Filippo Bistaffa

Constraint Optimisation Techniques for Real-World Applications

Ph.D. Thesis

XXVIII Cycle (January 2013–December 2015)

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
Prof. Alessandro Farinelli

ISBN: **9788869250132**
Series N°: **TD-01-16**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Abstract

Constraint optimisation represents a fundamental technique that has been successfully employed in Multi-Agent Systems (MAS) in order to face a number of multi-agent coordination challenges. In this thesis we focus on *Coalition Formation* (CF), one of the key approaches for coordination in MAS. CF aims at the formation of groups that maximise a particular objective functions (e.g., arrange shared rides among multiple agents in order to minimise travel costs). Specifically, we discuss a special case of CF known as Graph-Constrained CF (GCCF) where a network connecting the agents constrains the formation of coalitions. We focus on this type of problem given that in many real-world applications, agents may be connected by a communication network or only trust certain peers in their social network. In particular, the contributions of this thesis are the following.

We propose a novel representation of this problem and we design an efficient solution algorithm, i.e., CFSS. We evaluate CFSS on GCCF scenarios like *collective energy purchasing* and *social ridesharing* using realistic data (i.e., energy consumption profiles from households in the UK, GeoLife for spatial data, and Twitter as social network). Results show that CFSS outperforms state of the art GCCF approaches both in terms of runtime and scalability. CFSS is the first algorithm that provides solutions with good quality guarantees for large-scale GCCF instances with thousands of agents (i.e., more that 2700).

In addition, we address the problem of computing the transfer or payment to each agent to ensure it is fairly rewarded for its contribution to its coalition. This aspect of CF, denoted as *payment computation*, is of utmost importance in scenario characterised by agents with rational behaviours, such as collective energy purchasing and social ridesharing. In this perspective, we propose PRF, the first method to compute payments in large-scale GCCF scenarios that are also stable in a game-theoretic sense.

Finally, we provide an alternative method for the solution of GCCF, by exploiting the close relation between such problem and Constraint Optimisation Problems (COPs). We consider Bucket Elimination (BE), one of the most important algorithmic frameworks to solve COPs, and we propose CUBE, a highly-parallel GPU implementation of the most computationally intensive operations of BE. CUBE adopts an efficient memory layout that results in a high computational throughput. In addition, CUBE is not limited by the amount of memory of the GPU and,

hence, it can cope with problems of realistic nature. CUBE has been tested on the SPOT5 dataset, which contains several satellite management problems modelled as COPs. Moreover, we use CUBE to solve COP-GCCF, the first COP formalisation of GCCF that results in a linear number of constraints with respect to the number of agents. This property is crucial to ensure the scalability of our approach. Results show that COP-GCCF produces significant improvements with respect to state of the art algorithms when applied to a realistic graph topology (i.e., Twitter), both in terms of runtime and memory.

Overall, this thesis provides a novel perspective on important techniques in the context of MAS (such as CF and constraint optimisation), allowing to solve realistic problems involving thousands of agents for the first time.

Contents

1 Introduction 1

Part I Background & Related Work

2 Background 15
2.1 Coalition formation 15
2.1.1 Coalition structure generation 16
2.1.2 Graph-constrained coalition formation 16
2.1.3 Payment computation 18
2.1.4 The kernel 19
2.2 Constraint optimisation problems 20
2.3 Bucket elimination 22
2.3.1 Composition 24
2.3.2 Marginalisation 26
2.4 Belief propagation on junction trees 27
2.5 Graphics processing units 33
2.5.1 Memory management 34
2.5.2 Pipelining 36

3 Related work 37
3.1 Team formation 37
3.2 CSG algorithms 37
3.2.1 Complete approaches 38
3.2.2 Constrained approaches 39
3.2.3 Graph-constrained approaches 41
3.2.4 Approaches based on special characteristic function representations 42
3.2.5 Heuristic approaches 43
3.3 Computing payments in the ϵ-kernel 43
3.4 Constraint optimisation problems 44
3.5 Belief propagation 45

Part II Graph-Constrained Coalition Formation

4 CFSS: a branch-and-bound algorithm for GCCF 49
4.1 Generating feasible coalition structures via edge contractions 49
4.2 Generating the entire search space 51
4.3 $m + a$ functions 54
4.4 The CFSS algorithm 55
4.4.1 Edge ordering heuristic 57
4.4.2 Anytime approximate properties 58

5 Applications for GCCF 61
5.1 Collective energy purchasing 61
5.2 Edge sum with coordination cost 62
5.3 Coalition size with distance cost 64
5.4 Empirical evaluation 65
5.4.1 DyCE vs. CFSS: runtime comparison 67
5.4.2 Bounding technique effectiveness 69
5.4.3 Edge ordering heuristic 69
5.4.4 Anytime approximate performance 69
5.4.5 CFSS vs. C-Link: solution quality comparison 73
5.4.6 P-CFSS 74

Part III Cardinality-Constrained Coalition Formation

6 Social Ridesharing 77
6.1 Problem definition 79
6.1.1 Coalitional value definition 80
6.1.2 Optimal path computation 81
6.2 CFSS for the SR problem 83
6.2.1 Bound computation 84
6.3 Introducing time constraints 89
6.3.1 Optimal departure time computation 90
6.3.2 Time infeasible coalitions 92
6.3.3 Bound computation 94
6.4 Empirical evaluation 94
6.4.1 Social welfare improvement without time constraints 96
6.4.2 Social welfare improvement with time constraints 97
6.4.3 Runtime performance without time constraints 100
6.4.4 Runtime performance with time constraints 100
6.4.5 Approximate performance 102
6.4.6 SR-CFSS vs. C-Link: solution quality comparison 103

7 Payments for Social Ridesharing . . . 105
7.1 State of the art approach . . . 105
7.2 The PRF algorithm . . . 107
7.2.1 P-PRF . . . 110
7.3 Empirical evaluation . . . 111
7.3.1 Runtime performance . . . 111
7.3.2 Comparison with the state of the art . . . 112
7.3.3 Parallel performance . . . 113
7.3.4 Costs and network centrality without time constraints . . . 114
7.3.5 Costs and network centrality with time constraints . . . 115

Part IV Bucket Elimination on GPUs

8 CUBE: a CUDA implementation for Bucket Elimination . . . 119
8.1 Processing complete tables . . . 121
8.1.1 Table preprocessing . . . 121
8.1.2 GPU kernel for BP . . . 127
8.1.3 GPU kernel for COPs . . . 129
8.2 Processing incomplete tables . . . 134
8.2.1 Table preprocessing . . . 134
8.2.2 Join sum ($\oplus$) . . . 136
8.2.3 Maximisation ($\Downarrow$) . . . 137
8.3 Data transfers . . . 138
8.3.1 Global-shared memory transfers . . . 138
8.3.2 Host-device data transfers . . . 139
8.4 Experimental evaluation . . . 140
8.4.1 BP on JTs . . . 140
8.4.2 COPs with complete tables . . . 141
8.4.3 COPs with incomplete tables . . . 143

9 A COP model for Graph-Constrained Coalition Formation . . . 147
9.1 Variables . . . 147
9.2 The simple approach . . . 148
9.3 COP-GCCF . . . 148
9.3.1 Required variables . . . 149
9.3.2 Constructing constraint functions . . . 152
9.3.3 Reducing the size of constraint functions . . . 154
9.4 Experimental evaluation . . . 157
9.4.1 Experimental methodology . . . 157
9.4.2 Runtime . . . 157
9.4.3 Memory . . . 158

10 Conclusions and future work . . . 161

1

Introduction

Constraint optimisation [31] represents a fundamental technique that allows to address a wide variety of optimisation problems in several contexts. Constraint optimisation techniques have been successfully employed in Multi-Agent Systems (MAS) [21] to face a number of multi-agent coordination challenges such as planning, scheduling, resource allocation [53, 74] and satellite management [9]. In this thesis we focus on *coalition formation*, a particular application of constraint optimisation that represents one of the key approaches for coordination in MAS [21].

1.1 Coalition formation

Coalition Formation (CF) [95] aims at establishing collaborations in MAS with multiple entities provided with common or individual objectives. It involves the coming together of multiple, possibly heterogeneous, agents into groups, called *coalitions*, in order to achieve either their individual or collective goals, whenever they cannot do so on their own. For instance, in mobile sensor network applications, the optimal approach to patrol an area suggests to combine capabilities of different types of unmanned vehicles (aerial, ground, underwater) rather than using only one type. While this example involves agents that are fully cooperative (i.e., will forego their own benefit for the common good), in many cases, the agents act in a self-interested way (e.g., in collective energy purchasing where each user is meant to pay a fair price for its energy consumption), posing the problem of splitting the reward/cost resulting from the collaboration.

Building upon the seminal work of Shehory and Kraus [102], Sandholm et al. [95] identify the key computational tasks involved in the CF process. As a first step, it is necessary to define a performance measure for each group of agents by means of a particular function, called *characteristic function*, which associates each possible coalition to a value. Such value quantifies how well or bad a given group of agents performs together, and it is strongly related to the specific domain of application. For example, in the collective energy purchasing scenario [14, 42, 112] the value associated to a group of agents is the total cost of the energy that they consume as a collective. In ridesharing contexts [17, 18], coalitions represent groups of commuters sharing the same car in order to travel together and reduce

costs, hence the coalitional value is defined as the total travel expenses (e.g., fuel consumption, time costs) of a given car. In task assignment scenarios [68], the characteristic function models the reward associated to the tasks that are executed by a particular team of agents.

The classic formulation of CF assumes that the values of all the coalitions are stored in memory as a table [95]. Even if this assumption provides several advantages in terms of generality (i.e., the characteristic function can be of any form) and runtime performance (i.e., values can be retrieved in constant time), it suffers from one fundamental drawback that has prevented the application of CF techniques in real-world applications. In fact, this approach requires an exponential amount of memory to store all the values associated to the 2^n possible coalitions, which represent the actual input to the CF problem.

Once the coalitional values have been defined, the second fundamental aspect of CF involves the computation of the best *partition* (denoted as *coalition structure*) of the set of agents, i.e., the one that maximises (or minimises, in the case of a minimisation problem) the sum of the values of the associated coalitions. Such problem is denoted as *Coalition Structure Generation* (CSG), and it is equivalent to the *complete set partitioning* problem [119]. As a consequence, the number of possible coalition structures is equal to the number of ways in which a set of n agents can be partitioned, i.e., the n^{th} Bell number. Such quantity grows exponentially with respect to n (i.e., $\Omega((\frac{n}{\ln(n)})^n)$ [10]), therefore the CSG problem is generally untractable for large-scale instances. In addition, notice that CSG inherits the space complexity of the above discussed characteristic function representation. For this reason, state of the art algorithms that solve the general CSG problem are limited to a few tens of agents, and, to the best of our knowledge, they have never been applied to realistic CF scenarios.

The final step of the CF process, i.e., *payment computation*, aims at finding the transfer or payment to each agent to ensure it is adequately rewarded for its contribution to its coalition. As an example, in scenarios in which each coalitional value represents a collective cost (e.g., collective energy purchasing or ridesharing), payments dictate how such cost should be partitioned among the members the coalition. Payments play a crucial role in environments involving agents that are not fully cooperative, i.e., they are interested in the maximisation of their private benefit rather than acting for the common good. As such, payments have to be distributed to the agents according to their bargaining power [22].

These topics have been extensively studied in the cooperative-game theory literature, and a crucial concept in these settings is *stability*. In particular, stability ensures that agents will not deviate from the provided coalitions to different ones that are better from their individual point of view. In cooperative game-theory, stability has been defined with several concepts, including the stable set, the nucleous, the kernel, and the core [22]. The *core* is one of the most widely studied stability concepts, since it ensures a particularly strong and useful property that guarantees that no coalition can improve upon the considered payment allocation. However, computing payments that are core-stable has an exponential complexity with respect to the number of agents, and hence, it is not suitable for large-scale systems. Moreover, it is not guaranteed that core-stable solutions always exist [22].

The challenges discussed above represent the main objects of study this thesis, whose main contributions will be discussed in the following sections.

1.2 Graph-constrained CF

The standard formulation of the above discussed problems assumes that every coalition is valid and can be part of the final solution. However, in many real-world applications, there are constraints that may limit the formation of some coalitions [91, 114, 115]. For instance, anti-trust laws prohibit the formation of certain coalitions of companies to prevent oligopolies, or cardinality constraints may be introduced to either prohibit or allow coalitions of certain sizes [102]. Following the work of Myerson [79] and Demange [35], and more recent work by Voice et al. [115], in this thesis we focus on a specific type of constraints that encodes synergies or relationships among the agents and that can be expressed by a graph, where nodes represent agents and edges encode the relationships between the agents. In this setting, edges enable connected agents to form a coalition and a coalition is considered feasible only if its members represent the vertices of a connected subgraph. Such constraints are present in several real-world scenarios, such as social or trust constraints (e.g., energy consumers who prefer to group with their acquaintances in forming energy cooperatives [14, 112], or commuters sharing rides with their friends [17, 18]), and physical constraints (e.g., emergency responders may join specific teams in disaster scenarios where only certain routes are available). Hereafter, we shall refer to the CF problem where coalitions are encoded by means of graphs as *Graph-Constrained Coalition Formation* (GCCF). Notice that the addition of these constraints does not lower the complexity of the problem. In particular, Voice et al. [114] show that the GCCF problem remains NP-complete.

In this thesis, we are primarily interested in developing GCCF solutions that are deployable in the real world. Hence, our main objective is to develop an algorithm that can solve problems using real-world data (rather than focusing on purely synthetic environments) in scenarios involving hundreds or thousands of agents, such as collective energy purchasing [14, 112] and ridesharing [17, 18]. To achieve this objective, one of the fundamental contributions of this thesis is to exploit the structure of such scenarios in order to formulate the corresponding characteristic functions as closed-form expressions. As a consequence, we can compute each coalitional value only when needed, without the necessity of storing the entire characteristic function in memory. This allows us to avoid the exponential memory requirements inherent in the standard representation of such function.

In our first domain of interest, i.e., the collective energy purchasing scenario, agents form coalitions to buy energy together at cheaper prices. Specifically, each agent is characterised by an energy consumption profile that represents its energy consumption throughout a day. The characteristic function of a coalition of agents is the total cost that the group would incur if they bought energy as a collective in two different markets: the spot market, a short term market (e.g., half hourly, hourly) intended for small amounts of energy; and the forward market, a long term one in which larger amounts of energy (spanning weeks and months) can be bought at cheaper prices.

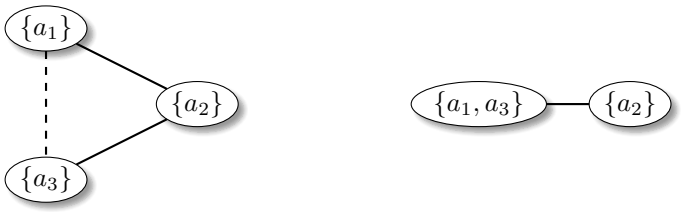
Such a scenario involves a thoughtful commitment by the agents belonging to a group, due to the purchasing of a significant amount of goods (i.e., energy in this particular case). Henceforth, it is reasonable to assume that agents may desire to participate to such contract along with trusted people, i.e., people engaged in a social connection. On these premises, it is natural to formalise the collective energy purchasing as a GCCF, in which purchasing groups are formed in order to establish a friend-of-friend relationship among their members.

1.3 Search-based GCCF

We propose a novel search-based approach to solve GCCF that employs the concept of *edge contraction*. Intuitively, the contraction of an edge of the graph represents the merge of the corresponding coalitions, as Figure 1.1 shows.
Edge contraction is the fundamental operation that allows us to represent the entire space of possible solutions (i.e., all the feasible coalition structures) of the GCCF problem. More precisely, we propose a technique to model such solution space as a rooted tree, in which each feasible coalition structure is represented exactly once. Moreover, we theoretically prove the completeness and the absence of any redundancy in our model.

Such search tree can then be explored with any known traversal technique [24]. We adopt a Depth-First Search (DFS) approach, since it is characterised by polynomial memory requirements. Crucially, since we consider closed-form characteristic functions (hence avoiding the exponentiality inherent in the classic formulation of CF), the memory requirements of our approach are polynomially bounded. This property allows us to tackle large-scale problems with thousands of agents, in contrast with previous GCCF approaches. Notice that, ideally, we could compute the optimal solution of any GCCF problem by traversing the entire tree and return the feasible coalition structure that maximises the sum of the corresponding coalitions. Nonetheless, even for sparse graphs the number of feasible coalition structures can be very large, making a complete traversal of the search space not affordable.

Against this background, we propose a technique that helps prune significant parts of the search space when the characteristic function belongs to a general class of functions (denoted as $m+a$) which can be seen as the sum of a superadditive and a subadditive part [84]. Such method is then employed within CFSS, our branch and bound algorithm to compute the optimal solution for any GCCF problem



(a) Before contraction. (b) After contraction.

Fig. 1.1: Example of an edge contraction.

based on an $m+a$ function. CFSS achieves a significant performance improvement with respect to the plain DFS algorithm discussed above, by generating only a minimal portion of the solution space (i.e., less than 0.32% in our experiments). We show that the family of $m+a$ functions is expressive enough to model realistic CF scenarios, including collective energy purchasing. Moreover, we discuss two other applications for $m+a$ functions, i.e., *edge sum with coordination cost* [27, 36] and *coalition size with distance cost*.

CFSS also serves as an anytime approximate algorithm, by stopping the search after a predefined time budget. This enables the computation of approximate solution for large-scale GCCF instances, for which the computation of the optimal solution is not feasible. Furthermore, CFSS is the first approach that can provide quality guarantees on such solutions, expressed by means of measure that quantifies the maximum amount by which the approximate solution can be worse with respect to the optimal one [4].

Our empirical evaluation considers both realistic and synthetic network topologies, i.e., Twitter [67], scale-free networks [1] and community networks [65]. Results show that CFSS outperforms DyCE [115], the state of the art algorithm, when applied to the above mentioned realistic functions, i.e., the edge sum with coordination cost, the collective energy purchasing and the coalition size with distance cost functions. Specifically, CFSS is at least 3 orders of magnitude faster than DyCE in the first scenario, while solving bigger instances for the remaining two. Finally, our algorithm provides approximate solutions with good quality guarantees (i.e., whose values are at least 88% of the optimal) for systems of unprecedented scale (i.e., more than 2700 agents).

1.4 Social ridesharing

In the above discussed scenarios, we have addressed CF problems by considering a type of constraints induced by a graph connecting the agents. On the other hand, in many realistic applications the formation of coalitions may also be influenced by constraints of different nature. For instance, if coalitions are mapped to physical objects with limited capacity, it is natural to enforce a constraint on the *cardinality* of such coalitions [102]. A straightforward real-world example is *ridesharing*, in which agents model users that need to commute across a geographical space (usually within a city), and coalitions represent cars that are shared among multiple agents with the objective of reducing travel costs, by sharing rides. In this case, the cardinality of coalitions is limited by the number of seats in each car, which is usually quite low (e.g., 5 seats [118]). In particular, we focus on a ridesharing scenario that involves a set of agents, connected through a social network, which necessitate to commute within a urban environment. In this context, agents arrange one-time rides at short notice (Figure 1.2), travelling together with friends, in contrast with complete strangers. This assumption is motivated by a clear tendency among ridesharing companies, which favour the formation of groups of users that are connected in such network. In fact, Uber and Lyft incentivise users to share rides with their friends, showing that social relationships play a central role in the ridesharing scenario, which is consequently referred as *Social Ridesharing* (SR).

The SR scenario can be naturally modelled as a GCCF problem, where the set of feasible coalitions is restricted by a graph (i.e., the social network) and by some additional feasibility constraints, e.g., the number of members of each coalition cannot exceed the number of seats of the corresponding cars.

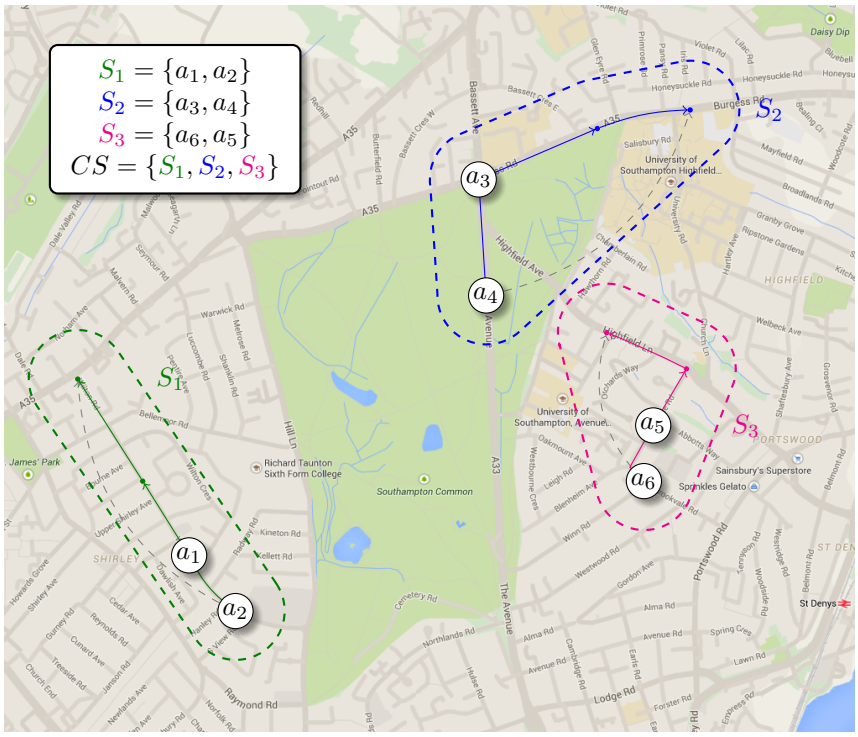


Fig. 1.2: Social Ridesharing with 6 agents and 3 coalitions (best viewed in colour).

Within such scenario, we first address the optimisation problem of minimising the total cost of all the cars formed by the system. As a consequence, we define the value of each coalition as the travel cost of the associated car. Specifically, we present the first model that encodes the above discussed scenario as a GCCF problem, and we formally define the value of each coalition on the basis of the spatial preferences of the agents. Subsequently, we generalise our model incorporating the temporal preferences of the agents, so to allow them to express constraints on the departure and the arriving time. Our approach allows us to derive efficient methods for the computation of the path and the departure time of the driver, which are optimal within the considered model.[1] Finally, we show how to solve the GCCF problem associated to the SR scenario by means of a modified version of CFSS, i.e., SR-CFSS, which differs from CFSS as it includes the cardinality constraints deriving from the SR model. Moreover, SR-CFSS employs a different bounding technique with respect to the original version, since the SR characteristic

[1] In general, both these problems are not tractable [62, 72].

function is not an $m + a$ function. We empirically evaluate SR-CFSS on realistic datasets for both spatial and social data, i.e., GeoLife by Microsoft Research [121] and Twitter [67]. Results show that our approach can produce significant cost reductions (up to -36.22%) and it features a good scalability, computing approximate solutions for very large systems (i.e., up to 2000 agents) and good quality guarantees (i.e., whose values are at least 71% of the optimal) within minutes.

1.5 Payments for SR

As mentioned above, payment computation is a crucial problem for CF with self-interested agents (such as the SR scenario we consider here). Hence, we tackle the problem of splitting the travel costs corresponding to each car among its passengers. Payoffs (corresponding to cash payments for sharing trip costs) to the commuters need to be computed given their distinct needs (e.g., shorter/longer trips), roles (e.g., drivers/riders, less/more socially connected) and opportunity costs (e.g., taking a bus, their car, or a taxi).

As previously introduced, one key aspect of payment distribution in CF is the game-theoretic concept of stability, which measures how agents are keen to maintain the provided payments instead of deviating to a configuration deemed to be more rewarding from their individual point of view. Here, we induce stable payments in the context of the SR problem, employing *the kernel* [29] stability concept. Kernel-stable payoffs are perceived as fair, since they ensure that agents do not feel compelled to claim part of their partners payoff. Kernel stability has been widely studied in cooperative game theory, and various approaches have been proposed to compute kernel-stable payments [63, 101]. However, state of the art approaches are not devised for GCCF, leading to inefficiency (i.e., they do not avoid considering unfeasible solutions) and redundancy (i.e., they consider coalitions more than once). This drawback severely limit the scalability of the entire algorithm. In contrast, a better way to tackle this problem is to exploit the structure of the graph in order to consider *only* the coalitions that are indeed feasible, so to avoid any unnecessary computation.

We achieve this by means of the PRF (Paying for Rides with Friends) algorithm [17], our method to compute a kernel-stable allocation, given a coalition structure that is a solution to the SR problem. In particular, we address the shortcomings of the state of the art algorithm in real-world scenarios, by designing an efficient parallel approach that scales up to thousands of agents. Specifically, we benchmark PRF adopting the same realistic environment used for testing SR-CFSS, showing that our method computes payments for 2000 agents in less than an hour and it is 84 times faster than the state of the art in the best case. Our approach is a practical solution technique for large-scale systems thanks to a speed-up of 10.6 on a 12-core machine with respect to the serial approach. Finally, we develop new insights into the relationship between payments incurred by a user by virtue of its position in its social network and its role (rider or driver). In general, our experimental results suggest that the kernel can be a valid stability concept in the context of SR. In fact, it induces a reasonable behaviour in the formation of groups, which can be directly correlated with some simple properties of the agents in the system (i.e., network centrality and being a driver or a rider).

1.6 Constraint optimisation for GCCF

The techniques discussed so far perform particularly well under the assumption that the value of each coalition can be expressed by means of a closed-form function (i.e., a general expression that, for each coalition, provides its value on the sole basis of its members), and it is possible to derive a method to compute an upper bound for such function, in order to apply the branch and bound approaches discussed above. However, in some GCCF scenarios it may be difficult (or not possible at all) to meet these premises, hence the application of the CFSS algorithm may be not convenient in certain settings. As an example, consider a scenario in which the value of each coalition measures the reward associated to a previously completed task, e.g., the box-office income generated by a group of actors starring in a particular movie. In this context, it may be impossible to characterise each coalitional value by means of a closed-form expression.

Against this background, in the remainder of the thesis we investigate an alternative solution method for GCCF. In the optimisation literature, Dynamic Programming (DP) [28] historically represents the counterpart approach with respect to search, especially in the context of GCCF [90, 115]. Moreover, DP-based algorithms represent the state of the art for solving CSG [89] and GCCF [115] in scenarios that consider a general characteristic function. Such facts warrant the study of an approach for GCCF based on DP, with the objective of developing a solution method that overcomes the drawbacks of previously discussed algorithms.

Now, our objective is the development of a DP solution framework for constraint optimisation (and, specifically, for GCCF) with a particular focus on the runtime performance. In recent years, Graphics Processing Units (GPUs) have been successfully used to speed-up the computation in different applications, achieving performance improvements of several orders of magnitude [41] in fields including computer vision [8], human-computer interaction [11], and artificial intelligence [108]. In particular, DP has been successfully parallelised on GPUs [20, 44, 54, 85, 107], motivating the study of a GPU-accelerated DP-based solution method for GCCF. We achieve this objective by observing that GCCF can be seen as optimisation problems subject to several feasibility constraints. In fact, GCCF aims at maximising the sum of the coalitional values while enforcing the constraint that groups must be feasible and disjoint.

Within the constrained optimisation literature, tasks of this nature are usually defined as Constraint Optimisation Problems (COPs) [31], a general class of problems that can be used to formalise several optimisation scenarios [30]. A COP is defined upon a Constraint Network (CN) [31], a theoretical model that encodes a knowledge-base theory as a graph, in which the nodes represent the variables of the optimisation problem and the edges encode the functions or relations over subsets of variables (e.g., clauses for propositional satisfiability, constraints, or conditional probability matrices for belief networks). Constraint functions are usually represented as tables, in which each row corresponds to a particular assignment of the variables in the scope of the function.

Several solution techniques for COPs have been proposed in the literature [31, 76, 77]. Motivated by the above discussion, in this thesis we adopt Bucket Elimination (BE) [31], a general algorithmic framework that represents one of the most

important approaches based on DP to solve COPs.[2] BE operates on functions in tabular form by means of a message passing technique that is realised with two fundamental operations, i.e., *composition* and *marginalisation*, which are the most computationally intensive tasks of the entire algorithm. Such operations are also the key ingredients in several close variants of the BE framework, including Belief Propagation (BP) on Junction Trees [69], and Distributed COP techniques such as ActionGDL [113] and DPOP [87].

Thus, we propose CUBE (CUda Bucket Elimination), a highly parallel implementation for the composition and marginalisation operations associated to BE. In the design of CUBE, we aim at developing a high-performance GPU framework that allows us to deal with the computational effort inherent in the message passing phase of several BE-based algorithms. To this end, our main objective is to devise a solution that fulfils three key requirements. First, since BE is a general algorithm that can be applied to several problems, our framework should be likewise general to allow a wide adoption among different domains. Second, our approach should be able to achieve a high computational throughput, by means of optimised memory accesses to avoid bandwidth bottlenecks, a careful load-balancing to fully exploit the available computational power, and the adoption of well-known parallel primitives [96, 100] to reduce the CPU workload to the minimum. Third, our solution should tackle large-scale real-world problems, and, hence, it should not be limited by the amount of GPU global memory.

CUBE achieves the objectives set above by means of a novel preprocessing algorithm that reorders the rows and the columns of the tables, enabling highly-optimised memory management. Specifically, we avoid unnecessary, expensive memory accesses by means of a technique that allows threads to efficiently locate their input data, and by taking advantage of the fastest memory in the GPU hierarchy [41]. Moreover, CUBE is not limited by the amount of GPU memory, as it can process large tables by splitting them into manageable chunks that meet the memory capabilities of the GPU.

CUBE is then employed to solve COP-GCCF, the first approach that models GCCF as a COP. Specifically, we propose a COP formalisation that results in a linear number of constraints with respect to the number of agents. We achieve this objective by means of a novel method that builds a hierarchy of agents and then exploits such structure in order to express the features of the GCCF problem while maintaining a manageable complexity. We formally characterise COP-GCCF and we prove that it correctly formalises the GCCF problem. Crucially, COP-GCCF does not require any assumption on the characteristic function, hence allowing us to solve completely general GCCF instances.

Furthermore, our model is devised to exploit the capability of CUBE to process incomplete tables, allowing us to avoid the explicit representation of unfeasible configurations, hence achieving an improved memory footprint. By doing so, we achieve the benefits of GPU parallelisation in the solution of the general GCCF

[2] In the context of this discussion, it is important to note that the complexity of BE is exponential with respect to the induced width of the CN [31], a parameter closely related to the number of constraints and to the structure of the COP (e.g., presence of loops in the CN). For this reason, the formalisation of a particular problem should result in a COP that yields an induced width of manageable complexity.

problem. Results show that our approach outperforms state of the art algorithms on sparse graphs, both in terms of runtime and memory. In particular, COP-GCCF allows to compute solutions at least one order of magnitude faster than counterpart approaches using Twitter [67] as a realistic graph topology.

1.7 Organisation of the thesis

This thesis is divided in 4 parts, where the first part discusses background knowledge and the three remaining parts present the contributions of the thesis.

1. In the first part, Chapter 2 defines the main concepts and problems later discussed in the thesis, and Chapter 3 positions our work with respect to the existing literature in the areas of CF and constraint optimisation.
2. The second part focuses on Graph-Constrained Coalition Formation. In particular, Chapter 4 discusses CFSS, our branch and bound algorithm to solve the GCCF problem, while Chapter 5 shows some real-world applications for GCCF.
3. In the third part, we discuss Cardinality-Constrained Coalition Formation, and specifically, the SR scenario. In particular, Chapter 6 details our GCCF model for SR and how we solve the corresponding GCCF problem with SR-CFSS. Chapter 7 discusses the payment computation aspect of such scenario.
4. Finally, the fourth part address the GCCF problem in the context of constraint optimisation. Chapter 8 discusses CUBE, a highly parallel implementation for the join sum and maximisation operations associated to BE. Chapter 9 defines COP-GCCF, our COP model for the GCCF problem.

Finally, Chapter 10 draws conclusions and proposes future research directions.

1.8 Publications

Most parts of this thesis have been published in different venues. In the context of GCCF, a CF approach for collective energy purchasing has been introduced in (1). The contents of Chapters 4 and 5 have been published in (2), whereas the comprehensive approach has been published in (8). In the context of SR, Chapter 6 has been published in (4), while Chapter 7 has been published in (5). The comprehensive approach has been submitted to a journal. In the context of constraint optimisation, Chapter 8 has been entirely published in (3), (6), and (7). Chapter 9 has been submitted to a conference.

1. Filippo Bistaffa, Alessandro Farinelli, Meritxell Vinyals, and Alex Rogers. “Coalitional energy purchasing in the smart grid”. In *IEEE International Energy Conference & Exhibition*, ENERGYCON, pages 848–853, 2012.
2. Filippo Bistaffa, Alessandro Farinelli, Jesús Cerquides, Juan A. Rodríguez-Aguilar, and Sarvapali D. Ramchurn. “Anytime coalition structure generation on synergy graphs”. In *International Conference on Autonomous Agents and Multiagent Systems*, AAMAS, pages 13–20, 2014.

3. Filippo Bistaffa, Alessandro Farinelli, and Nicola Bombieri. “Optimising memory management for belief propagation in junction trees using GPGPUs”. In *IEEE International Conference on Parallel and Distributed Systems*, ICPADS, pages 526–533, 2014.
4. Filippo Bistaffa, Alessandro Farinelli, and Sarvapali D. Ramchurn. “Sharing rides with friends: a coalition formation algorithm for ridesharing.” In *AAAI Conference on Artificial Intelligence*, AAAI, pages 608–614, 2015.
5. Filippo Bistaffa, Georgios Chalkiadakis, Alessandro Farinelli, and Sarvapali D. Ramchurn. “Recommending fair payments for large-scale social ridesharing”. In *ACM Conference on Recommender Systems*, RecSys, pages 139–146, 2015.
6. Filippo Bistaffa, Nicola Bombieri, and Alessandro Farinelli. “An Efficient Approach for Accelerating Bucket Elimination on GPUs”. In *IEEE Transactions on Cybernetics*, PP.99, pages 1–13, 2016.
7. Filippo Bistaffa, Nicola Bombieri, and Alessandro Farinelli. “CUBE: A CUDA Approach for Bucket Elimination on GPUs”. In *European Conference on Artificial Intelligence*, ECAI, pages 125–132, 2016.
8. Filippo Bistaffa, Alessandro Farinelli, Jesús Cerquides, Juan A. Rodríguez-Aguilar, and Sarvapali D. Ramchurn. “Algorithms for Graph-Constrained Coalition Formation in the Real World”. In *ACM Transactions on Intelligent Systems and Technology*, 2016.

Part I

Background & Related Work

2

Background

In this section we formally define the main concepts and problems later discussed in this thesis. In particular, in Section 2.1 we discuss coalition formation. Then, in Section 2.2 we define constraint optimisation problems, and we discuss the solution framework we consider, i.e., bucket elimination. Finally, in Section 2.5 we introduce GPUs, the hardware architecture used to parallelise such framework.

2.1 Coalition formation

In many applications involving multiple entities with common or individual needs, the formation of groups is fundamental to achieve such objectives [21]. For instance, in ridesharing scenarios, commuters share rides in order to minimise travel costs and reduce pollutant emissions. In patrolling applications adopting mobile sensors, it is more effective to combine capabilities of unmanned vehicles of different classes, rather than using only one type. These examples also highlight that agents may be characterised by fully cooperative behaviours (i.e., will forego their own benefit for the common good), or, in contrast, act as self-interested entities (e.g., in collective energy purchasing each home is required to pay for the amount of consumed energy). This may impede the formation of groups as agents would need to agree on their common actions.

Coalition Formation (CF) is one of the key approaches to create collaborations in multi-agent systems [21]. Building upon the seminal work of Shehory and Kraus [102], Sandholm et al. [95] identify the key tasks involved in the CF process:

- Coalitional value calculation: defining a *characteristic function* which, given a coalition as an argument, provides its coalitional value.
- Coalition Structure Generation (CSG): partitioning the set of agents into disjoint coalitions, with the objective of maximising the sum of their values.
- Payment computation: finding the transfer or payment to each agent to ensure it is fairly rewarded for its contribution to its coalition.

In what follows, we formally discuss the CSG and payment computation problems, which are the main focus of this thesis as they usually represent the most computationally intensive tasks in the CF process.

2.1.1 Coalition structure generation

The Coalition Structure Generation (CSG) problem [95, 102] takes as input a finite set of n agents $A = \{a_1, \ldots, a_n\}$ and a characteristic function $v : 2^A \to \mathbb{R}$, that maps each coalition $S \in 2^A$ to its value, describing how much collective payoff a set of players can gain by forming a coalition. It is important to notice that, in the standard definition, the input of the CSG problem (i.e., the characteristic function) is already of exponential size, since it has to represent the values of all the $2^{|A|}$ possible coalitions. For this reason, state of the art approaches that solve standard CSG are characterised by exponential memory requirements, which limit their application to instances with tens of agents (see Section 3.2.1).

A coalition structure CS is a partition of the set of agents into disjoint coalitions. The set of all coalition structures is $\Pi(A)$. The value of a coalition structure CS is assessed as the sum of the values of its composing coalitions, i.e.,

$$V(CS) = \sum_{S \in CS} v(S). \tag{2.1}$$

CSG aims at identifying CS^*, the most valuable coalition structure, i.e.,

$$CS^* = \arg\max_{CS \in \Pi(A)} V(CS).$$

The computational complexity of the CSG problem is due to the size of its search space. In fact, a set of n agents can be partitioned[1] in $\Omega((\frac{n}{\ln(n)})^n)$ ways, i.e., the n^{th} Bell number [10], since, in standard CSG, every possible subset of agents is potentially a valid coalition.

In what follows, we discuss graph-constrained coalition formation, in which the set of coalition is restricted by a graph.

2.1.2 Graph-constrained coalition formation

In many realistic scenarios, constraints influence the process of coalition formation [91, 114, 115]. For instance, a physical communication network with a particular topology may enforce or prevent the formation of certain coalitions, or cardinality constraints may be introduced to either prohibit or allow coalitions of certain sizes [102]. Following the work of Myerson [79] and Demange [35], and more recent work by Voice et al. [115], in this thesis we focus on a specific type of constraints that encodes synergies or relationships among the agents and that can be expressed by a graph, where nodes represent agents and edges encode the relationships between the agents. In this setting, edges enable connected agents to form a coalition and a coalition is considered feasible only if its members represent the vertices of a connected subgraph. Such constraints are present in several real-world scenarios, such as social or trust constraints (e.g., energy consumers who prefer to group with their acquaintances in forming energy cooperatives [13, 14], or commuters sharing rides with their friends [18]), and physical constraints (e.g., emergency responders may join specific teams in disaster scenarios where only certain routes are available).

[1] The CSG problem is equivalent to the *complete set partitioning* problem [119].

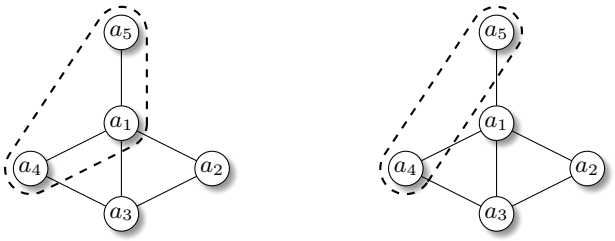
In order to model these settings, Myerson [79] first proposed a definition of *feasible* coalition by considering an undirected graph $G = (A, E)$, where $E \subseteq A \times A$ is a set of edges between agents, representing the relationships between them:

Definition 2.1 (feasible coalition). *A coalition S is* feasible *if all of its members are connected in the subgraph of G induced by S, i.e., for each pair of players $a_i, a_j \in S$ there is a path in G that connects a_i and a_j without going out of S.*

Thus, given a graph G the set of feasible coalitions is

$$\mathcal{FC}(G) = \{S \subseteq A \mid \text{The subgraph induced by } S \text{ on } G \text{ is connected}\}.$$

For instance, in the example graph in Figure 2.1, coalition $\{a_1, a_4, a_5\}$ is feasible, as the corresponding subgraph is connected. In contrast, $\{a_4, a_5\}$ is not feasible, since a_4 and a_5 are not connected by an edge.



(a) Feasible coalition (b) Unfeasible coalition

Fig. 2.1: Feasible vs. unfeasible coalitions, nodes are agents and edges are relationships (e.g., social connections, communication links).

Consequently, a Graph-Constrained Coalition Formation (GCCF) problem is a CSG problem together with a graph G, where a coalition S is considered feasible if $S \in \mathcal{FC}(G)$. In GCCF problems a coalition structure CS is considered feasible if each of its coalitions is feasible, i.e.,

$$\mathcal{CS}(G) = \{CS \in \Pi(A) \mid CS \subseteq \mathcal{FC}(G)\}.$$

Hence, the goal for a GCCF problem is to identify CS^*, which is the most valuable feasible coalition structure, i.e.,

$$CS^* = \underset{CS \in \mathcal{CS}(G)}{\arg\max}\ V(CS). \tag{2.2}$$

Notice that CSG represents a particular case of GCCF, i.e., CSG is a GCCF problem in which G is a complete graph. As a consequence, GCCF is characterised by the same worst-case complexity of the unconstrained case, i.e., $\Omega((\frac{n}{\ln(n)})^n)$. Now, even if such exponential complexity is not representative of the problems we are interested to solve (i.e., problems in which G is sparse and, hence, $\mathcal{CS}(G)$ contains a lower number of feasible coalition structures), GCCF remains a hard problem, i.e., it is NP-Complete [114]. As such, GCCF solution algorithm cannot solve large-scale instances (see Section 3.2.3).

In the next section, we discuss the second fundamental aspect of CF, i.e., payment computation.

2.1.3 Payment computation

The *payment computation* problem involves the computation of a *payoff vector* x,[2] which specifies a payoff $x[i]$ for each agent $a_i \in A$ as a compensation of their contributions. This problem has been thoroughly studied in the cooperative-game theory literature, thus we suggest the reader to refer to the book by Chalkiadakis et al. [22] for a more extensive discussion of all the technical aspect on this subject.

In the context of this discussion, we are particularly interested in computing payoff vectors that are *efficient* and *individually rational*.

Definition 2.2 (efficiency). *Given a coalition structure CS and a payoff vector x, x is* efficient *if, for each coalition $S \in CS$, the entire value of S is split among the members of S, i.e., $v(S) = \sum_{a_i \in S} x[i]$ for all $S \in CS$.*

Definition 2.3 (individual rationality). *Given a coalition structure CS and a payoff vector x, x is* individually rational *if each agent a_i receives a payoff $x[i]$ that is* at least *the value of its singleton, i.e., $x[i] \geq v(\{a_i\})$.*

Efficiency and individual rationality are fundamental in any real-world application, as they formalise natural properties that are often assumed in practice. In fact, efficiency expresses the principle that no portion of the value of S should be wasted. On the other hand, individual rationality states that a rational agent does not join a group if such action does not produce a reward.

Furthermore, computing payments that are *stable* is of utmost importance in systems with selfish rational agents, i.e., agents who are only interested in the maximisation of their payoffs [22]. As such, payoffs have to be distributed among agents to ensure that members are rewarded according to their bargaining power [22]. In particular, stability ensures that agents will not deviate from the provided solution to a different one that is better from their individual point of view. In cooperative game-theory, stability has been defined with several concepts, including the stable set, the nucleous, the kernel, and the core [22]. The *core* is one of the most widely studied stability concepts, since it ensures a particularly strong and useful property that grants that no subset of A can improve upon the considered payoff vector.

Definition 2.4 (the core). *A payoff vector x is* core-stable *if it satisfies* efficiency *and* coalitional rationality*, i.e., $x(S) \geq v(S)$ for all coalitions $S \subseteq A$, where $x(S)$ refers to the sum of the payments of the members of S, i.e., $x(S) = \sum_{a_i \in S} x[i]$.*

It is easy to prove that core-stability implies efficiency, in a sense that any core-stable payoff vector maximises the social welfare, i.e., the sum of all the coalitions. As said above, the core is a very strong stability concept, but its computation has an exponential complexity with respect to the number of agents. As such, it is not suitable for large-scale systems. Furthermore, it is not guaranteed that core-stable solutions always exist [22]. For the above reasons, in this thesis we focus on the kernel.

[2] A payoff vector can also be referred as *payoff allocation*.

2.1.4 The kernel

The *kernel* is a stability concept introduced by Davis and Maschler [29]. A key feature of the kernel is that it is always non-empty. Moreover, there are polynomial-time approaches that can compute an approximation of the kernel when the size of coalitions is limited [17, 63]. The kernel provides stability within a given coalition structure, and under a given payoff allocation, by defining how payoffs should be distributed so that agents cannot *outweigh* (cf. below) their current partners, i.e., the other members of their coalition. It is easy to see that the kernel provides a notion of stability which is *weaker* with respect to the core, as it is limited within single coalitions.

In order to define the kernel, we first define the *excess* of a coalition S with respect to a given payoff vector x as $e(S,x) = v(S) - x(S)$. In the kernel, a positive excess is interpreted as a measure of threat: in the current payoff distribution, if some agents deviate by forming coalition with positive excess, they are able to increase their payoff by redistributing the coalitional excess among themselves. On the basis of the excess, we define the notion of *surplus*.

Definition 2.5 (surplus). *Given a coalition structure CS and a coalition $S \in CS$, we consider $a_i, a_j \in S$. Then, the* surplus s_{ij} *of a_i over a_j with respect to a given payoff configuration x, is defined by*

$$s_{ij} = \max_{\substack{S' \in 2^A \\ a_i \in S', a_j \notin S'}} e(S', x), \tag{2.3}$$

In other words, s_{ij} is the maximum of the excesses of all coalitions S' that include a_i and exclude a_j, with S' not in the given coalition structure CS (since under CS agents a_i and a_j belong to the same coalition S). We say that agent a_i outweighs agent a_j if $s_{ij} > s_{ji}$. When this is the case, a_i can claim part of a_j's payoff by threatening to walk away (or to expel a_j) from their coalition. When any two agents in a coalition cannot outweigh one another, the payoff vector lies *in the kernel* – i.e., it is stable. Importantly, the set of kernel-stable payoff vectors is always non-empty [22].

Stearns [106] provides a *payoff transfer scheme* which converges to a vector in the kernel by means of payoff transfers from agents with less bargaining power to their more powerful partners, until the latter cannot claim more payoff from the former. Unfortunately, this may require an infinite number of steps to terminate. To alleviate this issue, Klusch and Shehory [63] introduced the ϵ-kernel in order to represent an allocation whose payoffs do not differ from an element in the kernel by more than ϵ. Formally, the authors propose a *truncated* (i.e., requiring $O(n)$[3] iterations) payoff transfer scheme, which computes a payoff vector x such as

$$\frac{\max_{(a_i,a_j)\in A^2}(s_{ij} - s_{ji})}{V(CS)} \leq \epsilon.$$

[3] This result expresses the complexity with respect to the size of the input (i.e., n) only. A more detailed discussion on how ϵ also affect the number of iterations is provided by Shehory and Kraus [101].

Notice that both GCCF and the computation of kernel-stable payments are optimisation problems subject to several feasibility constraints. This is particularly clear by looking at Equations 2.2 and 2.3. On the one hand, Equations 2.2 highlights how GCCF aims at maximising the sum of the coalitional values while enforcing the constraint that coalitions must be feasible and disjoint. On the other hand, Equation 2.3 seeks the coalition that results in the maximum excess among all the ones that contain a_i but exclude a_j. Such problems are usually referred as constraint optimisation problems, which we discuss in the following section.

2.2 Constraint optimisation problems

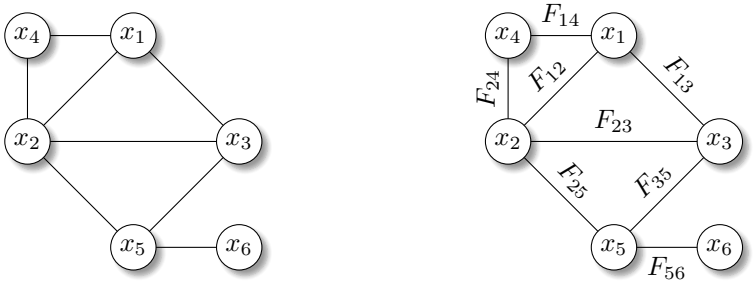
A Constraint Optimisation Problem (COP) is defined upon a Constraint Network (CN) [31], a theoretical model that encodes a knowledge-base theory as several functions or relations over subsets of discrete variables (e.g., clauses for propositional satisfiability, constraints, or conditional probability matrices for belief networks).

Definition 2.6 (constraint network). *A* constraint network *consists of a set* $X = \{x_1, \ldots, x_n\}$ *of* n *discrete variables such that* $x_1 \in D_1, \ldots, x_n \in D_n$*, where* D_i *represents the domain of the variable* x_i*, together with a set of* m *constraints* $\{C_1, \ldots, C_m\}$*.*

Definition 2.7 (constraint). *A* constraint C_i *is a relation defined on a set* $X_i = \{x_{i_1}, \ldots, x_{i_h}\}$ *of* h *discrete variables, called the* scope *of the constraint, such that* $X_i \subseteq X$*. Such a relation denotes the variables simultaneous legal assignments. Non-legal assignments are denoted as* unfeasible*.*

A particular CN corresponds to a *Constraint Satisfaction Problem* (CSP), which can be generalised obtaining a COP.

Definition 2.8 (constraint satisfaction problem). *Given a CN, the corresponding* constraint satisfaction problem *requires to find a variable assignment* $\bar{a}^* = (a_1^*, \ldots, a_n^*)$ *satisfying all the constraints in the CN.*



(a) CN. (b) COP.

Fig. 2.2: Example CN and the corresponding COP.

Definition 2.9 (constraint optimisation problem). *A* constraint optimisation problem *is a CN augmented with a set of functions. Let $F_1, \ldots, F_l$ be l real-valued functional components defined over the scopes $Q_1, \ldots, Q_l$, $Q_i \subseteq X$, let $\bar{a} = (a_1, \ldots, a_n)$ be an assignment of the variables, where $a_i \in D_i$. The* global cost function *F is defined by $F(\bar{a}) = \sum_{i=1}^{l} F_i(\bar{a})$, where $F_i(\bar{a})$ means F_i applied to the assignments in $\bar{a}$ restricted to the scope of F_i. Solving the COP requires to find $\bar{a}^* = (a_1^*, \ldots, a_n^*)$, satisfying all the constraints, such that $F(\bar{a}^*) = max_{\bar{a}} F(\bar{a})$ (or $F(\bar{a}^*) = min_{\bar{a}} F(\bar{a})$, in case of a minimisation problem).*

Figure 2.2a shows an example CN with $X=\{x_1, x_2, x_3, x_4, x_5, x_6\}$, while the corresponding COP is shown in Figure 2.2b. Cost functions are usually encoded as *tables*, in which each row represents a variable assignment and its resulting value.

Definition 2.10 (complete (resp. incomplete) tables). *A cost function F_i is* complete *if unfeasible assignments are explicitly represented with $-\infty$ ($+\infty$ in case of a minimisation problem) values. In contrast, if unfeasible assignments are not represented at all, F_i is said to be* incomplete.

COPs are a general class of problems, which can be used to model several optimisation scenarios [30]. COPs can be solved using diverse techniques ranging from search-based approaches to Dynamic Programming (DP) (see Section 3.4). In the context of this discussion, it is important to note that the complexity of such algorithms is dominated by a parameter known as *induced width*, which encodes the complexity of the COP and which is closely related to the number of constraints and the structure of the CN (e.g., presence of cycles). In order to formally define the induced width, we first introduce some background concepts [31].

Definition 2.11 (ordered graph). *Given a graph G corresponding to a CN, an* ordered graph *is a pair (G, o) where o is an ordering of the nodes of G.*

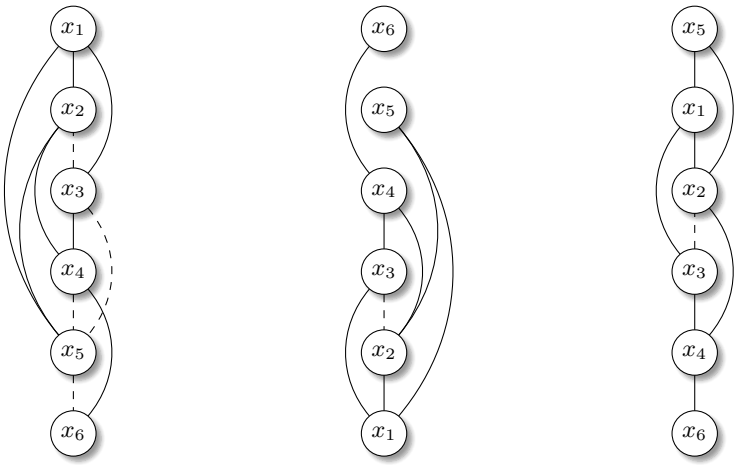


Fig. 2.3: Induced graphs of the CN in Figure 2.2a with different orderings.

Definition 2.12 (width of a graph). *Given an ordered graph* (G, o)*, the* width of a node *is its number of parents. The* width of the ordering o *is the maximum width over all nodes. The* width of the graph G *is the minimum width over all the possible orderings of the graph.*

Definition 2.13 (induced graph). *Given an ordered graph* (G, o)*, the corresponding* induced graph (G^*, o) *is an ordered graph obtained as follows. The nodes of* G *are processed from last to first (top to bottom) along* o*. When a node is processed, all of its parents are connected.*

Definition 2.14 (induced width). *The* induced width of an ordered graph (G, o)*, denoted as* $w^*(o)$*, is the width of the induced ordered graph* (G^*, o)*. The* induced width of a graph G *is the minimal induced width over all its orderings.*

Notice that computing the induced width of a graph with a brute-force approach requires to explore all the possible $n!$ orderings, where n is the number of variables in the CN. In general, computing the optimal ordering o^* that results in the minimal induced width is NP-Complete [31]. For this reason, a greedy procedure (Algorithm 1) [31] is usually adopted to compute a variable ordering of acceptable quality. Algorithm 1 can be parametrised with different $metric(\cdot)$ functions that evaluate each node on the basis of different properties. The most commonly used are the *min-degree* heuristic (in which $metric(x_i)$ is the number of neighbours of x_i) and the *min-fill* heuristic (in which $metric(x_i)$ is the number of edges that need to be added to the graph due to the elimination of x_i).

In the next section, we discuss the algorithmic framework we consider to solve COPs, i.e., bucket elimination.

2.3 Bucket elimination

Bucket Elimination (BE) is a general algorithmic framework that adopts DP to incorporate many reasoning techniques. In particular, here we focus on the version of BE that solves COPs. Specifically Algorithm 2 describes the BE approach following Dechter [31]. Such an algorithm operates on the basis of a *variable ordering* o, which is used to partition the set of functions into n sets $B_1, \dots, B_n$ called *buckets*, each associated to one variable of the COP. In particular, each function F_i is placed in the bucket associated to the last bucket that is associated with a variable in Q_i, i.e., the scope of F_i. Figure 2.4 shows the buckets corresponding to the example COP in Figure 2.2b, adopting the ordering $o = \langle x_1, x_3, x_2, x_5, x_4, x_6 \rangle$.

Algorithm 1 GreedyOrdering (CN, $metric(\cdot)$)

1: **for all** $k \leftarrow n$ down to 1 **do**
2: $x^* = \arg\min_{x_i \in X} metric(x_i)$
3: $o[k] \leftarrow x^*$
4: Introduce edges in CN between all neighbours of x^*
5: Remove x^* from CN
6: **return** o

Algorithm 2 BucketEliminationCOP $(\mathrm{CN}, F_1, \ldots, F_l, o)$

Partition $\{C_1, \ldots, C_m\}$ and $\{F_1, \ldots, F_l\}$ into n buckets according to o
for all $p \leftarrow n$ down to 1 **do**
  for all $C_k, \ldots, C_g$ over scopes $X_k, \ldots, X_g$, and
  for all $F_h, \ldots, F_j$ over scopes $Q_h, \ldots, Q_j$, **in** bucket p **do**
    if $x_p = a_p$ **then**
      $x_p \leftarrow a_p$ in each F_i and C_i
      Put each F_i and C_i in appropriate bucket
    else
      $U_p \leftarrow \bigcup_i X_i - \{x_p\}$
      $V_p \leftarrow \bigcup_i Q_i - \{x_p\}$
      $W_p \leftarrow U_p \cup V_p$
      $C_p \leftarrow \pi_{U_p} \left(\bowtie_{i=1}^{g} C_i \right)$
      for all tuples t over W_p **do**
        $H_p(t) \leftarrow \Downarrow_{a_p \colon (t, a_p) \text{ satisfies } \{C_1, \ldots, C_g\}} \bigoplus_{i=1}^{j} F_i(t, a_p)$
      Place H_p in the latest lower bucket mentioning a variable in W_p, and C_p in the latest lower bucket with a variable in U_p
Assign maximising values for the functions in each bucket
return $\bar{a}^*$

Then, buckets are processed from last to first (top to bottom), by means of two fundamental operations, i.e., *composition* (denoted as $\oplus$) and *marginalisation* (denoted as $\Downarrow$), which will be discussed hereafter. Specifically, all the cost functions in B_p, i.e., the current bucket, are *composed* with the $\oplus$ operation, and the result is the input of a $\Downarrow$ operation. Such operation removes x_p (i.e., the variable associated to B_p) from the table, and produces a new function H_p that does not involve x_p, which is then placed in the last bucket that is associated to a variable appearing in the scope of the new function.

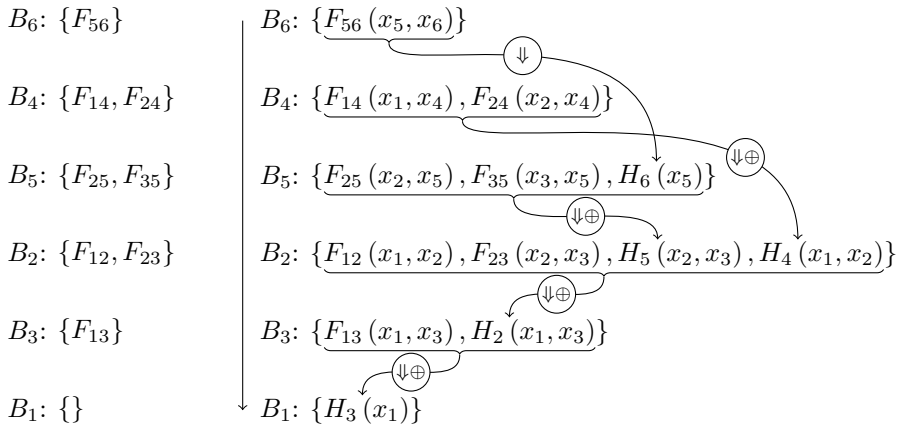


Fig. 2.4: Initial buckets.

Fig. 2.5: BE execution.

Figure 2.5 shows the execution of BE on the previous example. In particular, if a bucket, say B_4, contains more than one F_i, such functions are first composed with $\oplus$ and then the corresponding variable (i.e., x_4) is marginalised out. In Figure 2.5, we represent these two subsequent operations by means of the compact notation $\Downarrow\oplus$. In the case of B_4, the result of $\Downarrow\oplus$ is a function $h_4(x_1, x_2)$ without x_4, which is placed in B_2. By operating in such a way, we can guarantee that the resulting function in the first bucket (i.e., $H_3(x_1)$ in Figure 2.5) contains only the first variable in o, i.e., x_1, since all the remaining ones have been marginalised out during the previous steps. Hence, we compute the optimal assignment for x_1 as the one that maximises $H_3(x_1)$, and propagate such assignment back to the second bucket. Then, we proceed in the same way as before, computing the optimal assignment for the corresponding variable, and propagating the result until all buckets have been processed. Such process terminates when the optimal assignment for all variables has been computed.

The computational complexity of the BE algorithm is directly determined by the ordering o, as the following proposition states.

Proposition 2.15. *The complexity of BE is time and space exponential in $w^*(o)$, the induced width of the problem given the variable ordering o, i.e., $O\left(m \cdot k^{w^*(o)}\right)$, where k bounds the domain size and m is the number of constraints.*

Proof. The proof is provided by Dechter [31].

The variable elimination scheme realised by BE can be used to solve different problems, depending on the actual implementation of the $\oplus$ and $\Downarrow$ operators. In what follows, we discuss how such operators are realised in order to solve COPs. Then, in Section 2.4 we discuss how a different implementation of such operators leads to the solution of another interesting problem, i.e., belief propagation.

2.3.1 Composition

We now discuss how the $\oplus$ composition operator is implemented in Algorithm 2 by the *join sum*, an operation closely related to the *inner join* of relational algebra. For the remainder of this thesis, tables are represented according to Definition 2.16. Moreover, if L is a tuple of elements, we refer to its k^{th} element with $L[k]$. Finally, for all tuples we adopt the *zero-based* convention, i.e., tuples start at index 0.

Definition 2.16 (table). *A* table *$T_i = \langle Q_i, d_i, R_i, \phi_i \rangle$ is defined by:*

- *$Q_i \subseteq X$, a tuple of variables called the* scope *of T_i;*
- *d_i, a tuple of natural numbers such that $d_i[k] = D_j$ is the size of the domain $Q_i[k] = x_j$, where $k \in \{1, \ldots, |Q_i|\}$;*
- *R_i, a tuple of* rows*: in particular, each row $R_i[k]$ is a tuple of natural numbers, defining a particular assignment of the variables in Q_i, where $k \in \{1, \ldots, |R_i|\}$;*
- *ϕ_i, a tuple representing the actual values of the function, one for each row $R_i[k]$: in particular, $\phi_i[k]$ is the value associated to the variable assignment represented by $R_i[k]$, where $k \in \{1, \ldots, |R_i|\}$.*

As an example, consider T_1 in Figure 2.6. In this case, $Q_1 = \langle x_1, x_3, x_5, x_8 \rangle$, $d_1 = \langle 2, 2, 2, 2 \rangle$ (all variables are binary), $\phi_1 = \langle \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6 \rangle$, and finally

$$\begin{aligned} R_1 = \langle &\langle 0, 1, 0, 1 \rangle, \\ &\langle 1, 0, 0, 1 \rangle, \\ &\langle 1, 1, 0, 1 \rangle, \\ &\langle 0, 1, 0, 0 \rangle, \\ &\langle 0, 0, 0, 1 \rangle, \\ &\langle 1, 1, 1, 1 \rangle \rangle. \end{aligned}$$

The goal of the join sum is to identify matching tuples and compute the sum of the respective ϕ values. As an example, consider the join sum between T_1 and T_2 (shown in Figure 2.6), with $Q_2 = \langle x_1, x_2, x_3, x_4, x_6, x_{10} \rangle$, and $Q_1 \cap Q_2 = \langle x_1, x_3 \rangle$, representing the shared variables between T_1 and T_2.[4] Notice that some variable assignments are missing, i.e., T_1 and T_2 are *incomplete* (see Definition 2.10).

T_1

x_1	x_3	x_5	x_8	ϕ_1
0	1	0	1	α_1
1	0	0	1	α_2
1	1	0	1	α_3
0	1	0	0	α_4
0	0	0	1	α_5
1	1	1	1	α_6

$\oplus$

T_2

x_1	x_2	x_3	x_4	x_6	x_{10}	ϕ_2
1	0	0	1	1	0	β_1
1	0	1	1	1	0	β_2
0	1	0	0	1	1	β_3
1	1	0	1	0	1	β_4
0	0	0	1	1	0	β_5
1	1	1	1	1	1	β_6

Fig. 2.6: Original tables T_1 and T_2 (best viewed in colour).

A row in T_1 *matches* a row in T_2 if all the shared variables have the same values in both the rows (matching rows are highlighted with the same colour in Figure 2.6). It is important to note that this is a *many-to-many* relationship, because multiple rows in the first table can match multiple rows in the second table. For instance

x_1	x_3	x_5	x_8	ϕ_1
1	0	0	1	α_2

matches

x_1	x_2	x_3	x_4	x_6	x_{10}	ϕ_2
1	0	0	1	1	0	β_1
1	1	0	1	0	1	β_4

because they all have $x_1 = 1$ and $x_3 = 0$. Thus, the result table will have a row for each couple of matching rows in the input tables. In the above example, the corresponding rows in the result table $T_\oplus = T_1 \oplus T_2$ will be

x_1	x_3	x_5	x_8	x_2	x_4	x_6	x_{10}	$\phi_\oplus$
1	0	0	1	0	1	1	0	$\alpha_2 + \beta_1$
1	0	0	1	1	1	0	1	$\alpha_2 + \beta_4$

[4] If $Q_1 \cap Q_2 = \emptyset$, the join sum trivially outputs an empty table.

In particular, these resulting rows are obtained combining the second row of T_1 and, respectively, the first and the fourth rows of T_2. They both have the same values for the shared variables ($x_1 = 1$ and $x_3 = 0$). The values of the *non-shared variables* (i.e., x_5 and x_8 for T_1, and x_2, x_4, x_6 and x_{10} for T_2) are copied from the corresponding matching rows. Hence, in the above example, $x_5 = 0$ and $x_8 = 1$ for both the resulting rows (since there is only one matching row in T_1), and $x_2 = 0$, $x_4 = 1$, $x_6 = 1$ and $x_{10} = 0$ for the first resulting row (since it results from the match with the first matching row in T_2), and so on. Thus, the variable set of the resulting table is the union of the variable sets of the input tables. Finally, the values of the resulting rows are obtained summing the values of the corresponding matching rows, i.e., $\alpha_2 + \beta_1$ and $\alpha_2 + \beta_4$. Is it easy to see that if n rows in T_1 match m rows in T_2, they will result in $n \cdot m$ rows in the resulting table (Figure 2.7).

x_1	x_3	x_5	x_8	x_2	x_4	x_6	x_{10}	$\phi_\oplus$
0	0	0	1	1	0	1	1	$\alpha_5 + \beta_3$
0	0	0	1	0	1	1	0	$\alpha_5 + \beta_5$
1	0	0	1	0	1	1	0	$\alpha_2 + \beta_1$
1	0	0	1	1	1	0	1	$\alpha_2 + \beta_4$
1	1	0	1	0	1	1	0	$\alpha_3 + \beta_2$
1	1	0	1	1	1	1	1	$\alpha_3 + \beta_6$
1	1	1	1	0	1	1	0	$\alpha_6 + \beta_2$
1	1	1	1	1	1	1	1	$\alpha_6 + \beta_6$

Fig. 2.7: Join sum result $T_\oplus$ (best viewed in colour).

2.3.2 Marginalisation

The second fundamental operation, which implements the $\Downarrow$ marginalisation operator in Algorithm 2, is the maximisation. Suppose that, as a result of the inner join sum operation at line 13 of Algorithm 2, we obtain the table T shown in Figure 2.8. Now, suppose that $x_p = x_8$. Then, Algorithm 2 requires to maximise such table *marginalising out* x_8, i.e., removing the column corresponding to x_8 and selecting the maximum value among the ones that refer to the repeated entries. In fact, as a result of this removal, some rows may now be equal considering the remaining columns (e.g., $R[1]$ and $R[2]$ both contain $\langle 0, 0, 0\rangle$ in the first three columns, as well as $R[3]$ and $R[4]$, which contain $\langle 1, 0, 0\rangle$). Since one cannot have duplicate rows, the maximisation operations computes a single row that, as a value, stores the maximum of the original values.[5] The final result is in Figure 2.9.

Composition and marginalisation operators are also employed by modern versions of BE, e.g., Bucket-Tree Elimination (BTE) proposed by Kask et al. [59]. In the remainder of this thesis, we focus on BE since it was the first version of these message-passing techniques to tackle constrained optimisation, and its performance is generally comparable with BTE, which, in turn, is optimised for some specific problems, i.e., singleton-optimality problems.

[5] If we marginalise out the variable x_i, we maximise over up to d_i rows.

x_1	x_3	x_5	x_8	ϕ
0	0	0	0	α_1
0	0	0	1	α_2
1	0	0	0	α_3
1	0	0	1	α_4
1	1	0	1	α_5
1	1	1	1	α_6

Fig. 2.8: Initial table T.

x_1	x_3	x_5	$\phi_{\Downarrow}$
0	0	0	$\max(\alpha_1, \alpha_2)$
1	0	0	$\max(\alpha_3, \alpha_4)$
1	1	0	α_5
1	1	1	α_6

Fig. 2.9: Result of the maximisation.

We now discuss BP on JTs, which is a close variation of BE [31] that is also based on composition and marginalisation operators (i.e., scattering and reduction).

2.4 Belief propagation on junction trees

Belief propagation on junction trees [69, 86] is an algorithm used to propagate inference on a Bayesian network.

Definition 2.17 (Bayesian network). *A* Bayesian Network *(BN) [56] encodes a joint distribution over a set of n random variables X, structured as a Directed Acyclic Graph (DAG) whose vertices are the random variables and the directed edges represent the conditional probabilities among the variables, encoded as Conditional Probability Tables (CPTs).*

As an example, consider the BN in Figure 2.10a with $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$. BNs allow to compute various tasks on variables with stochastic nature. For example, a BN could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the BN can be used to compute the probabilities of the presence of various diseases. Moreover, BNs can be used to find out updated knowledge of the state of a subset of variables when other variables (i.e., the *evidence* variables) are observed. BNs are adopted to solve several problems [31], including i) *Belief Propagation* (BP), i.e., computing the posterior probability of each proposition given some evidence, ii) *Most Probable Explanation* (MPE), i.e., given some observed variables, finding a maximum probability assignment of the rest of the variables, iii) *Maximum A Posteriori hypothesis* (MAP), i.e., given some evidence, finding an assignment of a subset of hypothesis variables that maximises their probability, and finally, iv) given also a utility function, finding an assignment to a subset of variables that result in the *Maximum Expected Utility* (MEU).

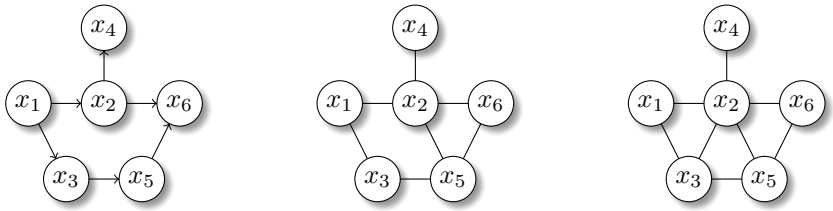
Here we focus on the BP problem, but our discussion can be easily extended to the above mentioned tasks, as they adopt similar operations. Several approaches have been proposed for the propagation of beliefs (or posteriors) [93]. In this thesis, we consider the BP on Junction Trees (BP on JTs) algorithm, first proposed by Lauritzen and Spiegelhalter [69]. Such an approach runs over a *junction tree*, a particular tree derived from the original BN that fulfils Definition 2.19.

Definition 2.18 (cluster graph). *A* cluster graph *T is built from a given BN as follows. Each maximal clique in the BN is a node in T. Such nodes are called* clusters*. Clusters that share at least one variable are connected by an edge in T.*

Definition 2.19 (junction tree). *A* Junction Tree *(JT) is a particular cluster graph that fulfils the following properties:*

1. ***Singly connected**: there is exactly one path between each pair of clusters.*
2. ***Running intersection**: for each pair of clusters N_i and N_j that contain an element k, each cluster on the unique path between N_i and N_j also contains k.*

A JT is generated from a BN by means of moralisation and triangulation [69]. The moralised counterpart of a DAG is constructed by adding edges between all pairs of nodes that have a common child, and then making all edges in the graph undirected, as shown in Figure 2.10b. Then, we triangulate the graph by connecting any two non-successive nodes in any given cycle (Figure 2.10c). Notice that the obtained graph is guaranteed to be chordal.[6]



(a) Original BN. (b) Moralised graph. (c) Triangulated graph.

Fig. 2.10: Effects of moralisation and triangulation on the BN.

The next step in the creation of the JT involves finding the set of *elimination cliques* in the moralised and triangulated graph, which will later represent the nodes of the JT. This is achieved by means of Algorithm 3. Notice that, similarly to BE, such an algorithm requires the definition of a variable ordering o, which affects the creation of the JT, i.e., different orderings can result in different JTs. Thus, it is possible to obtain different JTs from the same BN. Figure 2.11 shows the execution of Algorithm 3 on the above example adopting the ordering $o = \langle x_6, x_4, x_5, x_3, x_2, x_1 \rangle$, which produces the maximal cliques $\{x_2, x_5, x_6\}$, $\{x_2, x_4\}$, $\{x_2, x_3, x_4\}$, and $\{x_1, x_2, x_3\}$.

Algorithm 3 EliminationCliques (G, o)

```
1: max ← ∅ {Initialise the set of maximal cliques}
2: while |o| > 1 do
3:    x ← o.pop() {Get the first variable in the queue corresponding to o}
4:    clq ← maximal clique in G that contains x
5:    if ∄clq' ∈ max : clq ⊂ clq' then
6:       max ← max ∪ clq
7:    Remove x from G
8: return max
```

[6] A chordal graph is one in which all cycles of four or more vertices have a chord, i.e., an edge that is not part of the cycle but connects two vertices of the cycle.

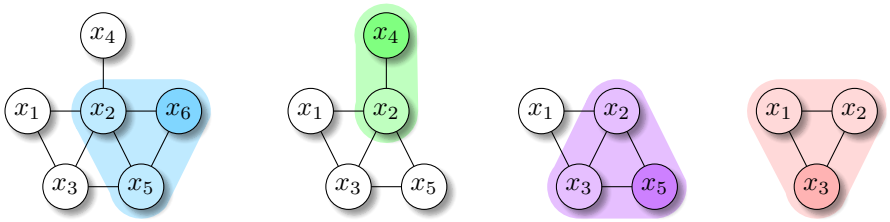
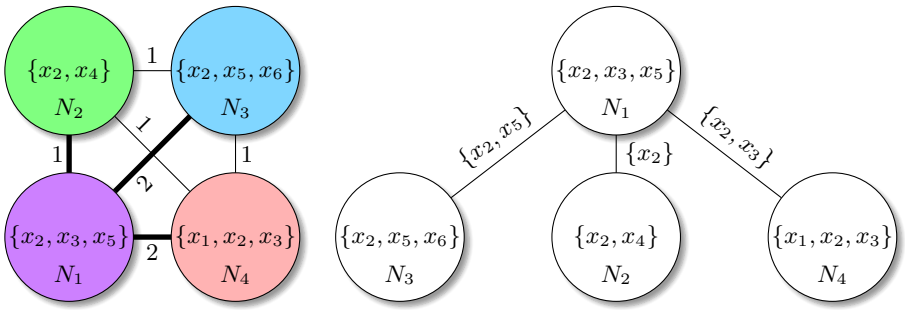


Fig. 2.11: Example execution of Algorithm 3 (best viewed in colour).

Such cliques are the nodes of the cluster graph (Figure 2.12a) resulting from the original BN. In such graph, we label each edge with a weight corresponding to the number of shared variables between the cliques incident on such edge. Finally, the maximum-weight spanning tree of the cluster graph (highlighted with bold edges in Figure 2.12a) is a JT of the original BN,[7] as shown in Figure 2.12b. We also report the shared variables for each pair of cliques on the corresponding edges.



(a) Cluster graph. (b) JT.

Fig. 2.12: Cluster graph and the corresponding JT (best viewed in colour).

Every vertex N_i of such JT contains a set $Q_i \subseteq X$ of random variables that forms a maximal clique in the moralised and triangulated BN, each associated to a *potential table* represented by $T_i = \langle Q_i, d_i, R_i, \phi_i \rangle$ (according to Definition 2.16). Specifically, each potential table is obtained by multiplying the appropriate CPTs, i.e., the CPTs relative to the variables in the scope of the potential table, as discussed in detail by Lauritzen and Spiegelhalter [69]. As an example, T_4, associated to $Q_4 = \langle x_1, x_2, x_3 \rangle$, is the result of $p(x_1)\, p(x_2 \mid x_1)\, p(x_3 \mid x_1)$, where $p(x_i \mid x_j)$ is the CPT of x_i given x_j. Notice that, since CPTs contain a row for each possible assignment of the variables in their scope, potential tables also inherit this property. In other words, the tables used in BP are *complete* (see Definition 2.10), in contrast with the COP case (cf. Section 2.2). Therefore,

$$|R_i| = |\phi_i| = \prod_{k=1}^{|Q_i|} d_i[k].$$

[7] Such JT is guaranteed to satisfy both properties in Definition 2.19 [25].

Each T_i encodes the joint probability[8] among the random variables in its scope Q_i. Figures 2.13–2.16 show an example of such tables, in which all the variables are assumed to be binary.

x_2	x_3	x_5	ϕ_1
0	0	0	0.5
1	0	0	0.2
0	1	0	0.1
1	1	0	0.0
0	0	1	0.1
1	0	1	0.0
0	1	1	0.0
1	1	1	0.1

Fig. 2.13: T_1.

x_2	x_4	ϕ_2
0	0	0.4
1	0	0.3
0	1	0.1
1	1	0.2

Fig. 2.14: T_2.

x_2	x_5	x_6	ϕ_3
0	0	0	0.1
1	0	0	0.0
0	1	0	0.0
1	1	0	0.1
0	0	1	0.1
1	0	1	0.0
0	1	1	0.5
1	1	1	0.2

Fig. 2.15: T_3.

x_1	x_2	x_3	ϕ_4
0	0	0	0.1
1	0	0	0.5
0	1	0	0.0
1	1	0	0.1
0	0	1	0.0
1	0	1	0.0
0	1	1	0.1
1	1	1	0.2

Fig. 2.16: T_4.

Assuming that T_i and T_j are potential tables corresponding to adjacent vertices in the JT, we associate a separator table $Sep_{ij} = \langle Q_{ij}, d_{ij}, R_{ij}, \phi_{ij} \rangle$ to the edge (N_i, N_j), whose scope Q_{ij} is represented by the shared variables between the two tables, i.e., $Q_{ij} = Q_i \cap Q_j$. The values for ϕ_{ij} can be initialised to any non-zero constant value (as discussed by Lauritzen and Spiegelhalter [69]), hence we consider 1, as shown in Figures 2.17–2.19.

x_2	ϕ_{12}
0	1.0
1	1.0

Fig. 2.17: Sep_{12}.

x_2	x_5	ϕ_{13}
0	0	1.0
1	0	1.0
0	1	1.0
1	1	1.0

Fig. 2.18: Sep_{13}.

x_2	x_3	ϕ_{14}
0	0	1.0
1	0	1.0
0	1	1.0
1	1	1.0

Fig. 2.19: Sep_{14}.

BP on JTs is invoked whenever we receive new evidence for a particular set of variables $Y \subset X$, so to update the potential tables associated to the BN in order to reflect this new information. To this end, a two-phase procedure is employed: first, in the *evidence collection* phase, messages are collected from each vertex N_i, starting from the leaves all the way up to an arbitrarily designated root vertex (Algorithm 4). Then, during *evidence distribution*, messages are distributed from the root to the leaves (Algorithm 5). In both phases, each recursive call comprises a MessagePass procedure, which realises the propagation of the evidence between the potential tables T_i and T_j associated to N_i and N_j, involving two steps:

[8] A rigorous application of the principles of statistics would require to normalise the values of each ϕ_i so to verify $\sum_{j=0}^{|\phi_i|-1} \phi_i[j] = 1$. Nonetheless, DeGroot [34] shows that this is not strictly necessary, and the results of BP is correct even if the normalisation is executed only at the end of the BP process.

1. **Reduction**: the potential table Sep_{ij} is updated to Sep^*_{ij}. In particular, each row of Sep^*_{ij} is obtained summing the corresponding rows of T_i, i.e., the ones with a matching variable assignment. Reduction implements the $\Downarrow$ marginalisation operator of BE, which is achieved with a summation in this case.
2. **Scattering**: T_j is updated with the new values of Sep^*_{ij}, i.e., every row of T_j is multiplied for the ratio between the corresponding rows in Sep^*_{ij} and Sep_{ij}. Following Zheng and Mengshoel [120], we assume that $\frac{0}{0} = 0$. Scattering employs the product operation to implement the composition, and it corresponds to the $\oplus$ operator of BE.

Algorithm 4 COLLECT (JT, N_i)

1: **for all** N_j child of N_i **do**
2: MESSAGEPASS $(N_i, \text{COLLECT}(\text{JT}, N_j))$
3: **return** N_i

Algorithm 5 DISTRIBUTE (JT, N_0)

1: **for all** N_j child of N_0 **do**
2: MESSAGEPASS (N_0, N_j)
3: DISTRIBUTE (JT, N_j)
4: **return** N_i

We now show how to execute the BP algorithm in order to update tables T_1–T_4 according to the evidence $x_2 = 0$. In the following example, we assume that the JT in Figure 2.12b is rooted in N_1. As a consequence, N_2, N_3 and N_4 are the leaves of the JT, and hence, they initiate the evidence collection phase. Specifically, they absorb the evidence by zeroing all the entries corresponding to the rows that disagree with $x_2 = 0$ (highlighted in grey in Figures 2.20–2.22).

x_2	x_4	ϕ_2
0	0	0.4
1	0	0.0
0	1	0.1
1	1	0.0

Fig. 2.20: Updated T_2.

x_2	x_5	x_6	ϕ_3
0	0	0	0.1
1	0	0	0.0
0	1	0	0.0
1	1	0	0.0
0	0	1	0.1
1	0	1	0.0
0	1	1	0.5
1	1	1	0.0

Fig. 2.21: Updated T_3.

x_1	x_2	x_3	ϕ_4
0	0	0	0.1
1	0	0	0.5
0	1	0	0.0
1	1	0	0.0
0	0	1	0.0
1	0	1	0.0
0	1	1	0.0
1	1	1	0.0

Fig. 2.22: Updated T_4.

Then, such nodes construct the messages to be sent upward to N_1 by updating the respective separator tables, which are obtained from the updated potential tables by marginalising out the variables not shared with T_1, i.e., by executing the *reduction* step. Specifically, Sep^*_{12}, Sep^*_{13}, and Sep^*_{14} are respectively obtained by marginalising out x_4, x_3, and x_1. This operation is almost equivalent to the one discussed in Section 2.3.2, with the only difference that here we marginalise by summation, instead of maximising. For the sake of brevity, we directly report the final ${}^{Sep^*_{ij}}/_{Sep_{ij}}$ tables in Figures 2.23–2.25, which result from the ratio between the updated and the original separator tables. Such ratio is necessary for the subsequent step of BP, i.e., scattering.

x_2	${}^{\phi^*_{12}}/_{\phi_{12}}$
0	0.5
1	0.0

Fig. 2.23: ${}^{Sep^*_{12}}/_{Sep_{12}}$.

x_2	x_5	${}^{\phi^*_{13}}/_{\phi_{13}}$
0	0	0.2
0	1	0.5
1	0	0.0
1	1	0.0

Fig. 2.24: ${}^{Sep^*_{13}}/_{Sep_{13}}$.

x_2	x_3	${}^{\phi^*_{14}}/_{\phi_{14}}$
0	0	0.6
1	0	0.0
0	1	0.0
1	1	0.0

Fig. 2.25: ${}^{Sep^*_{14}}/_{Sep_{14}}$.

Finally, N_1 completes the evidence collection phase by updating its potential table using the *scattering* operation, i.e., each row of T_1 is multiplied for the rows of the updated separator tables that have matching variable assignments. The updated T_1 is shown in Figure 2.26, while its normalised version is shown in Figure 2.27.

x_2	x_3	x_5	ϕ_1
0	0	0	$0.5 \cdot 0.5 \cdot 0.2 \cdot 0.6 = 0.030$
1	0	0	0.0
0	1	0	$0.1 \cdot 0.5 \cdot 0.2 \cdot 0.0 = 0.000$
1	1	0	0.0
0	0	1	$0.1 \cdot 0.5 \cdot 0.5 \cdot 0.6 = 0.015$
1	0	1	0.0
0	1	1	$0.0 \cdot 0.5 \cdot 0.5 \cdot 0.0 = 0.000$
1	1	1	0.0

Fig. 2.26: Updated T_1.

x_2	x_3	x_5	ϕ_1
0	0	0	$0.\overline{66}$
1	0	0	0.00
0	1	0	0.00
1	1	0	0.00
0	0	1	$0.\overline{33}$
1	0	1	0.00
0	1	1	0.00
1	1	1	0.00

Fig. 2.27: Updated and normalised T_1.

The *evidence distribution* phase is then carried out from the root to the leaves by employing the same message passing technique. Figures 2.28–2.30 show the final and normalised tables T_2, T_3 and T_4 after such phase. In particular, notice how the BP process has erased the probabilities of all the assignments disagreeing with the evidence $x_2 = 0$. Furthermore, the distribution process has also updated the potential tables of the leaves according to the new probabilities in the potential of the root.

x_2	x_4	ϕ_2
0	0	0.8
1	0	0.0
0	1	0.2
1	1	0.0

Fig. 2.28: Final T_2.

x_2	x_5	x_6	ϕ_3
0	0	0	$0.\overline{2}\overline{2}$
1	0	0	0.00
0	1	0	0.00
1	1	0	0.00
0	0	1	$0.\overline{2}\overline{2}$
1	0	1	0.00
0	1	1	$0.\overline{5}\overline{5}$
1	1	1	0.00

Fig. 2.29: Final T_3.

x_1	x_2	x_3	ϕ_4
0	0	0	$0.1\overline{6}$
1	0	0	$0.8\overline{3}$
0	1	0	0.00
1	1	0	0.00
0	0	1	0.00
1	0	1	0.00
0	1	1	0.00
1	1	1	0.00

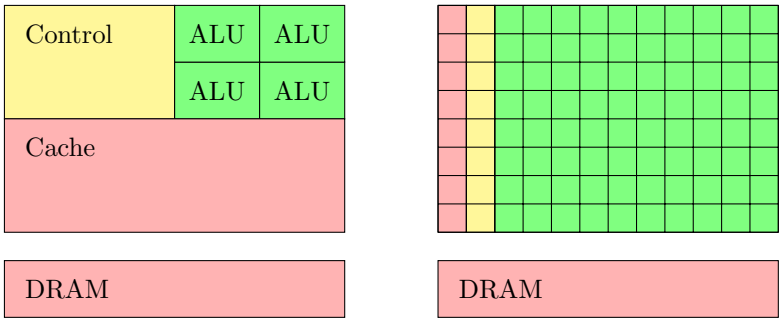
Fig. 2.30: Final T_4.

In both BE and BP on JTs, it is easy to see that the composition ($\oplus$) and the marginalisation ($\Downarrow$) steps are the most computationally intensive tasks. As a consequence, if such operations are not executed efficiently, the use of these algorithms is not practical for realistic applications. Thus, it is crucial to implement both $\oplus$ and $\Downarrow$ in a very efficient way. In Chapter 8 we achieve this objective by means of the CUBE algorithm. Our approach exploits the inherent parallel nature of such operators, which execute several independent computations spanning over multiple rows of the tables. This characteristic suggests a multi-threaded algorithm in which such degree of parallelism can be exploited by means of GPUs

2.5 Graphics processing units

GPUs are designed for compute-intensive, highly parallel computations. These architectures perform particularly well on problems that can be modelled as data-parallel computations where data elements correspond to parallel processing threads, as GPUs are designed on the basis of the Single Instruction Multiple Data (SIMD) model [41]. A widely used framework for GPU programming is the NVIDIA CUDA framework, which provides an Application Programming Interface (API) that allows the user to employ general purpose programming primitives (e.g., memory allocation, code execution, synchronisation) on the GPU. In particular, data processing is achieved through a particular function, called *kernel*, executed in parallel by thousands of threads on different inputs. Threads are grouped into thread *blocks*. Threads in the same block share fast forms of storage and synchronisation primitives. A fundamental difference between CPU and GPU hardware design is that the latter devotes the majority of the transistors to Arithmetic Logic Units (ALUs) subdivided among thousands of cores (i.e., $\sim$2000 for the GPUs employed in the experiments of this thesis). On the other hand, the computational capabilities of each of these cores are very limited with respect to common CPUs, as a direct consequence of the fact that, on GPU cores, cache and control hardware are substantially reduced, as shown in Figure 2.31. For this reason, GPUs usually do not support *branch prediction*,[9] and thus, branching should be generally avoided on GPUs. In fact, branching can result in a phenomenon called *divergence* [50], which causes the serialisation the threads executing in different branches.

[9] Branch prediction aims at guessing which way a branch (e.g., an *if-then-else* structure) will go, with the purpose of improving the performance in a pipelined execution model.



(a) CPU. (b) GPU.

Fig. 2.31: CPU vs. GPU hardware design (best viewed in colour).

2.5.1 Memory management

The second fundamental aspect of GPU programming is memory management. Memory plays a crucial role in the design of efficient GPU algorithms, since memory accesses are particularly expensive and have a significant impact on the performance. A comprehensive view of the CUDA memory model is shown in Figure 2.32. Modern GPUs contain very fast but small-size memories (i.e., registers, cache and *shared memory*, which can store up to few tens of kilobytes of data), intended to assist high performance computations, stacked above a slower but larger memory (i.e., *global memory*), suitable to hold large amounts of data (typically ∼4GB). Furthermore, the architecture provides two forms of read-only memory, i.e., *constant* and *texture memory*, designed to store data that is read multiple times during the computation (e.g., textures in computer vision applications). These memories can only be populated from the host, i.e., GPU threads cannot write to them.

The use of shared memory (which resides directly on GPU cores and thus, is characterised by very low latencies), is crucial for high throughput algorithms. To increase the performance, it is mandatory to exploit such a low latency memory to store information that needs to be used very often. On the other hand, accessing global memory is particularly costly (400–800 clock cycles), and should be minimised to achieve a good compute-to-memory ratio. To do that, a common practice suggests to exploit data locality, i.e., transferring small portions of frequently used data from global to shared memory and to complete all the computational tasks that use such data before accessing other data. This allows to minimise global memory accesses. The optimal way of executing such transfers is depicted in Figure 2.33. In particular, sparse and random accesses to input and output data force the hardware to serialise each memory operation, with a great impact on the performance, as shown in Figure 2.33a. In contrast, it is desirable to have consecutive threads fetching data from consecutive memory addresses (Figure 2.33b), which is denoted as memory *coalescing*. Coalesced accesses are related to the principle of locality of information and they allow the hardware to combine multiple transfers between global and shared memory into a single transaction, avoiding expensive operations and improving the computational throughput.

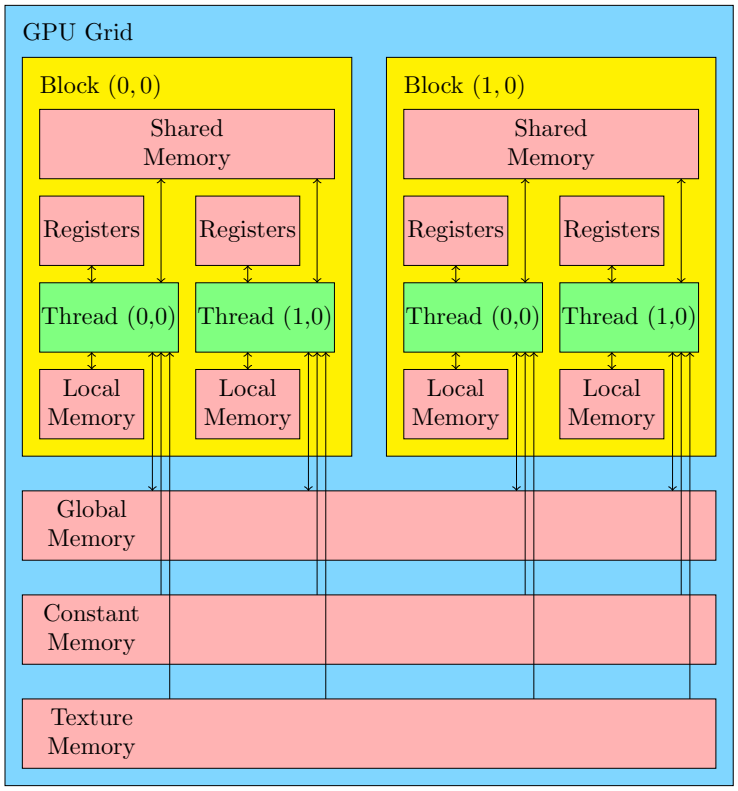
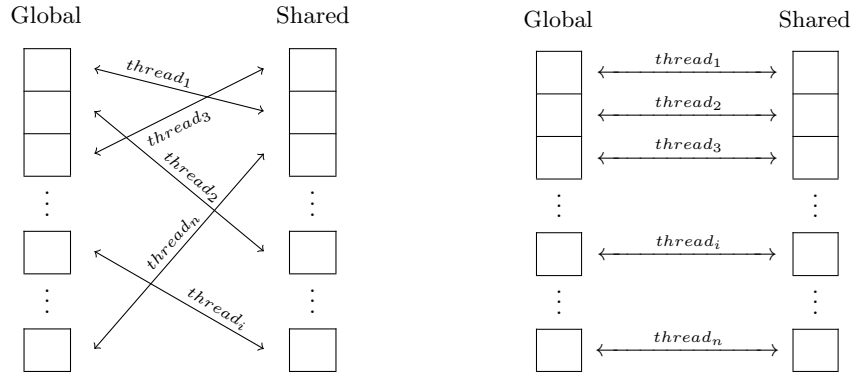


Fig. 2.32: CUDA memory model (best viewed in colour).



(a) Uncoalesced memory accesses.

(b) Coalesced memory accesses.

Fig. 2.33: Uncoalesced vs. coalesced memory accesses.

The design of an efficient GPU version of a sequential algorithm is rarely a trivial task, as all the above aspects need to be carefully taken into account. In fact, most of the times it is necessary to adopt a different approach, which is specifically devised for the underlying parallel architecture. On the other hand, GPUs can result in speed-ups of several orders of magnitude [41] if employed correctly.

2.5.2 Pipelining

The standard pattern of GPU computation requires the whole input to be transferred to the global memory before starting the kernel execution. The results are then copied back to the host memory. Such *synchronous* approach can be improved if the kernel starts on a partial set of input data, while the transfer is still running.

Figure 2.34 shows the a *pipelined* model of computation, in which a single GPU kernel execution has been split into three stages (marked by different colours). Each kernel K_i executes as soon as the corresponding input data subset has been transferred by means of $H \rightarrow D_i$. This solution applies to GPU architectures that feature only one *copy engine* (i.e., data between host and device can be transferred through a single channel only). Data segments are necessarily serialised, thus allowing overlapping between one kernel execution and one data transfer only. In our experiments, we found that, on average, this approach achieves a performance improvement of 50% with respect to synchronous data transfers. Most recent and advanced GPUs (e.g., NVIDIA Kepler) feature an additional copy engine, which enables a further degree of parallelism between data transfers and computation. On these architectures, this approach exploits the supplementary channel to overlap input and output data transfers (see Figure 2.35). Such pipelined computation achieves, in our experiments, an improvement of 75% with respect to synchronous transfers.

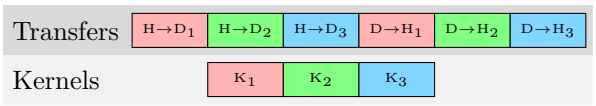


Fig. 2.34: Asynchronous data transfers (best viewed in colour).

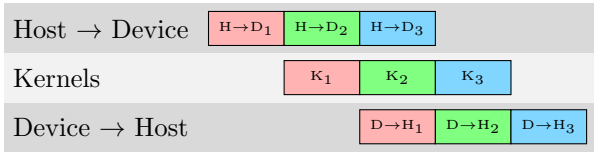


Fig. 2.35: Full pipeline (best viewed in colour).

3

Related work

In this section, we position the work discussed in this thesis with respect to the existing literature in the areas of team formation, CF and constraint optimisation.

3.1 Team formation

The problem of forming groups of agents has also been widely studied in the context of Team Formation, in which several formal definitions of such problem have been proposed. As an example, Gaston and desJardins [47] devise a heuristic to modify the graph connecting the agents based on local autonomous reasoning, without considering any concept of global optimal solution. The problem studied by Lappas et al. [68] focuses on finding a single group of agents who possess a given set of skills, so as to minimise the communication cost within such a group. Marcolino et al. [75] focus on forming a single group of agents that has the maximum strength in the set of world states. Finally, Liemhetcharat and Veloso [73] are interested in modelling the values of the characteristic function, based on observations of the agents.

In this thesis, we address the specific group formation problem in which groups must form a partition (into disjoint coalitions) of a given set of agents, with the objective of maximising the sum of the coalitional values. Such problem is equivalent to the *complete set partitioning* problem [119], i.e., the standard definition adopted in the CF literature.

3.2 CSG algorithms

The CF literature comprises a number of works that address the CSG problem with various techniques. In particular, in Section 3.2.1 we describe optimal and approximate algorithms that solve general CSG, where all coalitions can be formed. Then, in Section 3.2.2 we discuss the approaches considering CSG scenarios restricted by the presence of constraints. Specifically, Section 3.2.3 positions our work with respect to the existing literature about graph-constrained CSG. In Section 3.2.4, we discuss the approaches based on special characteristic function representations. Finally, Section 3.2.5 describes heuristic approaches for CSG.

3.2.1 Complete approaches

A number of algorithms have been developed to solve general CSG. Sandholm et al. [95] showed that is possible to find a coalition structure whose value is within some provable bound from the optimal one through an approximation algorithm. The authors devise a graph organising all possible coalition structures in n levels (where n is the number of agents), such that level i contains all the solution formed exactly by i coalitions. Thus, the first level always corresponds to the grand coalition, while the n^{th} level contains the coalition structure formed by the singletons. Two nodes of this particular graph are connected if it is possible to obtain one configuration from the other only by splitting one coalition in two, as shown in Figure 3.1.

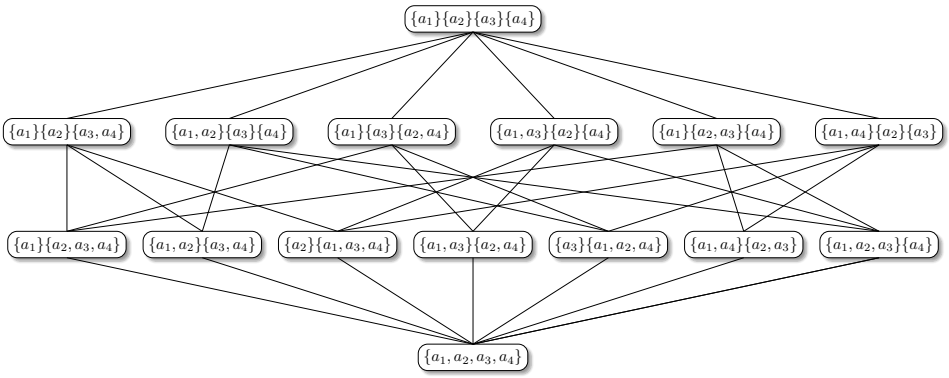


Fig. 3.1: Graph of coalition structures for $A = \{a_1, a_2, a_3, a_4\}$.

Sandholm et al. [95] notice that, while exploring the entire graph does not provide any advantage over the naïve enumeration of all coalition structures, it is possible to compute a solution whose value is within a factor of $\frac{1}{n}$ from the optimal by restrict the search to the first two levels. The proposed algorithm is *anytime*, i.e., it can return a valid solution even if interrupted before the completion. This property is particularly important in the context of CSG, in which the complete execution takes an unfeasible amount of time for large instances.

Dang and Jennings [26] later improved the scheme proposed by Sandholm et al. [95], by considering a similar layered-graph representation, but adopting a different approach to explore such graph that results in better performance. However, their solutions do not scale (as the associated computational complexity is $O(n^n)$). Moreover, as discussed by Voice et al. [115], such solutions cannot be employed to solve CSG for GCCF, since assigning artificially low values (such as $-\infty$) to infeasible coalitions would not be suitable for assessing valid bounds.

An important strand of literature [88, 89, 119] focused on solutions for CSG based on Dynamic Programming (DP). In particular, Yun Yeh [119] first proposed the idea of using DP for CSG, by associating each coalition structure to the best way of splitting it. Then, by proceeding through a bottom-up approach, the best splitting of the grand coalition, i.e., the optimal solution of the CSG problem, is eventually calculated.

Notice that, as mentioned above, a given coalition structure CS formed by i coalitions is positioned at level i in the graph in Figure 3.1. Furthermore, all its possible splittings belongs to levels $< i$, and are connected to CS by means of a path. Rahwan and Jennings [88] noticed that some of these paths are redundant (Figure 3.2), and hence, it is possible to improve the performance of the DP algorithm by pruning them. As a consequence, they proposed the Improved DP (IDP) algorithm, later extended by Rahwan et al. [89] with the IDP-IP* algorithm. IDP-IP* introduce a preliminary step based on Integer Programming (IP) that prunes the less promising parts of the space search. However, IDP-IP* is limited to tens of agents (30 at most) due to the large memory requirements, as such approaches need to hold all coalitional values in memory ($\Theta(2^n)$) during the computation.

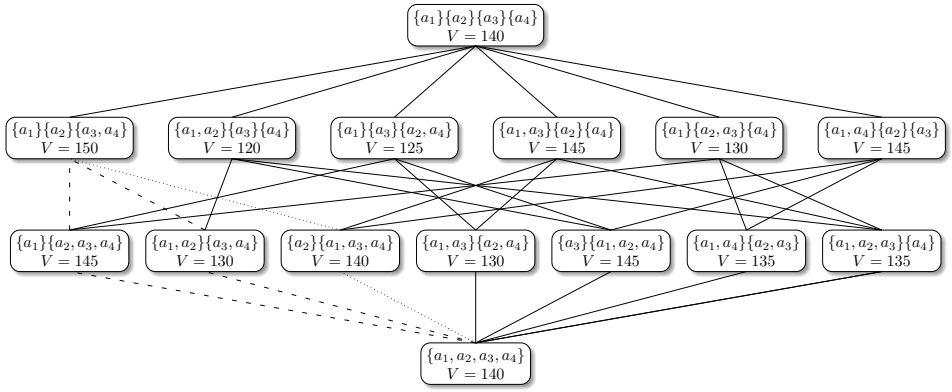


Fig. 3.2: Dashed paths are redundant with respect to the dotted one.

3.2.2 Constrained approaches

The above-described works focus on unconstrained CF, and they cannot be directly used in contexts where constraints of various types may limit the formation of some coalitions. In this respect, Shehory and Kraus [102] first introduced the idea, arising in many realistic scenarios, of restricting the maximum cardinality k of the coalitions in CSG, highlighting that, even though this constraint lowers the number of coalitions from exponential, i.e., 2^n, to polynomial, i.e., $O\left(n^k\right)$, the problem remains NP-hard. Therefore, the authors propose an approximate algorithm with quality guarantees. However, their approach is devised for scenarios in which all $O\left(n^k\right)$ coalitions are valid, and does not exploit the presence of other types of constraints, such as constraints induced by a graph (see Section 3.2.3).

Rahwan et al. [91] also considered scenarios in which constraints enforce (or prohibit) the co-existence of agents in a coalition, introducing the problem of Constrained Coalition Formation (CCF) to adequately deal with these constraints. In particular, provide a general formulation of a CCF problem, in which the set of feasible coalition structure is denoted as $\mathcal{CS} \subseteq \Pi(A)$. Notice that GCCF is a particular case of CCF problem, in which such set is represented by $\mathcal{CS}(G)$, i.e., the set of feasible coalition structures induced by a graph (see Section 2.1.2).

The authors also identify a natural, simpler subclass of CCF, namely Basic CCF (BCCF), in which the formation of coalition is subject to *positive* and *negative* constraints, providing a specialised solution algorithm. Notice that, even though the presence of these constraints allows a significant performance improvement, solving BCCF remains a hard problem.

Formally, the authors define the set of positive (resp. negative) constraints $\mathcal{P} \subseteq 2^A$ (resp. $\mathcal{N} \subseteq 2^A$) such that a coalition S satisfies a constraint $P \in \mathcal{P}$ (resp. $N \in \mathcal{N}$) if $P \subseteq S$ (resp. $N \nsubseteq S$). The solution of a BCCF problem must satisfy all the positive constraints, and cannot satisfy any of the negative constraints.

As an example, suppose that eight web-service providers $A = \{a_1, \ldots, a_8\}$ consider cooperation in order to provide cloud-computing capabilities to a major client. The client knows from prior experience that certain alliances of companies are indispensable to perform this task, and these are $\mathcal{P} = \{\{a_1, a_5, a_8\}, \{a_2, a_5, a_7\}, \{a_5, a_7, a_8\}\}$. Thus, only coalition involving any of these alliances are considered to be feasible. Furthermore, the client excludes any coalitions involving alliances of $\mathcal{N} = \{\{a_1, a_2, a_3\}, \{a_2, a_3, a_5\}\}$ due to the fact that, from prior experience, these specific combinations of providers are known to under-perform. At first sight, it appears that the GCCF and the BCCF problems may be related, since they both focus on constrained CF. Moreover, it is easy to see that such classes are not disjoint, since unconstrained CSG can be represented both as GCCF and as BCCF. Nonetheless, we show that GCCF and BCCF are *different* problems and hence, the algorithm for BCCF provided by Rahwan et al. cannot be applied to GCCF.

Proposition 3.1. *GCCF games are not a subset of BCCF games, and BCCF games are not a subset of GCCF games.*

Proof. **There are GCCF games which are not BCCF games**

Our first example is a GCCF game with three agents $A = \{a_1, a_2, a_3\}$. The set of edges of G is $E = \{(a_1, a_2), (a_2, a_3)\}$. Thus, the set of feasible coalitions is $\mathcal{FC}(G) = \{\{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}\}$, that is, the only non-feasible coalition is $\{a_1, a_3\}$. We will now show that this game cannot be encoded as a BCCF game. We show this by contradiction: let us assume that the above game can be encoded as a BCCF game. Since the single element coalitions $\{a_1\}, \{a_2\}, \{a_3\}$ are feasible, the set of positive constraints $\mathcal{P}$ should include $\{a_1\}, \{a_2\}, \{a_3\}$. Since the grand coalition $\{a_1, a_2, a_3\}$ is feasible, the set of negative constraints $\mathcal{N}$ should be empty. Since there are feasible coalitions with one, two, and three elements, the set of allowed sizes should be $\mathcal{S} = \{1, 2, 3\}$. It is easy to see that coalition $\{a_1, a_3\}$ is feasible on the candidate BCCF game, and thus that our example GCCF game is not a BCCF game.

There are BCCF games which are not GCCF games

Our second example is a BCCF game with two agents $A = \{a_1, a_2\}$. The set of positive constraints $\mathcal{P}$ is $\{\{a_1\}\}$, the set of negative constraints $\mathcal{N}$ is empty and the set of allowed sizes is just $\mathcal{S} = \{2\}$. Thus, the set of feasible coalitions is $\mathcal{FC}(G) = \{\{a_1, a_2\}\}$. Assume that we can encode this game as a GCCF game. Since in every GCCF game the singletons are feasible, we have a contradiction.

Thus, the relationship between GCCF and BCCF is represented in Figure 3.3.

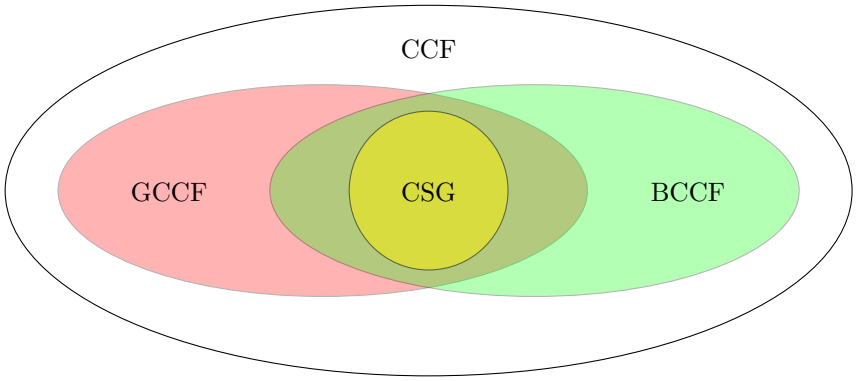


Fig. 3.3: Relationships between the CCF families (best viewed in colour).

3.2.3 Graph-constrained approaches

Voice et al. [114, 115] were the first to propose algorithms for the GCCF problem. Specifically, the approach presented in [114] focuses on scenarios fulfilling the *Independence of Disconnected Members* (IDM) property. The IDM property requires that, given two disconnected agents a_i and a_j, the presence of agent a_i does not affect the marginal contribution of agent a_j to a coalition. In this setting, Voice et al. [114] propose a solution algorithm that iterates over the connected components of all the acyclic subgraphs of the graph $G = (V, E)$. Due to the IDM property, this method computes the optimal solution within $O(|V|^2 \binom{|V|+|E|}{|V|})$ operations, and such complexity can be reduced to linear for graphs with bounded treewidth.

However, the IDM property is rather strong for real-world applications. As noticed by Shehory and Kraus [102] considering task allocation, the addition of a new agent to a coalition could result in intra-coalition coordination and communication costs, which increase with the size of the coalition. Hence, realistic functions capturing such costs (such as the ones in Section 4.3) do not satisfy the IDM property, hence this approach cannot be applied.

On the other hand, the DyCE algorithm [115] uses DP to find the optimal coalition structure by progressively splitting the current solution into its best partition. DyCE considers only the partitions that correspond to feasible coalition structures by adopting the SlyCE algorithm (also proposed by Voice et al. [115]) as a subroutine. Specifically, SlyCE solves the problem of enumerating all the $\hat{k}$-subgraphs of G, i.e., the set of connected subgraphs of G with *at most* k nodes (see Section 7.2). Voice et al. [115] also provide a parallelised version of SlyCE, i.e., D-SlyCE. DyCE is not an anytime algorithm and requires an exponential amount of memory in the number of agents (i.e., $\Theta(2^n)$). Hence, the scalability of this approach is limited to systems consisting of tens of agents (around 30). Moreover, DyCE cannot be efficiently parallelised due to its exponential memory requirements.

3.2.4 Approaches based on special characteristic function representations

The classic formulation of the CF problem assumes that each coalitional value is stored in memory, and can be accessed in constant time. Unfortunately, this approach requires an exponential amount of memory, i.e., $\Theta(2^n)$, limiting its applicability to systems with tens of agents. Thus, a number of works [23, 40, 55, 82, 109, 110, 111] have examined alternative characteristic function representations, which allow to overcome the intractability due to the size of the input and to reduce the computational complexity of the associated CF problems [49, 110]. One strand of literature [82, 111] has focused on CSG algorithms for coalitional games that can be modelled as Marginal Contribution (MC) networks [55], avoiding the exponentiality of the input representation in some particular scenarios. The basic idea behind MC networks is to represent coalitional games using sets of rules, which follows the syntax $pattern \rightarrow value$. A rule is said to apply to a coalition S if S meets the requirement of $pattern$. In the basic scheme, these patterns are conjunctions of agents, and S meets the requirement of the given pattern if S is a superset of it. The value of a group of agents is defined to be the sum over the values of all rules that apply to the group. As an example, consider the set of agents $A = \{a_1, a_2\}$, and the rules $\{a_1, a_2\} \rightarrow 5$ and $\{a_2\} \rightarrow 2$. As a consequence, $v(\{a_1\}) = 0$, $v(\{a_2\}) = 2$ and $v(\{a_1, a_2\}) = 5 + 2 = 7$.

Moreover, Conitzer and Sandholm [23] proposed a concise representation based on synergy coalition groups, which, unfortunately, does not reduce the complexity of the associated CSG problem [82]. Ueda et al. [110] showed that, by focusing on agent *types* (i.e., groups of agents whose contributions to the system are the same) rather than on agents themselves, it is possible to lower the computational complexity of several CF related problems. Finally, Bachrach and Rosenschein [5] studied coalitional games based on the notion of agent *skills*, later adopted by Tran-Thanh et al. [109] to propose a mixed-integer linear programming solution based on a vector representation that scales to a hundred agents.

While all the above-described works make significant contributions to the state of the art, the models they propose may not be able to capture the nature of realistic characteristic functions such as the collective energy purchasing one we consider here. On the one hand, this function cannot be concisely expressed as a MC network, as its MC network would require an exponential amount of memory with respect to the number of agents. On the other hand, the concepts of agent types/skills imply that it is possible to fully characterise the contribution of each agent on the basis of a small set of features, in order to achieve the conciseness of the representation. Now, some works [78, 116] have investigated the use of clustering techniques in the context of energy consumption analysis, suggesting the application of such methods to reduce the set of agents to a limited number of types (each characterised by a common energy consumption behaviour). However, results show that the consistency and the preciseness of user aggregation can vary significantly depending on various characteristics [116], leading to approximations of very poor quality. For these reasons, we do not compare against these works, since we are interested in developing techniques that can handle complex characteristic functions such as the collective energy purchasing function.

3.2.5 Heuristic approaches

Few heuristic approaches to CSG have been developed over the last few years. For example, Sen and Dutta [99] propose a solution based on genetic algorithms, Dos Santos and Bazzan [39] propose an approach based on swarm intelligence, and Farinelli et al. [42] propose an approach based on hierarchical clustering. Meta-heuristic approaches to CSG have also been investigated, for example Keinanen [61] proposes a CSG algorithm based on Simulated Annealing, while Di Mauro et al. [37] use a stochastic local search approach (GRASP) to iteratively build a coalition structure of high quality. Even if these approaches are not able to provide any guarantees on the solution quality, they can compute solutions for large-scale instances. Hence, in Section 5.4.5 we compare CFSS against C-Link, since it is the most recent heuristic approach for CSG and it has been tested using the collective energy purchasing function, which we also consider in this thesis.

3.3 Computing payments in the ϵ-kernel

The current state of the art approach to compute an ϵ-kernel payoff allocation for classic CF has been proposed by Shehory and Kraus [101] (Algorithm 6). Such an algorithm does not specify how x should be initialised, and assumes that a payoff vector is provided as an input. The first (and most expensive) phase is the computation of the *surplus matrix* s (lines 3–7), which iterates over the entire set of coalitions to assess the maximum excess (Equation 2.3) for each pair of agents in each coalition. Once the surplus matrix has been computed, a transfer between the pair of agents with the highest surplus difference (i.e., $s_{ij} - s_{ji}$) is set up, while ensuring that each payment is individually rational. These scheme is iteratively executed until the ratio between the maximum surplus difference δ and the value of the considered coalition structure is within a predefined parameter ϵ. This ensures that the computed payoff allocation is ϵ-kernel stable.

On the one hand, the computation of Equation 2.3 is a key bottleneck for classic CF, since it involves enumerating an exponential number of coalitions, i.e., $\Theta\left(2^n\right)$. On the other hand, when the size of the coalitions is limited to k members as in Social Ridesharing (see Chapter 6), such an algorithm has polynomial time complexity [101], since the coalitions are $O\left(n^k\right)$ [63].

Despite having polynomial time complexity under certain assumptions, such an approach has some drawbacks that hinder its applicability in real-world scenarios, and especially in the Social Ridesharing scenario we consider in Chapter 6. First, it is designed for classic CF, failing to exploit the graph-constrained nature of this problem. Second, this algorithm assumes that coalitional values can be assessed at no computational cost (e.g., stored in memory or provided by an oracle). This hypothesis, although appropriate in several settings, does not apply to SR, in which the value of a coalition is the solution of a routing problem and it cannot be stored in memory without limiting the scalability. These shortcomings lead to inefficiencies that prevent the application of the method proposed by Shehory and Kraus in our case, as discussed in detail in Section 7.1.

Algorithm 6 SHEHORYKRAUSKERNEL(x, CS, ϵ)

repeat
  for all $S \in CS$ **do** {For each coalition S in coalition structure CS}
    for all $a_i \in S$ **do** {For each pair of agents a_i and a_j in S}
      for all $a_j \in S - \{a_i\}$ **do**
        {s_{ij} is the maximum excess among all}
        {coalitions that include a_i but exclude a_j}
        $s_{ij} \leftarrow \max_{\{S' \in 2^A \mid a_i \in S', a_j \notin S'\}} e(S', x)$
  {a_{i^*} and a_{j^*} have the maximum surplus difference δ}
  $\delta \leftarrow \max_{(a_i, a_j) \in A^2} (s_{ij} - s_{ji})$
  $(a_{i^*}, a_{j^*}) \leftarrow \arg\max_{(a_i, a_j) \in A^2} (s_{ij} - s_{ji})$
  if $x[j^*] - v(\{a_{j^*}\}) < \delta/2$ **then** {Payments are individually rational}
    $d \leftarrow x[j^*] - v(\{a_{j^*}\})$
  else
    $d \leftarrow \delta/2$
  $x[j^*] \leftarrow x[j^*] - d$ {Transfer payment from a_{j^*}...}
  $x[i^*] \leftarrow x[i^*] + d$ {... to a_{i^*}}
until $\delta / V(CS) \leq \epsilon$

3.4 Constraint optimisation problems

Several solution techniques for COPs have been proposed in the constraint optimisation literature. On the one hand, Dynamic Programming (DP) approaches are primarily based on BE [31] (see Section 2.3), the solution framework adopted by our GPU implementation of BE, i.e., CUBE (see Chapter 8). This choice is motivated by the successful application of GPUs in the parallelisation of DP approaches [20, 44, 54, 107]. To the best of our knowledge, the only work that specifically focuses on BE is the one by Fioretto et al. [44], in which the authors devise an algorithm to realise the composition and marginalisation operations of BE (referred as *aggregate* and *project*) on GPUs, by exploiting the high degree of parallelism inherent in these operations. Hence, this work has been considered as a benchmark in our experimental evaluation in Section 8.4.2. In contrast with CUBE, our GPU implementation of BE (see Chapter 8, Fioretto et al. [44] realise the indexing of the tables is executed by using a *Minimal Perfect Hash* function [7], i.e., a hash function that maps n keys to n consecutive integers, which can be easily adopted as the indices of such keys. Although minimal perfect hash functions can be used in parallel by different threads to index the input, their construction is inherently sequential, since the index of a key depends on the indices assigned to the previously considered keys [2].

In the context of BE, Dechter [32] also proposed Mini-Bucket Elimination (MBE), an approximate version of such an algorithm that operates on smaller portions of the buckets (consequently referred as *mini*-buckets) rather than on the entire ones. By enforcing a limit on the maximum size of the mini-buckets, MBE is characterised by memory requirements which are exponential with respect to such limit. Nonetheless, such requirements are still significant and limit the application of MBE in realistic applications.

On the other hand, a recent strand of literature [76, 77] has investigated the use of AND/OR search trees, proposing several heuristic approaches and bounding methods to reduce the search space. Parallelisation has been investigated to speed-up search-based approaches for COP on multi-core CPUs [83], but the application of these techniques to GPUs is difficult for several reasons. On the one hand, general depth-first search is known to be difficult to parallelise [92], especially on highly parallel architectures such as GPUs. Moreover, the use of branch-and-bound may result in heavily unbalanced search trees, requiring complex techniques to balance the workload among the threads [83]. Such techniques are not effective on GPUs, where load balancing is crucial to achieve a high computational throughput. For the above reasons, we will investigate the parallelisation of such techniques on GPUs as future work.

Within the discussion of our COP model for GCCF (see Chapter 9), it is important to note that the time and space complexities of all the above algorithms are exponential with respect to the induced width w^* of the constraint network [31, 76]. For this reason, the formalisation of a particular problem (i.e., GCCF in our case) must result in a COP that yields an induced width of manageable complexity.

3.5 Belief propagation

BP on JTs represents a well-known inference algorithm, which has received significant attention in the parallel computing literature due to its high computational demand. In particular, Xia and Prasanna [117] proposed a distributed approach that combat this by decomposing the initial JT into a set of subtrees, and then performing the evidence propagation for each subtree in parallel on a cluster. In this thesis, we also focus on exploiting parallel architectures for BP on JTs, but, in contrast, we aim at parallelising the single propagation operation, which is the most computationally intensive task of the entire algorithm. Moreover, we focus on GPU parallelisation, rather than multi-core.

To the best of our knowledge, the work most related to BE on GPUs is presented by Zheng and Mengshoel [120], in which the authors propose a parallel approach for BP, and, in particular, they discuss a way to parallelise the atomic operations of propagation, so that it could be embedded in different algorithms. The authors devise a *two-dimensional parallelism*, in which an higher level *element-wise parallelism* is stacked on top of a lower level *arithmetic parallelism*, to better exploit the massive computational power provided by modern GPUs. In particular, element-wise parallelism is achieved by computing each of the $|R_{ij}|$ reduction-and-scattering operations (see Section 2.4) in parallel, which require $|R_{ij}|$ *mapping tables* (one per row of Sep_{ij}) to allow each thread to locate its input data from the corresponding potential tables. On the other hand, arithmetic parallelism represents the multi-threaded computation of each reduction-and-scattering operation, by means of well known parallel algorithms that can be found in literature [52].

Although this approach represents a significant contribution to the state of the art, there are some drawbacks that hinder its applicability. In particular, the proposed memory layout is not optimised for GPUs, for two main reasons:

- Threads need to access data in sparse and discontinuous memory locations using an additional indexing table, breaking coalescence and drastically reducing the throughput of memory transfers (see Section 2.5). Coalescence is crucial and it should be exploited in order to reduce memory accesses to the global memory, achieving a greater computational throughput.
- Since input data is organised in a discontinuous pattern rather than in continuous chunks, it is mandatory to transfer the entire potential tables to the global memory of the GPU before starting the computation of the BP algorithm, hindering two desirable properties: i) this approach is not applicable to potential tables that do not fit into global memory, since the sparsity of the data prevents any possibility of splitting them into smaller parts, and ii) since the computation cannot be started before the entire input has been copied to the GPU, such transfers cannot be amortised with pipelining (see Section 2.5.2).

Moreover, the authors devise this technique for BP, where tables are complete (i.e., they include a row for every possible assignment of the variables in their scope). Thus, this approach cannot be applied to problems in which tables are incomplete (see Definition 2.10), such as our COP model for GCCF discussed in Chapter 9.

Part II

Graph-Constrained Coalition Formation

4

CFSS: a branch-and-bound algorithm for GCCF

We now present a general algorithm to solve GCCF by showing that all feasible coalition structures induced by G can be modelled as the nodes of a search tree in which each feasible coalition structure is represented only once. This allows us to avoid any redundancy. Specifically, we first detail how we use edge contractions to represent the GCCF problem and then we provide a depth-first approach to build and traverse the search tree so to find the optimal solution.

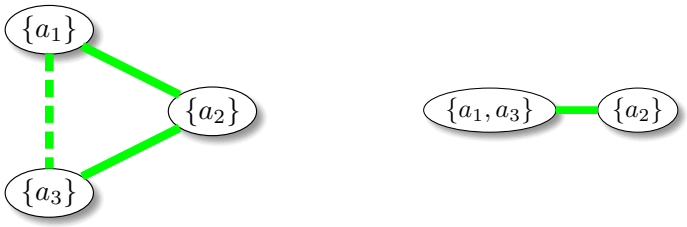
4.1 Generating feasible coalition structures via edge contractions

In this section we show that each $CS \in \mathcal{CS}(G)$ can be represented by a corresponding graph $\mathcal{G} = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A} \subseteq 2^A \setminus \{\emptyset\}$ and $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$, i.e., each node $u \in \mathcal{A}$ represents a particular coalition. Notice that in the initial graph $G = (A, E)$ each vertex $u \in A$ represents a single agent, and hence, G can be seen as the representation of the feasible coalition structure formed by all the singletons.

In what follows, we will show that, for each $CS \in \mathcal{CS}(G)$, the corresponding $\mathcal{G}$ can be obtained as the contraction of a set of edges of G, and that each contraction of a set of edges of G represents a feasible coalition structure $CS \in \mathcal{CS}(G)$. In more detail, let us define an *edge contraction* as follows.

Definition 4.1 (edge contraction). *Given a graph $\mathcal{G} = (\mathcal{A}, \mathcal{E})$ and an edge $e = (u, v) \in \mathcal{E}$, the result of the contraction of e is a graph $\mathcal{G}'$ obtained by removing e and the corresponding vertices u and v, and adding a new vertex $w = u \cup v$. Moreover, each edge incident to either u or v in $\mathcal{G}$ will become incident to w in $\mathcal{G}'$, merging the parallel edges (i.e., the edges that are incident to the same two vertices) that may result.*

Intuitively, one edge contraction represents the merging of the coalitions associated to the incident vertices. Figure 4.1 shows the contraction of the edge $(\{a_1\}, \{a_3\})$, which results in a new vertex $\{a_1, a_3\}$ connected to vertex $\{a_2\}$. Notice that edge contraction is a commutative operation (i.e., first contracting e and then e' results in the same graph as first contracting e' and then e). Hence, we define the contraction of a set of edges as the result of contracting each of the edges of the set in any given order.



(a) Before contraction. (b) After contraction.

Fig. 4.1: Example of an edge contraction.

Remark 4.2. Given a graph $\mathcal{G}$, the graph $\mathcal{G}'$ resulting from the contraction of any set of edges of $\mathcal{G}$ represents a feasible coalition structure, where coalitions correspond to the vertices of $\mathcal{G}'$.

Remark 4.3. Given a graph $\mathcal{G}$, any feasible coalition structure CS can be generated by contracting a set of edges of $\mathcal{G}$.

Thus, a possible way of listing all feasible coalition structures is to list the contraction of every subset of edges of the initial graph. However, notice that the number of subsets of edges is larger than the number of feasible coalition structures over the graph. For example, in the triangle graph in Figure 4.1a, the number of subsets of edges is $2^{|E|} = 2^3 = 8$, but the number of feasible coalition structures is 5 (i.e., $\{a_1\}\{a_2\}\{a_3\}$, $\{a_1, a_2\}\{a_3\}$, $\{a_1, a_3\}\{a_2\}$, $\{a_1\}\{a_2, a_3\}$ and $\{a_1, a_2, a_3\}$). This redundancy is due to the fact that the contraction of any two or three edges leads to the same coalition structure, i.e., the grand coalition $A = \{a_1, a_2, a_3\}$.

Hence, we need a way to avoid listing feasible coalition structures more than once. To avoid such redundancies, we mark each edge of the graph to keep track of the edges that have been contracted so far. Notice that there are only two different alternative actions for each edge: either we contract it, or we do not. If we decide to contract an edge, it will be removed from the graph in all the subtree rooted in the current node, but if we decide not to contract it, we have to mark such edge to make sure that we do not contract it in the future steps of the algorithm. To represent such marking, we will use the notion of *2-coloured graph*.

Definition 4.4 (2-coloured graph). *A 2-coloured graph $\mathcal{G}_c = (\mathcal{A}, \mathcal{E}, colour)$ is composed of a set of vertices $\mathcal{A} \subseteq 2^A \setminus \{\emptyset\}$ and a set of edges $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$, as well as a function $colour : \mathcal{E} \to \{red, green\}$ that assigns a colour (red or green) to each edge of the graph.*

In our case, a red edge means that a previous decision not to contract that edge was made. On the one hand, green edges can be still contracted. Figure 4.2a shows an example of a 2-colour graph in which edge $(\{a_1\}, \{a_4\})$ is coloured in red (dotted line). Hence, in any subsequent step of the algorithm it is impossible to contract it. On the other hand, all other edges in such graph can still be contracted. In a 2-coloured graph, we define a *green edge contraction* (e.g., dashed line in Figure 4.2a) as follows.

(a) Before contraction (b) After contraction

Fig. 4.2: Example of a 2-coloured edge contraction (best viewed in colour).

Definition 4.5 (green edge contraction). *Given a 2-coloured graph* $\mathcal{G}_c = (\mathcal{A}, \mathcal{E}, colour)$ *and a green edge* $e \in \mathcal{E}$*, the result of the contraction of* e *is a new graph* $\mathcal{G}'_c$ *obtained by performing the contraction of* e *on* $\mathcal{G}_c$*. Whenever two parallel edges are merged into a single one, the resulting edge is coloured in red if at least one of them is red-coloured, and it is green-coloured otherwise.*

The rationale behind marking parallel edges in this way is that, whenever we mark an edge $e = (u, v)$ to be *red*, we want the agents in u and v to be in separate coalitions, hence whenever we merge some edges with e we must mark the new edge as *red* to be sure that future edge contractions will not generate a coalition that contains both the agents corresponding to nodes u and v. For example, note that in Figure 4.2 the red edge $(\{a_1\}, \{a_4\})$ (dotted in the figure) and the green edge $(\{a_4\}, \{a_3\})$ are merged as a consequence of the contraction of edge $(\{a_1\}, \{a_3\})$, resulting in an edge $(\{a_4\}, \{a_1, a_3\})$ marked in red, so to enforce that any possible contraction in the new graph will keep agents a_1 and a_4 in separate coalitions.

Having defined how we can use the edge contraction operation to generate feasible coalition structures, we now provide a way to generate the whole search space of feasible coalition structures.

4.2 Generating the entire search space

Given the green edge contraction operation defined above, we can generate each feasible coalition structure exactly once. In more detail, at each point of the generation process, each red edge indicates that it has been discarded for contraction from that point onwards, and hence its vertices cannot be joined. Observe that the way we defined green edge contraction guarantees that the information encoded in red edges is always preserved. Thus, given a 2-coloured graph, its children can be readily assessed as follows: for each edge in the graph, we generate the graph that results from contracting that edge. Moreover, we colour the selected edge in red so that it cannot be contracted again in subsequent edge contractions. Algorithm 7 implements the depth-first[1] generation and traversal of our search tree, in which each feasible coalition structure is evaluated by means of the characteristic function and compared with the best (i.e., the one with the highest value) coalition structure so far. Once the search tree has been entirely traversed, Algorithm 7 outputs the optimal solution.

[1] DFS allows us to traverse the entire tree with polynomial memory requirements.

Algorithm 7 SolveGCCF(G)

1: $\mathcal{G}_c \leftarrow G$ with all green edges
2: $best \leftarrow \mathcal{G}_c$ {Initialise current best solution with singletons}
3: $Front \leftarrow \emptyset$ {Initialise search frontier $Front$ with an empty stack}
4: $Front$.PUSH $(\mathcal{G}_c)$ {Push $\mathcal{G}_c$ as the first node to visit}
5: **while** $Front \neq \emptyset$ **do** {Search loop}
6: $node \leftarrow Front$.POP $()$ {Get current node}
7: **if** $V(node) > V(best)$ **then** {Check function value}
8: $best \leftarrow node$ {Update current best solution}
9: $Front$.PUSH(Children $(node)$) {Update $Font$}
10: **return** $best$ {Return optimal solution}

Algorithm 8 Children($\mathcal{G}_c$)

1: $\mathcal{G}'_c \leftarrow \mathcal{G}_c = (\mathcal{A}, \mathcal{E}, colour)$ {Initialise graph $\mathcal{G}'_c$ with $\mathcal{G}_c$}
2: $Ch \leftarrow \emptyset$ {Initialise the set of children}
3: **for all** $e \in \mathcal{E} : colour(e) = green$ **do** {For all green edges}
4: $Ch \leftarrow Ch \cup \{$GreenEdgeContr $(\mathcal{G}'_c, e)\}$
5: Mark edge e with colour red in $\mathcal{G}'_c$
6: **return** Ch {Return the set of children}

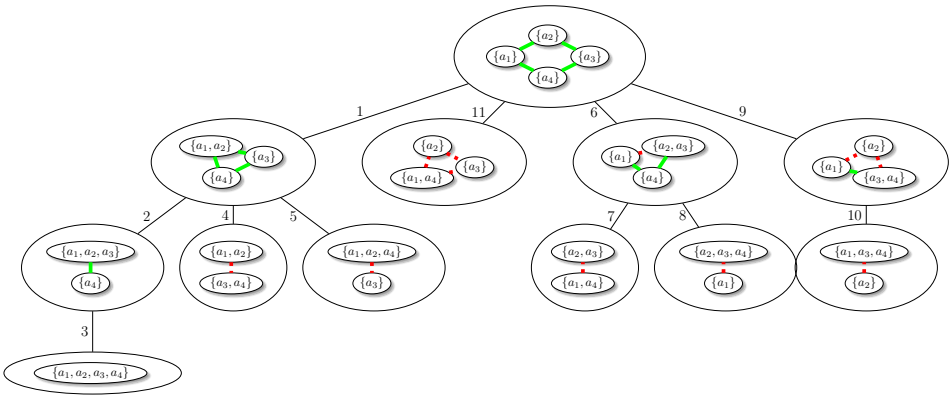


Fig. 4.3: Search tree for a square graph (best viewed in colour).

As an example, Figure 4.3 shows the search tree generated starting from a square graph, highlighting each generation step with labels on the edges. It is possible to show that Algorithm 7 visits all feasible coalition structures and each of them is visited only once. In particular, we can prove the following proposition.

Proposition 4.6. *Given a 2-coloured graph $\mathcal{G}_c$, the tree generated by Algorithm 7 rooted at $\mathcal{G}_c$ contains all the coalition structures compatible with $\mathcal{G}_c$, i.e., all the coalition structures that do not violate any constraint induced by the red edges in $\mathcal{G}_c$. Moreover, each of them appears only once.*

Proof. By induction on the number of green edges. If there is no green edge, then the tree has just one element which corresponds to the only coalition structure compatible with $\mathcal{G}_c$. Assume that the statement is true for $n-1$ green edges. Let $\mathcal{G}_c$ have n green edges and CS be a coalition structure compatible with $\mathcal{G}_c$. If no edge in $\mathcal{G}_c$ is contractible with respect to CS, then CS is the coalition represented by $\mathcal{G}_c$, and it cannot be in any of its children, because each of them contracts an edge in $\mathcal{G}_c$. Thus CS appears in the tree rooted at $\mathcal{G}_c$ only once (at the root). Assume then that there is at least one green edge in $\mathcal{G}_c$ contractible with respect to CS. Then CS cannot be the coalition structure at the root. We now identify a child $\mathcal{G}'_c$ such that CS is compatible with $\mathcal{G}'_c$. The first child of the root contracts an edge e. If e is contractible with respect to CS, then the first child of $\mathcal{G}_c$ is compatible with CS. Otherwise e is red-colourable with respect to CS. The same procedure goes on with the remaining children. Thus, by construction, the root has three kind of children with respect to CS: some which contract a red-colourable edge, a single child $\mathcal{G}'_c$ that contracts a contractible edge and red-colours some red-colourable edges, and from there on some that red-colours a contractible edge. It is easy to see that CS is compatible only with one child, namely $\mathcal{G}'_c$. Now $\mathcal{G}'_c$ has at most $n-1$ green edges and by induction CS must appear in that subtree only once. Thus, it appears in the tree rooted at $\mathcal{G}_c$ only once. □

As a consequence, we prove the time complexity of Algorithm 7.

Proposition 4.7. *The time complexity of Algorithm 7 is* $O(|\mathcal{CS}(G)| \cdot |\mathcal{E}|)$.

Proof. There is a bijection between $\mathcal{CS}(G)$ and the nodes visited by Algorithm 7, by direct application of Proposition 4.6 to G with all green edges. The creation of each new node yields a GreenEdgeContraction(G, e) operation, whose complexity is $O(|\mathcal{E}|)$ (see Definition 4.5). Hence, the complexity of creating the entire search tree is $O(|\mathcal{CS}(G)| \cdot |\mathcal{E}|)$.[2]

Therefore, we can always find the optimal solution of a GCCF problem by visiting the entire search tree and evaluating each feasible coalition structure, which is represented only once in the tree. Hence, if we search different branches in parallel (i.e., assigning different iterations of the loop at line 3 of Algorithm 8 to different threads/cores), we will not have redundant computations.

Nonetheless, even for sparse graphs the number of feasible coalition structures can be very large, making their visit not affordable. In fact, the GCCF problem is NP-complete [114], as, in general, the number of feasible coalition structures is exponential in the number of agents. For this reason, we propose a branch and bound technique that helps prune significant parts of the search space when the characteristic function belongs to a general class of function, i.e., $m+a$ functions.

[2] Notice that, since CSG is a particular case of GCCF (i.e., CSG is a GCCF problem with a complete graph), $|\mathcal{CS}(G)|$ can be, in the worst case, equivalent to the n^{th} Bell number, i.e., $\Omega(\frac{n}{\ln(n)}^n)$ [10]. Nonetheless, such exponential complexity is not representative of the problems we are interested to solve, i.e., problems in which G is sparse and, hence, $\mathcal{CS}(G)$ contains a lower number of feasible coalition structures.

4.3 $m + a$ functions

In this section we present a general class of characteristic functions, namely $m+a$ functions, showing that they can be seen as the sums of a superadditive and a subadditive part [84].[3] Such functions are particularly interesting as they can be used to model several realistic GCCF scenarios, which we describe in detail in Chapter 5. Furthermore, $m + a$ functions allow us to devise an algorithm (i.e., the CFSS algorithm) that can be employed to compute solutions for large-scale systems, thanks to two fundamental properties. On the one hand, $m+a$ functions are *closed-form* functions, hence it is not necessary to store the values of all feasible coalitions in memory[4] since each coalition can be evaluated on-the-fly only when needed. On the other hand, they enable an efficient bounding technique, discussed in Section 4.4, that helps prune significant parts of the search space. Such method allows us to compute the optimal solution for any GCCF problem based on an $m + a$ function by generating only a minimal portion of the solution space (i.e., less than 0.32% in our experiments in Section 5.4.2)

Definition 4.8 (superadditive (resp. subadditive) $v(\cdot)$ function). *Given a graph G, a function $v : \mathcal{FC}(G) \to \mathbb{R}$ is superadditive (resp. subadditive) if the value of the union of disjoint coalitions is no less (resp. no greater) than the sum of the coalitions' separate values, i.e., $v(S \cup T) \geq (\text{resp. } \leq)\, v(S) + v(T)$ for all $S, T \subseteq \mathcal{A}$ such that $S \cap T = \emptyset$.*

We define such properties for the function $V : \mathcal{CS}(G) \to \mathbb{R}$ defined in Equation 2.1.

Definition 4.9 (superadditive (resp. subadditive) $V(\cdot)$ function). *Given a graph G, a function $V : \mathcal{CS}(G) \to \mathbb{R}$ defined according to Equation 2.1 is superadditive (resp. subadditive) if the underlying function $v : \mathcal{FC}(G) \to \mathbb{R}$ is superadditive (resp. subadditive).*

Definition 4.10 ($m + a$ $V(\cdot)$ function). *Given a graph G, a function $V : \mathcal{CS}(G) \to \mathbb{R}$ is an $m + a$ function if it is the sum of a superadditive function $V^+ : \mathcal{CS}(G) \to \mathbb{R}$ and a subadditive function $V^- : \mathcal{CS}(G) \to \mathbb{R}$.*

Note that, a similar decomposition has been previously proposed by Shekhovtsov et al. [103, 104], focusing on general functions that can be decomposed as the sum of supermodular and submodular components, exploiting such a property to achieve better results in the solution of several optimisation problems. Submodular functions have been widely studied in the optimisation literature [98] in virtue of their natural diminishing returns[5] property [80, 81].

[3] Notice that if the characteristic function is superadditive (resp. subadditive), then the solution of the GCCF problem is trivially the grand coalition (resp. the singletons).

[4] The classic formulation of the CSG problem requires to store the value of each coalition in memory, which requires an exponential amount of memory.

[5] Informally, a submodular function has the property that the difference in the value of the function that a single element makes when added to an input set decreases as the size of the input set increases.

Definition 4.11 (submodular (resp. supermodular) $v(\cdot)$ function). *Given a set A, a submodular (resp. supermodular) function [98] is a function $v : 2^A \to \mathbb{R}$ which satisfies the property*

$$v(S) + v(T) \geq (\text{resp. } \leq)\, v(S \cup T) + v(S \cap T), \quad \forall S, T \subseteq A.$$

It is important to note that our result (i.e., Theorem 4.17) holds for superadditive and subadditive functions (cf. Definition 4.8), which are *weaker* (i.e., more general) properties with respect to supermodularity and submodularity. In fact, it is easy to show that supermodularity (resp. submodularity) implies superadditivity (resp. subadditivity), but the converse is not true [98].

4.4 The CFSS algorithm

We now describe CFSS [13, 14] (Coalition Formation for Sparse Synergies), our branch and bound approach to GCCF when applied to the family of $m+a$ characteristic functions. Specifically, we provide a technique to compute an upper bound for the value assumed by the characteristic function in every coalition structure of the subtree $ST(CS_i)$ rooted at a given coalition structure CS_i. In order to explain how to compute such an upper bound, we first define the element $\overline{CS_i}$.

Definition 4.12 ($\overline{CS_i}$). *Given a feasible coalition structure CS_i represented by a 2-coloured graph $\mathcal{G}_c$, the coalition structure $\overline{CS_i}$ can be obtained by removing all red edges from $\mathcal{G}_c$ and then contracting all the remaining green edges (which is equivalent to finding the connected components in the graph after the removal of all red edges).*

Furthermore, we also detail some properties of our domain.

Lemma 4.13. *$\mathcal{CS}(G)$ is a lattice, i.e., a partially ordered set, in which every two elements CS_i and CS_j have a supremum ($CS_i \vee CS_j$) and an infimum ($CS_i \wedge CS_j$).*

We define the following partial order over coalition structures.

Definition 4.14 (order among coalition structures). *Given any two coalition structures CS_i and CS_j, we say that $CS_i \leq CS_j$ if every element of CS_i is a subset of some element of CS_j.*

As an example, $\{a_1, a_2\}\{a_3\} \leq \{a_1, a_2, a_3\}$, but the order between $\{a_1, a_2\}\{a_3\}$ and $\{a_1\}\{a_2, a_3\}$ is not defined. It is well known that with this partial order the set of partitions forms a complete lattice (see Section V.4 in [48]), called the partition lattice or equivalence lattice. It is easy to see that our domain of interest, i.e., the set of feasible coalition structures induced by G, is sublattice of the partition lattice, and thus it is a lattice. Furthermore, in our scenario, the grand coalition represents a supremum of any two elements, while the coalition structure of all singletons represents an infimum.

Lemma 4.15. *CS_i is the infimum of the subtree rooted at CS_i, i.e., $CS_i = \bigwedge ST(CS_i) = \inf ST(CS_i)$, where $ST(CS_i)$ is the subtree rooted at CS_i.*

In the search tree defined in Section 4 each child is the result of contracting an edge in the parent. As a consequence of the contraction, two of the coalitions in the parent are merged, making the child partition coarser than that of the parent. Henceforth, the elements of $\mathcal{CS}(G)$ can be arranged in an order-preserving tree: whenever CS_j is a descendant of CS_i in the tree, then $CS_j \geq CS_i$. As a consequence, the above statement holds.

Lemma 4.16. *Given a node CS_i, $\overline{CS_i}$ is bigger than any of the elements of the subtree, i.e., $\overline{CS_i} \geq \bigvee ST(CS_i) = \sup ST(CS_i)$.*

Since $\overline{CS_i}$ represents the connected components in the graph after the removal of all red edges, it can be interpreted as the coarsest partition ignoring the constraints imposed by the red edges. Clearly, any partition in the subtree will be at most as coarse as this one, since red edges will prevent the merging of the coalition they connect. Given the above, we now prove the following theorem.

Theorem 4.17. *Given an $m+a$ function $V : \mathcal{CS}(G) \to \mathbb{R}$, then $M(CS_i) = V^-(CS_i) + V^+\left(\overline{CS_i}\right)$ is an upper bound for the value assumed by such function in every coalition structure of the subtree $ST(CS_i)$ rooted at CS_i, i.e.,*

$$M(CS_i) = V^-(CS_i) + V^+\left(\overline{CS_i}\right) \geq \max\{V(CS_j) \mid CS_j \in ST(CS_i)\}. \quad (4.1)$$

Proof. Consider that, for the subtree rooted at CS_i, the maximum of a subadditive function will be achieved at CS_i (Lemma 4.15), i.e., $V^-(CS_i) \geq \max\{V^-(CS_j) \mid CS_j \in ST(CS_i)\}$. On the other hand, the maximum of a superadditive function will be reached at one of the leaves. However, since assessing the supremum $\overline{CS_i}$ of the subtree is computationally efficient (Lemma 4.16), we can bound $V^+(\cdot)$ in the subtree as $V^+\left(\overline{CS_i}\right) \geq \max\left\{V^+(CS_j) \mid CS_j \in ST(CS_i)\right\}$. Since $V(\cdot)$ is an $m+a$ function, then we can provide an upper bound for such a function by composing these two results, i.e., $M(CS_i) = V^-(CS_i) + V^+\left(\overline{CS_i}\right)$. □

Algorithm 9 CFSS(G)

```
1: G_c ← G with all green edges
2: best ← G_c {Initialise current best solution with singletons}
3: Front ← ∅ {Initialise search frontier Front with an empty stack}
4: Front.PUSH(G_c) {Push G_c as the first node to visit}
5: while Front ≠ ∅ do {Branch and bound loop}
6:    node ← Front.POP() {Get current node}
7:    if M(node) > V(best) then {Check bound value}
8:       if V(node) > V(best) then {Check function value}
9:          best ← node {Update current best solution}
10:      Front.PUSH(CHILDREN(node)) {Update Front}
11: return best {Return optimal solution}
```

Building upon Theorem 4.17, we can efficiently assess an upper bound for the value of the characteristic function in any subtree and prune it, if such a value is smaller than the value of the best solution found so far. CFSS is implemented by Algorithm 9. Such an algorithm does not specify the order in which the children of the current node should be visited, namely the operation of the CHILDREN function. However, this order has a strong influence on the performance of CFSS (as shown in Section 5.4.3), since it can be used to compute a tighter upper bound on the characteristic function, to improve the effectiveness of the branch and bound.

4.4.1 Edge ordering heuristic

In this section we propose a heuristic to define a total ordering among the edges of a graph G, in order to guide the traversal of the search tree. This results in a significant speed-up of the algorithm, by means of an improvement of the upper bound. In particular, we notice that the value of $M\left(CS_i\right) = V^-\left(CS_i\right) + V^+\left(\overline{CS_i}\right)$ is heavily influenced by the value of $V^+\left(\overline{CS_i}\right)$. In fact, it is possible that $\overline{CS_i} = \{A\}$ (i.e., the grand coalition), when CS_i contains enough green edges to connect all the nodes of the graph G. This results in a poor bound, since V^+ is a superadditive function and it reaches its maximum value for A. On the other hand, if red edges form a cut-set for the 2-coloured graph, the procedure in Definition 4.12 results in a coalition structure $\overline{CS_i} = \{S_1, S_2\}$, as Figure 4.4 shows.

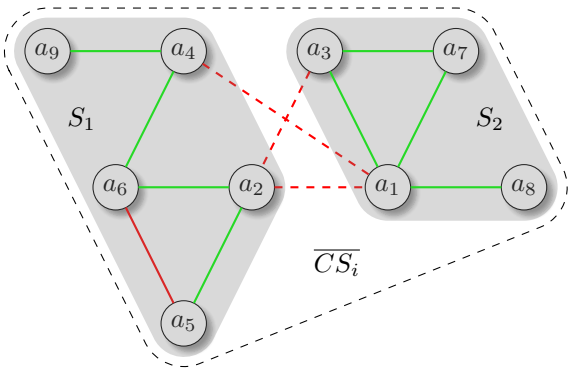


Fig. 4.4: Example of a partition with a cut-set of 3 edges (best viewed in colour).

In this case, our bounding technique produces a *lower* upper bound $M\left(CS_i\right) = V^-\left(CS_i\right) + v^+\left(S_1\right) + v^+\left(S_2\right)$, since $v^+\left(\cdot\right)$ is superadditive and, therefore, $v^+\left(S_1\right) + v^+\left(S_2\right) \leq v^+\left(A\right)$. Notice that, having an upper bound that provides a lower overestimation of the characteristic function is crucial for the performance of CFSS, as the condition at line 7 in Algorithm 9 would be verified less often, hence allowing us to prune bigger portions of the search space. Also, it easy to see that when the value of the characteristic function increases in a non-linear way with respect to the size of the coalitions (such as the functions we consider in this thesis), the more S_1 and S_2 are closer to a *bisection* of A (i.e., the more $|S_1|$ and $|S_2|$ are close to $|A|/2$), the more pronounced such improvement is. Following this observation, it

Algorithm 10 ORDER(G)

```
1: i ← 1 {Initialise edge counter}
2: G_o ← G {Initialise the ordered graph}
3: Q ← ∅ {Initialise an empty queue}
4: Q.PUSH (G) {Push G as the first graph to partition}
5: while Q ≠ ∅ do {Partitioning loop}
6:    ⟨G_1, G_2, E'⟩ ← CUT (Q.POP ()) {Partition current graph}
7:    Label in G_o each edge ∈ E' from i to i + |E'| − 1
8:    i ← i + |E'| {Increase edge counter}
9:    if |A_1| > 1 then {If the first subgraph has at least 2 nodes...}
10:      Q.PUSH (G_1) {... enqueue it}
11:   if |A_2| > 1 then {If the second subgraph has at least 2 nodes...}
12:      Q.PUSH (G_2) {... enqueue it}
13: return G_o {Return ordered graph}
```

is preferable to visit the edges that produce a cut of the graph in the first steps of the algorithm, since they will result in the above-explained improvement once such edges are marked in red. Henceforth, we define a total ordering among the edges of G, producing an *ordered* graph G_o by means of Algorithm 10. Intuitively, such algorithm computes small[6] cut-sets by means of the routine CUT(G), which outputs the subgraphs $G_1 = (A_1, E_1)$ and $G_2 = (A_2, E_2)$ resulting from the cut, and the cut-set E'. Once we find the cut-set, we label its edges as the first ones in the ordered graph, recursively applying such procedure for all the subsequent subgraphs, until every edge has been ordered.

In addition to this edge ordering heuristic, our bounding technique can be employed to provide anytime approximate solutions, as shown in the next section.

4.4.2 Anytime approximate properties

Theorem 4.17 can be directly applied to compute an overall bound of an $m + a$ function, with anytime properties. More precisely, let us consider frontier $Front$ in Algorithm 9 (line 3). When we expand such a frontier (line 10) we keep track of the highest value of $V(\cdot)$ in the visited nodes. Hence, given a frontier $Front$, the bound $B(Front)$ is defined as

$$B(Front) = \max\{V(best), \max_{CS \in Front} M(CS)\} \tag{4.2}$$

[6] Ideally, we would prefer to compute the *smallest* cut-set, in order to traverse the minimum number of edges necessary to partition the graph. Unfortunately, such a problem (known as the Minimum Bisection problem) is a well known NP-complete problem [46]. However, our heuristic does not need an optimal solution, since if a suboptimal cut-set (i.e., bigger than the optimal one) is used, our algorithm will still partition the graph in a higher number of steps, resulting in a slightly smaller improvement. We adopt an approximate algorithm implemented with the METIS graph partitioning library [58], a standard tool that allows to compute good-quality cut-sets.

Thus, $B(Front)$ is the maximum between the values assumed by $V(\cdot)$ inside the frontier (i.e., $V(best)$) and an estimated upper bound outside of it (i.e., $\max_{CS \in Front} M(CS)$). Notice that since each $M(CS)$ is an overestimation of the value of $V(\cdot)$ in the corresponding subtree, such a maximisation provides a valid upper bound for $V(\cdot)$ in the portion of search space not visited yet. Furthermore, the quality of $B(Front)$ can only be improved by expanding $Front$. More formally, if $Front'$ is such an expansion, then

$$B(Front) \geq B(Front') \geq \max\{V(CS) \mid CS \in \mathcal{CS}(G)\}. \tag{4.3}$$

This can be easily verified using the definition of $M(\cdot)$. In fact, each bound resulting from the children of a substituted node $u \in Front$ must be less or equal to $M(u)$ and, hence, Inequality 4.3 holds. Intuitively, the larger the search space explored, the better is the bound provided. Finally, notice that the fastest way to compute a bound for $V(\cdot)$ is to consider a frontier formed exclusively by the root (i.e., the coalition structure formed by all singletons). Assessing this bound has the same time complexity of computing M, i.e., $O(|E|)$, and, in some scenarios, its quality can be satisfactory, as shown in Section 5.4.4.

5

Applications for GCCF

In this chapter we present our experimental evaluation, in which we benchmark the CFSS algorithm on several application scenarios that can be modelled as GCCF problems. We present three scenarios: the *collective energy purchasing* scenario (Section 5.1), *edge sum with coordination cost* scenario (Section 5.2), and the *coalition size with distance cost* scenario (Section 5.3). Specifically, we are interested in the characterisation of such scenarios as $m + a$ functions, so to employ the CFSS algorithm discussed in Chapter 4.4.

Then, in Section 5.4 we discuss our experimental evaluation, showing that CFSS outperforms DyCE (i.e., the state of the art algorithm to solve GCCF problems) and that it can compute good-quality approximate solutions for systems with thousands of agents.

5.1 Collective energy purchasing

In the *collective energy purchasing* scenario, agents form coalitions to buy energy together at cheaper prices [112]. Specifically, each agent is characterised by an energy consumption profile that represents its energy consumption throughout a day. A profile records the energy consumption of a household at fixed intervals (every half hour in our case). Hence each profile is a vector representing the actual measurements collected over a month from 2732 households in the UK. The characteristic function of a coalition of agents is the total cost that the group would incur if they bought energy as a collective in two different markets: the spot market, a short term market (e.g., half hourly, hourly) intended for small amounts of energy; and the forward market, a long term one in which larger amounts of energy (spanning weeks and months) can be bought at cheaper prices [112]. Farinelli et al. [42] proposed to model the collective energy purchasing scenario with the characteristic function

$$v\left(S\right) = \underbrace{\sum_{t=1}^{T} q_S^t\left(S\right) \cdot p_S + T \cdot q_F\left(S\right) \cdot p_F}_{energy(S)} + \kappa\left(S\right),$$

where $T = 48$ is the number of energy measurements in each profile, $p_S \in \mathbb{R}^-$ and $p_F \in \mathbb{R}^-$ represent the unit prices of energy in the spot and forward market respectively, $q_F : \mathcal{FC}(G) \to \mathbb{R}^-$ stands for the time unit amount of electricity to buy in the forward market and $q_S^t : \mathcal{FC}(G) \to \mathbb{R}^-$ for the amount to buy in the spot market at time slot t.[1] $energy : \mathcal{FC}(G) \to \mathbb{R}^-$ is the total energy cost.

Finally, $\kappa : \mathcal{FC}(G) \to \mathbb{R}^-$ stands for a coalition management cost that depends on the size of the coalition and captures the intuition that larger coalitions are harder to manage. The definition of this cost depends on several low level issues (e.g., the capacity of the power networks connecting the customers in the groups, legal fees, and other costs associated to group contracts, etc.), hence a precise definition of this term goes beyond the scope of this thesis. Following Farinelli et al. [42] we use $\kappa(S) = -|S|^\gamma$ with $\gamma > 1$ to introduce a non-linear element that penalises the formation of larger coalitions, so that the grand coalition is not always the best coalition structure. Hence, the *collective energy purchasing* function is defined as

$$V(CS) = \underbrace{\sum_{S \in CS} \left[\sum_{t=1}^{T} q_S^t(S) \cdot p_S + T \cdot q_F(S) \cdot p_F \right]}_{V^+(CS)} + \underbrace{\sum_{S \in CS} \kappa(S)}_{V^-(CS)}. \tag{5.1}$$

Proposition 5.1. *The collective energy purchasing function is $m + a$.*

Proof. As shown in Equation 5.1, such function can be seen as an $m + a$ function, being the sum of a superadditive function, consisting of the cost of the energy necessary to fulfil the aggregated consumption profiles of the coalitions, and a subadditive one (i.e., the sum of the coalition management costs). On the one hand, since the baseline (i.e., the minimum) of the aggregate energy profile of a coalition $S_{12} = S_1 \cup S_2$ is no less than the sum of the baselines of the energy profiles of S_1 and S_2, the members of S_{12} can buy from the forward market an amount of energy which is no less than the sum of the amounts that could have been bought by S_1 and S_2 separately.[2] Therefore, the $energy(\cdot)$ function is superadditive. On the other hand, it is trivial to verify that $\kappa(\cdot)$ is a subadditive function. □

5.2 Edge sum with coordination cost

Our second test case is represented by the *edge sum with coordination cost* function, in which every edge of G is associated with a real value by means of a function $w : E \to \mathbb{R}$ [36]. This value defines a pairwise relation between two nodes, that represents how well (or bad) those agents perform together, or the cost of completing a coordination task in a robotic environment [27]. Each coalitional value is the sum of the weights of the edges among its members. We also introduce a

[1] Unit prices (whose values are reported in Section 5.4) are negative numbers, i.e., they belong to the set $\mathbb{R}^- = \{i \in \mathbb{R} \mid i \leq 0\}$, to reflect the direction of payments. Thus, the values of the characteristic function are negative as well, hence they represent costs that, maintaining the maximisation task, we aim to minimise.

[2] A more detailed discussion is provided by Vinyals et al. [112].

penalising factor $\kappa(S)$,[3] with the same definition given in the previous section. Such penalising factor takes into account management and communication costs in larger coalitions [102]. Hence, we define this characteristic function as

$$v(S) = \sum_{e \in edges(S)} w(e) + \kappa(S), \tag{5.2}$$

where the function $edges : \mathcal{FC}(G) \to 2^E$ provides the set of all the edges connecting any two members of a given coalition S, i.e.,

$$edges(S) = \{(v_1, v_2) \in E \mid v_1 \in S \text{ and } v_2 \in S\}.$$

In order to characterise this scenario with an $m + a$ function, we rewrite Equation 5.2 as

$$v(S) = \sum_{e \in edges(S)} \left[w^+(e) + w^-(e)\right] + \kappa(S),$$

$$\text{where} \quad w^+(e) = \begin{cases} w(e), \text{ if } w(e) \geq 0, \\ 0, \quad \text{otherwise,} \end{cases} \quad \text{and} \quad w^-(e) = \begin{cases} w(e), \text{ if } w(e) < 0, \\ 0, \quad \text{otherwise.} \end{cases}$$

In other words, $\sum_{e \in edges(S)} w^+(e)$ represents the sum of all the positive weights of the edges in $edges(S)$, while $\sum_{e \in edges(S)} w^-(e)$ represents the sum of the negative ones. The *edge sum with coordination cost* function is then defined as

$$V(CS) = \underbrace{\sum_{S \in CS} \left[\sum_{e \in edges(S)} w^+(e) \right]}_{V^+(CS)} + \underbrace{\sum_{S \in CS} \left[\sum_{e \in edges(S)} w^-(e) + \kappa(S) \right]}_{V^-(CS)}. \tag{5.3}$$

Proposition 5.2. *The edge sum with coordination cost function is $m + a$.*

Proof. Equation 5.3 highlights the $V^+(\cdot)$ and $V^-(\cdot)$ components of this function. On the one hand, $v^+(S) = \sum_{e \in edges(S)} w^+(e)$ is clearly superadditive, since a coalition $S_{12} = S_1 \cup S_2$ contains an amounts of positive edges which is no less than the total amount of positive edges in S_1 and S_2 taken separately, hence $v^+(S_{12}) \geq v^+(S_1) + v^+(S_2)$. Similarly, it is easy to show that $\sum_{e \in edges(S)} w^-(e)$ is a subadditive function. □

The *edge sum with coordination cost* function allows a more precise upper bound with respect to the one provided by Theorem 4.17, as shown in Lemma 5.3.

Lemma 5.3. *Given a 2-coloured graph $\mathcal{G}_c = (\mathcal{A}, \mathcal{E})$ corresponding to the coalition structure CS_i, $\sum_{\{e \in \mathcal{E} \mid colour(e) = green\}} w^+(e)$ is an upper bound for the values of the* edge sum with coordination cost *function in the subtree rooted at CS_i.*

Proof. Given a node CS_j in the subtree rooted at CS_i, the set of edges considered by $V^+(CS_j)$ (i.e., $\bigcup_{S \in CS_j} edges(S)$) is a proper subset of $\{e \in \mathcal{E} \mid colour(e) = green\}$, since we only contract green edges in the creation of such subtree. Since $V^+(CS_j) \geq V(CS_j)$, the value of $V(CS_j)$ is lower than the given expression. □

[3] Such penalising factor makes the edge sum with coordination cost function to violate the IDM property (cf. Section 3.2.3), therefore the approach proposed by Voice et al. [114] cannot be used.

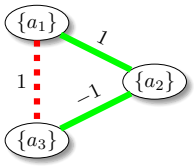


Fig. 5.1: 2-coloured graph with weighted edges (best viewed in colour).

We now provide an example in order to explain why such upper bound is more effective than the one provided by Theorem 4.17. Consider the coalition structure CS_i represented by the 2-coloured graph in Figure 5.1. In this case, $\overline{CS_i} = \{a_1, a_2, a_3\}$, since Definition 4.12 ignores red edges. As a consequence, the upper bound provided by Theorem 4.17 is $V^+\left(\overline{CS_i}\right) = 2$. Notice that such value also considers the red edge, even if such edge cannot be contracted, and hence, can never be part of any coalition. In contrast, Lemma 5.3 does not consider such edge and thus, it results in an upper bound equal to 1. In conclusion, a lower upper bound results in a more effective pruning, as confirmed by our experiments in Section 5.4.2.

5.3 Coalition size with distance cost

In our final scenario, we consider the formation of coalitions where bigger groups are favoured and which minimises the distance of the opinion among their members. Such application could be employed to cluster public opinion, or to detect the presence of "virtual coalitions" among members of a parliament based on their recorded votes (e.g., the votes by the Democratic and the Republican parties) in terms of similarity. Such a model can be modelled evaluating each S with

$$v(S) = |S|^{\alpha} - \sum\nolimits_{(i,j)\in S\times S} d(i,j), \tag{5.4}$$

where $\alpha \geq 1$, and $d : A \times A \to \mathbb{R}^+$ is a function that measures the distance between the opinions of agent i and agent j. From Equation 5.4 it follows that the input of our problem has size n^2, where n is the total number of agents, since we must know the distances between each pair or agents. The *coalition size with distance cost* function of a coalition structure CS is then defined as

$$V(CS) = \underbrace{\sum\nolimits_{S\in CS} |S|^{\alpha}}_{V^+(CS)} + \underbrace{\sum\nolimits_{S\in CS} \left[-\sum\nolimits_{(i,j)\in S\times S} d(i,j) \right]}_{V^-(CS)}.$$

Proposition 5.4. *The coalition size with distance cost function is $m + a$.*

Proof. On the one hand, it is easy to verify that $v^+(S) = |S|^{\alpha}$ is a superadditive function, assuming that $\alpha \geq 1$. On the other hand, $v^-(S) = -\sum_{(i,j)\in S\times S} d(i,j)$ is subadditive, since

$$v^-(S_1 \cup S_2) = v^-(S_1) + v^-(S_2) - \sum_{i\in S_1, j\in S_2} d(i,j) \leq v^-(S_1) + v^-(S_2). \quad \square$$

In what follows, we show how we benchmark our approach my means of the above discussed characteristic functions.

5.4 Empirical evaluation

The main goals of our empirical evaluation of CFSS are:

1. To evaluate its runtime performance with respect to DyCE considering a variety of graphs, both realistic (i.e., subgraphs of the Twitter network) and synthetic (i.e., scale-free networks and community networks).
2. To evaluate the effectiveness of our bounding technique.
3. To evaluate the anytime performance and guarantees that our approach can provide when scaling to very large numbers of agents (i.e., more than 2700).
4. To compare the quality of our approximate solutions with the ones computed by C-Link [42] (see Section 3.2.5) on large-scale instances.
5. To evaluate the speed-up that can be obtained with a parallel version of CFSS.
6. To evaluate the speed-up produced by our edge ordering heuristic.

Following Voice et al. [115], we consider scale-free networks generated with the Barabási-Albert model with the parameter m within $\{1, 2, 3\}$ (i.e., every newly added node is connected, on average, to m already existing nodes). We generated community networks with the BTER model [65], with an average degree (denoted as $\overline{deg}$) within $\{2, 4, 6\}$. We compare our approach with DyCE in our three reference domains. In our characteristic functions we use the following parameters:

- Following Farinelli et al. [42], in the *collective energy purchasing* function we fixed $p_S = -80$ and $p_F = -70$.
- In the *edge sum with coordination cost* function we assigned a uniformly distributed random weight within $[-10, 10]$ to each edge.
- Following Farinelli et al. [42], in both the above scenarios we considered $\gamma = 1.3$.
- In the *coalition size with distance cost* function we assigned a uniformly distributed random value within $[0, 100]$ to each distance between a pair of different agents (with $d(i, i) = 0$), and we considered $\alpha = 2.2$, motivated by the remarks in Section 5.4.4.

We conducted an additional set of experiments (considering the *collective energy purchasing* function) in which the graph G is a subgraph of a large crawl of the Twitter social graph. Specifically, such dataset is a graph with 41.6 million nodes and 1.4 billion edges published as part of the work by Kwak et al. [67]. In particular, G is obtained by means of a standard algorithm [94] to extract a subgraph from a larger graph, i.e., a breadth-first traversal starting from a random node of the whole graph, adding each node and the corresponding arcs to G, until the desired number of nodes is reached.

Moreover, we implemented a multi-threaded version of CFSS, namely P-CFSS (i.e., Parallel CFSS), and we analysed the speed-up of P-CFSS using Amdahl's law [3], as it provides the maximum theoretical speed-up that can be achieved. All our results refer to the average value over 20 repetitions for each experiment. CFSS[4] and C-Link are implemented in C, while we used the DyCE implementation provided by its authors. We run our tests on a machine with a 3.40GHz CPU and 16 GB of memory.

[4] Our implementation is available at `https://github.com/filippobistaffa/CFSS`.

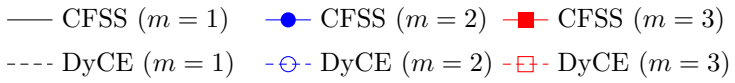


Fig. 5.2: Legend for scale-free networks.

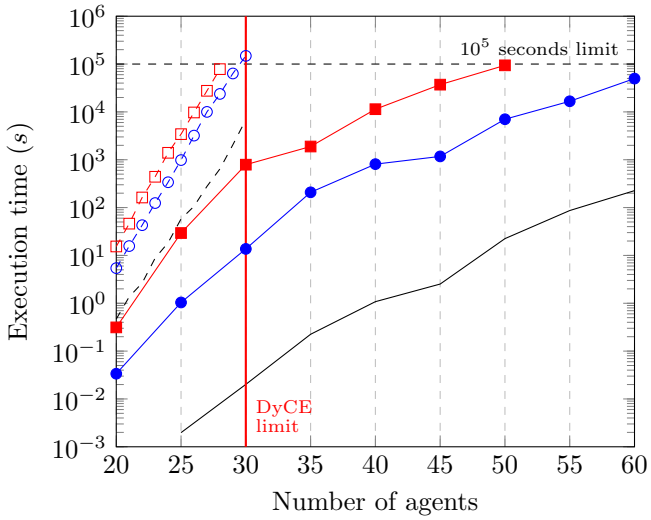


Fig. 5.3: Edge sum with coordination cost, scale-free networks.

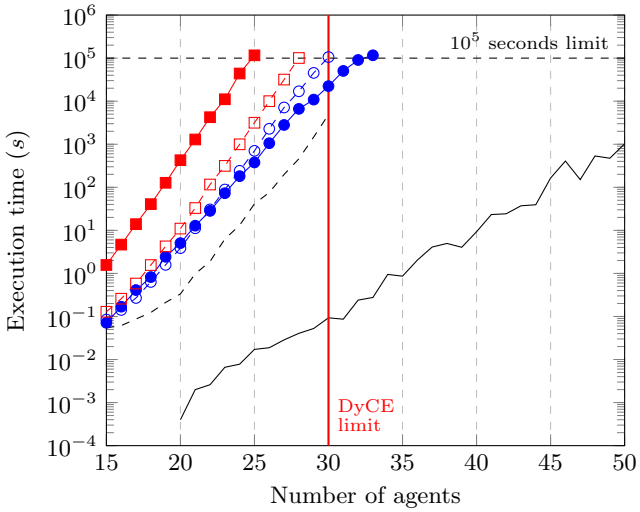


Fig. 5.4: Collective energy purchasing, scale-free networks.

5.4.1 DyCE vs. CFSS: runtime comparison

In our experiments using scale-free networks, CFSS outperforms DyCE when coalition values are shaped by the above-described benchmark functions (as shown in Figures 5.3–5.6). In all our tests, we increased the number of agents until the execution time reached 10^5 seconds.

Specifically, for the *edge sum with coordination cost* function (Figure 5.3), CFSS outperforms DyCE by 4 orders of magnitude on networks with average connectivity (i.e., for $m = 2$), and by 3 orders of magnitude on networks with higher connectivity (i.e., for $m = 3$). These results are due to the fact that the bounding technique in Lemma 5.3 allows us to prune significant portions of the search space. This is confirmed by the experiments in Section 5.4.2.

In the *collective energy purchasing* scenario (Figure 5.4) with 30 agents and $m = 2$, CFSS is 4.7 times faster than DyCE, and it is at least 2 orders of magnitude faster for $m = 1$. However, DyCE is significantly faster (44 times) than CFSS for $m = 3$. We notice a similar performance when we run both algorithms considering the community network topology (Figure 5.5).

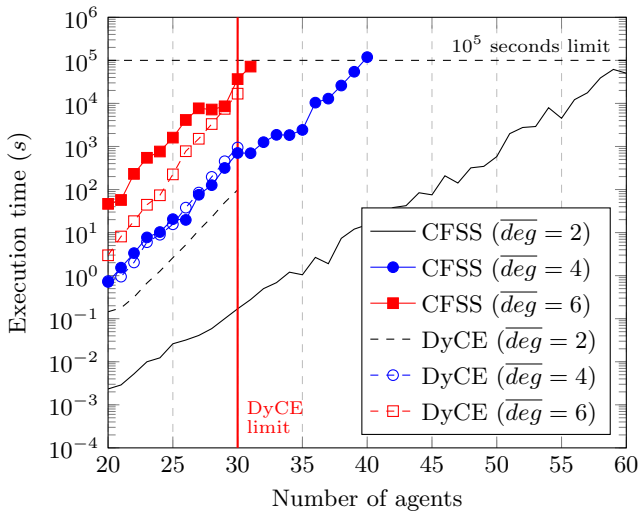


Fig. 5.5: Collective energy purchasing, community networks.

The adoption of the *coalition size with distance cost* function (Figure 5.6) produces a similar behaviour, with a performance improvement for our method. In fact, CFSS is 17 times faster than DyCE for $m = 2$, and only 3 times slower for $m = 3$. On the other hand, the runtime of DyCE equals the previous case, since this approach is not sensitive to the values of the characteristic function.

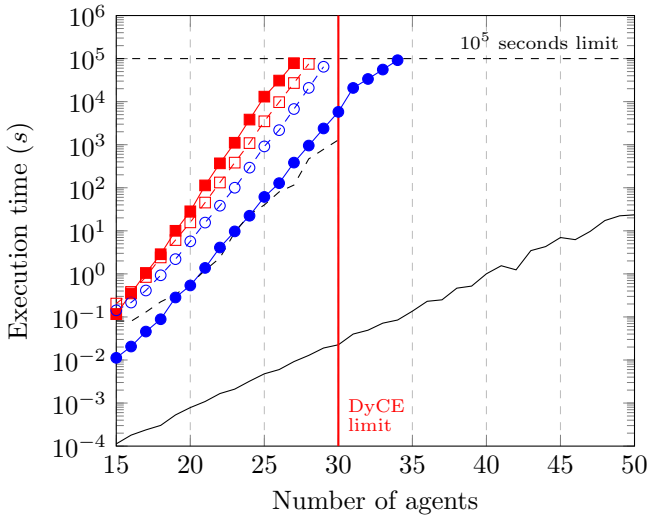


Fig. 5.6: Coalition size with distance cost, scale-free networks.

On Twitter subgraphs (Figure 5.7), CFSS is at least four orders of magnitude faster than DyCE when solving instances with 30 agents (the biggest instances that DyCE can solve), and it can scale up to 45 agents. Note that this is due to the runtime limit we imposed (i.e., 10^5 seconds), and not due to technical limitations of our approach. These results confirm the very good performance of CFSS when considering sparse networks. In fact, the average degree of these subgraphs is comparable with the one of a scale-free network with $1 < m < 2$.

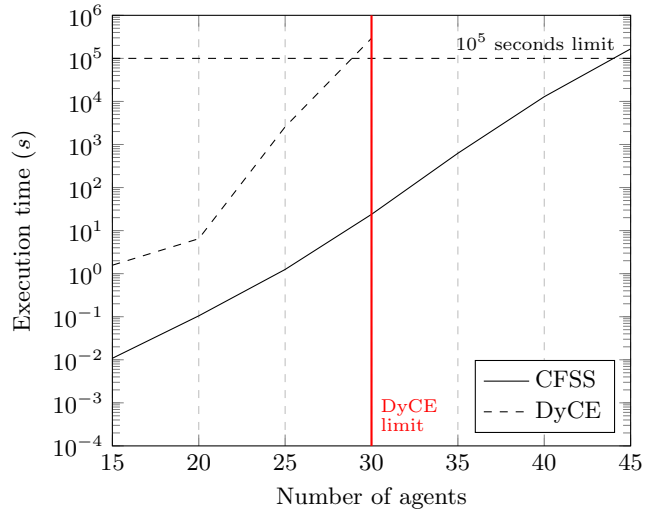


Fig. 5.7: Collective energy purchasing, Twitter subgraphs.

Notice that, in general, DyCE cannot scale over 30 agents (due to its exponential memory requirements), while CFSS does not have such limitation, hence it is possible to reach instances with thousands of agents (cf. Section 5.4.4).

5.4.2 Bounding technique effectiveness

To quantify the effectiveness of our bounding method, we compared the number of configurations explored by CFSS with respect to the entire search space, i.e., the one explored by Algorithm 7, to measure the number of search nodes pruned by our bounding technique. These test have been run on instances with $n = 30$, adopting scale-free networks with $m = 2$. The results show that, when the coalitional values are provided by the *collective energy purchasing* function, CFSS can compute the optimal solution exploring a number of configurations which is, on average, 0.32% of the entire search space. We measured a similar value in the *coalition size with distance cost* scenario (i.e., 0.28%). Notice that, in the *edge sum with coordination cost* scenario only 0.0045% of the entire search space is explored. Thanks to the effectiveness of our bounding technique, our algorithm can compute optimal solutions in scenarios that would otherwise be untractable.

5.4.3 Edge ordering heuristic

The following table shows the speed-up obtained by using the ordering heuristic described in Section 4.4.1 with respect to an arbitrary ordering, considering the *collective energy purchasing* and the *coalition size with distance cost* functions. Even though our heuristic is applicable also in the *edge sum with coordination cost* scenario, such function has not been included in this analysis since, as stated in Lemma 5.3, it allows an ad-hoc bounding method that is more effective than the general one (see above section). Our experiments show a clear benefit in the adoption of such a heuristic, producing a maximum performance gain of 843% in the first scenario and 338% in the second one. Across all experimental scenarios, such a heuristic allows an average speed-up of 295% considering both domains.

Characteristic function	**Minimum**	**Average**	**Maximum**
Collective energy purchasing	176%	367%	843%
Coalition size with distance cost	136%	222%	338%

5.4.4 Anytime approximate performance

In this section we evaluate the performance of the approximate version of CFSS on a very large set of agents (i.e., 2732). A standard measure to evaluate approximate algorithms is the *Performance Ratio* (PR) [4].

Definition 5.5 (performance ratio). *Given an instance I of an optimisation problem, its optimal solution $Optim(I)$ and an approximate solution $Approx(I)$, the* performance ratio $PR(I)$ *is defined as*

$$PR(I) = \max\left(\frac{Approx(I)}{Optim(I)}, \frac{Optim(I)}{Approx(I)}\right).$$

Both in the case of minimisation and maximisation problems, the PR is equal to 1 in the case of an optimal solution, and can assume arbitrarily large values in the case of poor approximate solutions. In our case, computing the optimal solution $Optim(I)$ for large-scale GCCF problems is not possible, hence the PR is not an applicable measure of quality. Thus, we define the *Maximum Performance Ratio* (MPR) following the above definition, and considering the upper bound on the optimal solution defined in Equation 4.2 instead of the optimal solution itself.

Definition 5.6 (maximum performance ratio). *Given a GCCF instance I, we denote the approximate solution computed by CFSS as $Approx(I)$ and the bound on the optimal solution defined in Equation 4.2 as $Bound(I)$. Then, we define the Maximum Performance Ratio $MPR(I)$ as*

$$MPR(I) = \max\left(\frac{Approx(I)}{Bound(I)}, \frac{Bound(I)}{Approx(I)}\right).$$

Since $|Bound(I)| \leq |V(CS_I^*)|$ (Equation 4.3), where CS_I^* is the optimal solution of I, $MPR(I)$ represents an upper bound for $PR(I)$. The MPR provides an important quality guarantee on the approximate solution $Approx(I)$, since $Approx(I)$ cannot be worse than by a factor of $MPR(I)$ with respect to the optimal solution.

Figure 5.8 shows the value of the MPR in the *collective energy purchasing* scenario, using $n \in \{100, 500, 1000, 1500, 2000, 2732\}$ and $m = 4$ and Twitter subgraphs as network topologies, and considering a time budget of 100 seconds. Other values for m show a similar behaviour (not reported here). We plot the average and the standard error of the mean over 20 repetitions. It is clear that the network topology does not impact the quality guarantees of our approach, hence we only adopt scale-free networks in the following experiments. In contrast, the MPR is heavily influenced by the nature of the characteristic function, as clarified later.

In addition, the results show that, for 100 agents, the provided bound is only 4.7% higher than the solution found within the time limit, reaching a maximum of +11.65% when the entire dataset is considered, i.e., with 2732 agents. Such small decrease is due to the fact that, for bigger instances, it is possible to explore a smaller part of the search space in the considered time budget, leaving a bigger portion to the estimation of the bound. Nonetheless, in this experiment CFSS provides a MPR of at most 1.12 and thus solutions that are at least 88% of the optimal. This confirms the effectiveness of this bounding technique applied to the energy domain, which allows us to provide solutions and quality guarantees for problems involving a very large number of agents. In our experiments, the bound is assessed at the root, without any frontier expansion. In this way, the bound can be computed almost instantly, thus devoting all the available runtime to the search for a solution. This choice is further motivated by the fact that, in this scenario, the bound improves of a negligible value in the first levels of the search tree, due to the particular definition of the characteristic function. More precisely, if we consider a frontier formed by the children of the root, in each of them the bound of $V^-(\cdot)$ will improve by a factor of $2^\gamma - 2 \approx 1.5$ (i.e., the difference between the coalition management cost of the new coalition and the ones of the two merged singletons). On the other hand, the bound of $V^+(\cdot)$ will remain constant: in fact, since we are taking the maximum (i.e., the worst) bound at the frontier (as shown

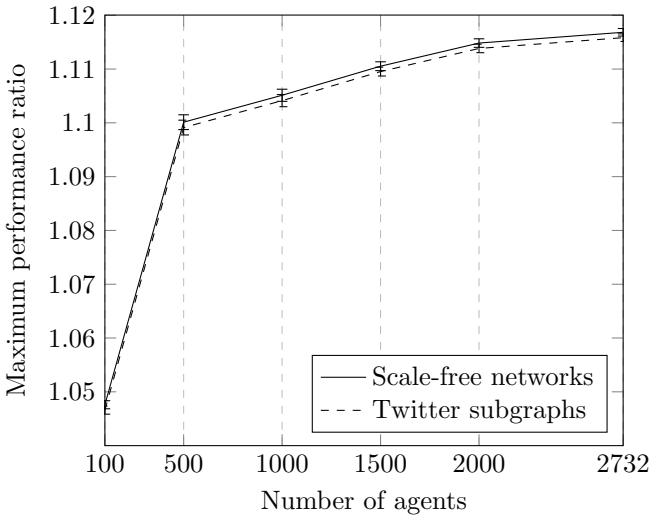


Fig. 5.8: MPR for collective energy purchasing.

in Equation 4.2), the result of this maximisation will still be equal to the $V^{+}(\cdot)$ value of the grand coalition (as it was at the root), because in at least one of the children nodes the computation of $\overline{CS}$ will result in joining all the agents together. For this domain it is not worth to expand the frontier from the root, since the gain would be insignificant with respect to the additional computational cost.

The MPR exhibits a different behaviour when considering the *coalition size with distance cost* function, being heavily influenced by the value of the α exponent. Figure 5.9 shows how the MPR varies significantly with respect to $\alpha \in [2,3]$, growing up to 41825.6 for $\alpha = 2.4$ and then reducing to 1.13 for $\alpha = 2.7$, with a tendency to 1 when increasing this exponent. This behaviour can be explained by considering the structure of the characteristic function. Up to $\alpha = 2.4$, the subadditive component (i.e., $-\sum_{C \in CS} \sum_{(i,j) \in C \times C} d(i,j)$) dominates the superadditive one (i.e., $\sum_{C \in CS} |C|^{\alpha}$), hence the search for a solution is not able to find any coalition structure better than the initial one (i.e., the coalition structure with all singletons, which is probably the optimal one). Nonetheless, the MPR keeps growing when we increase α, since it equals $\frac{n^{\alpha}}{n} = n^{\alpha-1}$, i.e., the bound computed at the root (i.e., $V^{+}(A) = n^{\alpha}$) divided by the value of the initial solution (i.e., n). On the other hand, when α is sufficiently large (i.e., for $\alpha = 2.5$), this behaviour is inverted, because $V^{+}(\cdot)$ has a greater impact and the entire characteristic function tends to become superadditive. Thus, coalition structures closer to the grand coalition represent good solutions, hence the MPR tends to 1 when we increase α.

These remarks motivate us to study the impact of α also on the optimal algorithm. Figure 5.10 displays the runtime needed to find the optimal solution on random instances with 25 agents on scale-free networks with $m = 2$, showing that the performance of CFSS decreases when we increase α from 2 to 3. The value of the bound provided by Equation 4.1 is larger when α grows, hence its quality decreases, producing a less effective pruning and, thus, a higher runtime.

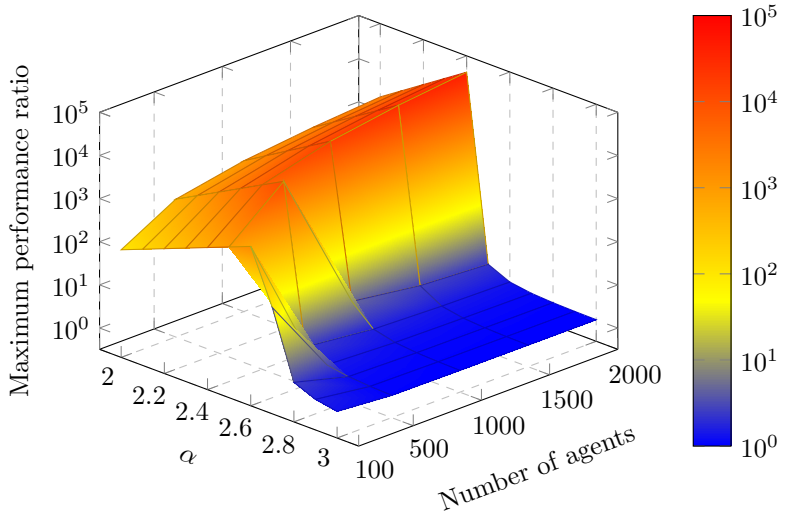


Fig. 5.9: MPR for coalition size with distance cost (best viewed in colour).

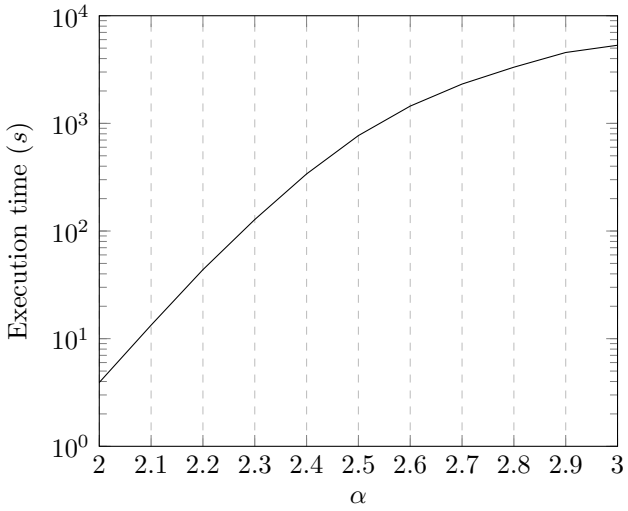


Fig. 5.10: Execution time with respect to α.

To summarise, the adoption of a bigger α in the *coalition size with distance cost* function negatively impacts the performance of our approach when computing optimal solutions, while improving approximate solutions as α grows. This motivates our choice of defining $\alpha = 2.2$ in the previous experiments, as it represents a good value to benchmark CFSS. In fact, it is big enough to avoid excessively low runtimes in the optimal version, but it does not exceed 2.4, beyond which the quality guarantees it provides are extremely good (i.e., the MPR tends to 1).

5.4.5 CFSS vs. C-Link: solution quality comparison

We further evaluate the approximate performance of CFSS by comparing it against C-Link [42], an heuristic approach to solve CSG based on hierarchical clustering. As noted in Section 3.2.5, heuristic approaches are usually very fast in computing an approximate solution, but they cannot provide any quality guarantee on such solution. Hence, in this section we are interested in comparing the quality of the solution computed by the approximate version of CFSS (which does provide quality guarantees) with respect of the one computed by C-Link. We chose C-Link among the other approaches discussed in Section 3.2.5 because it is the most recent one and it has also been tested using the *collective energy purchasing* function by its authors. We adopt the same experimental setting discussed in the previous section, i.e., we consider scale-free networks with $n \in \{100, 500, 1000, 1500, 2000, 2732\}$ and $m = 4$ (generating 20 random repetitions of each experiment), and we adopt the *collective energy purchasing* characteristic function. We solve each instance with C-Link (adopting the best heuristic proposed by Farinelli et al. [42], i.e., Gain-Link) and then we run CFSS on the same instance with a time budget equal to C-Link's runtime.

Figure 5.11 shows the average and the standard error of the mean of the ratio between the value of the solution computed by C-Link and the one computed by CFSS. Since we consider solutions with negative values, when such ratio is > 1 the solution computed by C-Link is better (i.e., it corresponds to a lower cost) than the one computed by CFSS. Our results show that, even though C-Link can compute better solutions, the quality of our solutions is worse only by 3% for 100 agents. When we consider the entire dataset (i.e., with 2732 agents) the quality of our solutions is still within the 9% with respect to the counterpart. Notice that CFSS also provides quality guarantees on such solutions, while C-Link does not provide any guarantees.

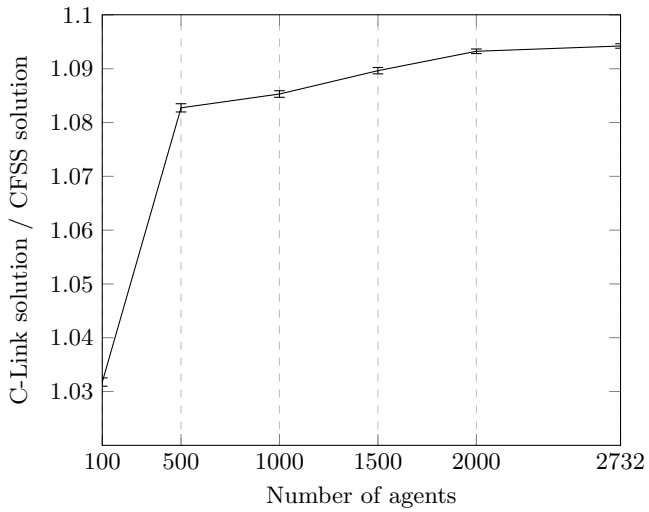


Fig. 5.11: Ratio between C-Link and CFSS approximate solutions.

5.4.6 P-CFSS

Here we detail the parallelisation approach of the multi-threaded version of CFSS, analysing the speed-up with respect to its serial version. Following Bader et al. [6], parallelisation is achieved by having different threads searching different branches of the search tree. The only required synchronisation point is the computation of the current best solution that must be read and updated by every thread. In particular, the distribution of the computational burden among the t_a available threads is done by considering the first i subtrees rooted in every node of the first generation (starting from the left) and assigning each of them to t_j threads ($1 \leq j \leq i$). The remaining rightmost subtrees are computed by a team of $t_a - \sum_{j=1}^{i} t_j$ threads using a dynamic schedule, i.e., once a thread has completed the computation of one subtree, it starts with one of the remaining ones. Parameters i and t_j are manually set, since it is assumed (and verified by an empirical analysis) that the distribution of the nodes over the search tree does not significantly vary among different instances. These parameters can be obtained by means of more advanced techniques [71], which involve the estimation of the number of nodes in the search tree. On the other hand, a preliminary investigation of such an approach revealed that the obtained estimation is not precise enough to result in an effective load balancing. We run P-CFSS on random instances with 27 agents on scale-free networks with $m = 2$, using a machine with 2 Intel® Xeon® E5-2420 processors. The speed-up measured during these tests has been compared with the maximum theoretical one provided by Amdahl's Law, considering an estimated non-parallelisable part of 5%, due to memory allocation and thread initialisation.

As can be seen in Figure 5.12, the actual speed-up follows the theoretical one quite closely as long as the number of threads does not exceed 12, the number of physical cores. After that, hyper-threading still provides some improvement, reaching a final speed-up of 9.44 with all 24 threads active.

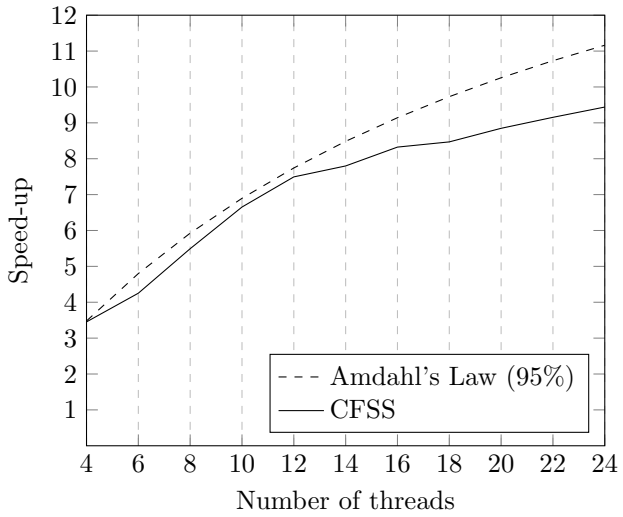


Fig. 5.12: Multi-core speed-up.

Part III

Cardinality-Constrained Coalition Formation

6

Social Ridesharing

In the previous chapters, we have tackled the CF problem by considering a type of constraints induced by a graph connecting the agents. On the other hand, in many realistic applications the formation of coalitions may also be influenced by constraints of different nature. For instance, if coalitions are mapped to physical objects with limited capacity, it is natural to enforce a constraint on the *cardinality* of such coalitions [102]. A straightforward real-world example is *ridesharing*, in which agents represent users that need to commute across a geographical space (usually within a city), and coalitions represent cars that are shared among multiple agents with the objective of reducing travel costs. In this case, the cardinality of coalitions is limited by the number of seats in each car, which is usually quite low (e.g., 5 seats [118]).

Formally, we denote a CF problem that involves constraints on the cardinality of coalitions as *cardinality-constrained CF*. In particular, in the following two chapters we focus on a cardinality-constrained CF scenario in the context of ridesharing, denoted as *Social Ridesharing*. In such scenario, a set of commuters, connected through a social network, arrange one-time rides at short notice. Specifically, in this chapter we focus on the associated optimisation problem of forming cars in order to minimise the travel cost of the overall system, i.e., the CSG problem. Then, in Chapter 7 we address the other fundamental aspect of CF, i.e., payment computation, in the context of SR.

Figure 6.1 shows a system with 6 agents that have to commute across a urban scenario. Now, it is easy to see that a_2 could share a portion of the trip with a_1, if the latter decides to travel through a path that covers a_2's start and destination points. The same discussion applies to a_3 and a_4, and finally to a_5 and a_6. Furthermore, it is reasonable to assume that a social network exist among such agents, and that an agent prefers to share the ride with a friend, in contrast to travelling with complete strangers. This assumption is motivated by a clear tendency among ridesharing companies, which tend to favour the formation of groups of users that are connected in such network. In fact, Uber and Lyft incentivise users to share rides with their friends, showing that social relationships play a fundamental role in the ridesharing scenario, which is consequently referred as Social Ridesharing.

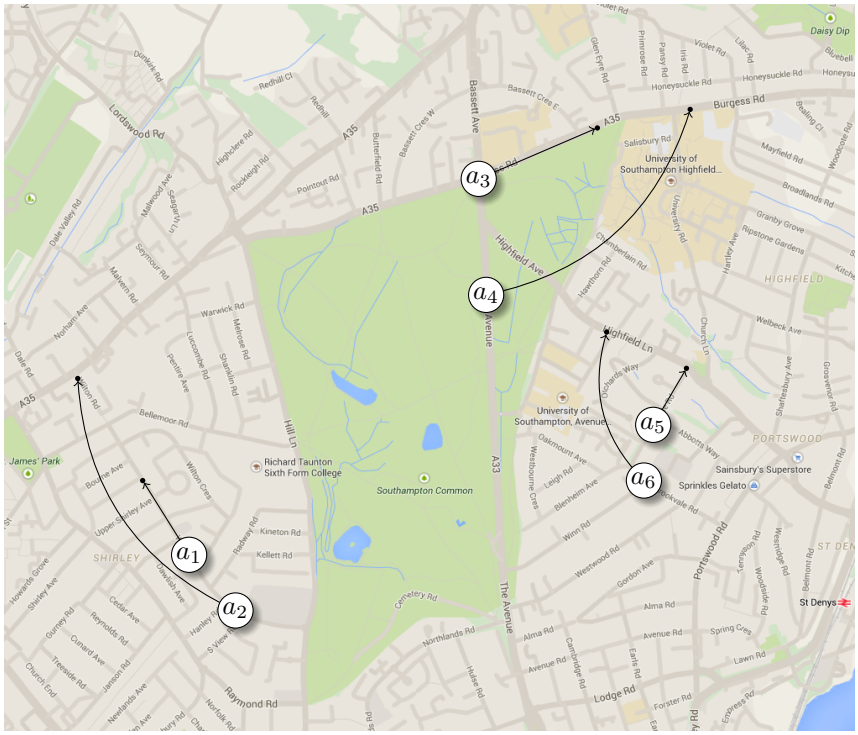


Fig. 6.1: Example of starting and destination points of 6 agents.

The Social Ridesharing scenario can be naturally modelled as a GCCF problem, where the set of feasible coalitions is restricted by a graph (i.e., the social network) and by some additional feasibility constraints (see Section 6.1), e.g., the number of members of each coalition cannot exceed the number of seats of the corresponding cars. Figure 6.2 shows how the above example results in a coalition structure CS formed by three coalitions, each composed by 2 agents.

The Social Ridesharing problem aims at minimising the total cost of all the cars formed by the system. As a consequence, it is natural to define the value of each coalition as the travel cost of the associated car, as suggested by Kamar and Horvitz [57]. However, such work is mostly focused on incentive design aspects for ridesharing, while we are interested in solving the optimisation problem posed by SR. Against this background, we present the first model that encodes the above discussed scenario as a GCCF problem (Section 6.1), and we formally define the value of each coalition on the basis of the spatial preferences of the agents. Then, we show how to solve such problem with the CFSS algorithm (Section 6.2).

Subsequently, we generalise our model by incorporating the temporal preferences of the agents (Section 6.3), so to allow them to express constraints on the departure and the arriving time. Finally, in Section 6.4 we empirically evaluate our approach, showing that it can produce significant cost reductions (up to -36.22%) and it features a good scalability, computing approximate solutions for very large systems (i.e., up to 2000 agents) and good quality guarantees within minutes.

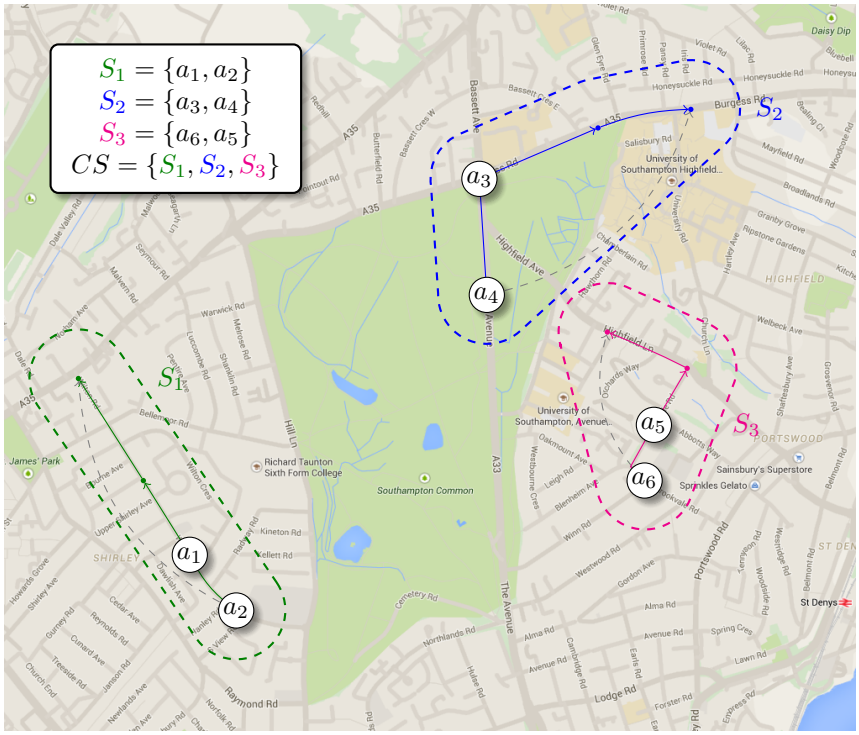


Fig. 6.2: Social Ridesharing with 6 agents and 3 coalitions (best viewed in colour).

6.1 Problem definition

The Social Ridesharing (SR) problem [18] considers a set of riders $A = \{a_1, \ldots, a_n\}$, where $n > 0$ is the total number of riders, and a non-empty[1] set of drivers $D \subseteq A$, containing the riders owning a car. Every driver $a_i \in D$ can host up to $seats\,(a_i)$ riders in his car, including himself, where the function $seats : A \to \mathbb{N}^0$ provides the number of seats of each car. If $a_i \notin D$, then $seats\,(a_i) = 0$.

The *map* of the geographic environment in which the SR problem takes place is represented by a connected graph $M = (P, Q)$, where P is the set of geographic points of the map and $Q \subseteq P \times P$ is the set of edges among these points. Each edge is associated to a *length* by means of the function $\lambda : Q \to \mathbb{R}^+$.

Definition 6.1 (path). *A path composed by m points is represented as a tuple $L \in P^m$, denoting as $L[k]$ the k^{th} point of L.*

Definition 6.2 ($\mathcal{L}$). *$\mathcal{L}$ is the set of all the possible paths among the points in P.*

Each rider $a_i \in A$ has to commute from a starting point $p_i^a \in P$, i.e., its pick-up point, to a destination point $p_i^b \in P$.

A key aspect of the SR scenario is the presence of a social network, modelled as a graph $G = (A, E)$ with $E \subseteq A \times A$, which restricts the formation of groups. To this end, we adopt Definition 2.1, defining a *feasible* coalition as follows.

[1] If $D = \emptyset$ the problem is trivial, as the only solution is represented by the singletons.

Definition 6.3 (feasible coalition for SR). *Given a graph G and a set of riders $S \subseteq A$, S is a* feasible *coalition if it induces a connected subgraph on G, and if it contains at least one rider whose car has enough seats for all the members. Formally, we state such a requirement as:*

Constraint 1 *$|S| > 1 \implies \exists a_i \in S \cap D : seats(a_i) \geq |S|$, i.e., at least one rider has a car with enough seats for all the riders.*

Notice that such a constraint allows a rider $a_i \notin D$ to be in a singleton. In fact, if the total number of available seats is less than the total number of riders in the system, such a rider might need to resort to public transport paying a cost $k(\{a_i\})$ for the ticket. Formally, the function $k : A_{single} - D_{single} \rightarrow \mathbb{R}^-$ provides such a cost, where $A_{single} = \{\{a_i\} \mid a_i \in A\}$, $D_{single} = \{\{a_i\} \mid a_i \in D\}$, and $\mathbb{R}^-$ represents the set of negative real numbers including zero.[2] Notice that if $a_i \in D$, then $\{a_i\}$ is not associated to any value by $k(\cdot)$, as we assume that such riders always prefer to use their car with respect to public transport.

Now, in several ridesharing online services (e.g., *Lyft* and *Uber*) a commuter declares whether he is available as a driver or as a rider, hence the two sets are disjoint and a feasible set of riders S contains at most one driver. Formally, the following additional constraint must hold:

Constraint 2 *$|S \cap D| \leq 1$, i.e., the number of drivers per car can be at most 1.*

Notice that Constraint 2 is optional, but it holds in several established real-world services, arising from aspects of practical nature. Nonetheless, since our approach supports a more general model, it can also be applied to scenarios where such a constraint does not hold (we will clearly specify this in what follows).

Having defined our notion of feasible coalition, in the following section we detail how we associate a value to each feasible coalition, i.e., we define the *characteristic function* adopted by our SR problem.

6.1.1 Coalitional value definition

When a car is formed, it drives through a path that contains all the starting and destination points of its passengers. Notice that not all the permutation of these points are valid (e.g., it is not reasonable to go to a rider's destination point and then to its starting point). More formally, a *valid* path must fulfil two constraints to correctly accommodate the needs of all the passengers.

Definition 6.4 (valid path). *Given a feasible set of riders S and a path $L \in \mathcal{L}$ of m points, L is said to be* valid *if the following constraints hold:*

Constraint 3 *$\exists a_i \in S : seats(a_i) \geq |S| \wedge L[1] = p_i^a \wedge L[m] = p_i^b$, i.e., L goes from the driver's starting point to its destination.*

Constraint 4 *$\forall a_i \in S \, \exists x, y : L[x] = p_i^a \wedge L[y] = p_i^b \wedge x < y$, i.e., for each rider, its starting point precedes its destination.*

[2] We define $\mathbb{R}^+ = \{i \in \mathbb{R} \mid i \geq 0\}$. $\mathbb{N}^+$ and $\mathbb{N}^-$ are defined in the corresponding ways.

Henceforth, we refer to the set of all valid paths for a given feasible set of riders S with $\mathcal{VL}(S)$. Following Kamar and Horvitz [57], we define the total cost $v(S)$ of a feasible set of riders S as

$$v(S) = \begin{cases} k(S) & \text{if } S \cap D = \emptyset, \\ \underbrace{t(L_S^*) + c(L_S^*) + f(L_S^*)}_{value(L_S^*)} & \text{otherwise.} \end{cases} \tag{6.1}$$

On the one hand, if $S \cap D = \emptyset$, Constraint 1 imposes that S is formed by a single rider without a car, hence its cost is provided by $k(S)$. On the other hand, if S contains at least one driver, its coalitional value is provided by the sum of the following negative[3] cost functions:

- $t : \mathcal{L} \to \mathbb{R}^-$, i.e., the time cost of driving through a given path,
- $c : \mathcal{L} \to \mathbb{R}^-$, i.e., the cognitive cost[4] of driving through a given path,
- $f : \mathcal{L} \to \mathbb{R}^-$, i.e., the fuel cost of driving through a given path,

We assume that such functions are *additive*, i.e., they fulfil the following definition.

Definition 6.5 (additivity). *A function $z : \mathcal{L} \to \mathbb{R}^-$ is said to be* additive *if, given two paths $L_1, L_2 \in \mathcal{L}$, then $z(L_1 \oplus L_2) = z(L_1) + z(L_1)$, where $\oplus$ represents the concatenation of paths.*

Notice that additivity trivially applies to any cost function in real-world ridesharing scenario. Finally, L_S^* represents the optimal path for S, defined as

$$L_S^* = \arg\max_{L \in \mathcal{VL}(S)} value(L). \tag{6.2}$$

Considering this, a SR problem can be easily translated into a GCCF problem, as each feasible set of riders is indeed a feasible coalition and $v(\cdot)$ provides its coalitional value. Hence, CS^* represents the optimal coalition structure which maximises the social welfare (i.e., minimises the total cost) for the system. However, the computation of the optimal path in Equation 6.2 represents a hard problem [72], which could be not solvable in realistic scenarios. Hence, in the next section we show how a reasonable assumption on the cost functions allows us to reduce this complexity making such computation tractable.

6.1.2 Optimal path computation

The computational complexity of Equation 6.2 derives from the size of its search space, formed by all the *valid* paths for S, i.e., all the paths in the graph M that contain the starting and destination points of the members of S in an order that satisfies Constraints 3 and 4. Notice that, given a particular sequence of starting and destination points that satisfies such constraints, Equation 6.2 requires to consider multiple valid paths, as the following example shows.

[3] Since we consider a maximisation problem, we represent costs as negative values.
[4] The fatigue incurred by the driver during the trip, see [57] for more details.

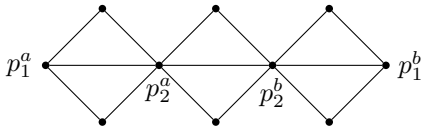


Fig. 6.3: Example start and destination points for 2 riders.

Figure 6.3 shows an example map containing the starting and destination points of 2 agents, in which only one sequence of points is valid, i.e., $L = \langle p_1^a, p_2^a, p_2^b, p_1^b \rangle$. Nonetheless, the set of valid paths is much larger (i.e., $3^3 = 27$ valid paths), since there exist 3 possible paths for each of the 3 couples of consecutive points in L. However, it is reasonable to assume that the driver will go through the shortest path for each of these 3 couples of points.

Assumption 1 *When the driver has to go from one point in L to the next one, it will choose the* shortest path *(considering the distances provided by $\lambda(\cdot)$) connecting such points.*

Assumption 1 allows us to collapse the search space of Equation 6.2 to $\mathcal{VT}(S)$, defined as follows.

Definition 6.6 ($\mathcal{VT}(S)$). *Given a feasible coalition S, $\mathcal{VT}(S)$ is the set of tuples that contain all and only the start and destination points of the members of S (without repetitions) and that satisfy Constraints 3 and 4.*

In order to explain how to simplify the solution of Equation 6.2 given the above assumption, we define the function $concat(\cdot)$.

Definition 6.7 ($concat(\cdot)$). *Given $L \in \mathcal{VT}(S)$, the function $concat : \mathcal{VT}(S) \to \mathcal{L}$ provides the path obtained as the concatenation of all the shortest paths between one point in L and the following one. Formally, $concat(L)$ is a tuple defined as*

$$concat(L) = \bigoplus_{k=1}^{|L|-1} sp(L[k], L[k+1]),$$

where the function $sp : P \times P \to \mathbb{R}^+$ provides the shortest path between two points, considering the length provided by the $\lambda(\cdot)$ function.

The function $concat(\cdot)$ can be computed in $O((|L|-1) \cdot (|Q| + |P| \cdot \log |P|))$, assuming that $sp(\cdot)$ is implemented using Dijkstra's algorithm [38]. Moreover, if M is an euclidean graph, $concat(\cdot)$ can be computed in $O((n-1) \cdot |Q|)$ with the A* algorithm [51]. Against this background, Equation 6.2 can be rewritten as

$$L_S^* = \underset{L \in \mathcal{VT}(S)}{\arg\max}\ value(concat(L)). \tag{6.3}$$

Notice that the search space of Equation 6.3 is $\mathcal{VT}(S)$, which is significantly smaller than $\mathcal{VL}(S)$ in Equation 6.2, and although being still exponential with respect to $|S|$, such computational complexity is manageable for reasonably sized groups of riders. In fact, $\mathcal{VT}(S)$ contains only 2520 valid tuples for $|S| = 5$ (i.e.,

the number of seats of an average car). Such result allows us to evaluate each coalition S by means of Equation 6.1, and hence we can address the SR scenario as a GCCF problem.

Furthermore, on the basis of Assumption 1 we can formulate Theorem 6.18, the fundamental theoretical result that allows us to compute an upper bound for the SR characteristic function. As a consequence, we can employ the CFSS algorithm presented in Section 4.4 to solve the SR problem.

6.2 CFSS for the SR problem

In order to solve the SR problem, the original version of CFSS [13, 14] must be modified to assess the additional constraints introduced in Section 6.1. In particular, to ensure that Constraint 1 and optionally Constraint 2 hold, we must avoid the formation of coalitions which are not feasible sets of riders. This is achieved by preventing the contractions of the green edges that would result in the violation of such constraints. Notice that such edges must be marked in red, even if we are not visiting the corresponding subtrees: in fact, this is equivalent to traversing such search spaces and discarding any possible solution they may contain, because such solutions would violate one of the above mentioned constraints.

A key enhancement for the efficiency of CFSS is the use of a branch and bound search strategy to prune significant parts of the search space, enabling a general bounding technique for $m + a$ functions (Theorem 4.17). However, the characteristic function defined in Equation 6.1 is not $m + a$, since it depends on L_S^*, and in particular on the actual position of the start and destination points of the riders.

As an example, consider Figure 6.4, which shows the start and destination points for 3 riders, i.e., $A = \{a_1, a_2, a_3\}$, in which only a_1 owns a car, i.e., $D = \{a_1\}$. For simplicity, we assume that $v(S)$ is equal to the length of L_S^*, and $k(\{a_2\}) = k(\{a_3\}) = -1$. In this example, $v(\{a_1\}) = -3$, $v(\{a_2\}) = -1$, $v(\{a_3\}) = -1$. However, we notice that p_2^a and p_2^b are actually part of the path travelled by a_1, hence it is reasonable for a_2 to join a_1 in the coalition $\{a_1, a_2\}$. In fact, $v(\{a_1, a_2\}) = v(\{a_1\}) = -3 > v(\{a_1\}) + v(\{a_2\}) = -3 - 1 = -4$. The optimal path for $S = \{a_1, a_3\}$ is $L_S^* = \langle p_1^a, p_3^a, p_3^b, p_1^b \rangle$. On the other hand, a_3 start and destination points are outside a_1's path,hence ridesharing is not effective in this case: $v(\{a_1, a_3\}) = -7 < v(\{a_1\}) + v(\{a_3\}) = -3 - 1 = -4$. Notice that this particular characteristic function cannot be seen as the sum of a superadditive and a subadditive part, since it exhibits a superadditive behaviour for some coalition structures, i.e., $v(\{a_1, a_2\}) > v(\{a_1\}) + v(\{a_2\})$, while it is subadditive for some others, i.e., $v(\{a_1, a_3\}) < v(\{a_1\}) + v(\{a_3\})$.

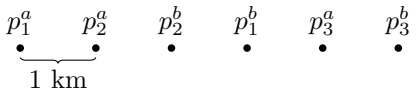


Fig. 6.4: Example start and destination points for 3 riders.

Hence, in the next section we provide alternative bounding techniques that can be used in our ridesharing scenario.

6.2.1 Bound computation

Given a feasible coalition structure CS in our search tree, we now show how to compute an upper bound $M(CS)$ for the values assumed by the characteristic function in $ST(CS)$, i.e., $M(CS) \geq V(CS_i)\ \forall CS_i \in ST(CS)$. We use this value to avoid visiting $ST(CS)$ if $M(CS)$ is not greater than the current best solution. This allows us to realise the same pruning technique discussed in Section 4.4 in the context of $m + a$ functions.

First, we provide a method to compute $M(CS)$ in scenarios where Constraint 2 holds. In these environments it is not possible to merge coalitions both containing a driver, since only single riders not owning a car are allowed to join existing groups. The addition of a rider to a feasible coalition S can only result in a greater cost, as shown by the following lemma.

Lemma 6.8. *Given two feasible coalitions S and S' such that $S' = S \cup \{a_i\}$ and $a_i \notin D$, then $v(S) \geq v(S')$.*

Proof. By contradiction. Suppose $v(S) < v(S')$, i.e., $value(L^*_S) < value(L^*_{S'})$, since both S and S' contain one driver. Notice that such driver, namely a_j, is the same for both S and S', by definition of S'. Now, $L^*_{S'}$ is a valid path also for S, since $L^*_{S'}$ starts at p^a_j and ends at p^b_j (satisfying Constraint 3), and Constraint 4 is trivially verified since $S \subset S'$. This violates Equation 6.2, since $v(S) = value(L^*_S)$ does *not* consider the optimal path for S, producing a contradiction. ↯

Therefore, the sum of the costs of all the cars (i.e., all the coalitions containing a driver) can only increase after such an addition. The above lemma allows us to prove Theorem 6.10. For convenience, we first make the following definition.

Definition 6.9 ($A_d(CS)$). *Given a coalition structure CS, $A_d(CS)$ is the set of coalitions in CS that contain at least one driver, i.e., the set of cars. Formally,*

$$A_d(CS) = \{S \in CS \mid S \cap D \neq \emptyset\}.$$

Theorem 6.10. *If Constraint 2 holds, for any feasible coalition structure CS*

$$M_1(CS) = \sum_{S \in A_d(CS)} v(S) \tag{6.4}$$

is an upper bound for the value of any CS' in $ST(CS)$, i.e., the subtree rooted in CS. Formally, $M_1(CS) \geq V(CS')$ for all $CS' \in ST(CS)$.

Proof. By contradiction. Suppose there exists a coalition structure $CS' \in ST(CS)$ such that $V(CS') > M_1(CS)$, i.e., CS' results in a cost *lower*[3] than $M_1(CS)$. Now, since $CS' \in ST(CS)$ and Constraint 2 holds, CS' must have been formed by adding single riders to already formed cars in CS. All such cars correspond to coalitions whose values are lower than the original ones in CS (Lemma 6.8). This contradicts $V(CS') > M_1(CS)$. ↯

We now discuss how to compute an upper bound without assuming Constraint 2. For convenience, we first define some theoretical concepts.

Definition 6.11 (P_{ab}). *P_{ab} is the set of the start and destination points of all the riders, i.e.,*

$$P_{ab} = \left\{ p \in P \;\middle|\; \exists a_i \in A : p = p_i^a \text{ or } p = p_i^b \right\}.$$

Definition 6.12 ($P_{couples}$). *$P_{couples}$ is the set of all the couples of different points in P_{ab}, i.e.,*

$$P_{couples} = \{(p, q) \in P_{ab} \times P_{ab} \mid p \neq q\}.$$

Definition 6.13 ($P_{1,a}(a_i)$ and $P_{1,b}(a_i)$). *Given a rider $a_i \in A$, $P_{1,a}(a_i)$ is the set of all the shortest paths from a_i's starting point to the start and destination points of any other rider, i.e.,*

$$P_{1,a} = \{L \mid L = sp(p_i^a, p)\ \forall p \in P_{ab} : p \neq p_i^a\}.$$

Similarly, we define $P_{1,b}(a_i)$ considering a_i's destination point.

Definition 6.14 ($P_{2,a}(a_i)$ and $P_{2,b}(a_i)$). *Given a rider $a_i \in A$, $P_{2,a}(a_i)$ is the concatenation of all the couples of shortest paths from a_i's starting point to the start and destination points of any other rider, i.e.,*

$$\{L \mid L = sp(p_i^a, p) \oplus sp(p_i^a, q)\ \forall (p, q) \in P_{couples} : p \neq p_i^a \text{ and } q \neq p_i^a\}.$$

Similarly, we define $P_{2,b}(a_i)$ considering a_i's destination point.

Definition 6.15 ($m(\cdot)$). *The function $m : A \to \mathbb{R}^-$ is defined as*

$$m(a_i) = \begin{cases} \max\limits_{L \in P_{1,a}(a_i)} value(L) + \max\limits_{L \in P_{1,b}(a_i)} value(L), & \text{if } a_i \in D \quad (6.5) \\ \max\limits_{L \in P_{2,a}(a_i)} value(L) + \max\limits_{L \in P_{2,b}(a_i)} value(L), & \text{otherwise.} \quad (6.6) \end{cases}$$

Intuitively, the purpose of $m(a_i)$ is to provide an upper bound on the $value(\cdot)$ function corresponding to the edges incident on p_i^a and p_i^b, when such edges are part of a path L driven by a car. If a_i is a driver, such an upper bound is calculated by considering the best edges (i.e., the ones that maximise $value(\cdot)$) incident on each point (Equation 6.5). In contrast, if a_i is not a driver, Equation 6.6 considers the best couples of edges incident on each point. We now provide an example to better explain how the $m(\cdot)$ function is calculated.

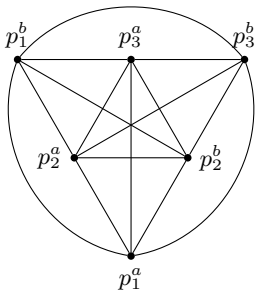


Fig. 6.5: Example start and destination points for 3 riders.

Figure 6.5 shows the start and destination points of 3 riders, in which the edges represent the shortest paths between any couple of points (under Assumption 1). Furthermore, assume that a car is formed among such agents, and that a_1 and a_2 are drivers, while a_3 is not. Notice that we do not know in advance whether a_1 or a_2 will be the optimal driver of such car. For the sake of brevity, in the following discussion we only refer to the agents' starting points, but the same concepts apply symmetrically to the destination ones. Notice that, since a_3 is not a driver, p_3^a will necessarily be an inner point in L, as a consequence of Constraint 3. It follows that L will contain exactly two edges incident on p_3^a. Now, since we are interested in computing an upper bound on $value(\cdot)$, we consider the couple of edges incident on p_3^a that maximises such function (Equation 6.6).

On the other hand, since we do not know in advance if a_1 (resp. a_2) will be the optimal driver of the car, we cannot predict whether p_1^a (resp. p_2^a) will be the first point or an inner point in L. In other words, we do not know exactly whether one or two edges incident on p_1^a (resp. p_2^a) will be part of L. Therefore, in Equation 6.5 we assume that only one edge is present in L, as a conservative measure. This is guaranteed to provide an upper bound on $value(\cdot)$, as such function is negative and, hence, the value of the best couple of edges is lower than the value of the best single edge. We now define the function $M_2(\cdot)$ on the basis of $m(\cdot)$.

Definition 6.16 ($M_2(\cdot)$). *The function $M_2 : \mathcal{CS}(G) \to \mathbb{R}^-$ is defined as*

$$M_2(CS) = \frac{1}{2} \cdot \sum_{a_i \in U_d(CS)} m(a_i), \quad \text{where } U_d(CS) = \bigcup_{S \in A_d(CS)} S. \tag{6.7}$$

Intuitively, $U_d(CS)$ is the set of all agents (both riders and drivers) that are passengers of a car in CS.

We now prove the following lemma, that will support the proof of Theorem 6.18.

Lemma 6.17. *Given a feasible coalition structure CS and a coalition structure $CS' \in ST(CS)$ such that $V(CS') > M_2(CS)$, then*

$$\exists S' \in A_d(CS') : v(S') > \frac{1}{2} \cdot \sum_{a_i \in S'} m(a_i). \tag{6.8}$$

Proof. By contradiction. First notice that

$$V(CS') = V(A_d(CS')) + V(CS' \setminus A_d(CS')),$$

i.e., $V(CS')$ is the sum of the values of all the cars in CS' plus the values of the singletons of riders that are not drivers. From $V(CS') > M_2(CS)$, it follows that

$$V(A_d(CS')) + V(CS' \setminus A_d(CS')) > \frac{1}{2} \cdot \sum_{a_i \in U_d(CS)} m(a_i).$$

Since $V(CS' \setminus A_d(CS')) = \sum_{S \in A_d(CS')} k(S) \leq 0$ (Equation 6.1), it follows that

$$V(A_d(CS')) > \frac{1}{2} \cdot \sum_{a_i \in U_d(CS)} m(a_i).$$

Since we only merge coalitions in the formation of new coalition structures in $ST(CS)$, it is impossible that a rider exits a car, i.e., $U_d(CS) \subseteq U_d(CS')$. Moreover, since the function $m(\cdot)$ is negative (see Definition 6.15), it is also true that

$$V(A_d(CS')) > \frac{1}{2} \cdot \sum_{a_i \in U_d(CS')} m(a_i). \tag{6.9}$$

Now, suppose that (6.8) is not true, i.e.,

$$v(S') \leq \frac{1}{2} \cdot \sum_{a_i \in S'} m(a_i), \quad \forall S' \in A_d(CS').$$

If we apply such property to all the coalitions S' considered in the summation $\sum_{S' \in A_d(CS')} v(S') = V(CS')$, we obtain

$$V(A_d(CS')) \leq \frac{1}{2} \cdot \sum_{a_i \in U_d(CS')} m(a_i),$$

which contradicts (6.9). $\lightning$

Building upon such lemma, we now prove the following theorem. Notice that, as previously introduced, such theorem is based on the validity of Assumption 1, which is also the key concept that allows us to compute the optimal path for a given coalition thanks to Equation 6.3 in a feasible amount of time.

Theorem 6.18. *If Assumption 1 holds, for any feasible coalition structure CS $M_2(CS)$ is an upper bound for the value of any CS' in $ST(CS)$, i.e., the subtree rooted in CS. Formally, $M_2(CS) \geq V(CS')$ for all $CS' \in ST(CS)$.*

Proof. By contradiction. Suppose there exists a coalition structure $CS' \in ST(CS)$ such that $V(CS') > M_2(CS)$. By applying Lemma 6.17, there exists $S' \in A_d(CS')$ such that $v(S') > \frac{1}{2} \cdot \sum_{a_i \in S'} m(a_i)$. Since Assumption 1 holds, it follows that

$$value(concat(L^*_{S'})) > \frac{1}{2} \cdot \sum_{a_i \in S'} m(a_i), \tag{6.10}$$

for some $L^*_{S'} \in \mathcal{VT}(S')$. Now, $value(\cdot)$ is additive (Definition 6.5), thus it can be seen as the sum of the costs of all the subpaths that form $concat(L^*_{S'})$. Formally,

$$value(concat(L^*_{S'})) = \sum_{k=1}^{|L^*_{S'}|-1} value(sp(L^*_{S'}[k], L^*_{S'}[k+1])). \tag{6.11}$$

By combining (6.10) and (6.11) we obtain

$$\sum_{k=1}^{|L^*_{S'}|-1} value(sp(L^*_{S'}[k], L^*_{S'}[k+1])) > \frac{1}{2} \cdot \sum_{a_i \in S'} m(a_i). \tag{6.12}$$

Now, it is easy to see that the cost provided by $\sum_{a_i \in S'} m(a_i)$ cannot be higher than *twice*[5] the cost of any valid path that goes through the start and destination points of the members of S'. It follows that $\frac{1}{2} \cdot \sum_{a_i \in S'} m(a_i)$ cannot be *lower* than the corresponding $value(\cdot)$ for any of such valid paths, since we consider negative cost functions. This contradicts (6.12). $\lightning$

Theorems 6.10 and 6.18 allows to compute an upper bound on $V(\cdot)$ for all the coalition structures in $ST(CS)$. As a consequence, we can solve the SR problem by adopting a branch and bound approach based on CFSS (Algorithm 9), i.e., SR-CFSS (Algorithm 11). In addition to the different bounding technique, SR-CFSS differs from CFSS as it includes the constraints deriving from the SR model, i.e., Constraint 1 and, optionally, Constraint 2. Specifically, this is achieved by including an additional check (i.e., line 4 in the SR-CHILDREN routine) that discards the solutions that violate such constraints (cf. Algorithm 8). Notice that SR-CFSS derives all the anytime approximate properties of CFSS (see Section 4.4.2), since we can apply the technique in Equation 4.2 using $M_1(\cdot)$ or $M_2(\cdot)$ respectively defined in Equations 6.4 and 6.7. The model defined in Section 6.1 takes into account only the spatial aspect of the SR problem. In what follows, we show how to incorporate the time preferences of the commuters in our model and algorithms.

Algorithm 11 SR-CFSS(G)

1: $\mathcal{G}_c \leftarrow G$ with all green edges
2: $best \leftarrow \mathcal{G}_c$ {Initialise current best solution with singletons}
3: $Front \leftarrow \emptyset$ {Initialise search frontier $Front$ with an empty stack}
4: $Front$.PUSH$(\mathcal{G}_c)$ {Push $\mathcal{G}_c$ as the first node to visit}
5: **while** $Front \neq \emptyset$ **do** {Branch and bound loop}
6: $node \leftarrow Front$.POP$()$ {Get current node}
7: **if** $M_2(node) > V(best)$ **then** {Can also use $M_1(node)$ with Constraint 2}
8: **if** $V(node) > V(best)$ **then** {Check function value}
9: $best \leftarrow node$ {Update current best solution}
10: $Front$.PUSH(SR-CHILDREN$(node)$) {Update $Front$}
11: **return** $best$ {Return optimal solution}

Algorithm 12 SR-CHILDREN($\mathcal{G}_c$)

1: $\mathcal{G}'_c \leftarrow \mathcal{G}_c = (\mathcal{A}, \mathcal{E}, colour)$ {Initialise graph G' with $\mathcal{G}_c$}
2: $Ch \leftarrow \emptyset$ {Initialise the set of children}
3: **for all** $e \in \mathcal{E} : colour(e) = green$ **do** {For all green edges}
4: **if** GREENEDGECONTR$(\mathcal{G}'_c, e)$ meets Constraint 1 (and Constraint 2) **then**
5: $Ch \leftarrow Ch \cup \{$GREENEDGECONTR$(\mathcal{G}'_c, e)\}$
6: Mark edge e with colour red in $\mathcal{G}'_c$
7: **return** Ch {Return the set of children}

[5] If we sum all the values of the couples of edges incident to the points that form a given path, we consider each edge twice.

6.3 Introducing time constraints

In order to consider the riders' time preferences in the computation of the optimal coalition structure, we assume that each rider $a_i \in A$ specifies its desired departure time τ_i^a within the interval $\theta_i^a = [\tau_i^a - \alpha_i^a, \tau_i^a + \beta_i^a] \subseteq \mathbb{N}^+$.[6] Similarly, a_i expresses its preferences on the arriving time τ_i^b by means of the interval $\theta_i^b = [\tau_i^b - \alpha_i^b, \tau_i^b + \beta_i^b] \subseteq \mathbb{N}^+$. Figure 6.6 shows an example of departure and arriving time constraints.

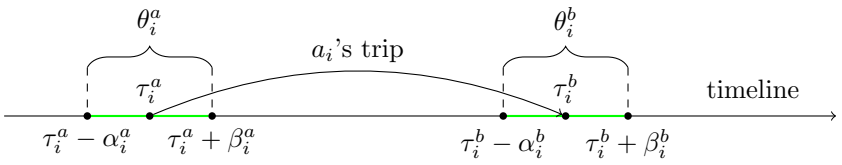


Fig. 6.6: Example of departure and arriving time constraints.

Then, we include these preferences modifying the characteristic function definition in Equation 6.1 as

$$v(S) = \begin{cases} k(S) & \text{if } S \cap D = \emptyset, \\ \underbrace{t(L_S^*) + c(L_S^*) + f(L_S^*) + \theta_S(L_S^*, \tau_S^*)}_{value\left(L_S^*, \tau_S^*\right)} & \text{otherwise.} \end{cases} \tag{6.13}$$

In particular, we introduce the term $\theta_S : \mathcal{VL}(S) \times \mathbb{N}^+ \to \mathbb{R}^-$ as a measure of the extent to which the time preferences of the members of S have been fulfilled by a trip starting at a given time and going through a given valid path. We quantify such an extent with a cost for each starting and destination point that is proportional to the difference between the actual pick-up/arriving time and the desired one, i.e.,

$$\theta_S(L, \tau) = \sum_{a_i \in S} \Delta_S^a(a_i, L, \tau) + \Delta_S^b(a_i, L, \tau), \tag{6.14}$$

where $\Delta_S^a : S \times \mathcal{VL}(S) \times \mathbb{N}^+ \to \mathbb{R}^-$ and $\Delta_S^b : S \times \mathcal{VL}(S) \times \mathbb{N}^+ \to \mathbb{R}^-$ represent such costs (for pick-up and destination points respectively) for each $a_i \in S$ i.e.,

$$\Delta_S^a(a_i, L, \tau) = \begin{cases} \gamma_1 \cdot |time_L(p_i^a, \tau) - \tau_i^a| & \text{if } time_L(p_i^a, \tau) \in \theta_i^a, \\ -\infty & \text{otherwise,} \end{cases} \tag{6.15}$$

$$\Delta_S^b(a_i, L, \tau) = \begin{cases} \gamma_2 \cdot |time_L(p_i^b, \tau) - \tau_i^b| & \text{if } time_L(p_i^b, \tau) \in \theta_i^b, \\ -\infty & \text{otherwise,} \end{cases} \tag{6.16}$$

where $\gamma_1, \gamma_2 \in \mathbb{R}^+$ are the costs associated to one time unit of delay/anticipation for pick-up and destination points respectively. Notice that, even if other formulations for θ_S are possible, the crucial feature is the enforcement of the hard constraint (i.e., $\theta_S = -\infty$) outside θ_i.

[6] We consider a *discrete* time domain, e.g., seconds or minutes.

The function $time_L : L \times \mathbb{N}^+ \rightarrow \mathbb{N}^+$ represents the arrival time at each of the points of L considering a given departure time, i.e.,

$$time_L\left(p, \tau\right) = \tau + \sum_{k=1}^{j-1} \delta\left(L[k], L[k+1]\right), \quad \text{where } L[j] = p \tag{6.17}$$

and the function $\delta : Q \rightarrow \mathbb{N}^+$ measures the travel time trough a given edge.

Equations 6.15 and 6.16 induce a series of hard constraints on the departure/arriving time for each rider $a_i \in S$, as each a_i is *not* willing to leave/arrive earlier that $\tau_i - \alpha_i$ nor later than $\tau_i + \beta_i$. Thus, we define $\theta_S = -\infty$ if any of these constraints is violated. If such set of time constraints is *not satisfiable*, we denote S as *time infeasible* (see Section 6.3.2). We define the optimal path L_S^* and the optimal departure time τ_S^* accordingly to Equation 6.2:

$$\left(L_S^*, \tau_S^*\right) = \underset{\substack{L \in \mathcal{VL}(S) \\ \tau \in \theta_j^a \forall a_j \in S \cap D}}{\arg\max} \; value\left(L, \tau\right). \tag{6.18}$$

We reduce the search space for L_S^* in Equation 6.18 by applying the same techniques discussed in Section 6.1.2 (i.e., by considering Assumption 1) and obtaining

$$\left(L_S^*, \tau_S^*\right) = \underset{\substack{L \in \mathcal{VT}(S) \\ \tau \in \theta_j^a \forall a_j \in S \cap D}}{\arg\max} \; value\left(concat\left(L\right), \tau\right). \tag{6.19}$$

Notice that, in Equation 6.19, the computation of τ_S^* is still carried out in a naïve way, going through every possible timestep in the time intervals specified by the drivers in S. In the following section, we explain a better approach to compute τ_S^*.

6.3.1 Optimal departure time computation

In this section we address the problem of computing the optimal departure time τ_S^* for a given coalition S. Problems involving time constraints arise in various areas of computer science, especially in scheduling contexts [33]. In particular, Dechter et al. [33] define the so-called *Simple Temporal Problems* (STP), a particular type of CSP (see Definition 2.8) in which a variable τ_i corresponds to a continuous time point and a binary constraint (τ_i, τ_j) is associated to one time range that contains the valid values for $\tau_j - \tau_i$. In the context of SR, if τ_1 and τ_2 are respectively the departure and the arrival time for a particular agent, the constraint (τ_1, τ_2) associated to the range $[0', 60']$ means that its arrival cannot happen more than 60 minutes after its departure. Khatib et al. [62] later extended the concept of STP associating a function (i.e., a *preference*) to each constraint, in order to differentiate among valid solutions and select the one that best meets such preferences. The authors also characterise the complexity of solving such problem (denoted as STPP) as NP-Complete in the general case, while it is tractable if preferences are expressed by linear functions. Such complexity results by virtue of the fact that STPPs with linear preferences can be expressed as Linear Programming (LP) problems, which can be solved in polynomial time [24]. However, even if our preferences are linear (Equations 6.15 and 6.16), our time domain is discrete, resulting in a problem of Integer LP, which is NP-Hard in the general case [24].

Nonetheless, our formalisation allows us to restrict such problem to a particular, tractable case. Specifically, our scenario requires to compute only the optimal departure time for the first point in the path, i.e., τ_S^*, since we assume no delay between the arrival to a point and the departure for the next point in the path (Equation 6.17).[7] Against this background, we now propose an algorithm to compute the best departure time for a car S (given a tuple $L \in \mathcal{VT}(S)$ and a driver $a_j \in S \cap D$), so to avoid trying every possible departure time for the trip of S. Algorithm 13 achieves this task by considering the ideal departure time of the driver, i.e., τ_j^a, and by applying a sequence of shifts so to obtain the optimal τ.

First (lines 1–7), we initialise τ with the ideal departure time of the driver, and we initialise p, n and z, which will respectively count the number of points in which we register an arriving time that is late, early or on time, with respect to the desire expressed by the agents for those points. The variables *post* and *antic* function as guards to check to what extent it is possible to delay/anticipate departure without violating any time constraint. Finally, we also define *diffs* which contains the difference between the actual and the ideal time, for every point in L. Lines 8–11 set these variables. After this, at line 12 we check whether it is possible to satisfy all the time constraints. If the conditional statement is true, then at least two points are outside of their interval, one is late and one is ahead of time, or it may be necessary to anticipate τ of a given amount, but such action would result in the violation of another constraint. In such cases we return a null solution. If no constraints are violated, we improve τ in the cycle at lines lines 14–31 so that $\sum_{i=1}^{|L|} |\mathit{diffs}[i]|$ is minimised and does not invalidate any constraint. Notice that, to achieve this result, it is enough to check the direction of the points of the path. On the one hand, if the majority of the points are positive (the car is late) then we will have to anticipate τ. On the other hand, if the majority of the points are negative (the car is early) then we will have to delay τ.

Once we have a method to compute the optimal τ given a tuple $L \in \mathcal{VT}(S)$ and a driver $a_j \in S \cap D$, we can finally compute the optimal path L_S^* and the optimal driver a_S^* by modifying Equation 6.19 in the following way:

$$(L_S^*, a_S^*) = \underset{\substack{L \in \mathcal{VT}(S) \\ a_j \in S \cap D}}{\arg\max}\, \mathit{value}\left(\mathit{concat}\left(L\right), \textsc{OptimalDepTime}\left(L, a_j\right)\right). \tag{6.20}$$

L_S^* and a_S^* are computed by selecting the best combination over the possible valid tuples in $\mathcal{VT}(S)$ and drivers in S. For each of these combinations, we consider the corresponding optimal departure time provided by Algorithm 13. If such algorithm returns a null solution, the corresponding $\mathit{value}(\cdot)$ is $-\infty$, and hence, that particular combination is discarded. Equation 6.20 inherently provides τ_S^*, which is the optimal departure time for the maximising L_S^* and a_S^*. Notice that, following the same discussion at the end of Section 6.1.2, the search space of Equation 6.20 is at most $2520 \cdot 5 = 12600$ combinations of valid tuples and drivers for $|S| = 5$, and thus, it can be exhausted with a manageable computational effort.

[7] Our model can be easily generalised by dropping such an assumption, hence allowing a delay between the arrival to each point and the departure for the next one. Notice that, even if such problem is still untractable in the general case due to the discretisation of the time domain, it can be transformed to a tractable LP problem by means of LP relaxation techniques [24]. This topic will be considered as future work.

Algorithm 13 OptimalDepTime(L, a_j)

```
1: τ ← τ_j^a {Initialise current best solution with driver's ideal departure time}
2: p ← 0 {Positive points counter (i.e., points where L is late)}
3: n ← 0 {Negative points counter (i.e., points where L is early)}
4: z ← 0 {Zero points counter (i.e., points where L is on time)}
5: post ← +∞ {Maximum delay without constraint violation}
6: antic ← −∞ {Maximum anticipation without constraint violation}
7: diffs ← ⟨{Differences among ideal times and actual times}
8: for all i ∈ {1, ..., |L|} do {For all tuple points}
9:    diffs[i] ← difference between time_L (i, τ) and ideal arriving time at L[i]
10:   Increment p or n or z based on the sign of diffs[i]
11:   Update post and antic
12: if post < antic then {Conflict between two constraints}
13:   return ∅
14: repeat
15:   shift ← 0
16:   if post < 0 then
17:      shift ← post
18:   else if antic > 0 then
19:      shift ← antic
20:   else if p > n + z and antic < 0 then {Majority of points are late}
21:      lwp ← lowest positive in diffs
22:      shift ← − min{lwp, −antic}
23:   else if n > p + z and post > 0 then {Majority of points are early}
24:      grn ← greatest negative in diffs
25:      shift ← min{−grn, post}
26:   if shift ≠ 0 then
27:      τ ← τ + shift
28:      Update diffs
29:      Recompute p and n and z
30:      Update antic and post
31: until shift = 0
32: return τ
```

As said above, for some coalitions it is impossible to satisfy all time constraints. Such coalitions are associated to $-\infty$ by Equations 6.15 and 6.16, and hence, they can never be part of the optimal solution. However, at the moment we detect such infeasibility only after the execution of Algorithm 13. In contrast, it would be desirable to identify such coalitions in advance and avoid their formation within SR-CFSS. By doing so, we could reduce the search space, hence improving the performance of our approach. We now show how we achieve this objective.

6.3.2 Time infeasible coalitions

In this section we propose an improved method to detect *time infeasible* coalitions. Intuitively, such coalitions are characterised by a set of time constraints that is not satisfiable, as formalised by the following definition.

Definition 6.19 (time infeasible coalition). *A feasible coalition S is said to be* time infeasible *if $\theta_S(L, \tau) = -\infty$ for all $L \in \mathcal{VT}(S)$ and all $\tau \in \theta_j^a \, \forall a_j \in S \cap D$.*

Since time infeasible coalitions can never be part of the optimal solution, we are interested in exploiting such property so to reduce the search space of the SR problem. One simple approach would be to check, for each solution CS computed during the traversal of the search tree, if CS contains a time infeasible coalition, and in such case, discard CS and the corresponding subtree $ST(CS)$.

Unfortunately, such a technique can lead to the exclusion of valid solutions, since a time infeasible coalition can become time feasible as a consequence of an edge contraction. We provide the following example to better explain this concept. Let $S = \{a_1, a_2\}$ with $a_1 \in D$ (i.e., a_1 is the driver), and let $\theta_1^a = [09{:}00 - 15', 09{:}00 + 15']$ and $\theta_2^a = [09{:}45 - 15', 09{:}45 + 15']$. Assuming that the path that joins p_1^a and p_2^a (i.e., the starting points of a_1 and a_2 respectively) corresponds to a travel time of 10 minutes, it is not possible to find a departure time τ such that a_1 does not arrive too early at p_2^a. Thus, θ_2^a will always be violated, and hence, S is time infeasible. Now, assume that, as the result of an edge contraction, $S' = S \cup \{a_3\}$ is formed, with $\theta_3^a = [09{:}20 - 15', 09{:}20 + 15']$. If the paths from p_1^a to p_3^a and from p_3^a to p_2^a both require 10 minutes, it is possible to satisfy the time constraints of all the members of S'. Hence, S' is no longer time infeasible.

Nevertheless, under certain conditions it is possible to identify a particular type of time infeasible coalitions that will always result in other time infeasible coalitions as a result of an edge contraction. Such coalitions can be safely discarded from $\mathcal{FC}(G)$, pruning a significant portion of the search space.

Proposition 6.20. *Let $a_i, a_j \in A$ with $a_i \in D$ and $a_j \notin D$. If we consider Constraint 2 (i.e., one driver per car) and $[\tau_j^a + \beta_j^a, \tau_j^b - \alpha_j^b] \not\subseteq [\tau_i^a - \alpha_i^a, \tau_i^b + \beta_i^b]$, then a_i and a_j can never be in a time feasible coalition together, i.e., $\forall S \in \mathcal{FC}(G) : \{a_i, a_j\} \subseteq S$, S is a time infeasible coalition.*

Proof. If $[\tau_j^a + \beta_i^a, \tau_j^b - \alpha_j^b] \not\subseteq [\tau_i^a - \alpha_i^a, \tau_i^b + \beta_i^b]$, then $\tau_j^a + \beta_i^a < \tau_i^a - \alpha_i^a$ or $\tau_j^b - \alpha_j^b > \tau_i^b + \beta_i^b$. Intuitively, a_j's latest departure time is earlier than a_i's earliest departure time or a_j's earliest arriving time is later than a_i's latest arriving time. Since we consider Constraint 2, a_i can be the only driver of any coalition containing both a_i and a_j. Thus, it is trivial to verify that the above time constraint will always be violated, since travelling back in time is not (yet) possible. □

If we consider a scenario that enforces Constraint 2, then Proposition 6.20 can be used to identify couples of agents (a_i, a_j) that can never be part of the same coalition, effectively introducing some additional hard constraints on the formation of coalitions. Such constraints can be easily expressed by marking each edge (a_i, a_j) (if existent) as red in the initial graph G, so to avoid the formation of a coalition in which a_i and a_j are together. On the other hand, if a_i and a_j are not connected by an edge in G, we *introduce* a new edge marked in red, since if we do not do so, then a_i and a_j will be part of the same coalition for at least one coalition structure in the search tree. As an example, consider Figure 6.7.

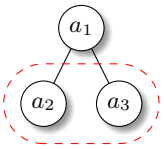


Fig. 6.7: Example of a social network with 3 agents.

Assume that a_2 and a_3's time constraints verify Proposition 6.20. If we do not introduce a new red edge between a_2 and a_3, the grand coalition will be evaluated during the traversal of the search tree, even if such coalition is guaranteed to be time infeasible. On the other hand, the introduction of such red edge avoids such inefficiency in our approach.

Against this background, we can exploit time constraints to restrict the formation of coalitions. Moreover, we can also employ the upper bound techniques discussed in Section 6.2.1, as we motivate in the following section.

6.3.3 Bound computation

The upper bound methods proposed in Section 6.2.1 can also be applied when we introduce time constraint, as shown by the following proposition.

Proposition 6.21. *Theorems 6.10 and 6.18 are valid even if we substitute the definition of* $v(S)$ *in Equation 6.1 with the definition in Equation 6.13.*

Proof. Given a coalition $S \in \mathcal{FC}(G)$, the value provided by $v(S)$ in Equation 6.1 is necessarily greater than the one provided by Equation 6.13, since the latter is equal to the former with the addition of $\theta_S(L_S^*, \tau_S^*)$, which is negative by definition. Notice that the $t(L_S^*) + c(L_S^*) + f(L_S^*)$ is exactly the same, since we make Assumption 1 in both cases, and we assess L_S^* in the same way. As a consequence, given a feasible coalition structure CS, $V(CS)$ is greater if we consider Equation 6.1 with respect to Equation 6.13. Therefore, since Theorems 6.10 and 6.18 provide upper bounds and are valid considering Equation 6.1, they are also valid with Equation 6.13. □

In what follows, we show how we test our approach to solve the SR problem.

6.4 Empirical evaluation

The main goals of the empirical analysis are the following:

1. To estimate the social welfare improvement when our SR model is employed.
2. To evaluate the performance of the optimal version of SR-CFSS in terms of runtime and scalability.
3. To evaluate the approximate performance and guarantees that SR-CFSS can provide when scaling to very large number of agents, i.e., up to 2000 agents.
4. To investigate the impact of time constraints on the above properties.

Since there are no publicly available datasets which include *both* spatial and social data for the same users, in our empirical evaluation we consider two separate real-world datasets and we superimpose the first on the second one. In particular, our map $M = (P, Q)$ is a realistic representation of the city of Beijing (Figure 6.8), with $|P| = 8330$ points and $|Q| = 13290$ edges, equivalent to an average resolution of a point every $\sim$10 meters. This map has been derived from the GeoLife dataset[8] [121] provided by Microsoft Research, which comprises 17621 trajectories with a total distance of about 1.2 million km, recorded by different GPS loggers with a variety of sampling rates. This pool of trajectories is also adopted to sample random paths used to provide the start and destination points of the riders.

Moreover, such a dataset also includes the timestamp of each trajectory, allowing us to create a distribution of the departure and arrival times (Figure 6.9), which is used to sample such parameters for each agent in all our experiments, unless otherwise stated (i.e., in all experiments considering time constraints except Section 6.4.2). As expected, this distribution exhibit two peaks, one in the morning from 7:00 to 9:00 and one in the evening from 17:00 to 19:00.

In each experiment, G is obtained from the same Twitter dataset discussed in Section 5.4. In our experimental evaluation there is no mapping between the trajectory data and the social graph, since they belong to independent projects.

In all our experiments, the default number of agents n is 50. We adopt a cost model that considers fuel expenses, i.e., $v(C) = K_{fuel} \cdot \overline{L_C^*}$, where $\overline{L_C^*}$ represents the length of L_C^* in km, $K_{fuel} = -0.0\overline{6}$ €/km (considering a fuel cost of -1 € per litre and an average consumption of 1 litre of fuel every 15 km) and $k(\{a_i\}) = -3$ € $\forall a_i \in A$, which represents the average public transportation cost, i.e., a bus or a train ticket. Moreover, we assume that each car has a capacity of 5 seats, i.e., $seats(a_i) = 5\,\forall a_i \in D$. When time constraints are considered, we define $\gamma = -2$ €/h and a time interval (i.e., the duration of θ_i) of 30', unless otherwise stated.

Fig. 6.8: The map of Beijing derived from the GeoLife dataset.

[8] Available at `http://research.microsoft.com/en-us/projects/geolife`.

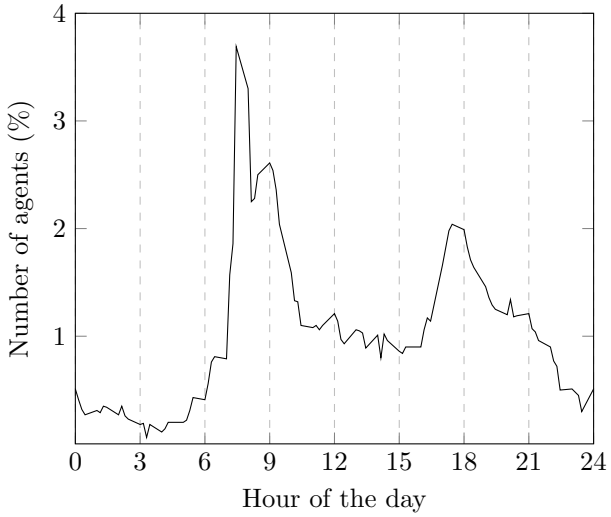


Fig. 6.9: Default distribution of departure/arrival times (obtained from GeoLife).

All our test are done considering Constraint 2 (drivers always drive their cars), as it models many real-world online services, e.g., *Lyft* and *Uber*. Hence, we employ both the bounding techniques detailed in Section 6.2.1 and we take the minimum one, since both are valid. Each experiment is repeated on 20 random instances, and we report the average and the standard error of the mean of the results. Our approach is implemented in C[9] and executed on a machine with a quad-core 3.40GHz processor and 16 GB of memory.

6.4.1 Social welfare improvement without time constraints

In our first experiment we consider the improvement of the social welfare (i.e., the cost reduction for the overall system) when using our SR model without time constraints, compared to the scenario in which every rider adopts its own conveyance (i.e., no ridesharing). This gives an indication of what gain can be achieved by the overall community when using our system for ridesharing. Formally, we define the social welfare improvement as $100 \cdot \left| \frac{V(CS^*) - V(A_{single})}{V(A_{single})} \right|$. Such an improvement is influenced by the percentage of drivers in the system (Figure 6.10), which determines the number of available seats and the number of riders which can share a ride without having to resort to public transport. Moreover, with more drivers it is more probable that a rider can join a car whose path is closer to him/her. On the other hand, if the majority of the riders own a car (i.e., $> 80\%$), ridesharing is not very effective since too few riders without a car can benefit from sharing their commutes with a driver. In particular, when only the 10% of the total riders own a car, the average cost reduction is -23.49%, reaching -36.22% when half of the riders owns a car.

[9] Our implementation is available at `https://github.com/filippobistaffa/SR-CFSS`.

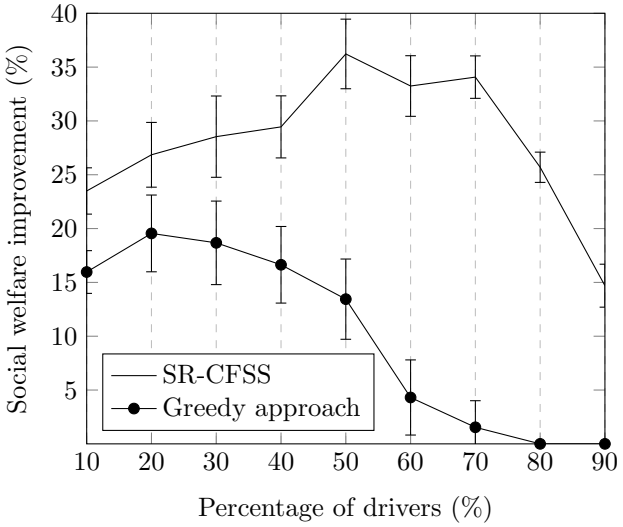


Fig. 6.10: Social welfare improvement.

To show the importance of an optimal approach, we benchmark our algorithm against a greedy one, in which every driver chooses its next stop as the closest among the destinations points of his current passengers and the starting points of the remaining riders. This choice is made considering the constraints imposed by the social network, avoiding the formation of unfeasible coalitions. As Figure 6.10 shows, our method allows superior cost reductions with respect to such a greedy approach, which can provide a maximum improvement of -19.55% for $|D| = 20\%$. Notice that, when the majority of the riders owns a car, the greedy approach cannot improve upon the value of the baseline solution (i.e., the one with no ridesharing).

6.4.2 Social welfare improvement with time constraints

We now investigate how the social welfare improvement varies when we introduce time constraints. Specifically, we now study the influence of the duration of the θ_i interval (i.e., the difference between the latest and the earliest departure/arrival time) and the distribution of the agents' departure times on the social welfare.

To evaluate such influence, we vary these two parameters as follows. On the one hand, we sample the departure times of the agents within a time window of 6 hours according to 3 probability distributions (Figure 6.11). Specifically, we consider a uniform distribution (i.e., the departure times are distributed uniformly in the time window) and two Gaussian distributions, in which the agents who desire to leave in the two central hours of the time window are respectively the 30% (soft peak) and the 40% (hard peak) of the total. On the other hand, we vary the duration of each θ_i by varying the difference between the earliest/latest tolerated time and the ideal time for each agent. For simplicity, we assume that $\alpha_i^a = \beta_i^a = \alpha_i^b = \beta_i^b$ are all equal for all agents, and we vary such value, namely θ_i's radius, within $[5', 60']$.

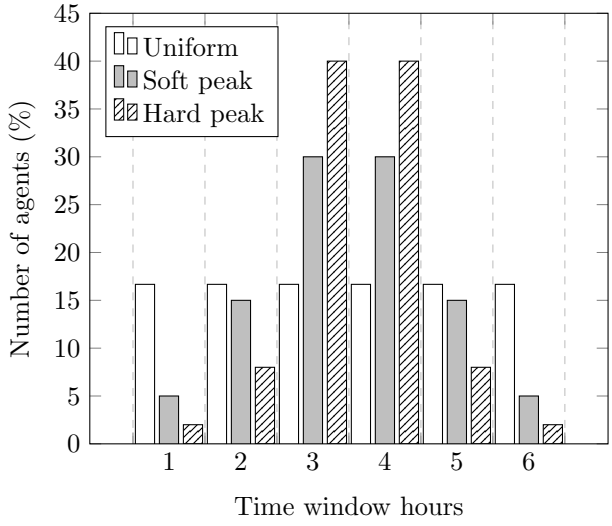


Fig. 6.11: Probability distributions in a time window of 6 hours.

Following the result of the experiments in the previous section, we only consider $D = 50\%$, i.e., the scenario that results in the highest social welfare improvement. Figure 6.12 shows that, in general, the social welfare improvement increases when we increase the θ_i's radius, since it is easier to form coalitions, and consequently, to reduce the overall travel cost, if agents are more tolerant with respect to time constraints. Notice that such an improvement saturates when the radius exceeds 30 minutes, since larger θ_i's radiuses are associated to larger costs by the θ_C component, which contributes to reduce the social welfare improvement. In addition, Figure 6.12 also shows that the hard peak distribution provides the highest social welfare improvement (8.79%) with respect to the soft peak (6.62%) and the uniform (3.62%) ones. In fact, if the departure times of more agents are concentrated in a shorter time period, the cost provided by the θ_C component is lower. Moreover, SR-CFSS can evaluate a larger amount of feasible solutions, since less time infeasible coalition structures have to be discarded. Finally, these results show that the introduction of time constraints leads to a reduction of the social welfare improvement, which is a consequence of the additional costs and, more important, a reduced solution space, as further confirmed by the experiments in Section 6.4.4.

We further investigate the above discussed experiment by studying the effect on the social welfare improvement of increasing the θ_i's radiuses only of a particular class of commuters. By doing so, we aim at identifying which classes are more sensitive to the variation of such parameter in terms of overall cost reduction. Specifically, we observe 3 interesting classes of agents, i.e., drivers, riders, and hubs (i.e., agents whose connectivity in the social graph is significantly above the average), and we vary the θ_i's radius within $[15', 60']$ only for the considered class, while setting such parameter equal to $15'$ for the other classes. Figure 6.13 shows that the social welfare improvement has the biggest increase for the drivers (+6.28%), reaching a final maximum of 14.24%. Such increase is slightly lower for

hubs (+5.1%), while it is only +1.27% for riders. These results prove the significant impact of a larger θ_i's radius for drivers and hubs, which results in a larger number of potential coalitions, and, consequently, a larger social welfare improvement.

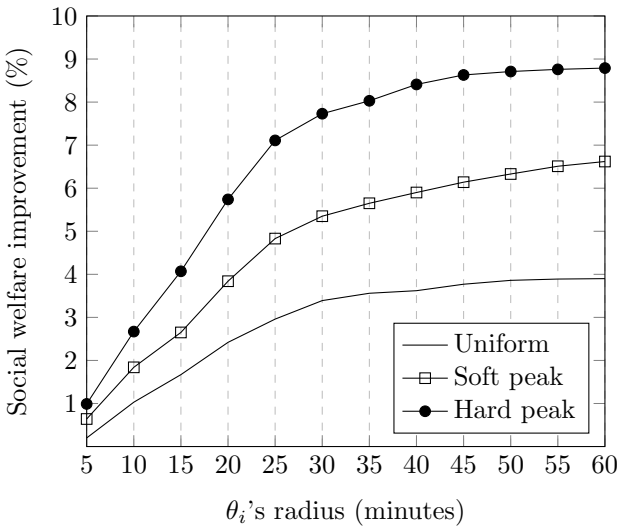


Fig. 6.12: Social welfare improvement with respect to θ_i's radius.

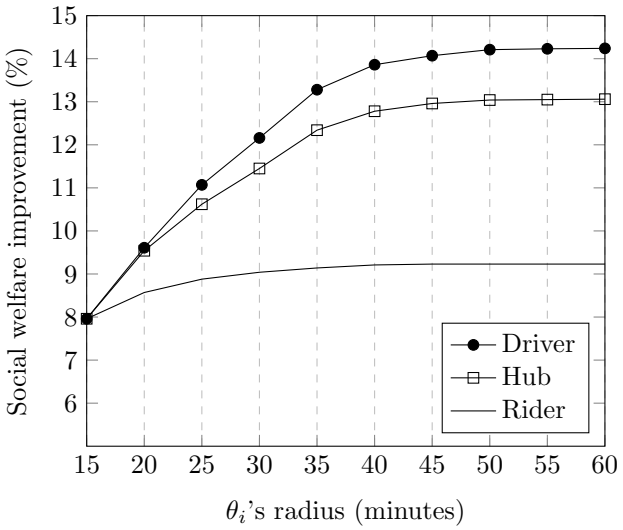


Fig. 6.13: Social welfare improvement with respect to θ_i's radius.

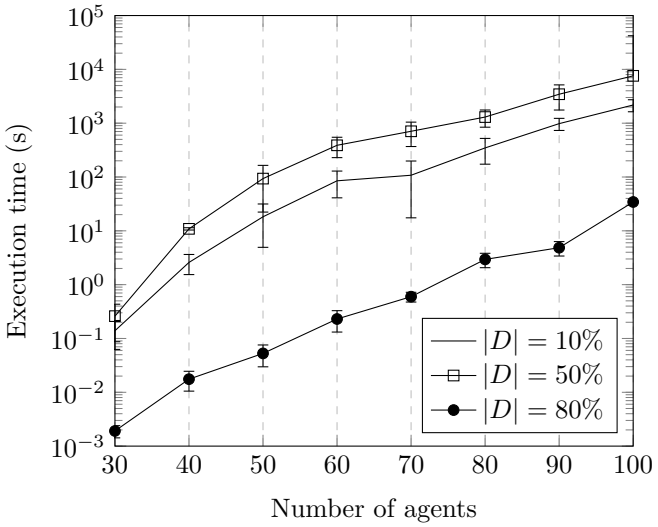


Fig. 6.14: Runtime without time constraints.

6.4.3 Runtime performance without time constraints

In this section we discuss the performance of our approach in terms of runtime needed to compute the optimal solution of a SR problem without time constraints. Figure 6.14 shows the runtime with respect to the number of agents adopting our SR model without time constraints. Our approach is tested in 3 scenarios, i.e., with low (10%), medium (50%) and high (80%) percentage of drivers, showing that this parameter has a significant influence on the performance of our algorithm. In fact, the size of the search space is determined by the number of available seats (reduced when such a percentage is low) and the number of riders without a car who can benefit from sharing their commutes (reduced when the majority of the agents owns a car), consistently with the behaviour of the social welfare improvement detailed in the previous section. Notice that, in any case, our approach can solve systems with 100 agents in a reasonable amount of time, i.e., about 2 hours at most for $|D| = 50\%$. This runtime is suitable for services with day-ahead or week-ahead requests (e.g., Lyft). Such a performance is possible thanks to our bounding techniques (see Section 6.2.1), which allow to prune a significant part of the search space. In more detail, such techniques allow an average pruning of the 97.5% of the search space (resulting in an average runtime improvement of about 4 hours) on 20 random instances with $n = 60$ and $|D| = 50\%$.

6.4.4 Runtime performance with time constraints

When we consider time constraints (Figure 6.15), we notice a significant performance improvement of SR-CFSS, which can compute the optimal solution for 100 agents in 30 seconds, i.e., over two orders of magnitude faster than the above case. This increased performance also results in an increased scalability, as SR-CFSS can

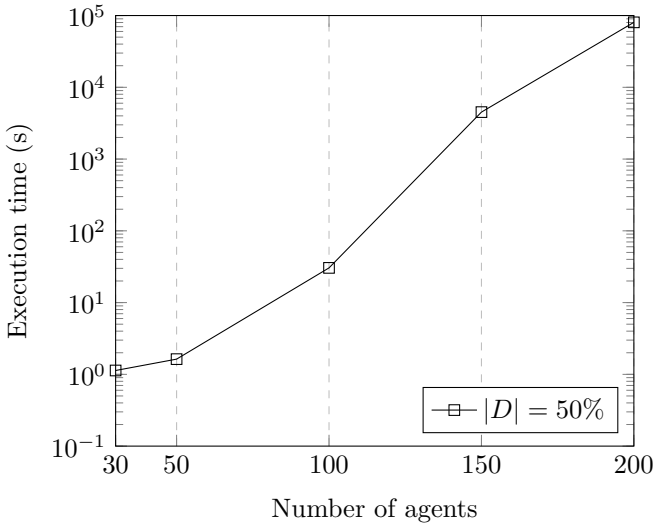


Fig. 6.15: Runtime with time constraints.

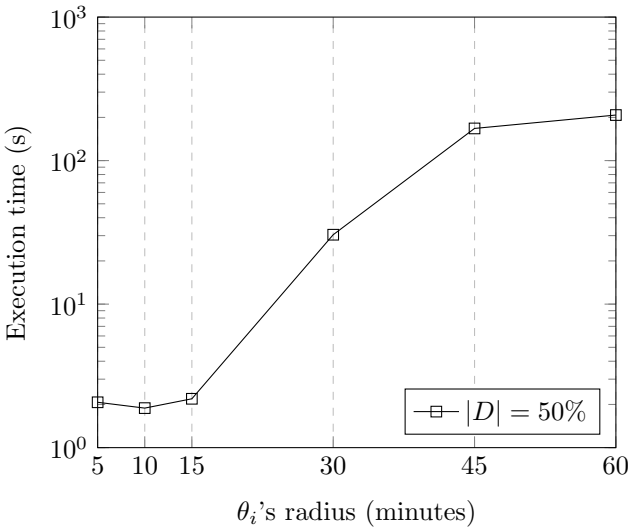


Fig. 6.16: Runtime with respect to θ_i's radius.

solve systems with 150 agents, i.e., 50 additional agents with respect to 100 agents in the previous experiment, in the same amount of time, and 200 in less than a day. We further investigate the impact of time constraints on the performance of SR-CFSS by varying the θ_i's radius, as discussed in Section 6.4.2. Figure 6.16 shows that larger radiuses correspond to harder SR problems. As an example, instances with a θ_i's radius equal to 15 minutes are solved by SR-CFSS more than two orders of magnitude faster with respect to when we consider 45 minutes. As discussed in

Section 6.4.2, larger radiuses correspond to a larger number of feasible solutions, since less time infeasible coalition structures have to be discarded. These results further confirm the significant impact of time constraints on the dimension of the solution space, which results in a twofold consequence. On the one hand, scenarios with time constraints are easier to solve, since the amount of feasible solutions is lower. On the other hand, the reduced amount of possible solutions allows a lower social welfare improvement in such scenarios (see Section 6.4.2).

6.4.5 Approximate performance

Here we benchmark the performance of our SR-CFSS on large-scale systems with thousands of agents. We adopt the same methodology discussed in Section 5.4.4, i.e., we study the maximum performance ratio (see Definition 5.6) as a measure of the quality guarantees of the approximate solution computed by our approach. Specifically, we run SR-CFSS on instances adopting the model without time constraints with $n \in \{500, 1000, 2000\}$ and we stop the execution after a time budget of 100 seconds. Then, we compute $Bound(I)$ with the method in Section 4.4.2 by applying the upper bounds defined in Theorems 6.10 and 6.18.

Figure 6.17 shows that, on average, $Bound(I)$ is only 6.65% higher than $Approx(I)$ (i.e., the solution found within the time limit) for $n = 500$ and $|D| = 80\%$, reaching a maximum of +29.92% when $n = 2000$ and $|D| = 50\%$. In the worst case, SR-CFSS provides a maximum performance ratio of 1.41 and thus solutions whose values are at least 71% of the optimal. We obtained very similar results also when we consider time constraints, and hence, we do not report them here. Such a behaviour is reasonable since the maximum performance ratio is heavily influenced by the value of $Bound(I)$ and, as detailed in Section 6.3.3, we apply the same technique whether or not we consider time constraints.

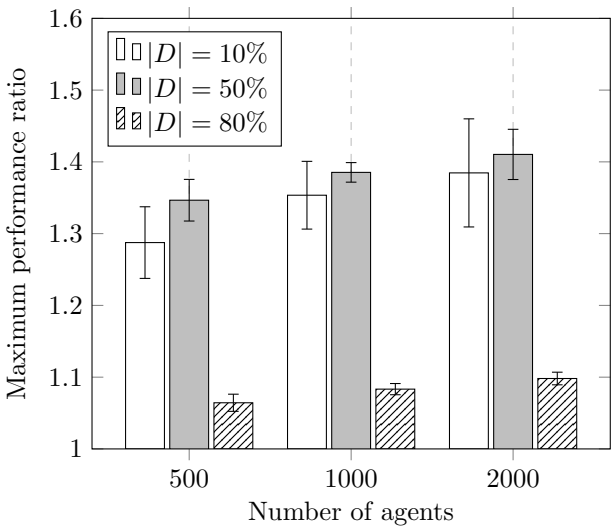


Fig. 6.17: Maximum performance ratio of approximate solutions.

6.4.6 SR-CFSS vs. C-Link: solution quality comparison

In our final experiment we further evaluate the approximate performance of SR-CFSS by comparing it against C-Link [42]. Specifically, we generate random SR instances with $n \in \{1000, 1200, \ldots, 2000\}$, considering 20 repetitions for each n. Then, following Section 5.4.5, we solve each instance with C-Link (adopting the best heuristic proposed by Farinelli et al. [42], i.e., Gain-Link) and then we run SR-CFSS on the same instance with a time budget equal to C-Link's runtime.

Figure 6.18 shows the average and the standard error of the mean of the ratio between the value of the solution computed by C-Link and the one computed by SR-CFSS. Since we consider solutions with negative values, when such ratio is > 1 the solution computed by C-Link is better (i.e., corresponds to a lower cost) than the one computed by SR-CFSS. Our results show that, for $n < 1600$, the quality of C-Link's solutions is better than SR-CFSS. Then, for $n \geq 1600$ SR-CFSS outperforms C-Link in terms of solution quality. In particular, for $n = 2000$ the solutions provided by our approach correspond to costs that are, on average, $2.28\times$ lower than the counterpart ones.

This behaviour is in contrast with the results obtained in the collective energy purchasing scenario, in which C-Link performed slightly better than CFSS even on the largest instances (see Section 5.4.5). This difference is due to the additional constraints (i.e., Constraint 1 and, optionally, Constraint 2) inherent in the SR scenario with respect to the collective energy purchasing one. Since C-Link is a greedy approach, the myopic choices in the first stages of the algorithm often lead to solutions of poor quality, since the constraints greatly restrict the number of alternative options. Moreover, such choices cannot be recovered, since C-Link does not involve any form of backtracking. In contrast, SR-CFSS can backtrack and hence, it can explore more options, leading to solutions of better quality.

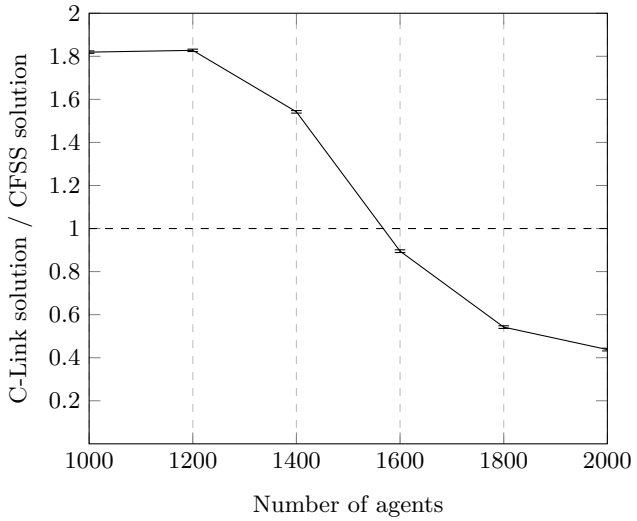


Fig. 6.18: Ratio between C-Link and SR-CFSS approximate solutions.

7

Payments for Social Ridesharing

In Chapter 6 we discussed the SR scenario, and, in particular, we showed how to model it as a GCCF problem in order to solve the associated optimisation problem, i.e., the CSG problem. In this chapter, we tackle the second fundamental aspect of CF, i.e., payment computation (see Section 2.1.3).

Such a task represents a key challenge in the CF process and it is of utmost importance when offering ridesharing services, especially when considering commuters with rational behaviours. In fact, payoffs (corresponding to cash payments for sharing trip costs) to the commuters need to be computed given their distinct needs (e.g., shorter/longer trips), roles (e.g., drivers/riders, less/more socially connected) and opportunity costs (e.g., taking a bus, their car, or a taxi).

One key aspect of payment distribution in CF is the game-theoretic concept of *stability*, which measures how agents are keen to maintain the provided payments instead of deviating to a configuration deemed to be more rewarding from their individual point of view. Here, we induce stable payments in the context of the SR problem, employing *the kernel* [29] stability concept. Kernel-stable payoffs are perceived as fair, since they ensure that agents do not feel compelled to claim part of their partners payoff. Kernel stability has been widely studied in cooperative game theory, and various approaches have been proposed to compute kernel-stable payments [63, 101]. Specifically, Shehory and Kraus [101] adopt a transfer scheme that represents the state of the art approach to compute kernel-stable payments.

7.1 State of the art approach

As introduced in Section 3.3, Algorithm 6 has been designed to compute payments for CF scenarios in which the set of coalitions is *not* restricted by a graph. Such an approach can be readily applied also when the size of coalitions is limited to k members, as it happens in a SR scenario in which all cars have k seats.[1]

Definition 7.1 (k-CF). *A CF problem is said to be a k-CF problem if the size of coalitions is limited to k members.*

[1] In real-world scenarios it is reasonable to assume that a car has 5 seats [118].

Algorithm 6 SHEHORYKRAUSKERNEL(x, CS, ϵ)

1: **repeat**
2: **for all** $S \in CS$ **do**
3: **for all** $a_i \in S$ **do**
4: **for all** $a_j \in S - \{a_i\}$ **do**
5: $s_{ij} \leftarrow \max_{\{S' \in 2^A \mid a_i \in S', a_j \notin S'\}} e(S', x)$
6: $\{a_{i^*}$ and a_{j^*} have the maximum surplus difference $\delta\}$
7: $\delta \leftarrow \max_{(a_i, a_j) \in A^2} (s_{ij} - s_{ji})$
8: $(a_{i^*}, a_{j^*}) \leftarrow \arg\max_{(a_i, a_j) \in A^2} (s_{ij} - s_{ji})$
9: **if** $x[j^*] - v(\{a_{j^*}\}) < \delta/2$ **then** {Payments are individually rational}
10: $d \leftarrow x[j^*] - v(\{a_{j^*}\})$
11: **else**
12: $d \leftarrow \delta/2$
13: $x[j^*] \leftarrow x[j^*] - d$ {Transfer payment from a_{j^*}...}
14: $x[i^*] \leftarrow x[i^*] + d$ {... to a_{i^*}}
15: **until** $\delta / V(CS) \leq \epsilon$

In k-CF, the maximisation at line 5 has to be assessed among the coalitions of size up to k which include a_i but exclude a_j. This set, denoted as $\mathcal{R}$, can be easily obtained as $\mathcal{R} = \{\{a_i\} \cup S \mid S$ is a h-combination of $A - \{a_i, a_j\}, \forall h \in \{1, \ldots, k-1\}\}$. Unfortunately, in GCCF scenarios like SR this simple approach would iterate over several unfeasible coalitions (i.e., which do not induce a connected subgraph of the social network), leading to inefficiency and reducing the scalability of the entire algorithm. In contrast, a better way to tackle this problem is to exploit the structure of the graph in order to consider *only* the coalitions that are indeed feasible, so to avoid any unnecessary computation.

Moreover, Algorithm 6 considers many coalitions more than once at the maximisation in the loop at lines 2–7. We provide the following example to clarify why this redundancy exists. Consider the set of agent $A = D = \{a_1, a_2, a_3, a_4\}$ and the graph G shown in Figure 7.1. Such graph induces the set of feasible coalitions $\mathcal{FC}(G) = \{\{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_1, a_4\}, \{a_1, a_2, a_3\}, \{a_1, a_2, a_4\}, \{a_1, a_3, a_4\}, \{a_1, a_2, a_3, a_4\}\}$, and assume a coalition structure $CS = \{\{a_1, a_2, a_3, a_4\}\}$.

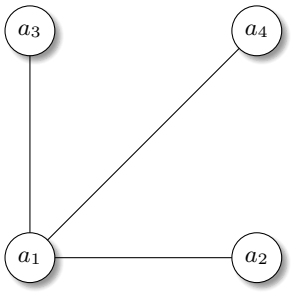


Fig. 7.1: Example of a social network with 4 agents.

a_i a_j	Coalitions
a_1 a_2	$\{a_1\}$ $\{a_1, a_3\}$ $\{a_1, a_4\}$ $\{a_1, a_3, a_4\}$
a_1 a_3	$\boldsymbol{\{a_1\}}$ $\{a_1, a_2\}$ $\boldsymbol{\{a_1, a_4\}}$ $\{a_1, a_2, a_4\}$
a_1 a_4	$\boldsymbol{\{a_1\}}$ $\boldsymbol{\{a_1, a_2\}}$ $\boldsymbol{\{a_1, a_3\}}$ $\{a_1, a_2, a_3\}$
a_2 a_1	$\{a_2\}$
a_2 a_3	$\boldsymbol{\{a_2\}}$ $\boldsymbol{\{a_1, a_2\}}$ $\boldsymbol{\{a_1, a_2, a_4\}}$
a_2 a_4	$\boldsymbol{\{a_2\}}$ $\boldsymbol{\{a_1, a_2\}}$ $\boldsymbol{\{a_1, a_2, a_3\}}$
a_3 a_1	$\{a_3\}$
a_3 a_2	$\boldsymbol{\{a_3\}}$ $\boldsymbol{\{a_1, a_3\}}$ $\boldsymbol{\{a_1, a_3, a_4\}}$
a_3 a_4	$\boldsymbol{\{a_3\}}$ $\boldsymbol{\{a_1, a_3\}}$ $\boldsymbol{\{a_1, a_2, a_3\}}$
a_4 a_1	$\{a_4\}$
a_4 a_2	$\boldsymbol{\{a_4\}}$ $\boldsymbol{\{a_1, a_4\}}$ $\boldsymbol{\{a_1, a_3, a_4\}}$
a_4 a_3	$\boldsymbol{\{a_4\}}$ $\boldsymbol{\{a_1, a_4\}}$ $\boldsymbol{\{a_1, a_2, a_4\}}$

Table 7.1: Coalitions computed by the loop at lines 2–7 of Algorithm 6.

In this case, such a loop requires 12 iterations, each looking at the coalitions reported in Table 7.1. Note that 23 (marked in bold) out of 33 coalitions (i.e., 70%) are evaluated more than once. This fact substantially reduces the efficiency and the scalability of the algorithm in SR scenarios, where the computation cost required to assess coalitional values is not negligible and caching is not an option. In fact, storing all these values in memory is not affordable even for systems with hundreds of agents: since $\mathcal{FC}(G)$ can contain up to $O\left(n^k\right)$ coalitions, for $k = 5$ and $n = 100$, storing all coalitional values requires tens of GB of memory. Thus, each coalitional value must be computed only when needed, since computing them more than once significantly reduces efficiency and scalability, as shown in Section 7.3.2.

To overcome these issues, in the next section we present the PRF algorithm, our payment scheme that scales up to systems with thousands of agents.

7.2 The PRF algorithm

We now present the PRF (Paying for Rides with Friends) algorithm [17], our method to compute an ϵ-kernel payoff allocation, given a coalition structure CS that is a solution to the SR problem.[2] Our contribution improves on the k-CF version of Algorithm 6 by adopting a novel approach to calculate the surplus matrix s. Instead of computing each value s_{ij} using the maximisation at line 7 for each pair of agents in each $S \in CS$, we iterate over the set of feasible coalitions (as specified in Definition 6.3) induced by G, and we update the appropriate values of the surplus matrix for each of such coalitions. Specifically, this is achieved by iterating over the set of $\hat{k}$-subgraphs of G, i.e., the set of connected subgraphs of G with *at most* k nodes, and then executing the update by means of the UpdateMax routine only for those $\hat{k}$-subgraphs that actually correspond to feasible coalitions. This additional check is mandatory since not all $\hat{k}$-subgraphs necessarily satisfy Constraint 1, and hence, represent feasible coalitions. By so doing, we ensure the exact coverage of $\mathcal{FC}(G)$, as proved by Proposition 7.3.

[2] While we present our contribution in the context of SR, our approach can be applied to all k-CF scenarios.

Algorithm 14 PRF(CS, ϵ)

1: **for all** $S \in CS$ **do**
2: **for all** $a_i \in S$ **do**
3: $x_i \leftarrow {}^{v(S)}/_{|S|}$ {Equally split coalitional value}
4: **repeat**
5: {Compute surplus matrix}
6: $s \leftarrow$ ComputeMatrix (CS, x)
7: {a_{i^*} and a_{j^*} have the maximum surplus difference δ}
8: $\delta \leftarrow \max_{(a_i,a_j)\in A^2} (s_{ij} - s_{ji})$
9: $(a_{i^*}, a_{j^*}) \leftarrow \arg\max_{(a_i,a_j)\in A^2} (s_{ij} - s_{ji})$
10: {Ensure that payments are individually rational}
11: **if** $x[j^*] - v(\{a_{j^*}\}) < {}^{\delta}/_{2}$ **then**
12: $d \leftarrow x[j^*] - v(\{a_{j^*}\})$
13: **else**
14: $d \leftarrow {}^{\delta}/_{2}$
15: $x[j^*] \leftarrow x[j^*] - d$ {Transfer payment from a_{j^*}...}
16: $x[i^*] \leftarrow x[i^*] + d$ {... to a_{i^*}}
17: **until** ${}^{\delta}/_{v(CS)} \leq \epsilon$

PRF is detailed in Algorithm 14. After having initialised the payoff vector x by equally splitting each coalitional value among the members of the coalition, ComputeMatrix computes the surplus matrix in each iteration of the main loop. In such a routine, UpdateMax is executed for each coalition that induces a $\hat{k}$-subgraph of G. These coalitions are computed with the SlyCE algorithm [115], which can list all the subgraphs of a given graph without redundancy (i.e., each subgraph is computed only once). Then, UpdateMax only considers the coalitions that satisfy Constraint 1 of the SR problem (line 1). For every S of such coalitions, lines 3–8 update all the values s_{ij} for which a_i is a member of S and a_j is part of S' (i.e., the coalition in CS that contains a_i) but is not part of S. The correctness of our approach is ensured by Proposition 7.2.

Proposition 7.2. *Algorithm 15 computes each s_{ij} correctly.*

Proof. Once the loop has ended, each s_{ij} stores the maximum excess among *all* feasible coalitions with a_i but without a_j, with both a_i and a_j part of the same coalition in CS. This matches line 7 of Algorithm 6. □

Our surplus matrix-calculating method has polynomial time complexity, while allowing to compute all feasible coalitions only once, as shown by Proposition 7.3.

Algorithm 15 ComputeMatrix(CS, x)

1: $s \leftarrow -\infty$ {Initialise the entire matrix with $-\infty$}
2: **for all** S that induce a $\hat{k}$-subgraph of G **do**
3: $s \leftarrow$ UpdateMax (S, CS, s, x)
4: **return** s

Algorithm 16 UpdateMax(S, CS, s, x)

```
1: if S satisfies Constraint 1 then
2:   e_S ← e(S, x) {Compute the excess of coalition S}
3:   for all a_i ∈ S do {For each agent a_i in coalition S}
4:     S' ← the coalition in CS that contains a_i
5:     for all a_j ∈ S' − S do {For each a_j ∈ S' but ∉ S}
6:       {s_ij is updated with the maximum between}
7:       {its old value and the excess of coalition S}
8:       s_ij ← max(s_ij, e_S)
9: return s
```

Proposition 7.3. *Algorithm 15 lists all feasible coalitions only once and it has a worst-case time complexity of* $O\left(n^k\right)$.

Proof. Algorithm 15 lists all $\hat{k}$-subgraph of G exactly once [115]. Note that the number of $\hat{k}$-subgraphs is $O\left(n^k\right)$, since we only consider coalitions with up to k members [101]. Hence, Algorithm 15 makes at most $O\left(n^k\right)$ calls to UpdateMax. Finally, note that the time complexity of UpdateMax is constant with respect to n, since computing $e(S, x)$ requires the computation of $v(S)$ (which has constant time complexity [18]), and the loop at lines 3–8 requires $O\left(k^2\right)$ iterations. Moreover, UpdateMax only considers coalitions that satisfy Constraint 1 (whose check is constant with respect to n) and it computes each coalitional value only once at line 2. Thus, Algorithm 15 computes all feasible coalitions only once and its worst-case time complexity is $O\left(n^k\right)$. □

In the next proposition, we prove that PRF has a polynomial time complexity.

Proposition 7.4. *Algorithm 14 has a polynomial worst-case time complexity with respect to* n, *i.e.,* $O\left(-\log_2(\epsilon) \cdot n^{k+1}\right)$.

Proof. Here we refer to equations and lemmas provided by Stearns [106]. Each iteration of Algorithm 14 identifies the agents a_i and a_j with the maximum surplus difference $\delta = s_{ij} - s_{ij}$, performing a transfer of size d from a_j to a_i. Thus, by Lemma 1 [106], in the following iteration these surpluses will be $s'_{ij} = s_{ij} - d$ and $s'_{ji} = s_{ji} + d$. Notice that $s'_{ij} - s'_{ji} = s_{ij} - s_{ji} - 2 \cdot d = \delta - 2 \cdot d$. Now, by definition of d (lines 11–14 of Algorithm 14), $d \leq \delta/2$, hence $s'_{ij} - s'_{ji} \geq 0$. Therefore, we can affirm that the transfer from a_j to a_i is indeed a *K-transfer*, since it satisfies Equation 4, 5, 6 and 7 [106]. Lemma 2 [106] ensures the convergence of Algorithm 2, by affirming that a K-transfer *cannot* increase the larger surpluses in the system. Specifically, in the next iteration the difference between the surpluses between a_j to a_i will be half of what was in the previous one. After λ iterations, its value will be $\frac{1}{2^\lambda}$ of the original one. Thus, it will take $\lambda = \log_2(\lceil \delta_0 / v(CS) \rceil / \epsilon)$ iterations to ensure that $\lceil \delta_0 / v(CS) \rceil / 2^\lambda \leq \epsilon$, with δ_0 being the original maximum s_{ij} surplus. Since we have n agents into the setting, it will take $\lambda \cdot n = O\left(-\log_2(\epsilon) \cdot n\right)$ iterations to convergence. Then, we know by Proposition 2 that ComputeMatrix, which dominates the time complexity of each iteration, has a worst-case time complexity of $O\left(n^k\right)$. Given this, Algorithm 2 has a worst-case time complexity of $O\left(-\log_2(\epsilon) \cdot n^{k+1}\right)$. □

Given this, PRF provides a polynomial method to compute kernel-stable payments. Nonetheless, the $O\left(n^k\right)$ operations required for surplus matrix calculation may not be affordable in real-world scenarios with thousands of agents and $k = 5$ (i.e., the number of seats of an average sized car). Hence, we next propose a parallel version of PRF, which allows us to distribute the computational burden among different threads, taking advantage of modern multi-core hardware.

7.2.1 P-PRF

We now detail P-PRF, the parallel version of our approach, in which the most computation-intensive task, i.e., the computation of the matrix s, is distributed among T available threads. In particular, Algorithm 17 details our parallel version of the ComputeMatrix routine, obtained by having each thread t to compute a separate matrix s^t. Such a matrix is constructed considering the coalitions in $\mathcal{DIV}(G, t, k)$, i.e., the t^{th} fraction of the set of all $\hat{k}$-subgraphs of G, computed using the D-SlyCE algorithm [115].[3] Specifically, this fraction is obtained by splitting the first generation of children nodes in the search tree generated by the SlyCE algorithm [115] among the available threads, allowing a fair division of the set of the $\hat{k}$-subgraphs while ensuring that all feasible coalitions are computed exactly once. As such, it also distributes the computation of the coalitional values.

Algorithm 17 P-ComputeMatrix(CS, x, T)

1: $s \leftarrow -\infty$ {Initialise all matrix elements with $-\infty$}
2: **for all** $t \in \{1, \ldots, T\}$ **do in parallel**
3: **for all** $S \in \mathcal{DIV}(G, t, k)$ **do**
4: $s^t \leftarrow$ UpdateMax(S, CS, s^t, x)
5: **for all** $i \in \{1, \ldots, n\}$ **do in parallel**
6: **for all** $j \in \{1, \ldots, n\}$ **do in parallel**
7: $s_{ij} \leftarrow \max_{t \in \{1,\ldots,T\}} s_{ij}^t$
8: **return** s

We provide the following example to clarify how this division is realised. Consider the same $\mathcal{FC}(G)$ of the example in Section 7.1, and assume $T = 4$. Then, the necessary coalitions are distributed by doing the following partitioning:

1. $\mathcal{DIV}(G, 1, k) = \{\{a_1\}, \{a_2\}, \{a_3\}\}$
2. $\mathcal{DIV}(G, 2, k) = \{\{a_4\}, \{a_1, a_2\}, \{a_1, a_3\}\}$
3. $\mathcal{DIV}(G, 3, k) = \{\{a_1, a_4\}, \{a_1, a_2, a_3\}\}$
4. $\mathcal{DIV}(G, 4, k) = \{\{a_1, a_2, a_4\}, \{a_1, a_3, a_4\}\}$

Note that, since each matrix s^t is modified only by thread t,[4] Algorithm 17 contains only one synchronisation point (i.e., before line 5), hence providing a full paralleli-

[3] Notice that nor SlyCE neither D-SlyCE solve the payment computation problem, as they address the enumeration of the $\hat{k}$-subgraphs of G.

[4] P-PRF requires storing t separate surplus matrices, one per thread. Hence, its memory requirements are $O\left(t \cdot n^2\right)$, i.e., still polynomial in the number of agents.

sation. After that, the final surplus matrix s is computed with a maximisation on all the above matrices (lines 5–7), ensuring that the output of P-ComputeMatrix is equal to the one of ComputeMatrix, since each feasible coalition in $\mathcal{FC}(G)$ has been computed by a thread. The effectiveness of our parallel approach will be demonstrated through the empirical evaluation, detailed in the following section.

7.3 Empirical evaluation

Having discussed our approach, we now benchmark it on real-world datasets. The main goals of the empirical analysis are:

1. To test the performance of PRF when computing payments for systems of thousands of agents.
2. To perform an analysis of the features that influence the distribution of payments among the agents.
3. To investigate the impact of time constraints on the above properties.
4. To compare the efficiency of PRF with respect to the state of the art approach proposed by Shehory and Kraus.
5. To estimate the speed-up obtainable by using P-PRF with respect to PRF.

In all our tests, we adopt the same methodology and datasets discussed in Section 6.4 (i.e., we adopt the GeoLife and Twitter datasets), unless otherwise stated. In the experiments looking at the performance of PRF (i.e., Sections 7.3.1, 7.3.2 and 7.3.3) we only consider the SR model without time constraints, since the performance of PRF is negligibly affected by them.[5]

7.3.1 Runtime performance

In our first experiment, we aim at evaluating the performance of our approach when computing payments in large-scale instances. Figure 7.2 shows the runtime needed to execute P-PRF on systems with $n \in \{100, 500, 1000, 1500, 2000\}$. In each test, the coalition structure has been computed using the approximate version of SR-CFSS using our SR model without time constraints.

Our results show that P-PRF is able to compute payments for 2000 agents with a runtime ranging from 13 to 50 minutes, hence it can successfully scale to large systems. In particular, for each value of n, we consider $|D| \in \{10\%, 50\%, 80\%\}$. Our results also show the influence of the percentage of drivers on the complexity of the problem. On average, computing payments on an instance with $|D| = 80\%$ is easier with respect to $|D| = 10\%$ and $|D| = 50\%$. Our findings are consistent with the results in Section 6.4.1, showing that the scenario with $|D| = 50\%$ is more difficult to solve since more drivers are available, hence it is possible to form more cars, resulting in a larger search space. In fact, the number of feasible coalitions is determined by the number of available seats (reduced when such a percentage is low) and the number of riders without a car who can benefit from sharing their commutes (reduced when the majority of the agents owns a car).

[5] The complexity of computing each coalitional value is comparable whether or not we consider time constraints.

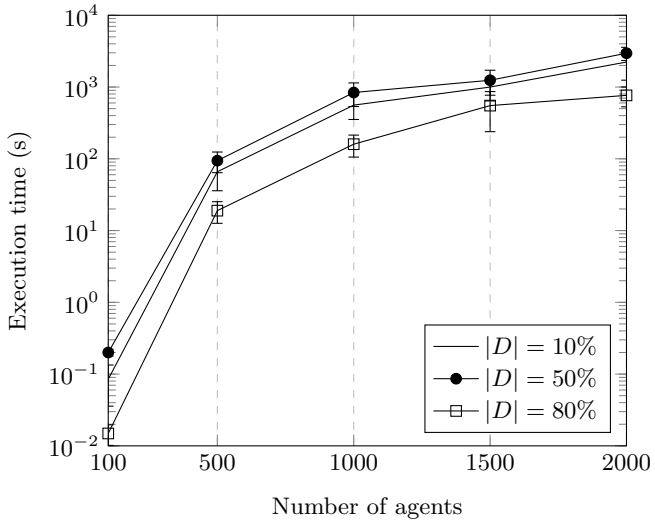


Fig. 7.2: Runtime needed to compute payments.

7.3.2 Comparison with the state of the art

Figure 7.3 shows the runtime needed by our approach to compute a kernel-stable payoff vector, comparing it with the state of the art approach by Shehory and Kraus [101], i.e., Algorithm 6. In particular, we consider the runtime needed to solve SR instances with $n \in \{30, 40, 50, 60, 70, 80, 90, 100\}$ and $|D| = 50\%$. We employ the sequential version of PRF, since Algorithm 6 is also sequential.

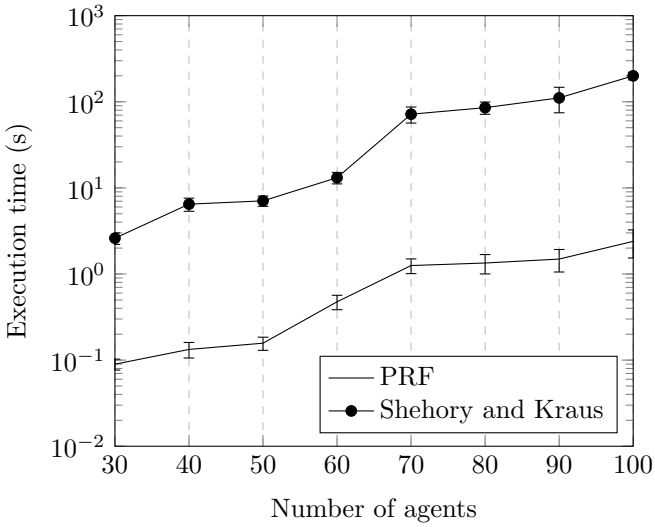


Fig. 7.3: Runtime needed to compute payments.

Our results show that PRF is at least one order of magnitude faster than the counterpart approach, outperforming the state of the art by 27× in the worst case, with an average improvement of 53×, and a best case improvement of 84×. Thus, our comparison has been run only up to $n = 100$, since the counterpart approach becomes impractical for instances with thousands of agents. In fact, with 1000 agents it requires over one day of computation, compared to a runtime of 2 hours required by PRF, and 14 minutes required by P-PRF. In particular, the approach by Shehory and Kraus is slower since it makes several redundant computations of many coalitional values, resulting in a significant impact on its runtime.

7.3.3 Parallel performance

Here we analyse the speed-up that can be achieved by using P-PRF with respect to PRF, i.e., its sequential version. We ran the algorithms on instances with 500 agents and $|D| = 50\%$, using a machine with 2 Intel® Xeon® E5-2420.

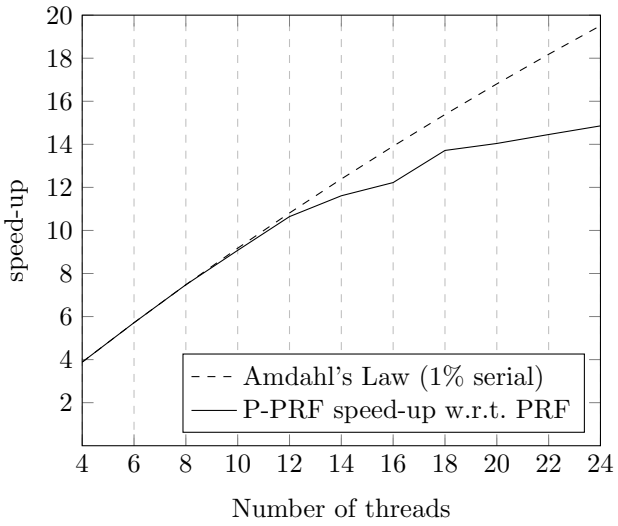


Fig. 7.4: Multi-threading speed-up.

The speed-up measured during these tests has been compared with the maximum theoretical one provided by the Amdahl's Law [3], considering an estimated non-parallelisable part of 1%, due to memory allocation and thread initialisation. Figure 7.4 shows that the actual speed-up follows the theoretical one for up to 12 threads (i.e., the number of physical cores for this machine), reaching a final speed-up of 14.85× with all 24 threads active.

7.3.4 Costs and network centrality without time constraints

The purpose of this section is to analyse the relationship between the cost incurred by a commuter and its importance in the environment, i.e., being a node with a high degree in the social network, or being driver or rider. To this end, we first compute the optimal solution of a SR problem without time constraints on random instances with $n \in \{30, 40, 50, 60, 70, 80, 90, 100\}$ and $|D| \in \{10\%, 50\%, 80\%\}$, and we use our algorithm to compute a kernel-stable payoff vector. Then, to assess this correlation in a quantified manner, we define the *normalised cost* $\overline{c}_i$ and the *normalised degree* $\overline{d}_i$ for each agent a_i as follows:

- For any a_i in a coalition S with $|S| > 1$, we define its *normalised cost* $\overline{c}_i$ as

$$\overline{c}_i = \frac{-x[i] - \min_x^S}{\max_x^S - \min_x^S},$$

  where $\min_x^S$ and $\max_x^S$ are the minimum and the maximum values of the negative values of x among the members of S, i.e., $\min_x^S = \min_{a_i \in S} -x[i]$ and $\max_x^S = \max_{a_i \in S} -x[i]$. Note that we consider negative values since in our model, costs are represented by negative values for $x[i]$.
- For any a_i in a coalition S with $|S| > 1$, we define its *normalised degree* $\overline{d}_i$ as

$$\overline{d}_i = \frac{deg\,(a_i) - \min_d^S}{\max_d^S - \min_d^S},$$

  where $deg\,(a_i)$ represents the degree of a_i in the social network, and $\min_d^S$ and $\max_d^S$ are the minimum and the maximum degrees among the members of S.

When the denominator of $\overline{c}_i$ is 0, i.e, when $\max_x^S = \min_x^S$, it means that all the agents in C have the same payoff. In these cases, $\overline{c}_i$ is defined to be 0.5 as a middle point between 0 and 1 (the same discussion applies to $\overline{d}_i$).

Notice that, a direct comparison of two agents that are not part of the same coalition would not be appropriate for determining their overall power or benefits derived from participation in the SR setting, since payments computed according to the kernel do not consider agents belonging to different coalitions (see Section 2.1.4). Nonetheless, it would definitely be interesting to have a way to measure and compare the power of the agents, regardless of the coalition to which each one belongs. To allow this comparison, both $\overline{c}_i$ and $\overline{d}_i$ are normalised between 0 (for the agents having the minimum costs/degrees in their coalitions) and 1 (similarly for the agents with maximum costs/degrees). The normalisation is done with respect to the coalition the agent belongs to, because to reach kernel-stability, payment transfers only take place among agents within the same coalition. Finally, note that agents in singletons have been excluded from this analysis, as they do not have to split their coalitional value.

In Figure 7.5 we report the average and the standard error of the mean for the normalised cost with respect to the normalised degree. Our results clearly show that costs are strongly influenced by the degree of the agents, and whether they are drivers or riders. Specifically, in our tests drivers had to pay costs that were on average 16% lower than riders. Moreover, agents with the minimum number of social connections in their coalition (i.e., with a normalised degree of 0) paid a cost 171% higher than the ones with the highest degree.

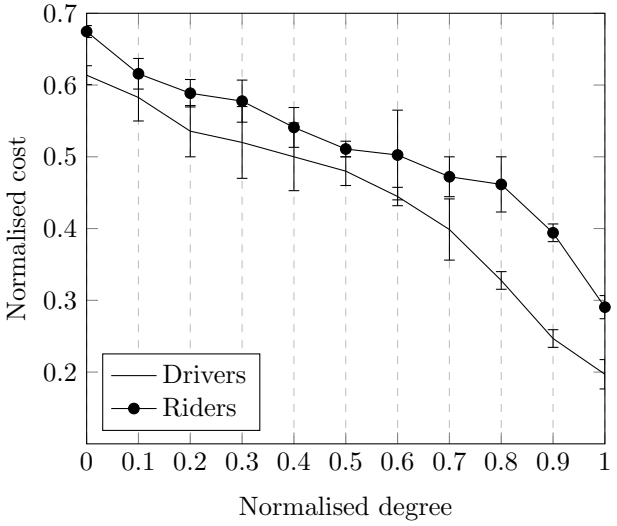


Fig. 7.5: Normalised cost w.r.t. normalised degree without time constraints.

7.3.5 Costs and network centrality with time constraints

In this section we investigate how the same features of the cost distributions studied in the previous section are affected by the introduction of time constraints. To this end, we repeat the above experiment using our SR model with time constraints.

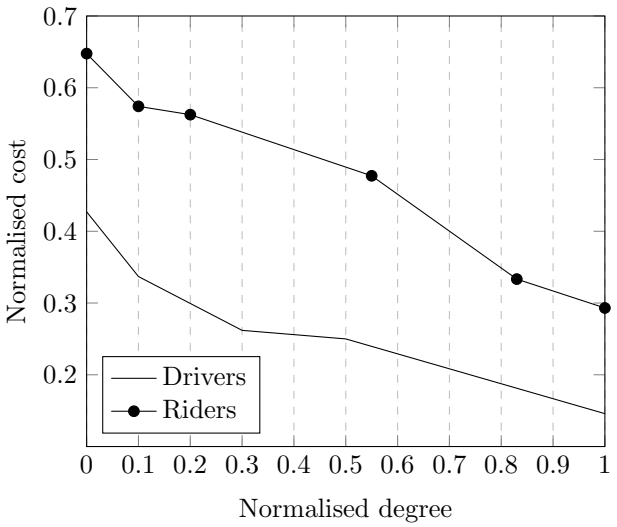


Fig. 7.6: Normalised cost with respect to normalised degree with time constraints.

Figure 7.6 shows a behaviour similar to the one discussed in the above section. Moreover, we notice that the introduction of time constraints results in significantly lower costs for the drivers (i.e., drivers pay costs that are on average 35% lower than the previous experiment), while riders' costs are comparable in both scenarios. These results can be explained by recalling that time constraints significantly reduce the solution space (see Sections 6.4.2 and 6.4.4), and hence, the influence of drivers (who are crucial to determine whether or not a coalition can be formed) is even stronger if the pool of possible alternative coalitions is smaller.

We further investigate the role of time constraints in the payment distribution process by studying to what extent more tolerant agents are rewarded with lower costs. To this end, we assign a random θ_i's radius within $\{5', 10', 15', 20', 25', 30'\}$ to each agent and we look at the corresponding normalised cost.

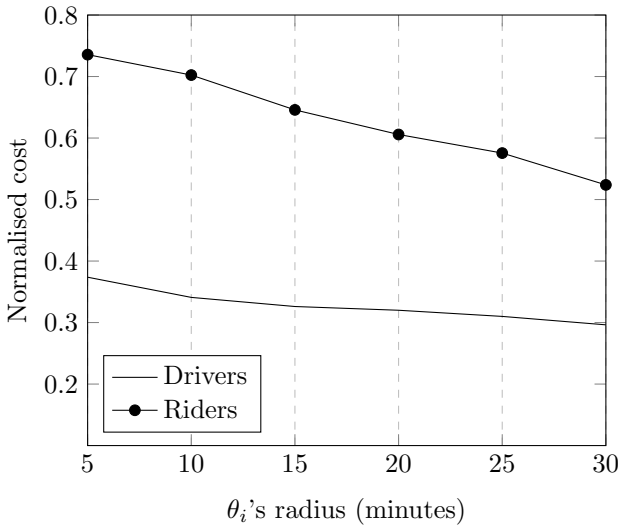


Fig. 7.7: Normalised cost with respect to θ_i's radius

Figure 7.7 shows that the agents that are willing to tolerate more with respect to their ideal departure/leaving time are rewarded by the system with lower costs, as a consequence of the fact that, by having a larger θ_i's radius, they can choose among a larger pool of alternatives and hence, they achieve a higher bargaining power in the payment distribution process.

In general, our experimental results suggest that the kernel can be a valid stability concept in the context of SR. In fact, it induces a reasonable behaviour in the formation of groups, which can be directly correlated with some simple properties of the agents in the system (i.e., network centrality and being a driver or a rider). Moreover, the computation of kernel-stable payments has a tractable complexity and hence, it is suitable for large-scale environments, in contrast with stronger stability concepts (e.g., the core).

Part IV

Bucket Elimination on GPUs

8

CUBE: a CUDA implementation for Bucket Elimination

The main objective of this thesis is the study and the design of novel computational methods to solve combinatorial optimisation problems (such as GCCF), in Multi-Agent Systems. The techniques discussed so far perform particularly well under the assumption that the value of each coalition can be expressed by means of a closed-form function, and it is possible to derive a method to compute an upper bound for such function to apply CFSS (see Sections 4.4 and 6.2). However, in some GCCF scenarios it may be difficult (or not possible at all) to meet these premises, hence the application of CFSS may be not convenient. As an example, in the context of ridesharing, coalitional values may be characterised by a random component expressing the costs due to traffic.

Against this background, in the remainder of the thesis we investigate an alternative solution method for GCCF that is applicable in scenarios where CFSS is not a viable approach. In the optimisation literature, Dynamic Programming (DP) [28] historically represents the counterpart approach with respect to search, especially in the context of GCCF [90, 115]. Moreover, DP-based algorithms represent the state of the art for solving CSG [89] and GCCF [115] in scenarios that consider a general characteristic function. Such facts warrant the study of an approach for GCCF based on DP, with the objective of developing a high-performance solution method that overcomes the drawbacks of previously discussed algorithms.

In recent years, Graphics Processing Units (GPUs) have been successfully used to speed-up the computation in different applications that feature a high level of parallelism, achieving performance improvements of several orders of magnitude [41] in fields including computer vision [8], human-computer interaction [11], and artificial intelligence [85, 108]. Parallelisation has also been investigated to speed-up search-based approaches on multi-core CPUs [83], but the application of these techniques to GPUs is difficult for several reasons. On the one hand, general Depth-First Search (DFS) is known to be difficult to parallelise [92],[1] especially on highly parallel architectures such as GPUs. Moreover, the use of branch and bound strategies to guide the search may result in heavily unbalanced search trees, requiring complex techniques to balance the workload among the threads [83]. Such

[1] Even if Reif [92] focuses on the problem of enumerating the nodes of a graph adopting DFS, the same negative result also applies when DFS is used in an optimisation context to traverse a search tree.

techniques are not effective on GPUs, where load balancing is crucial to achieve a high computational throughput. For these reasons, the solutions proposed in the previous chapters would be ineffective if implemented on GPUs, since our main result, i.e., the CFSS algorithm, is a branch and bound DFS algorithm.

On the other hand, DP has been successfully parallelised on GPUs [20, 44, 54, 85, 107], motivating the study of a new research line in this direction. In the remainder of the thesis we discuss how to benefit from the computing capabilities of GPUs for AI problems, and, specifically, for GCCF. We achieve this objective by exploiting the close relation between such a problem and Constraint Optimisation Problems (COPs) (see Section 2.2), proposing a novel formalisation for GCCF based on a COP (see Chapter 9). Specifically, in this chapter we propose CUBE (CUda Bucket Elimination) [12, 15], a highly parallel implementation for the join sum and maximisation operations associated to Bucket Elimination (BE), which we use to solve COPs as discussed in Section 2.3. CUBE employs a novel methodology for the parallelisation of such operations, which is specifically designed to consider two fundamental aspects of the GPU algorithmic design, i.e., thread independence and memory management, as discussed in Section 2.5.

In the design of CUBE, we aim at developing a high-performance GPU framework that allows us to deal with the computational effort inherent in the message passing phase of several BE-based algorithms. To this end, our main objective is to devise a solution that fulfils three key requirements. First, since BE is a general algorithm that can be applied to several problems, our framework should be likewise general to allow a wide adoption among different domains. Second, our approach should be able to achieve a high computational throughput, by means of optimised memory accesses to avoid bandwidth bottlenecks, a careful load-balancing to fully exploit the available computational power, and the adoption of well-known parallel primitives [96, 100] to reduce the CPU workload to the minimum. Third, our solution should not be limited by the amount of GPU memory (see Section 2.5.1).

CUBE achieves the objectives set above by means of a novel preprocessing algorithm that reorders the rows and the columns of the tables representing the constraints of the COP (see Section 2.2) so to achieve optimised memory accesses. Such an arrangement enables pipelined data transfers (see Section 2.5.2), hence optimising the transfer time, and it allows the use of highly efficient routines for the fundamental parts of BE, i.e., composition (see Section 2.3.1) and marginalisation (see Section 2.3.2). CUBE is not limited by the amount of GPU memory, as our data layout allows us to process large tables by splitting them into manageable chunks that meet the memory capabilities of the GPU.

Now, the capability of each thread to efficiently access its input data is a crucial aspect in the design of GPU algorithms [41], as it directly determines the final computational throughput. Within CUBE, we avoid unnecessary, expensive memory accesses by proposing a technique that allows threads to locate their input data only on the base of their own ID. We take advantage of the *data reuse* pattern inherent in the composition and the marginalisation operations by caching the input data in the *shared memory*, i.e., the fastest form of memory in the GPU hierarchy [41] (see Section 2.5.1). Bandwidth efficiency is also ensured by the high spatial locality inherent in our data representation, as discussed in detail in Section 8.3.

Unlike previous approaches [120], CUBE does not require input tables to be complete (i.e., to contain all the rows corresponding to every possible assignment of the variables in the table scope), as it is designed to exploit the structure of tables with missing rows inherent in COPs (see Section 8.2). Nonetheless, we also provide a specialised version of our method that can take advantage of the completeness of tables and that can be used in scenarios where such tables contain all the possible assignments (e.g., belief propagation on junction trees or COPs in which unfeasible assignments are explicitly represented with $-\infty$ values). Such method is discussed in the following section.

8.1 Processing complete tables

In this section, we describe how we preprocess complete tables in order to index their rows efficiently and achieve coalesced memory accesses. Specifically, we first discuss our approach in the context of Belief Propagation (BP), showing how we exploit this table layout within the GPU kernel executing the actual message passing phase of BP through highly efficient routines [16]. Then, in Section 8.1.3 we show how to adapt such method in the context of COPs.

8.1.1 Table preprocessing

Suppose we have to employ the BP algorithm as discussed in Section 2.4 in order to propagate new evidence from the potential table T_1 to the potential table T_2, respectively associated to two tuples of variables $Q_1 = \langle x_3, x_2, x_1 \rangle$ and $Q_2 = \langle x_5, x_4, x_1 \rangle$, with the shared variables $Q_{12} = Q_1 \cap Q_2 = \langle x_1 \rangle$ (Figure 8.1). We assume that x_1, x_3 and x_5 are binary variables, while x_2 and x_4 can assume 3 values. In the approach by Zheng and Mengshoel [120], each row of the separator table Sep_{12} is assigned to a different block of threads, which are responsible for the reduction of the rows of T_1 with a matching variable assignment and the subsequent scattering on matching rows in T_2. In Figure 8.2, rows associated to different blocks of threads have been marked in different colours, i.e., white and grey for $x_1 = 0$ and $x_1 = 1$ respectively. The organisation of input data provided by these tables is undesirable for GPU architectures. In fact, threads responsible for the computation of white rows cannot access consecutive memory addresses, as their data is interleaved with grey rows, thus breaking memory coalescence. Moreover, even if the computation of white rows requires half of the input data, its sparsity forces us to transfer the entire tables to the global memory before starting the algorithm.

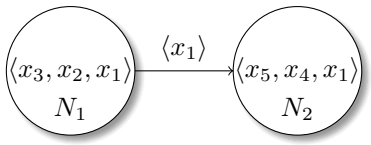


Fig. 8.1: Junction tree example.

We propose to solve these issues by means of a preprocessing phase, in which rows associated to the same row in Sep_{12} (i.e., rows of the same colour, in the above example) are stored in *consecutive* addresses in the corresponding potential tables, as shown in Figure 8.3. Threads responsible for white rows execute coalesced memory accesses, and start the computation while grey rows are still being transferred to the GPU. Each block of threads easily retrieves its input data, in contrast with the approach by Zheng and Mengshoel [120] that adopts costly mapping tables resulting in a higher memory footprint.

Consider $T_1^p = \langle Q_1^p, d_1^p, R_1^p, \phi_1^p \rangle$ in Figure 8.3, resulting from a permutation σ of Q_1 in which all the shared variables (i.e., $Q_{12} = \langle x_1 \rangle$) are brought to the *Most Significant*[2] (MS) positions in $Q^p = \sigma(Q)$. In this way, we can assure that rows with the same assignment of the variables in Q_{12} form a contiguous chunk of memory. Our table representation by means of ordered tuples imposes that d^p, R^p and ϕ^p are coherently defined, to guarantee the equivalence to the original table. While the former can be easily obtained by applying σ to d, the computation of R^p can be avoided, therefore only ϕ^p requires a particular discussion, which is covered in the following sections.

Table indexing

Since in any table $T = \langle Q, d, R, \phi \rangle$, R contains *all* the possible variable assignments, we can avoid storing R in memory. In fact, since the order of variables is fixed, given any row $r = R[k]$, k can be computed with:

$$k = \sum_{i=1}^{|Q|-1} \Bigg(r[i] \underbrace{\prod_{j=i+1}^{|Q|} d[j]}_{\mathcal{D}[i]} \Bigg) + r[|Q|] = \sum_{i=1}^{|Q|-1} \left(r[i] \cdot \mathcal{D}[i] \right) + r[|Q|] \tag{8.1}$$

where $r[i]$ represents the value assumed by the variable $Q[i]$ in r.

T_1

x_3	x_2	x_1	ϕ_1
0	0	0	α_0
0	0	1	α_1
0	1	0	α_2
0	1	1	α_3
0	2	0	α_4
0	2	1	α_5
1	0	0	α_6
1	0	1	α_7
1	1	0	α_8
1	1	1	α_9
1	2	0	α_{10}
1	2	1	α_{11}

S_{12}

x_1	ϕ_{12}
0	γ_0
1	γ_1

T_2

x_5	x_4	x_1	ϕ_2
0	0	0	β_0
0	0	1	β_1
0	1	0	β_2
0	1	1	β_3
0	2	0	β_4
0	2	1	β_5
1	0	0	β_6
1	0	1	β_7
1	1	0	β_8
1	1	1	β_9
1	2	0	β_{10}
1	2	1	β_{11}

Fig. 8.2: Original tables.

[2] Variables are listed from the most significant to the least significant.

T_1^p

x_1	x_3	x_2	ϕ_1^p
0	0	0	α_0
0	0	1	α_2
0	0	2	α_4
0	1	0	α_6
0	1	1	α_8
0	1	2	α_{10}
1	0	0	α_1
1	0	1	α_3
1	0	2	α_5
1	1	0	α_7
1	1	1	α_9
1	1	2	α_{11}

S_{12}^p

x_1	ϕ_{12}^p
0	γ_0
1	γ_1

T_2^p

x_1	x_5	x_4	ϕ_2^p
0	0	0	β_0
0	0	1	β_2
0	0	2	β_4
0	1	0	β_6
0	1	1	β_8
0	1	2	β_{10}
1	0	0	β_1
1	0	1	β_3
1	0	2	β_5
1	1	0	β_7
1	1	1	β_9
1	1	2	β_{11}

Fig. 8.3: Preprocessed tables.

Definition 8.1 ($\mathcal{D}$). *Each $\mathcal{D}[i]$ represents the product of all the elements starting from position $i+1$ in d, hence we refer to the tuple $\mathcal{D}$ as the* exclusive postfix product *of d. Such an operation can be seen as a variation of the standard* exclusive prefix sum *operation, in which the result is computed by summing all the elements up to $i-1$. We define $\mathcal{D}[|Q|] \coloneqq 1$ (the identity element for the product), similarly to the definition of the first element of the exclusive prefix sum as 0.*

On the other hand, each $r[i]$ can be retrieved from k as $r[i] = \lfloor k/\mathcal{D}[i] \rfloor \bmod d[i]$. Thus, R can be dropped from our representation in memory, hence, as previously claimed, the computation of R^p is unnecessary. For a better understanding, let r with $Q = \langle x_1, x_2, x_3 \rangle$, and $d = \langle 2, 16, 10 \rangle$:

$r =$

x_1	x_2	x_3	ϕ
1	10	7	v_{267}

From Equation 8.1, r is in position $k = 1 \cdot d[2] \cdot d[3] + 10 \cdot d[3] + 7 = 267$ in ϕ. Moreover, $x_1 = 1 = \lfloor 267/\mathcal{D}[1] \rfloor \bmod d[1]$, $x_2 = 10 = \lfloor 267/\mathcal{D}[2] \rfloor \bmod d[2]$ and $x_3 = 7 = \lfloor 267/\mathcal{D}[3] \rfloor \bmod d[3]$.

As mentioned before, to maintain a coherent representation of the preprocessed table $T^p = \langle Q^p, d^p, R^p, \phi^p \rangle$, the values in ϕ must be correctly permuted into ϕ^p.

Table reordering

This section will cover our approach to achieve the column reordering detailed in Section 8.1.1. As mentioned before, we do not store R, since each row $r \in R$ can be retrieved from its index with the above detailed technique, hence the computation of R^p will not be covered. On the other hand, for any $\phi[k]$ at index k in ϕ it is necessary to compute its index k^p in the preprocessed table T^p to compute ϕ^p.

A naïve approach would require to apply the permutation σ on each row $r = R[k]$, which comprises 3 steps: for each k, compute the corresponding variable assignment $\langle r[1], \ldots, r[i], \ldots, r[|Q|]\rangle$, apply σ on the now available sequence of $r[i]$ and, finally, obtain k^p using Equation 8.1. Since each of the 3 above mentioned steps has a complexity of $O(|Q|)$, such approach requires $O(|\phi||Q|)$. This complexity depends on both the dimensions of the table (i.e., the number of rows and columns), and can be problematic when increasing the table size.

In what follows, we show a more efficient approach to calculate k^p. For simplicity, we first explain how to compute the index resulting from swapping the variables at positions i and j. Then, we provide an algorithm to compute k^p by means of a sequence of swaps.

Proposition 8.2. *Given $T = \langle Q, d, R, \phi\rangle$ and $T^s = \langle Q^s, d^s, R^s, \phi^s\rangle$, where Q^s and d^s has been respectively obtained swapping $Q[i]$ with $Q[j]$ and $d[i]$ with $d[j]$ (with $i > j$), ϕ^s is a permutation of ϕ, i.e., $\phi[k] = \phi^s[k']$ and k' is:*

$$\begin{aligned} k' = {} & r[1]{\cdot}d[2]\cdots d[i]\cdots d[j]\cdots d[|Q|]{+}\cdots && (2a)\\ & + r[i]{\cdot}d[j{+}1]\cdots d[j]\cdots d[|Q|] && (2b)\\ & + r[j{+}1]{\cdot}d[j{+}2]\cdots d[|Q|]{+}\cdots{+}r[i{-}1]{\cdot}d[j]\cdots d[|Q|] && (2c)\\ & + r[j]{\cdot}d[i{+}1]\cdots d[|Q|] && (2d)\\ & + r[i{+}1]{\cdot}d[i{+}2]\cdots d[|Q|]{+}\cdots{+}r[|Q|] && (2e) \end{aligned}$$

Then, $k' = f(k, i, j)$ can also be calculated as:

$$k' = \overbrace{k - k \bmod \mathcal{D}[j{-}1]}^{(2a)} + \overbrace{\mathcal{D}[j]\cdot {}^{d[j]}/_{d[i]}\cdot \lfloor {}^{k}/_{\mathcal{D}[i]}\rfloor \bmod d[i]}^{(2b)} + \overbrace{k \bmod \mathcal{D}[i]}^{(2e)}$$
$$+ \underbrace{{}^{d[j]}/_{d[i]}\cdot \overbrace{(k \bmod \mathcal{D}[j] - k \bmod \mathcal{D}[i{-}1])}^{(2c')}}_{(2c)} + \underbrace{\mathcal{D}[i]\cdot \lfloor {}^{k}/_{\mathcal{D}[j]}\rfloor \bmod d[j]}_{(2d)}$$

Proof. We use the following properties, which can be demonstrated by means of basic algebraic procedures:

$$k = \sum_{h=1}^{|Q|} r[h]{\cdot}\mathcal{D}[h] \implies \sum_{h=l}^{|Q|} r[h]{\cdot}\mathcal{D}[h] = k \bmod \mathcal{D}[l-1] \tag{8.3}$$

$$k = \sum_{h=1}^{|Q|} r[h]{\cdot}\mathcal{D}[h] \implies r[h] = \lfloor {}^{k}/_{\mathcal{D}[h]}\rfloor \bmod d[h] \tag{8.4}$$

From Equation 8.1 we can easily verify that $k' = (2a) + (2b) + (2c) + (2d) + (2e)$. Similarly, k can be written as:

$$\begin{aligned} k = {} & \overbrace{r[1]\cdot\mathcal{D}[1] + \cdots + r[j-1]\cdot\mathcal{D}[j-1]}^{(2a)} + r[j]\cdot\mathcal{D}[j]\\ & + \overbrace{r[j+1]\cdot\mathcal{D}[j+1] + \cdots + r[i-1]\cdot\mathcal{D}[i-1]}^{(2c')} + r[i]\cdot\mathcal{D}[i]\\ & + \overbrace{r[i+1]\cdot\mathcal{D}[i+1] + \cdots + r[|Q|]}^{(2e)} \end{aligned}$$

To prove the correctness of Proposition 8.2, we show that:

- $(2a) = k - k \bmod \mathcal{D}[j-1]$
- $(2b) = \mathcal{D}[j] \cdot {}^{d[j]}/_{d[i]} \cdot \lfloor {}^{k}/_{\mathcal{D}[i]} \rfloor \bmod d[i]$
- $(2c) = {}^{d[j]}/_{d[i]} \cdot (k \bmod \mathcal{D}[j] - k \bmod \mathcal{D}[i-1])$
- $(2d) = \mathcal{D}[i] \cdot \lfloor {}^{k}/_{\mathcal{D}[j]} \rfloor \bmod d[j]$
- $(2e) = k \bmod \mathcal{D}[i]$.

$(2a)$, $(2d)$ and $(2e)$ are not affected by the swap of $Q[i]$ and $Q[j]$: since $(2a)$ refers to all the variables before $Q[j]$ (and, consequently, before $Q[i]$), all the terms $\mathcal{D}[1] \cdots \mathcal{D}[j-1]$ contain both $d[i]$ and $d[j]$, hence the swap does not produce any effect. On the other hand, $(2e)$ contains neither $d[i]$ nor $d[j]$, since it refers to all the variables after $Q[i]$, thus $(2e)$ is not affected either. Using Property 8.3, we can calculate $(2a)$ as the difference between k and all the terms from $r[j] \cdot \mathcal{D}[j]$ on, i.e., $(2a) = k - k \bmod \mathcal{D}[j-1]$, while $(2e)$ is given by $k \bmod \mathcal{D}[i]$. Finally, in $(2d)$ none of the terms $d[i+1] \cdots d[|Q|] = \mathcal{D}[i]$ contains either $d[i]$ or $d[j]$, thus $(2d) = r[j] \cdot \mathcal{D}[i] = \mathcal{D}[i] \cdot \lfloor {}^{k}/_{\mathcal{D}[j]} \rfloor \bmod d[j]$.

On the contrary, $(2b)$ and $(2c)$ are affected by the swap of $Q[i]$ and $Q[j]$: to calculate them, we first calculate $(2c')$ as the difference between all the terms from $r[j+1] \cdot \mathcal{D}[j+1]$ on and all the terms from $r[i] \cdot \mathcal{D}[i]$ on, i.e., $(2c') = k \bmod \mathcal{D}[j] - k \bmod \mathcal{D}[i-1]$. By using Property 8.4, we compute $r[i] = \lfloor {}^{k}/_{\mathcal{D}[i]} \rfloor \bmod d[i]$ and $r[j] = \lfloor {}^{k}/_{\mathcal{D}[j]} \rfloor \bmod d[j]$. Finally, $d[i]$ needs to be substituted with $d[j]$ in all the terms $\mathcal{D}[j] \cdots \mathcal{D}[i-1]$ in $(2b)$ and $(2c)$, since $Q[j]$ has been moved to a less significant position, taking the place of $Q[i]$ which has been moved to a more significant one. Thus, we multiply $(2c')$ by ${}^{d[j]}/_{d[i]}$ to compensate for this effect and obtain $(2c)$. Equivalently, we calculate $(2b) = \mathcal{D}[j] \cdot {}^{d[j]}/_{d[i]} \cdot \lfloor {}^{k}/_{\mathcal{D}[i]} \rfloor \bmod d[i]$ by swapping $d[i]$ with $d[j]$ in $\mathcal{D}[j]$. □

Proposition 8.2 is then used to reorder any potential table T according to the layout detailed in Section 8.1.1. More formally, let $\mathcal{S} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ be a sequence of n swaps, each represented by an ordered[3] pair of positions $\mathcal{S}[i] = \langle a_i, b_i \rangle$, so that we permute Q into $\sigma(Q)$ (moving the desired subset of variables to the MS positions) by means of the sequence of i swaps of the variables in positions a_i and b_i, as described in Proposition 8.2. Then, ϕ^p is computed with Algorithm 18.

Algorithm 18 PREPROCESSCOMPLETE$(\phi, \mathcal{S})$

```
1: for all k ∈ {1, ..., |φ|} do in parallel
2:    k^p ← k
3:    for all ⟨a_i, b_i⟩ ∈ S do {For every swap in S}
4:       k^p ← f(k^p, a_i, b_i) {Proposition 8.2}
5:       SWAP(Q[a_i], Q[b_i]) {Swap variables}
6:       SWAP(d[a_i], d[b_i]) {Swap variable domains}
7:    φ^p[k^p] ← φ[k] {Write φ[k] in position k^p of φ^p}
8: return φ^p
```

[3] We assume that, for every pair $\langle a_i, b_i \rangle$, $a_i > b_i$.

The tuple of swaps $\mathcal{S}$ required to move $|Q_{12}|$ variables to the MS positions of the tables T_1 and T_2 is computed as follows. Consider T_1 and let us assume that s shared variables (with $0 \leq s \leq |Q_{12}|$) are already within the first $|Q_{12}|$ positions of the corresponding variable tuple Q_1. Then, it is sufficient to swap the $|Q_{12}|-s$ shared variables with index greater than $|Q_{12}|$ with the non-shared ones which are placed within the first $|Q_{12}|$ positions. On the other hand, table T_2 can be preprocessed by swapping each shared variable $Q_2[h]$ with $Q_2[k]$ such that $Q_2[h] = Q_1[k]$ for $k \in \{1, \ldots, |Q_{12}|\}$. This algorithm ensures the same order of the variables in Q_{12} in both tables.

As an example, we reorder the row $R_1[10] = \langle 1, 2, 0\rangle$ in position $k = 10$ of T_1 in Figure 8.2 and compute its index k^p in T_1^p. In this case, $Q_1 = \langle x_3, x_2, x_1\rangle$ and the desired order is obtained first with $\mathcal{S}[1] = \langle 3, 1\rangle$, and then with $\mathcal{S}[2] = \langle 3, 2\rangle$,[4] i.e, by swapping $Q[3] = x_1$ with $Q[1] = x_3$, then swapping $Q[3] = x_3$ with $Q[2] = x_2$. This produces the desired result $Q_1^p = \langle x_1, x_3, x_2\rangle$.

We first execute $\mathcal{S}[1]$, which swaps the variables in positions 3 and 1. The corresponding domains are $d[3] = d[1] = 2$, thus $\mathcal{D}[3] = 1$ and $\mathcal{D}[1] = 6$. Then, applying Proposition 8.2 to the row with index $k = 10$ results in $(2a) = (2e) = 0$ (since there are no variables before x_3 and after x_1), $(2b) = 6 \cdot {}^2\!/_2 \cdot 0 = 0$, $(2c) = {}^2\!/_2\,(10 \bmod 6 - 10 \bmod 2) = 4$ and $(2d) = 1 \cdot 1 = 1$, hence $f\,(10, 3, 1) = 5$, meaning that α_{10} would have index 5 after $\mathcal{S}[1]$. To compute its final index, we apply $\mathcal{S}[2] = \langle 3, 2\rangle$. At this point $Q_1 = \langle x_1, x_2, x_3\rangle$, $\mathcal{D}[3] = 1$, $\mathcal{D}[2] = d[3] = 2$ and $d[2] = 3$, hence $(2c) = (2e) = 0$ (since there are no variables after x_3 and between x_2 and x_3). On the other hand, $(2a) = 5 - 5 \bmod 6 = 0$, $(2b) = 2 \cdot {}^3\!/_2 \cdot 1 = 3$ and $(2d) = 1 \cdot 2 = 2$, thus $\phi^p[5] = \alpha_{10}$ (see T_1^p).

Algorithm 18 provides a method to rearrange any couple of potential tables T_i and T_j such that the variables of their separator are moved to the MS positions, according to Section 8.1.1. In the next section, the impact of this preprocessing phase on the overall performance of the algorithm will be analysed in detail, by showing how it is more efficient than the naïve approach previously mentioned.

Computational complexity

Proposition 8.3. *Algorithm 18 has a time complexity of* $O\,(|\phi||\mathcal{S}|) \leq O\,(|\phi|^{|Q_{12}|}/_2)$ $< O\,(|\phi||Q|)$.

Proof. The external loop (line 1) requires $|\phi|$ iterations, the inner loop (line 3) requires $|\mathcal{S}|$ iterations, equal to ${}^{|Q_{12}|}\!/_2$ assuming the worst case of reordering all the variables in Q_{12}. Since lines 4-6 can be computed in $O\,(1)$, the resulting time complexity is $O\,(|\phi||\mathcal{S}|) \leq O\,(|\phi|^{|Q_{12}|}/_2)$. □

In our experimental evaluation, we performed the variable ordering with an average of $|\mathcal{S}| = 3$ swaps, resulting in an improvement of an order of magnitude with respect to the naïve approach, which, in contrast, requires tens of operations for each row. It is important to note that this preprocessing phase is done *once for all*, while compiling the BN in the corresponding JT (see Section 2.4). In fact, the

[4] Swapping x_3 and x_2 is not necessary since neither of them belongs to Q_{12}, but it has been included in our example to better explain the algorithm.

acquisition of new evidence does not change the structure of the network itself, hence we can avoid to reorder each potential table at each BP by storing and updating the couple of corresponding reordered tables for each separator. Notice that such preprocessing phase does not result in additional memory requirements since the original tables can be discarded as they are not needed in any subsequent phase of the algorithm. Furthermore, each iteration of the external loop (lines 1–7) of Algorithm 18 is independent and can be computed in parallel. As a consequence, the worst-case time complexity of the parallel version of Algorithm 18 is $O(|\phi||\mathcal{S}|/t)$, where t is the number of threads. Given a JT $= (V, E)$, our algorithm needs to store a couple of potential tables for each separator. Since threads can index input rows on-the-fly, mapping tables can be avoided. Thus, the memory requirements are $O(2 \cdot |E|)$. In contrast, the approach proposed by Zheng and Mengshoel [120] maintains one potential table for each clique, but it needs two mapping tables for each separator table. Hence, it requires $O(V + 2 \cdot |E|)$ tables, resulting in a greater memory footprint.

8.1.2 GPU kernel for BP

In our approach to BP on GPUs, each block of threads is responsible for one *element* of the separator table, which is associated to a corresponding group of rows in potential tables. Such high-level organisation of the computation allows us to carry out the entire reduction and scattering stages within a single thread block, hence avoiding any costly inter-block synchronisation structure. On one hand, the performance of our algorithm clearly benefits from the lack of interdependence among different blocks, which would reduce the overall computation parallelism. On the other hand, since the size of thread blocks has an intrinsic limit imposed by the hardware architecture (e.g., 2048 threads in Kepler GPUs), the proposed organisation may serialise part of the workload if the number of rows to manage exceeds such limit. Nevertheless, such an issue is not problematic in our test cases, since the above mentioned case rarely verifies. In fact, in our experimental evaluation, each block has to reduce an average of 14 elements,[5] hence allowing a full parallelisation. If the serialisation is small (i.e., each thread has to reduce and scatter few rows), the effect on the overall performance is negligible. This is due to the fact that the task is computed extremely efficiently in thread-private memory space using registers. In what follows, we explain the actual implementation of the above mentioned concepts in detail.

Reduction (⇓)

Once the input data is in the shared memory, the kernel starts the reduction phase that, in our approach, is implemented with the NVIDIA CUB library[6] by means of a *block reduce raking algorithm*. The algorithm consists of three steps: i) an initial sequential reduction in registers (if each thread contributes to more than one input), in which warps other than the first one place their partial reductions

[5] This is the average, over all BNs, of the ratio between the average potential table size and the average separator table size (see Table 8.1).

[6] Available at `http://nvlabs.github.io/cub`.

into shared memory, ii) a second sequential reduction in shared memory, in which threads within the first warp accumulate data by ranking across segments of shared partial reductions, and iii) a final reduction within the raking warp based on the Kogge-Stone algorithm [64] produces the final output.

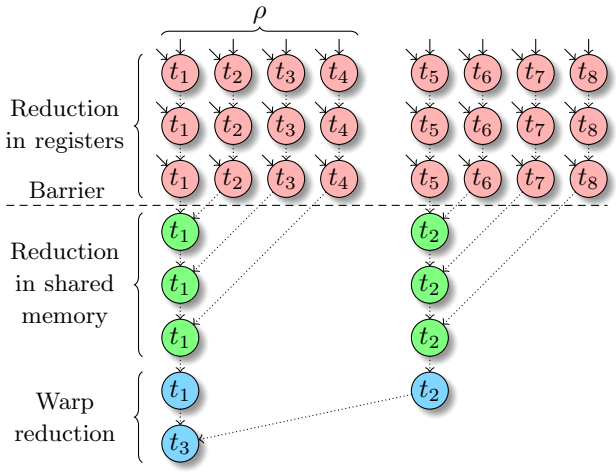


Fig. 8.4: Block reduce raking algorithm (best viewed in colour).

Figure 8.4 shows a reduction of 32 input values performed by a block of 8 threads $\{t_1, \ldots, t_8\}$, in which each thread operation is pictured as a circle and each input value is represented with a solid arrow, while the dotted ones represent partial results propagated among threads. In our implementation, the definition of parameter ρ is managed by the aforementioned library, in order to achieve a good balance between parallelisation degree and communication among threads. This scheme is particularly efficient, since it involves a single synchronisation barrier after the first phase it incurs zero bank conflicts[7] for primitive data types. On newer CUDA architectures (e.g., NVIDIA Kepler), such implementation exploits *shuffle* instructions, which are a new set of primitives provided by the CUDA programming language. Shuffle instructions enable threads within the same warp to exchange data through direct register accesses, hence avoiding shared memory accesses and improving the computational throughput of the algorithm. In particular, such scheme is collectively performed by the block of threads associated to a particular element of the separator table, in order to compute its updated value as the sum of the corresponding rows of the first potential tables, i.e., the ones with a matching variable assignment. Once the reduction of the entire chunk has been completed, the output of the reduction serves as input for the subsequent scattering phase of belief propagation.

[7] If multiple memory accesses map to the same memory bank, the accesses are serialised and split into as many separate conflict-free requests as necessary, thus decreasing the effective bandwidth.

Scattering ($\oplus$)

The final stage of BP consists of the *scattering* operation (see Section 2.4), which performs the actual update of T_2^p by means of R_{ij} computed in the above mentioned phase. The implementation of such operation benefits from the proposed memory layout, since it is realised with maximum parallelism and computational throughput. Each row of T_2^p is assigned to one thread, which multiplies its current value for R_{ij}, computed in the reduction phase. Once the kernel has been executed by all blocks, the propagation of belief has completed the inclusion of new evidence in T_2^p, which can be finally transferred back to the CPU memory.

Having discussed how we implement a CUDA kernel for BP that exploits our improved table layout, we now show how we realise a kernel for the solution of COPs that achieves the same performance benefits.

8.1.3 GPU kernel for COPs

In this section we discuss our approach for the implementation of the message passing phase of the version of BE that solves COPs (Algorithm 2) [15]. Specifically, we provide a highly parallel algorithm that computes the $\oplus$ and the $\Downarrow$ operators of BE, i.e., the join sum (see Section 2.3.1) and the maximisation (see Section 2.3.2). Notice that the approach discussed in this section is specifically devised to exploit the improved table layout described in Section 8.1.1, which requires tables to be complete. Therefore, here we assume that tables contain all possible assignments of the variables in their scope, including unfeasible assignments that are explicitly represented with $-\infty$ values. A more general approach, which can also process incomplete tables,[8] will be detailed in Section 8.2.

Join sum ($\oplus$)

We first discuss the implementation of the join sum operation on GPUs. Such operation, denoted as $\oplus$, is very similar to the *join* of relational algebra, in which the output table contains one row for each couple of rows of the input tables that have a matching assignment of the shared variables. In the case of the join sum, the value of each row is given by the sum of the values of the corresponding input rows. To better explain how the join sum works, we consider the tables T_1^p and T_2^p in Figure 8.3. In what follows, we denote as *group* a set of rows that all have the same assignment over the shared variables, or, more intuitively, the same colour.

In order to achieve a full parallelisation of the join sum, we adopt a *gather* paradigm [66], in which each thread is responsible for the computation of exactly one element of the output. Such a paradigm offers many advantages with respect to the counterpart approach, i.e., the *scatter*[9] paradigm, in which each thread is associated to one element of input and contributes to the computation of many

[8] Representing all variable assignments can lead to tables of untractable size, since, in general, the number of rows is exponential in the number of variables.

[9] Even if this paradigm shares the same name with the *scattering* phase of belief propagation introduced in Section 3, it refers to a completely different concept.

Algorithm 2 BucketEliminationCOP $(\mathrm{CN}, F_1, \ldots, F_l, o)$

Partition $\{C_1, \ldots, C_m\}$ and $\{F_1, \ldots, F_l\}$ into n buckets according to o
for all $p \leftarrow n$ down to 1 **do**
  for all $C_k, \ldots, C_g$ over scopes $X_k, \ldots, X_g$, and
  for all $F_h, \ldots, F_j$ over scopes $Q_h, \ldots, Q_j$,
  in bucket p **do**
    if $x_p = a_p$ **then**
      $x_p \leftarrow a_p$ in each F_i and C_i
      Put each F_i and C_i in appropriate bucket
    else
      $U_p \leftarrow \bigcup_i X_i - \{x_p\}$
      $V_p \leftarrow \bigcup_i Q_i - \{x_p\}$
      $W_p \leftarrow U_p \cup V_p$
      $C^p \leftarrow \pi_{U_p} \left(\bowtie_{i=1}^{g} C_i\right)$
      for all tuples t over W_p **do**
        $F^p(t) \leftarrow \underset{\substack{a_p \mid (t,a_p) \text{ satisfies} \\ \{C_1,\ldots,C_g\}}}{\Downarrow} \oplus_{i=1}^{j} F_i(t, a_p)$
      Place F^p in the latest lower bucket mentioning a variable in W_p,
      and C^p in the latest lower bucket with a variable in U_p
Assign maximising values for the functions in each bucket
return $F(\bar{a}^*)$, i.e., the optimal cost computed in the first bucket and $\bar{a}^*$, i.e., the optimal assignment

output elements. In fact, scatter-based algorithms have a reduced degree of parallelism since they often require atomic primitives (which inherently serialise parts of the computation) to avoid having multiple threads concurrently operating on the same output. As previously discussed, only the array ϕ is stored in memory, since we assume that tables are *complete*[10] and, hence, it is not necessary to store the variable assignment part. Therefore, we only discuss how we compute the array ϕ of the output table $T_i \oplus T_j$, denoted as $\phi_\oplus$. We map one GPU thread t to each element of $\phi_\oplus$, denoted as $\phi_\oplus[t]$.

Our main goal is that each thread should be capable of computing the indices of its input rows in T_1^p and T_2^p in a closed form only on the base of its own ID t, with the aim of avoiding unnecessary memory accesses to the input data. To achieve this, we now introduce some background concepts needed to explain our indexing approach. First, notice that the number of rows in each group is equal to the number of all the possible assignments of the non-shared variables in the scope of the table, i.e., the product of the domain sizes of such variables. In particular, each group in T_1^p consists of 6 rows, as $D_2 \cdot D_3 = 6$, and the same applies to T_2^p, i.e., $D_4 \cdot D_5 = 6$. Since the join sum operation associates each of these 6 rows in T_1^p to each of the 6 matching rows in T_2^p, the corresponding group in the output table will contain $D_2 \cdot D_3 \cdot D_4 \cdot D_5 = 36$ rows.

[10] A table T_i with the scope X_i is *complete* if it contains all the possible assignments over the domains of the variables in X_i. We represent *unfeasible* rows as $-\infty$ values.

In general, it is easy to verify that, if $X_i = \{x_{i_1}, \ldots, x_{i_h}\}$ and $X_j = \{x_{j_1}, \ldots, x_{j_k}\}$ are the scopes of the input tables T_i and T_j, the output table $T_i \oplus T_j$ (where the $\oplus$ operator represents the join sum) contains a number of rows equal to:

$$\left(\prod_{x_a \in X_i \cap X_j} D_a\right) \cdot \underbrace{\left(\prod_{x_b \in X_i - X_j} D_b\right) \cdot \left(\prod_{x_c \in X_j - X_i} D_c\right)}_{rows(X_i, X_j)}. \tag{8.5}$$

For convenience, we define the function $rows$ to denote the number of rows in each group of the output table induced by the scopes X_i and X_j. Formally, $rows : 2^X \times 2^X \to \mathbb{N}$, where X is the set of variables and 2^X denotes the powerset of X.

Algorithm 19 JoinSumCompleteKernel(t, X_i, X_j)

1: $g \leftarrow \lfloor \frac{t}{rows(X_i, X_j)} \rfloor$ {Output group t belongs to}
2: $idx \leftarrow t \bmod rows\,(X_i, X_j)$ {ID of t within g}
3: $\#_i \leftarrow \prod_{x_b \in X_i - X_j} D_b$ {# of rows associated to g in T_i}
4: $\#_j \leftarrow \prod_{x_c \in X_j - X_i} D_c$ {# of rows associated to g in T_j}
5: $\gamma \leftarrow g \cdot \#_i + \lfloor \frac{idx}{\#_j} \rfloor$ {Input row in T_i}
6: $\delta \leftarrow g \cdot \#_j + idx \bmod \#_j$ {Input row in T_j}
7: $\phi_\oplus[t] \leftarrow \phi_i[\gamma] + \phi_j[\delta]$ {Compute and store output}

Algorithm 19 summarises the approach we propose to compute the join sum of two tables T_i and T_j, which is executed in parallel by each thread to index the input tables and to compute each row of the output table. As a first step, each thread t identifies which group it belongs to (line 1), by dividing its index t for the number of rows in each output group, i.e., $rows\,(X_i, X_j)$. Specifically, t operates within the g^{th} group. Furthermore, t computes its index idx relative to the first row of its group in line 2. Then, to compute the indices γ and δ of its two input rows, t first calculates $\#_i$ and $\#_j$, representing the number of rows of each group in T_i and T_j respectively, by multiplying the sizes of the domains of the non-shared variables in each table (lines 3 and 4).

A further inspection of lines 5 and 6 reveals how Algorithm 19 organises the $rows\,(X_i, X_j)$ elements of the g^{th} output group among the corresponding GPU threads. It associates the first $\#_j$ rows of such group to the first row of the g^{th} group in T_i, and each of these threads is then associated to each of the $\#_j$ rows of the g^{th} group in T_j. This pattern is then repeated for the second row of the g^{th} group in T_i, and so on for all the $\#_i$ rows of the g^{th} group in T_i (Figure 8.5). Note that the offsets $g \cdot \#_i$ and $g \cdot \#_j$ ensure the selection of the g^{th} group in T_i and T_j, as they represent the total number of rows in the g groups that precede the g^{th} one in each input table.

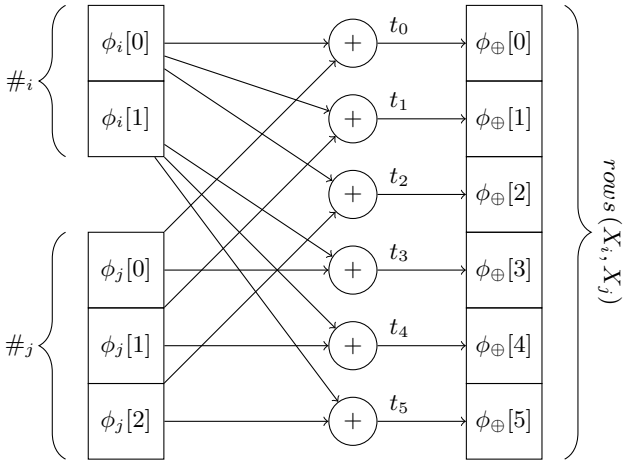


Fig. 8.5: Join sum output computation.

For a better understanding, we show how Algorithm 19 computes the row at index 59 of $T_1^p \oplus T_2^p$. Such a row would be computed by the thread $t = 59$, associated to the index $idx = 23$ of the output group $g = 1$, i.e., the grey one. In fact, as introduced earlier in this section, $rows(X_1, X_2) = 36$. It is easy to verify that $\#_i = \#_j = 6$. Then, t computes the indices of its input rows in T_1^p and T_2^p, i.e., $\gamma = 6 + 3 = 9$ and $\delta = 6 + 5 = 11$. Hence, $t = 59$ computes the element at index 23 of the output grey group, i.e., the one associated to the line at index 3 of the grey group in T_1^p and the last line of the grey group in T_2^p, as represented by γ and δ.

Note that the only input required by each thread executing Algorithm 19 is its own ID t, since X_i and X_j are equal and known in advance by all threads. The thread ID t does not determine *which* operations are executed (as they are equal for all threads), but only *where* the input data is located. For these reasons, Algorithm 19 fits perfectly the SIMD model adopted by GPU architectures. In addition, Algorithm 19 does not contain any branching instruction, which would cause a phenomenon called *divergence*, which reduces the degree of parallelism by forcing the serialisation of threads executing different branches of the program [50].

Finally, Algorithm 19 relies on a *data reuse* pattern, as each row of T_i is the input of $\#_j$ output elements and, symmetrically, each row of T_j is the input of $\#_i$ output elements. We avoid expensive accesses to the GPU global memory[11] by first transferring each coloured group to the shared memory, which allows threads to fetch data roughly 100× faster [41]. Notice that the use of the shared memory is possible only because we represent the input data with the table layout discussed in Section 8.1, in which coloured groups are in small, contiguous chunks of memory. Since GPUs only have tens of KB of shared memory available, it is not possible to achieve the same benefits with the original tables (Figure 8.2), which should be transferred *in toto*, possibly exceeding the hardware capabilities of the GPU.

[11] Global memory, in which the data is initially stored, is the slowest type of memory of the GPU hierarchy [41].

Having discussed our GPU implementation of the join sum, in what follows we show how we realise the maximisation operation of BE in parallel on the GPU.

Maximisation (⇓)

Maximisation can be seen as a particular case of the relational algebra *project* operation. In the case of BE, maximisation operates by removing the variable associated to the current bucket from the input table T_i. As a consequence, the resulting table contains D_p copies of each unique assignment of the variables in its scope, i.e., $X_i - \{x_p\}$. Maximisation then maps each unique assignment to the maximum of the D_p values mentioned above. For example, if we want to compute the maximisation of T_1 (Figure 8.2) by removing x_3, we first obtain the table shown in Figure 8.6, in which each unique variable assignment is highlighted with a different colour (here $D_p = D_3 = 2$). The final output is computed as shown in Figure 8.8. Figures 8.6 and 8.8 highlight the high degree of parallelisation inherent in the maximisation operation, as each coloured group can be processed independently from the others.

x_2	x_1	ϕ
0	0	α_0
0	1	α_1
1	0	α_2
1	1	α_3
2	0	α_4
2	1	α_5
0	0	α_6
0	1	α_7
1	0	α_8
1	1	α_9
2	0	α_{10}
2	1	α_{11}

Fig. 8.6: T_1 without x_3.

x_2	x_1	x_3	ϕ^p
0	0	0	α_0
0	0	1	α_6
0	1	0	α_1
0	1	1	α_7
1	0	0	α_2
1	0	1	α_8
1	1	0	α_3
1	1	1	α_9
2	0	0	α_4
2	0	1	α_{10}
2	1	0	α_5
2	1	1	α_{11}

Fig. 8.7: T_1 after the preprocessing.

x_2	x_1	$\phi_{\Downarrow}$
0	0	$\max(\alpha_0, \alpha_6)$
0	1	$\max(\alpha_1, \alpha_7)$
1	0	$\max(\alpha_2, \alpha_8)$
1	1	$\max(\alpha_3, \alpha_9)$
2	0	$\max(\alpha_4, \alpha_{10})$
2	1	$\max(\alpha_5, \alpha_{11})$

Fig. 8.8: Maximisation output (best viewed in colour).

Nonetheless, Figure 8.6 also highlights the poor data locality of this table layout (similar to the one in Figure 8.2), which causes the same issues discussed in Section 8.1.1. To overcome these problems, we preprocess the input table to achieve the row arrangement shown in Figure 8.7. In particular, we aim at placing each coloured group in consecutive memory locations, so to achieve a better data locality and improve the efficiency of the maximisation operation. This is equivalent to moving x_p to the last column, and can be implemented with the technique presented in Section 8.1.1, and specifically by considering as *shared* all the variables in the scope of the table minus x_p.

This table layout enables an efficient GPU algorithm to compute the final output of the maximisation operation, i.e., $\phi_{\Downarrow}$ in the above example. In general, the array ϕ of the output table can be computed with a *segmented reduction* algorithm [100], a well-known GPU primitive that differs from the standard reduction in that the latter operates on the entire set of input elements (e.g., it computes the maximum over the entire length of the input array), while the former operates on several fractions of the input data, i.e., the coloured groups in our case.

Finally, we further improve the efficiency of our CUBE implementation by avoiding unnecessary data transfers between the host and the device when computing the maximisation operation. In particular, since the BE algorithm always applies the maximisation operation on the result of the join sum operation (line 13 of Algorithm 2), we can avoid to transfer the join sum result (produced on the GPU memory) from the GPU to the CPU and directly run the maximisation on the GPU, hence saving two data transfers.

All the techniques discussed in this section assume that tables are complete. Unfortunately, representing complete tables in memory is not practical for several applications,[12] as, in general, the number of rows of a complete table is exponential in the number of variables in its scope. Against this background, in what follows we propose a more general version of CUBE that can process incomplete tables.

8.2 Processing incomplete tables

In this section we elaborate our approach to the parallelisation of the BE algorithm for COPs[13] that does not require complete tables [12]. In this way, unfeasible assignments can be dropped from the representation in memory, so to achieve a better scalability thanks to lower memory requirements.

8.2.1 Table preprocessing

In order to explain our approach, we consider the example introduced in Section 2.3.1 (Figure 2.6). The goal is to rearrange the rows of these tables so to have the same final placement discussed in Section 8.1.1, since the current data organisation suffers from very poor data locality.

[12] Including the COP formalisation for GCCF proposed in Chapter 9.

[13] In contrast with Section 8.1, here we only discuss our method for COPs, since BP typically does not involve hard constraints and hence, it results in complete tables.

T_1

x_1	x_3	x_5	x_8	ϕ_1
0	1	0	1	α_0
1	0	0	1	α_1
1	1	0	1	α_2
0	1	0	0	α_3
0	0	0	1	α_4
1	1	1	1	α_5

$\oplus$

T_2

x_1	x_2	x_3	x_4	x_6	x_{10}	ϕ_2
1	0	0	1	1	0	β_0
1	0	1	1	1	0	β_1
0	1	0	0	1	1	β_2
1	1	0	1	0	1	β_3
0	0	0	1	1	0	β_4
1	1	1	1	1	1	β_5

Fig. 2.6: Original tables T_1 and T_2 (best viewed in colour).

In particular, the preprocessing of these tables aims at achieving two fundamental requirements: i) rows of the same colour should be in consecutive memory addresses, to have full coalescence in memory accesses and to reduce the sparsity of data; ii) coloured groups should be in the same order (considering the set of shared variables) in both tables, to locate them efficiently when computing the join sum result. This is required since tables can be incomplete. We achieve these objectives by means of Algorithm 20.

Algorithm 20 PREPROCESSINCOMPLETE (T_1, T_2)

1: Move shared variables in Q_1 and Q_2 to the $|Q_{12}|$ LS places
2: Sort R_1, R_2, ϕ_1, ϕ_2 using a LSD radix sort on the $|Q_{12}|$ LS places
3: Remove row groups that do not match between T_1 and T_2

The first step of the preprocessing phase requires to move the $|Q_{12}|$ columns corresponding to the shared variables to the $|Q_{12}|$ Least Significant (LS) places. Notice that this step is an *embarrassingly parallel* [45] task, and it can be trivially divided among $|R|$ threads, each independently processing a single row. Subsequently, the algorithm reorders R and ϕ by means of a LSD radix sort algorithm [96] implemented with the NVIDIA CUB library. It is not necessary to adopt a radix sort algorithm in this phase (as every sorting algorithm that operates on the basis of the LS $|Q_{12}|$ places would work). However, we decide to use such an algorithm since it can be parallelised very efficiently [96].

As a final step, the algorithm remove the non-matching groups of rows (white rows in Figure 2.6), since they do not generate any output row in the result table, obtaining the preprocessed tables T_1^p and T_2^p in Figure 8.9. Notice that none of these three steps requires to have an entire table stored in the global memory, thus it is possible to easily split the input tables into manageable chunks meeting the memory capabilities of the GPU and preprocess them.[14]

[14] If it is necessary to sort a table larger that the GPU global memory, it is possible to split it into chunks, sort each of them (using the above mentioned radix sort algorithm), and then merge the sorted chunks (adopting the merge sort algorithm) on the CPU.

T_1^p

x_5	x_8	x_1	x_3	ϕ_1^p
0	1	0	0	α_4
0	1	1	0	α_1
0	1	1	1	α_2
1	1	1	1	α_5

$\oplus$

T_2^p

x_6	x_{10}	x_2	x_4	x_1	x_3	ϕ_2^p
1	1	1	0	0	0	β_2
1	0	0	1	0	0	β_4
1	0	0	1	1	0	β_0
0	1	1	1	1	0	β_3
1	0	0	1	1	1	β_1
1	1	1	1	1	1	β_5

Fig. 8.9: Final preprocessed tables (best viewed in colour).

In the next section, we discuss how it is possible to exploit this row layout to index these tables and have multiple thread efficiently locate their input to compute the join sum in parallel on the GPU.

8.2.2 Join sum ($\oplus$)

Similar to what we discussed in Section 8.1.3, we adopt a *gather* paradigm [66], in which each thread is responsible for the computation of exactly one element of the output. As mentioned before, such a paradigm offers many advantages with respect to the *scatter* paradigm, in which each thread is associated to one element of input and contributes to the computation of many output elements.

In our particular case, one thread computes one particular output row at index i (i.e., both $R[i]$, the variable assignment part, and $\phi[i]$, the value part), on the basis of the two associated input rows, which can be identified as explained below. First, we compute the number of rows in each coloured group for T_1 and T_2. As a result, we obtain a tuple $\mathcal{H}$ such that $\mathcal{H}[i]$ is the number of rows of the i^{th} coloured group. This operation can be seen as the computation of the *histogram* of the rows of the tables, which is a well-know primitive that can be parallelised very efficiently. In the above example, $\mathcal{H}_1{=}\langle 1,1,2\rangle$, and $\mathcal{H}_2{=}\langle 2,2,2\rangle$. These histograms are also useful to compute the number of rows of the result table, a crucial information when we have to allocate the exact amount of memory to store the result. Each group of output rows has a number of elements equal to the product of the numbers of rows of the corresponding input groups. Hence, the histogram of the result table, namely $\mathcal{H}_\oplus$, is computed as the *element-wise product*[15] (denoted as $*$) of the input histograms. It is easy to verify that $\langle 1,1,2\rangle * \langle 2,2,2\rangle = \langle 2,2,4\rangle$ is the histogram of the result table in Figure 2.7. The sum of the values of such histogram is the total number of rows of the result table.

These histograms also allow each thread to efficiently locate its input rows, as well as the index of the output row it is responsible for, by indexing the coloured groups in T_1 and T_2. As a first step, we compute the *exclusive prefix sum*[16] of the input and output histograms, which can be done very efficiently on the GPU [100] and, in our case, it is implemented with the NVIDIA CUB library. Given an

[15] Element-wise product is an embarrassingly parallel operation.

[16] In contrast with the *exclusive postfix product* introduced in Definition 8.1, in the *exclusive prefix sum* the result at index i is computed by summing all the elements of the input up to the position $i-1$.

Algorithm 21 JoinSumIncompleteKernel $(\mathcal{H}_1, \mathcal{H}_\oplus, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_\oplus)$

1: $bx \leftarrow$ block ID
2: $tx \leftarrow$ thread ID
3: **if** $tx < \mathcal{H}_\oplus[bx]$ **then** {If the thread ID is within the size of the block...}
4: $idx_1 \leftarrow \mathcal{P}_1[bx] + tx \bmod \mathcal{H}_1[bx]$ {Index of the input row in T_1}
5: $idx_2 \leftarrow \mathcal{P}_2[bx] + \lfloor tx / \mathcal{H}_1[bx] \rfloor$ {Index of the input row in T_2}
6: $idx_\oplus \leftarrow \mathcal{P}_\oplus[bx] + tx$ {Index of the output row}
7: Compute the join sum of input rows at indices idx_1 and idx_2 in T_1 and T_2
8: Store the results in $R_\oplus[idx_\oplus]$ and $\phi_\oplus[idx_\oplus]$

histogram $\mathcal{H}$, its exclusive prefix sum $\mathcal{P}$ is a tuple in which each element $\mathcal{P}[i]$ represents the index of the first row of the i^{th} coloured group. With these data structures, each thread can compute its row in the join sum result, as summarised in Algorithm 21. Such an algorithm represents the actual kernel function executed by the GPU, which receives as inputs the histograms of T_1 and the output histogram (i.e., $\mathcal{H}_1$ and $\mathcal{H}_\oplus$),[17] as well as the corresponding prefix sum tuples (i.e., $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_\oplus$). The variable assignment and the value parts of the output table are respectively denoted as $R_\oplus$ and $\phi_\oplus$.

It is important to note the absence of divergence in Algorithm 21, thanks to the fact that the only branch instruction (line 3) is used to limit the number of running threads to the amount needed, i.e., $\mathcal{H}_\oplus[bx]$. For the sake of clarity, we made a number of simplifications in Algorithm 21. First, here we do not explicitly mention the use of *shared memory*, which is used to exploit the data reuse[18] inherent to the join sum operation, so to avoid unnecessary memory accesses to the global memory. The properties of these memory transfers between shared and global memory are discussed in Section 8.3.1. Furthermore, we assume that each coloured group of rows is computed by exactly one block of threads. In contrast, our actual implementation realises a *dynamic load balancing* by assigning the appropriate number of groups to each block, in order to achieve a higher GPU occupancy and computational throughput. This number is determined by the amount of shared memory of the GPU and the maximum number of threads per block. Our approach can also process groups of rows which are larger than the available shared memory: this case is managed by our implementation by splitting such a group into a number of sub-groups, once again with the objective of maximising the GPU occupancy.

8.2.3 Maximisation (⇓)

In this section, we describe how we implement the maximisation operation of BE on GPUs, exploiting the data layout discussed in Section 8.2.1. In particular, we adopt the same preprocessing phase detailed by Algorithm 20, with the sole difference that the set of shared variables is represented by all the variables in

[17] We do not explicitly provide $\mathcal{H}_2$ to the kernel, since this information is implicitly included in $\mathcal{H}_1$ and $\mathcal{H}_\oplus$, i.e., $\mathcal{H}_2[i] = \mathcal{H}_\oplus[i] / \mathcal{H}_1[i]$.

[18] The blue rows in Figure 2.7 both refer to the same input row in T_1, hence both the corresponding threads can reuse the same input data.

the scope of the table, excluding the one we want to marginalise. Intuitively, this corresponds to move such a variable to the Most Significant (MS) place. In this case, if we compute the histogram $\mathcal{H}$ and the exclusive prefix sum $\mathcal{P}$ as previously described, we are able to index the groups of rows that must be considered when computing the maximum for the output, denoted by $R_{\Downarrow}$ and $\phi_{\Downarrow}$ for the variable assignment and the value part.

Algorithm 22 shows a simplified version of our actual implementation, in which we generate the appropriate number of threads and blocks on the basis of the size of the input. In contrast with the join sum operation, we do not have any data reuse (i.e., each input row is accessed by exactly one thread), hence the use of shared memory is not necessary. As a final performance remark, notice that the maximisation of the $\mathcal{H}[tx]$ elements at line 5 is sequentially executed by the thread. Nevertheless, this aspect has a negligible impact of the computational throughput of our approach, since $\mathcal{H}[tx]$ depends on the size of the domain of the marginalised variable and, in our experiments, it is usually a small value. When the considered variable has a binary domain, line 5 collapses to one single max operation. However, our approach can be easily extended by devising a parallel reduction operation that implements the marginalisation in the case of variables with bigger domains.

8.3 Data transfers

In the following sections we detail how our preprocessing technique allows an optimised management of data transfers, thanks to full memory coalescence and pipelining (see Section 2.5).

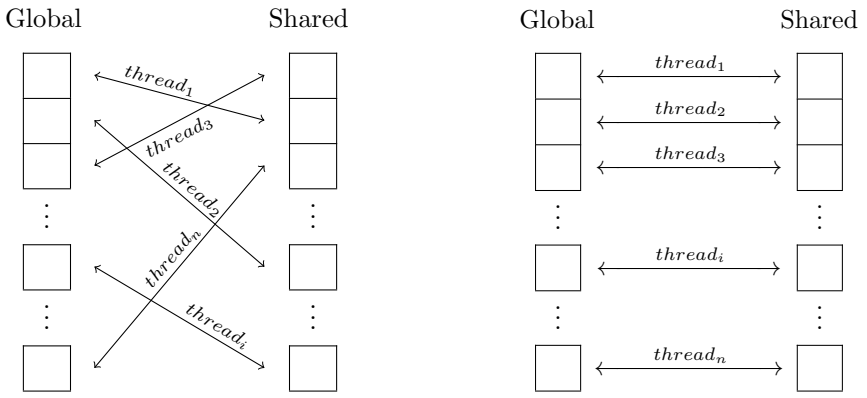
8.3.1 Global-shared memory transfers

Thanks to our preprocessing phase, the portion of input data needed by each thread block is read from global memory with fully coalesced memory accesses, since such data is already organised in consecutive addresses. Transfers are further optimised using *vectorised*[19] memory accesses provided by CUDA architectures to increase bandwidth, reduce instruction count and improve latency.

Algorithm 22 MaximisationIncompleteKernel $(\mathcal{H}, \mathcal{P})$

1: $tx \leftarrow$ thread ID
2: **if** $tx < |\mathcal{H}|$ **then**
3: $\quad idx \leftarrow \mathcal{P}[tx]$
4: $\quad R_{\Downarrow}[tx] \leftarrow R[idx]$ without the column in the MS place
5: $\quad \phi_{\Downarrow}[tx] \leftarrow$ max of the $\mathcal{H}[tx]$ values starting at $\phi[idx]$

[19] Vectorised memory instructions compile to single `LD.E.128` and `ST.E.128` instructions to transfer chunks of 128 bits at a time.



(a) Uncoalesced memory accesses. (b) Coalesced memory accesses.

Fig. 2.33: Uncoalesced vs. coalesced memory accesses.

8.3.2 Host-device data transfers

The table layout presented in Sections 8.1 and 8.2 allows tables to be split into several data segments and threads to independently operate in each segment. This leads to a twofold improvement: i) we devise a pipelined flow of smaller copy-and-compute operations (see Section 2.5.2), by amortising the cost of CPU-GPU data transfers on the overall algorithm performance; ii) we can process tables that do not fit into global memory, by breaking them into more manageable chunks. This allows our approach to perform BE even on problems that were intractable for previous approaches [120].

Large tables processing

Our technique can be applied to execute BE-based algorithm even when potential tables do not fit into the GPU global memory. Tables are split into small data structures, by limiting the maximum number of host-device data transfers that can run concurrently on the GPU. We define max_s as the maximum number of kernels whose total amount of input and output data can be stored into global memory. Figure 8.10 shows an example, in which $max_s = 2$. Each kernel K_i is enqueued in stream $i \bmod max_s$. Transaction $H \rightarrow D_3$ cannot be scheduled in parallel with $D \rightarrow H_2$ (unlike the example of Figure 2.35), as it would violate the above mentioned memory constraint. Thus, one time slot is skipped in order to complete the copy $D \rightarrow H_1$ and to free an adequate amount of memory before starting $H \rightarrow D_3$. The serialisation of these two operations is a direct consequence of their execution in the same stream (i.e., stream 1). Even though the hardware constraints limit the size of data to be transferred and processed, the proposed approach allows oversized tables to be processed in multiple steps, improving scalability. As an example, 5 out of 7 instances in the considered dataset (see Section 8.4.3) cannot be solved without this capability of handling large tables.

Having discussed our implementation of BP and BE on GPUs, we benchmark its performance in our experimental evaluation.

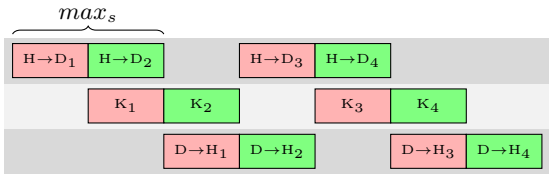


Fig. 8.10: Limited number of streams (best viewed in colour).

8.4 Experimental evaluation

In order to evaluate our approach presented in Sections 8.1 and 8.2, we conducted three different sets of experiments, discussed respectively in Sections 8.4.1, 8.4.2 and 8.4.3. We first test the performance of CUBE when processing complete tables, and specifically when solving BP on JTs and COPs, comparing our algorithm with the most recent GPU approaches in both scenarios [44, 120]. Then, we test our approach that handles incomplete tables, adopting a standard COP dataset [9]. CUBE is implemented in CUDA,[20] and all our experiments are run on a machine with an AMD A8-7600 processor, 16 GB of memory and an NVIDIA Tesla K40.

8.4.1 BP on JTs

In this section we benchmark the approach described in Section 8.1.2, i.e., the one that exploits the completeness of tables to achieve an efficient indexing of potential tables when executing BP on JTs. We compare our approach with the best approach (i.e., the SVR regression model) published by Zheng and Mengshoel [120], i.e., the most recent GPU implementation for BP on JTs, using the authors' implementation. We use the same BN dataset,[21] which comprises various BNs with heterogeneous structures and variable domains. We compile each BN into a JT, which is then used as input for both approaches in order to guarantee a fair comparison. Table 8.1 details some features of our JTs, i.e., the number of junction tree nodes resulting from their compilation and the minimum, maximum and average size of the potential and separator tables. Following Zheng and Mengshoel [120], the compilation of these networks into the corresponding junction trees has been done offline, before the execution of the belief propagation algorithm. For this reason, it has been excluded from the runtime measurements.

	Mildew	**Diabetes**	**Barley**	**Munin$_1$**	**Munin$_2$**	**Munin$_3$**	**Munin$_4$**	**Water**
Number of JT nodes	29	337	36	162	854	904	877	21
Max potential size	1249280	84480	7257600	38400000	151200	156800	448000	589824
Avg potential size	117257	29157	476133	516887	2400	3404	10102	144205
Min potential size	336	495	216	4	4	4	4	9
Max separator size	62464	5280	907200	2400000	6048	22400	56000	147456
Avg separator size	3950	1698	38237	58691	204	528	1376	28527
Min separator size	72	16	7	2	2	2	2	3

Table 8.1: Bayesian Networks.

[20] Our implementation is available at `https://github.com/filippobistaffa/CUBE`.
[21] Available at `http://bndg.cs.aau.dk/html/bayesian_networks.html`.

Table 8.2 reports the runtime in milliseconds corresponding to the following phases of the BP on JTs algorithm:

- The total time required to complete all the reduce and scatter phases in the sequential version.
- The total time required to preprocess all potential tables using our technique.
- The total time required to complete the data transfers between the host and the device.
- The total time required to complete all the reduce and scatter phases in CUBE.
- The GPU speed-up achieved by CUBE.
- The total time required to complete all the reduce and scatter phases in the GPU approach by Zheng and Mengshoel based on the SVR regression model.
- Zheng and Mengshoel's speed-up.

Since the preprocessing phase must be done only once and can be avoided when any new evidence is received and propagated, it has not been considered in the calculation of the speed-up. Moreover, we do not consider the runtimes relative to data transfers in such calculation, as such time is amortised thanks to our pipelining technique (Figure 2.35). For a fair comparison, transfers are also excluded when calculating the speed-up for Zheng and Mengshoel's approach.

In our tests, our algorithm outperforms the counterpart in the majority of the instances, i.e., all except in the Water network, where runtimes are comparable. In more detail, our approach achieves speed-ups at least 56% higher than the counterpart in the Barley dataset (i.e., 33.03× vs 21.14×). Our best improvement with respect to the counterpart happens on the Mildew network, where our approach runs 39× faster than the CPU version, and it produces a GPU speed-up that is the 466% higher than the counterpart. In general, our approach produces speed-ups that increase when the average potential table size increases (see Table 8.1). In fact, we achieve speed-ups less than 10× only with small instances (i.e., Munin_2 and Munin_3).

	Mildew	**Diabetes**	**Barley**	**Munin_1**	**Munin_2**	**Munin_3**	**Munin_4**	**Water**
CPU R/S	117	219	1057	6584	54	109	315	123
Preprocessing	28	71	294	2395	24	37	106	46
Transfers	12	57	110	1464	16	39	45	15
CUBE R/S	3	14	35	193	14	19	31	12
CUBE speed-up	39×	15.64×	33.03×	34.11×	3.85×	5.73×	10.16×	10.25×
SVR R/S	17	34	50	648	48	38	71	14
SVR speed-up	6.88×	6.44×	21.14×	10.16×	1.12×	2.86×	4.43×	8.78×

Table 8.2: BP on JTs results (time values are in milliseconds).

8.4.2 COPs with complete tables

In this section we evaluate the performance of CUBE when solving COPs in which unfeasible assignments are explicitly represented as $-\infty$ values, i.e., the tables are complete. The main goals of this set of experiments are: i) to evaluate the parallel speed-up that CUBE achieves with respect to a sequential version of BE, ii) to

compare CUBE against the most recent approach to parallelise BE on GPU, i.e., the work by Fioretto et al. [44], and iii) to evaluate the scalability of our approach with respect to the size of the problem.

Notice that, even if standard, realistic COP datasets are available [9] (see Section 8.4.3), such instances cannot be represented as COPs with complete tables, as they exceed the memory capabilities of our machine.[22] For this reason, and since we directly compare with Fioretto et al.'s approach, we adopted the same experimental methodology proposed by the authors. In particular, we tested CUBE on 3 different Constraint Network (CN) topologies: i) *random networks* with a graph density of 0.3, ii) *scale-free networks* generated with the Albert and Barabási [1] model using $m = 2$, and iii) *2-dimensional square grid networks*, in which internal nodes are connected to four neighbours, while nodes on the edges (resp. corners) are connected to two (resp. three) neighbours. Each function F_i is generated using uniformly distributed random integer values in $[0, 100]$ and the constraint tightness (i.e., ratio of entries in such tables different from $-\infty$) is set to 0.5 for all experiments. Domain size is 5 for all experiments.

We compared both GPU approaches with FRODO [70], a standard sequential COP solver also considered by Fioretto et al. as baseline benchmark. In particular, we solve all the instances using the DPOP algorithm [87]. To ensure a fair comparison, we run all the algorithms on the same instances and adopting the same variable ordering, i.e., the one produced by FRODO. We consider the entire execution time for all the algorithms, including data transfers.[23] For Fioretto et al.'s approach we use the authors' implementation.

Figures 8.11–8.13 show the speed-up of both GPU approaches with respect to FRODO when increasing the number of variables in the CN. Each data point in the plots represents the average over 20 random instances of the ratio between the runtime required by the GPU approach and FRODO's runtime. The results show that CUBE allows a dramatic runtime reduction with respect to FRODO, by computing the solution at least one order of magnitude faster than the sequential approach in every experiment. In particular, CUBE is, on average, 530 times faster than FRODO when considering the biggest instances in our experiments (i.e., random networks with $n \geq 30$ and scale-free and grid networks with $n \geq 70$), by reaching a maximum speed-up of 652×. More important, such speed-ups increase when the complexity of the problem grows, thus confirming the scalability of CUBE, which correctly exploits the additional degree of parallelism inherent in the problem. In contrast, the speed-up of the approach by Fioretto et al. decreases when the size of the problem increases.

Finally, the results show that the speed-up saturates after a certain number of variables (25 for random networks, 70 for scale-free networks, and 36 for grid networks). This saturation happens when the GPU reaches a full occupancy and it runs the maximum number of concurrent threads (i.e., 30720 for our GPU model). After that, hardware constraints force blocks of threads to run sequentially, hence limiting the maximum speed-up.

[22] Recall that the number of rows of a complete table is exponential with respect to the number of variables in its scope.

[23] We measured that, on average, data transfers take approximately 20% of the entire CUBE runtime.

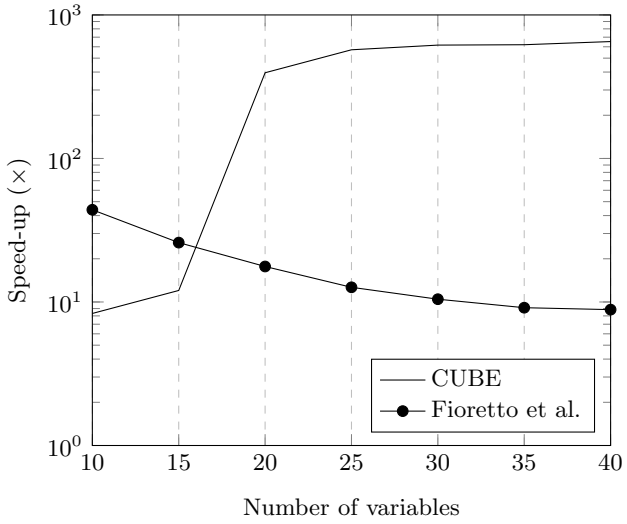


Fig. 8.11: Speed-up on random networks.

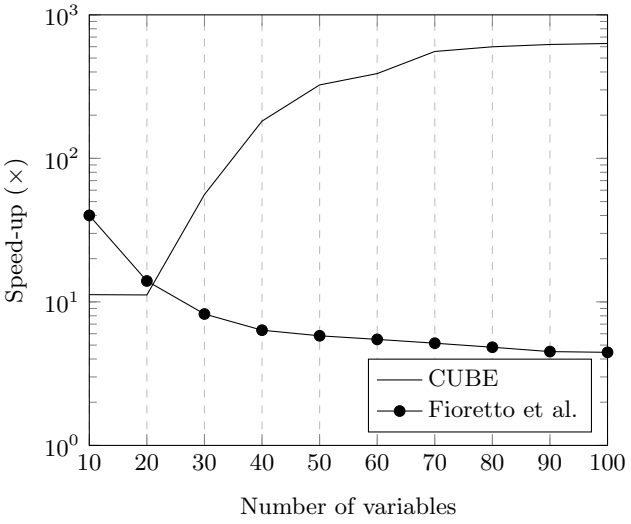


Fig. 8.12: Speed-up on scale-free networks.

8.4.3 COPs with incomplete tables

In our final set of experiments, we benchmark the performance of CUBE when solving COPs involving incomplete tables. To this end, we consider the SPOT5 dataset [9], a standard dataset that models the problem of managing an Earth observing satellite as a Weighted Constraint Satisfaction Problems (WCSPs) [19, 97], to maximise the importance of the captured images, while satisfying some feasibility constraints. WCSPs can be seen as a particular case of COPs, and they

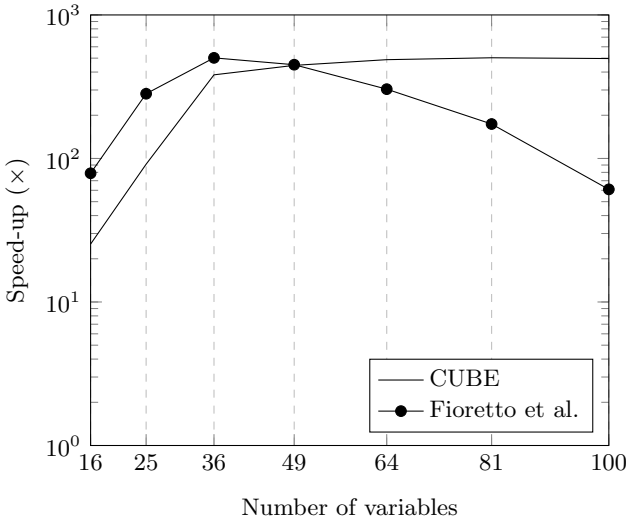


Fig. 8.13: Speed-up on grid networks.

involve incomplete tables, hence requiring the use of the techniques discussed in Section 8.2. The main objective of these experiments is to evaluate the speed-up that can be achieved adopting our parallel approach, which is compared to a completely sequential BE version that uses a naïve implementation for the join sum and the marginalisation operations. This choice is motivated by the fact that, to the best of our knowledge, none of the parallel implementations for the $\oplus$ and $\Downarrow$ operators available in the literature can handle incomplete tables. In addition, representing these instances with complete tables is not feasible as they exceed the memory capabilities of our machine.[24]

Table 8.3 shows the runtime in seconds (including preprocessing and data transfers) needed to solve the instances of our reference domain by CUBE compared to its sequential version, i.e., BE. Such table also reports the number of variables and the induced width of these instances. The results show that CUBE provides a speed-up of at least 2 orders of magnitude with respect to the sequential algorithm, by reaching a maximum of 696.02×. Such speed-up increases consistently with the size of the instances (i.e., the induced width and the number of variables), showing that the proposed approach correctly exploits the increased amount of parallelism in bigger tables. In fact, the speed-up provided by CUBE monotonically increases in the first three WCSP instances (i.e., 54, 29 and 404), in which both the number of variables and the induced width increase. On the other hand, such speed-up decreases in instance 503, which, despite having a larger number of variables, is characterised by a lower induced width. Recall that the induced width encodes the complexity of the problem (see Proposition 2.15). The ability of our method of handling large tables is crucial in this scenario. In fact, 5 out of 7 instances (i.e., 404, 503, 42b, 505b and 408b) cannot be solved without this feature, as their tables exceed the amount of GPU memory.

[24] For this reason, we could not adopt the SPOT5 dataset for the tests in Section 8.4.2.

	54	**29**	**404**	**503**	**42b**	**505b**	**408b**
Variables	67	82	100	143	190	240	200
Induced Width	11	14	19	9	18	16	24
CUBE Runtime	3.01	5.36	12.40	17.46	58.42	77.17	120.79
BE Runtime	965.66	2656.72	7584.12	6347.98	31637.91	53710.41	76456.15
GPU speed-up	321.03×	495.38×	611.67×	363.63×	541.55×	696.02×	632.97×

Table 8.3: WCSPs results (time values are in seconds).

9

A COP model for Graph-Constrained Coalition Formation

In this chapter we propose COP-GCCF, a novel formalisation of the GCCF problem, which allows us to employ the CUBE algorithm discussed in Chapter 8. By doing so, we aim at exploiting the benefits of GPU parallelisation for the solution of such problem. Furthermore, COP-GCCF allows us to deal with some limitations of the approaches proposed in Chapters 4 and 6. On the one hand, DFS-based approaches are difficult to parallelise [92], as discussed in Chapter 8. On the other hand, COP-GCCF does not require any assumption on the characteristic function, hence allowing us to solve completely general GCCF instances.

Now, as detailed in Section 2.1, GCCF is essentially an optimisation problem (aiming at maximising the sum of the coalitional values) subject to feasibility constraints (i.e., coalitions must be feasible and disjoint). Nonetheless, to the best of our knowledge none of the constrained optimisation techniques present in the literature [31] has ever been applied to GCCF.

Against this background, COP-GCCF is the first approach that models GCCF as a Constrained Optimisation Problem (COP). Within COP-GCCF, we exploit the structure of the graph so to achieve a model of manageable complexity. Specifically, we propose a formalisation of the COP that builds a hierarchy of agents resulting in a linear number of constraints (with respect to the number of agents). Furthermore, our model is devised to exploit the capability of CUBE of processing incomplete tables (see Section 8.2), allowing us to avoid the explicit representation of unfeasible configuration, hence achieving an improved memory footprint. This feature is crucial for the solution of COPs involving large tables (such as COP-GCCF), which would not be tractable if represented adopting complete tables.

9.1 Variables

Our COP model for the GCCF problem comprises $|\mathcal{FC}(G)|$ *binary* variables, i.e., one per feasible coalition. Formally, $X = \{x_S \mid S \in \mathcal{FC}(G)\}$. The computation of X involves the enumeration of all the subgraphs of G and can be solved using one of the existing techniques in the literature [105, 115]. We use the SlyCE algorithm by Voice et al. [115], which also provide a parallelised version, i.e., D-SlyCE.

9.2 The simple approach

The most simple and natural approach to formalise GCCF as a COP would be to represent each feasible coalition structure $CS \in \mathcal{CS}(G)$ with a variable assignment in which $x_S = 1$ if and only if $S \in CS$, and map such an assignment to a value equal to $V(CS)$. Intuitively, each x_S set to 1 enables the corresponding coalition S, and adds $v(S)$ to the value of the assignment. This approach requires to specify a number of constraints in order to allow *only* the variable assignments that correspond to valid partitions of A, i.e., valid solutions of the GCCF problem. Specifically, a set of coalitions $\mathcal{S}$ is a valid partition of A if each agent is part of *exactly one* coalition, or equivalently:

Property 9.1. There are no overlapping coalitions in $\mathcal{S}$, i.e., $\nexists S, S' \in \mathcal{S} \colon S \cap S' \neq \emptyset$.

Property 9.2. Each agent in A is part of a coalition in $\mathcal{S}$, i.e., $\bigcup_{S \in \mathcal{S}} = A$.

Despite being a valid representation of the GCCF problem, the above COP would be impractical to our purposes, as it would result in a very large number of constraints, an hence, would be too complex[1] to solve with existing COP techniques, including CUBE. Specifically, Property 9.1 would require one binary constraint for each couple of variables in X, i.e., ${}^{|X| \cdot (|X|-1)}/_{2}$ binary constraints, each verifying that the variables in its scope are not both set to 1 if the corresponding coalitions overlap, while Property 9.2 would lead to $2^{|X|}$ n-ary constraints, i.e., one for each subset of variables controlling that the corresponding coalitions contain all the agents in A. We now propose a different way to enforce Properties 9.1 and 9.2, which results in a manageable number of constraints.

9.3 COP-GCCF

In this section we detail COP-GCCF, our COP formalisation of the GCCF problem. In the remainder of this chapter, we discuss our approach referring to the example in Figure 9.1. Within COP-GCCF, we exploit the structure of the graph G in order to reduce the number of constraints necessary to ensure the correctness of the model. First, we construct a *pseudotree* $PT(G)$ [87] from G, establishing a partial order among the agents in A.

Definition 9.3 (pseudotree). *A pseudotree $PT(G)$ of a graph G is a rooted tree with the same nodes as G and the property that adjacent nodes from the original graph fall in the same branch of $PT(G)$.*

$PT(G)$ is a Directed Acyclic Graph, in which edges are directed from children nodes to parent ones. Back-edges, i.e., edges present in G but removed from its tree representation, are marked with a dashed line. Then, we partition the set of variables X in n sets X_i, each corresponding to $a_i \in A$, such that

$$X_i = \{x_S \mid S \text{ contains only } a_i \text{ or its descendants in } PT(G)\}.$$

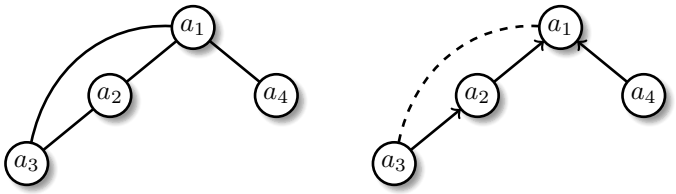


Fig. 9.1: Example graph G and the corresponding $PT(G)$.

Each X_i represents the set of *local* variables of the agent a_i. Here, $X_1 = \{x_1, x_{12}, x_{13}, x_{123}, x_{14}, x_{124}, x_{134}, x_{1234}\}$, $X_2 = \{x_2, x_{23}\}$, $X_3 = \{x_3\}$, and $X_4 = \{x_4\}$.

As stated above, Properties 9.1 and 9.2 must be ensured in order to correctly represent the GCCF problem. Property 9.1 requires that the activation of a particular variable/coalition *excludes* the activation of incompatible variables, i.e., variables whose concurrent activation would generate overlapping coalitions. Now, within COP-GCCF we enforce Property 9.1 among the variables in X_i, i.e., the variables local to the agent a_i, by constructing n constraint functions F_i (each associated to X_i), and allowing only the assignments in which exactly one local variable is activated (see Section 9.3.2). On the other hand, Property 9.1 cannot be directly enforced for variables that are local to different agents, i.e., that belong to different X_i, and hence, to different constraint functions. Notice that introducing additional binary constraints between overlapping variables would lead to the above discussed simple approach, resulting in $^{|X|\cdot(|X|-1)}/_{2}$ constraints. In contrast, we achieve this exploiting the concept of *required variables*.

9.3.1 Required variables

The main idea behind *required variables* is that the formation of a variable/coalition $x_S \in X_i$ can be achieved exploiting the hierarchy induced by $PT(G)$. Intuitively, the agent a_i can negotiate the formation process only with its children nodes, allowing a more succinct representation of the problem and saving computational resources. As an example, x_{1234} (local to a_1) requires the participation of a_2, a_3, and a_4, but a_1 can force the participation of a_3 through a_2. In other words, x_{1234} *requires* x_4 and x_{23}, which indirectly *requires* x_3 through a_2.

Formally, we represent such dependencies with the *requires* relation, denoted as $req(PT(G)) \subseteq X^2$, i.e., a set of couples of variables. Figure 9.2 illustrates such relation corresponding to the above example. Intuitively, if $(x_S, x_{S'}) \in req(PT(G))$, it means that x_S requires $x_{S'}$. In terms of variable assignments, the *requires* relation acts as an implication, i.e., $x_S = 1 \implies x_{S'} = 1$. Notice that, as a consequence, if x_S requires $x_{S'}$ and $x_{S'}$ requires $x_{S''}$, the activation of x_S results in the activation of $x_{S''}$, i.e.,

$$\forall x_S, x_{S'}, x_{S''} \in X : (x_S, x_{S'}) \in req(PT(G)) \text{ and } (x_{S'}, x_{S''}) \in req(PT(G)), \\ x_S = 1 \implies x_{S''} = 1. \quad (9.1)$$

[1] The number of constraints is closely related to the induced width of the constraint network, which encodes its complexity (see Proposition 2.15).

Due to the above property, *requires* exhibits a property similar to *transitivity*. On the other hand, we construct such a relation in a way such that if $(x_S, x_{S'}) \in req\,(PT\,(G))$ and $(x'_S, x_{S''}) \in req\,(PT\,(G))$, then $(x_S, x_{S''}) \notin req\,(PT\,(G))$, hence, strictly speaking, *requires* is *not* a transitive relation. This choice is motivated by the fact that the amount of required variables directly determines the scopes of the constraints within COP-GCCF (see Section 9.3.2), hence including the additional couple $(x_S, x_{S''})$ in $req\,(PT\,(G))$ would be redundant, since the relation between such variables is still expressed by the property in Equation 9.1.

Definition 9.4 (indirect requirement). *A variable* $x_S \in X$ indirectly requires $x_{S''} \in X$ *if* $\exists x_{S'} \in X$ *such that* $(x_S, x_{S'}) \in req\,(PT\,(G))$ *and* $(x'_S, x_{S''}) \in req\,(PT\,(G))$.

Notice that any required variable $x_{S'}$ activated as a result of the *requires* relation does *not* correspond to the formation of the coalition S'.

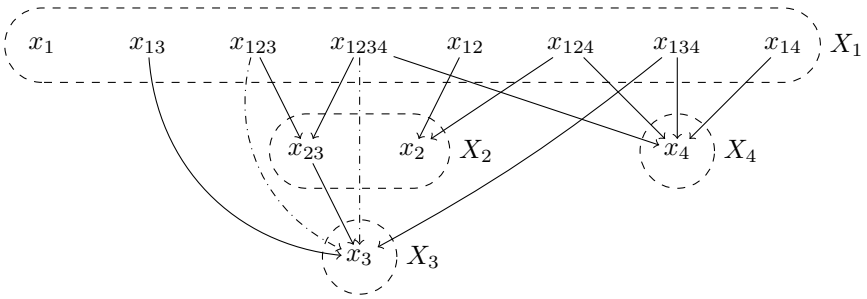


Fig. 9.2: The *requires* relation (indirect requirements drawn as dash-dotted lines).

Given an agent a_i and a local variable x_S representing the coalition S, its required variables are computed with Algorithm 23, which iterates over the children of a_i (line 2) and computes the required variables relative to each child with the recursive routine in Algorithm 24. As an example, we compute the variables required by x_{1234} with such algorithms. The first iteration of the loop in Algorithm 23 refers to the first child of a_1, i.e., a_2. Within the corresponding invocation of RecReq, $S' = \{a_2, a_3\}$, resulting in the required variable x_{23}. Notice that the inner RecReq call returns $\emptyset$, since $\{a_1, a_4\} \cap PT_3 = \emptyset$. Similarly, the second iteration of the loop (i.e., the one for a_4) yields the required variable x_4. Note that x_{1234} indirectly requires x_3, since x_{23} requires such variable.

Algorithm 23 ComputeReq $(S, a_i, PT\,(G))$

1: $req_S \leftarrow \emptyset$ {Initialise empty set of required variables}
2: **for all** a_j children of a_i **do**
3: $\quad req_S \leftarrow req_S \cup$ RecReq $(S, a_j, PT\,(G))$
4: **return** req_S

Algorithm 24 RecReq $(S, a_j, PT(G))$

1: $req_S^j \leftarrow \emptyset$
2: $PT_j \leftarrow$ agents in the subtree of $PT(G)$ rooted in a_j
3: **if** $S \cap PT_j = \emptyset$ **then**
4: **return** $\emptyset$
5: **else**
6: $S^* = \arg\max_{\{x_{\bar{S}} \in X_j \mid \bar{S} \subseteq (S \cap PT_j)\}} \left|\bar{S} \cap S\right|$
7: {x_{S^*} is the variable in X_j, strictly composed of agents}
8: {in $S \cap PT_j$, with the maximum intersection with S}
9: **for all** a_k children of a_j **do**
10: $req_S^j \leftarrow req_S^j \cup \{x_{S'}\} \cup$ RecReq $(S \setminus S^*, a_k, PT(G))$
11: **return** req_S^j

The fundamental characteristic of required variables is that any two variables that require the same variable *cannot* be enabled simultaneously. Generally, since we cannot activate two variables both local to the same agent, two variables that require variables local to the same agent cannot be active at the same time, i.e.,

$$\forall x_S, x_{S'} \in X, a_k \in A : \exists x_{S''}, x_{S'''} \in X_k : (x_S, x_{S''}) \in req(PT(G)) \\ \text{and } (x_{S'}, x_{S'''}) \in req(PT(G)) \implies \neg(x_S \wedge x_{S'}). \quad (9.2)$$

By enforcing Equation 9.2, we ensure that no overlapping variables local to different agents are activated at the same time. Proposition 9.6 proves that such property holds within COP-GCCF.

As background for such proof, we first prove Lemma 9.5, that ensures that any variable, local to an agent a_i, and involving an agent a_k descendant of a_i, requires a variable local to a_k, possibly by indirect requirement. In the example in Figure 9.2, x_{13} (local to a_1) corresponds to a coalition containing a_3, which is a descendant of a_1. As a consequence, x_{13} requires a variable local to a_3, i.e., x_3. On the other hand, x_{123} indirectly requires x_3 through x_{23}.

Lemma 9.5. *Given a variable $x_S \in X_i$, i.e., local to a_i, such that $a_k \in S$, where a_k is a descendant of a_i in $PT(G)$, it exists a variable $x_{S'} \in X_k$, i.e., local to a_k, such that x_S requires $x_{S'}$, possibly by indirect requirement.*

Proof. Let a_j be the child of a_i such that $a_k \in PT_j$, i.e., a_k is a descendant of a_j, and consider the routine call RecReq $(S, a_i, PT(G))$ at line 3 of Algorithm 23 corresponding to a_j. We now show that RecReq $(S, a_j, PT(G))$ either falls into a base case (directly proving this lemma), or into a recursive one. In such case, we show that one of the inner recursive calls still verifies the hypotheses of this lemma, hence recursively applying this proof to such call.

- **Base case ($a_j = a_k$)**: line 6 of RecReq $(S, a_j, PT(G))$ results in a coalition S^* containing a_k as the highest agent in such coalition, considering the hierarchy induced by $PT(G)$. Hence, x_S requires $x_{S'} = x_{S^*}$, local to a_k.

- **Recursive case ($a_j \neq a_k$)**: we distinguish between two cases:
  - $a_j \in S$: line 6 of RecReq $(S, a_j, PT(G))$ results in a coalition S^* containing both a_j and a_k, since $a_j \in S$ and $a_k \in PT_j$. Notice that $S \setminus S^*$ at line 10 does not contain a_k. Hence, x_S requires $x_{S'} = x_{S^*}$, which is *not* local to a_k, but to a_j. On the other hand, the recursive procedure that computes the variables required by $x_{S'} = x_{S^*}$ verifies the hypotheses of this lemma, since $a_k \in S^*$ and a_k is a descendant of a_j.
  - $a_j \notin S$: line 6 of RecReq $(S, a_j, PT(G))$ results in $\emptyset$. In fact, since $a_j \notin S$, then $a_j \notin S \cap PT_j$, and line 6 only considers coalitions local to a_j (which all contain a_j) *strictly* composed of agents in $S \cap PT_j$, which does not contain a_j. Then, line 9 (which iterates over the children of a_j) contains one iteration referring to a child of a_j who still is an ancestor of a_k. Such iteration recursively calls RecReq with the same S, since $S^* = \emptyset$. Thus, the hypotheses of the current lemma are verified.

Notice that both cases in the recursive step refer to agents a_j (all ancestors of a_k) that are gradually closer to a_k. Thus, the base case is eventually executed. □

Proposition 9.6. *Overlapping variables local to different agents cannot be activated at the same time, i.e., $\forall x_S \in X_i, x_{S'} \in X_j$ such that $a_i \neq a_j$, if $S \cap S' \neq \emptyset$, then $\neg(x_S \wedge x_{S'})$.*

Proof. Let a_k be an agent in $S \cap S'$. Notice that a_k always exists since $S \cap S' \neq \emptyset$. As a direct consequence of how $PT(G)$ is constructed, $a_k \in PT_i$ and $a_k \in PT_j$, where PT_h represents the set of agents in the subtree of $PT(G)$ rooted in a_h. Now, since $PT(G)$ is a tree, it follows that either $a_i \in PT_j$ or $a_j \in PT_i$. Without loss of generality, assume that a_i is an ancestor of a_j, i.e., $a_j \in PT_i$, and thus, a_k is a descendant of both a_i and a_j. By applying Lemma 9.5 to x_S and $x_{S'}$, it follows that x_S and $x_{S'}$ require variables both local to a_k. Finally, Equation 9.2 ensures that $\neg(x_S \wedge x_{S'})$. □

As an example, our technique ensures that x_{13} and x_{23} cannot be both set to 1, since they both require x_3. In the next section, we show how to implement n constraint functions that realise the above discussed concepts.

9.3.2 Constructing constraint functions

As mentioned in Section 9.3, COP-GCCF involves n constraint functions F_i, one for each agent a_i. Such an amount is dramatically lower with respect to the simple approach previously discussed, i.e., n vs. $2^{|X|} + {}^{|X| \cdot (|X|-1)}/_{2}$ (cf. Section 9.2). Each F_i is constructed according to the following definition.

Definition 9.7 (F_i). *Each F_i is responsible for the variables local to a_i, hence we initialise the scope Q_i of each F_i as $Q_i = X_i$. Then, to represent the* requires *relation, we include all the non-local variables that require a variable in X_i. Formally,*

$$Q_i = X_i \cup \{x_S \mid \exists x_{S'} \in X_i : (x_S, x_{S'}) \in req(PT(G))\}.$$

The scope Q_i of each F_i comprises X_i, i.e., the variables local to a_i, plus all the non-local variables that require a variable in X_i. Each F_i contains $|Q_i|$ feasible

assignments, i.e., one for each variable in the scope. Notice that, in our model, we do not explicitly represent unfeasible assignments, since CUBE can process incomplete tables, resulting in reduced memory requirements (see Section 8.2). The variable assignment in each row is constructed by activating the corresponding variable, namely x_S. If x_S is non-local, i.e., $x_S \notin X_i$, we also activate the variable required by x_S. Then, for each assignment in which a local variable $x_S \in X_i$ is activated, we define the corresponding value equal to $v(S)$, while such value is 0 when a non-local variable is considered. This avoids the duplication of $v(S)$ when x_S is propagated as a non-local variable across the constraint functions.

In contrast with CFSS (see Chapters 4 and 6), COP-GCCF does not make any assumption on the characteristic function, allowing us to solve completely GCCF instances. Figures 9.3–9.6 show the constraint functions (with non-local variables highlighted in grey) corresponding to the example in Figure 9.1. Notice that, at the moment, our model propagates several variables down the pseudotree. As an example, X_4 contains only one variable, but $Q_4 = X_4 \cup \{x_{1234}, x_{124}, x_{134}, x_{14}\}$. We will show how to reduce this amount in the next section.

We now formally prove that COP-GCCF is correct, i.e., that the optimal solution of COP-GCCF is the optimal solution of the corresponding GCCF problem (Theorem 9.10). First, we prove two necessary lemmas for such theorem.

Lemma 9.8. *Property 9.1 holds within COP-GCCF.*

Proof. Definition 9.7 guarantees that exactly one local variable is activated, and that overlapping variables local to different agents are not enabled at the same

	x_1	x_{12}	x_{13}	x_{14}	x_{123}	x_{124}	x_{134}	x_{1234}	Value
Local	1	0	0	0	0	0	0	0	$v(\{a_1\})$
	0	1	0	0	0	0	0	0	$v(\{a_1, a_2\})$
	0	0	1	0	0	0	0	0	$v(\{a_1, a_3\})$
	0	0	0	1	0	0	0	0	$v(\{a_1, a_4\})$
	0	0	0	0	1	0	0	0	$v(\{a_1, a_2, a_3\})$
	0	0	0	0	0	1	0	0	$v(\{a_1, a_2, a_4\})$
	0	0	0	0	0	0	1	0	$v(\{a_1, a_3, a_4\})$
	0	0	0	0	0	0	0	1	$v(\{a_1, a_2, a_3, a_4\})$

Fig. 9.3: Constraint function F_1.

	x_2	x_{23}	x_{12}	x_{124}	x_{123}	x_{1234}	Value
Local	1	0	0	0	0	0	$v(\{a_2\})$
	0	1	0	0	0	0	$v(\{a_2, a_3\})$
Non-local	1	0	1	0	0	0	0
	1	0	0	1	0	0	0
	0	1	0	0	1	0	0
	0	1	0	0	0	1	0

Fig. 9.4: Constraint function F_2.

	x_3	x_{134}	x_{13}	x_{23}	Value
Local	1	0	0	0	$v(\{a_3\})$
Non-local	1	1	0	0	0
	1	0	1	0	0
	1	0	0	1	0

Fig. 9.5: Constraint function F_3.

	x_4	x_{1234}	x_{124}	x_{134}	x_{14}	Value
Local	1	0	0	0	0	$v(\{a_4\})$
Non-local	1	1	0	0	0	0
	1	0	1	0	0	0
	1	0	0	1	0	0
	1	0	0	0	1	0

Fig. 9.6: Constraint function F_4.

time, as a consequence of Proposition 9.6. As such, Property 9.1 holds within COP-GCCF. □

Lemma 9.9. *Property 9.2 holds within COP-GCCF.*

Proof. Each F_i only contains variable assignments in which exactly one local variable is enabled. As such, assignments in which a_i is not part of any coalition, i.e., the ones violating Property 9.2, are unfeasible and cannot be a solution. □

Theorem 9.10. *The optimal solution of COP-GCCF is the optimal solution of the corresponding GCCF problem.*

Proof. COP-GCCF ensures that the variable assignment produced as solution satisfies Properties 9.1 and 9.2. Furthermore, such an assignment maximises the sum of the values of the constraints (see Definition 2.9), which, in COP-GCCF, is the sum of the values of the corresponding coalitions. As such, the solution of COP-GCCF correctly represents the solution of the GCCF problem. □

In what follows, we further improve our model by discussing how to reduce the scope of constraint functions, so to improve the memory requirements of our approach while maintaining its correctness.

9.3.3 Reducing the size of constraint functions

Our method of constructing each F_i involves the addition to its scope of every non-local variable that requires a local one. Now, line 6 of Algorithm 24 implies that the set of required variables of a particular variable x_S local to the agent a_i contains variables that are local to a_i's descendants, i.e., agents lower than a_i in the hierarchy induced by $PT(G)$. As a consequence, the *requires* relation never involve variables that are local to the same agent. In contrast, if we could express the same dependencies by using variables that are also local to a_i, we could reduce the amount of variables added to the scope of F_i, and hence, its size.

In our improved model, we achieve this by introducing a slight modification in how the *requires* relation is expressed. As an example, notice that x_{1234}, local to a_1, requires x_{23} and x_4 in our original model, which belong to different branches in $PT(G)$ (Figure 9.7a). When this happens, we can augment each required variable adding the considered agent (i.e., a_1 in this example), obtaining x_{123} and x_{14} as new required variables (Figure 9.7b).

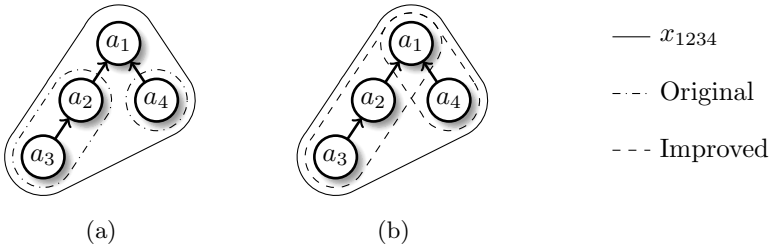


Fig. 9.7: Required variables for x_{1234} (original vs. improved).

This method is applicable only when we have more than one required variable. In fact, if we applied it to the only required variable of x_{123}, i.e., x_{23}, we would re-obtain x_{123}. This modification allows us to greatly reduce the scope of the constraint functions in COP-GCCF. In fact, notice that the new required variables of x_{1234}, i.e., x_{123} and x_{14}, are both local to a_1, in contrast with the original ones that were local to a_2 and a_4. Thus, we can avoid adding x_{1234} to the scope of F_2 and F_4, since it no longer requires x_{23} and x_4. Furthermore, this improvement does not affect the correctness of our model, since the new *requires* relation is equivalent to the original one, as proven by the following proposition.

Proposition 9.11. *The* requires *relation obtained with our improved technique is equivalent to the original one.*

Proof. Let $x_S \in X_i$ be a variable local to a_i, and $x_{S'} \in X_j$ a variable local to a_j child of a_i, such that x_S requires $x_{S'}$ in our original model. Assume that, in our improved model, $(x_S, x_{S''}) \in req(PT(G))$, where $S'' = \{a_i\} \cup S'$ is equal to S' augmented with a_i, as a consequence of our improved technique of constructing required variables. Notice that, since S'' contains $\{a_i\}$, it is local to such agent. Furthermore, S'' cannot contain agents (apart from a_i) that are not part of PT_j, i.e., agents outside the subtree of $PT(G)$ rooted in a_j. Hence, a_j is the only child of a_i that does not result in $\emptyset$ at line 3 of Algorithm 24 when we construct the required variables for $x_{S''}$. Instead, the invocation of RecReq$(S'', a_j, PT(G))$ yields $S^* = S'$ as the result of the operation at line 6, since S' is precisely the coalition, strictly composed of agents in $S'' \cap PT_j$, with the maximum intersection with S''. Thus, $(x_{S''}, x_{S'}) \in req(PT(G))$. Now, since x_S requires $x_{S''}$, and $x_{S''}$ requires $x_{S'}$, then x_S indirectly requires $x_{S'}$ (see Definition 9.4) in our improved model. Therefore, the *requires* relation obtained with our improved technique is equivalent to the original one. □

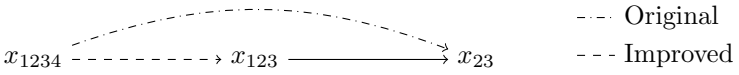


Fig. 9.8: Relationships among x_{1234}, x_{123}, and x_{23} (original vs. improved).

As an example, Figure 9.8 shows that we indirectly achieve the original dependency between x_{1234} and x_{23} by means of a dependency with x_{123}, since the relation between x_{123} and x_{23} is unchanged. Figures 9.9–9.12 show the constraint functions obtained with our improved model considering the above example. It is clear that our technique allows to greatly reduce the number of non-local variables (cf. Figures 9.3–9.6), reducing the total number of columns from 23 to 17 (i.e., −26%).

	x_1	x_{12}	x_{13}	x_{14}	x_{123}	x_{124}	x_{134}	x_{1234}	Value
Local	1	0	0	0	0	0	0	0	$v(\{a_1\})$
	0	1	0	0	0	0	0	0	$v(\{a_1, a_2\})$
	0	0	1	0	0	0	0	0	$v(\{a_1, a_3\})$
	0	0	0	1	0	0	0	0	$v(\{a_1, a_4\})$
	0	0	0	0	1	0	0	0	$v(\{a_1, a_2, a_3\})$
	0	1	0	1	0	1	0	0	$v(\{a_1, a_2, a_4\})$
	0	0	1	1	0	0	1	0	$v(\{a_1, a_3, a_4\})$
	0	0	0	1	1	0	0	1	$v(\{a_1, a_2, a_3, a_4\})$

Fig. 9.9: Improved constraint function F_1.

	x_2	x_{23}	x_{12}	x_{123}	Value
Local	1	0	0	0	$v(\{a_2\})$
	0	1	0	0	$v(\{a_2, a_3\})$
Non-local	1	0	1	0	0
	0	1	0	1	0

Fig. 9.10: Improved constraint function F_2.

	x_3	x_{13}	x_{23}	Value
Local	1	0	0	$v(\{a_3\})$
Non-local	1	1	0	0
	1	0	1	0

Fig. 9.11: Improved constraint function F_3.

	x_4	x_{14}	Value
Local	1	0	$v(\{a_4\})$
Non-local	1	1	0

Fig. 9.12: Improved constraint function F_4.

9.4 Experimental evaluation

The main goals of our empirical analysis are i) to evaluate the performance of COP-GCCF in terms of runtime and memory requirements and ii) to compare it with DyCE, i.e., the state of the art algorithm to solve general GCCF, and with IDP^G, i.e., the only GPU implementation of an algorithm that solves CSG.

9.4.1 Experimental methodology

Following Voice et al. [115] and our experiments in Sections 5.4 and 6.4, we generate random GCCF instances considering three different network topologies, i.e., scale-free networks obtained with the Albert and Barabási [1] model with the m parameter equal to 1 and 2, and subgraphs of a large crawl of the Twitter social graph [67]. Similarly to Sections 5.4 and 6.4, G is obtained by means of a breadth-first traversal starting from a random node of the whole graph, adding each node and the corresponding arcs to G, until the desired number of nodes is reached [94]. Each feasible coalition has an uniformly distributed random value within $[-10, 10]$, while, for IDP^G, unfeasible coalitions have a value of $-\infty$.

We vary n within $[20, 30]$,[2] generating 20 random instances for each of the above network topologies and solving each of them with the three considered algorithms. For each n, we report the average and the standard error of the mean of such 20 repetitions. All our experiments are run on a machine with a 3.40GHz CPU, 16GB of memory and a GeForce GTX 680 GPU. For DyCE and IDP^G, we used the implementations provided by the respective authors.[3]

9.4.2 Runtime

In order to fully understand our results, it is important to notice that the complexity of any GCCF problem is significantly influenced by the density of the graph G, as a larger number of connections results in a larger number of feasible coalitions, and, therefore, a larger solution space. For scale-free networks, density is directly determined by the parameter m, which represents the number of edges incident to each newly added node using the Barabási-Albert generation model. For Twitter subgraphs, we verified that such topology results in a density equivalent to a scale-free network with $1 < m < 2$.

Figures 9.13–9.15 show the runtime needed to solve the GCCF problems considering the above discussed network topologies. Our approach outperforms DyCE and IDP^G on scale-free networks with $m = 1$ and Twitter subgraphs, computing solutions up to 4 orders of magnitude faster than counterparts in the former scenario, and 1 order of magnitude faster than IDP^G in the latter one. For scale-free networks with $m = 2$, our approach is one order of magnitude slower than IDP^G, but it computes solutions 2 times faster than DyCE for 30 agents. In general, our results show that COP-GCCF outperforms DyCE in all the considered network topologies, and it is one order of magnitude faster than IDP^G when considering a realistic dataset, i.e., Twitter.

[2] DyCE and IDP^G cannot solve instances larger than 30 agents due to their exponential memory requirements.

[3] The implementation of IDP^G is available at `https://github.com/idpg/idpg`.

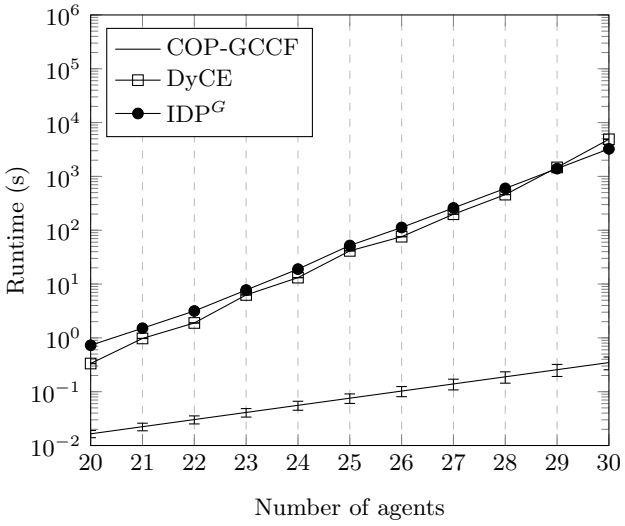


Fig. 9.13: Runtime for scale-free networks with $m = 1$.

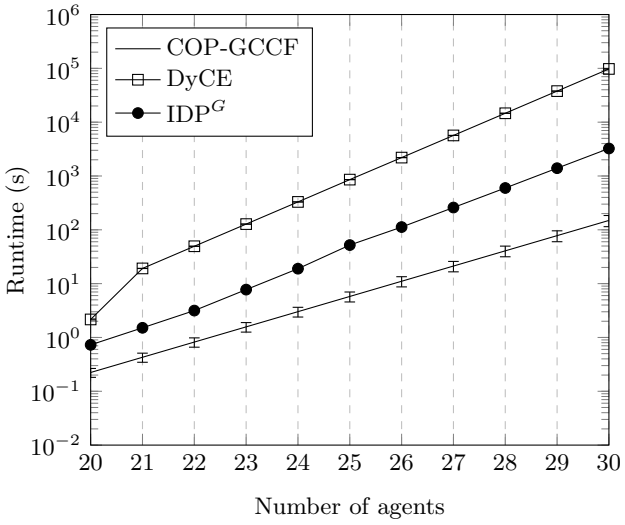


Fig. 9.14: Runtime for Twitter subgraphs.

9.4.3 Memory

Figures 9.16–9.18 show the memory requirements of the considered approaches. For COP-GCCF, we measure the size of the largest table generated during the entire execution of the algorithm, while the memory requirements of DyCE and IDP^G are both $\Theta(2^n)$, regardless of the network topology. Specifically, DyCE and IDP^G require $4 \cdot 2^n$ bytes, since coalitional values are stored as `float` values.

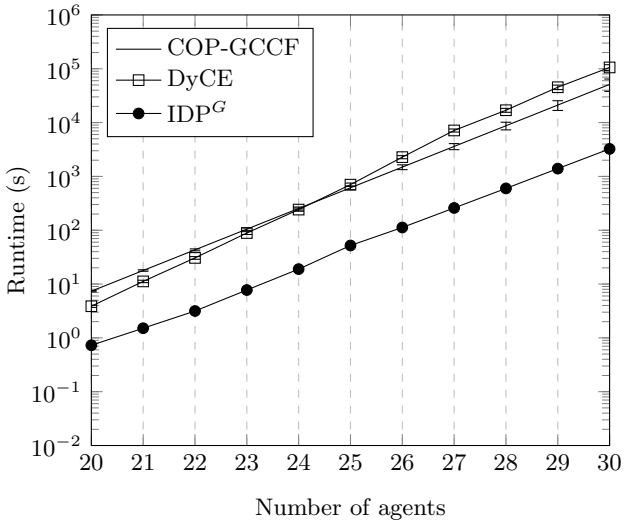


Fig. 9.15: Runtime for scale-free networks with $m = 2$.

Our results follow the behaviour discussed in the previous section, i.e., the memory requirements of COP-GCCF are lower with respect to DyCE and IDP^G for scale-free networks with $m = 1$ and Twitter subgraphs. Specifically, the memory consumption of our approach is 2 orders of magnitude lower in the former case, and 1 order of magnitude lower in the latter one. For scale-free networks with $m = 2$, COP-GCCF requires twice as much memory with respect to DyCE and IDP^G, due to the higher density of G that results in a larger number of variables.

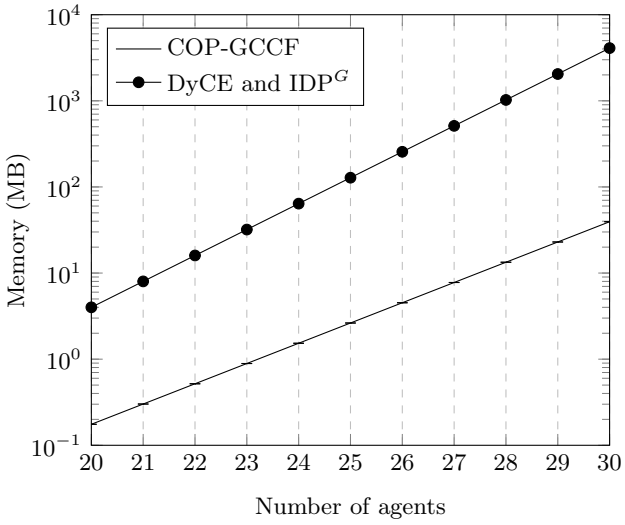


Fig. 9.16: Memory for scale-free networks with $m = 1$.

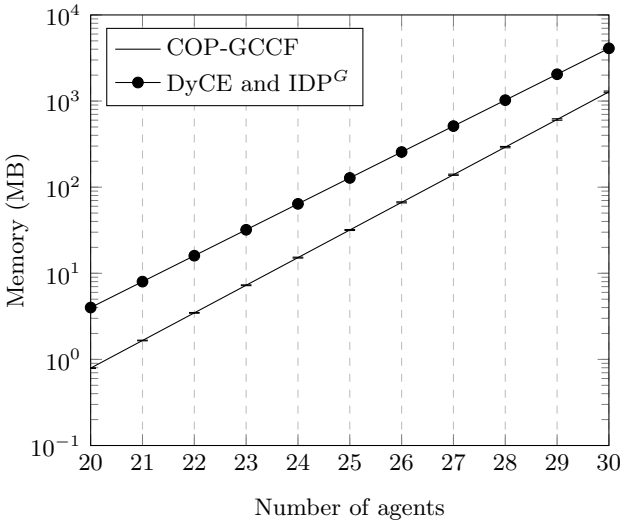


Fig. 9.17: Memory for Twitter subgraphs.

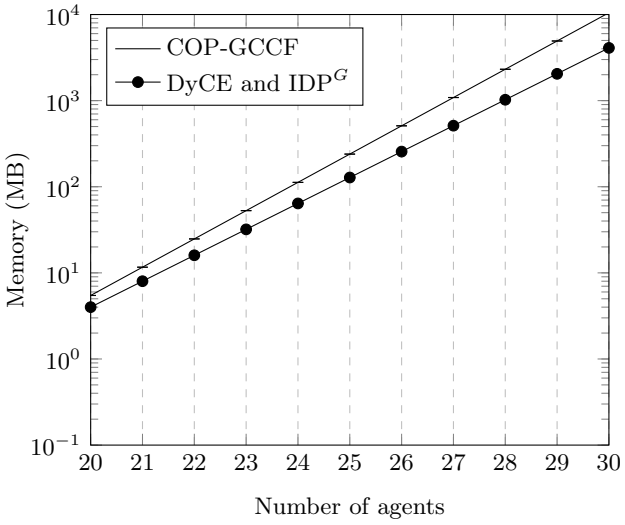


Fig. 9.18: Memory for scale-free networks with $m = 2$.

10

Conclusions and future work

This thesis proposes novel solutions for several interesting problems in the field of multi-agent systems, especially in the context of Graph-Constrained Coalition Formation (GCCF). So far, such problem has never been applied to realistic contexts, and only on small-scale problems. For the first time, we show that GCCF is a viable approach method for important real-world scenarios such as collective energy purchasing and ridesharing, which are receiving a considerable amount of focus within the AI community. Our solutions comprise both search and dynamic programming techniques, in order to employ the best solution technique on the basis of the nature of the characteristic function. In the former case, the proposed approaches overcome the limitations of previous techniques, solving problems of unprecedented scale. In the context of dynamic programming, we propose a new parallelisation scheme that allows to benefit from the computational capabilities of Graphics Processing Units (GPUs) in the solution of hard combinatorial optimisation problems. Furthermore, our contributions establish a clear link between GCCF and COPs, which, to the best of our knowledge, has never been investigated in the literature, and opens to the use of COP techniques for GCCF.

In particular, this thesis advances the state of the art in the following ways:

- We proposed CFSS, a branch and bound algorithm for GCCF that can compute optimal and approximate solutions with quality guarantees for realistic large-scale applications (i.e., more than 2700 agents).
- We proposed the first GCCF model for Social Ridesharing (SR), and we showed that it can lead to significant cost reductions when benchmarked on large-scale instances (i.e., with 2000 agents) using realistic datasets.
- We proposed PRF, the first algorithm able to compute kernel-stable payments in the context of large-scale SR, thanks to an improved computation scheme that overcomes the shortcomings of previous algorithms.
- We proposed CUBE, a highly parallel implementation for the most computationally intensive operations of the Bucket Elimination algorithm. CUBE adopts a novel technique that allows us to realise an optimal memory management on GPUs, hence achieving a high computational throughput.
- We proposed COP-GCCF, the first COP model for GCCF of manageable complexity. We solved such COP with CUBE, outperforming state of the art algorithms on a realistic dataset, both in terms of runtime and memory.

This thesis opens several new research possibilities, both in the short and in the long term. On the one hand, we aim at studying other realistic GCCF scenarios that can be modelled as $m + a$ functions, so to employ the CFSS algorithm.

One interesting example is represented by the work of Lappas et al. [68], who focus on a task assignment context involving team of experts connected by a social network. In particular, as discussed in Section 3.1, the authors tackle the problem of forming a *single* group of agents, who collectively possess the skills to complete a given task, and who minimise the communication costs within such group. Notice that, similarly to the scenarios discussed in this thesis, the presence of the social network constraints the formation of the group, as disconnected agents cannot collaborate/communicate at all. A promising research direction involves addressing this scenario from a GCCF perspective, by considering a set of multiple tasks and partitioning the set of agents into feasible coalitions (i.e., that form connected subgraphs of the social network), each able to complete a subset of the set of tasks. In this context, each coalition corresponds to a value that considers both the sum of the rewards associated to the completed tasks, and, following Lappas et al. [68], the communication costs within the coalition. This characteristic function clearly exhibits a $m + a$ nature. On the one hand, the former component has a superadditive behaviour, as bigger groups can complete more tasks, and hence, receive higher rewards. On the other hand, communication costs are subadditive, as already discussed in Section 5. Furthermore, notice that, similarly to Social Ridesharing (SR), this scenario is subject to additional feasibility constraints, e.g., one tasks cannot simultaneously be assigned to more than one coalition, in order to avoid the duplication of its reward. Such constraints can be easily integrated into our search-based techniques, reducing the search space and improving the performance.

In the context of SR, we plan to extend our approach by focusing on the development of an *online* SR system, motivated by the inherent dynamic nature of realistic ridesharing systems like Uber and Lyft. In this perspective, we aim at the design of a SR model in which agents can join and leave the system over an extended amount of time. Such a scenario suggests a solution scheme that employs an offline method (e.g., SR-CFSS) at each time step. Myopic, short-sighted solutions are then avoided taking into account future requests. This can be achieved by means of *virtual agents*, i.e., agents who are not yet in the system, but could join it in the future. Such information, which clearly exhibits a probabilistic nature, could be provided by the history of previous requests. As an example, within a urban scenario the majority of the commuters probably moves towards the city centre in the morning, in order to reach their work places. On the other hand, it is reasonable to assume an opposite trend at the end of the day, when people return to residential areas. Uncertainty can also be included in the SR characteristic function, so to express costs deriving from traffic.

With this respect, it would also be interesting to formalise the above scenario as a Markov Decision Process (MDP) [43], and investigate the use of approximate solution techniques [60] to deal with the computational complexity of such MDP and calculate a good policy in a feasible amount of time.

In the context of constraint optimisation, this thesis opens to the application of the proposed GPU techniques to other algorithmic frameworks also based on the composition and marginalisation operations discussed in Chapter 8. As an example, these methods can be directly extended to Mini-Bucket Elimination (MBE) (see Section 3.4), so to achieve the same benefits in terms of runtime speed-up.

In addition, a very interesting research direction is represented by the joint application of Dynamic Programming (DP) and search-based techniques within a hybrid solution method for large-scale COPs. Such an approach allows to overcome the unpractical memory requirements inherent in DP while maintaining its benefits in terms of runtime performance. In this context, Marinescu and Dechter [77] proposed an algorithm that realises this scheme by employing MBE-based heuristics to implement a branch and bound traversal of a particular search-tree, i.e., an AND/OR tree. The authors later improved such an approach by incorporating a best-first strategy [76]. The successful use of such techniques for the solution of realistic COPs warrants the application of GPUs in this context. Specifically, we aim at studying if the employment of CUBE within the heuristic computation can result in more precise bounds, so to improve the pruning of the search space and, consequently, provide better performance.

Overall, this thesis proposes a novel way of employing MAS techniques, such as CF, previously employed only in small-scale environments. Our contributions pave the way to the use of these solutions and methods in realistic, large-scale scenarios, which find application in contexts that are central for future developments of the research within the AI community, such as computational sustainability. We believe that our work provides a comprehensive set of tools that allows to address these real-world challenges under a novel and exciting perspective.

References

1. R. Albert and A. L. Barabási. "Statistical mechanics of complex networks". In: *Reviews of Modern Physics* 74.1 (2002), pp. 47–97.
2. D. A. F. Alcantara. *Efficient hash tables on the GPU*. University of California at Davis, 2011.
3. G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *American Federation of Information Processing Societies*. 1967, pp. 483–485.
4. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, 2012.
5. Y. Bachrach and J. S. Rosenschein. "Coalitional skill games". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2008, pp. 1023–1030.
6. D. A. Bader, W. E. Hart, and C. A. Phillips. "Parallel Algorithm Design for Branch and Bound". In: *Emerging Methodologies and Applications in Operations Research*. Springer, 2005, pp. 5–44.
7. R. Baeza-Yates and P. V. Poblete. "Algorithms and Theory of Computation Handbook". In: CRC Press, 2010. Chap. Searching.
8. S. Balla-Arabe, X. Gao, D. Ginhac, V. Brost, and F. Yang. "Architecture-Driven Level Set Optimization: From Clustering to Subpixel Image Segmentation". In: *Cybernetics, IEEE Transactions on* PP.99 (2015), pp. 1–14.
9. E. Bensana, M. Lemaitre, and G. Verfaillie. "Earth observation satellite management". In: *Constraints* 4.3 (1999), pp. 293–299.
10. D. Berend and T. Tassa. "Improved bounds on Bell numbers and on moments of sums of random variables". In: *Probability and Mathematical Statistics* 30.2 (2010), pp. 185–205.
11. D. Berjon, G. Gallego, C. Cuevas, F. Moran, and N. Garcia. "Optimal Piecewise Linear Function Approximation for GPU-Based Applications". In: *Cybernetics, IEEE Transactions on* PP.99 (2015), pp. 1–12.
12. F. Bistaffa, N. Bombieri, and A. Farinelli. "An Efficient Approach for Accelerating Bucket Elimination on GPUs". In: *Cybernetics, IEEE Transactions on* PP.99 (2016), pp. 1–13.
13. F. Bistaffa, A. Farinelli, J. Cerquides, J. Rodríguez-Aguilar, and S. D. Ramchurn. "Algorithms for Graph-Constrained Coalition Formation in the Real World". In: *Intelligent Systems and Technology, ACM Transactions on* (2016).
14. F. Bistaffa, A. Farinelli, J. Cerquides, J. Rodríguez-Aguilar, and S. D. Ramchurn. "Anytime Coalition Structure Generation on Synergy Graphs". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2014, pp. 13–20.
15. F. Bistaffa, N. Bombieri, and A. Farinelli. "CUBE: A CUDA Approach for Bucket Elimination on GPUs". In: *European Conference on Artificial Intelligence*. 2016, pp. 125–132.
16. F. Bistaffa, A. Farinelli, and N. Bombieri. "Optimising memory management for Belief Propagation in Junction Trees using GPGPUs". In: *IEEE Inter-*

national Conference on Parallel and Distributed Systems. 2014, pp. 526–533.

17. F. Bistaffa, A. Farinelli, G. Chalkiadakis, and S. D. Ramchurn. "Recommending Fair Payments for Large-Scale Social Ridesharing". In: *ACM Conference on Recommender Systems*. 2015, pp. 139–146.
18. F. Bistaffa, A. Farinelli, and S. D. Ramchurn. "Sharing Rides with Friends: a Coalition Formation Algorithm for Ridesharing". In: *AAAI Conference on Artificial Intelligence*. 2015, pp. 608–614.
19. S. Bistarelli, U. Montanari, and F. Rossi. "Semiring-based constraint satisfaction and optimization". In: *Journal of the ACM* 44.2 (1997), pp. 201–236.
20. S. D. Canto, Á. P. de Madrid, and S. D. Bencomo. "Parallel Dynamic Programming on Clusters of Workstations". In: *Parallel and Distributed Systems, IEEE Transactions on* 16.9 (2005), pp. 785–798.
21. J. Cerquides, A. Farinelli, P. Meseguer, and S. D. Ramchurn. "A tutorial on optimization for multi-agent systems". In: *The Computer Journal* (2013), bxt146.
22. G. Chalkiadakis, E. Elkind, and M. Wooldridge. *Computational Aspects of Cooperative Game Theory*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer, 2011.
23. V. Conitzer and T. Sandholm. "Complexity of constructing solutions in the core based on synergies among coalitions". In: *Artificial Intelligence* 170.6 (2006), pp. 607–619.
24. T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.
25. R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. "Probabilistic Networks and Expert Systems". In: Springer, 1999. Chap. Building and Using Probabilistic Networks, pp. 25–41.
26. V. D. Dang and N. R. Jennings. "Generating coalition structures with finite bound from the optimal guarantees". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2004, pp. 564–571.
27. P. Dasgupta, V. Ufimtsev, C. Nelson, and S. G. M. Hossain. "Dynamic reconfiguration in modular robots using graph partitioning-based coalitions". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2012, pp. 121–128.
28. S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Education, 2006.
29. M. Davis and M. Maschler. "The kernel of a cooperative game". In: *Naval Research Logistics Quarterly* 12.3 (1965), pp. 223–259.
30. R. Dechter. "Bucket elimination: A unifying framework for reasoning". In: *Artificial Intelligence* 113.1–2 (1999), pp. 41–85.
31. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
32. R. Dechter. "Mini-buckets: A general scheme for generating approximations in automated reasoning". In: *International Joint Conference on Artificial Intelligence*. 1997, pp. 1297–1303.
33. R. Dechter, I. Meiri, and J. Pearl. "Temporal constraint networks". In: *Artificial intelligence* 49.1-3 (1991), pp. 61–95.

34. M. H. DeGroot. *Optimal statistical decisions*. Vol. 82. John Wiley & Sons, 2005.
35. G. Demange. "On Group Stability in Hierarchies and Networks". In: *Political Economy* 112.4 (2004), pp. 754–778.
36. X. Deng and C. H. Papadimitriou. "On the Complexity of Cooperative Solution Concepts". In: *Mathematics of Operations Research* 19.2 (1994), pp. 257–266.
37. N. Di Mauro, T. M. A. Basile, S. Ferilli, and F. Esposito. "Coalition Structure Generation with GRASP". In: *International Conference on Artificial Intelligence: Methodology, Systems, Applications*. 2010, pp. 111–120.
38. E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.
39. D. Dos Santos and A. Bazzan. "Distributed clustering for group formation and task allocation in Multi-Agent systems: A swarm intelligence approach". In: *Applied Soft Computing* 12.8 (2012), pp. 2123–2131.
40. E. Elkind, L. A. Goldberg, P. W. Goldberg, and M. Wooldridge. "A tractable and expressive class of marginal contribution nets and its applications". In: *Mathematical Logic Quarterly* 55.4 (2009), pp. 362–376.
41. R. Farber. *CUDA Application Design and Development*. Elsevier, 2012.
42. A. Farinelli, M. Bicego, S. Ramchurn, and M. Zucchelli. "C-link: A Hierarchical Clustering Approach to Large-scale Near-optimal Coalition Formation". In: *International Joint Conference on Artificial Intelligence*. 2013, pp. 106–112.
43. E. A. Feinberg and A. Shwartz. *Handbook of Markov decision processes: methods and applications*. Springer Science & Business Media, 2012.
44. F. Fioretto, T. Le, E. Pontelli, W. Yeoh, and T. Son. "Exploiting GPUs in Solving (Distributed) Constraint Optimization Problems with Dynamic Programming". In: *Principles and Practice of Constraint Programming*. 2015, pp. 121–139.
45. I. Foster. *Designing and building parallel programs*. 1995.
46. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.
47. M. E. Gaston and M. desJardins. "Agent-organized Networks for Dynamic Team Formation". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2005, pp. 230–237.
48. G. Grätzer. *Lattice Theory: Foundation*. Springer, 2011.
49. G. Greco, E. Malizia, L. Palopoli, and F. Scarcello. "On the complexity of core, kernel, and bargaining set". In: *Artificial Intelligence* 175.12 (2011), pp. 1877–1910.
50. T. D. Han and T. S. Abdelrahman. "Reducing branch divergence in GPU programs". In: *ACM GPGPUs Workshop*. 2011.
51. P. Hart, N. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (1968), pp. 100–107.
52. W. D. Hillis and G. L. Steele Jr. "Data Parallel Algorithms". In: *Communications of the ACM*. 1986, pp. 1170–1183.

53. K. Hirayama and M. Yokoo. "Distributed partial constraint satisfaction problem". In: *Principles and Practice of Constraint Programming*. 1997, pp. 222–236.
54. S. Huang, H. Liu, and V. Viswanathan. "Parallel dynamic programming". In: *Parallel and Distributed Systems, IEEE Transactions on* 5.3 (1994), pp. 326–328.
55. S. Ieong and Y. Shoham. "Marginal contribution nets: A compact representation scheme for coalitional games". In: *ACM Conference on Electronic Commerce*. 2005, pp. 193–202.
56. F. V. Jensen. *An introduction to Bayesian networks*. Vol. 210. Springer, 1996.
57. E. Kamar and E. Horvitz. "Collaboration and Shared Plans in the Open World: Studies of Ridesharing". In: *International Joint Conference on Artificial Intelligence*. 2009, pp. 187–194.
58. G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20 (1 1998), pp. 359–392.
59. K. Kask, R. Dechter, J. Larrosa, and A. Dechter. "Unifying tree decompositions for reasoning in graphical models". In: *Artificial Intelligence* 166.1 (2005), pp. 165–193.
60. M. Kearns, Y. Mansour, and A. Y. Ng. "A sparse sampling algorithm for near-optimal planning in large Markov decision processes". In: *Machine Learning* 49.2-3 (2002), pp. 193–208.
61. H. Keinanen. "Simulated Annealing for Multi-Agent Coalition Formation". In: *Agent and Multi-Agent Systems: Technologies and Applications*. Lecture Notes. 2009, pp. 30–39.
62. L. Khatib, P. Morris, R. Morris, and F. Rossi. "Temporal Constraint Reasoning with Preferences". In: *International Joint Conference on Artificial Intelligence*. 2001, pp. 322–327.
63. M. Klusch and O. Shehory. "A Polynomial Kernel-Oriented Coalition Algorithm for Rational Information Agents". In: *International Conference on Multi-Agent Systems*. 1996.
64. P. M. Kogge and H. S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *Computers, IEEE Transactions on*. 1973, pp. 786–793.
65. T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri. "A scalable generative graph model with community structure". In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. 424–452.
66. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company, 1994.
67. H. Kwak, C. Lee, H. Park, and S. Moon. "What is Twitter, a Social Network or a News Media?" In: *International World Wide Web Conference*. Raleigh, North Carolina, USA, 2010, pp. 591–600.
68. T. Lappas, K. Liu, and E. Terzi. "Finding a team of experts in social networks". In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2009, pp. 467–476.

69. S. L. Lauritzen and D. J. Spiegelhalter. “Local computations with probabilities on graphical structures and their application to expert systems”. In: *Journal of the Royal Statistical Society* (1988), pp. 157–224.
70. T. Léauté, B. Ottens, and R. Szymanek. “FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization”. In: *IJCAI DCR Workshop*. 2009, pp. 160–164.
71. L. H. S. Lelis, L. Otten, and R. Dechter. “Predicting the Size of Depth-First Branch and Bound Search Trees”. In: *International Joint Conference on Artificial Intelligence*. 2013, pp. 594–600.
72. J. K. Lenstra and A. Kan. “Complexity of vehicle routing and scheduling problems”. In: *Networks* 11.2 (1981), pp. 221–227.
73. S. Liemhetcharat and M. Veloso. “Weighted synergy graphs for effective team formation with heterogeneous ad hoc agents”. In: *Artificial Intelligence* 208 (2014), pp. 41–65.
74. J.-S. Liu and K. P. Sycara. “Exploiting Problem Structure for Distributed Constraint Optimization.” In: *International Conference on Multi-Aagent Systems*. Vol. 95. 1995, pp. 246–254.
75. L. S. Marcolino, A. X. Jiang, and M. Tambe. “Multi-agent Team Formation: Diversity Beats Strength?” In: *International Joint Conference on Artificial Intelligence*. 2013, pp. 279–285.
76. R. Marinescu and R. Dechter. “Best-first AND/OR search for graphical models”. In: *AAAI Conference on Artificial Intelligence*. 2007, pp. 1171–1176.
77. R. Marinescu and R. Dechter. “Dynamic orderings for AND/OR branch-and-bound search in graphical models”. In: *Frontiers in Artificial Intelligence and Applications* 141 (2006), p. 138.
78. F. McLoughlin, A. Duffy, and M. Conlon. “A clustering approach to domestic electricity load profile characterisation using smart metering data”. In: *Applied energy* 141 (2015), pp. 190–199.
79. R. B. Myerson. “Graphs and Cooperation in Games”. In: *Mathematics of Operations Research* 2.3 (1977), pp. 225–229.
80. H Narayanan. *Submodular functions and electrical networks*. Elsevier, 1997.
81. G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. “An analysis of approximations for maximizing submodular set functions”. In: *Mathematical Programming* 14.1 (1978), pp. 265–294.
82. N. Ohta, V. Conitzer, R. Ichimura, Y. Sakurai, A. Iwasaki, and M. Yokoo. “Coalition structure generation utilizing compact characteristic function representations”. In: *Principles and Practice of Constraint Programming*. Springer, 2009, pp. 623–638.
83. L. Otten and R. Dechter. “A case study in complexity estimation: Towards parallel branch-and-bound over graphical models”. In: *Conference on Uncertainty in Artificial Intelligence*. 2012, pp. 665–674.
84. G. Owen. *Game Theory*. Academic Press, 1995.
85. K. Pawłowski, K. Kurach, K. Svensson, S. Ramchurn, T. P. Michalak, and T. Rahwan. “Coalition structure generation with the graphics processing unit”. In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2014, pp. 293–300.

86. J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.
87. A. Petcu. "A Class of Algorithms for Distributed Constraint Optimization". PhD. Thesis No. 3942. Swiss Federal Institute of Technology (EPFL), 2007.
88. T. Rahwan and N. R. Jennings. "An improved dynamic programming algorithm for coalition structure generation". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. Estoril, Portugal, 2008, pp. 1417–1420.
89. T. Rahwan, T. P. Michalak, and N. R. Jennings. "A hybrid algorithm for coalition structure generation". In: *AAAI Conference on Artificial Intelligence*. 2012, pp. 1443–1449.
90. T. Rahwan, T. P. Michalak, M. Wooldridge, and N. R. Jennings. "Coalition structure generation: A survey". In: *Artificial Intelligence* 229 (2015), pp. 139–174.
91. T. Rahwan, T. P. Michalak, E. Elkind, P. Faliszewski, J. Sroka, M. Wooldridge, and N. R. Jennings. "Constrained Coalition Formation". In: *AAAI Conference on Artificial Intelligence*. 2011, pp. 719–725.
92. J. H. Reif. "Depth-first search is inherently sequential". In: *Information Processing Letters* 20.5 (1985), pp. 229–234.
93. F. Ruggeri, R. Kenett, and F. W. Faltin. *Encyclopedia of statistics in quality and reliability*. John Wiley, 2007.
94. M. A. Russell. *Mining the Social Web*. O'Reilly Media, 2013.
95. T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohmé. "Coalition structure generation with worst case guarantees". In: *Artificial Intelligence* 111.1 (1999), pp. 209–238.
96. N. Satish, M. Harris, and M. Garland. "Designing efficient sorting algorithms for manycore GPUs". In: *IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–10.
97. T. Schiex, H. Fargier, and G. Verfaillie. "Valued Constraint Satisfaction Problems: Hard and Easy Problems". In: *International Joint Conference on Artificial Intelligence*. 1995, pp. 631–637.
98. A. Schrijver. *Combinatorial optimization: polyhedra and efficiency*. Springer, 2003.
99. S. Sen and P. S. Dutta. "Searching for optimal coalition structures". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2000, pp. 287–292.
100. S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. "Scan primitives for GPU computing". In: *Graphics hardware*. 2007, pp. 97–106.
101. O. Shehory and S. Kraus. "Feasible Formation of Coalitions Among Autonomous Agents in Non-Super-Additive Environments". In: *Computational Intelligence* 15.3 (1999).
102. O. Shehory and S. Kraus. "Methods for task allocation via agent coalition formation". In: *Artificial Intelligence* 101.1-2 (1998), pp. 165–200.
103. A. Shekhovtsov. "Supermodular decomposition of structural labeling problem". In: *Control systems and computers* 1.39-48 (2006), p. 20.

104. A. Shekhovtsov, V. Kolmogorov, P. Kohli, V. Hlavác, C. Rother, and P. Torr. *LP-relaxation of binarized energy minimization*. Tech. rep. Czech Tech. University, 2008.
105. O. Skibski, T. P. Michalak, T. Rahwan, and M. Wooldridge. "Algorithms for the Shapley and Myerson Values in Graph-restricted Games". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2014, pp. 197–204.
106. R. E. Stearns. "Convergent Transfer Schemes for N-Person Games". In: *Transactions of the American Mathematical Society* 134.3 (1968), pp. 449–459.
107. G. Tan, N. Sun, and G. R. Gao. "Improving performance of dynamic programming via parallelism and locality on multicore architectures". In: *Parallel and Distributed Systems, IEEE Transactions on* 20.2 (2009), pp. 261–274.
108. Y. Tan and K. Ding. "A Survey on GPU-Based Implementation of Swarm Intelligence Algorithms". In: *Cybernetics, IEEE Transactions on* PP.99 (2015), pp. 1–14.
109. L. Tran-Thanh, T.-D. Nguyen, T. Rahwan, A. Rogers, and N. R. Jennings. "An efficient vector-based representation for coalitional games". In: *International Joint Conference on Artificial Intelligence*. 2013, pp. 383–389.
110. S. Ueda, M. Kitaki, A. Iwasaki, and M. Yokoo. "Concise characteristic function representations in coalitional games based on agent types". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2011, pp. 1271–1272.
111. S. Ueda, T. Hasegawa, N. Hashimoto, N. Ohta, A. Iwasaki, and M. Yokoo. "Handling Negative Value Rules in MC-net-based Coalition Structure Generation". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2012, pp. 795–804.
112. M. Vinyals, F. Bistaffa, A. Farinelli, and A. Rogers. "Coalitional energy purchasing in the smart grid". In: *IEEE International Energy Conference*. 2012, pp. 848–853.
113. M. Vinyals, J. A. Rodriguez-Aguilar, and J. Cerquides. "Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law". In: *Autonomous Agents and Multi-Agent Systems* (2011), pp. 439–464.
114. T. Voice, M. Polukarov, and N. R. Jennings. "Coalition structure generation over graphs". In: *Journal of Artificial Intelligence Research* 45 (2012), pp. 165–196.
115. T. Voice, S. Ramchurn, and N. Jennings. "On coalition formation with sparse synergies". In: *International Conference on Autonomous Agents and Multi-Agent Systems*. 2012, pp. 223–230.
116. T. K. Wijaya, T. Ganu, D. Chakraborty, K. Aberer, and D. P. Seetharam. "Consumer Segmentation and Knowledge Extraction from Smart Meter and Survey Data." In: *SIAM International Conference on Data Mining*. 2014, pp. 226–234.

117. Y. Xia and V. K. Prasanna. "Distributed Evidence Propagation in Junction Trees on Clusters". In: *Parallel and Distributed Systems, IEEE Transactions on* 23.7 (2012), pp. 1169–1177.
118. L. Yang, P. Luo, C. C. Loy, and X. Tang. "A Large-Scale Car Dataset for Fine-Grained Categorization and Verification". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 3973–3981.
119. D. Yun Yeh. "A dynamic programming approach to the complete set partitioning problem". In: *BIT Numerical Mathematics* 26.4 (1986), pp. 467–474.
120. L. Zheng and O. Mengshoel. "Optimizing Parallel Belief Propagation in Junction Trees using Regression". In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2013, pp. 757–765.
121. Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. "Understanding mobility based on GPS data". In: *International Conference on Ubiquitous Computing*. 2008, pp. 312–321.

Sommario

L'ottimizzazione a vincoli rappresenta una tecnica fondamentale che è stata applicata con successo nell'ambito dei Sistemi Multi-Agente (MAS), con lo scopo di risolvere numerosi problemi di coordinamento tra gli agenti. In questa tesi affrontiamo il problema della *Formazione di Coalizioni* (CF), uno degli approcci chiave per affrontare problemi di coordinamento nei MAS. In particolare, CF ha l'obiettivo di formare gruppi che massimizzano una funzione obiettivo (e.g., formare macchine condivise da più agenti in modo da minimizzare i costi di trasporto). Ci concentriamo su un caso particolare di CF denominato *CF su Grafi* (GCCF), dove una rete tra gli agenti vincola la formazione delle coalizioni. Questo problema si riscontra in molte applicazioni realistiche, ad esempio nel caso di reti di comunicazione o di relazioni sociali. Nello specifico, i contributi principali della tesi sono i seguenti.

Proponiamo un nuovo modo di formalizzare il problema GCCF, e un algoritmo efficiente (denominato CFSS) per risolverlo. CFSS è stato testato in contesti realistici quali il *collective energy purchasing* e il *social ridesharing*, utilizzando dati reali (i.e., profili di consumo energetico domestico del Regno Unito, GeoLife per le coordinate di spostamento nell'ambito del ridesharing, e Twitter come rete sociale). CFSS è il primo algoritmo in grado di risolvere GCCF su larga scala fornendo buone garanzie di qualità.

In aggiunta, affrontiamo il problema di dividere il valore associato ad ogni coalizione tra i suoi membri, in modo da garantire che siano ricompensati adeguatamente per il contributo apportato al gruppo. Questo aspetto di CF, chiamato *calcolo dei pagamenti*, è cruciale in ambiti caratterizzati da agenti con un comportamento razionale, quali il *collective energy purchasing* e il *social ridesharing*. Questo problema è risolto tramite il nostro algoritmo denominato PRF, il primo metodo in grado di risolvere questo problema su larga scala. In aggiunta, i pagamenti calcolati soddisfano la proprietà derivante dalla teoria dei giochi chiamata *stabilità*, che garantisce che tali pagamenti siano considerati imparziali dagli agenti.

Infine, proponiamo un metodo alternativo per la soluzione del problema GCCF, sfruttando la relazione tra GCCF e i problemi di ottimizzazione a vincoli (COP). In particolare, consideriamo Bucket Elimination (BE), uno dei framework più importanti per la risoluzione dei COP, e proponiamo CUBE, un implementazione parallela di BE su GPU. CUBE adotta uno schema della gestione della memoria innovativo, che porta notevoli benefici dal punto di vista delle performance e per-

mette a CUBE di non essere limitato dal quantitativo di memoria della GPU, così da poter risolvere problemi di carattere reale. CUBE è stato testato su SPOT5, un dataset realistico che contiene problemi di coordinamento tra satelliti modellati tramite COP. Inoltre, CUBE è stato usato per risolvere COP-GCCF, la nostra formalizzazione tramite COP del problema GCCF. COP-GCCF è il primo modello che comprende un numero lineare di vincoli rispetto al numero di agenti, caratteristica fondamentale per garantire la scalabilità della nostra tecnica risolutiva. I nostri esperimenti, che utilizzano Twitter come dataset reale, dimostrano che COP-GCCF apporta numerosi vantaggi rispetto allo stato dell'arte, sia in termini di memoria e di runtime.

In generale, questa tesi propone una nuova prospettiva su importanti tecniche nell'ambito dei MAS, quali CF e l'ottimizzazione a vincoli, permettendo di risolvere per la prima volta problemi di carattere reale su larga scala.