Generating subtour constraints for the TSP from pure integer solutions

Ulrich Pferschy* Rostislav Staněk*

Abstract

The *traveling salesman problem* (*TSP*) is one of the most prominent combinatorial optimization problems. Given a complete graph G = (V, E) and non-negative distances d for every edge, the TSP asks for a shortest tour through all vertices with respect to the distances d. The method of choice for solving the TSP to optimality is a *branch-and-bound-and-cut approach*. Usually the *integrality constraints* are relaxed first and all separation processes to identify violated inequalities are done on *fractional solutions*.

In our approach we try to exploit the impressive performance of current ILP-solvers and work only with integer solutions without ever interfering with fractional solutions. We stick to a very simple ILP-model and relax the *subtour constraints* only. The resulting problem is solved to integer optimality, violated constraints (which are trivial to find) are added and the process is repeated until a feasible solution is found.

In order to speed up the algorithm we pursue several attempts to find as many *relevant* subtours as possible. These attempts are based on the clustering of vertices with additional insights gained from empirical observations and random graph theory. Computational results are performed on test instances taken from the *TSPLIB95* and on *random Euclidean graphs*.

Keywords. traveling salesman problem; subtour constraint; ILP solver; random Euclidean graph

1 Introduction

The *Traveling Salesman/Salesperson Problem TSP* is one of the best known and most widely investigated combinatorial optimization problems with four famous books entirely devoted to its study ([11], [18], [8], [2]). Thus, we will refrain from giving extensive references but mainly refer to the treatment in [2]. Given a complete graph $G = (V, E)$ with $|V| = n$ and $|E| = m = n(n-1)/2$, and nonnegative distances d_e for each $e \in E$, the TSP asks for a shortest tour with respect to the distances d_e containing each vertex exactly once.

*{pferschy, rostislav.stanek}@uni-graz.at. Department of Statistics and Operations Research, University of Graz, Universitätsstraße 15, A-8010 Graz, Austria

Let $\delta(v) := \{e = (v, u) \in E \mid u \in V\}$ denote the set of all edges adjacent to $v \in V$. Introducing binary variables x_e for the possible inclusion of any edge $e \in E$ in the tour we get the following classical ILP formulation:

$$\text{minimize} \quad \sum_{e \in E} d_e x_e \tag{1}$$

$$\text{s.t.} \quad \sum_{e \in \delta(v)} x_e = 2 \qquad \forall\, v \in V, \tag{2}$$

$$\sum_{\substack{e=(u,v)\in E \\ u,v \in S}} x_e \leq |S| - 1 \qquad \forall\, S \subset V, S \neq \emptyset, \tag{3}$$

$$x_e \in \{0, 1\} \qquad \forall\, e \in E \tag{4}$$

(1) defines the *objective function*, (2) is the obvious *degree equation* for each vertex, (3) are the *subtour constraints*, which forbid solutions consisting of several disconnected tours, and finally (4) defines the *integrality constraints*. Note also that some subtour constraints are redundant: For the vertex sets $S \subset V$, $S \neq \emptyset$, and $S' = V \backslash S$ we get pairs of subtour constraints both enforcing the connection of S and S'.

The established standard approach to solve TSP to optimality, as pursued successfully during the last 30+ years, is a branch-and-cut approach, which solves the LP-relaxation obtained by relaxing the integrality constraints (4) into $x_e \in [0, 1]$. In each iteration of the underlying branch-and-bound scheme cutting planes are generated, i.e. constraints that are violated by the current fractional solution, but not necessarily by any feasible integer solution. Since there exists an exponential number of subsets $S \subset V$ implying subtour constraints (3), the computation starts with a small collection of subsets $S \subset V$ (or none at all), and identifies violated subtour constraints as cutting planes in the so-called separation problem. Moreover, a wide range of other cutting plane families were developed in the literature together with heuristic and exact algorithms to find them (see e.g. [20, ch. 58], [2]). Also the undisputed champion among all TSP codes, the famous *Concorde* package (see [2]) is based on this principle.

In this paper we introduce and examine another concept for solving the TSP. In Section 2 we introduce the basic idea of our approach. We follow with some improvement strategies in Section 3 with our best approach presented in Subsection 3.6. Since the main tool of this paper are computational experiments, we discuss them in detail in Section 4. The common details of all these tests will be given in Subsection 4.1. In Section 5, we deal with some theoretical results and further empirical observations. Finally, we provide an Appendix with many illustrations, graphs and two summarizing tables (Tables 8 and 9).

2 Solving TSP in integers

Clearly, the performance of the above branch-and-cut approach depends crucially on the performance of the used LP-solver. Highly efficient LP-solvers have been available for

quite some time, but also ILP-solvers have improved rapidly during the last decades and reached an impressive performance. This motivated the idea of a very simple approach for solving TSP without using LP-relaxations explicitly.

The general approach works as follows (see Algorithm 1). First, we relax all subtour constraints (3) from the model and solve the remaining ILP model (corresponding to a *weighted 2-matching problem*). Then we check if the obtained integer solution contains subtours. If not, the solution is an optimal TSP tour. Otherwise, we find all subtours in the integral solution (which can be done by a simple scan) and add the corresponding subtour constraints to the model, each of them represented by the subset of vertices in the corresponding subtour. The resulting enlarged ILP model is solved again to optimality. Iterating this process clearly leads to an optimal TSP tour.

Input: TSP instance
Output: an optimal TSP tour

```
1: define current model as (1), (2), (4);
2: repeat
3:   solve the current model to optimality by an ILP-solver;
4:   if solution contains no subtour then
5:     set the solution as optimal tour;
6:   else
7:     find all subtours of the solution and add the corresponding subtour constraints
       into the model;
8:   end if
9: until optimal tour found;
```

Algorithm 1: Main idea of our approach.

Every execution of the ILP-solver (see line 3) will be called an *iteration*. We define the *set of violated subtour constraints* as the set of all included subtour constraints which were violated in an iteration (see line 7). An example illustrating the execution of this algorithm is given in Figures 3 and 6 – 17 in the Appendix.

It should be pointed out that the main motivation of this framework is its simplicity. The separation of fractional subtour inequalities amounts to the solution of a max-flow min-cut problem. Based on the procedure by Padberg and Rinaldi [15], extensive work has been done to construct elaborate algorithms for performing this task efficiently. In contrary, violated subtour constraints of integer solutions are trivial to find. Moreover, we refrain from using any other additional inqualities known for classical branch-and-cut algorithms, which might also be used to speed up our approach, since we want to underline the strength of modern ILP-solvers in connection with a refined subtour selection process (see Section 3.6).

This approach for solving TSP was available since the earliest ILP formulation going back to [6] and can be seen as folklore nowadays. Several authors followed the concept of generating integer solutions for some kind of relaxation of an ILP formulation and iteratively adding violated integer subtour constraints. However, it seems that the lack

of fast ILP-solvers prohibited its direct application in computational studies although it was used in an artistic context by [4].

Miliotis [12] also concentrated on generating integer subtour constraints, but within a fractional LP framework. The classical paper by Crowder and Padberg [5] applies the iterative generation of integer subtour constraints as a second part of their algorithm after generating fractional cutting planes in the first part to strengthen the LP-relaxation. They report that not more than three iterations of the ILP-solver for the strengthened model were necessary for test instances up to 318 vertices. Also Grötschel and Holland [7] follow this direction of first improving the LP-model as much as possible, e.g. by running preprocessing, fixing certain variables and strengthening the LP-relaxation by different families of cutting planes, before generating integer subtours as last step to find an optimal tour. It turns out that about half of their test instances never reach this last phase. In contrast, we stick to the pure ILP-formulation without any previous modifications.

From a theoretical perspective, the generation of subtours involves a certain trade-off. For an instance (G, d) there exists a minimal set of subtours $\mathcal{S}^*$, such that the ILP model with only those subtour constraints implied by $\mathcal{S}^*$ yields an overall feasible, and thus optimal solution. However, in practice we can only find collections of subtours larger than $\mathcal{S}^*$ by adding subtours in every iteration until we reach optimality. Thus, we can either collect as many subtours as possible in each iteration, which may decrease the number of iterations but increases the running of the ILP-solver because of the larger number of constraints. Or we try to control the number of subtours added to the model by trying to judge their relevance and possibly remove some of them again, which keeps the ILP-model smaller but may increase the number of iterations. In the following we describe various strategies to find the "right" subtours.

2.1 Representation of subtour constraints

Clearly, the performance of the suggested approach depends crucially on the performance of the underlying ILP-solver. In this respect it should be noted that the subtour constraints (3) can be expressed equivalently by the following cut constraints:

$$\sum_{\substack{e=(u,v)\in E \\ u\in S, v\notin S}} x_e \geq 2 \qquad \forall\, S \subset V, S \neq \emptyset \tag{5}$$

Although mathematically equivalent, the two versions of forbidding a subtour in S may result in quite different performances of the ILP-solver.

It was observed that in general the running time for solving an ILP increases with the number of non-zero entries of the constraint matrix. Hence, we also tested a hybrid variant which chooses between (3) and (5) by picking for each considered set S the version with the smaller number of nonnegative coefficients on the left-hand side as follows:

$$\begin{array}{lll} \sum_{\substack{e=(u,v)\in E \\ u,v\in S}} x_e \leq |S| - 1 & \multirow{2}{*}{$\forall\, S \subset V, S \neq \emptyset$} & \text{if } |S| \leq \frac{2n+1}{3} \\ \sum_{\substack{e=(u,v)\in E \\ u\in S, v\notin S}} x_e \geq 2 & & \text{if } |S| > \frac{2n+1}{3} \end{array} \tag{6}$$

We performed computational tests of our approach to compare the three representations of subtour constraints, namely (3), (5) and (6), and list the results in Table 1. Technical details about the setup of the experiments can be found in Subsection 4.1.

instance	**s.c. as in (3)**			**s.c. as in (5)**			**s.c. as in (6)**		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
kroA150	89	12	82	75	12	82	**62**	**12**	**82**
kroB150	**52**	**13**	**77**	237	13	77	54	13	77
u159	**9**	**5**	**39**	13	5	39	**9**	**5**	**38**
brg180	62	14	56	**36**	**5**	**29**	64	16	67
kroA200	2153	11	95	**1833**	**11**	**95**	2440	11	95
kroB200	45	7	65	146	7	65	**37**	**7**	**65**
tsp225	**149**	**15**	**102**	376	16	105	155	16	106
a280	**114**	**10**	**59**	249	10	56	132	10	63
lin318	7171	13	177	8201	13	177	**7158**	**13**	**177**
gr431	5973	22	186	19111	22	187	**5925**	**22**	**186**
pcb442	4406	43	215	6186	41	197	**2393**	**43**	**207**
gr666	**33259**	**14**	**216**	189421	14	217	40111	14	216
mean ratio (sec.)	*"s.c. as in (6) / s.c. as in (3)": 0.971694*								
PSS_A_150	**23**	**12**	**61**	65	12	61	26	12	61
PSS_A_200	81	15	84	139	15	84	**76**	**15**	**84**
PSS_A_250	156	14	82	208	14	82	**133**	**14**	**82**
PSS_A_300	**534**	**14**	**123**	4819	14	123	692	14	123
PSS_A_350	**404**	**9**	**110**	789	9	110	650	9	110
PSS_A_400	49234	16	179	247511	16	179	**24619**	**16**	**179**
PSS_A_450	4666	8	117	13806	8	117	**3022**	**8**	**117**
PSS_A_500	68215	12	167	155977	12	167	**30809**	**12**	**167**
mean ratio (sec.)	*"s.c. as in (6) / s.c. as in (3)": 0.928176*								
mean ratio – all	*"s.c. as in (6) / s.c. as in (3)": 0.954287*								

Table 1: Comparison of the behavior of the algorithm for different representations of subtour constraints. "sec." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP before starting the last iteration.

It turned out that the three versions sometimes (but not always) lead to huge differences in running time (up to a factor of 5). This is an interesting experience that should be taken into consideration also in other computational studies. From our limited experiments it could be seen that version (5) was inferior most of the times (with sometimes huge deviations) whereas only a small dominance of the hybrid variant (6) in comparison with the standard version (3) could be observed. This is due to the small size of most subtours occurring during the solution process (the representation (3) equals to the representation (6) in these cases). But since also bigger subtours can occur (mostly

in the last iterations), we use the representation (6) for all further computational tests.

For more details about different ILP-models see [14].

3 Generation of subtours

As pointed out above, the central point of our approach is the generation and selection of a "good" set of subtour constraints, including as many as possible of those required by the ILP-solver to determine an optimal solution which is also feasible for TSP, but as few as possible of all others which only slow down the performance of the ILP-solver.

Trying to strike a balance between these two goals we followed several directions, some of them motivated by theoretical results, others by visually studying plots of all subtours generated during the execution of Algorithm 1.

3.1 Subtours from suboptimal integer solutions

Many ILP-solvers report all feasible integer solutions found during the underlying branch-and-bound process. In this case, we can also add all corresponding subtour constraints to the model. These constraints can be considered simply as part of the set of violated subtour constraints. Not surprisingly, these additional constraints always lead to a decrease in the number of iterations for the overall computation and to an increase in the total number of subtour constraints generated before reaching optimality (see Table 2). While the time consumed in each iteration is likely to increase, it can also be observed that the overall running time is often decreased significantly by adding all detected subtours to the model. On the other hand, for the smaller number of instances, where this is not the case, only relatively modest increases of running times are incurred. Therefore, we stick to adding all detected subtour constraints for the remainder of the paper. The algorithm in this form will be called *BasicIntegerTSP*.

3.2 Subtours of size 3

The next idea we tried was to add subtour inequalities corresponding to some subtours of size 3 into the model before starting the iteration process (i.e. in line 1 of Algorithm 1). This idea was motivated by the observations that in many examples smaller subtours (with respect to their cardinality) occur more often than the larger ones. However, there are $\binom{|V|}{3}$ such subtours and thus we should concentrate only on a relevant subset of them. After studying our computational tests we decided to use the shortest ones with respect to their length. Table 3 summarizes our computational results and it can be seen that this idea actually tends to slow down our approach. Thus we did not follow it any more.

3.3 Subtour selections

As mentioned above, a large number of subtour inequalities which are not really needed only slow down our approach. Thus we also tried not to use all subtour inequalities we

instance	add all subtours sec.	#i.	#c.	only subtours from ILP-optima sec.	#i.	#c.
kroA150	62	12	82	**19**	**7**	**136**
kroB150	**54**	**13**	**77**	179	8	148
u159	9	5	38	**6**	**4**	**49**
brg180	64	16	67	**44**	**4**	**103**
kroA200	2440	11	95	**677**	**8**	**237**
kroB200	37	7	65	**31**	**5**	**121**
tsp225	**155**	**16**	**106**	178	9	261
a280	**132**	**10**	**63**	157	11	143
lin318	7158	13	177	**6885**	**8**	**357**
gr431	5925	22	186	**2239**	**9**	**453**
pcb442	**2393**	**43**	**207**	2737	11	501
gr666	40111	14	216	**17711**	**8**	**789**
mean ratio (sec.)			*0.946130*			
PSS_A_150	26	12	61	**23**	**8**	**100**
PSS_A_200	76	15	84	**72**	**7**	**163**
PSS_A_250	**133**	**14**	**82**	138	9	186
PSS_A_300	**692**	**14**	**123**	866	6	295
PSS_A_350	650	9	110	**411**	**5**	**252**
PSS_A_400	24619	16	179	**8456**	**8**	**454**
PSS_A_450	3022	8	117	**2107**	**5**	**279**
PSS_A_500	30809	12	167	**15330**	**6**	**436**
mean ratio (sec.)			*0.786451*			
mean ratio – all			*0.882259*			

Table 2: Using all constraints generated from all feasible solutions found during the solving process vs. using only the constraints generated from the final ILP solutions of each iteration. "sec." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP before starting the last iteration.

are able to generate during one iteration, but to make a proper selection. We again used our computational tests in order to identify two general properties which seem to point to such "suitable" subtour inequalities.

- Sort all obtained subtours with respect to their cardinality, chose the smallest ones and added the corresponding subtour inequalities into the model.
- Sort all obtained subtours with respect to their length and proceed as above.

The corresponding results are summarized in Tables 4 and 5 and it is obvious that this idea does not speed up our approach as intended. Thus we dropped it from our considerations.

instance	$p=0$			$p=\frac{1}{10000}$			$p=\frac{1}{1000}$		
	t.	**#i.**	**#c.**	**t.**	**#i.**	**#c.**	**t.**	**#i.**	**#c.**
kroA150	**19**	**7**	**136**	**19**	**7**	**97**	40	5	116
kroB150	179	8	148	**71**	**7**	**178**	134	5	105
u159	**6**	**4**	**49**	8	4	46	**6**	**3**	**24**
brg180	44	4	103	**34**	**15**	**108**	82	9	270
kroA200	677	8	237	879	5	157	**504**	**4**	**133**
kroB200	**31**	**5**	**121**	32	5	61	43	5	60
tsp225	178	9	261	**149**	**10**	**224**	167	9	202
a280	157	11	143	**138**	**9**	**98**	156	6	101
lin318	6885	8	357	5360	8	302	**1435**	**8**	**291**
gr431	**2239**	**9**	**453**	3196	10	534	3648	10	571
pcb442	**2737**	**11**	**501**	3483	15	414	3989	14	466
gr666	**17711**	**8**	**789**	–	–	–	–	–	–
mean ratio				*1.002535*			*1.188732*		
PSS_A_150	**23**	**8**	**100**	30	7	130	30	6	77
PSS_A_200	72	7	163	74	8	135	**57**	**6**	**76**
PSS_A_250	**138**	**9**	**186**	155	7	163	140	6	109
PSS_A_300	**866**	**6**	**295**	884	6	203	1344	7	211
PSS_A_350	**411**	**5**	**252**	642	6	147	879	6	150
PSS_A_400	8456	8	454	6623	7	285	**4876**	**8**	**296**
PSS_A_450	2107	5	279	**1226**	**4**	**220**	5386	5	215
PSS_A_500	15330	6	436	13473	6	366	**6114**	**5**	**237**
mean ratio				*1.035264*			*1.291607*		
mean ratio of all				*1.016316*			*1.232048*		

Table 3: Using no subtours of size 3 vs. using the shortest subtours of size 3 for generation of subtour constraints before starting the solving process. The parameter p defines the proportion of used subtour constraints. "t." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP for the start of the last iteration. The entries "–" by TSPLIB instances cannot be computed with 16 GB RAM.

3.4 Clustering into subproblems

It can be observed that many subtours have a local context, meaning that a small subset of vertices separated from the remaining vertices by a reasonably large distance will always be connected by one or more subtours, independently from the size of the remaining graph (see also Figures 3 and 6 to 17 in the Appendix). Thus, we aim to identify *clusters* of vertices and run the algorithm on the induced subgraphs with the aim of generating within a very small running time the same subtours occurring in the execution of the approach on the full graph. Furthermore, we can use the optimal tour from every cluster to generate a corresponding subtour constraint for the original instance

instance	$p = 1$			$p = \frac{2}{3}$			$p = \frac{1}{3}$		
	t.	**#i.**	**#c.**	**t.**	**#i.**	**#c.**	**t.**	**#i.**	**#c.**
kroA150	**19**	**7**	**136**	34	8	109	69	19	115
kroB150	179	8	148	**51**	**8**	**135**	477	15	134
u159	**6**	**4**	**49**	30	4	52	19	11	56
brg180	44	4	103	**27**	**6**	**77**	59	19	80
kroA200	**677**	**8**	**237**	714	7	171	2846	14	131
kroB200	**31**	**5**	**121**	39	6	98	89	13	77
tsp225	178	9	261	**100**	**14**	**183**	173	34	166
a280	157	11	143	**141**	**12**	**154**	239	27	127
lin318	**6885**	**8**	**357**	7069	12	367	9444	32	392
gr431	**2239**	**9**	**453**	3210	20	522	4924	38	413
pcb442	2737	11	501	**1867**	**18**	**384**	5129	85	386
gr666	17711	8	789	**7643**	**7**	**505**	71594	25	597
mean ratio				*1.252892*			*2.488345*		
PSS_A_150	**23**	**8**	**100**	28	9	109	52	17	94
PSS_A_200	72	7	163	**69**	**8**	**134**	112	23	98
PSS_A_250	138	9	186	**131**	**10**	**149**	208	20	119
PSS_A_300	866	6	295	**792**	**10**	**259**	1720	29	293
PSS_A_350	**411**	**5**	**252**	715	7	232	849	19	177
PSS_A_400	**8456**	**8**	**454**	129380	8	311	107987	26	299
PSS_A_450	2107	5	279	**1544**	**7**	**236**	7987	11	238
PSS_A_500	15330	6	436	18594	8	324	**13738**	**16**	**308**
mean ratio				*2.878162*			*3.354102*		
mean ratio of all				*1.903000*			*2.834648*		

Table 4: Using all subtours vs. using only the smallest subtours with respect to their cardinality for generation of subtour constraints. The parameter p defines the proportion of used subtour constraints. "t." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP for the start of the last iteration.

and thus enforce a connection to the remainder of the graph.

For our purposes the clustering algorithm should fulfill the following properties:

- *clustering quality:* The obtained clusters should correspond well with the distance structure of the given graph, as in a classical geographic clustering.
- *running time:* Should be low relative to the running time required for the main part of the algorithm.
- *cluster size:* If clusters are too large, solving the TSP takes too much time. If clusters are too small, only few subtour constraints are generated.

instance	$p=1$			$p=\frac{2}{3}$			$p=\frac{1}{3}$		
	t.	**#i.**	**#c.**	**t.**	**#i.**	**#c.**	**t.**	**#i.**	**#c.**
kroA150	**19**	**7**	**136**	41	10	131	46	16	90
kroB150	**179**	**8**	**148**	495	7	152	250	16	112
u159	**6**	**4**	**49**	14	5	60	23	12	55
brg180	44	4	103	**24**	**13**	**86**	161	8	78
kroA200	**677**	**8**	**237**	862	6	124	1829	13	132
kroB200	**31**	**5**	**121**	59	7	121	79	11	89
tsp225	178	9	261	**112**	**13**	**197**	197	32	159
a280	157	11	143	**94**	**9**	**101**	212	21	96
lin318	**6885**	**8**	**357**	7688	13	355	9593	36	390
gr431	**2239**	**9**	**453**	6091	15	565	9434	45	530
pcb442	2737	11	501	**2365**	**18**	**487**	5913	70	399
gr666	17711	8	789	**14713**	**10**	**735**	–	–	–
mean ratio				*1.478194*			*2.434945*		
PSS_A_150	**23**	**8**	**100**	24	9	115	45	22	81
PSS_A_200	72	7	163	**60**	**10**	**123**	113	25	108
PSS_A_250	**138**	**9**	**186**	**138**	**7**	**117**	209	22	103
PSS_A_300	**866**	**6**	**295**	1099	10	321	953	23	201
PSS_A_350	**411**	**5**	**252**	876	7	231	934	16	167
PSS_A_400	**8456**	**8**	**454**	29625	9	311	301125	27	378
PSS_A_450	**2107**	**5**	**279**	2926	7	259	4789	14	237
PSS_A_500	**15330**	**6**	**436**	15786	7	329	37460	16	330
mean ratio				*1.524891*			*6.092589*		
mean ratio of all				*1.496873*			*3.975006*		

Table 5: Using all subtours vs. using only the smallest subtours with respect to their length for generation of subtour constraints. The parameter p defines the proportion of used subtour constraints. "t." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP for the start of the last iteration. The entries "–" by TSPLIB instances cannot be computed with 16 GB RAM.

Clearly, there is a huge body of literature on clustering algorithms (see e.g. [10]) and selecting one for a given application will never satisfy all our objectives. Our main restriction was the requirement of using a clustering algorithm which works also if the vertices are not embeddable in Euclidean space, i.e. only arbitrary edge distances are given. Simplicity being another goal, we settled for the following approach described in Algorithm 2:

First, we fix the number of clusters c with $1 \leq c \leq n$ and sort the edges in increasing order of distances (see line 1). Then we start with the empty graph $G' = (V', E')$ (line 2) containing only isolated vertices (i.e. n clusters) and add iteratively edges in increasing order of distances until the desired number of clusters c is reached (see lines 5 and 6).

Input: Complete graph $G = (V, E)$, where $|V| = n$ and $|E| = m = \frac{n(n-1)}{2}$, distance function $d\colon E \to R_0^+$ and parameter c, where $1 \leq c \leq n$.
Output: Clustering $\mathcal{C} = \{V_1, \ldots, V_c\}$, where $V_1 \cup \ldots \cup V_c = C$.
1: sort the edges such that $d_{e_1} \leq \ldots \leq d_{e_m}$;
2: define $G' = (V', E')$ such that $V' = V$ and $E = \emptyset$;
3: let $i := 1$;
4: define $\mathcal{C} := \{v_1, \ldots, v_n\}$;
5: **while** $|\mathcal{C}| > c$ **do**
6: set $E' := E' \cup \{e_i\}$;
7: set $\mathcal{C} := \{V_1, \ldots, V_{|\mathcal{C}|}\}$, where $V_1, \ldots, V_{|\mathcal{C}|}$ are the connected components of graph G';
8: **end while**

Algorithm 2: Clustering algorithm.

In each iteration the current clustering is implied by the connected components of the current graph (see line 7).

Note that this clustering algorithm does not make any assumptions about the underlying TSP instance and does not exploit any structural properties e.g. of the *Metric TSP* or the *Euclidean TSP*.

It was observed in our computational experiments that the performance of the TSP algorithm is not very sensitive to smaller changes of the cluster number c and thus a rough estimation of c is sufficient. The behavior of the running time as a function of c can be found for particular test instances in Figure 19, see Section 4.2 for further discussion.

3.5 Restricted clustering into subproblems

Although the clustering algorithm (see Algorithm 2) decreases the computational time of the whole solution process for some test instances, we observed a certain shortcoming. There may easily occur clusters consisting of isolated points or containing only two vertices. Clearly, these clusters do not contribute any subtour on their own. Moreover, the degree constraints (2) guarantee that each such vertex is connected to the remainder of the graph in any case. The connection of these vertices to some "neighboring" cluster enforced in BasicIntegerTSP implies that the clustering yields different subtours for these neighbors and not the violated subtour constraints arising in BasicIntegerTSP.

To avoid this situation, we want to impose a minimum cluster size of 3. An easy way to do so works as follows: After reaching the c clusters, continue to add edges in increasing order of distances (as before), but only add an edge, if it is incident to one of the vertices in a connected component (i.e. cluster) of size one or two. This means basically that we simply merge these small clusters to their nearest neighbor with respect to the actual stage of this restricting algorithm. (Note that two clusters of size 1 can merge first and then the resulting cluster always merges to the nearest neighbor with respect to the actual clustering.) The resulting *restricted clustering approach* will be

denoted by $RC \mid c$.

Against our expectations, the computational experiments (see Section 4) show that this approach often impacts the algorithm in the opposite way (see also Figure 19 and Table 9 in the Appendix) if compared for the same original cluster size c.

Surprisingly, we could observe an interesting behavior if $c \approx n$. In this case, the main clustering algorithm (see Algorithm 2) has almost no effect, but the "post-phase" which enforces the minimum cluster size yields a different clustering on its own. This variant often beats the previous standard clustering algorithm with $c \ll n$ (see Table 9 in the Appendix). Note that we cannot determine the real number of clusters c' in this case. But our computational results show that $c' \approx \frac{n}{3}$ usually holds if the points are distributed relatively uniformly in the Euclidean plane and if the distances correspond to their relative Euclidean distances (see Figure 18 in the Appendix).

3.6 Hierarchical clustering

It was pointed out in Section 3.4 that the number of clusters c is chosen as an input parameter. The computational experiments in Subsection 4.2 give some indication on the behavior of Algorithm 2 for different values of c, but fail to provide a clear guideline for the selection of c. Moreover, from graphical inspection of test instances, we got the impression that a larger number of relevant subtour constraints might be obtained by considering more clusters of moderate size. In the following we present an idea that takes both of these aspects into account.

In our *hierarchical clustering* process denoted by HC we do not set a cluster number c, but let the clustering algorithm continue until all vertices are connected (basically, we set $c = 1$). The resulting clustering process can be represented by a binary *clustering tree* which is constructed in a bottom-up way. The leaves of the tree correspond to isolated vertices, i.e. the n trivial clusters given at the beginning of the clustering algorithm. Whenever two clusters are merged by the addition of an edge, the two corresponding tree vertices are connected to a new common parent vertex in the tree representing the new cluster. At the end of this process we reach the root of the clustering tree corresponding to the complete vertex set. An example of such a clustering tree is shown in Figures 1 and 2.

Figure 1: Example illustrating the hierarchical clustering: Vertices of the TSP instance. Distances between every two vertices correspond to their respective Euclidean distances in this example.

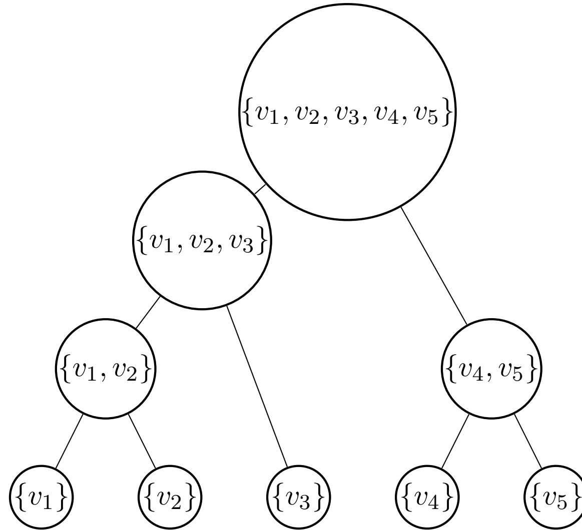

Figure 2: Example illustrating the hierarchical clustering: Clustering tree.

Now, we go through the tree in a bottom-up fashion from the leaves to the root. In each tree vertex we solve the TSP for the associated cluster, after both of its child vertices were resolved. The crucial aspect of our procedure is the following: All subtour constraints generated during such a TSP solution for a certain cluster are propagated and added to the ILP model used for solving the TSP instance of its parent cluster. Obviously, at the root vertex the full TSP is solved.

The advantage of this strategy is the step-by-step construction of the violated subtour constraints. A disadvantage is that many constraints can make sense in the local context but not in the global one and thus too many constraints could be generated in this way. Naturally, one pays for the additional subtour constraints by an increase in computation time required to solve a large number of – mostly small – TSP instances. To avoid the solution of TSPs of the same order of magnitude as the original instance, it makes sense to impose an upper bound u on the maximum cluster size. This means that the clustering tree is partitioned into several subtrees by removing all tree vertices corresponding to clusters of size greater than u. After resolving all these subtrees we collect all generated subtour constraints and add them to the ILP model for the originally given TSP. Computational experiments with various choices of u indicated that $u = 4\frac{n}{\log_2 n}$ would be a good upper bound.

Let us take a closer look at the problem of including too many subtour constraints which are redundant in the global graph context. Of course the theoretical “best” way would be to check which of the propagated subtour constraints were not used during the runs of the ILP solver and drop them. To do this, it would be necessary to get this information from the ILP solver which often is not possible.

However, we can try to approximately identify subtours which are not only locally relevant in the following way: All subtour constraints generated in a certain tree vertex,

i.e. for a certain cluster, are marked as *considered subtour constraints.* Then we solve the TSP for the cluster of its parent vertex in the tree without using the subtours marked as *considered.* If we generate such a considered subtour again during the solution of the parent vertex, we take this as an indicator of global significance and add the constraint permanently for all following supersets of this cluster. If we set the upper bound u, we take also all subtour constraints found in the biggest solved clusters. This approach will be denoted as $HCD \mid u = 4n/\log_2 n$.

Of course, it is only a heuristic rule and one can easily find examples, where this prediction on a subtour's relevance fails, but our experiments indicate that $HCD \mid u = 4n/\log_2 n$ is the best approach we considered. A comparison with other hierarchical clustering methods for all test instances can be found in Table 8 in the Appendix. It can be seen that without an upper bound we are often not able to find the solution at all (under time and memory constraints we made on the computational experiments). In the third and fourth column we can see a comparison between approaches both using the upper bound $u = 4\frac{n}{\log_2 n}$ where the former collects all detected subtour constraints and the latter allows to drop those which seem to be relevant only in a local context. Both these methods beat BasicIntegerTSP(for the comparison of this approach with other presented algorithms see the computational experiments in Section 4).

4 Computational experiments

In the following the computational experiments and their results will be discussed.

4.1 Setup of the computational experiments

All tests were run on an *Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz with 16 GB RAM* under *Linux*[*] and all programs were implemented in *C++*[†] by using the *SCIP* MIP-solver [1] together with *CPLEX* as LP-solver[‡]. It was often discussed in the literature (see e.g. [13]) and in personal communications that ILP-solvers are relatively unrobust and often show high variations in their running time performance, even if the same instance is repeatedly run on the same hardware and same software environment. Our first test runs also exhibited deviations up to a factor of 2 when identical tests were repeated. Thus we took special care to guarantee the relative reproducibility of the computational experiments: No additional swap memory was made available during the tests, only one thread was used and no other parallel user processes were allowed. This lead to a high degree of reproducibility in our experiments. However, this issue makes a comparison to other simple approaches, which were tested on other computers under other hardware and software conditions, extremely difficult.

[*] precise version: *Linux 3.8.0-29-generic #42~precise1-Ubuntu SMP x86_64 x86_64 x86_64 GNU/Linux*

[†] precise compiler version: *gcc version 4.6.3*

[‡] precise version: *SCIP version 3.0.1 [precision: 8 byte] [memory: block] [mode: optimized] [LP solver: CPLEX 12.4.0.0] [GitHash: 9ee94b7] Copyright (c) 2002-2013 Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)*

We used two groups of test instances: The first group is taken from the well-known TSPLIB95 [17], which contains the established benchmarks for TSP and related problems. From the collection of instances we chose all those with (i) at least 150 and at most 1000 vertices and (ii) which could be solved in at most 12 hours by our BasicIntegerTSP. It turned out that 25 instances of the TSPLIB95 fall into this category (see Table 9), the largest having 783 vertices.

We also observed some drawbacks of these instances: Most of them (23 of 25) are defined as point sets in the Euclidean plane with distances corresponding to the Euclidean metric or as a set of geographical cities, i.e. points on a sphere. Moreover, they often contain substructures like meshes or sets of colinear points and finally, since all distances are rounded to the nearest integer, there are many instances which have multiple optimal solutions. These instances are relatively unstable with respect to solution time, number of iterations, and – important for our approach – cardinality of the set of violated subtour constraints. For our approach instances with a mesh geometry (e.g. *ts225* taken from the TSPLIB95) were especially prone to unstable behavior, such as widely varying running times for minor changes in the parameter setting. This seems to be due to the fact that these instances contain many 2-matchings with the same objective function value. In particular, the number of optimal 2-matchings is larger by an exponential factor than the number of optimal TSP tours with the same objective value for $n \to \infty$.

In order to provide further comparisons, we also defined a set of instances based on *random Euclidean graphs*: In a unit square $[0, 1]^2$ we chose n uniformly distributed points and defined the distance between every two vertices as their respective Euclidean distance[§]. These *random Euclidean instances* eliminate the potential influence of substructures and always have only one unique optimal solution in all stages of the solving process. We created 40 such instances named PSS_X_n where $n \in \{150, 200, 250, \ldots, 500\}$ indicates the number of vertices and $X \in \{A, B, C, D, E\}$.

The running times of our test instances, most of them containing between 100 and 500 vertices, were often within several hours. Since we tested many different variants and configurations of our approach, we selected a subset of these test instances to get faster answers for determining the best algorithm settings for use in the final tests. This subset contains 12 (of the 25) TSPLIB instances and one random instances for every number of vertices n (see e.g. Table 1.)

All our running time tables report the name of the instance, the running time (**sec.**) in wall-clock seconds (rounded down to integers), the number of iterations (**#i.**), i.e. the number of calls to the ILP-solver in the main part of our algorithm (without the TSP solutions for the clusters) and the number of subtour constraints (**#c.**) added to the ILP model in the last iteration, i.e. the number of constraints of the model which yielded an optimal TSP solution. We often compare two columns of a table by taking the **mean ratio**, i.e. computing the quotient between the running times on the same instance and taking the arithmetic mean of these quotients.

[§]We represented all distances as integers by scaling with 2^{14} and rounding to the nearest integer.

4.2 Computational details for concrete examples

Let us now take a closer look at two instances in detail. While this serves only as an illustration, we studied lots of these special case scenarios visually during the development of the clustering approach to gain a better insight into the structure of subtours generated by BasicIntegerTSP.

We selected instances *kroB150* and *u159* whose vertices are depicted in Figures 4 and 5 in the Appendix. Both instances consist of points in the Euclidean plane and the distances between every two vertices correspond to their respective Euclidean distances.

First, Figure 19 in the Appendix illustrates the behavior of the running time t in seconds as a function of the parameter c for the instances *kroB150* and *u159*. The full lines correspond to standard clustering approach described in Section 3.4 (see Algorithm 2), while the dashed line corresponds to the restricted clustering of Section 3.5 with minimum cluster size 3. The standard BasicIntegerTSP without clustering arises for $c = 1$.

Instance *kroB150* consists of relatively uniformly distributed points in the Euclidean plane, but in fact, it has a specific property: By using Algorithm 2 we can observe the occurrence of two main components also for relatively small coefficient c (already for $c = 6$). This behavior is rather atypical for random Euclidean graphs, cf. [16, ch. 13], but it provides an advantage for our approach since we do not have to solve cluster instances of the same order of magnitude as the original graph but have several clusters of moderate size also for small cluster numbers c.

Considering the standard clustering approach (Algorithm 2) in Figure 19, upper graph, it can be seen that only a small improvement occurs for c between 2 and 5. Looking at the corresponding clusterings in detail, it turns out in theses cases that there exists only one "giant connected component" and all other clusters have size 1. This structure also implies that for the restricted clustering, these isolated vertices are merged with the giant component and the effect of clustering is lost completely.

For larger cluster numbers c, a considerable speedup is obtained, with some variation, but more or less in the same range for almost all values of $c \geq 6$ (in fact, the giant component splits in these cases). Moreover, the restricted clustering performs roughly as good as the standard clustering for $c \geq 6$.

Instance *u159* is much more structured and has many colinear vertices. Here, we can observe a different behavior. While the standard clustering is actually beaten by BasicIntegerTSP for smaller cluster numbers and has a more or less similar performance for larger cluster numbers, the restricted clustering is almost consistently better than the other two approaches.

This is due to the following facts: We can observe that there exists a large component which consumes as much computation time as the whole instance for c between 2 and 10. Moreover, most of the time is needed for the last iteration which then has to be solved again for the complete graph.

These two instances give some indication how to characterize the "good" instances for our algorithm: They should

- consist of more clearly separated clusters and

- they should not contain mesh substructures and colinear vertices (as mentioned above, we have to include many subtour constraints in these cases).

4.3 General computational results

A summary of the computational results for BasicIntegerTSP and the most promising variants of clustering based subtour generations can be found in Table 9. For random Euclidean instances we report only the mean values of all five instances of the same size. It turns out that $HCD \mid u = 4\frac{n}{\log_2 n}$, i.e. the hierarchical clustering approach combined with dropping subtour constraints and fixing them only if they are generated again in the subsequent iteration and with the upper bound on the maximum cluster size $u = 4\frac{n}{\log_2 n}$, gives the best overall performance. A different behavior can be observed for instances taken from the TSPLIB and for random Euclidean instances. On the TSPLIB instances this algorithm $HCD \mid u = 4\frac{n}{\log_2 n}$ is in average about 20% faster than pure BasicIntegerTSP and beats the other clustering based approaches for most instances. In those cases, where it is not the best choice, it is usually not far behind.

As already mentioned, we could observe that the best setting for $HCD \mid u = 4\frac{n}{\log_2 n}$ are instances with a strong cluster structure and without mesh substructures (e.g. *pr299*). On the opposite side are instances with mesh substructures where it is difficult to find an optimal 2-matching which is also a TSP tour (see also Subsection 4.1).

For random Euclidean instances, the results are less clear. But approaches with fixed number of clusters seem to be better then the hierarchical ones.

It was a main goal of this study to find a large number of "good" subtour constraints, i.e. subtours that are present in the last iteration of the ILP-model of BasicIntegerTSP. Therefore, we show the potentials and limitations of our approach in reaching this goal. In particular, we will report the relation between the set S_1 consisting of all subtours generated by running a hierarchical clustering algorithm with an upper bound u (set as in the computational tests to $u = 4\frac{n}{\log_2 n}$) before starting the main part (i.e. before solving the root vertex) and the set S_2 containing only the subtour constraints included in the final ILP model of BasicIntegerTSP. We tested the hierarchical clustering with and without the dropping of non-repeated subtours.

There are two aspects we want to describe: At first, we want to check whether S_1 contains a relevant proportion of "useful" subtour contraints, i.e. constraints also included in S_2, or whether S_1 contains "mostly useless" subtours. Therefore, we report the *proportion of used subtours* defined as

$$p_u := \frac{|S_1 \cap S_2|}{|S_1|}. \tag{7}$$

Secondly, we want to find out to what extend it is possible to find the "right" subtours by our approach. Hence, we define the *proportion of covered subtours* defined as

$$p_c := \frac{|S_1 \cap S_2|}{|S_2|}. \tag{8}$$

The values of p_u and p_c are given in Table 6.

instance	$HC \mid u = 4\frac{n}{\log_2 n}$		$HCD \mid u = 4\frac{n}{\log_2 n}$	
	p_u	p_c	p_u	p_c
kroA150	0.262712	0.455882	0.476190	0.367647
kroB150	0.222222	0.351351	0.396040	0.270270
u159	0.085271	0.448980	0.153226	0.387755
brg180	0.133929	0.145631	0.714286	0.145631
kroA200	0.209713	0.324895	0.450704	0.270042
kroB200	0.206612	0.413223	0.423423	0.388430
tsp225	0.134752	0.218391	0.297143	0.199234
a280	0.064935	0.314685	0.161943	0.279720
lin318	0.234589	0.383754	0.440273	0.361345
gr431	0.073701	0.209713	0.221053	0.185430
pcb442	0.056759	0.151697	0.133117	0.163673
gr666	0.076048	0.271229	0.220379	0.235741
mean	*0.146770*	*0.307453*	*0.340648*	*0.271243*
PSS_A_150	0.179191	0.310000	0.289157	0.240000
PSS_A_200	0.122642	0.239264	0.212329	0.190184
PSS_A_250	0.120773	0.268817	0.172727	0.204301
PSS_A_300	0.191235	0.325424	0.331915	0.264407
PSS_A_350	0.151274	0.376984	0.285714	0.333333
PSS_A_400	0.170455	0.297357	0.254157	0.235683
PSS_A_450	0.148148	0.415771	0.311178	0.369176
PSS_A_500	0.165485	0.321101	0.276596	0.268349
mean	*0.156150*	*0.319340*	*0.266722*	*0.263179*
mean of all	*0.150522*	*0.312207*	*0.311078*	*0.268018*

Table 6: Proportion of used and proportion of covered subtours for our hierarchical clustering approaches with the upper bound $u = 4\frac{n}{\log_2 n}$ which (i) does not allow ($HC \mid u = 4\frac{n}{\log_2 n}$) and which (ii) does allow ($HCD \mid u = 4\frac{n}{\log_2 n}$) to drop the unused subtour constraints.

It can be seen that empirically there is the chance to find about 26 – 31 % (p_c) of all required violated subtour constraints. If subtour constraints are allowed to be dropped, we are able to find fewer such constraints, but our choice has a better quality (p_c is smaller, but p_u is larger), i.e. the solver does not have to work with a large number of constraints which only slow down the solving process and are not necessary to reach an optimal solution.

Furthermore we can observe a relative big difference between the values of the proportion of used subtour constraints (p_u) for the TSPLIB instances and for random Euclidcean instances if the dropping of redundant constraints is allowed.

4.4 Adding a starting heuristic

Of course, there are many possibilities of adding improvements to our basic approach. Lower bounds and heuristics can be introduced, branching rules can be specified, or cutting planes can be generated. We did not pursue these possibilities since we want to focus on the simplicity of the approach. Moreover, we wanted to take the ILP solver as a "black box" and not interfere with its execution.

Just as an example which immediately comes to mind, we added a starting heuristic to give a reasonably good TSP solution as a starting solution to the ILP solver. We used the improved version of the classical Lin-Kernighan heuristic in the code written by Helsgaun [9]. The computational results reported in Table 7 show that a considerable speedup (roughly a factor of 3, but also much more) can be obtained in this way.

instance	**without**			**with starting heuristic**		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
kroA150	19	7	136	**16**	**10**	**34**
kroB150	179	8	148	**17**	**8**	**104**
u159	6	4	49	**4**	**5**	**40**
brg180	44	4	103	**0**	**2**	**15**
kroA200	677	8	237	**42**	**8**	**135**
kroB200	31	5	121	**28**	**6**	**124**
tsp225	178	9	261	**73**	**13**	**176**
a280	157	11	143	**32**	**8**	**58**
lin318	6885	8	357	**4941**	**8**	**259**
gr431	2239	9	453	**838**	**10**	**318**
pcb442	2737	11	501	**447**	**18**	**207**
gr666	17711	8	789	**13225**	**11**	**485**
mean ratio (sec.)	*0.432074*					
PSS_A_150	23	8	100	**14**	**11**	**65**
PSS_A_200	72	7	163	**38**	**11**	**99**
PSS_A_250	138	9	186	**63**	**9**	**124**
PSS_A_300	866	6	295	**146**	**8**	**173**
PSS_A_350	411	5	252	**126**	**6**	**151**
PSS_A_400	8456	8	454	**1274**	**6**	**251**
PSS_A_450	2107	5	279	**482**	**7**	**197**
PSS_A_500	15330	6	436	**1997**	**9**	**241**
mean ratio (sec.)	*0.322231*					
mean ratio – all	*0.388137*					

Table 7: Results for BasicIntegerTSP used without / with the Lin-Kernighan heuristic for generating an initial solution. "sec." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP before starting the last iteration.

5 Some theoretical results and further empirical observations

Although our work mainly aims at computational experiments, we also tried to analyze BasicIntegerTSP theoretically. In particular: What is the typical behavior for random Euclidean instances? And what can we say about the expected cardinality of the minimal set of subtours $\mathcal{S}^*$ defined in Section 2? In order to find some empirical evidence to these questions, we performed extensive computational tests, some of them presented in Figures 20 and 21 in the Appendix.

First, the upper graph in Figure 20 illustrates the mean number of iterations needed by BasicIntegerTSP to reach optimality for different numbers of vertices n (we evaluated 100 random Euclidean instances for every value n). The lower graph of the same Figure 20 shows the mean length of the optimal TSP tour and of the optimal 2-matching (i.e. the objective value after solving the ILP in the first iteration) by using the same setting. The expected length of an optimal TSP tour was proven to be asymptotically $\beta\sqrt{n}$, where β is a constant [3]. This approach was later generalized for other settings and other properties of the square root asymptotic were identified [19, 22]. We used these properties to prove the square root asymptotic also for the 2-matching problem (see Figure 20, lower graph, dashed). In particular, we proved the following theorem (for the proof see [21]).

Definition 1 (complete convergence)**.** *A sequence of random variables X_n, $n \geq 1$,* converges completely (c.c.) *to a constant C if and only if for all $\varepsilon > 0$ we have*

$$\sum_{n=1}^{\infty} \mathbb{P}\left[|X_n - C| > \varepsilon\right] < \infty \ . \tag{9}$$

Theorem 1. *Let $G = (V, E)$ be a random Euclidean graph with $n = |V|$ vertices and let $d\colon E \to \mathbb{R}_0^+$ be the distance function defined as the Euclidean distance between the vertices u and v for every edge $e = (u, v)$. Furthermore, let $M_2(G, d)$ be the length of an optimal 2-matching. Then*

$$\lim_{n\to\infty} \frac{M_2(G,d)}{\sqrt{n}} = \alpha \ c.c. \tag{10}$$

where $\alpha > 0$.

Proof. See [21]. □

Based on these results the following idea might lead to a proof that the expected cardinality $\mathcal{S}^*$ is polynomially bounded: After the first iteration of the algorithm we have a solution possibly consisting of several separate subtours of total asymptotic length $\alpha\sqrt{n} = \alpha_1\sqrt{n}$. If there are subtours, we add subtour constraints (in fact at most $\lfloor \frac{n}{3} \rfloor$), resolve the enlarged ILP and get an other solution whose asymptotic length is $\alpha_2\sqrt{n}$. By proving that the expected length of the sequence $\alpha = \alpha_1, \ldots, \alpha_{\#i} = \beta$ is polynomially bounded in n, one would obtain that also $\mathbb{E}\left[|S^*|\right]$ is also polynomially bounded since

only polynomially many subtours are added in each iteration. Our intuition and computational tests illustrated in Figure 20, upper graph, indicate that the length of this sequence could be $\gamma\sqrt{n}$ as well. Unfortunately, we could not find the suitable techniques to show this step.

Another approach is illustrated by Figure 21, where we examine the mean number of subtours contained in every iteration. In particular, we chose $n = 60$, generated 100000 random Euclidean instances and sorted them by the number of iterations #i. required by BasicIntegerTSP. The most frequent number of ILP solver runs was 7 (dotted line), but we summarize the results for 5 (full line), 6 (dashed), 8 (loosely dashed) and 9 (loosely dotted) necessary runs in this figure as well. For every iteration of every class (with respect to the number of involved ILP runs) we computed the mean number of subtours contained in the respective solutions. As can be expected these number of subtours are decreasing (in average) over the number of iterations. To allow a better comparison of this behavior for different numbers of iterations we scaled the iteration numbers into the interval $[0, 10]$ (horizontal axis of Figure 21). It can be seen that the average number of subtours contained in an optimal 2-matching (first iteration) is about 9.2 while in the last iteration we obviously have only one tour. Between these endpoints we can first observe a mostly convex behavior, only in the last step before reaching the optimal TSP tour a sudden drop occurs. It would be interesting to derive an asymptotic description of these curves. A intuitive guess would point to an exponential function, but so far we could not find a theoretical justification of this claim.

It is well known that no polynomially bounded representation of the TSP polytope can be found and there also exist instances based on a mesh-structure for which $\mathbb{E}[|S^*|]$ has exponential size (see [21] for details), but the question for the expected size of $|S^*|$ for random Euclidean instances and thus for the expected number of iterations of our solution algorithm remains an interesting open problem.

6 Conclusions

In this study we provide a "test of concept" of a very simple approach to solve TSP instances of medium size to optimality by exploiting the power of current ILP solvers. The approach consists of iteratively solving ILP models with relaxed subtour constraints to integer optimality. Then it is easy to find integral subtours and add the corresponding constraints to the ILP model. Iterating this process until no more subtours are contained in the solution obviously solves the TSP to optimality.

In this work we focus on the structure of subtour constraints and how to find a "good" set of subtour constraints in reasonable time. Therefore, we aim to identify the local structure of the vertices of a given TSP instance by running a clustering algorithm. Based on empirical observations and results from random graph theory we further extend this clustering-based approach and develop a hierarchical clustering method with a mechanism to identify subtour constraints as "relevant", if they appear in consecutive iterations of the algorithm.

We mostly refrained from adding additional features which are highly likely to im-

prove the performance considerably, such as starting heuristics (cf. Section 4.4), lower bounds or adding additional cuts. It might be interesting to explore in the future how far one can reach with a purely integer linear programming approach by adding these improvements.

Clearly, we can not expect such a basic approach to compete with the performance of *Concorde* [2], which has been developed over many years and basically includes all theoretical and technical developments known so far. However, it turns out that most of the standard benchmark instances with up to 500 vertices can be solved in a few minutes by this purely integer strategy.

Finally, we briefly discussed some theoretical aspect which could lead to polyhedral results in the expected case.

Acknowledgements

The research was funded by the Austrian Science Fund (FWF): P23829-N13.

At this place we would like to thank the developers of the SCIP MIP-solver from the Konrad-Zuse-Zentrum für Informationstechnik Berlin, especially Mr Gerald Gamrath, for their valuable support.

References

[1] T. Achterberg, "Scip: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, Jul. 2009, http://mpc.zib.de/index.php/MPC/article/view/4.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2006.

[3] J. Beardwood, J. H. Halton, and J. M. Hammersley, "The shortest path through many points," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 55, pp. 299–327, 1959.

[4] R. Bosch, "Connecting the dots: The ins and outs of tsp art," in *Bridges Leeuwarden: Mathematics, Music, Art, Architecture, Culture.* Winfield, Kansas: Southwestern College, 2008, pp. 235–242, available online at http://archive.bridgesmathart.org/2008/bridges2008-235.html.

[5] H. Crowder and M. W. Padberg, "Solving large-scale symmetric travelling salesman problems to optimality," *Management Science*, vol. 26, no. 5, pp. 495–509, 1980.

[6] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Operations Research*, vol. 2, pp. 393–410, 1954.

[7] M. Grötschel and O. Holland, "Solving large-scale symmetric travelling salesman problems to optimality," *Mathematical Programming*, vol. 51, pp. 141–202, 1991.

[8] G. Gutin and A. Punnen, *The Traveling Salesman Problem and Its Variations.* Springer, 2006.

[9] K. Helsgaun, "LKH – version 2.0.2," Website, 2008, available at www.akira.ruc.dk/~keld/research/LKH/LKH-2.0.2.tgz.

[10] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data.* Prentice Hall PTR, 1988.

[11] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* J. Wiley, 1985.

[12] P. Miliotis, "Integer programming approaches to the travelling salesman problem," *Mathematical Programming*, vol. 10, pp. 367–378, 1976.

[13] D. Naddef and S. Thienel, "Efficient separation routines for the symmetric traveling salesman problem ii: separating multi handle inequalities," *Mathematical Programming*, vol. 92, pp. 257–283, 2002.

[14] T. Öncan, İ. Kuban Altınel, and G. Laporte, "A comparative analysis of several asymmetric traveling salesman problem formulations," *Computers & Operations Research*, vol. 36, no. 3, pp. 637–654, 2009.

[15] M. Padberg and G. Rinaldi, "An efficient algorithm for the minimum capacity cut problem," *Mathematical Programming*, vol. 47, pp. 19–36, 1990.

[16] M. Penrose, *Random Geometric Graphs.* Oxford University Press, 2003.

[17] G. Reinelt, "TSPLIB95," Website, 1995, available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[18] G. Reinelt, *The Traveling Salesman: Computational Solutions for TSP Applications.* Springer, 1994.

[19] W. T. Rhee, "A matching problem and subadditive euclidean functionals," *The Annals of Applied Probability*, vol. 3, no. 3, pp. 794–801, 1993.

[20] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency.* Springer, 2003.

[21] R. Staněk, "Graph problems with knapsack constraints," 2014, Graz University of Technology, PhD thesis under preparation.

[22] J. E. Yukich, *Probability Theory of Classical Euclidean Optimization Problems.* Springer, 1998.

Appendix

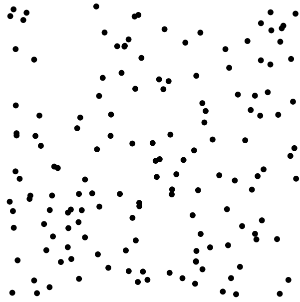

Figure 3: Instance *PSS_A_150*. Euclidean distances between vertices.

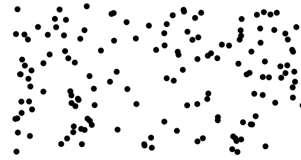

Figure 4: Instance *kroB150*. Euclidean distances between vertices.

Figure 5: Instance *u159*. Euclidean distances between vertices.

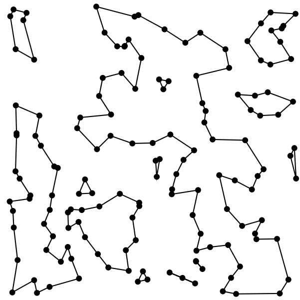

Figure 6: Instance *PSS_A_150*: Main idea of our approach – iteration 0.

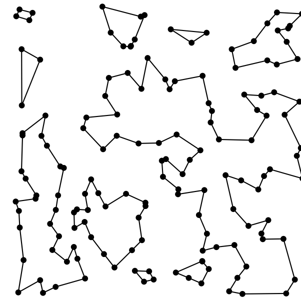

Figure 7: Instance *PSS_A_150*: iteration 1.

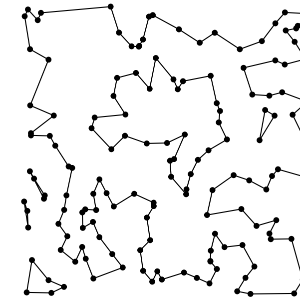

Figure 8: Instance *PSS_A_150*: iteration 2.

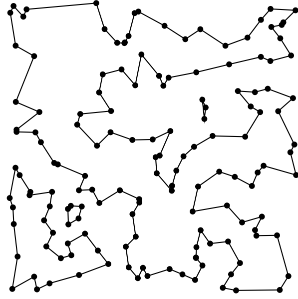

Figure 9: Instance *PSS_A_150*: iteration 3.

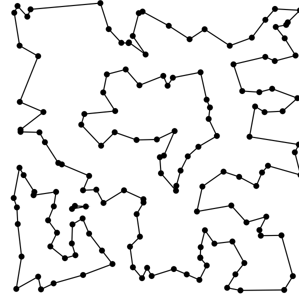

Figure 10: Instance *PSS_A_150*: iteration 4.

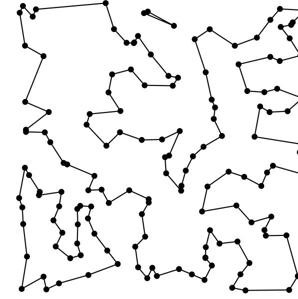

Figure 11: Instance *PSS_A_150*: iteration 5.

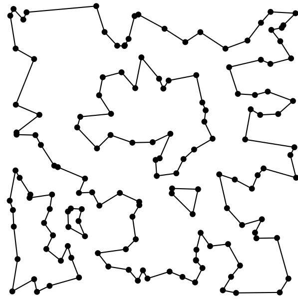

Figure 12: Instance *PSS_A_150*: iteration 6.

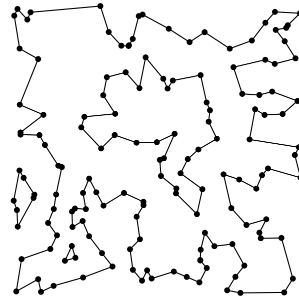

Figure 13: Instance *PSS_A_150*: iteration 7.

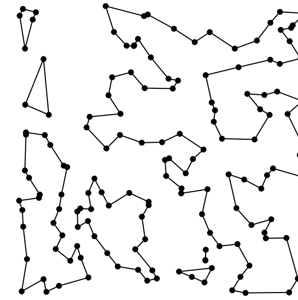

Figure 14: Instance *PSS_A_150*: iteration 8.

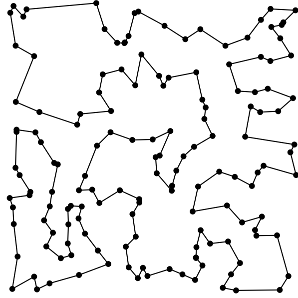

Figure 15: Instance *PSS_A_150*: iteration 9.

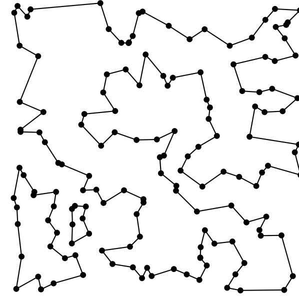

Figure 16: Instance *PSS_A_150*: iteration 10.

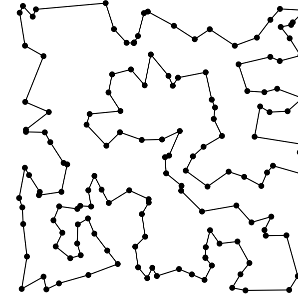

Figure 17: Instance *PSS_A_150*: iteration 11.

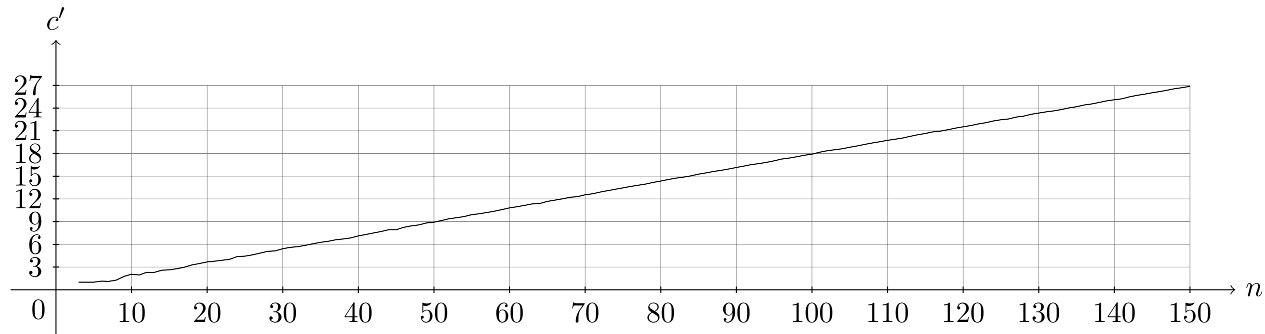

Figure 18: Restricted clustering with $c = n$ on random Euclidean graphs with minimum cluster size 3. The number of obtained clusters c' is plotted for every n. For every number of vertices n we created 100000 graphs.

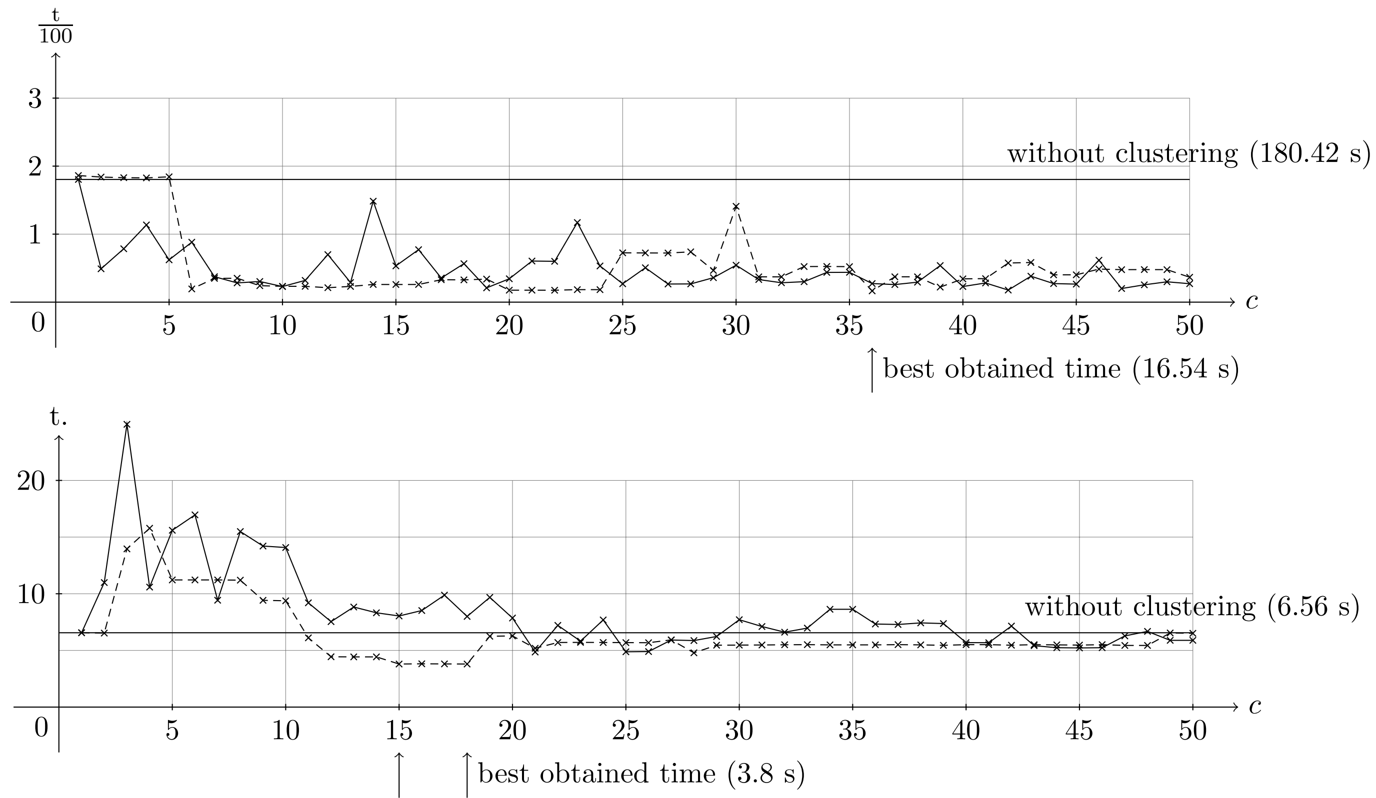

Figure 19: Computation time t in seconds depending on the number of clusters c for clustering (full line) and for restricted clustering (dashed). Illustrative instances *kroB150* (upper figure) and *u159* (lower figure).

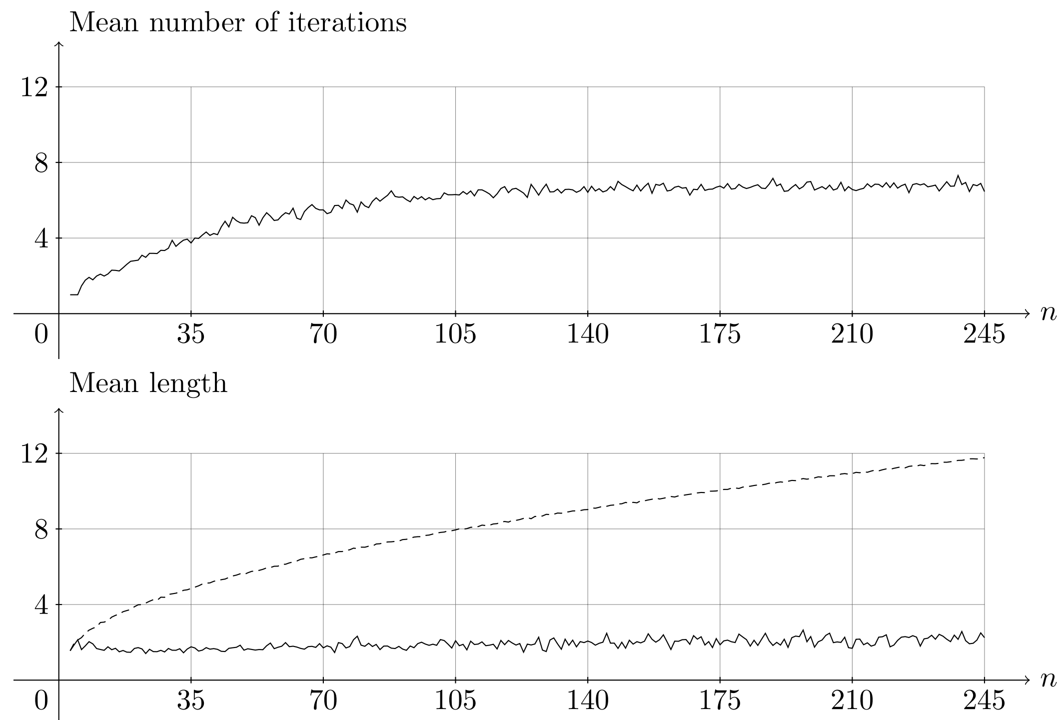

Figure 20: Mean number of iterations used by the BasicIntegerTSP (upper figure), mean length of an optimal TSP tour (lower figure, dashed) and mean length of an optimal weighted 2-matching (lower figure, full line) in random Euclidean graphs. For every number of vertices n we created 100 graphs.

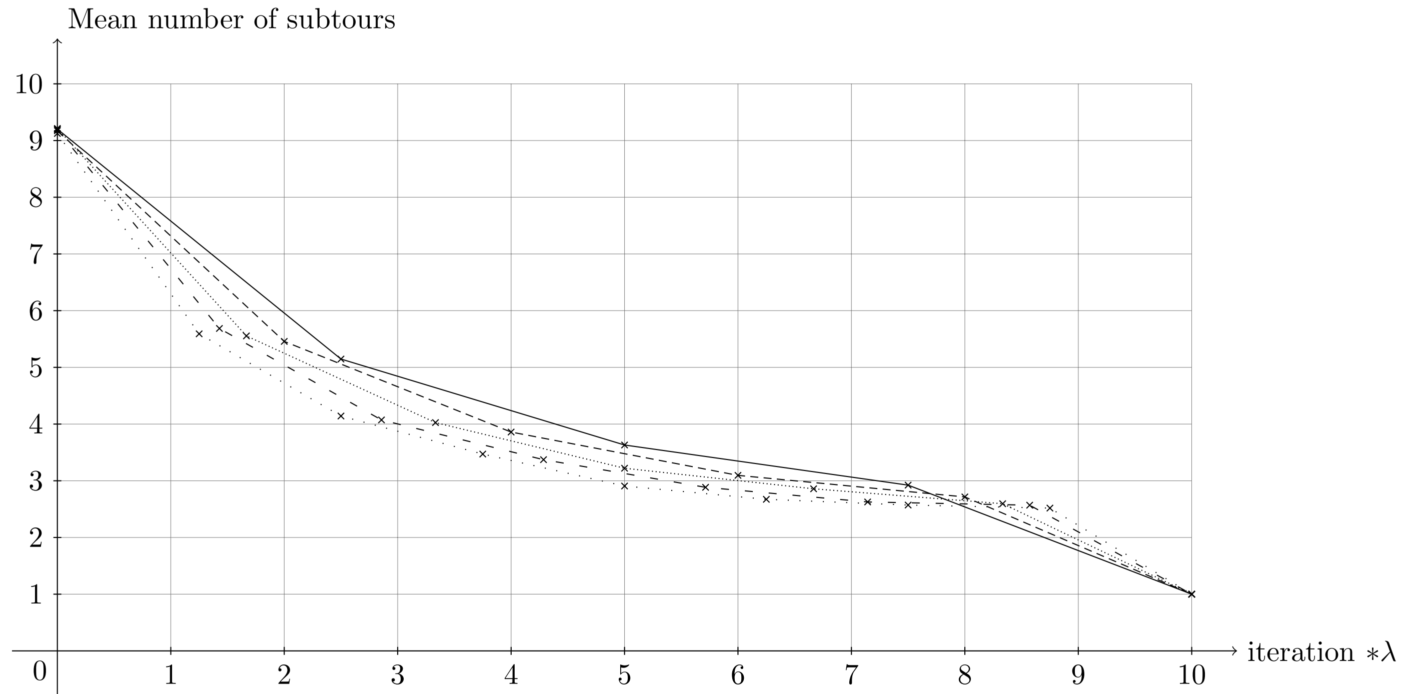

Figure 21: Mean number of subtours during the BasicIntegerTSP in random Euclidean graphs for $n = 60$ sorted according to the number of iterations ($\lambda = 4/10$ (full line), 5/10 (dashed), 6/10 (dotted), 7/10 (loosely dashed), 8/10 (loosely dotted)). We created 100000 graphs.

instance	**BasicIntegerTSP**			$\boldsymbol{HC \mid u = n}$			$\boldsymbol{HC \mid u = 4n/\log_2 n}$			$\boldsymbol{HCD \mid u = 4n/\log_2 n}$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
ch150	13	7	74	150	2	435	**9**	**5**	**223**	14	6	129
kroA150	19	7	136	11	2	268	**8**	**2**	**245**	11	4	130
kroB150	179	8	148	72	2	301	34	4	315	**21**	**4**	**168**

instance	**BasicIntegerTSP**			$\boldsymbol{HC \mid u = n}$			$\boldsymbol{HC \mid u = 4n/\log_2 n}$			$\boldsymbol{HCD \mid u = 4n/\log_2 n}$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
pr152	16	13	184	**5**	**3**	**214**	**5**	**3**	**205**	9	4	174
u159	**6**	**4**	**49**	29	3	292	14	4	303	11	4	140
si175	52	10	183	99	6	494	**40**	**8**	**415**	44	7	263
brg180	44	4	103	54	2	185	102	18	359	**24**	**2**	**27**
rat195	347	6	274	**241**	**3**	**491**	272	4	419	267	5	322
d198	10986	10	301	**483**	**7**	**894**	1094	10	582	3986	9	326
kroA200	677	8	237	**177**	**2**	**362**	941	3	353	690	5	238
kroB200	31	5	121	37	3	292	**23**	**3**	**269**	31	4	164
gr202	**39**	**11**	**77**	2430	3	1216	61	8	330	60	8	217
tsp225	178	9	261	1113	3	981	**138**	**6**	**551**	151	6	341
pr226	5183	10	409	18	1	593	**13**	**1**	**585**	59	3	357
gr229	239	6	311	2984	4	1056	**172**	**7**	**490**	173	8	324
gil262	179	7	268	1052	2	807	**169**	**3**	**564**	217	4	368
a280	157	11	143	–	–	–	**124**	**3**	**733**	181	7	352
pr299	9263	9	413	4051	2	782	1998	5	745	**1716**	**5**	**455**
lin318	6885	8	357	457	2	756	**274**	**5**	**660**	275	5	355
rd400	2401	9	467	9329	5	1494	**983**	**6**	**1018**	1579	8	539
gr431	**2239**	**9**	**453**	–	–	–	4748	9	1734	4214	10	833
pcb442	2737	11	501	–	–	–	3830	16	1796	**2277**	**15**	**888**
u574	17354	6	423	–	–	–	18050	4	1290	**8664**	**4**	**629**
gr666	**17711**	**8**	**789**	–	–	–	23212	4	3104	18031	7	1408
rat783	**30156**	**6**	**457**	–	–	–	–	–	–	–	–	–
mean ratio				*6.016088*			*0.890955*			*0.804727*		
PSS_A_150	**23**	**8**	**100**	103	5	363	36	5	238	28	6	162
PSS_B_150	**13**	**7**	**78**	99	1	424	**13**	**3**	**255**	14	4	146
PSS_C_150	9	5	70	21	1	235	**7**	**3**	**195**	**7**	**3**	**98**
PSS_D_150	8	6	60	50	2	274	**7**	**4**	**197**	**7**	**4**	**112**
PSS_E_150	**9**	**7**	**55**	76	1	339	22	4	274	17	5	149

instance	**BasicIntegerTSP**			$HC \mid u = n$			$HC \mid u = 4n/\log_2 n$			$HCD \mid u = 4n/\log_2 n$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
mean PSS_150	***12.4***	***6.6***	***72.6***	*69.8*	*2*	*327*	*17*	*3.8*	*231.8*	*14.6*	*4.4*	*133.4*
PSS_A_200	**72**	**7**	**163**	560	3	613	77	4	376	157	6	304
PSS_B_200	125	7	148	452	2	582	**76**	**6**	**348**	205	5	250
PSS_C_200	84	8	178	107	1	373	**35**	**4**	**341**	43	4	220
PSS_D_200	**29**	**5**	**102**	173	2	425	40	3	336	44	5	190
PSS_E_200	65	9	139	411	2	561	29	4	301	**21**	**2**	**151**
mean PSS_200	*75*	*7.2*	*146*	*340.6*	*2*	*510.8*	***51.4***	***4.2***	***340.4***	*94*	*4.4*	*223*
PSS_A_250	**138**	**9**	**186**	1154	4	923	158	6	540	163	7	334
PSS_B_250	642	7	263	689	3	599	**198**	**5**	**497**	533	5	359
PSS_C_250	156	6	219	1545	1	846	**57**	**3**	**333**	136	4	275
PSS_D_250	273	6	220	501	2	542	**192**	**6**	**479**	199	5	292
PSS_E_250	103	5	156	339	1	675	**70**	**4**	**511**	105	4	252
mean PSS_250	*262.4*	*6.6*	*208.8*	*845.6*	*2.2*	*717*	***135***	***4.8***	***472***	*227.2*	*5*	*302.4*
PSS_A_300	866	6	295	3574	2	1142	575	5	648	**460**	**4**	**357**
PSS_B_300	1411	8	348	4297	3	1059	**627**	**4**	**672**	865	6	402
PSS_C_300	1071	8	339	2071	3	848	**236**	**6**	**567**	687	7	474
PSS_D_300	**229**	**6**	**290**	2419	4	962	320	5	544	339	5	416
PSS_E_300	577	7	272	1543	3	726	**283**	**6**	**526**	436	6	344
mean PSS_300	*830.8*	*7*	*308.8*	*2780.8*	*3*	*947.4*	***408.2***	***5.2***	***591.4***	*557.4*	*5.6*	*398.6*
PSS_A_350	411	5	252	3186	2	904	332	3	657	**286**	**4**	**377**
PSS_B_350	1021	8	339	11818	2	1102	1234	7	691	**985**	**5**	**463**
PSS_C_350	248	6	207	1243	2	936	**232**	**4**	**750**	358	5	390
PSS_D_350	1718	9	412	4271	3	1087	**529**	**3**	**691**	957	5	428
PSS_E_350	556	5	261	4560	3	1208	485	4	695	**323**	**4**	**408**
mean PSS_350	*790.8*	*6.6*	*294.2*	*5015.6*	*2.4*	*1047.4*	***562.4***	***4.2***	***696.8***	*581.8*	*4.6*	*413.2*
PSS_A_400	8456	8	454	82054	3	1328	10463	5	980	**8245**	**5**	**594**
PSS_B_400	88849	7	438	1043519	2	1661	57469	5	879	**39759**	**6**	**589**
PSS_C_400	780	6	312	53580	3	1755	**779**	**6**	**858**	875	4	450
PSS_D_400	2052	8	451	122357	2	1998	1546	4	796	**1081**	**4**	**434**

instance	BasicIntegerTSP			$HC \mid u = n$			$HC \mid u = 4n/\log_2 n$			$HCD \mid u = 4n/\log_2 n$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
PSS_E_400	2847	7	332	222902	2	1409	2450	4	660	**2151**	**4**	**515**
mean PSS_400	*20596.8*	*7.2*	*397.4*	*304882.4*	*2.4*	*1630.2*	*14541.4*	*4.8*	*834.6*	***10422.2***	***4.6***	***516.4***
PSS_A_450	**2107**	**5**	**279**	–	–	–	2947	3	872	3595	4	535
PSS_B_450	68338	8	413	–	–	–	**57921**	**8**	**906**	94135	6	575
PSS_C_450	46360	10	596	–	–	–	33505	6	1166	**13425**	**7**	**723**
PSS_D_450	**1212**	**6**	**368**	–	–	–	1718	3	902	1644	4	520
PSS_E_450	1539	8	391	–	–	–	2438	5	1098	**1345**	**7**	**637**
mean PSS_450	*23911.2*	*7.4*	*409.4*	–	–	–	***19705.8***	***5***	***988.8***	*22828.8*	*5.6*	*598*
PSS_A_500	15330	6	436	–	–	–	**7531**	**4**	**1000**	10323	5	629
PSS_B_500	**16883**	**6**	**352**	–	–	–	33464	4	1558	79362	5	727
PSS_C_500	3724	5	428	–	–	–	**1337**	**4**	**955**	1437	5	585
PSS_D_500	322951	9	567	–	–	–	339694	7	1236	**307921**	**6**	**743**
PSS_E_500	243378	9	679	–	–	–	**110212**	**4**	**1113**	134563	9	889
mean PSS_500	*120453.2*	*7*	*492.4*	–	–	–	***98447.6***	***4.6***	***1172.4***	*106721.2*	*6*	*714.6*
mean ratio				*12.132531*			*0.898420*			*1.040018*		
mean ratio of all				*9.760849*			*0.895621*			*0.951784*		

Table 8: Results for BasicIntegerTSP and for different variants of the approach which uses the hierarchical clustering. “sec.” is the time in seconds, “#i.” the number of iterations and “#c.” the number of subtour constraints added to the ILP before starting the last iteration. The entries “–” by TSPLIB instances cannot be computed with 16 GB RAM or would take more than 12 hours.

- **BasicIntegerTSP**
- $\boldsymbol{HC \mid u = n}$ – hierarchical clustering; the constraints cannot be dropped and the maximum size of a solved cluster is $u = n$ (i.e. in fact, there is no upper bound)
- $\boldsymbol{HC \mid u = 4n/\log_2 n}$ – hierarchical clustering; the constraints cannot be dropped and the maximum size of a solved cluster is $u = 4\frac{n}{\log_2 n}$

- $\boldsymbol{HCD \mid u = 4n/\log_2 n}$ – hierarchical clustering; the constraints can be dropped and the maximum size of a solved cluster is $u = 4\frac{n}{\log_2 n}$

instance	**BasicIntegerTSP**			$\boldsymbol{C \mid c = \lfloor n/5 \rfloor}$			$\boldsymbol{RC_3 \mid c = \lfloor n/5 \rfloor}$			$\boldsymbol{RC_3 \mid c = n}$			$\boldsymbol{HCD \mid u = 4n/\log_2 n}$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
ch150	13	7	74	12	6	114	**9**	**6**	**109**	16	7	117	14	6	129
kroA150	19	7	136	25	6	187	43	6	166	33	5	185	**11**	**4**	**130**
kroB150	179	8	148	53	4	219	138	5	215	44	5	202	**21**	**4**	**168**
pr152	16	13	184	17	12	181	17	11	204	18	12	181	**9**	**4**	**174**
u159	6	4	49	6	4	149	5	5	151	**3**	**3**	**70**	11	4	140
si175	52	10	183	**31**	**10**	**213**	55	13	250	35	9	196	44	7	263
brg180	44	4	103	**17**	**3**	**81**	19	8	102	121	11	316	24	2	27
rat195	347	6	274	246	4	268	275	6	315	**114**	**6**	**257**	267	5	322
d198	10986	10	301	4253	11	315	–	–	–	4762	9	321	**3986**	**9**	**326**
kroA200	677	8	237	332	6	214	350	5	190	**287**	**4**	**171**	690	5	238
kroB200	31	5	121	29	5	148	**21**	**5**	**147**	32	4	123	31	4	164
gr202	39	11	77	50	8	233	36	6	174	**25**	**6**	**143**	60	8	217
tsp225	178	9	261	100	9	223	**84**	**10**	**235**	100	8	300	151	6	341
pr226	5183	10	409	3614	6	363	36744	5	403	12944	9	415	**59**	**3**	**357**
gr229	239	6	311	335	6	289	**152**	**6**	**256**	311	7	341	173	8	324
gil262	179	7	268	250	8	250	**133**	**7**	**268**	152	6	274	217	4	368
a280	157	11	143	**61**	**4**	**299**	196	11	350	117	9	221	181	7	352
pr299	9263	9	413	6376	7	387	7410	6	416	16059	6	414	**1716**	**5**	**455**
lin318	6885	8	357	537	7	331	386	6	364	1560	6	391	**275**	**5**	**355**
rd400	2401	9	467	**1212**	**7**	**420**	1827	7	438	1522	8	398	1579	8	539
gr431	**2239**	**9**	**453**	3098	9	626	3384	9	647	2496	10	704	4214	10	833
pcb442	2737	11	501	3868	16	770	**1815**	**17**	**567**	2626	16	594	2277	15	888
u574	17354	6	423	11702	4	498	35204	5	580	13722	5	572	**8664**	**4**	**629**
gr666	17711	8	789	**11756**	**7**	**919**	14223	7	1001	13573	7	1002	18031	7	1408

instance	BasicIntegerTSP			$C \mid c = \lfloor n/5 \rfloor$			$RC_3 \mid c = \lfloor n/5 \rfloor$			$RC_3 \mid c = n$			$HCD \mid u = 4n/\log_2 n$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
rat783	**30156**	**6**	**457**	184381	5	701	37805	5	735	38630	6	779	–	–	–
mean ratio				*1.014009*			*1.170299*			*0.983280*			*0.804727*		
PSS_A_150	23	8	100	**18**	**5**	**142**	26	6	141	36	7	155	28	6	162
PSS_B_150	13	7	78	8	4	117	13	5	129	**7**	**4**	**86**	14	4	146
PSS_C_150	9	5	70	**6**	**4**	**63**	8	4	111	9	5	89	7	3	98
PSS_D_150	8	6	60	7	4	100	8	5	97	**6**	**4**	**78**	7	4	112
PSS_E_150	9	7	55	9	6	103	**8**	**4**	**103**	10	4	114	17	5	149
mean PSS_150	*12.4*	*6.6*	*72.6*	***9.6***	***4.6***	***105***	*12.6*	*4.8*	*116.2*	*13.6*	*4.8*	*104.4*	*14.6*	*4.4*	*133.4*
PSS_A_200	72	7	163	**54**	**7**	**218**	69	6	237	81	6	223	157	6	304
PSS_B_200	125	7	148	97	6	217	64	6	213	**58**	**5**	**199**	205	5	250
PSS_C_200	84	8	178	39	6	181	**30**	**6**	**140**	54	7	159	43	4	220
PSS_D_200	**29**	**5**	**102**	52	4	204	34	3	194	36	4	147	44	5	190
PSS_E_200	65	9	139	36	5	217	26	6	144	54	6	193	**21**	**2**	**151**
mean PSS_200	*75*	*7.2*	*146*	*55.6*	*5.6*	*207.4*	***44.6***	***5.4***	***185.6***	*56.6*	*5.6*	*184.2*	*94*	*4.4*	*223*
PSS_A_250	138	9	186	160	8	258	338	9	287	**119**	**7**	**242**	163	7	334
PSS_B_250	642	7	263	**306**	**6**	**295**	542	5	313	366	6	259	533	5	359
PSS_C_250	156	6	219	**104**	**4**	**175**	110	6	229	135	5	211	136	4	275
PSS_D_250	273	6	220	**186**	**6**	**262**	377	6	316	403	7	293	199	5	292
PSS_E_250	103	5	156	**66**	**5**	**207**	68	4	271	110	5	239	105	4	252
mean PSS_250	*262.4*	*6.6*	*208.8*	***164.4***	***5.8***	***239.4***	*287*	*6*	*283.2*	*226.6*	*6*	*248.8*	*227.2*	*5*	*302.4*
PSS_A_300	866	6	295	1233	7	343	576	6	324	467	5	311	**460**	**4**	**357**
PSS_B_300	1411	8	348	1139	6	391	1100	7	431	1146	7	372	**865**	**6**	**402**
PSS_C_300	1071	8	339	608	6	392	**331**	**6**	**314**	458	6	312	687	7	474
PSS_D_300	**229**	**6**	**290**	276	6	321	268	7	307	396	7	374	339	5	416
PSS_E_300	577	7	272	**353**	**7**	**322**	**353**	**5**	**320**	464	6	334	436	6	344
mean PSS_300	*830.8*	*7*	*308.8*	*721.8*	*6.4*	*353.8*	*525.6*	*6.2*	*339.2*	*586.2*	*6.2*	*340.6*	*557.4*	*5.6*	*398.6*
PSS_A_350	411	5	252	695	5	275	513	5	277	375	4	268	**286**	**4**	**377**
PSS_B_350	1021	8	339	**793**	**7**	**363**	1027	8	362	900	7	353	985	5	463

instance	**BasicIntegerTSP**			$C \mid c = \lfloor n/5 \rfloor$			$RC_3 \mid c = \lfloor n/5 \rfloor$			$RC_3 \mid c = n$			$HCD \mid u = 4n/\log_2 n$		
	sec.	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**	**sec.**	**#i.**	**#c.**
PSS_C_350	248	6	207	**196**	**5**	**232**	326	7	280	296	5	310	358	5	390
PSS_D_350	1718	9	412	**749**	**8**	**385**	1047	5	381	781	6	428	957	5	428
PSS_E_350	556	5	261	471	6	356	364	4	368	352	4	339	**323**	**4**	**408**
mean PSS_350	*790.8*	*6.6*	*294.2*	*580.8*	*6.2*	*322.2*	*655.4*	*5.8*	*333.6*	***540.8***	***5.2***	***339.6***	*581.8*	*4.6*	*413.2*
PSS_A_400	8456	8	454	16648	6	471	24941	7	489	28803	7	516	**8245**	**5**	**594**
PSS_B_400	88849	7	438	72010	7	496	77325	7	503	59499	7	497	**39759**	**6**	**589**
PSS_C_400	**780**	**6**	**312**	1198	6	430	831	5	406	1095	7	453	875	4	450
PSS_D_400	2052	8	451	1639	5	436	**591**	**5**	**454**	1595	6	452	1081	4	434
PSS_E_400	2847	7	332	**1602**	**6**	**434**	3724	5	390	1608	6	408	2151	4	515
mean PSS_400	*20596.8*	*7.2*	*397.4*	*18619.4*	*6*	*453.4*	*21482.4*	*5.8*	*448.4*	*18520*	*6.6*	*465.2*	***10422.2***	***4.6***	***516.4***
PSS_A_450	2107	5	279	2921	4	333	2915	5	456	**1385**	**4**	**383**	3595	4	535
PSS_B_450	68338	8	413	15587	7	439	**12828**	**5**	**442**	24941	8	494	94135	6	575
PSS_C_450	46360	10	596	38930	9	697	35388	6	632	22898	7	647	**13425**	**7**	**723**
PSS_D_450	1212	6	368	**948**	**6**	**460**	2175	6	429	2120	7	388	1644	4	520
PSS_E_450	1539	8	391	2210	8	480	1786	7	434	1901	7	432	**1345**	**7**	**637**
mean PSS_450	*23911.2*	*7.4*	*409.4*	*12119.2*	*6.8*	*481.8*	*11018.4*	*5.8*	*478.6*	***10649***	***6.6***	***468.8***	*22828.8*	*5.6*	*598*
PSS_A_500	15330	6	436	10907	6	576	14786	5	543	**6118**	**6**	**531**	10323	5	629
PSS_B_500	16883	6	352	**12299**	**5**	**453**	19681	4	483	186708	5	535	79362	5	727
PSS_C_500	3724	5	428	3063	6	519	2643	4	471	2339	5	440	**1437**	**5**	**585**
PSS_D_500	322951	9	567	514403	8	701	**231961**	**6**	**618**	314232	9	684	307921	6	743
PSS_E_500	243378	9	679	167194	8	718	125303	9	685	**82051**	**9**	**671**	134563	9	889
mean PSS_500	*120453.2*	*7*	*492.4*	*141573.2*	*6.6*	*593.4*	***78874.8***	***5.6***	***560***	*118289.6*	*6.8*	*572.2*	*106721.2*	*6*	*714.6*
mean ratio				*0.898743*			*0.964008*			*1.180427*			*1.040018*		
mean ratio of all				*0.943076*			*1.041367*			*1.104601*			*0.951784*		

Table 9: Comparison between different variants of our approach. "sec." is the time in seconds, "#i." the number of iterations and "#c." the number of subtour constraints added to the ILP before starting the last iteration. The entries "–" by TSPLIB instances cannot be computed with 16 GB RAM.

- **BasicIntegerTSP**
- $\boldsymbol{C \mid c = \lfloor n/5 \rfloor}$ – clustering for $c = \lfloor \frac{n}{5} \rfloor$
- $\boldsymbol{RC_3 \mid c = \lfloor n/5 \rfloor}$ – restricted clustering for $c = \lfloor \frac{n}{5} \rfloor$; the minimum size of a cluster is 3
- $\boldsymbol{RC_3 \mid c = n}$ – restricted clustering for $c = n$; the minimum size of a cluster is 3
- $\boldsymbol{HCD \mid u = 4n/\log_2 n}$ – hierarchical clustering; the constraints can be dropped and the maximum size of a solved cluster is $u = 4\frac{n}{\log_2 n}$