An Efficient Iterated Local Search Heuristic for the Split Delivery Vehicle Routing Problem

Weibo Lin
Alkaid Lab of Huawei Cloud
Huawei Technologies Co., Ltd.

Zhu He
Alkaid Lab of Huawei Cloud
Huawei Technologies Co., Ltd.

Shibiao Jiang
Alkaid Lab of Huawei Cloud
Huawei Technologies Co., Ltd.

Fuda Ma
Alkaid Lab of Huawei Cloud
Huawei Technologies Co., Ltd.

Zhipeng Lü
School of Computer Science and Technology
Huazhong University of Science and Technology

Email of Corresponding Author: linweibo@huawei.com

Abstract: This paper addresses the Split Delivery Vehicle Routing Problem (SDVRP), a relaxation of the Capacitated Vehicle Routing Problem (CVRP) where the same customer can be served by multiple vehicles. Our approach is based on Iterated Local Search (ILS) and employs an improved perturbation mechanism and a new neighborhood structure called *SD-Swap**. By implementing a Lazy Static Move Descriptor (LSMD) mechanism, the proposed algorithm is quite efficient.

Team name: [Alkaid-X]
Solver name: [AlkaidSD]
VRP tracks: [VRP with Split Deliveries]

1 Introduction

The Split Delivery Vehicle Routing Problem (SDVRP) is a relaxation of the classical Capacitated Vehicle Routing Problem (CVRP) where the customers' demands are allowed to be split. Dror and Trudeau first introduced this problem in [5] and demonstrated that splitting can improve both the routing cost and the number of vehicles. The SDVRP is defined on a complete and undirected graph $G = (V, E)$, where $V = \{0, 1, \ldots, n\}$ is the set of vertices. Vertex 0 corresponds to a depot with identical vehicles of capacity Q and vertices $\{1, \ldots, n\}$ represent n customers with positive demands $d_i (1 \leq i \leq n)$. Each edge $(i, j) \in E$ is associated with a non-negative travel cost c_{ij}. A feasible solution to SDVRP is a set of routes that begin and end at the depot together with amounts q_{ik} specifying the amount of the demand by customer i assigned to route k, such that the total amount delivered to i over all routes k is equal to its demand d_i, and the total demand assigned to each route does not exceed the capacity Q of a vehicle. The objective of SDVRP is to find a feasible solution which minimizes the sum of the travel costs.

2 Solution Approach

The proposed solver, called AlkaidSD, is a multi-start heuristic mostly based on the iterated local search (ILS) [9] paradigm. Algorithm 1 in the appendix presents the high-level pseudocode of AlkaidSD. The algorithm executes multiple iterations (lines 3-20) until the time limit is met. Each iteration consists of a construction phase (line 4), which builds an initial feasible solution using the constructive heuristic proposed in [10]. Next, we perform local search using Randomized Variable Neighborhood Descent (RVND) [7] to further improve the solution (line 9). The late acceptance hill climbing criterion [3] is employed to determine whether the new solution should be accepted (lines 18-19). Finally, the last accepted solution is perturbed utilizing the perturbation mechanism described in Section 2.5. The main ILS loop (lines 8-20) is terminated if it fails to find a better solution in successive I_{ILS} steps. In the following subsections, we describe the components of our AlkaidSD solver in details.

2.1 Shortest Path Matrix

In the SDVRP track of 12th DIMACS Implementation Challenge, a rounded Euclidean distance is used as the travel cost between customers, i.e., the cost to travel from customer i to customer j is

$$d_{ij} = \lfloor \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + 0.5 \rfloor, \tag{1}$$

where (x_i, y_i) and (x_j, y_j) are geometric coordinates of customers i and j. The distance metric defined in Eq. 1 may violate the triangle inequality. Hence we adopt the Floyd–Warshall algorithm

[6] to compute a shortest path matrix of customers. The subsequent search procedure is performed based on the shortest path matrix. At last, the best solution found so far will be restored by unfolding the shortest path between each customer. For example, if there is a vehicle traveling from customer i to customer j and the shortest path between i and j goes through k, customer k with zero delivery will be inserted between i and j in the restored solution.

2.2 Construction

The initial solution is built using an insertion-based heuristic by Penna et al. [10]. It consists of two insertion criteria, namely the modified cheapest feasible insertion criterion and nearest feasible insertion criterion. Also, two insertion strategies were employed, specifically the sequential insertion strategy and the parallel insertion strategy. First, $K_{min} = \lceil \sum_{i=1}^{n} d_i/Q \rceil$ empty routes are generated where d_i is the demand of each customer and Q is the capacity of the vehicle. Next, each route is filled with a distinct and randomly selected customer. An insertion criterion and an insertion strategy is also chosen at random. The initial solution is then generated by inserting the unrouted customers to the routes iteratively. Once the algorithm fails to find a feasible insertion place, a new route is added. We refer to [10] for more details of the construction procedure.

2.3 Local Search

As mentioned earlier, the local search is performed based on RVND and uses a number of inter-route neighborhood operators:

- $Swap(n, m)$: An exchange of a contiguous sequence of n customers with another sequence of m customers belonging to a different route [12]. The reverse of each sequence is also considered if the length of the sequence is greater than 1, thus yielding up to 4 possible combinations. In this paper, we implement the neighborhoods associated with $n, m = (1, 0), (2, 0), (2, 1), (2, 2)$.

- $Swap^*$: The $Swap^*$ neighborhood is proposed by Vidal [13] for the CVRP. It can also be directly applied to the SDVRP. Exchanges of two customers i and j from different routes r_1 and r_2 without an insertion in place are considered in this neighborhood. In the process, i can be inserted in any position of r_2, and j can likewise be inserted in any position of r_1. Vidal [13] also provided an efficient method to explore the $Swap^*$ neighborhood in $O(n^2)$-time where n is the number of customers.

- $SD\text{-}Swap^*$: Boudia et al. [2] introduced two split swap operators, namely $SD\text{-}Swap(1, 1)$ and $SD\text{-}Swap(2, 1)$. They are extensions of the classical $Swap(1, 1)$ and $Swap(2, 1)$ operators. In this paper, we extend the $Swap^*$ neighborhood [13] to the split adaptation for the SDVRP, yielding to the $SD\text{-}Swap^*$ operator. Let $i \in r_1$ and $j \in r_2$ be two customers and d'_i and d'_j the amounts of the demand by i and j assigned to routes r_1 and r_2. Without loss of generality, we assume that $d'_i > d'_j$. Our $SD\text{-}Swap^*$ operator removes customer j from r_2 with its amount

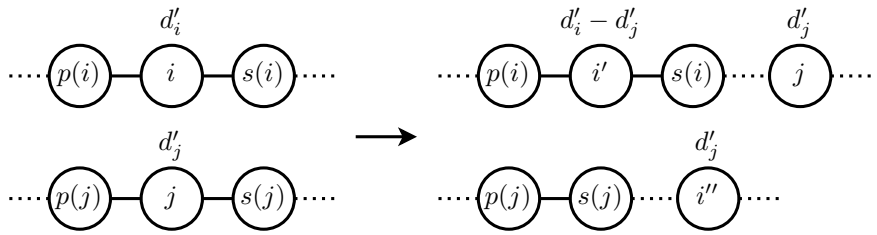

Figure 1: The *SD-Swap** operator.

d'_j and inserts it to r_1. In parallel, customer i is split to two copies: the one i' in the original position of r_1 with amount $d'_i - d'_j$ and the other one i'' in r_2 with amount d'_j. In this process, j can be inserted in any position of r_1, and the copy i'' in r_2 can likewise be placed in any position. Figure 1 shows an example of a move of *SD-Swap**. The case $d'_i = d'_j$ corresponds to a case of *Swap** and it is not considered in this neighborhood structure. The *SD-Swap** neighborhood can also be explored in $O(n^2)$-time using a method similar to that in [13].

- *Cross*: An inter-route adaptation of the 2-opt procedure designed for the Traveling Salesman Problem (TSP) [8]. *Cross* swaps the tail of two involved routes at some position, where the tail of a route is a contiguous sequence of customers belonging to the final part of the route.

The above operators are structured in RVND and the best feasible improvement strategy is considered for each operator. Similar to [2] and [11], we also adopt a repair operator to remove duplicates if there is a route containing customers that are visited more than once. Such route can be generated due to some inter-route move. The repair operator iteratively deletes the duplicate whose removal yields the largest cost reduction until there is no duplicate customers in the route. In addition, an intra-route local search is also performed on the modified routes to further improve the solution quality. For simplicity, we only adopt two intra-route neighborhoods in this paper, and they are also organized using RVND.

- *Exchange*: Swap the position of two customers in the same route.
- *Reinsertion*: Remove a customer and insert it back to another position of the same route.

Algorithms 2 and 3 in the appendix give the complete framework of the local search procedure.

2.4 Lazy Static Move Descriptor

Static Move Descriptor (SMD) [14] is a technique that significantly speeds up local search based algorithms. It replaces the classical for-loop exploration of neighborhoods with a structured inspection of the moves associated with a local search operator, through the careful design of specialized data structures and procedures. In this paper, we design a lazy version of the SMD strategy, called Lazy Static Move Descriptor (LSMD). The proposed LSMD is quite simple and easy to implement.

Specifically, we maintain a matrix cache **BestMove**[][] for each inter-route operator where **BestMove**$[r_1][r_2]$ stores the best move of the corresponding neighborhood associated with routes

r_1 and r_2. Since each local search step in our algorithm affects two routes of the solution, only two rows and two columns of the matrix need to be updated after a neighborhood move. Furthermore, we adopt a lazy update mechanism where the matrix will be updated only when the corresponding operator is about to be used. Note that a route r may be modified more than once before the application of an operator in the RVND procedure. By adopting lazy update, we only need to reevaluate the moves associated with r at most once, and hence can avoid more unnecessary reevaluations.

2.5 Perturbation Mechanism

Local optimal solutions are perturbed using an improved algorithm which combines the Slack Induction by String Removals (SISRs) method [4] and the SplitReinsertion approach [2, 11]. The pseudocode of the perturbation mechanism is presented in Algorithm 4 of the appendix. First, k customers are selected using adjacent string removal, the SISRs' ruin method, and all the occurrences of the selected customers are removed from the solution (lines 2-3). Next, the absent customers are sorted using one of the following orders: *Random*, *Demand*, *Far* or *Close*. The sorting mechanism adopted here is the same as that in [4], we refer to [4] for more details. Finally, each removed customer is inserted using SplitReinsertion with route-blink (lines 5-6).

Algorithm 5 in the appendix gives the pseudocode of our route-blink adaptation of SplitReinsertion. First, each route, whose residual capacity a_r is greater than zero, is added to a list R. Then R is sorted by u_r/a_r where u_r is the least additional cost of inserting customer c in r. The customer c is inserted to each route in R until it receives its full demand d_c. The main difference between our adaptation and SplitReinsertion is that we skip a route with a probability of β if the remainder routes still hold enough capacity for customer c (lines 14-15). The blink strategy is used to balance the exploration-exploitation trade-off, which is important for a perturbation mechanism.

3 Experiments

We carry out experiments to evaluate AlkaidSD on all benchmarks of the SDVRP track in 12th DIMACS Implementation Challenge. Our algorithm is implemented in C++ and compiled by Clang 14.0.0 with optimization option '-O3'. All the experiments are conducted on a server equipped with Intel Core i7-8700 CPU at 3.20GHz. In the SDVRP track, the time limit for each instance is to be 30 minutes on a processor with a CPU mark of 2,000. According to CPU marks[1], the time limit is fixed to 1,347 seconds per instance in our environment. There are 11 parameters in AlkaidSD. They are optimized using Optuna [1] based on a number of randomly, selected instances from the benchmarks. The parameter values used in the experiment are presented in Table 1 of the appendix.

In order to assess the effectiveness of our proposed components, two alternative algorithms are designed where AlkaidSD1 utilizes $SD\text{-}Swap(1,1)$ and $SD\text{-}Swap(1,2)$ [2] instead of $SD\text{-}Swap^*$

[1] `https://www.cpubenchmark.net/singleThread.html`

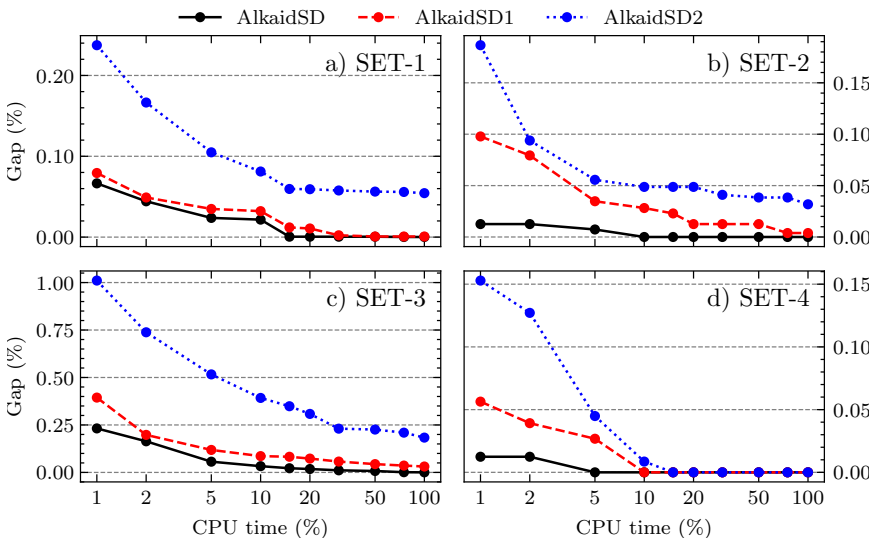

Figure 2: Convergence of the algorithms over time for different datasets.

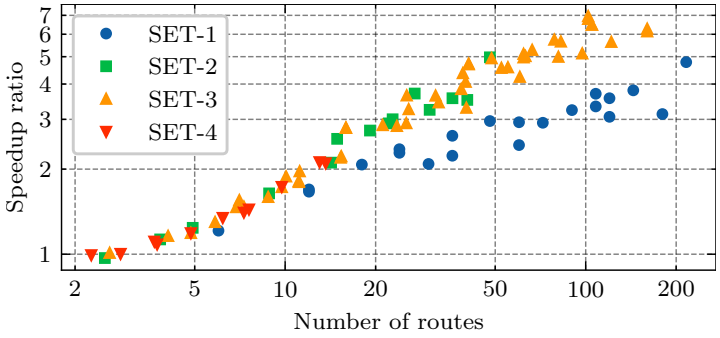

Figure 3: Speedup ratio brought by LSMD.

operator and AlkaidSD2 utilizes Multiple-k-Split [11] instead of our perturbation mechanism. Tables 2-5 give the full results of each algorithm on all instances used in the SDVRP track. Column 'Result' refers to the best solution value obtained by each algorithm in 1,347 seconds. The time for obtaining the best solution is presented in column 'Time' (the unit is second). Every algorithm is executed once with a fixed random seed. As visible in these experiments, AlkaidSD obtains the best final results on 92 out of 95 instances.

We further show the statistical results in Figure 2 to compare the three algorithms. The best solution value after 1%, 2%, 5%, 10%, 15%, 20%, 30%, 50%, 75% and 100% of the time limit is recorded to measure the performance of the algorithms at different stages of the search. The vertical axis in Figure 2 represents the average gap between the solution value of the algorithm and the best solution value found in this paper. From Figure 2, we can observe that the proposed *SD-Swap** operator and perturbation mechanism also play key roles in terms of convergence speed.

Finally, Figure 3 displays the speedup ratio brought by LSMD on instances of different types. As shown, LSMD is powerful to speed up our local search algorithm, especially on the instances having a large number of routes.

References

[1] T. Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[2] M. Boudia, C. Prins, and M. Reghioui. "An effective memetic algorithm with population management for the split delivery vehicle routing problem". In: *International Workshop on Hybrid Metaheuristics*. Springer. 2007, pp. 16–30.

[3] E. K. Burke and Y. Bykov. "The late acceptance hill-climbing heuristic". In: *European Journal of Operational Research* 258.1 (2017), pp. 70–78.

[4] J. Christiaens and G. Vanden Berghe. "Slack induction by string removals for vehicle routing problems". In: *Transportation Science* 54.2 (2020), pp. 417–433.

[5] M. Dror and P. Trudeau. "Split delivery routing". In: *Naval Research Logistics (NRL)* 37.3 (1990), pp. 383–402.

[6] R. W. Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), p. 345.

[7] P. Hansen, N. Mladenović, and J. A. M. Pérez. "Variable neighbourhood search: methods and applications". In: *Annals of Operations Research* 175.1 (2010), pp. 367–407.

[8] M. Jünger, G. Reinelt, and G. Rinaldi. "The traveling salesman problem". In: *Handbooks in Operations Research and Management Science* 7 (1995), pp. 225–330.

[9] H. R. Lourenço, O. C. Martin, and T. Stützle. "Iterated local search". In: *Handbook of Metaheuristics*. Springer, 2003, pp. 320–353.

[10] P. H. V. Penna, A. Subramanian, and L. S. Ochi. "An iterated local search heuristic for the heterogeneous fleet vehicle routing problem". In: *Journal of Heuristics* 19.2 (2013), pp. 201–232.

[11] M. M. Silva, A. Subramanian, and L. S. Ochi. "An iterated local search heuristic for the split delivery vehicle routing problem". In: *Computers & Operations Research* 53 (2015), pp. 234–249.

[12] É. Taillard et al. "A tabu search heuristic for the vehicle routing problem with soft time windows". In: *Transportation science* 31.2 (1997), pp. 170–186.

[13] T. Vidal. "Hybrid genetic search for the CVRP: open-source implementation and swap* neighborhood". In: *Computers & Operations Research* 140 (2022), p. 105643.

[14] E. E. Zachariadis and C. T. Kiranoudis. "A strategy for reducing the computational complexity of local search-based methods for the vehicle routing problem". In: *Computers & Operations Research* 37.12 (2010), pp. 2089–2105.

A Appendix

A.1 Algorithm Framework

Algorithm 1: Overall framework of AlkaidSD

Input: an instance I of the SDVRP

Output: the best solution s^* found so far

```
1  f* ← ∞
2  Floyd-Warshall()
3  while time limit is not met do
4      s ← Construction()
5      s' ← s
6      f*_iter ← f(s)
7      stagnation ← 0
8      while stagnation < I_ILS do
9          s ← RVND(s)
10         if f(s) < f* then
11             f* ← f(s)
12             s* ← s
13         if f(s) < f*_iter then
14             f*_iter ← f(s)
15             stagnation ← 0
16         else
17             stagnation ← stagnation + 1
18         if LateAcceptanceHillClimbing(s', s, history) then
19             s' ← s
20         s ← Perturb(s')
21 s* ← Restore(s*)
22 return s*
```

Algorithm 2: RVND

```
1  Procedure RVND(s):
2      NL ← list containing all inter-route neighborhoods
3      Shuffle(NL)
4      e ← 0
5      while e < Length(NL) do
6          s' ← the best neighbor of s in NL_e
7          if f(s') < f(s) then
8              s ← s'
9              s ← IntraRouteSearch(s)
10             e ← 0
11             Shuffle(NL)
12         else
13             e ← e + 1
14     return s
```

Algorithm 3: IntraRouteSearch

```
1  Procedure IntraRouteSearch(s):
2      NL ← list containing all intra-route neighborhoods
3      Shuffle(NL)
4      e ← 0
5      while e < Length(NL) do
6          s' ← the best neighbor of s in NL_e
7          if f(s') < f(s) then
8              s ← s'
9              e ← 0
10             Shuffle(NL)
11         else
12             e ← e + 1
13     return s
```

Algorithm 4: Perturb

Procedure `Perturb(`s`):`
- $C \leftarrow$ select customers using SISRs' ruin [4]
- $s \leftarrow s \setminus C$
- Sort(C)
- **foreach** customer $c \in C$ **do**
  - $s \leftarrow$ SplitReinsertionWithRouteBlink(s, c)
- **return** s

Algorithm 5: SplitReinsertionWithRouteBlink

Procedure `SplitReinsertionWithRouteBlink(`s, c`):`
- $R \leftarrow$ empty list
- $sumResidual \leftarrow 0$
- **foreach** route $r \in s$ **do**
  - $a_r \leftarrow$ the residual capacity in route r
  - **if** $a_r > 0$ **then**
    - $sumResidual \leftarrow sumResidual + a_r$
    - $u_r \leftarrow$ the least additional cost of inserting customer c in r
    - $R \leftarrow R \cup \{(u_r, a_r)\}$
- Sort R by u_r/a_r in increasing order
- $demand \leftarrow d_c$
- **foreach** $(u_r, a_r) \in R$ **do**
  - $sumResidual \leftarrow sumResidual - a_r$
  - **if** $sumResidual \geq demand$ and rand(0, 1) $< \beta$ **then**
    - **continue**
  - $d'_c \leftarrow \min(a_r, demand)$
  - $demand \leftarrow demand - d'_c$
  - Insert c at the least cost position of r with amount d'_c
- **return** s

A.2 Experimental Results

Scope	Notation	Meaning	Value
SISRs' ruin method [4]	$\bar{c}$	average number of removed customers	36
	L^{max}	maximum cardinality of the removed strings	8
	P_{split}	probability for executing 'split string'	0.740
	α	preserved probability of the 'split string' procedure	0.096
sorting mechanism in [4]	w_{random}	weight of the 'Random' order	0.078
	w_{demand}	weight of the 'Demand' order	0.225
	w_{far}	weight of the 'Far' order	0.942
	w_{close}	weight of the 'Close' order	0.120
blink strategy	β	blink rate	0.021
late acceptance hill climbing	L_h	length of the maintained history list	83
iterated local search	I_{ILS}	maximum allowed steps without improvement	$\min\{K_{min} \cdot n, 5000\}$

Table 1: Parameter values used in the experiment.

Instance	AlkaidSD		AlkaidSD1		AlkaidSD2		Instance	AlkaidSD		AlkaidSD1		AlkaidSD2	
	Result	Time	Result	Time	Result	Time		Result	Time	Result	Time	Result	Time
SD1	**22828**	0.0	**22828**	0.0	**22828**	0.0	SD12	**721353**	2.3	**721353**	1.4	**721353**	3.1
SD2	**70828**	0.0	**70828**	0.0	**70828**	0.0	SD13	**1011040**	0.7	**1011040**	0.5	**1011040**	1.3
SD3	**43060**	0.0	**43060**	0.0	**43060**	0.0	SD14	**1071550**	16.0	**1071550**	61.0	**1071550**	26.2
SD4	**63108**	0.0	**63108**	0.0	**63108**	0.0	SD15	**1508948**	753.5	**1508948**	35.0	1508950	769.6
SD5	**139059**	0.0	**139059**	0.0	**139059**	0.0	SD16	**338109**	202.0	**338109**	419.9	339510	162.9
SD6	**83120**	0.0	**83120**	0.1	**83120**	0.0	SD17	**2649273**	12.9	**2649273**	9.6	**2649273**	15.8
SD7	**364000**	0.0	**364000**	0.0	**364000**	0.0	SD18	1419369	293.0	**1419368**	416.3	1419844	319.3
SD8	**506828**	0.0	**506828**	0.0	**506828**	0.1	SD19	1998716	964.5	**1998656**	540.4	1999510	262.3
SD9	**204424**	0.1	**204424**	0.2	**204424**	0.5	SD20	**3963513**	408.4	**3963513**	139.8	3963514	1282.0
SD10	**268473**	2.5	**268473**	0.6	**268473**	0.9	SD21	**1129352**	492.2	1129530	363.8	1136708	1316.7
SD11	**1328000**	0.0	**1328000**	0.1	**1328000**	0.9							

Table 2: Full results on instances of SET-1.

Instance	AlkaidSD		AlkaidSD1		AlkaidSD2		Instance	AlkaidSD		AlkaidSD1		AlkaidSD2	
	Result	Time	Result	Time	Result	Time		Result	Time	Result	Time	Result	Time
S51D1	**458**	0.0	**458**	0.0	**458**	0.2	S76D2	**1080**	7.0	**1080**	73.7	**1080**	1095.3
S51D2	**703**	1.1	**703**	44.0	**703**	0.9	S76D3	**1418**	13.0	**1418**	11.8	1419	34.8
S51D3	**942**	1.6	**942**	2.0	**942**	31.2	S76D4	**2068**	75.4	**2068**	705.9	**2068**	104.8
S51D4	**1551**	0.7	**1551**	2.0	**1551**	11.0	S101D1	**716**	3.5	**716**	4.0	717	22.7
S51D5	**1328**	1.1	**1328**	5.8	**1328**	2.7	S101D2	**1360**	52.7	**1360**	954.8	**1360**	33.9
S51D6	**2153**	0.5	**2153**	0.8	**2153**	24.9	S101D3	**1854**	79.7	1855	49.4	1855	304.0
S76D1	**592**	0.9	**592**	0.8	**592**	2.0	S101D5	**2758**	2.4	**2758**	258.3	2763	513.9

Table 3: Full results on instances of SET-2.

Instance	AlkaidSD		AlkaidSD1		AlkaidSD2		Instance	AlkaidSD		AlkaidSD1		AlkaidSD2	
	Result	Time	Result	Time	Result	Time		Result	Time	Result	Time	Result	Time
p01_00	**521**	0.0	**521**	0.0	**521**	0.0	p04_1090	**4474**	29.0	4478	526.6	4484	741.8
p01_110	**458**	0.0	**458**	0.1	**458**	0.1	p04_3070	**4273**	360.3	4277	475.0	4288	676.5
p01_1030	**753**	0.0	**753**	15.9	**753**	0.1	p04_7090	**6300**	1150.3	6304	405.2	6317	1085.4
p01_1050	**998**	0.8	**998**	0.4	**998**	0.4	p05_00	**1262**	101.4	**1262**	34.7	1274	43.3
p01_1090	**1480**	1.1	**1480**	7.2	**1480**	6.9	p05_110	**1055**	141.2	**1055**	14.4	1057	1160.6
p01_3070	**1473**	1.0	**1473**	4.7	**1473**	3.9	p05_1030	**2442**	103.8	2443	271.7	2445	358.0
p01_7090	**2142**	0.4	**2142**	3.2	**2142**	36.8	p05_1050	**3419**	386.7	3427	1100.4	3444	143.2
p02_00	**818**	1.9	**818**	4.0	**818**	5.9	p05_1090	**5442**	1004.4	5447	795.0	5458	342.4
p02_110	**612**	0.4	**612**	0.6	**612**	9.0	p05_3070	**5320**	882.9	5321	1048.7	5340	346.8
p02_1030	**1101**	13.4	**1101**	78.3	**1101**	17.9	p05_7090	**8069**	126.5	8076	691.4	8102	1331.3
p02_1050	**1491**	0.7	**1491**	1.7	**1491**	895.2	p10_00	**1262**	101.3	**1262**	34.0	1274	42.9
p02_1090	**2284**	39.2	2287	16.4	2286	256.1	p10_110	**1055**	140.0	**1055**	14.3	1057	1150.1
p02_3070	**2206**	13.4	**2206**	354.5	**2206**	207.8	p10_1030	**2442**	103.7	2443	272.7	2445	356.9
p02_7090	**3198**	149.5	**3198**	101.7	3200	98.9	p10_1050	**3419**	386.7	3427	1100.8	3444	142.9
p03_00	**814**	0.4	**814**	8.0	**814**	52.1	p10_1090	**5442**	1003.2	5447	796.3	5458	335.0
p03_110	**749**	0.7	**749**	0.3	**749**	6.4	p10_3070	**5320**	888.0	5321	1045.6	5340	319.9
p03_1030	**1441**	27.3	**1441**	26.9	**1441**	55.6	p10_7090	**8069**	127.9	8076	695.5	8102	1327.9
p03_1050	**1976**	22.5	**1976**	175.7	1978	366.6	p11_00	**1023**	0.7	**1023**	0.7	**1023**	1051.3
p03_1090	**3051**	39.1	3052	468.9	3054	438.4	p11_110	**1028**	0.9	**1028**	1.0	**1028**	25.8
p03_3070	**2956**	76.0	2957	962.3	2958	12.7	p11_1030	**2862**	47.1	**2862**	296.1	2864	173.6
p03_7090	4331	522.7	**4330**	1339.1	4336	198.8	p11_1050	**4166**	468.2	**4166**	256.7	4169	230.5
p04_00	**1008**	54.0	**1008**	518.4	**1008**	730.7	p11_1090	**6764**	664.0	**6764**	515.1	6770	870.8
p04_110	**909**	2.3	**909**	7.2	**909**	119.0	p11_3070	**6556**	920.9	6558	1066.8	6596	395.2
p04_1030	**1988**	408.1	**1988**	828.3	1992	1270.5	p11_7090	**10068**	10.7	**10068**	636.2	10097	246.4
p04_1050	**2808**	201.6	2811	159.3	2811	762.5							

Table 4: Full results on instances of SET-3.

Instance	AlkaidSD		AlkaidSD1		AlkaidSD2		Instance	AlkaidSD		AlkaidSD1		AlkaidSD2	
	Result	Time	Result	Time	Result	Time		Result	Time	Result	Time	Result	Time
eil22	**375**	0.0	**375**	0.0	**375**	0.0	eilA101	**814**	0.3	**814**	8.0	**814**	51.7
eil23	**569**	0.0	**569**	0.0	**569**	0.0	eilB76	**1002**	1.1	**1002**	2.6	**1002**	84.7
eil30	**503**	0.0	**503**	0.1	**503**	0.0	eilB101	**1059**	4.7	**1059**	15.1	**1059**	179.2
eil33	**835**	0.0	**835**	0.0	**835**	0.0	eilC76	**732**	28.1	**732**	30.9	**732**	10.3
eil51	**521**	0.0	**521**	0.0	**521**	0.0	eilD76	**679**	1.4	**679**	113.3	**679**	31.1
eilA76	**818**	1.9	**818**	3.9	**818**	5.9							

Table 5: Full results on instances of SET-4.