Conditional Markov Chain Search for the Generalised Travelling Salesman Problem for Warehouse Order Picking

1st Olegs Nalivajevs
Computer Science and Electric Engineering
University of Essex
Colchester, UK
olegnalivajev@gmail.com

2nd Daniel Karapetyan
Computer Science and Electric Engineering
University of Essex
Colchester, UK
daniel.karapetyan@gmail.com

Abstract—The Generalised Travelling Salesman Problem (GTSP) is a well-known problem that, among other applications, arises in warehouse order picking, where each stock is distributed between several locations – a typical approach in large modern warehouses. However, the instances commonly used in the literature have a completely different structure, and the methods are designed with those instances in mind. In this paper, we give a new pseudo-random instance generator that reflects the warehouse order picking and publish new benchmark testbeds. We also use the Conditional Markov Chain Search framework to automatically generate new GTSP metaheuristics trained specifically for warehouse order picking. Finally, we report the computational results of our metaheuristics to enable further competition between solvers.

Index Terms—Generalised Travelling Salesman Problem; Conditional Markov Chain Search; Warehouse Order Picking; Automated Algorithm Generation

I. Introduction

The Generalised Travelling Salesman Problem (GTSP) is an well-known extension of the Travelling Salesman Problem (TSP). In GTSP, you are given a set of nodes partitioned into clusters. You are also given the cost of travelling between each pair of nodes (in this paper we assume that the distances symmetric). The objective is to find the shortest cycle that visits exactly one node in each cluster.

GTSP has significantly higher modelling power compared to TSP. Many real world applications can be modelled using GTSP with a good accuracy. Consider any delivery problem; it is common that a delivery driver can choose between several nearby locations (such as either side of the road) where to stop the truck. Note that these locations might be very distant in terms of driving (think of reversing a truck) and thus this decision may have a significant effect on the cost of the route. Another example which we will consider in more detail in this paper is the warehouse order picking problem [1]. Assuming that each stock is located in one place within the warehouse, the order picking problem can be modelled using TSP. However, if a stock is distributed between several locations (potentially remote), one needs to use GTSP to model the problem. Modern warehouses deliberately distribute

their stocks between different locations, partly because this allows shorter order picking times.

There are numerous studies of GTSP algorithms. Some successful heuristics were proposed in [2]–[5]. Particularly effective approaches include metaheuristics such as [6]–[8].

In this paper, we focus on the warehouse order picking application. Our contributions are as follows:

- We developed a pseudo-random instance generator that produces instances simulating the warehouse order picking. We note here that our instances have structure completely different to that commonly used in the literature.
- Using our instance generator, we produced two testbeds, with medium and large instances.
- We use the Conditional Markov Chain Search to automatically design a metaheuristic tuned for warehouse picking instances.
- We give our solutions to our benchmark instances thus enabling other researchers to compare their methods to our solver.

II. Generation of a Metaheuristic

Conditional Markov Chain Search (CMCS) is a modern framework designed for automated generation of optimisation heuristics [9]–[11]. It is a single-point metaheuristic based on multiple components treated as black boxes. Each component is a subroutine that takes a solution and modifies it according to the internal logic. Examples of components are hill climbers and mutations. The behaviour of the control mechanism within CMCS is defined by a set of numeric parameters thus enabling automated generation of CMCS configurations; by tuning these parameters, one can find the 'optimal' control mechanism. Despite being defined by only a small set of numeric parameters, CMCS supports a wide range of control mechanisms. E.g., it can model Variable Neighbourhood Search, (Weighted) Random Hyperheuristic and Iterated Local Search [9].

CMCS performs as follows. It takes as an input the initial solution (usually produced by some construction heuristic, e.g. random solution) and then at each iteration applies one of the

components to it. The component modifies the solution according to the internal logic. The modification is always 'accepted', i.e. there is no backtracking[1]. CMCS only records whether the component improved the solution or not. The choice of the next component depends only on which component was used in the current iteration and whether it improved the solution. Thus the sequence of applied components is a Markov chain, and the control mechanism can be defined by two transition matrices: one for the case when the solution was improved and another one for the case when the solution was not improved. The transitions can be probabilistic, however in this research we only consider deterministic transitions, i.e. the transition matrices consist of zeros and ones.

Since CMCS may worsen the solution, it keeps track of the best solution found during the search and at the end returns that solution.

III. COMPONENTS

CMCS requires a pool of components to draw from when generating configurations. Our pool consists of four components that can be found in the literature, see e.g. [2].

Cluster Optimisation (CO) is a component that selects an optimal route given a fixed sequence of clusters. Such a neighbourhood is exponential in size but it can be explored in polynomial time as this subproblem can be reduced to the shortest path problem. Thus, Cluster Optimisation is a Very Large Scale Neighbourhood Search.

Insertion Hill Climber (IHC) is a stochastic improvement component. It randomly picks a node in the solution, randomly picks a new position for it within the solution and then inserts it into this new position. If the modified solution is not better than the old one, the change is backtracked. (In fact, we use incremental evaluation and hence the time complexity of IHC is $O(1)$.)

Order Mutation (OM) is a stochastic component that may improve or worsen the solution which is identical to IHC except that it never backtracks any changes. In other words, it randomly selects a node in the solution, randomly picks a new position for it within the solution and then inserts it into this new position and returns the modified solution.

Vertex Mutation (VM) is another stochastic component that may improve or worsen the solution. It randomly picks a node within the solution and then replaces it with a randomly picked node from the same cluster.

There have been several data structures used for storing GTSP solutions. We adopted the data structure proposed in [2]. It separates the ordering in the tour from the vertex selection. The ordering is stored in a double-linked list. As the objects in the list are simply the cluster indices from 1 to m, the list is represented by only two integer arrays of size m. The double-linked list is particularly convenient as it naturally represent the cyclic tour. The vertex selection is another integer array of

---

[1] While CMCS control mechanism always accepts any changes, whether improving or worsening the solution, the components such as hill climbers may internally include backtracking.

size m. This data structure enables efficient operations on it, elegant implementations of components and compact (cache-efficient) data structures [12].

IV. WAREHOUSE INSTANCES

We developed a Warehouse GTSP Instances Generator which more accurately models warehouse order pickup problem compared to the standard approach adopted in the literature, see e.g. [6]. Specifically, we did not assume compact clusters with little overlapping; we argue that in a modern warehouse, the locations of stocks of each item are deliberately distributed across the entire warehouse floor. Indeed, storing the items compactly defeats the purpose of distributing them in multiple locations; with compact storage, chances of a pickup route visiting a remote location for only one or a few items would increase whereas with an even distribution, there is a good chance of some item locations being close to the other fragments of the tour.

Our instance generator takes two parameters: the number of clusters m and the number of nodes $n \geq m$. First, it generates coordinates of the nodes on a plane, randomly drawing the x and y coordinates from the range $[0, 200]$. The distances between nodes are computed using Manhattan distance to reflect the typical topology of warehouses. We form the clusters by placing one node in each cluster and then distributing the rest of the nodes randomly between the clusters.

Our instance generator can be downloaded from https://csee.essex.ac.uk/staff/dkarap/warehousegtsp/Instance.java. The instances are in the format of GTSP Instances Library, see https://csee.essex.ac.uk/staff/dkarap/gtsp.html [6].

We generated two benchmark testbeds: Medium and Large, 30 instances each. The Medium instances range from 150 to 202 nodes and 30 to 44 clusters. The Large instances range from 550 to 602 nodes and 105 to 119 clusters. The instances can be downloaded from https://csee.essex.ac.uk/staff/dkarap/warehousegtsp/instances.zip. Each instance is given a name in the form '$\langle m \rangle$wop$\langle n \rangle$'.

V. COMPUTATIONAL EXPERIMENTS

The algorithm described in this paper has been implemented in Java, and the experiments were conducted on MacBook Pro 15-inch 2017 (4 core Intel Core i7, 2.9 GHz processor and 16 GB of memory).

First, we needed to generate CMCS configurations. To generate a configuration, we use a training instance set. Each configuration is evaluated on each of the training instances. Specifically, a solution is produced that visits the first node in each cluster, and the order of clusters in the solution is chosen randomly. Then CMCS is applied to this solution. The time budget given to CMCS is calculated as follows:

$$t = \alpha nm, \tag{1}$$

where α is a constant. We heuristically selected $\alpha = 1.8 \cdot 10^{-5}$ for Medium instances and $\alpha = 3.6 \cdot 10^{-6}$ for Large instances to make sure that each CMCS configuration is given sufficient

time to perform at least a few iterations but at the same time not to run for too long (otherwise many configurations would reach optimal solutions and ranking them would become impossible).

We normalised the objective values obtained by different configurations for each instance:

$$v'_{cI} = \frac{v_{cI} - P_0}{P_{50} - P_0},$$

where v_{cI} is the objective value obtained by configuration c on instance I, v'_{cI} is the corresponding normalised objective value and P_i is the ith percentile in v_{cI} for all configurations c and fixed instance I. The idea to use P_{50} for the upper bound of the normalisation interval is to focus on the high quality solutions and ignore the outliers that otherwise could have a significant effect on the configuration quality metric. The quality of a configuration is computed as

$$q_c = \sum_I v'_{cI}.$$

The configuration c that minimises q_c is then selected.

With a pool of four components, there are more than quarter of a million configurations. Testing all these configurations would be impractical. We follow the idea proposed in [11] and only consider 'meaningful' configurations. We further restrict ourselves to configurations that use exactly three components. As a result, our set of configurations is reduced to 2972 configurations.

Using the above methodology, we generated two configurations: one being trained on medium instances (which we call Conf1) and one using large instances (which we call Conf2). The configurations are shown in Figures 1 and 2. It is interesting to note that the mutation selected in each of these configurations is VM which would be a weak mutation for the typical instances with compact clusters however in the warehouse order picking instances replacing a node with another node from the same cluster may significantly affect the solution.

To enable future competition between solvers, we also report the objective values obtained in our experiments in Tables I and II. The 'Time, sec' column gives the time budget, the 'Best' column reports the best objective value observed in our experiments. The 'Conf1' and 'Conf2' columns report the objective values achieved by the corresponding CMCS configurations. The winning configuration is underlined in each row. Conf2 outperforms Conf1 on both testbeds but particularly on the Large instances. However note that training Conf2 was significantly more expensive computationally. This also demonstrates that the similarity between the training and evaluation instances is important for performance of the generated CMCS configuration.

We also conducted preliminary tests of Conf1 and Conf2 on the standard instances from the GTSP Instance Library to compare them to the state-of-the-art solvers. Our early conclusions are that Conf1 and Conf2 outperform the state-of-the-art metaheuristics in terms of the running time on small
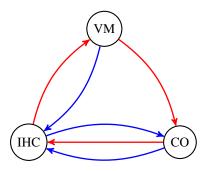

Fig. 1. Conf1, the CMCS configuration trained on small instances.
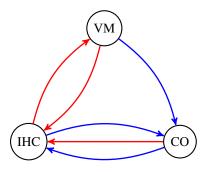

Fig. 2. Conf2, the CMCS configuration trained on medium instances.

instances, however perform poorly on larger instances. This was expected as (a) Conf1 and Conf2 are trained on Warehouse Order Picking Instances and hence are not supposed to perform well on the instances from the GTSP Instances Library, and (b) CMCS is a single-point metaheuristic whereas most powerful metaheuristics are usually population-based. Further experiments are required to compare CMCS configurations to the other solvers on the Warehouse Order Picking Instances. We expect to see that the generated CMCS configurations will perform better than the other solvers as they were specifically trained for these instances. That will be a valuable contribution, as it will demonstrate the advantage of using automated metaheuristic generation.

VI. CONCLUSIONS

This paper discusses an important application of the GTSP to the warehouse order picking. We argue that the GTSP benchmark instances commonly used in the literature do not adequately reflect the structure of the warehouse order picking problem and thus the available GTSP solvers might not be well-suited for the warehouse picking problem. In this paper, we give a new instance generator, benchmark testbeds and automatically generate a metaheuristic using the Conditional Markov Chain Search specifically for the warehouse order picking application. We then report our computational results. This is still work in progress, and further experiments will be needed to establish how our methods compare to the existing solvers.

Instance	Best	Time, sec	Conf1	Conf2
150wop30	808	0.0810	948	980
151wop30	812	0.0815	826	1000
153wop31	702	0.0854	816	1098
155wop31	724	0.0865	766	920
157wop32	694	0.0904	726	712
159wop32	774	0.0916	774	940
160wop33	876	0.0950	1020	1002
162wop33	804	0.0962	1014	890
164wop34	914	0.1004	920	1068
166wop34	844	0.1016	898	1022
168wop35	974	0.1058	1112	1236
169wop35	1014	0.1065	1164	1166
171wop36	898	0.1108	898	1360
173wop36	866	0.1121	1116	942
175wop37	884	0.1166	1050	1110
177wop37	840	0.1179	1034	1012
178wop38	988	0.1218	1058	1028
180wop38	1080	0.1231	1282	1192
182wop39	978	0.1278	1238	1048
184wop39	1084	0.1292	1358	1238
186wop40	1032	0.1339	1400	1264
187wop40	994	0.1346	1322	1218
189wop41	1030	0.1395	1122	1052
191wop41	1020	0.1410	1184	1086
193wop42	1040	0.1459	1438	1040
195wop42	1038	0.1474	1396	1126
196wop43	1072	0.1517	1286	1106
198wop43	1150	0.1533	1278	1242
200wop44	1166	0.1584	1330	1208
202wop44	1194	0.1600	1302	1382

TABLE I
COMPUTATIONAL RESULTS FOR THE MEDIUM WAREHOUSE ORDER PICKING INSTANCES.

Instance	Best	Time, sec	Conf1	Conf2
550wop105	2958	0.2079	3334	3306
551wop105	3170	0.2083	3552	3170
553wop106	2288	0.2110	2812	2698
555wop106	2224	0.2118	2530	2224
557wop107	2432	0.2146	2952	2534
559wop107	2264	0.2153	2626	2348
560wop108	2444	0.2177	2632	2544
562wop108	2486	0.2185	3036	2686
564wop109	2270	0.2213	2750	2270
566wop109	2402	0.2221	2990	2574
568wop110	2430	0.2249	2944	2670
569wop110	2246	0.2253	2800	2692
571wop111	2246	0.2282	2862	2246
573wop111	2338	0.2290	3052	2500
575wop112	2358	0.2318	2892	2466
577wop112	2444	0.2326	2978	2762
578wop113	2296	0.2351	2866	2342
580wop113	2600	0.2359	3086	2626
582wop114	2518	0.2389	3130	2584
584wop114	2188	0.2397	2560	2188
586wop115	2588	0.2426	3192	2588
587wop115	2612	0.2430	3312	2802
589wop116	2650	0.2460	2946	2650
591wop116	2600	0.2468	3070	2696
593wop117	2584	0.2498	2850	2696
595wop117	2450	0.2506	2834	2450
596wop118	2748	0.2532	3450	2974
598wop118	2384	0.2540	3256	2554
600wop119	2574	0.2570	3204	2574
602wop119	2548	0.2579	3042	2548

TABLE II
COMPUTATIONAL RESULTS FOR THE LARGE WAREHOUSE ORDER PICKING INSTANCES.

REFERENCES

[1] J. Karasek, "An overview of warehouse optimization," *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 2, no. 3, pp. 111–117, 2013.

[2] D. Karapetyan and G. Gutin, "Efficient local search algorithms for known and new neighborhoods for the generalized traveling salesman problem," *European Journal of Operational Research*, vol. 219, pp. 234–251, 2012.

[3] ——, "Lin-Kernighan heuristic adaptations for the generalized traveling salesman problem," *European Journal of Operational Research*, vol. 208, no. 3, pp. 221–232, 2011.

[4] K. Helsgaun, "Solving the equality generalized traveling salesman problem using the lin–kernighan–helsgaun algorithm," *Mathematical Programming Computation*, vol. 7, no. 3, pp. 269–287, Sep 2015.

[5] S. L. Smith and F. Imeson, "Glns: An effective large neighborhood search heuristic for the generalized traveling salesman problem," *Computers & Operations Research*, vol. 87, pp. 1–19, 2017.

[6] G. Gutin and D. Karapetyan, "A memetic algorithm for the generalized traveling salesman problem," *Natural Computing*, vol. 9, no. 1, pp. 47–60, 2009.

[7] J. Silberholz and B. Golden, *The Generalized Traveling Salesman Problem: A New Genetic Algorithm Approach*. Boston, MA: Springer US, 2007, pp. 165–181.

[8] C.-M. Pintea, P. C. Pop, and C. Chira, "The generalized traveling salesman problem solved with ant algorithms," *Complex Adaptive Systems Modeling*, vol. 5, no. 1, p. 8, Aug 2017.

[9] D. Karapetyan, A. P. Punnen, and A. J. Parkes, "Markov chain methods for the bipartite boolean quadratic programming problem," *European Journal of Operational Research*, vol. 260, no. 2, pp. 494–506, 2017.

[10] D. Karapetyan, A. J. Parkes, and T. Stützle, "Algorithm configuration: Learning policies for the quick termination of poor performers," in *Proceedings of LION 2018*, ser. LNCS, vol. 11353, 2018, pp. 220–224.

[11] D. Karapetyan and B. Goldengorin, "Conditional markov chain search for the simple plant location problem improves upper bounds on twelve korkel-ghosh instances," in *Optimization Problems in Graph Theory*, B. Goldengorin, Ed. Springer, 2018, pp. 123–147.

[12] D. Karapetyan, G. Gutin, and B. Goldengorin, "Empirical evaluation of construction heuristics for the multidimensional assignment problem," in *London Algorithmics 2008: Theory and Practice*, ser. Texts in algorithmics. London, UK: College Publications, 2009, pp. 107–122.