Problem Solving and Search in Artificial Intelligence

Exercise 1

Project Topic B: The Windy Postman Problem

Jakob Hitzelhammer, 11734084
Thomas Depian, 11807882

Summer term 2022 – 9th May 2022

Contents

1 Model

This section is intended to describe the model that we developed to solve the *Windy Postman Problem* using *Constraint Programming (CP)* techniques. In Section 1.1 we introduce our model and briefly argue why it is suited to solve the problem and in Section 1.2 we sketch other modeling ideas that we had and discuss their disadvantages due to which we rejected them.

1.1 Description of the Model

One key observation helps us in reducing the complexity of the problem and hence results in an succinct model.

Observation 1. *Given the* undirected *graph* $G = (V, E)$*, we can convert it to the* directed *graph* $G' = (V, A)$ *with* $A = \{(u, v), (v, u) \mid \{u, v\} \in E\}$ *without affecting the quality of the solution to the problem.*

While the correctness of the observation is imminent, in the end we are only concerned about the direction in which we travel over a particular edge, it simplifies the modeling of the problem formulation a lot. Because if we work with G' instead of G, we can directly associate the traversal of an edge in a particular direction to the corresponding arc. Hence, the weight function is now a simple function that maps an arc to a real number and must no longer differentiate between the travel direction. Therefore, the reduction mentioned in Observation 1 is a natural step towards a simpler model. What we have to keep in mind now though, is the fact that we have to alter the constraint regarding the validity of the solution. While we previously, i.e., in G, had to traverse each edge at least once, we now, i.e., in G', must traverse between each pair of adjacent vertices at least once, i.e., either in one direction, e.g., (u, v), or in the other, e.g., (v, u), but not necessarily in both.

Apart from Observation 1, we can furthermore observe that the solver is only required to determine the cost of the tour, not the tour, i.e., the sequence of edges or vertices, itself. This immediately leads to following observation regarding the quality of the solution, i.e., the objective function.

Observation 2. *To determine the quality of a solution, we are only required to know how often we traverse an edge (arc), the actual tour is irrelevant.*

Having both observations at our hand, we can state our model as follows.

Variables: Once we have reduced the undirected graph G to the directed graph G' as described in Observation 1, we create for each arc $(u, v) \in A(G')$ an integer variable $x_{u,v}$ that keeps track of the number of times we traverse the arc (u, v).

Constraints: Apart from the trivial constraints that require that each variable $x_{u,v}$ is a non-negative integer, we have two further constraints. The first one ensures the validity of the solution, i.e., that we travel each edge at least once. From our argumentation regarding Observation 1 follows that we have to add for each edge $\{u, v\} \in E(G)$ the constraint $x_{u,v} + x_{v,u} \geq 1$. Furthermore, to ensure that we find a proper tour in the graph, we require that for each vertex v holds, that the number of times we enter v is equal to the number of times we leave v, i.e., the sum over all $x_{u,v}$ is equal to the sum over all $x_{v,u}$ for u being the neighbors of v. While at first glance it might seem that this constraint (alone) does not prevent that we have subtours in the graph, e.g., two disconnected tours, in combination with the first constraint, this ensures that a proper

solution corresponds to a single tour in the graph. This can also be formally shown. For the sake of contradiction, assume that this would not be the case, i.e., that we have two disconnected tours $C_1 = \{c_{1_1}, \ldots, c_{1_k}\}$ and $C_2 = \{c_{2_1}, \ldots, c_{2_l}\}$. W.l.o.g. assume that C_1 and C_2 are maximal, i.e., they are not part of a bigger subtour. Since we have a valid solution, we must cover each edge at least once. And since the underlying graph is connected, there must exist an edge between the tours that we have to use. Assume w.l.o.g. that this is the edge $\{c_{1_1}, c_{2_1}\}$. This implies that we can walk from C_1 to C_2. However, as for each vertex the number of ingoing arcs must be identical to the number of outgoing arcs, this must also be the case for C_1 and C_2, i.e., the number of arcs that go "into" (vertices of) C_1 must be the same as the number of arcs that "leave" (vertices of) C_1. Since we have shown that we must be able to leave C_1 and enter C_2, and since C_1 and C_2 are maximal, we must also be able to leave C_2 and enter C_1. A contradiction to the assumption that they are disconnected subtours. In case of multiple subtours, the argument can then be extended in an inductive manner to show that indeed we always get a valid tour.

Objective Function: Since we have a constraint *optimization* problem, we need an objective function which we want to optimize for. From the way we transferred the graph (see Observation 1) and the way we set up the variables (which follows from Observation 2), the objective function can be defined as follows, where $c_{u,v}$ denotes the cost of traversing the edge $\{u, v\} \in E(G)$ in the direction from u to v.

$$f := \sum_{(u,v) \in A(V')} c_{u,v} \cdot x_{u,v}$$

Clearly, we want to minimize f, i.e., $\min f$ is our objective.

The correctness of the model follows from our argumentation that we gave above. Regarding the realization, i.e., implementation, of the model we give some details in Section 2. In the following, we will sketch some approaches we had, to give an idea on what could be possible and what the limitations of these formulations are.

1.2 Discussion of Alternative Formulations

Before coming up with the model described in Section 1.1, we first tried to model the problem either as a permutation (with repetition) of the edges or vertices.

Especially the permutation of the edges is, according to us, a natural way of modeling the solution of the problem, as a tour is essentially a sequence of edges. But while formulating the constraints that ensure the validity of the solution is not the problem, we identified two major limitations with this model. First, the formulation of the objective function is not as clear/direct as in the model we propose, since the direction in which we travel the edge is not directly encoded in the solution but must be derived by looking at the previous and/or following edge in the permutation. And second, since we might have to visit an edge multiple times, we cannot bound the length of the sequence/permutation and hence the number and meaning of the variables in this model is highly non-trivial.

Similar problems arise when we try to formulate the problem via a permutation (with repetition) of the vertices of the graph. This generalizes in some sense the first mentioned alternative as the direction in which we travel over an arc can be read off more directly from the solution. However, this approach has the same limitation regarding the number of variables as the previous one.

Due to the limitations of both alternative approaches, we ended up with the model described in Section 1.1.

2 Implementation

In this section we state the programming language we used to implement our model in addition to the solver we selected. As the implementation of the model itself is a one-to-one translation of the description in Section 1 we actively refrain from repeating it but rather highlight unusual parameters or deviations from "naive" implementations.

We used the constraint satisfaction/optimization solver which comes with the *OR-Tools* from Google. A version of the OR-Tools for Python 3 can be obtained via PIP and we used the latest version, which was at the time of creation version `9.3.10497`. [2]

OR-Tools provide the class `CpModel` which can be populated with variables and constraints as required and serves then as the input for the actual solver. Noteworthy is that for integer variables, which we require, we have to explicitly restrict the domain by stating a lower and upper bound. While the lower bound can be set to `0`, finding a value for the upper bound is non-trivial and we leave this as a "parameter" for the solver (see therefore Section 3.1).

2.1 Retrieval of the Tour

The attentive reader might notice that our model actually does not "compute" a tour but only guarantees that a tour with the given objective value exists. The actual tour hides in the variable assignment. Therefore, we provide here a way to compute a possible tour from the variable assignment. For a variable $x_{u,v}$ with value y, we say that y is the budget of the arc (u, v) and write $x_{u,v} = y$. On an abstract way, we search for an Euler Tour in the multigraph $G'_M = (V, A')$ with $A' = \{(u, v)_i \mid (u, v) \in A, i = 1, \ldots, x_{u,v}\}$, i.e., in the directed graph where we replace each arc with the number of identical arcs according its budget. Since we ensured in Section 1.1 that the number of times we enter a vertex equals to the number of times we leave it, each vertex in G'_M has the same in- and outdegree. Hence, an Euler Tour must exist. We can therefore use and adapt the approach of finding an Euler Tour in a directed graph to use it in our model. Please refer to Algorithm 1 for the pseudo code. The idea of Algorithm 1 is as follows. We start at an arbitrary vertex v and follow the outgoing arcs with budget > 0 to create a tour T, until we eventually arrive again at v, or any other vertex $u \in T$. Once we arrived at such a vertex, we have found a cycle $C \subseteq T$. We store C and decrease the budget for each arc $a \in C$ by one, as we have used this arc once in C. We repeat this process, starting each time at a vertex v with an outgoing arc with non-zero budget, until we eventually arrive at a set $\mathcal{C}$ of cycles C. Note that we maintain the necessary condition for a closed

Algorithm 1: Retrieval of Windy Postman-Tour

Input: Solution S of CP-Solver where each variable $x_{u,v}$ has budget y
Output: Possible tour $\mathcal{T}$ with cost of a tour as described by S

$\mathcal{C} \leftarrow \emptyset$;
$\mathcal{T} \leftarrow \emptyset$;
while *there is a variable* $x_{v,u} > 0$ **do**
  $v \leftarrow$ vertex with outgoing arc (v, u) with $x_{v,u} > 0$;
  $T \leftarrow \{v\}$;
  $n \leftarrow u$;
  while $n \notin T$ **do**
    $T \leftarrow T \cup \{n\}$;
    $(n, w) \leftarrow$ arc outgoing from n with $x_{n,w} > 0$;
    $n \leftarrow w$;
  end
  $C \leftarrow$ cycle in T starting from n;
  reduce budget of arcs induced by C by 1;
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$;
end
$\mathcal{T} \leftarrow$ cycle $C \in \mathcal{C}$;
$\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\}$;
while $\mathcal{T}$ *is not fully traversed* **do**
  $v \leftarrow$ next vertex in traversal;
  if v *appears in* $C' \in \mathcal{C}$ **then**
    append C' at vertex v to $\mathcal{T}$ (also at vertex v);
    extend traversal of $\mathcal{T}$ by C;
    $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C'\}$;
  end
end
return $\mathcal{T}$;

Euler Tour as an invariant throughout this process, which guarantees us that we will eventually find each of the cycles. Note furthermore that $\mathcal{C}$ is not unique, however, as the cost of the tour only depends on the edges we use and the direction in which we travel them, which is given by the budget of the arcs, the overall cost of the tour will not change. Having $\mathcal{C} = \{C_1, \dots C_k\}$ we can build our final tour $\mathcal{T}$ as follows. We set $\mathcal{T} = C_1$ and traverse C_1. Once we arrive at a vertex $v \in C_1$ that is also part of another cycle $C_i \in \mathcal{C}$ that we *have not* considered yet, we insert C_i in $\mathcal{T}$ at that position, and continue to traverse C_i, repeating the steps as before. Once we have fully considered C_i, we will eventually arrive again at $v \in C_1$, where we continue to consider the remaining vertices in C_1. If we consider each cycle exactly once, we will in the end have a tour $\mathcal{T}$ with a cost as given by the objective value of the solution found by the solver. Observe how we exploit the arguments that we gave already in Section 1.1, hence the correctness of Algorithm 1 follows immediately. Regarding the running time of Algorithm 1 we can observe that the construction of $\mathcal{C}$ depends on the total budget, i.e., the sum of the budgets W of all arcs, and the number of vertices n, as we have to check whether a vertex n is already part of the tour we create. As we decrease in each iteration the budget of at least one arc by at least one, the loop runs in $\mathcal{O}(Wn)$. A similar argument can be given in the second part of Algorithm 1, where we construct the final tour $\mathcal{T}$. As W is polynomial in the number of edges, this final retrieval of the solution is polynomial in the input + solution size and therefore should not dominate the overall running time, which is most likely dominated by the running time of the solver for finding a (good) solution to an NP-hard problem.

3 Experiments

We experimented with different parameters and configurations of the solver in order to evaluate and compare its efficiency depending on the configuration. We start with describing the parameters in Section 3.1, followed by the used instances in Section 3.2 and complete this section with stating the results of our experiments in Section 3.3.

3.1 Parameters and Configurations

To ensure comparability among the tests, we forced the solver to use only one worker-/thread, i.e., we set `num_workers = 1`. By doing this we try to minimize the variation of the result depending on the scheduling by the operating system. Furthermore, as we are dealing with an NP-complete problem, we cannot expect to find the optimal solution in a reasonable amount of time for all instances. Inspired by the real-world, where we cannot wait hours to find a solution, we give the solver a hard upper bound on the running time of five minutes, i.e., `max_time_in_seconds = 300`, and try to see with which configuration we can find in limited time the best results. Each test was performed twice to reduce the influence of outliers, where we in addition recreated the instance of the model and solver after each run. The experiments were performed on an AMD Ryzen 7 5800X 8-Core Processor, with 32GB of RAM. The software ran under Windows 10 Pro

with Python 3.9. After each run, we collect the objective value of the solution as well as the number of created conflicts and explored branches as reported by the solver.

In the following, we state the parameters of the solver and the model that we varied for the experiments. We describe them here briefly, where we use the documentation of the OR-Tools [1], i.e., the `proto`-file of the model, and the user's manual [3, Chapter 5.6] as reference.

Variable Selection Strategy: This parameter influences which of the unassigned variables will be considered next. For this parameter, we select the values `CHOOSE_LOWEST_MIN`, `CHOOSE_MIN_DOMAIN_SIZE`, and `CHOOSE_MAX_DOMAIN_SIZE` for our experiments. When selecting the first value, the solver selects among all unassigned variables the one, to which we could potentially assign the smallest value. We think that this is a natural value for this parameter, as we are dealing with a minimization problem. However, observe that in our model (nearly) all variables have (for most of the time) the same smallest possible value. Hence, this selection strategy can degenerate to a variant of "simply select next unassigned variable". To circumvent this, we use the other two values, which select the variable that has the least/most possible values left in its domain. However, to use this strategy in a meaningful way, we should vary the domain size of the variables (see the last parameter). Also here we expect that the value that prefers smaller values leads to better results.

Domain Reduction Strategy: The second parameter is tightly coupled to the first one. After we select a variable, we have to decide which value we assign to it. This is influenced by this parameter. We consider the values `SELECT_MIN_VALUE`, `SELECT_LOWER_HALF`, and `SELECT_UPPER_HALF` in our experiments. Selecting the first option, the solver will assign to a selected variable the lowest possible value left in its domain, whereas for the other two options the solver will partition the domain into a lower and an upper half, and a value out of the lower or upper half, respectively, is then selected for the assignment. While the first option would be a natural selection for a minimization problem, we do not expect good results from it, as the main complexity of the problem hides in selecting some edge-directions several times. Assigning to each edge-direction the smallest possible value seems therefore not that sensible. However, from the degree of freedom, which is given with the other two parameters, we expect better results, where intuitively `SELECT_LOWER_HALF` should give among the tree options the best results.

Upper Bound on the Domain Size for the Variables: Determining the upper bound of the domains of the variables is a critical part for our model, since this parameter influences the overall search space as well as the variable selection and domain reduction strategy. For C1 and C2 we tested beforehand the parameter with values of the range 10-10000, resulting in 100 having the best objective value. Having that, it seems sensible to assign different upper bounds to different edge-directions, depending on their cost. This means that we determine the upper bound of each variable/edge w.r.t. the cost of the edge, such that edges with

Identifier	Parameters		
	variable_selection	domain_reduction	upper_bound
C1	CHOOSE_LOWEST_MIN	SELECT_MIN_VALUE	$[100, 100]$
C2	CHOOSE_LOWEST_MIN	SELECT_LOWER_HALF	$[100, 100]$
C3	CHOOSE_MIN_DOMAIN_SIZE	SELECT_UPPER_HALF	$[3, 10]$
C4	CHOOSE_MAX_DOMAIN_SIZE	SELECT_LOWER_HALF	$[3, 10]$
C5	CHOOSE_MIN_DOMAIN_SIZE	SELECT_LOWER_HALF	$[3, 10]$

Table 1: The selected configurations with which we perform the tests.

cheaper cost can be traversed more often than expensive ones. This also allows us to try different variable selection strategies, e.g., `CHOOSE_MIN_DOMAIN_SIZE` or `CHOOSE_MAX_DOMAIN_SIZE`. To achieve this, we scale the edge costs for each instance from the range, cheapest edge to most expensive edge, to the range, 3 to 10, such that cheap edges have a higher upper bound and expensive ones a lower upper bound.

The above parameters give rise to a variety of possible configurations, out of which we select the one from Table 1. Due to space constraints, we write `variable_selection` instead of `variable_selection_strategy`, and `domain_reduction` instead of `domain_reduction_strategy`.

Remark: We would also like to mention that we tried to add a redundant constraint as we thought it might improve the results/runtime. We have added a global constraint that ensures that at most half of the variables can be assigned 0, since we need to traverse each edge at least once. We thought that this would, especially in C1, where we use min value and therefore try 0 first on each variable, enforce a conflict faster and therefore result in the solver not trying out irrelevant branches. But the constraint actually lead to a much worse performance. It could be the case that the solver itself learns this behavior pretty fast independently of our redundant constraint and our constraint would only lead to more bookkeeping for the solver, which could have a negative impact on its "runtime" and hence also on the solution quality that it can find in 5 minutes.

3.2 Used Instances

The provided dataset contains 121 instances, from which we exclude the toy instance, which is intended for testing purposes. The remaining 120 instances can be partitioned into five sets, depending on the number of vertices the graph has. They are as follows.

500 1000 1500 2000 3000

Each set consists of 24 instances. To ensure that the results of the experiments are representative, we select 15 out of the 120 instances, where we ensure that we select

Number of vertices (n)				
500	1000	1500	2000	3000
WA0555	WA1032	WA1545	WA2032	WA3031
WA0561	WA1035	WB1545	WA2061	WA3041
WB0542	WA1042	WB1555	WB2031	WA3052

Table 2: Selected instances partitioned by the number of vertices they contain and sorted alphabetically.

from each partition 3 instances. The instances were selected randomly and are stated in Table 2 (sorted alphabetically).

3.3 Results

Below we state the result of our experiments. As already mentioned, we collected three measurements, the objective value, the number of created conflicts and the number of explored branches. We will in the following sections discuss the results in further detail. Section 3.3.1 evaluates the experiments w.r.t. the objective value of the solutions, whereas Section 3.3.2 evaluates them w.r.t. the number of created conflicts and explored branches. Generally speaking, we noticed the following. For all problem instances and configurations, the solver were able to obtain *some* solution, however, even for small instances the solver was not able to provide an optimal solution. Hence, the solver reported for all tests `FEASIBLE` and never `OPTIMAL`. We could furthermore observe that the measurements did not differ significantly between the two runs for one configuration and instance. While this is interesting, as not all configurations result in a deterministic behavior for the solver, this can be explained by the fact that the solver might cache partial results etc. outside of the instance, i.e., in the second run the solver might have used results from the previous run to guide its search. As a consequence, we consider for our evaluation always the best measurement of the two runs.

3.3.1 Objective Value of the Solution

In general, it can be observed that the obtained solution qualities differ only marginally across the configurations. However, some tendencies can still be seen. We provide the minimum objective value obtained in the experiments in Table 3. We highlight the two best objective values for convenience. Furthermore, we plot the objective values as bar-charts to better see the relative difference. As the objective values across instances varies, we group instances with similar objective values into one plot and adapt the y-axis so that the difference among the solution qualities can be better seen. However, one has now to be aware of that the difference in the size of the bars is not comparable across different plots. The plots are given in Figure 1. When considering Table 3 and Figure 1, we can see that none of the configurations outperformed the other configurations in terms of solution quality. However, it turns out that on smaller instances, configuration C1 and

	Configuration				
Instance	C1	C2	C3	C4	C5
WA0555	650254	**649289**	650336	649715	650793
WA0561	816623	**815816**	817393	821398	816848
WB0542	48667	**48521**	49005	48677	48742

(a) $n = 500$ vertices.

	Configuration				
Instance	C1	C2	C3	C4	C5
WA1032	1018309	1018380	1020974	**1018104**	1019462
WA1035	**943977**	944282	944818	946601	944399
WA1042	1176058	**1173448**	1182611	1177658	1179703

(b) $n = 1000$ vertices.

	Configuration				
Instance	C1	C2	C3	C4	C5
WA1545	1633099	**1666035**	1627580	1633195	1629835
WB1545	79574	**77663**	78181	79206	78696
WB1555	101087	100995	99666	99779	**98796**

(c) $n = 1500$ vertices.

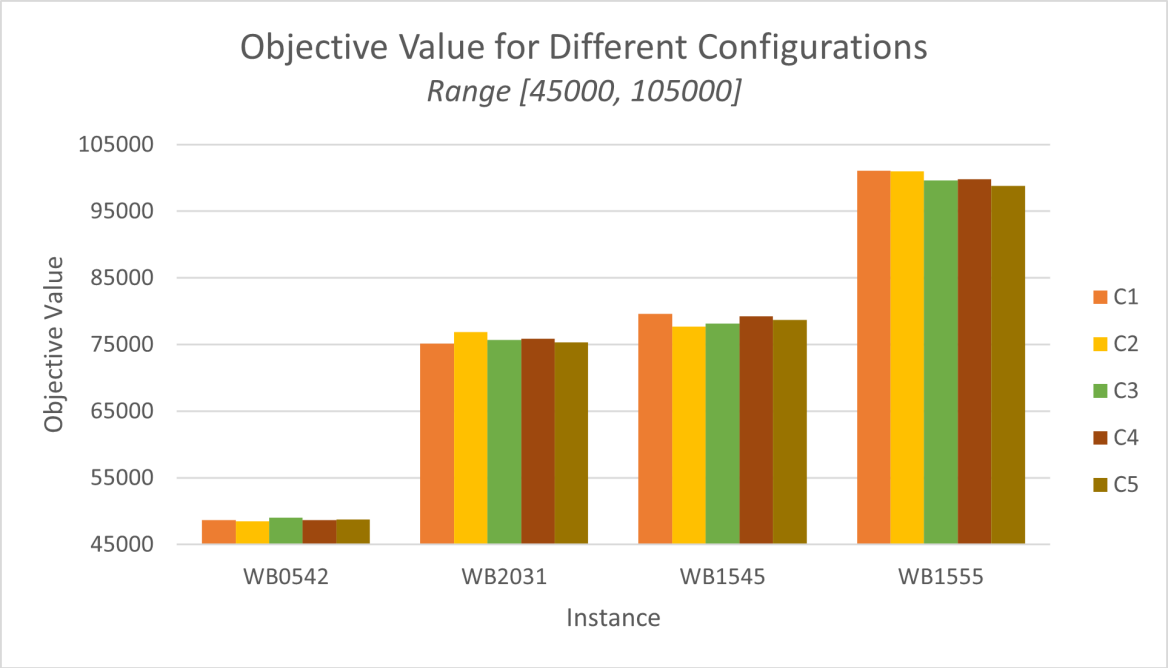
	Configuration				
Instance	C1	C2	C3	C4	C5
WA2032	2070914	2087441	2067198	2071420	**2060365**
WA2061	3353992	3414515	**3296050**	3317413	3300988
WB2031	**75141**	76905	75714	75834	75349

(d) $n = 2000$ vertices.

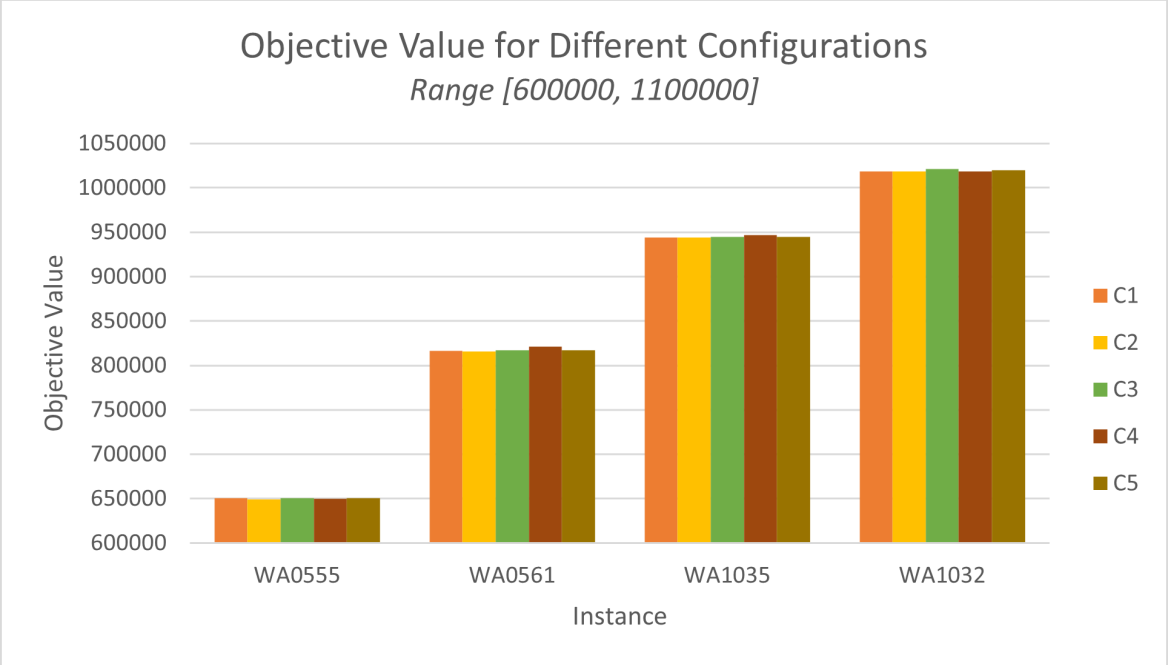
	Configuration				
Instance	C1	C2	C3	C4	C5
WA3031	3138456	3142497	3143381	**3137467**	3137525
WA3041	3825402	3694637	3674550	3658770	**3652633**
WA3052	4235573	4224333	4194967	**4193535**	4200235

(e) $n = 3000$ vertices.

Table 3: Objective values obtained during our result – split up by instance size. The **best** and second-best objective values for each instance are highlighted.

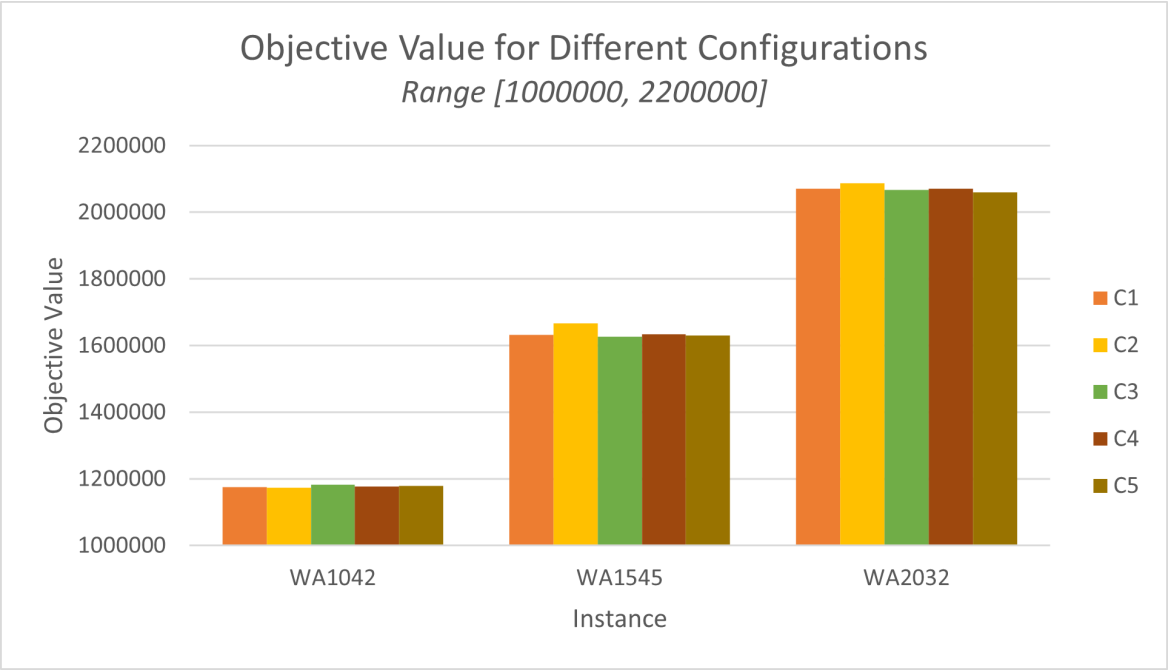


(a) Range 45000 – 105000.

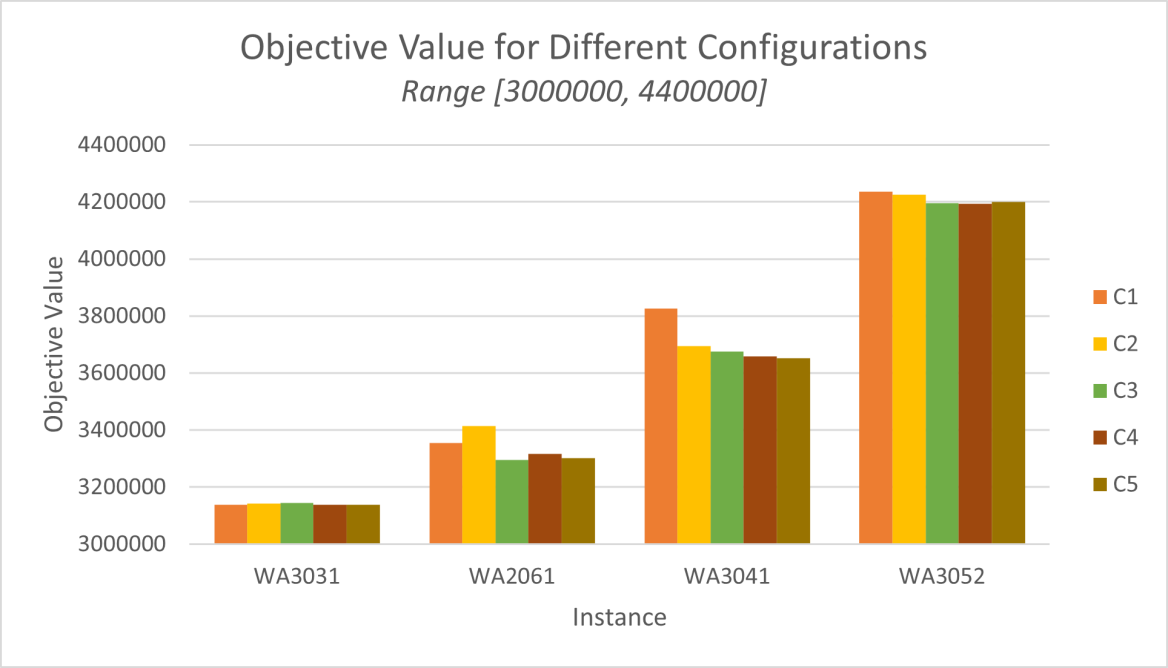


(b) Range 600000 – 1100000.

Figure 1: Influence of the configuration on the objective value – split up by range of the objective value. Note: y-axis not at uniform scale across different plots.



(c) Range 1000000 – 2200000.



(d) Range 3000000 – 4400000.

Figure 1 (continued)

C2 seem to perform better, whereas on larger instances this is the case with configuration C4 and C5. One reason for this could be the following. Especially configuration C1, but also configuration C2, perform their search in an "incremental" way, meaning that they try to assign to each variable one of the lowest possible values and, if it turns out that this is not successful, try the next larger value. This can be seen when we look at the option for the variable selection strategy in configuration C1 and C2. Both of them choose the variable, or one of the variables if there are multiple, which has the minimum lower value in the domain, i.e., allows the lowest value. However, when selected such a variable, the value it gets assigned in C1 is exactly this value, whereas for C2 it is one in the lower half. In Table 3a and Table 3b we can see that for small instance sizes, where we also have not too many variables, for instance in `WA0555` we only have around 2500 variables, this rather enumerative approach is promising. Especially if we add some degree of freedom in choosing the assignment to a variable, this seems promising. This rather "naive" configuration then also outperforms the more sophisticated ones. However, as the instance size enlarges, this approach becomes less promising (see for example Table 3d or Table 3e). This is probably due to the increasing number of variables, which leads to many illegal, i.e., unsatisfying, or unnecessary, i.e., high tour cost, assignments. If for example for some variable $x_{u,v}$ it holds that we need an assignment of 5 in order to obtain a good solution, it takes now considerably longer until the solver can, according to the variable and domain selection strategy, perform this assignment. Which, given the hard limit of five minutes, leads then to low-quality solutions. But if we "guide" the solver with domain sizes adapted to reasonable assignments, it appears that the solver can handle well also larger domain sizes. One further (intuitive) explanation of this is that with this varying domain size of the variables, and a variable selection strategy that depends on the domain size, the solver tries first to fix the "cheap" or "expensive" edges, depending on whether we select variables according to max or min domain size, respectively, before going on with the other edges. That this guidance in combination with the variable and value selection strategy should work better especially for larger instances was also what we expected in the first place. However, unexpected was that configuration C5 performs that well w.r.t. the solution quality, as it has a rather unintuitive variable selection strategy, that considers the expensive edge(-direction)s first. Although one would expect that this leads to worse solutions, we can see that considering variables that have a small domain size, which will be in our case the expensive edge-directions, in combination with selecting a value from the lower half, which will be in our case for such variables most often some value in $[0, 2]$, we force expensive edges to be used only a few times and can then "build" valid solutions around this partial assignment with the help of cheaper edge-directions, where we can "afford" to use them more often. However, if we would do it the other way round, as in C4, one could arrive at a partial-solution where we need to use rather expensive edge-directions in order to complete it to a valid solution. So overall, it seems that C5, i.e., the guided domain size in combination with fixing expensive edges first, is a good strategy for finding acceptable solutions also in limited time.

3.3.2 Number of Created Conflicts and Explored Branches

Overall, it can be said that all 5 configurations have different numbers of conflicts/explored branches and, depending on the instance, sometimes one configuration has more conflicts/branches, sometimes the other. However, as we can see in Figure 2 and 3, one trend that can be observed is that C1 and C2 tend to have fewer conflicts/branches than C3, C4 and C5. Our intuition was that the reason behind this pattern lies in the domain size, since this is the main difference between those configurations. Therefore we tried C1 and C2 with a lower upper bound which indeed led to a higher number of conflicts/ branches but also worse objective values. The reason for this could be that the presolving step takes significantly longer or the steps in the constraint learning process simply take longer when the domain size is larger. However, it is difficult to find out exactly if that is the case, since the documentation of the CP-SAT solvers is not very insightful. What can also be seen, and was also expected, is that the number of conflicts becomes significantly smaller as the number of variables increases (from left to right the size of the instances increases, see Figure 2). One reason for this is probably that with larger instance sizes, one step, i.e., selecting one variable and assigning a variable to it, takes more time due to more variables. Hence, the solver can, in the same amount of time, perform fewer of these steps.

References

[1] OR-Tools cp_model.proto – GitHub. `https://raw.githubusercontent.com/google/or-tools/23e3a7b9891ecf44e7ec6b9d97b82a66832ba7df/ortools/sat/cp_model.proto`. Last accessed: 2022-05-08.

[2] ortools 9.3.10497 – PyPi. `https://pypi.org/project/ortools/9.3.10497/`. Last accessed: 2022-05-08.

[3] Nikolaj van Omme, Laurent Perron, and Vincent Furnon. or-tools user's manual. `https://acrogenesis.com/or-tools/documentation/user_manual/index.html`, 2014. Last accessed: 2022-05-08.

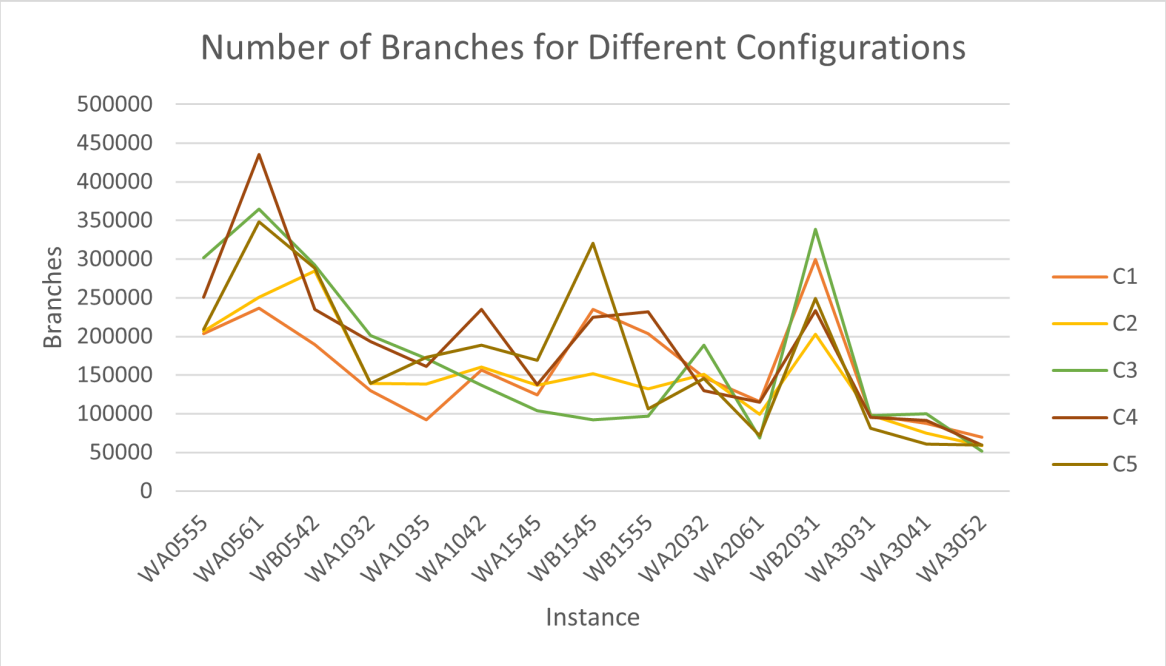


Figure 2: Number of explored branches for different Configurations.

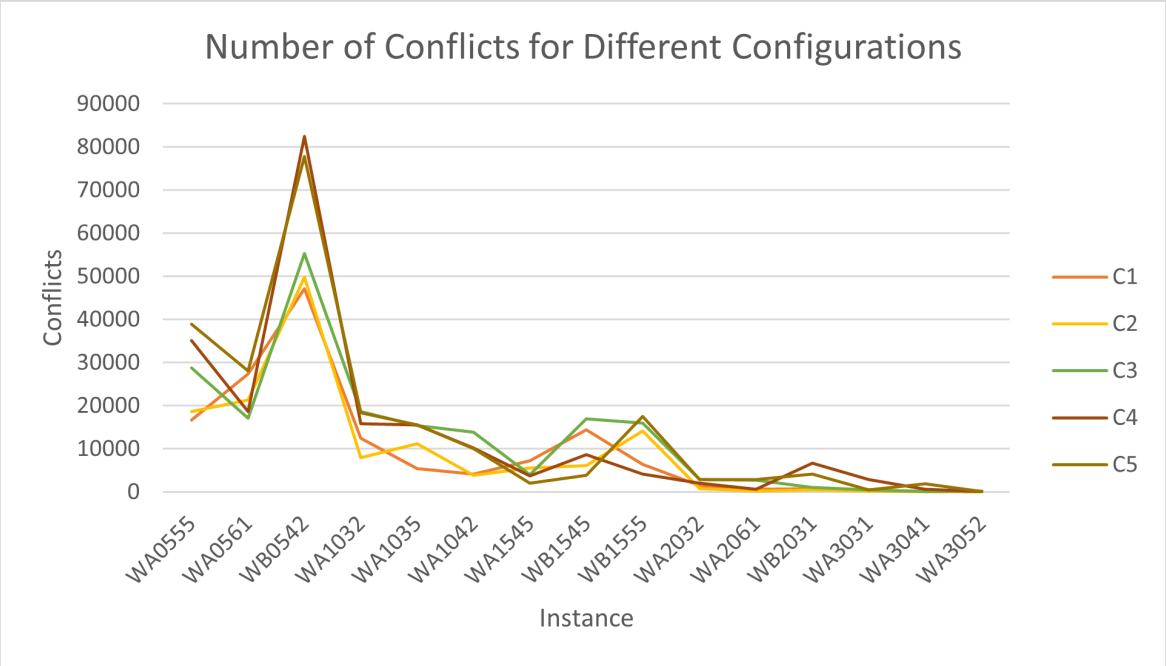


Figure 3: Number of conflicts for different Configurations.