. Bekris, and Lydia E. Kavraki. Asynchronous distributed motion planning with safety guarantees under second-order dynamics. In *Algorithmic Foundations of Robotics IX*, volume 68 of *Springer Tracts in Advanced Robotics*, pages 53–70. Springer, 2011. ISBN 978-3-642-17451-3. doi: 10.1007/978-3-642-17452-0_4. Distributed motion planning with second-order safety guarantees.

Florian Grenouilleau, Willem-Jan van Hoeve, and J. N. Hooker. A multi-label a* algorithm for multi-agent pathfinding. In *Proceedings of the 2019 International Conference on Automated Planning and Scheduling (ICAPS)*, 2019. URL https://www.andrew.cmu.edu/user/vanhoeve/papers/MAPD_ICAPS_2019.pdf. MLA* algorithm for MAPF.

Yu Jing and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3612–3617, 2013.

Samuel Kloder and Seth Hutchinson. Path planning for permutation-invariant multirobot formations. *IEEE Transactions on Robotics*, 22(4):650–665, 2006. ISSN 1552-3098. doi: 10.1109/TRO.2006.878935. Focus on multirobot formation planning invariant to agent permutation.

Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. In *arXiv*, 2021. URL `https://par.nsf.gov/servlets/purl/10285107`. Lifelong MAPF applied to large-scale warehouse environments.

Jiarui Li, Alessandro Zanardi, and Gioele Zardini. Multi-agent path finding via finite-horizon hierarchical factorization. *arXiv*, 2025. URL `https://arxiv.org/abs/2505.07779`. Finite-horizon hierarchical MAPF approach.

Sam Loyd. *Mathematical Puzzles of Sam Loyd*. Dover Math Games & Puzzles. Dover Publications, New York, dover reprint edition, June 1 1959. ISBN 0486204987. Classic puzzle book by Sam Loyd, reprinted by Dover.

Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1144–1152, 2016.

Hang Ma and Sven Koenig. Ai buzzwords explained: Multi-agent path finding (mapf). *ACM SIGAI Newsletter*, 8(2):3–9, 2017. ISSN 2372-3483. doi: 10.1145/3137574.3137579. URL `https://doi.org/10.1145/3137574.3137579`. Column explaining MAPF terminology for practitioners.

Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of the Sixteenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 837–845. IFAAMAS, 2017. ISBN 978-1-4503-4918-2. Extends MAPF to lifelong online task assignment.

Hang Ma, Guni Wagner, Ariel Felner, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding with deadlines. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI)*, pages 417–423. IJCAI Organization, 2018. doi: 10.24963/ijcai.2018/58. Addresses MAPF problems with strict deadline constraints.

Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nancy Ayanian, and Sven Koenig. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI-19)*, pages 7651–7658. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33017651. Incorporates kinematic constraints in lifelong MAPF.

Hang Ma, Guni Wagner, Ariel Felner, Jiaoyang Li, and T. K. Satish Kumar. Learning multi-agent coordination for emergency response. In *Proceedings of the 2025 ACM Conference on Artificial Intelligence*, 2025. doi: 10.1145/3730436.3730491. URL https://dl.acm.org/doi/10.1145/3730436.3730491. Focus on multi-agent coordination in emergency scenarios.

Hiroya Makino and Seigo Ito. Mapf-hd: Multi-agent path finding in high-density environments. *arXiv preprint arXiv:2509.06374*, 2025. URL https://arxiv.org/abs/2509.06374. High-density MAPF for real robot dynamics.

T. Pulikottil. Agent-based manufacturing — review and expert evaluation. *The International Journal of Advanced Manufacturing Technology*, 115(1-2):1–19, 2023. doi: 10.1007/s00170-023-11517-8. URL https://link.springer.com/article/10.1007/s00170-023-11517-8. Survey of agent-based manufacturing systems.

M. Reda. Path planning algorithms in the autonomous driving system. *ScienceDirect*, 2024. URL https://www.sciencedirect.com/science/article/pii/S0921889024000137. Overview of path planning methods in autonomous driving.

Malcolm R. K. Ryan. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence Research*, 31:497–542, 2008. ISSN 1076-9757. doi: 10.1613/jair.2676. URL https://jair.org/index.php/jair/article/view/10539. Focus on subgraph decomposition for efficient multi-robot planning.

William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879. ISSN 0002-9327. doi: 10.2307/2369492. URL https://www.jstor.org/stable/2369492. Early study of the 15-puzzle.

Pavel Surynek. An optimization variant of multi-robot path planning is intractable. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 1261–1263. AAAI Press, 2010. ISBN 978-1-57735-409-4. doi: 10.5555/2898607.2898808. URL https://ojs.aaai.org/index.php/AAAI/article/view/7767. Computational complexity analysis of MAPF optimization.

Pavel Surynek, Ariel Felner, Roni Stern, and Evgeny Boyarski. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search (SoCS)*, pages 145–153. AAAI Press, 2016. URL https://ojs.aaai.org/index.php/SOCS/article/view/18273. Empirical evaluation of MAPF difficulty metrics.

Jan Svancara, Petr Váňa, Tomas Musil, and Pavel Surynek. Online multi-agent pathfinding. In *Proceedings of the Eighteenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1891–1893. IFAAMAS, 2019. Focus on online real-time MAPF methods.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv*, 2025. URL https://arxiv.org/abs/2501.06322. LLM-based multi-agent collaboration mechanisms survey.

Jing Yu and Steven M. LaValle. Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X*, volume 86 of *Springer Tracts in Advanced Robotics*, pages 157–173. Springer, 2013. ISBN 978-3-642-36278-1. doi: 10.1007/978-3-642-36279-8_10. Linking MAPF to network flow optimization techniques.

Xiaoyang Zhong et al. Optimal target assignment and path finding for teams of agents. *arXiv preprint arXiv:2208.01222*, 2022.