Constraint Games for stable and optimal allocation of demands in SDN

Anthony Palmieri, Arnaud Lallouet, Luc Pons

Constraint Games for stable and optimal allocation of demands in SDN

Anthony Palmieri[1,2], Arnaud Lallouet[1,2], and Luc Pons[1]

[1] Huawei Technologies Ltd, French Research Center
{anthony.palmieri,arnaud.lallouet,luc.pons}@huawei.com
[2] GREYC - Université de Caen - Normandie

Abstract. Software Defined Networking (or SDN) allows to apply a centralized control over a network of commuters in order to provide better global performances. One of the problem to solve is the multi-commodity flow routing where a set of demands have to be routed at minimum cost. In contrast with other versions of this problem, we consider here problems with congestion that change the cost of a link according to the capacity used. We propose here to study centralized routing with Constraint Programming and selfish routing with Constraint Games. Selfish routing is important for the perceived quality of the solution since no user is able to improve his cost by changing only his own path. We present real and synthetic benchmarks that show a good scalability.

1 Introduction

With the internet of things, all kinds of devices are going to communicate, from washing machines, lightbulbs to autonomous cars. By 2020, the forecasts estimate the number of connected devices to the internet are growing over 31 billions [14]. The amount of data transfer increases with the rise of the number of connected devices. Recently, Software Defined Networking (or SDN) is replacing traditional network routing because it allows fast and remote network reconfiguration, which enables a plethora of flexible architectures, like the upcoming network slicing [24]. SDN (see Figure 1) allows to apply a centralized control over a network of commuters in order to increase the overall performance.

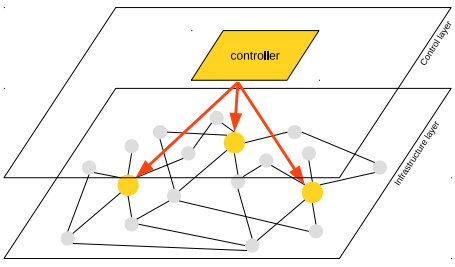

Fig. 1. Software Defined Networking

In this paper, we consider the independent routing of multiple demands across a network, also called *multi-commodity flow routing problem*. Each demand has a source and a destination and each link has a capacity. This problem has been studied since a long time [3], mostly with Linear Programming or with other incomplete methods [1]. A survey can be found in [9]. Interesting theoretical results have been found, like the one which states that when the problem has a sufficient size and capacity, all flows are actually routed along single-paths [18]. This justifies the modern interest in unsplittable routing of demands. Since capacity constrained shortest path is already NP-complete, we do not consider other side constraints such as must-pass/cannot-pass or redundant routing, although they can be easily introduced in our constraint model. But we do consider a congestion model that increases the cost of a link according to the traffic routed and we propose a Constraint Programming model to solve it optimally. For this, we use a natural heuristic based on increasing paths and we introduce a lower bound that can be used in branch and bound efficiently. Small instances of the problem correspond to networks of aggregated traffic for which the users (often network providers) are very sensitive to the quality of service. This is why an allocation which is a Nash equilibrium will be preferred as it ensures the user that its quality of service cannot be improved by any selfish move. For this, we make use of the recently introduced Constraint Games framework [15] to compute routings which are at Nash equilibrium. In addition, we derive exact and approximate bounds for Price of Anarchy [4] that allow to evaluate the loss of efficiency of decentralized algorithms. In the benchmarks, we show games with hundreds or even thousands of players solved up to optimality. These results have been obtained with ConGa, an extension of the Choco solver for Constraint Games [15] and show that a practical use of game theory is now possible at industrial scale.
Path heuristics have been introduced in CP in [16] in the context of network design, which includes as a subproblem the routing problem we focus on (but without taking congestion into consideration). More recently, in [8], the authors model a SDN problem with CP. Our work differs in multiple points. We consider only single-path routing, we take congestion into account, and more important, we take into account the quality of service through the computation of Nash equilibria. From the game theory point of view, the closer class of game studied are the congestion games [21]. This class of games tries to study the impact of the congestion over a network. However this work is different of ours because it never considers hard constraints in the model.
The paper is organizing as follow: first we introduce the problem in Section 2, then in Section 3 and 4 we present the CP model and the heuristics used to compute a solution in practice. We present the Constraint Games framework used to compute selfish routing in Section 5 and end by presenting benchmarks for a range of real-world and synthetic instances.

2 Multicommodity path routing in SDN

2.1 Problem statement

A multicommodity path routing problem (MCPRP) consists in a graph defining a network and a set of commodities (flow demands) to be routed on this graph. We consider here the simple problem in which we compute for each demand a single route from the

source to the destination node such that the sum of bandwidth routed by a link does not exceed its capacity. Congestion occurs when a link is taken and is reflected by a congestion cost which helps to ensure an homogeneous distribution of the routes. The overall objective is to minimize the sum of costs of the routed demands, and in case of games, while preserving optimality for each player.

We assume we have a network $N = (V, E)$, which is a directed graph composed of a set of vertices (or nodes) V and a set of edges (or links) $E \subseteq V^2$. For each edge $e = (x, y) \in E$, we associate a cost $cost(e) \in \mathcal{R}^+$ and a capacity $cap(e) \in \mathcal{R}^+$. Let D be the set of demands to be routed. For a demand $d \in D$, we define $src(d) \in V$ and $dst(d) \in V$ to be respectively the source and destination node, and $bw(d) \in \mathcal{R}^+$ to be the required bandwidth for this demand.

A *path* is a sequence of nodes $p = (v_i)_{i \in [0..n]}$ such that $\forall i \in 0..n-1, (v_i, v_{i+1}) \in E$. We denote by $src(p)$ the node v_0 and by $dst(p)$ the node v_n. We consider here only acyclic paths, i.e. such that $i \neq j \rightarrow v_i \neq v_j$. By a slight abuse of notation, we write $(x, y) \in p$ to denote that the arc (x, y) is taken in the path p.

A *solution* for the MCPRP is the assignment of a path $path(d)$ to each demand d such that:

1. (correctness) $\forall d \in D,\ src(path(d)) = src(d)$ and $dst(path(d)) = dst(d)$
2. (admissibility) $\forall e \in E, \left(\sum_{\{d \in D \mid e \in path(d)\}} bw(d) \right) \leq cap(e)$

2.2 Congestion model

In order to ensure a good balance over the network, we incorporate to the model a model of congestion. Basically, congestion will increase the cost of a link when this link will be close to saturation. For this, we define the load of an edge e to be:

$$load(e) = \left(\sum_{\{d \in D \mid e \in path(d)\}} bw(d) \right) \ / \ cap(e)$$

The congestion model we use has an exponential increase of the form $cong(x) = e^{ax+b}$ where x is the load of the arc. In order to choose the parameters a and b, we pose some conditions the function. First we should have a sufficiently high value of $cong(e)$ when the load is 1. By sufficiently high we mean that a demand should not prefer to take a heavily congested link while there are some (maybe longer) available paths. It can be done by fixing this limit to the highest link cost of the network Mxc. We then have the equation $e^{a+b} = Mxc$. Then, in order to set when the exponential starts to overtake on a linear increase, we impose a condition on the derivative to be 1 at a given point α. The derivative of the congestion function is given by $cong'(x) = ae^{ax+b}$. If we impose that the derivative should be 1 for $x = \alpha$, we get the equation $ae^{a\alpha+b} = 1$. By solving numerically these equations we get the values of a and b for a given problem. For example, in Figure 2 is a plot of the congestion function for $Mxc = 1000$ and $cong'(0.2) = 1$.

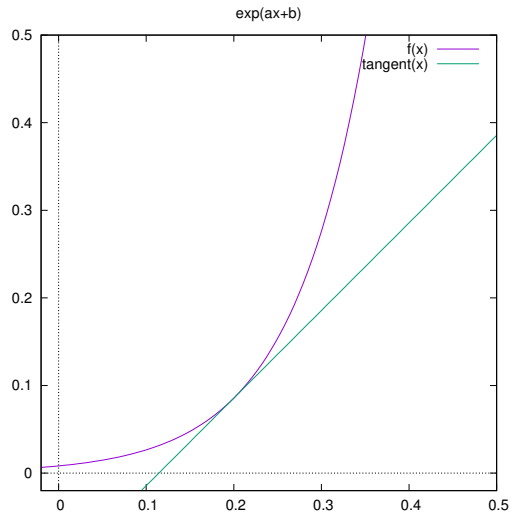

Fig. 2. A plot of the congestion function for $Mxc = 1000$ and $cong'(0.2) = 1$

2.3 Optimization

Solving a MCPRP P to optimality means finding a solution minimizing the global cost of the demands. For this, we first define the cost of a demand. It is obtained by aggregating the cost of each traversed arc with the cost coming from congestion:

$$cost(d) = bw(d) * \sum_{e \in path(d)} (cost(e) + cong(e)) \tag{1}$$

Then the cost of the whole problem is given by:

$$cost(P) = \sum_{d \in D} cost(d) \tag{2}$$

Note that this function is strictly monotonic, resulting that each addition of demand increases the edge cost. The problem of finding a Nash equilibrium adds to this the constraint that there should not exist a better path for each demand provided that the other paths remain identical. In other words, path should also be optimal for each demand. This model will be presented in more details in Section 5.

3 Constraint model

In order to implement this problem as a constraint program, we need to first represent paths, which will be the solutions of our problem. Then we need to link the computed paths to the network data: costs, capacity and provide a support to compute congestion.

3.1 Model of path

A path is represented by an array $path$ of $|V|$ variables, each one being assigned to the next node along the path. The initial domain of a variable associated to a node v is given by the the set of neighbors of v in the graph. In order to ensure the correct representation of a path, we use the global constraint $subPath(path, src, dst)$ which ensures that the node from src to dst form a valid subpath of the graph. This constraint is a variant of $subCircuit$. Unused nodes of the path point to themselves and an extra variable is appended to the array to allow any vertex to be starting the path (see Figure 3 for the encoding of the path $(2, 1, 4, 3, 5)$ from $src = 2$ to $dst = 5$).

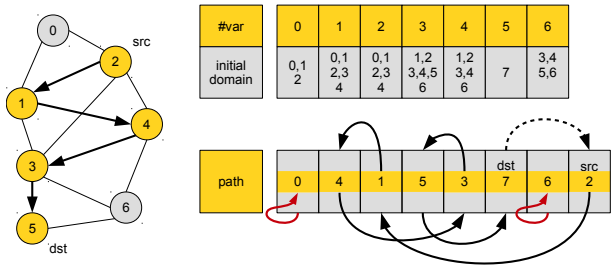

Fig. 3. Encoding of a path

For each demand $d \in D$ we associate an array $path^d = (v_i^d)_{i \in V}$ constrained by

$$subPath([v_1^d, \ldots, v_n^d], src(d), dst(d)) \tag{3}$$

3.2 Graph model

In order to ensure that no link is overloaded and in order to compute congestion, we need to know which demands are routed by a given arc. In this model, we use a Boolean variable $EdgeIsUsed_{(i,j)}^d$ which is true if the path $[v_1^d, \ldots, v_n^d]$ assigned to demand d uses the arc (i, j). This connection is done with the following channeling constraint:

$$\forall (i, j) \in E, \forall d \in D, \quad EdgeIsUsed_{(i,j)}^d \leftrightarrow v_i^d = j \tag{4}$$

We compute the amount of bandwidth routed by an arc in a variable $f(e)$ with the constraint:

$$\forall e \in E, \quad f(e) = \sum_{d \in D} EdgeIsUsed_e^d * bw(d) \tag{5}$$

We ensure that the capacity of each arc is not exceeded:

$$\forall e \in E, \quad f(e) \leq cap(e) \tag{6}$$

Then we can compute the congestion of a given edge in a variable $cong(e)$:

$$\forall e \in E, \quad cong(e) = e^{a * \frac{f(e)}{cap(e)} + b} \tag{7}$$

The cost $cost(d)$ of routing a demand by a given path is given by the constraint:

$$\forall d \in D, \quad cost(d) = \sum_{e \in E} EdgeIsUsed_e^d * bw(d) * (cost(e) + cong(e)) \tag{8}$$

A variable $ProblemCost$ sums the costs of all demands:

$$ProblemCost = \sum_{d \in D} cost(d) \tag{9}$$

We shall minimize this variable.
This model is quite easy and intuitive. It defines one Boolean variable by edge and by demand. Since the number of edges is quadratic in the number of vertices, this number may grow a lot for some large networks.

4 Heuristics and Branch & Bound

We have tried a variety of combinations of heuristics (or search strategy) and Branch and Bound to improve the resolution of this problem. In this paper, we will refer to a particular combination by `A/B/C` where `A` is the variable heuristic, `B` the value heuristic and `C` the type of Branch and Bound, as explained below. At a given node of the search tree, some demands or some partial paths may already be assigned. All heuristics and upper bound computation use the *residual graph* obtained by considering this part already fixed.
Path-oriented problems are particularly sensitive to heuristics, and not surprisingly, a standard dynamic CP heuristic (denoted by `CP` in this paper) would be of weak efficiency for this type of problem. Indeed, it is likely that this heuristic will label any node in the path without knowing if it could be linked to the source or destination. Therefore, we have chosen to label path variables in order of *increasing* path length. Note that this is a partial variable heuristic since it is only once the demand is chosen that the actual variable is determined by the next step to be extended. For the variable heuristics to be completely defined, we have considered three strategies for choosing the demand. The first one, called `MB` (for *Max_Bandwidth*), consists in routing the next remaining demand with the maximum bandwidth up to its completion. Then we have defined two strategies based on conflicts which analyze the current solution once a first solution has been found by `MB` or the situation after a fail. For each demand and each link, we compute the marginal cost (with congestion) induced by the presence of the demand on this very link. Then we sum up all these numbers for each demand along the taken path to obtain a score. The first one, called `CO` (for *Conflict*), chooses the demand of highest score and develop its path up to the destination. The second one, called `CO1` (for *Conflict_1_Step*), also chooses the demand of highest score but only develops one step in the path before reconsidering the situation. In `CO1`, the conflicts are stored for each path variable for each demand and score are only computed for the uninstantiated variables.

Example 1. In this example, we show the selection process of `CO` (upper part of Figure 4) and `CO1` (lower part of Figure 4). Three demands d_1, d_2 and d_3 with a respective

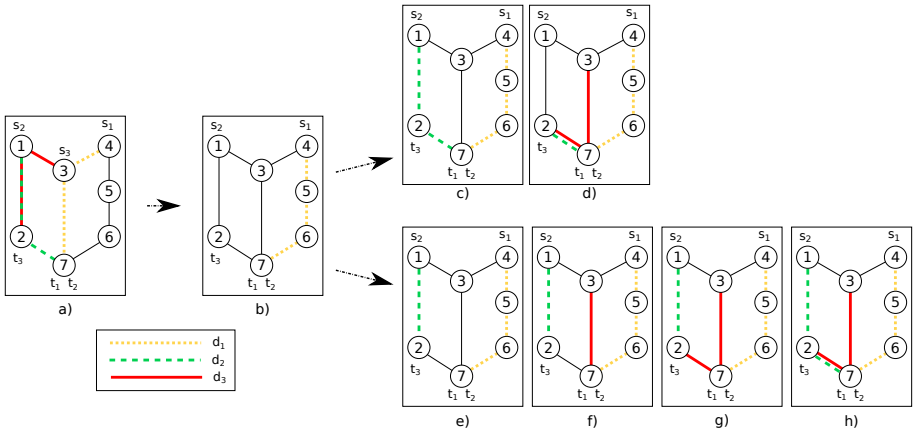

Fig. 4. Comparison of the variable selection of the different strategies

bandwidth of $4, 3$ and 2 have to be routed in the 7-nodes network depicted in Figure 4. To keep things easy, all edges have a capacity of 7, a cost of 0 and congestion parameters are set to $a = 1$ and $b = -0.5$. The Figure 4 compares the behaviors of the two strategies: `CO` which chooses to develop a complete path and `CO1` which chooses to develop one edge after the other. At first (4a), a feasible solution is found. Path for d_1 is $[4, 3, 7]$, for d_2 $[1, 2, 7]$ and for d_3 $[3, 1, 2]$. After this first solution, conflicts are analyzed. On the edge going from the node 1 to 2, the demands 2 and 3 are considered in "conflict". These conflicts are monitored differently depending on the strategy: `CO` has a counter for each demand while `CO1` maintains one counter for each variable, and for each demand. The corresponding counters (variable/demands or just demand) of d_2 and d_3 will be respectively increased by the cost of routing d_3 knowing that d_1 ($\Delta(price)_{d_3}(d_2)$) is already routed and the opposite ($\Delta(price)_{d_2}(d_3)$):

- $\Delta(price_{d_3}(d_2)) = 4 \times (e^{\frac{3}{7}+\frac{2}{7}-0.5} - e^{\frac{2}{7}-0.5}) = 0.19$
- $\Delta(price_{d_2}(d_3)) = 2 \times (e^{\frac{4}{7}+\frac{3}{7}-0.5} - e^{\frac{4}{7}-0.5}) = 0.06$

Then, the solver backtracks to a possible solution of lower cost, and finds another path for d_1 (4 b).
The `CO` heuristic chooses the demand having the highest conflict score. In (4c and d) `CO` chooses d_2 and then d_3.
`CO1` still selects the demand with the highest conflict score but, it is computed over the non instantiated vertices variables. And thus, the path of d_2 is continued(4e), since it is the demand having the highest conflict score. Afterwards, the score of d_2 becomes 0, because all its non-instantiated variable have a conflict score of 0. d_3 is continued twice because it has the highest score (4e ad f) and then reaches its destination. The demand d_2 is then continued up to its end (4e ad g).

For each variable, the value heuristics determines the direction the path will take. Since the goal is to find the best path for each demand, it would be inefficient to start the path in a wrong direction. In order to start with the most promising path, we maintain

at each node of the search tree the shortest path to destination in the residual network for each demand in isolation. We call this heuristic `SP` (for *Shortest_Path*). It is done with Dijkstra's algorithm, considering the progression of the already assigned part of the other demands. This information on the best future path is used to choose the next node of the path when needed. Note that the Dijkstra algorithm only considers the nodes of the paths already assigned at a given point of the search tree for computing the congestion. In particular, the congestion is not cumulative for two demands which share the same future link. The same idea has been implemented in [2] but with specific path variables.

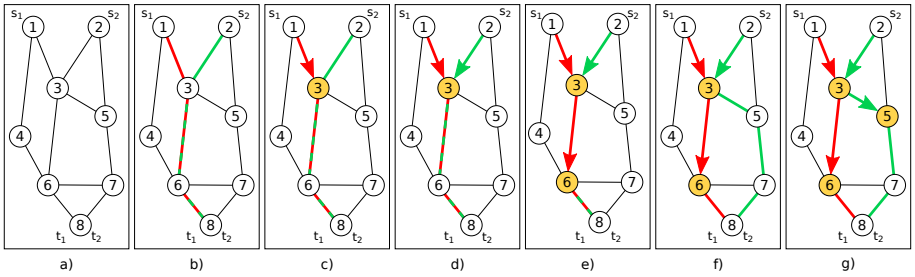

Fig. 5. Labeling of paths for two demands

Example 2. In Figure 5 is depicted a small example of two demands being routed on a 8-nodes network (5a) by the `CO1` heuristic. Demand d_1 has to be routed from node 1 to 8 and demand d_2 from 2 to 8. Actual paths taken by the demands are depicted by solid arrows while shortest paths computed by the heuristic are with solid lines. At first (5b), the two demands compute their respective shortest paths: $(1-3-6-8)$ for d_1 and $(2-3-6-8)$ for d_2. In (5c), one labeling step is performed for d_1. Since there is no change on d_2's path, no update of d_2's shortest path is necessary. Hence in (5d) one step is performed for d_2. In (5e), the next move of d_1 causes congestion on the link from 3 to 6. Thus d_2 updates its shortest path to $(2-3-5-7-8)$ in (5f) to lower its minimal cost. It yields a next move by d_2 in the direction of node 5 in (5g).

Branch and bound is a common technique used in constraint optimization. However, CP solvers offer a restricted and uninformed version of branch and bound: when minimizing the variable $ProblemCost$ and after having found a solution of value A, it simply adds to the end of the search the constraint $ProblemCost < A$. While efficient, it requires that the lower bound of $ProblemCost$ to exceed A to cut the search tree and backtrack. In our case, the possible values of $ProblemCost$ are strongly constrained by the current branch of the search tree leading to a node, but very loosely for the remaining part of the problem. In order to cut earlier, we need a better estimation of the lower bound of $ProblemCost$. This is done by adding to the lower bound the cost of individual routing along the path computed by the Dijkstra algorithm used for the value heuristic. Congestion is taken into account only for the already assigned nodes in

demand paths and the current demand to estimate. It means that two demands whose future path would take the same link do not create congestion in their future paths. We need this to provide a better yet safe estimate of the lower bound which does not exceed the future real cost. We call the classical CP branch & bound `CP` and the one which uses the bound provided by the shortest path `SP`.
Let $[a_1^d, \ldots, a_i^d, \ldots, a_{n_d}^d]$ be the path computed by the Dijkstra algorithm for a demand d from node a_i^d. We have $a_1^d = src(d)$ and $a_{n_d}^d = dst(d)$ and $\forall j < i$, the value of a_j^d is given by the instantiated part of the path in $[v_1^d, \ldots, v_n^d]$ (up to the current node of the search tree). The cost contribution of demand d is given by:

$$cost(d) = \sum_{\{e=(a_j^d, a_{j+1}^d) \mid j<i\}} bw(d) * (cost(e) + cong(e)) + \sum_{\{e=(a_j^d, a_{j+1}^d) \mid i \leq j < n_d\}} bw(d) * cost(e) \quad (10)$$

Proposition 1. *Given a monotonic cost function (see equation (1)), the bound given in equation 10 is sound.*

Proof. Since Dijkstra algorithm computes the shortest paths according to costs, all other paths for this demand are of greater cost. In addition, routing additional demands can only increase congestion, thus increase the total cost of each demand.

Note that, due to the presence of link capacity constraints, a fail is triggered when Dijkstra algorithm is unable to find a path from the source to the destination [22].

5 Constraint Games

In this section, we briefly introduce Constraint Games [11, 15], which are an extension of Constraint Programming allowing to find Nash equilibria.
A *game* [4] is a situation where a set of players $\mathcal{P}$ can perform actions and get a reward which depends on their own choice of action, but also on the actions of the other players. Players are selfish and always aim to increase their utility by changing their own action if they have an opportunity to do so. A (pure) *Nash equilibrium* (PNE) [25, 10] is a situation where all players cannot improve their own utility by changing their own action. A game may or may not have an equilibrium, and the existence of an equilibrium is an NP-complete problem [6].
In many cases, the efficiency of a solution can be evaluated by an external measure called *social welfare* function which should be maximized. This global function allows to compute the best centralized solution (by discarding the players objectives). Then it is possible to quantify the loss of efficiency induced by the selfish behavior of the players by considering the ratio "best centralized solution / best equilibrium" called Price of Stability (PoS) and "best centralized solution / worst equilibrium" called Price of Anarchy (PoA).
Constraint Games allow to represent in a compact and natural way games with multiple players and also give a powerful solving method by lifting consistency techniques to the

equilibrium property [15]. In Constraint games, actions are represented by the possible assignments of controlled variables. Utility is represented with constraint optimization, and the rich language of most constraint solvers is available to express a large spectrum of problems in a concise and meaningful way.

A *Constraint Satisfaction Game* (or CSG) is a 4-tuple $(\mathcal{P}, V, D, G)$ where $\mathcal{P}$ is a finite set of players, V is a finite set of variables composed of a family of disjoint sets $(V_i)_{i\in\mathcal{P}}$ for each player and a set V_E of *existential* variables disjoint of all the players variables, D is defined as for CSP, and $G = (G_i)_{i\in\mathcal{P}}$ is a family of CSP on V representing the *goal* of each player. In a CSG, all players seek for satisfaction of their goal. However, it may happen that a player is not satisfied in an equilibrium if none of his/her move allows for satisfaction. Determining whether a game has a PNE in a Constraint Satisfaction Game is Σ_2^P-complete. Note that [11] has introduced satisfaction and optimization variants of Constraint Games. A *Constraint Optimization Game* (COG) is a variant $(\mathcal{P}, V, D, G, opt)$ where $opt = (opt_i)_{i\in\mathcal{P}}$ and $\forall i \in \mathcal{P}, opt_i \in V$ is the variable whose value defines the utility function u_i of Player i. All players want to maximize their utility.

In addition, Constraint Games are able to represent easily *hard constraints* that define situations which are globally possible or forbidden [20] by adding a global CSP C to the problem. Nash equilibria can only be sought in the satisfiable part of the hard constraints. A global optimization condition on a variable w allows to model the social welfare function. Without further information, we call *Constraint Game* a COG with constraints and social welfare and we refer to it by $CG = (\mathcal{P}, V, D, G, opt, C, w)$.

The solving technique introduced in [11] and further developed in [15] is based on tree search. Players' preferences are represented by *Nash constraints* and their filtering is based on the detection of *never best responses*, which are values that never lead to an improvement. The strong filtering of [15] works only for Constraint Games without hard constraints (or if the hard constraints are functional), otherwise we can fall back to the weaker form of [11], which is the case in this problem because of the capacity constraints on the links.

Incomplete algorithms can also be used to find quickly a first Nash equilibrium. *Iterated Best Response* (or IBR) [23] is the simplest local search algorithm to find a PNE in any game representation. This iterative process starts from any strategy profile. At each step, if there exists a player for whom the current strategy profile is not a best response, then this player deviates to his best response which will be considered as the candidate in the next step. The process stops when all players are no longer able to change their strategy or if the algorithm fails to find an equilibrium in a given time credit *Max_Step*. In the first case, the last profile is a Nash equilibrium. In this paper, we have used IBR as a heuristic to go from the first solution to the first equilibrium.

The MCPRP defined in section 2 can be simply extended to a game by considering each demand as a player who wants to find the best route from source to destination. Then each player wants to minimize her/his own cost as defined in equation 1.

If we denote by $S = D^V$ the total search space and by N the set of Nash equilibria, we can define formally the welfare of the best centralized solution adapted to our cost minimization problem by $W^* = \min\{w(s) \mid s \in S\}$. The welfare of the best Nash equilibrium is defined in a similar way by $N^* = \min\{w(s) \mid s \in N\}$ and the one

of the worst one by $n^* = \max\{w(s) \mid s \in N\}$. Thus the Price of Stability is simply $PoS = W^*/N^*$ and the Price of Arnarchy $PoA = W^*/n^*$. Note that usually the classical definitions of PoS and PoA yield a result greater than 1, this is not the case here because we have a minimization problem.
In our problem, the social welfare function is simply the global cost to be minimized as defined in equation 2. We proceed in two steps. First the best centralized solution is computed as a Constraint Optimization Problem, then the Nash equilibria using our Constraint Games solver. We can immediately see that PoS and PoA are asymetric in term of the branch and bound we can implement. For PoS, the problem is still a minimization. Thus we can use the same branch and bound as the one we use in the centralized version (equation 10).
For the PoA, we have a maximization problem. But each player still wants to minimize her/his cost. The situation is then to find a set of *shortest* paths of *maximal* global cost. The standard branch and bound provided by the CP solver provides a loose upper bound for this problem by summing up all upper bounds of the costs of the edges. But we know that the upper bound is at most the cost of the longest path in the residual network. Unfortunately, computing the longest path is NP-complete in the general case, since it corresponds to determine if it exists an Hamiltonian cycle, which is NP-complete [5]. This problem has been already addressed in CP [17] where the authors propose a model and a local search algorithm to solve this problem. In our case, we are interested in a polynomial sound algorithm. This is why we propose to approximate the longest path by a Maximum Spanning Tree (MST) in the residual graph. The MST is computed by considering the upper bound value of the cost of the edges. The algorithm is like Prim's algorithm, we add to all remaining edges the cost of the demand to compute congestion, then we start by taking the most costly edge and add edges linking a new node in descending order of cost. It is clear that the cost of the MST is always greater than the cost of the longest path.

6 Instances and experimental results

We have tested our framework on a library of instances called SNDlib [13] and a personal problem generator that is able to generate instances close to real ones.

6.1 Generator

We have designed a generator to create synthetic problems that allow to test the algorithms against different hypothesis. Several parameters allow to obtain a great variety of graphs. The generation process is mainly constituted of two phases:

- Generation of the topology, that is nodes as well as arcs and their respective costs;
- Generation of demands, along with their bandwidth requests, which also determines the capacities of the arcs.

During the generation of the topology, N_{nodes} nodes are created. Each of these nodes n_i with $i \in [1, N_{nodes}]$ is assigned to random coordinates in a fixed size space of $topologyDimension$ dimensions. In case the boolean $topologicalCost$ is set to $true$,

the cost of an arc is given by the distance between the source and destination node. Note that for dimension 2, this is not sufficient to ensure that the resulting graph is planar. The size of the space in one given dimension is irrelevant, as we refer to it only with percentage. Each node is also assigned to a degree, randomly chosen in an interval $[deg_{min}, deg_{max}]$. To obtain graphs similar to actual networks, we introduce hubs which are nodes of higher degree than regular nodes. Each node has a probability P_{hub} of being a hub. If a node is a hub, then its degree is randomly chosen in a different interval $[degh_{min}, degh_{max}]$.
We first build a spanning tree over all nodes to ensure that the graph is fully connected, then we create the remaining links in the graph. For each node, we look for candidates so the desired degree is reached. For a link to be created, we ensure that a) the other node is not already connected with this edge and b) its distance in the space is not greater than $maxDistance$, expressed as a percentage of the space size ($\sqrt{topologyDimension}$ is the maximum). Using this process, it is possible that certain nodes do not reach the desired degree, but as the network grow larger, this situation becomes less and less likely to happen.
Once the topology is generated, $N_{demands}$ are generated. For each of these, a starting node is randomly selected, as well as a bandwidth in a $[bw_{min}, bw_{max}]$ interval. We then generate what we refer to as an "initial path". For that purpose, different strategies are available. The first strategy, called random generation, consists in selecting a random number of hop h in the $[hop_{min}, hop_{max}]$ interval, and randomly navigating in the graph for h hops, starting from the initial node. The last node is then considered to be the destination node of the request. During the navigation, we only make sure to never reach a node that is already in the initial path. The second strategy consists in randomly selecting a destination node, and applying a shortest path algorithm to find the the path from the source node to the destination node with the least number of hops. The path yielded by the algorithm is considered to be the initial path. Regardless of how the initial path is constructed, for each of its arc, there is a probability P_{bw} that we increase its capacity of the amount of bandwidth of the request. The list of all generation parameters, as well as short description can be found in Table 1.

6.2 Settings

Due to the large number of parameters of the generator, we have applied a benchmark method called combinatorial testing [12, 7] using the ACTS software [26]. This technique allows for p parameters and a size c to generate a set of instances where all possibilities of combinations of parameters of cardinality c are inside the set. For example, if we have 3 Boolean parameters a, b and c, a complete test of all possibilities would require $2^3 = 8$ tests. But if we decide to test only all combinations of pairs, we can achieve this with only 4 instances (see Figure 6). With our generator, by choosing an appropriate sampling of the intervals described in 1, we get roughly 500 instances to get a covering of all 3-sets of parameters. From these 500 instances, we have discarded those whose resolution lead to a timeout for all techniques. This gives a total of 123 instances which give a meaningful picture of the range of problems that can be solved.
The tests have been performed on a cluster of Intel Xeon E5-2690, each having 10 cores sequenced at 3GHz and 256 GB of RAM. We have computed experimental results for

Parameter	Type	Range used in the benchmarks
N_{nodes}	integer	$50, 75, 100, 120, 140, 160, 180, 200, 500$
$topologyDimension$	integer	2
$topologicalCost$	boolean	$true, false$
deg_{min}, deg_{max}	integer	$[1, 2, 4, 8]$, $[2, 5, 7, 10]$
P_{hub}	integer	$0, 1, 5, 10, 20$
$degh_{min}, degh_{max}$	integer	$[25, 50, 75, 100]$, $[25, 50, 75, 100]$
$initbw_{min}, initbw_{max}$	integer	$[50, 100, 200]$, $[50, 100, 200]$
$initcost_{min}, initcost_{max}$	integer	$[100, 200]$, $[200, 500]$
$maxDistance$	integer	$25, 50, 100$
$N_{demands}$	integer	$30, 50, 100, 120, 130, 150, 200$
bw_{min}, bw_{max}	integer	$[50, 100, 200]$, $[50, 100, 200]$
hop_{min}, hop_{max}	integer	$[0]$, $[10]$
P_{bw}	integer	$[10, 30, 50, 70, 90, 100]$

Table 1. Parameters of the generator

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

a	b
0	0
0	1
1	0
1	1

a	c
0	0
0	1
1	1
1	0

b	c
0	0
1	1
0	1
1	0

Fig. 6. All pairs of parameters are covered by 4 tests

the CP approach described in Section 3 and the Constraint Game model of Section 5 with a timeout fixed at 1 hour.

6.3 Experimental results

For the synthetic benchmarks, we have displayed the results in Figure 7. As a preliminary test, we have tried the pure `CP` heuristic based on impact [19] to measure the gap with the `SP` value heuristics. A problem which should be easy (13 nodes and 9 demands) is solved in less than 1 second by using the shortest path strategy, whereas the impact strategy took 878.969 seconds. Due to this, we have not displayed this `CP/CP/CP` heuristics in the figure and we only present results for the `SP` strategy.

For each instance, we have run the combinations `MB/SP/CP` and `MB/SP/SP`, and the two conflict variants `CO/SP/SP` and `CO1/SP/SP`. The plot in Figure 7 show how many instances are solved in a specific delay. Clearly, the `MB/SP/SP` heuristics outperforms the other ones. This is not surprising compared with the CP-style B&B, but it shows that a more dynamic heuristics based on conflicts is not effective on this type of problems.

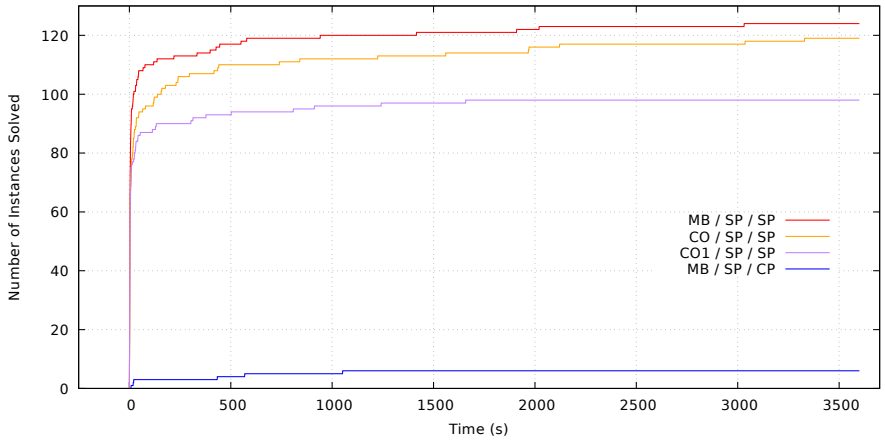

Fig. 7. Comparison of different heuristics on synthetic benchmarks

Then for the Constraint Game model, we have only used the combination `MB/SP/SP`, with and without improvement of the first solution by IBR. Results show that IBR improves branch and bound by giving a better first solution which is also an equilibrium.

instance	#Demands	#Nodes	#Edges	MB/SP/CP	MB/SP/SP	[NASH] MB/SP/SP
dfn-bwin	90	10	45	*TO*	0.670	3.871
dfn-gwin	110	11	47	*TO*	0.746	5.681
di-yuan	22	11	42	*TO*	0.472	2.012
giul39	1471	39	172	*TO*	26.554	1571.197
india35	595	35	80	*TO*	8.739	215.716
newyork	240	16	49	*TO*	3.486	18.173
nobel-eu	378	28	41	*TO*	4.919	41.861
norway	702	27	51	*TO*	7.962	154.520
pdh	24	11	34	*TO*	0.553	2.016
geant	462	22	36	2.729	2.550	2.92
germany	662	50	88	6.489	6.241	6.783
janos-us	650	26	84	3.508	3.605	5.174
janos-us-ca	1482	39	122	25.926	25.926	50.486
france	300	25	15	*TO*	*TO*	*TO*
pioro40	780	40	89	*TO*	*TO*	*TO*
polska	66	12	18	*TO*	*TO*	*TO*

Table 2. Results on real-world instances of SDNlib

We present in Table 2 the run-time comparison in second of the different strategies on the SNDlib instances. Each instance is described by its name (which corresponds usually to a network in a particular country), then the number of demands, nodes and links of the network. Without surprise, the improved branch and bound allows to solve

many instances up to optimality. On the Constraint Game side, it is interesting to see that games of unprecedented size (up to 1482 players in the *janos-us-ca* instance) can be solved up to optimality by Conga [15]. Interestingly, we have observed that IBR slightly degrades the time of computation, this is why we did not include the column in the table. We believe that in these problems, most first solutions computed by the `MB` heuristics were already at equilibrium, and thus adding IBR only adds another check.

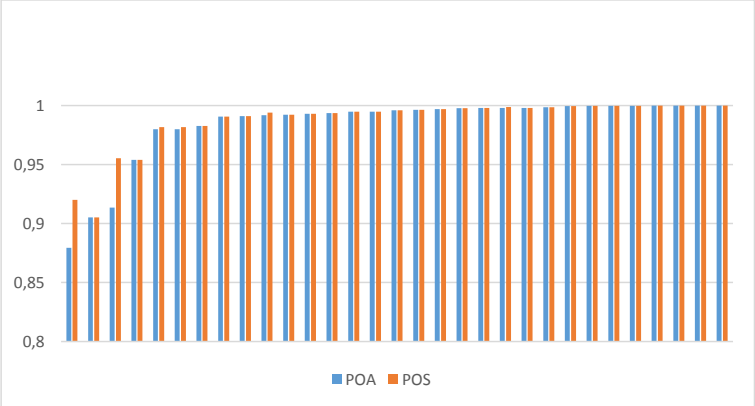

Fig. 8. Price of Anarchy and Price of Stability for small synthetic instances

We report the results for the computation of PoA and PoS for small synthetic instances in Figure 8. In most instances, we observe that the PoA and PoS are very close, and also very close to the centralized optimum. It means that on these problems, a decentralized algorithm would be very interesting to implement if we assume it scales up to larger problems. We have used much smaller instances because the PoA is very difficult to reach. The upper bound computed by the MST overestimates the longest path which also overestimates the longest shortest path. We pay these two approximations by a limited pruning of the search tree which has a major impact on the computation time.

7 Conclusion

This paper includes two practical contributions. First we have modeled and solved efficiently the unsplittable multi-commodity flow routing problem with congestion in Constraint Programming. We have provided an accurate branch and bound technique that allows to solve real-world size instances up to optimality. Our second contribution is a Constraint Game model that allows to evaluate the potential of decentralized routing in this context. Decentralized routing is crucial for the online version of the problem where demands come as a flow. We have found all Nash equilibria for problems with thousands of player thanks to the Constraint Game solver Conga. This is the first time that such large instances are solved up to optimality by a general-purpose Game Theory solver.

References

1. Azzouni, A., Boutaba, R., Pujolle, G.: Neuroute: Predictive dynamic routing for software-defined networks. In: 13th International Conference on Network and Service Management, CNSM 2017, Tokyo, Japan, November 26-30, 2017. pp. 1–6. IEEE Computer Society (2017), https://doi.org/10.23919/CNSM.2017.8256059
2. Chabrier, A., Danna, E., Pape, C.L., Perron, L.: Solving a network design problem. Annals OR 130(1-4), 217–239 (2004), https://doi.org/10.1023/B:ANOR.0000032577.81139.84
3. Even, S., Itai, A., Shamir, A.: On the complexity of time table and multi-commodity flow problems. In: Proceedings of the 16th Annual Symposium on Foundations of Computer Science. pp. 184–193. SFCS '75, IEEE Computer Society, Washington, DC, USA (1975), http://dx.doi.org/10.1109/SFCS.1975.21
4. Fudenberg, D., Tirole, J.: Game Theory. The MIT Press (1991)
5. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
6. Gottlob, G., Greco, G., Scarcello, F.: Pure Nash Equilibria: Hard and Easy Games. J. Artif. Intell. Res. (JAIR) 24, 357–406 (2005)
7. Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints 11(2-3), 199–219 (2006), https://doi.org/10.1007/s10601-006-7094-9
8. Layeghy, S., Pakzad, F., Portmann, M.: SCOR: constraint programming-based northbound interface for SDN. In: 26th International Telecommunication Networks and Applications Conference, ITNAC 2016, Dunedin, New Zealand, December 7-9, 2016. pp. 83–88. IEEE (2016), http://doi.ieeecomputersociety.org/10.1109/ATNAC.2016.7878788
9. Mendiola, A., Astorga, J., Jacob, E., Higuero, M.: A survey on the contributions of software-defined networking to traffic engineering. IEEE Communications Surveys and Tutorials 19(2), 918–953 (2017), https://doi.org/10.1109/COMST.2016.2633579
10. Nash, J.: Non-cooperative Games. Annals of Mathematics 54(2), 286–295 (1951)
11. Nguyen, T., Lallouet, A.: A Complete Solver for Constraint Games. In: O'Sullivan, B. (ed.) CP 2014, Lyon, France, September 8-12, 2014. LNCS, vol. 8656, pp. 58–74. Springer (2014), http://dx.doi.org/10.1007/978-3-319-10428-7_8
12. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. 43(2), 11:1–11:29 (Feb 2011), http://doi.acm.org/10.1145/1883612.1883618
13. Orlowski, S., Pióro, M., Tomaszewski, A., Wessäly, R.: SNDlib 1.0 – Survivable Network Design Library. In: Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium (April 2007), http://www.zib.de/orlowski/Paper/OrlowskiPioroTomaszewskiWessaely2007-SNDlib-INOC.pdf.gz, http://sndlib.zib.de, extended version accepted in Networks, 2009.
14. Osseiran, A., Braun, V., Hidekazu, T., Marsch, P., Schotten, H., Tullberg, H., Uusitalo, M.A., Schellman, M.: The foundation of the mobile and wireless communications system for 2020 and beyond: Challenges, enablers and technology solutions. In: Vehicular Technology Conference (VTC Spring), 2013 IEEE 77th. pp. 1–5. IEEE (2013)
15. Palmieri, A., Lallouet, A.: Constraint games revisited. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 729–735. ijcai.org (2017), https://doi.org/10.24963/ijcai.2017/101
16. Pape, C.L., Perron, L., Régin, J., Shaw, P.: Robust and parallel solving of a network design problem. In: Hentenryck, P.V. (ed.) Principles and Practice of Constraint Programming -

CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings. LNCS, vol. 2470, pp. 633–648. Springer (2002), https://doi.org/10.1007/3-540-46135-3_42
17. Pham, Q., Deville, Y.: Solving the longest simple path problem with constraint-based techniques. In: Beldiceanu, N., Jussien, N., Pinson, E. (eds.) Integration of Constraint Programming, Artificial Intelligence and Operations Research, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. LNCS, vol. 7298, pp. 292–306. Springer (2012), https://doi.org/10.1007/978-3-642-29828-8_19
18. Puri, A., Tripakis, S.: Algorithms for the multi-constrained routing problem. In: Penttonen, M., Schmidt, E.M. (eds.) Algorithm Theory - SWAT 2002, 8th Scandinavian Workshop on Algorithm Theory, Turku, Finland, July 3-5, 2002 Proceedings. Lecture Notes in Computer Science, vol. 2368, pp. 338–347. Springer (2002), https://doi.org/10.1007/3-540-45471-3_35
19. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3258, pp. 557–571. Springer (2004), https://doi.org/10.1007/978-3-540-30201-8_41
20. Rosen, J.B.: Existence and Uniqueness of Equilibrium Points for Concave n-Person Games. Econometrica 33(3), 520–534 (July 1965)
21. Rosenthal, R.W.: A class of games possessing pure-strategy nash equilibria. International Journal of Game Theory 2(1), 65–67 (1973)
22. Sellmann, M., Gellermann, T., Wright, R.: Cost-based filtering for shorter path constraints. Constraints 12(2), 207–238 (2007), https://doi.org/10.1007/s10601-006-9006-4
23. Shoham, Y., Leyton-Brown, K.: Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press (2009)
24. Vassilaras, S., Gkatzikis, L., Liakopoulos, N., Stiakogiannakis, I., Qi, M., Shi, L., Liu, L., Debbah, M., Paschos, G.: The algorithmic aspects of network slicing. IEEE Communications Magazine (2017)
25. Von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior. Princeton University Press (1944), http://jmvidal.cse.sc.edu/library/neumann44a.pdf
26. Yu, L., Lei, Y., Kacker, R., Kuhn, D.R.: ACTS: A combinatorial test generation tool. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 370–375. IEEE Computer Society (2013), https://doi.org/10.1109/ICST.2013.52