Algorithm and Knowledge Engineering for the TSPTW Problem

Stefan Edelkamp*, Max Gath*, Tristan Cazenave†, and Fabien Teytaud‡

*Institute for Artificial Intelligence, University of Bremen, Germany, Email: {edelkamp,mgath}@tzi.de
†LAMSADE, Universite Paris Dauphine, France, Email: cazenave@lamsade.dauphine.fr
‡Univ Lille Nord de France, ULCO, LISIC, BP 719, 62228 CALAIS Cedex, Email: teytaud@lisic.univ-littoral.fr

Abstract—The well-known traveling salesman problem (TSP) is concerned with determining the shortest route for a vehicle while visiting a set of cities exactly once. We consider knowledge and algorithm engineering in combinatorial optimization for improved solving of complex TSPs with Time Windows (TSPTW). To speed-up the exploration of the applied Nested Monte-Carlo Search with Policy Adaption, we perform beam search for an improved compromise of search breadth and depth as well as automated knowledge elicitation to seed the distribution for the exploration. To evaluate our approach, we use established TSPTW benchmarks with promising results. Furthermore, we indicate improvements for real-world logistics by its use in a multiagent system. Thereby, each agent computes individual TSPTW solutions and starts negotiation processes on this basis.

I. INTRODUCTION

In the Traveling Salesman Problem (TSP) a set of \( N \) cities (one of which is the depot) and their pairwise distances are given. The task is to find the shortest route that starts and ends at the depot and visits each city only once.

In the Traveling Salesman Problem with Time Windows (TSPTW), additionally to the TSP, each city has to be visited and left within a given time interval. As the Hamiltonian Path problem is a subproblem, TSP, TSPTW and most other TSP variants are computationally hard [7], so that no algorithm polynomial in \( N \) is to be expected.

A genetic search solver for TSPTW problems has been contributed by Potvin and Bengio [20]. Alternative algorithms are constraint logic programming [19], ant colony optimization [16], and generalized insertion heuristics [13]. Exact TSP solvers often apply branch-and-bound search, are usually based on refined bounds [7] and have, e.g., been suggested by [3], [8], [9], [11]. Their scaling, however, is limited.

In real-world logistics applications more general TSP(TW) variants are common: capacities limitations on the vehicle of the traveling salesman, combined delivery and backhaul transportation of goods, premium vs. non-premium tasks, vehicle routing problems with several salesmen to jointly solve a logistics problem too large for one salesman, see [18] for a survey. Solomon [24] provides benchmarks for vehicle routing problems with time window constraints.

Numerous complex TSP problems have a huge state space, but no good heuristic for ordering moves to guide the search toward the best solution. Therefore, randomized search is often applied. For example, Nested Monte-Carlo Search uses random rollouts at its base level. It combines nested calls with randomness in the rollouts and memorization of the best tour.

Nested Monte-Carlo search has been combined with expert knowledge and an evolutionary algorithm [21]. The Monte-Carlo algorithm (with a small level) is reapeatedly called and optimized by a Self-Adaptation Evolution Strategy. However, the effectiveness of this hybrid Nested Monte-Carlo algorithm decreases as the number of cities increases. Biasing Monte-Carlo simulation through Rapid Action Value Estimation (RAVE) in the TSPTW domain has been investigated by Rimmel et al. [22], while Rosin [23] invented Policy Adaption for Nested Monte-Carlo search.

In this paper we perform algorithm engineering to speed-up the process of finding good TSPTW solutions. Among other implementation refinements we use beam search and policy priors. For the TSP, policy priors can be deduced from the pairwise city-to-city distance table, or from a lower bound function like the Hungarian algorithm solving the Assignment Problem (AP) or one of its refinements [14]. Moreover, we show how to elicit knowledge from the definition of the TSP to drive the solution process towards finding good solutions even more quickly.

The paper is structured as follows: we start with (Nested) Monte-Carlo Search and Policy Adaption, and describe known domain-dependent heuristic enhancements to reduce the set of successors in the randomized search. Then, we consider code refactoring and further speed-up techniques that we jointly cast as algorithm engineering. Next, we will see that prior knowledge implemented into an initial policy can greatly reduce search efforts. The experiments are drawn on a selection of TSPTW benchmarks and show improvements to Nested Monte-Carlo search, so that more instances can be solved optimally. Finally, we conclude and discuss the adaptation of the algorithms in an industrial strength multiagent system.

II. MONTE-CARLO SEARCH

Monte-Carlo Tree Search is a class of randomized tree search algorithms that backups values from the leaves of the search tree back to the decision nodes to direct the search towards the best solution found, while maintaining exploration breadth. Thus, the algorithm is a proposed solution to the well-known exploration vs. exploitation dilemma in state space search.
A. UCT

One noticeable representative in this class is the UCT algorithm [15] that dynamically builds a search tree, from whose leaves random walks to the end of the game (rollouts) are initiated. Within the explicit UCT search tree, the algorithm chains down from the root node, and applies a specialized formula that mimics a multi-armed bandit problem to select the best successor [2] of each node. The UCT-formula includes the expected payoff in form of an exploration term, which in more recent work sometimes is dropped or substituted in favor to search knowledge in form of a value of an expert-given or otherwise learned evaluation function. UCT is very successful in playing games and outperforms traditional approaches minimax search with a constraint. For single-agent search challenges, Nested Monte-Carlo (NMC) has been suggested and successfully applied to games like Morpion Solitaire, SameGame, Sudoku and many others [5].

NMC is a recursive algorithm which performs a certain number of rollouts, where a rollout is a random path in the search tree starting from the root and ending at a leaf that can be evaluated to some score value. In each position, a NMC search of level $l$ (initiated by $k$) will perform a level $n-1$ NMC for each action. If the value has decreased to 1 (or to 0 depending on the implementation), a rollout is initiated. At each choice point of a rollout the algorithm chooses the successor that gives the best score when followed by a single random rollout. Similarly, for a rollout of level $l$ it chooses the successor node that gives the best score when followed by a rollout of level $l-1$. Hence, the core objective in NMC is that the search is intensified with increasing depth of the search.

A level 1 maximization example is presented in Figure 1. The leftmost tree illustrates the start. A Monte-Carlo playout is performed for all 3 possible decisions. At the end of each playout a reward is given, and the decision with the best reward is chosen.

In [21] domain-dependent TSP heuristics have been added to bias the Monte-Carlo simulations according to a (Boltzmann softmax) policy, e.g., preferring states with a smaller distance to the last city, a smaller amount of wasted time because a city is visited too-early, or a smaller amount of time left until the end of the time window of a city.

C. Policy Adaption in Nested Monte-Carlo Search

The basis for our algorithm is the implementation of Cazenave and Teytaud [6] for NMC with Policy Rollout Adaption (NPRA), an algorithm originally proposed by [23]. The NMC edge learning algorithm shares similarities the RAVE adaptations applied to UCT search [22]. The rollout is thus biased on a policy $P(u, v)$ for the state space edges $(u, v)$. NPRA applies a level $l$ search with $i$ iterations in which the best policy $P_i(u, v)$ is updated. Moreover, in the first iteration $P_i(u, v)$ is initialized with $P_{i-1}(u, v)$.

D. Domain-Dependent Pruning

Cazenave and Teytaud [6] have enhanced NPRA with pruning rules. These extensions are domain-dependent and are based on the following two preference rules for successor set selection within the rollout.

1) If a time-window constraint is violated extending a partial tour to a successor node $v$, reduce the successor node set to $\{v\}$. The reason is that this violated city should have been visited earlier.

2) Avoid visiting a node if it makes another node fail a time-window constraint. For this case it considers all successor nodes that do not make any other node to fail a constraint.

III. ALGORITHMIC REFINEMENTS

We have performed extensive refactoring and algorithm engineering to enhance the exploration efficiency. The objective of the tuning is simple: the faster the node expansion and the rollouts implementations are, the better the nested search, as it will have more back-up information for decision making.

A. Avoiding Copy Construction

In the original implementation for TSPTW solving with NPRA by [6] the copy constructor is called in each iteration and each level of the search (thus, for each search node). This elegant solution (realized by calling copy = *this) helps understanding the difference of recursive invocation of the search and reinitializing it in each of the iterations. Moreover, parallelization of the search is made easy, as each constructor call can be given to a different computing node. However, for this case of copy construction all non-static member variables of the search class are replicated and copied to the new NMC search class instance.

B. Reducing Memory Allocation

At each search node our implementation (see Fig. 2) copies the policy when doing down from level $l - 1$ to level $l$. To avoid dynamic memory allocation all policies $P_l$, $l = k, \ldots, 0$, in a Level-$k$ search are pre-allocated. Moreover, we moved the copying of the policy to a working temporary and back from inside the iteration loop to its outside.
Pair search(int level) {
    Pair best;
    best.score = MAX;
    if (level == 0) {
        best.score = rollout();
        best.tour = tour.clone();
    } else {
        clone[level] = policy;
        for(int i=0; i<ITERATIONS; i++) {
            Pair r = search(level - 1);
            double score = r.score;
            if (score < best.score) {
                best.score = score;
                best.tour = r.tour.clone();
                adapt(best.tour,level);
            }
            delete r;
        }
        policy = clone[level];
    }
    return best;
}

Fig. 2. Nested Monte-Carlo with Policy Adaption.

At each search node, memory for one tour is reserved and deleted in case no better solution has been found. Hence, the memory consumption of the refined algorithm is bounded by $O(k \cdot N^2)$. The time efforts at each node are bounded by $O(N^2)$. This includes the efforts for policy adaption in case of an established improved solution. Since successor generation takes $O(N^2)$ steps (due to the second heuristics) and given that it is applied to each node in the tour to be generated, for each rollout $O(N^3)$ operations are required.

C. Merging Rollout and Evaluation

In our engineered implementation we avoided the replay of the partial tour in the successor generation function for determining its makespan. Thus, we merged successor generation with the rollout. Moreover, we integrated the evaluation of a tour to a score value into the rollout procedure (see Figure 3). The offset penalizes constraint violations and is set to the predefined maximum value for the distances divided by the number of cities $N$ (This is the largest possible values if MAX is used as an upper bound for the worst possible score. (In related research $10^6$ is taken.)

D. Varying Nestedness

It is known that a Level-$k$ NMC search for a smaller value of $k$ tends to saturate earlier for a larger number of node expansions than a Level-$(k+1)$ search, so that in order to find optimal solutions in bigger problem instances, larger values $k$ are often more effective. Thus, we varied $k$ and adapted the number of iterations $t$ for learning, so that between 100 million and one billion rollouts are performed for an entire exploration. As the number of rollouts is fixed ($t^k$), finding an appropriate value $t$ for a given value $k$ and tree size is immediate. For example, we choose $(k,t) = (5,50)$ with a total of 312,500,000 rollouts (used by [6], [21]), $(k,t) = (8,12)$ with

double rollout() {
    visited = 0;
    tourSize = 1;
    int n = 0;
    int u = 0;
    double makespan = 0;
    int violations = 0;
    double cost = 0;
    while(tourSize < N) {
        delete r;
        policy = clone[level];
        return best;
    }
    if (!visited[i]) {
        int j=1;
        while (j < N) {
            if (j != i) {
                if (!visited[j]) {
                    if (l[i] > r[j] || makespan + d[n][i] > r[j])
                        break;
                    j++;
                }
            }
            if (j==N)
                moves[succs++] = i;
            if (!succs)
                for(int i = 1; i < N; i++)
                    moves[succs++] = i;
            if (!succs)
                for(int i = 1; i < N; i++)
                    moves[succs++] = i;
            if (!visited[i]) {
                moves[succs++] = i;
            }
            for(int i = 0; i < succs; i++)
                sum += value[i] = EXP(policy[n][succ[i]])
                double m = rand(0,..,sum);
                int i=0;
                sum = value[0];
                while(sum<m)
                    sum += value[++];
            u = n;
            n = succ[i];
            tour[tourSize++] = n;
            visited[n] = true;
            cost += d[u][n];
            makespan = max(makespan + d[u][n], l[n]);
            if (makespan > r[n])
                violations++;
        }
        tour[tourSize++] = 0;
        cost += d[n][0];
        makespan = max(makespan + d[n][0], l[0]);
        if (makespan > r[0])
            violations++;
        return offset * violations + cost;
    }
}

Fig. 3. Rollout with score evaluation at search tree leaf.
The policy is stored in form of a \( z \) the denominator to a cost-improving tour as shown in the algorithm. First, \( P \) the policy.

Influence of the (chosen node / successor node) pair updates function, so we chose known approximation for it (see Fig. 6).

Adapting Knowledge

search depth.

If (succs > b) {
    for(int i = 0; i < b; i++)
        swap(succ[i],moves[rand() % succs]);
    succs = b;
}

Fig. 4. Implementing beam search.

void adapt(int tour[], int level) {
    visited = (true, false, ..., false);
    int succs = 0;
    for(int p = 0; p < N; p++)
        int n = 0;
    int succs = 0;
    for(int i = 0; i < N; i++)
        if (!visited[i])
            moves[succs++] = i;
    clone[level][n][tour[p]] += 1.0;
    double z = 0.0;
    for(int i = 0; i < succs; i++)
        z += exp(policy[n][succ[i]]) / z;
    n = tour[p];
    visited[n] = true;
}

Fig. 5. Policy adaption.

a total of 429981696 rollouts and \((k,t)=(10,7)\) with a total of 282475249 rollouts.

E. Employing Beam Search

We also experimented with Monte-Carlo beam search, as this was effective in many single-agent search domains [4]. Morpion Solitaire this enhancement helped to match the record score of 82 moves. There is, of course, a trade-off between depth and width. It is often the case that a smaller set of successors already yields to good solutions and that early failures do not harm. The smaller the number of successors the faster the rollout. Our implementation of beam search (see Fig. 4) is a simplification of Monte-Carlo beam search as recently proposed in [4]. For the experiments, we chose a beam width \( b \) of \( N/2 \) so that at the root node half of the successors are neglected from the search. During the rollout the relative size of the set of successors increases with the search depth.

F. Adapting Knowledge

The NMC algorithm with Policy Adaption (see Fig. 5) usually is invoked with \( P_0(u,v) = 0 \) for all \( u,v \in \{0, \ldots, N-1\} \). The policy is stored in form of a \( (N \times N) \)-sized array and is updated with the edge probabilities \( P(u,v) = P_1(u,v) \) wrt. to a cost-improving tour as shown in the algorithm. First, the denominator \( z \) for normalization is computed. Then, the influence of the (chosen node / successor node) pair updates the policy.

We found that much of the time is spent in evaluating the \( e \)-function, so we chose known approximation for it (see Fig. 6).

G. Elicitation of Knowledge

For a probabilistic prior policy in form of an initial seed we aim at the simple strategy of including city-to-city distances. This matches the idea of reordering in depth-first branch-and-bound solvers for the problem. As we want to direct the search towards successors with small distances, given that the \( e \)-function is applied to the policy values, we take the negative of the distance value for the policy.

In order to adjust the amplitude of these numbers, and contributing to the fact that we can exclude some edges (e.g., \((u,v)\)), or \((u,v)\) with \( l_u + d_{u,v} > r_v \) by setting their distances to infinity, we divide each value by the smallest value in one column (equivalently row) of the distance matrix. More formally, let \( c_{u,v} = \min_{v=0}^{N-1}(d_{u,v}) \) be the column minima for \( u \in \{0, \ldots, N-1\} \). Then, we define the initial policy by \( P_0(u,v) = -d_{u,v}/c_u \) for \( u,v = \{0, \ldots, N-1\} \).

There are other forms of knowledge available. For example, after applying the Assignment Problem heuristic (e.g., with the cubic time Hungarian algorithm), the distance matrix is reduced to the solution of one minimal assignment. Even though for seeding the policy this lower bound has to be computed only once, we took an engineered version of its computation documented by [9].

IV. EXPERIMENTS

We executed our experiments on one core of an Intel (R) Core (TM) i7 CPU PC at 2.668 MHz that is equipped with 8192 MB cache and 8 GB RAM running Ubuntu Linux 11.10. We used the GNU c-compiler g++ (version 4.3.3), and all program compilations were optimized with -O3. For easy referencing we call our approximate TSPTW solver mTSP.

A. Dumas Benchmark

Table I shows that mTSP always finds an optimal tour for simpler benchmark instances with \( N = 20 \). However, since the algorithm does not stop automatically, no proof certificate for optimality is derived. As expected, mTSP is much faster than the two provably optimal algorithms suggested by [9]. Since it is an anytime algorithm, the running time is set to a predefined threshold to terminate the search. We used mTSP also in larger Dumas’ benchmarks with a timeout of 15 minutes (see Table II). For \( N = 40 \) all but 3 problems (total deviation from optimum \( 4 + 4 + 9 = 17 \) were solved with the state-of-the-art scores. For the \( N = 60 \) problems all but 9 problems (total deviation \( 4 + 4 + 1 + 7 + 14 + 10 + 7 + 29 + 5 = 71 \) were solved with the state-of-the-art scores.
Table I

Results in smaller instances of the Dumas TSPTW Benchmark (Expanded nodes E \( \times \) Initiated Rollouts \( R \) and CPU times \( T \) are shown. Index \( B \) and \( c \) refer to a depth-first branch-and-bound solver with two different admissible heuristics, while \( m \) refers to mTSP. Cost shows the best known solutions).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cost</th>
<th>( E_B )</th>
<th>( E_c )</th>
<th>( T_B )</th>
<th>( T_c )</th>
<th>( R_B )</th>
<th>( R_c )</th>
<th>( T_B )</th>
<th>( T_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>n20w20.001</td>
<td>378</td>
<td>39</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w20.002</td>
<td>286</td>
<td>97</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w20.003</td>
<td>394</td>
<td>138</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w20.004</td>
<td>396</td>
<td>156</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w20.005</td>
<td>352</td>
<td>41</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w40.001</td>
<td>254</td>
<td>38</td>
<td>2</td>
<td>3.65</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w40.002</td>
<td>333</td>
<td>88</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w40.003</td>
<td>317</td>
<td>140</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w40.004</td>
<td>388</td>
<td>76</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n20w40.005</td>
<td>288</td>
<td>10</td>
<td>2</td>
<td>3.65</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n60w60.001</td>
<td>335</td>
<td>40</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n60w60.002</td>
<td>244</td>
<td>97</td>
<td>1</td>
<td>1.8</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n60w60.003</td>
<td>352</td>
<td>396</td>
<td>2</td>
<td>3.65</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n60w60.004</td>
<td>280</td>
<td>4055</td>
<td>258</td>
<td>1.25</td>
<td>1.25</td>
<td>0.75</td>
<td>1.25</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>n60w60.005</td>
<td>338</td>
<td>105</td>
<td>10</td>
<td>10049</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td>n80w80.001</td>
<td>329</td>
<td>31692</td>
<td>35</td>
<td>28865</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td>n80w80.002</td>
<td>338</td>
<td>260</td>
<td>36</td>
<td>16680</td>
<td>50000</td>
<td>12500</td>
<td>50000</td>
<td>12500</td>
<td></td>
</tr>
<tr>
<td>n80w80.003</td>
<td>329</td>
<td>15959</td>
<td>3</td>
<td>208526</td>
<td>420</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td>n80w80.004</td>
<td>304</td>
<td>12588</td>
<td>80</td>
<td>37307</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td></td>
</tr>
<tr>
<td>n80w80.005</td>
<td>264</td>
<td>52244</td>
<td>438</td>
<td>16606</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td>n100w100.001</td>
<td>237</td>
<td>16351</td>
<td>52</td>
<td>123296</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td></td>
</tr>
<tr>
<td>n100w100.002</td>
<td>222</td>
<td>68954</td>
<td>7</td>
<td>22031</td>
<td>62500</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td>n100w100.003</td>
<td>310</td>
<td>133802</td>
<td>765</td>
<td>25867</td>
<td>62500</td>
<td>20736</td>
<td>12500</td>
<td>20736</td>
<td></td>
</tr>
<tr>
<td>n100w100.004</td>
<td>349</td>
<td>3429</td>
<td>2</td>
<td>12135</td>
<td>41728</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td></td>
</tr>
<tr>
<td>n100w100.005</td>
<td>258</td>
<td>68888</td>
<td>44</td>
<td>230871</td>
<td>82944</td>
<td>12500</td>
<td>20736</td>
<td>12500</td>
<td></td>
</tr>
</tbody>
</table>

C. Solomon-Peasant Benchmark

In Table IV we consider the benchmark by Solomon and Peasant, for which we have not found a recent publication to compare with. We provide solution cost results, the number of rollouts performed, and the CPU time of a Level-8 search at or close to the moment, when the best solution is found or when it hits the time threshold of 15m. In contrast to the Solomon-Potvin-Bengio benchmark, we document results of one straight run of the algorithm, which solves all but 4 of the 27 problem instances with the state-of-the-art cost value.

D. AFG Benchmark

In Table V we show the AFG benchmark results. We provide explorations results of a Level-8 search at (or close to) the moment, when the best solution was found, or when it hit the time threshold of 15m. Again we document results of one straight run of the algorithm, which solves all of the simpler problems (\( N < 40 \)) with the state-of-the-art scores, but not the harder problems in the set (results for \( N > 100 \) are skipped).

E. Langevin Benchmark

The Langevin benchmark is a larger set of problems that appears to be simpler for mTSP. As documented in Table VI the entire set of \( N = 20 \) problems is solved matching the state-of-the-art cost in 1s, the entire set of \( N = 40 \) benchmark problems is presumably optimally solved in less than 10s, and in one single 15m runs only three problems are not resulting in the state-of-the-art, cost leaving only a small margin to improve.
V. Conclusion and Outlook

Nested Monte-Carlo (NMC) search is a more recent randomized single-agent state space search technique which has proven to quickly find good solutions to a growing number of combinatorial problems with huge state spaces and large branching factors.

We have seen that knowledge and algorithm engineering greatly improve NMC search for solving the TSPTW problems. Algorithm engineering for the existing code leads to an improvement of more than factor 10 in the exploration efficiency, whereas knowledge engineering is included to seed an improvement of more than factor 10 in the exploration of combinatorial problems with huge state spaces and large branching factors.

Our goal is to improve our application scenario in logistics [9] which is related to the well-known vehicle routing problem (VRP) [1]. We implement a multiagent system, where the general problem is split into smaller problems which agents solve locally concurrent within short time windows to optimize the behavior of the overall system. Here, the agents solve individual TSP problems, and trade their found solutions for improving the overall costs. In order to implement sound planning and control processes in the logistic domain, we import transport infrastructures from OpenStreetMap (see: www.openstreetmap.org) (OSM) databases. The TSP are generated by shortest path reduction of a map wrt. to pickup and delivery locations of customers as well as the depot(s), and scaled with the vehicle fleet of the distributor. Besides time window constraints, we are confronted with a variety of additional side-constraints: limited driving times and requested breaks for the drivers, premium contracts, pickup and backhaul tasks, just to name a few. The multiagent-based simulation platform PlaSMA (see http://plasma.informatik.uni-bremen.de), enables the simulation of real world scenarios with orders provided by our industrial partner. Firstly, we cluster all orders of the day statistically for each cluster. Next, we solve the TSP with the solver described in this paper. For solving the TSP of each cluster a lower bound is computed e.g., with the Hungarian algorithm. As result, we solve different TSPs based on a real transport infrastructure and simulate real properties of orders. Through negotiation processes and agent interaction, the TSP solver enables reliable planning and control logistic processes in dynamic environments.
# TABLE V
RESULTS IN AFG TSPTW BENCHMARK

<table>
<thead>
<tr>
<th>Problem</th>
<th>N</th>
<th>Cost</th>
<th>$C_m$</th>
<th>$E_m$</th>
<th>$T_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>rgb010a</td>
<td>11</td>
<td>671</td>
<td>671</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb016a</td>
<td>17</td>
<td>938</td>
<td>939</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb016b</td>
<td>17</td>
<td>1304</td>
<td>1304</td>
<td>100000</td>
<td>&lt; 5s</td>
</tr>
<tr>
<td>rgb017.2</td>
<td>18</td>
<td>852</td>
<td>852</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb017</td>
<td>16</td>
<td>893</td>
<td>893</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb017a</td>
<td>18</td>
<td>4296</td>
<td>4296</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb019a</td>
<td>20</td>
<td>1262</td>
<td>1262</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb019b</td>
<td>20</td>
<td>1866</td>
<td>1866</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb018c</td>
<td>20</td>
<td>4536</td>
<td>4536</td>
<td>2566080</td>
<td>15m</td>
</tr>
<tr>
<td>rgb019d</td>
<td>20</td>
<td>1356</td>
<td>1356</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb020a</td>
<td>21</td>
<td>4689</td>
<td>4689</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb021.2</td>
<td>22</td>
<td>4528</td>
<td>4528</td>
<td>100000</td>
<td>&lt; 5s</td>
</tr>
<tr>
<td>rgb021.3</td>
<td>22</td>
<td>4528</td>
<td>4528</td>
<td>150000</td>
<td>&lt; 5s</td>
</tr>
<tr>
<td>rgb021.4</td>
<td>22</td>
<td>4525</td>
<td>4525</td>
<td>200000</td>
<td>&lt; 10s</td>
</tr>
<tr>
<td>rgb021.5</td>
<td>22</td>
<td>4515</td>
<td>4515</td>
<td>200000</td>
<td>&lt; 1m</td>
</tr>
<tr>
<td>rgb021.6</td>
<td>22</td>
<td>4480</td>
<td>4480</td>
<td>150000</td>
<td>&lt; 5s</td>
</tr>
<tr>
<td>rgb021.7</td>
<td>22</td>
<td>4479</td>
<td>4479</td>
<td>100000</td>
<td>&lt; 5s</td>
</tr>
<tr>
<td>rgb021.8</td>
<td>22</td>
<td>4478</td>
<td>4478</td>
<td>150000</td>
<td>&lt; 5s</td>
</tr>
<tr>
<td>rgb021.9</td>
<td>22</td>
<td>4478</td>
<td>4478</td>
<td>200000</td>
<td>&lt; 10s</td>
</tr>
<tr>
<td>rgb021</td>
<td>22</td>
<td>4536</td>
<td>4536</td>
<td>50000</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>rgb027a</td>
<td>28</td>
<td>5001</td>
<td>5001</td>
<td>150000</td>
<td>&lt; 30s</td>
</tr>
<tr>
<td>rgb031a</td>
<td>32</td>
<td>1863</td>
<td>1863</td>
<td>200000</td>
<td>&lt; 10s</td>
</tr>
<tr>
<td>rgb033a</td>
<td>34</td>
<td>2069</td>
<td>2069</td>
<td>400000</td>
<td>&lt; 20s</td>
</tr>
<tr>
<td>rgb034a</td>
<td>35</td>
<td>2222</td>
<td>2222</td>
<td>500000</td>
<td>&lt; 30s</td>
</tr>
<tr>
<td>rgb035a.2</td>
<td>36</td>
<td>2056</td>
<td>2056</td>
<td>7558272</td>
<td>15m</td>
</tr>
<tr>
<td>rgb035a</td>
<td>36</td>
<td>2144</td>
<td>2144</td>
<td>1000000</td>
<td>15m</td>
</tr>
<tr>
<td>rgb038a</td>
<td>39</td>
<td>2480</td>
<td>2480</td>
<td>150000</td>
<td>&lt; 30s</td>
</tr>
<tr>
<td>rgb040a</td>
<td>41</td>
<td>2378</td>
<td>2378</td>
<td>18242496</td>
<td>15m</td>
</tr>
<tr>
<td>rgb041a</td>
<td>42</td>
<td>2598</td>
<td>2598</td>
<td>16842816</td>
<td>15m</td>
</tr>
<tr>
<td>rgb042a</td>
<td>43</td>
<td>2772</td>
<td>2772</td>
<td>15723072</td>
<td>15m</td>
</tr>
<tr>
<td>rgb048a</td>
<td>49</td>
<td>9383</td>
<td>9383</td>
<td>8398080</td>
<td>15m</td>
</tr>
<tr>
<td>rgb049a</td>
<td>50</td>
<td>10018</td>
<td>10018</td>
<td>9097020</td>
<td>15m</td>
</tr>
<tr>
<td>rgb050a</td>
<td>51</td>
<td>10732</td>
<td>10732</td>
<td>1000000</td>
<td>15m</td>
</tr>
<tr>
<td>rgb050b</td>
<td>51</td>
<td>9863</td>
<td>9863</td>
<td>7247272</td>
<td>15m</td>
</tr>
<tr>
<td>rgb050c</td>
<td>51</td>
<td>10024</td>
<td>10024</td>
<td>6994000</td>
<td>15m</td>
</tr>
<tr>
<td>rgb055a</td>
<td>56</td>
<td>3761</td>
<td>3761</td>
<td>100000</td>
<td>&lt; 10s</td>
</tr>
<tr>
<td>rgb067a</td>
<td>68</td>
<td>4625</td>
<td>4625</td>
<td>4000000</td>
<td>&lt; 5m</td>
</tr>
<tr>
<td>rgb068a</td>
<td>87</td>
<td>8400</td>
<td>8418</td>
<td>3405888</td>
<td>15m</td>
</tr>
<tr>
<td>rgb092a</td>
<td>93</td>
<td>7160</td>
<td>7197</td>
<td>2846016</td>
<td>15m</td>
</tr>
<tr>
<td>rgb125a</td>
<td>125</td>
<td>7936</td>
<td>8012</td>
<td>1772928</td>
<td>15m</td>
</tr>
<tr>
<td>rgb132.2</td>
<td>133</td>
<td>8200</td>
<td>8489</td>
<td>1259712</td>
<td>15m</td>
</tr>
<tr>
<td>rgb132</td>
<td>133</td>
<td>8470</td>
<td>8668</td>
<td>1259712</td>
<td>15m</td>
</tr>
<tr>
<td>rgb152.3*</td>
<td>153</td>
<td>9797</td>
<td>10571</td>
<td>793152</td>
<td>15m</td>
</tr>
<tr>
<td>rgb152*</td>
<td>153</td>
<td>10032</td>
<td>10229</td>
<td>793152</td>
<td>15m</td>
</tr>
</tbody>
</table>
REFERENCES


