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 TSPTW 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 backhault 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 Adaptation, 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.

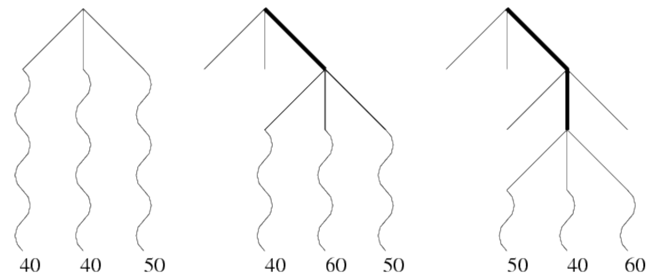

Fig. 1. This figure illustrates three successive decision steps of a level 1 search for a maximization problem. Numbers at the bottom of branches represent the rewards. Level 1 exploration is represented in thin lines. Monte-Carlo playouts (i.e. level 0) are shown with wavy lines and decisions are represented with bold lines.

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 $\alpha\beta$ pruning, e.g., in Go [12] and General Game Playing [10], [17].

B. Nested Monte-Carlo Search

For single-agent search challenges, Nested Monte-Carlo (NMC) has been suggested and successively been 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 $P_l(u, v)$ is updated. Moreover, in the first iteration $P_l(u, v)$ is initialized with $P_{l-1}(u, v)$.

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 going 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 large 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) {
    double sum = 0;
    int succs = 0;
    for(int i = 1; i < N; i++)
      if (!visited[i])
        if(makespan + d[n][i] > r[i]) {
          moves[0] = i;
          succs = 1;
          break;
        }
    if (!succs)
      for(int i = 1; i < N; i++)
        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++)
        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[++i];
    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.

```
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;
  int n = 0;
  for(int p=0; p<N; p++)  {
    succs = 0;
    for(int i = 1; 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]]);
    for (int i=0; i<succs; i++)
      clone[level][n][succ[i]] -=
         exp(policy[n][succ[i]])/z;
    n = tour[p];
    visited[n] = true;
  }
}
```

Fig. 5. Policy adaption.

a total of $429\,981\,696$ rollouts and $(k,t) = (10,7)$ with a total of $282\,475\,249$ 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 $P(u,v) = P_l(u,v)$ 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).

```
static union{double d; struct{int j, i;} n;} e;
#define A (1048576/M_LN2)
#define C 60801
#define exp(y) (e.n.i=A*(y)+(1072693248-C),e.d)
```

Fig. 6. Approximating the e-function.

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,u), 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 $c_u = \min_{v=0}^{N-1}\{d_{u,v}\}$ 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 8 192 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 / INITIATED ROLLOUTS R AND CPU TIMES T ARE SHOWN, INDEX h 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).

Problem	Cost	E_h	T_h	E_c	T_c	R_m	T_m
n20w20.001	378	49	$< 1s$	2 784 766	$< 1s$	20 736	$< 1s$
n20w20.002	286	97	$< 1s$	3 234 936	$< 1s$	125 000	$< 2s$
n20w20.003	394	138	$< 1s$	4 944 477	$< 1s$	125 000	$< 2s$
n20w20.004	396	156	$< 1s$	2 331 312	$< 1s$	20 736	$< 1s$
n20w20.005	352	41	$< 1s$	4 017 260	$< 1s$	125 000	$< 2s$
n20w40.001	254	38 022	3s	103 087 541	18s	125 000	$< 2s$
n20w40.002	333	88	$< 1s$	11 388 523	2s	20 736	$< 1s$
n20w40.003	317	1 409	$< 1s$	21 158 796	3s	125 000	$< 2s$
n20w40.004	388	7 676	1s	35 117 607	6s	20 736	$< 1s$
n20w40.005	288	10 287	2s	20 801 644	3s	20 738	$< 1s$
n20w60.001	335	40 810	14s	223 904 879	43s	41 472	$< 1s$
n20w60.002	244	97 144	7s	81 367 918	15s	20 738	$< 1s$
n20w60.003	352	399 127	27s	31 292 739	5s	125 000	$< 10s$
n20w60.004	280	4 055 453	258s	1 245 195 466	238s	2 509 056	$< 10s$
n20w60.005	338	105 393	10s	104 049 862	18s	2 757 888	$< 10s$
n20w80.001	329	316 992	35s	288 653 549	56s	125 000	$< 2s$
n20w80.002	338	260 552	36s	166 880 630	33s	500 000	$< 10s$
n20w80.003	320	15 959	3s	208 526 467	42s	20 736	$< 1s$
n20w80.004	304	1 258 898	80s	373 077 547	73s	20 736	$< 1s$
n20w80.005	264	5 224 435	438s	1 660 621 704	332s	125 000	$< 30s$
n20w100.001	237	1 635 101	52s	1 232 596 799	279s	20 736	$< 1s$
n20w100.002	222	68 954	7s	2 203 174 867	531s	625 000	$< 30s$
n20w100.003	310	13 382 035	765s	2 586 795 810	538s	20 736	$< 1s$
n20w100.004	349	34 289	2s	1 213 551 958	266s	41 472	$< 1s$
n20w100.005	258	688 887	44s	2 308 713 055	548s	82 944	$< 1s$

TABLE II

RESULTS IN LARGER INSTANCES OF THE DUMAS TSPTW BENCHMARK (COSTS OF THE OBTAINED SOLUTIONS C_m AND CPU TIMES T_m ARE SHOWN, COST REFER TO THE BEST KNOWN SOLUTIONS).

Problem	Cost	C_m	T_m	Problem	Cost	C_m	T_m
n40w20.001	500	500	$< 15m$	n60w20.001	551	551	$< 15m$
n40w20.002	552	552	$< 15m$	n60w20.002	605	605	$< 15m$
n40w20.003	478	478	$< 15m$	n60w20.003	533	533	$< 15m$
n40w20.004	404	404	$< 15m$	n60w20.004	616	616	$< 15m$
n40w20.005	499	499	$< 15m$	n60w20.005	603	603	$< 15m$
n40w40.001	465	465	$< 15m$	n60w40.001*	591	595	$= 15m$
n40w40.002	461	461	$< 15m$	n60w40.002	621	621	$< 15m$
n40w40.003	474	474	$< 15m$	n60w40.003	603	603	$< 15m$
n40w40.004	452	452	$< 15m$	n60w40.004	597	597	$< 15m$
n40w40.005	453	453	$< 15m$	n60w40.005	539	539	$< 15m$
n40w60.001	494	494	$< 15m$	n60w60.001	609	609	$< 15m$
n40w60.002	470	470	$< 15m$	n60w60.002*	566	571	$= 15m$
n40w60.003*	408	412	$= 15m$	n60w60.003*	485	486	$= 15m$
n40w60.004	382	382	$< 15m$	n60w60.004	571	571	$< 15m$
n40w60.005	328	328	$< 15m$	n60w60.005*	569	576	$= 15m$
n40w80.001	395	395	$< 15m$	n60w80.001	458	458	$< 15m$
n40w80.002*	431	435	$= 15m$	n60w80.002	498	498	$< 15m$
n40w80.003	412	412	$< 15m$	n60w80.003	550	550	$< 15m$
n40w80.004	417	417	$< 15m$	n60w80.004*	566	580	$= 15m$
n40w80.005	344	344	$< 15m$	n60w80.005*	468	478	$= 15m$
n40w100.001*	429	438	=15m	n60w100.001	515	515	$< 15m$
n40w100.002	358	358	$< 15m$	n60w100.002*	538	545	$= 15m$
n40w100.003	364	364	$< 15m$	n60w100.003*	560	589	$= 15m$
n40w100.004	357	357	$< 15m$	n60w100.004	510	510	$< 15m$
n40w100.005	377	377	$< 15m$	n60w100.005*	451	556	$= 15m$

B. Solomon-Potwin-Bengio Benchmark

Table III shows the solution qualities of our implementation (C_4) wrt. the results of [21] (C_1) and [6] (C_2 and C_3). The solution qualities align with the best-known values (e.g., reported at http://iridia.ulb.ac.be/~manuel/files/TSPTW/SolomonPotvinBengio.best). With our algorithm only one state-of-the-art solution (marked with an asterisk) has not been found in the allocated time period. We also give approximate timing information and the level l (in brackets) to which the search was initialized. Aligning with the precursor work, every experiment has been repeated 2-4 times in case no state-of-the-art solution has been established.

To measure the savings for algorithm engineering we compared the number of evaluations in a complete (level 5 – 10-iteration) search (beam search option disabled) for the largest problem in the benchmark suite (rc204.1). The original implementation took 2m31s, while the refined implementation required only 11s. This documents that we have improved the mere running time of the source by for more than one order of magnitude!

We conducted one extra experiment (the only one for which extra information was included into the system). Seeding expert knowledge on the sequence of the first 5 or 10 cities visited by the salesman, helped solving the remaining problem ($< 1m$ for a prefix of 10 cities and $< 5m$ for a prefix of 5 cities). For this case, we simply set $P_0(u, v) = 100$ if v is a successor of u in the optimal solution. More complex non-automated hints provided by domain experts are possible.

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-Potwin-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 $N > 160$ 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.

TABLE III

RESULTS IN SOLOMON-POTWIN-BENGIO TSPTW BENCHMARK (COST REFERS TO THE BEST KNOWN SOLUTION, C_1 IS THE COST COMPUTED BY NMC+DOMAIN-DEPENDENT PRUNING RULES, C_2 THE COST COMPUTED BY NPRA, AND C_3 THE COST COMPUTED BY NPRA+DOMAIN-DEPENDENT EXTENSIONS; C_m (T_m) THE COST (CPU TIME) COMPUTED BY MTSP).

Problem	N	Cost	C_1	C_2	C_3	C_m	T_m
rc206.1	4	117.85	117.85	117.85	117.85	117.85 (5)	$< 1m$
rc207.4	6	119.64	119.64	119.64	119.64	119.64 (5)	$< 1m$
rc202.2	14	304.14	304.14	304.14	304.14	304.14 (5)	$< 1m$
rc205.1	14	343.21	343.21	343.21	343.21	343.21 (5)	$< 1m$
rc203.4	15	314.29	314.29	314.29	314.29	314.29 (5)	$< 1m$
rc203.1	19	453.48	453.48	453.48	453.48	453.48 (5)	$< 1m$
rc201.1	20	444.54	444.54	444.54	444.54	444.54 (5)	$< 1m$
rc204.3	24	455.03	455.03	455.03	455.03	455.03 (5)	$< 1m$
rc206.3	25	574.42	574.42	574.42	574.42	574.42 (5)	$< 1m$
rc201.2	26	711.54	711.54	711.54	711.54	711.54 (5)	$< 30m$
rc201.4	26	793.64	793.64	793.64	793.64	793.64 (5)	$< 30m$
rc205.2	27	755.93	755.93	755.93	755.93	755.93 (5)	$< 30m$
rc202.4	28	793.03	793.03	800.18	793.03	793.03 (5)	$< 30m$
rc205.4	28	760.47	760.47	765.38	760.47	760.47 (5)	$< 2h$
rc202.3	29	837.72	837.72	839.58	839.58	837.72 (5)	$< 2h$
rc208.2	29	533.78	536.04	537.74	533.78	533.78 (5)	$< 2h$
rc207.2	31	701.25	707.74	702.17	701.25	701.25 (8)	$< 2h$
rc201.3	32	790.61	790.61	796.98	790.61	790.61 (5)	$< 30m$
rc204.2	33	662.16	675.33	673.89	664.38	662.16 (8)	$< 30m$
rc202.1	33	771.78	776.47	775.59	772.18	771.78 (8)	$< 30m$
rc203.2	33	784.16	784.16	784.16	784.16	784.16 (5)	$< 30m$
rc207.3	33	682.40	687.58	688.50	682.40	682.40 (5)	$< 2h$
rc207.1	34	732.68	743.29	743.72	738.74	732.68 (5)	$< 2h$
rc205.3	35	825.06	828.27	828.36	825.06	825.06 (5)	$< 30m$
rc208.3	36	634.44	641.17	656.40	650.49	634.44 (8)	$< 2h$
rc203.3	37	817.53	837.72	820.93	817.53	817.53 (5)	$< 2h$
rc206.2	37	828.06	839.18	829.07	828.06	828.06 (8)	$< 2h$
rc206.4	38	831.67	859.07	831.72	831.67	831.67 (5)	$< 30m$
rc208.1*	38	789.25	797.89	799.24	793.60	793.60 (8)	$< 2h$
rc204.1	46	868.76	899.79	883.85	880.89	878.64 (8)	$< 2h$

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 the policy for finding good solutions early. The results are promising and a challenge to state-of-the-art TSPTW solvers, including evolutionary algorithms and ant colony optimization. We have matched most, but not improved any best-known solution from the repository (which we assume are all optimal).

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

TABLE IV

RESULTS IN SOLOMON-PESANT TSPTW BENCHMARK (COST IS THE BEST KNOWN SOLUTION; C_m, E_m AND T_m ARE THE PERFORMANCE INDICATORS COMPUTED BY MTSP).

Problem	N	Cost	C_m	E_m	T_m
rc201.0	26	628.62	628.62	40 000	$< 1s$
rc201.1	29	654.70	654.70	50 000	$< 1s$
rc201.2	29	707.65	707.65	10 000	$< 1s$
rc201.3	20	422.54	422.54	10 000	$< 1s$
rc202.0	26	496.22	496.22	120 000	$< 1s$
rc202.1	23	426.53	426.53	60 000	$< 1s$
rc202.2	28	611.77	611.77	20 000	$< 1s$
rc202.3	27	627.85	627.85	130 000	$< 2s$
rc203.0	36	727.45	727.45	3 000 000	$< 3m$
rc203.1	38	726.99	726.99	1 500 000	$< 3m$
rc203.2	29	617.46	617.46	20 000	$< 20s$
rc204.0	33	541.45	541.45	11 100 000	$< 10m$
rc204.1	29	485.37	485.37	20 000 000	$< 14m$
rc204.2	41	778.40	778.40	4 150 000	$< 6m$
rc205.0	27	511.65	511.65	80 000	$< 5s$
rc205.1	23	491.22	491.22	20 000	$< 1s$
rc205.2*	29	714.69	717.56	35 310 000	$= 15m$
rc205.3	25	601.24	601.24	90 000	$< 5s$
rc206.0	36	835.23	835.23	1 500 000	$< 1m$
rc206.1	34	664.73	664.73	1 140 000	$< 40s$
rc206.2*	33	655.37	670.80	33 110 000	$= 15m$
rc207.0	38	806.69	806.69	6 000 000	$< 4m$
rc207.1	34	726.36	726.36	20 000 000	$< 11m$
rc207.2	31	546.41	546.41	3 600 000	$< 2m$
rc208.0*	45	820.56	884.96	14 220 000	$= 15m$
rc208.1*	28	509.04	558.48	35 210 000	$= 15m$
rc208.2	30	503.92	503.92	7 700 000	$< 4m$

planing 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 costumers 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.,w̃ith 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

Problem	N	Cost	C_m	E_m	T_m
rbg010a	11	671	671	50 000	$< 1s$
rbg016a	17	938	939	50 000	$< 1s$
rbg016b	17	1304	1304	100 000	$< 5s$
rbg017.2	18	852	852	50 000	$< 1s$
rbg017	16	893	893	50 000	$< 1s$
rbg017a	18	4296	4296	50 000	$< 1s$
rbg019a	20	1262	1262	50 000	$< 1s$
rbg019b	20	1866	1866	50 000	$< 1s$
rbg019c	20	4536	4545	2 566 080	$= 15m$
rbg019d	20	1356	1356	50 000	$< 1s$
rbg020a	21	4689	4689	50 000	$< 1s$
rbg021.2	22	4528	4528	100 000	$< 5s$
rbg021.3	22	4528	4528	150 000	$< 5s$
rbg021.4	22	4525	4525	200 000	$< 10s$
rbg021.5	22	4515	4515	2 000 000	$< 1m$
rbg021.6	22	4480	4480	150 000	$< 5s$
rbg021.7	22	4479	4479	100 000	$< 5s$
rbg021.8	22	4478	4478	150 000	$< 5s$
rbg021.9	22	4478	4478	200 000	$< 10s$
rbg021	22	4536	4536	50 000	$< 1s$
rbg027a	28	5091	5051	1 000 000	$< 30s$
rbg031a	32	1863	1863	200 000	$< 10s$
rbg033a	34	2069	2069	400 000	$< 20s$
rbg034a	35	2222	2222	500 000	$< 30s$
rbg035a.2	36	2056	2056	7 558 272	$< 15m$
rbg035a	36	2144	2144	1 000 000	$< 30s$
rbg038a	39	2480	2480	150 000	$< 30s$
rbg040a*	41	2378	2413	18 242 496	$= 15m$
rbg041a*	42	2598	2625	16 842 816	$= 15m$
rbg042a*	43	2772	2805	15 723 072	$= 15m$
rbg048a*	49	9383	9480	8 398 080	$= 15m$
rbg049a*	50	10018	10075	9 097 020	$= 15m$
rbg050a*	51	2953	2974	6 625 152	$= 15m$
rbg050b*	51	9863	9921	8 724 672	$= 15m$
rbg050c*	51	10024	10088	6 998 400	$= 15m$
rbg055a	56	3761	3761	10 000	$< 10s$
rbg067a	68	4625	4625	4 000 000	$< 5m$
rbg086a*	87	8400	8418	3 405 888	$= 15m$
rbg092a*	93	7160	7197	2 846 016	$= 15m$
rbg125a*	125	7936	8012	1 772 928	$= 15m$
rbg132.2*	133	8200	8489	1 259 712	$= 15m$
rbg132*	133	8470	8668	1 259 712	$= 15m$
rbg152.3*	153	9797	10571	793 152	$= 15m$
rbg152*	153	10032	10229	793 152	$= 15m$

TABLE VI
RESULTS IN LANGEVIN TSPTW BENCHMARK.

Problem	N	Cost	C_m	E_m	T_m
N20ft301	20	661.60	661.60	46 656	$< 1s$
N20ft302	20	684.20	684.20	46 656	$< 1s$
N20ft303	20	746.40	746.40	46 656	$< 1s$
N20ft304	20	817.00	817.00	46 656	$< 1s$
N20ft305	20	716.50	716.50	46 656	$< 1s$
N20ft306	20	727.80	727.80	46 656	$< 1s$
N20ft307	20	691.80	691.80	46 656	$< 1s$
N20ft308	20	788.20	788.20	46 656	$< 1s$
N20ft309	20	730.70	730.70	46 656	$< 1s$
N20ft310	20	683.00	683.00	46 656	$< 1s$
N20ft401	20	660.80	660.80	46 656	$< 1s$
N20ft402	20	684.20	684.20	46 656	$< 1s$
N20ft403	20	746.40	746.40	46 656	$< 1s$
N20ft404	20	817.00	817.00	46 656	$< 1s$
N20ft405	20	716.50	716.50	46 656	$< 1s$
N20ft406	20	727.80	727.80	46 656	$< 1s$
N20ft407	20	691.80	691.80	46 656	$< 1s$
N20ft408	20	757.30	757.30	46 656	$< 1s$
N20ft409	20	730.70	730.70	46 656	$< 1s$
N20ft410	20	683.00	683.00	46 656	$< 1s$
N40ft201	40	1100.60	1100.60	46 656	$< 1m$
N40ft202	40	1010.40	1010.40	46 656	$< 1m$
N40ft203	40	876.80	876.80	93 312	$< 1m$
N40ft204	40	885.80	885.80	46 656	$< 1m$
N40ft205	40	940.90	940.90	46 656	$< 1m$
N40ft206	40	1054.20	1054.20	46 656	$< 1m$
N40ft207	40	867.50	867.50	233 280	$< 1m$
N40ft208	40	1050.70	1050.70	46 656	$< 1m$
N40ft209	40	1013.90	1013.90	46 656	$< 1m$
N40ft210	40	1026.30	1026.30	46 656	$< 1m$
N40ft401	40	1085.00	1085.00	46 656	$< 1m$
N40ft402	40	995.60	995.60	46 656	$< 1m$
N40ft403	40	845.80	845.80	46 656	$< 1m$
N40ft404	40	868.00	868.00	139 968	$< 1m$
N40ft405	40	936.50	936.50	139 968	$< 1m$
N40ft406	40	969.10	969.10	93 312	$< 1m$
N40ft407	40	831.20	831.20	46 656	$< 1m$
N40ft408	40	1002.70	1002.70	139 968	$< 1m$
N40ft409	40	1000.50	1000.50	139 968	$< 1m$
N40ft410	40	983.80	983.80	46 656	$< 1m$
N60ft201	60	1353.50	1353.50	93 312	$< 15m$
N60ft202	60	1161.60	1161.60	93 312	$< 15m$
N60ft203	60	1182.90	1182.90	46 656	$< 15m$
N60ft204	60	1257.50	1257.50	46 656	$< 15m$
N60ft205	60	1184.10	1184.10	93 312	$< 15m$
N60ft206	60	1199.60	1199.60	1 213 056	$< 15m$
N60ft207	60	1299.00	1299.00	46 656	$< 15m$
N60ft208	60	1113.00	1113.00	46 656	$< 15m$
N60ft209	60	1171.30	1171.30	186 624	$< 15m$
N60ft210	60	1234.30	1234.30	139 968	$< 15m$
N60ft301	60	1337.00	1337.00	1 539 648	$< 15m$
N60ft302	60	1089.50	1089.50	606 528	$< 15m$
N60ft303	60	1179.00	1179.00	233 280	$< 15m$
N60ft304	60	1230.00	1230.00	373 248	$< 15m$
N60ft305	60	1151.60	1151.60	886 464	$< 15m$
N60ft306	60	1167.90	1167.90	3 125 952	$< 15m$
N60ft307	60	1220.10	1220.10	2 239 488	$< 15m$
N60ft308	60	1097.60	1097.60	2 472 768	$< 15m$
N60ft309	60	1140.60	1140.60	1 726 272	$< 15m$
N60ft310	60	1219.20	1219.20	2 099 520	$< 15m$
N60ft401	60	1335.00	1335.00	3 452 544	$< 15m$
N60ft402*	60	1088.10	1089.20	9 844 416	$= 15m$
N60ft403	60	1173.70	1173.70	1 446 336	$< 15m$
N60ft404	60	1184.70	1184.70	326 592	$< 15m$
N60ft405	60	1146.20	1146.20	373 248	$< 15m$
N60ft406*	60	1140.20	1141.60	93 312	$= 15m$
N60ft407*	60	1198.90	1203.90	9 844 416	$= 15m$
N60ft408	60	1029.40	1029.40	2 006 208	$< 15m$
N60ft409	60	1121.40	1121.40	466 560	$< 15m$
N60ft410	60	1189.60	1189.60	2 612 736	$< 15m$

REFERENCES

[1] B. Golden, S. Raghavan, and E. Wasil, *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer Verlag, 2008.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.

[3] E. Baker. An exact algorithm for the time-constrained traveling salesman problem. *Operations Research*, 31(5):938–945, 1983.

[4] T. Cazenave. Monte-Carlo Beam Search. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1): 68-72, 2012.

[5] T. Cazenave. Nested Monte-Carlo search. In *IJCAI*, pages 456–461, 2009.

[6] T. Cazenave and F. Teytaud. Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In *LION*, pages 42–54, 2011.

[7] N. Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

[8] Y. Dumas, J. Desrosiers, E. Gelinas, and M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.

[9] S. Edelkamp and M. Gath. Optimal decision making in agent-based autonomous groupage traffic. In *ICAART*, 2013. To appear.

[10] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 1134–1139, 2008.

[11] F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002.

[12] S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS-Workshop on On-line Trading of Exploration and Exploitation*, 2006.

[13] M. Gendreau, A. Hertz, G. Laporte, and M. Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, 1998.

[14] R. Jonker and A. Volgenant. Improving the hungarian assignment algorithm. *Operations Research Letters*, 5:171–175, 1986.

[15] L. Kocsis and C. Szepesvari. Bandit based Monte–Carlo planning. In *ICML*, pages 282–293, 2006.

[16] M. Lopez-Ibanez and C. Blum. Beam-ACO for the travelling salesman problem with time windows. *Computers & OR*, 37(9):1570–1583, 2010.

[17] N. C. Love, T. L. Hinrichs, and M. R. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, 2006.

[18] S. Parragh, K. Doerner, and Richard Hartl. A survey on pickup and delivery problems. *Journal für Betriebswirtschaft*, 58(2):81–117, 2008.

[19] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[20] J. Potvin and S. Bengio. The vehicle routing problem with time windows part II: genetic search. *INFORMS Journal on Computing*, 8(2):165, 1996.

[21] A. Rimmel, F. Teytaud, and T. Cazenave. Optimization of the nested Monte-Carlo algorithm on the traveling salesman problem with time windows. In *Applications of Evolutionary Computation*, pages 501–510, 2011.

[22] A. Rimmel, F. Teytaud, and O. Teytaud. Biasing Monte-Carlo simulations through rave values. In *Computers and Games*, pages 59–68, 2011.

[23] C. D. Rosin. Nested rollout policy adaptation for Monte-Carlo tree search. In *IJCAI*, pages 649–654, 2011.

[24] M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.