A Local Search Algorithm for Large Maximum Weight Independent Set Problems

Yuanyuan Dong[1]
Dallas, TX, USA

Andrew V. Goldberg
Amazon.com, Santa Clara, CA, USA

Alexander Noe
Amazon.com, Bellevue, WA, USA

Nikos Parotsidis[1]
Department of Computer Science, University of Copenhagen, Denmark

Mauricio G.C. Resende
Amazon.com, Bellevue, WA, USA
Industrial & Systems Engineering, University of Washington, Seattle, WA, USA

Quico Spaen
Amazon.com, Santa Clara, CA, USA

Abstract

Motivated by a real-world vehicle routing application, we consider the maximum-weight independent set problem: Given a node-weighted graph, find a set of independent (mutually nonadjacent) nodes whose node-weight sum is maximum. Some of the graphs arising in the vehicle routing application are large, having hundreds of thousands of nodes and hundreds of millions of edges.

To solve instances of this size, we develop a new local search algorithm, which is a metaheuristic based on the greedy randomized adaptive search (GRASP) framework. This algorithm, named METAMIS, uses a wider range of simple local search operations than previously described in the literature. We introduce data structures that make these operations efficient. A new variant of path-relinking is introduced to escape local optima and so is a new alternating augmenting-path local search move that improves algorithm performance.

We compare an implementation of our algorithm with a state-of-the-art publicly available code on public benchmark sets, including some large instances. Our algorithm is, in general, competitive and outperforms this openly available code on large vehicle routing instances of the maximum weight independent set problem. We hope that our results will lead to even better maximum-weight independent set algorithms.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases GRASP, local search, maximum-weight independent set, path-relinking, heuristic, metaheuristic

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.45

Related Version *Full Version*: https://arxiv.org/abs/2203.15805

1 Introduction

Given an undirected graph $G = (V, E)$, where V is the set of nodes and E the set of edges, an *independent set* $S \subseteq V$ is a set of mutually non-adjacent nodes of graph G. If each node $v \in V$ is assigned a weight w_v, a maximum-weight independent set (*MWIS*) of nodes $S^* \subseteq V$ is an independent set whose sum of weights, $W(S^*) = \sum_{v \in S^*} w_v$ is maximum. We denote $n = |V|$ and $m = |E|$.

[1] The work was done while the author was at Amazon.com

A simple way to state MWIS is as an *Integer Linear Program (ILP).* Let x_v be a binary decision variable such that $x_v = 1$ if node $v \in S \subseteq V$ and $x_v = 0$ otherwise, where S is an independent set of nodes. A simple integer programming (IP) formulation for selecting a maximum-weight independent set of nodes is

$$\max \sum_{v \in V} w_v x_v$$

subject to

$$x_u + x_v \leq 1, \forall\ (u, v) \in E$$
$$x_v \in \{0, 1\}, \forall\ v \in V.$$

A well-known way to strengthen the formulation is to add *clique inequalities.* Let C be a subset of all cliques in the input graph. We add the constraints

$$\sum_{v \in Q} x_v \leq 1 \quad \forall Q \in C.$$

While these constraints are redundant for the ILP problem, they strengthen the linear programming relaxation of the problem.

MWIS is a classical optimization problem that has been extensively studied and has many applications [2]. Solving the MWIS problem is hard. It is one of Karp's original NP-complete problems [10, 12]. The problem is also hard to approximate [11]. Over the years, heuristics have been the workhorse for solving large instances of the maximum independent set problem approximately [18]. In particular, the most successful heuristics have been the ones based on metaheuristic algorithms, such as GRASP [8], tabu search [9], and iterated local search [1, 17].

In this paper we introduce *METAMIS*, a new metaheuristic algorithm for the MWIS problem. METAMIS is based on the *greedy randomized adaptive search procedure* – GRASP [20], with *truncated path-relinking* [19]. GRASP is a procedure consisting of iterations made up from successive constructions of a greedy randomized solution and subsequent iterative improvements of it through a local search, and path-relinking is a technique for escaping local optima by generating intermediate solutions along a path that connects two known high-quality solutions. Our motivation is a long-haul vehicle routing (VR) application that yields large MWIS problems, some with close to 900 thousand nodes. Compared to benchmark instances used in previously published work, the VR-MWIS instances are often larger and have a very different structure. We conduct experiments with METAMIS on MWIS instances arising in different applications, including on our VR-MWIS instances and on other publicly available ones. Due to page limit, we omit some of the details of our implementation. See the full version of the paper [4] for details.

We start with known local search moves and perturbation techniques and introduce new local search moves with data structures to make these moves efficient. We also introduce improved perturbation technique variants. Although our algorithm is a general-purpose heuristic, our motivation comes from the VR problem. A variant of our algorithm takes advantage of the application-specific features. In this application, we have a good initial solution which can be used to for warm-start. In addition, graphs from this application come with a large set of known cliques. This allows us to get a good relaxed LP solution, which we use to guide local search.

Due to the page limit, we omit some of the algorithm and implementation details and focus only on the benchmark from our motivating vehicle routing application. We also omit some intuition and discussions. The full paper [4] covers this material.

2 High Level Description

The MWIS algorithm is an iterative local search algorithm based on the *Greedy Randomized Adaptive Search Procedure (GRASP)* metaheuristic, which is a general metaheuristic for combinatorial optimization [6, 7, 20]. The algorithm also uses *path-relinking* to escape local optima [15, 20].

Algorithm 1 Algorithm Overview.

1: **procedure** MWIS($G = (V, E, w)$, maxTime, S_0)
2: $S \leftarrow$ localSearch(G, S_0)
3: $\mathcal{ES} \leftarrow \{\}$ ▷ Empty set of elite solutions
4: $\mathcal{ES}$.add(S)
5: **while** $t \leq$ maxTime **do**
6: $S_G \leftarrow$ findRandomizedGreedySolution(G)
7: **if** LsBeforeRelinking **then** ▷ Optional local search
8: $S_G \leftarrow$ localSearch(G, S_G)
9: **end if**
10: $S_e \leftarrow \mathcal{ES}$.randomEliteSolution()
11: $S' \leftarrow$ pathRelinking(G, S_G, S_e)
12: $S' \leftarrow$ localSearch(G, S')
13: $\mathcal{ES}$.tryToAddAndEvict(S') ▷ Add solution to elite set, if full evict similar solution of lesser value (or don't insert if no worse elite solution exists)
14: **end while**
15: **return** $\mathcal{ES}$.bestSolution()
16: **end procedure**

Algorithm 1 gives a high-level view of the algorithm. In addition to the graph, the input to the algorithm includes a stopping criterion, e.g., a time limit, and an initial solution. When no such solution is available, one can find a solution using the randomized greedy algorithm described later in this section. The algorithm applies local search to improve the initial solution and enters the main loop. At termination of the local search procedure, we are at a local optimum.

The algorithm maintains a set of *elite* solutions $\mathcal{ES}$, which are the best solutions we have seen so far. We add a solution to $\mathcal{ES}$ immediately after a local search, so the elite solutions are always locally optimal. At each iteration of the loop, we first attempt to escape the local optimum corresponding to the elite solution. In this process, we can decrease the objective function. To escape a local optimum, we first find a randomized greedy solution S_G. Optionally, we apply local search to improve S_G. Then we apply path-relinking to S_G and a random elite solution from $\mathcal{ES}$ to find a new solution S'. Then we apply local search to improve S', and update S^* if we find a better solution.

For the VR-MWIS instances, the algorithm variant without the optional local search (on line 8) works better, so we omit the search for these instances. We also set the size of the elite set $\mathcal{ES}$ to 1, so we only retain the best solution. This setting works best for the VR-MWIS instances. For other problem families, different parameter choices were found to work better [13, 14].

2.1 Greedy Algorithm

The GRASP framework needs a randomized greedy procedure that produces diverse initial solutions.

2.2 Local Search

Algorithm 2 Local Search Procedure

```
procedure LocalSearch(G = (V, E, w), S, numIterations)
    i ← 1
    S* ← S
    while i ≤ numIterations do
        S_i ← {}                                        ▷ Empty solution
        while w(S_i) < w(S) do                          ▷ Repeat until no improvement is found
            S_i ← S
            S ← starOneMoves(G, S)
            S ← AAPMoves(G, S)
            S ← oneStarMoves(G, S)
            if w(S_i) < w(S) then break                 ▷ Solution improved
            end if
            S ← twoStarMoves(G, S)
        end while
        if w(S) > w(S*) then
            S* ← S
            i ← 1
        else
            S ← perturb(S)
        end if
    end while
    return S*
end procedure
```

The local search procedure, outlined in Algorithm 2, repeatedly performs local moves with positive gain. We aim to find positive gain (*improving*) moves until we reach a local optimum, and then we perform a random perturbation of the solution. If we find an improving move, we apply it immediately. We use a subset of (x, y) moves and *alternating augmenting path moves (AAP-moves)*. While the (x, y) moves have been studied previously, the AAP moves are new. We describe the moves at a high level in this section, and give a detailed description in Section 3.

An (x, y) move removes x nodes from the solution and adds y nodes to it while maintaining solution independence. We use $*$ instead of x or y to denote an arbitrary positive integer. Note that the number of applicable moves increases significantly as x and y increase. Previous algorithms used (x, y) moves for small values of x and y. In particular, the algorithm of [17] uses $(*, 1)$ and $(1, *)$ moves. Our algorithm uses $(*, 1)$, $(1, *)$, and $(2, *)$ moves. The number of $(2, *)$ moves is large. We use data structures and operation ordering that make improving moves more likely, which makes our algorithm more efficient. If an (x, y) move renders S non-maximal, we add nodes without a neighbor in S to the independent set in random order until S becomes maximal. Note that through this update sequence, S remains an independent set.

A $(*, 1)$ move inserts a single node u into the current solution S and removes its neighbors from S. Procedure starOneMoves(G, S) applies the $(*, 1)$ moves until these is no such improving move.

A $(1, *)$ move removes a node v from S and adds to S an independent subset I of the nodes whose only neighbor in S before the removal is v. Usually one has multiple choices of independent sets to add. A good heuristic is to add a maximum weight set of the neighbors that maintains independence. This is done when the number of neighbors is small (at most seven in our experiments). We use a naive recursive algorithm: Pick a node u in the neighborhood and recursively solve two subproblems. The first subproblem results by adding u to S and deleting its neighbors from the graph. We get the second subproblem by deleting u without adding it to S. The better of the two corresponding solutions is returned. Procedure oneStarMoves(G, S) applies the $(1, *)$ moves until there is no such improving move.

A $(2, *)$ move removes two nodes, u and v, from S and adds to S an independent subset I of the nodes whose only neighbors in S before the removal is u, or v, or both u and v. Generally, this set is significantly larger than the corresponding set for the $(1, *)$ moves, and the recursive operation used for the $(1, *)$ moves is too expensive. One could use greedy addition, but in our experiments a random addition, that adds to S a random node from I that has no neighbors in S, was better. Procedure twoStarMoves(G, S) applies the $(2, *)$ moves until it finds an improving move or there is no improving $(2, *)$ move. Note that unlike the corresponding procedures for other moves, twoStarMoves exits as soon as it finds an improving move.

Our idea for AAP moves comes from matching algorithms [5], although we use a somewhat different definition. Given an independent set S, we define an AAP P as follows. Let $I = S \cap P$ and $O = P - S$ be nodes of P that are in and out of S, respectively.

1. if $v \in I$, then the neighbors of v on P are in O,
2. if $v \in O$, then the neighbors of v on P are in I,
3. if we *flip* the path, i.e., set $S = S - I + O$, S remains an independent set.

An AAP move finds an alternating augmenting path, flips it, and looks at the change in $w(S)$. If the change is positive, we accept the AAP move; otherwise we reject the move. For efficiency, we apply a limited number of AAP moves. Procedure AAPMoves(G, S) applies the AAP moves until there is no such improving move or we reach the limit on the number of AAP moves.

During an execution of the algorithm, most local search moves do not improve solution quality and thus do not change the solution. Note that complexity of evaluating $(2, *)$ moves is significantly higher than those for the other moves. Our local search repeatedly applies starOneMoves, AAPMoves, and oneStarMoves procedures while these procedures find improving moves. If we find an improving move, an immediate application of these procedures may find additional improvements due to neighborhood changes, so we iterate. Only when these procedures fail to find improving moves we call twoStarMoves. If twoStarMoves fails to improve the solution, we perform a random perturbation.

The perturbation adds a small set of random nodes to S and removes their neighbors. After perturbing, we resume local search. The local search algorithm terminates if there has been no improvement to the best solution after a predefined number of iterations.

2.3 Using the Relaxed LP Solution

In our VR application, we use clique information and get a relaxed LP solution to the relaxed problem. The solution assigns a value $x_v \in [0, 1]$ to each node v. We use these values to bias random node selection in the perturbation step of the local search. When performing a random perturbation in Algorithm 2, we add a node v to the solution with probability proportional to $x_v + \epsilon$. Here ϵ is a positive value (set to $\epsilon = 0.005$) that ensures that each node can be picked, even if $x_v = 0$. This guides the local search by biasing route selection toward nodes with higher fractional relaxed solution value. Using prefix sums we can pick a random node in time $\mathcal{O}(\log |V|)$: We draw a random floating-point number $z \in [0, \sum_{v \in V} x_v)$ and use binary search on the prefix sum array to pick a node such that the sum up to but excluding the node is less than z, and the sum up to and including the node is greater or equal to z.

2.4 Adaptive Path-relinking

Path-relinking is a technique for escaping local optima by generating intermediate solutions along a path that connects two known high-quality solutions. We discuss this technique in the context of MWIS and reversible local search moves. Define an undirected graph associated with the search space MWIS, where the nodes correspond to feasible solutions and the edges correspond to local search moves that transform the solution corresponding to the tail of the edge to the solution corresponding to the head. A path in this graph corresponds to a sequence of the moves that transform the solution at one end of the path into a solution at the other end. Note that the moves need not improve the objective function value. The underlying assumption of path-relinking is that if the end-points of a path correspond to high quality solutions, then the path will contain previously undiscovered high-quality solutions.

For our local search, given two solutions S and T, we can transform S into T as follows. Initialize $S' = S$. At every step, we do either a $(*, 1)$ move or a $(1, *)$ move. In the former case, pick a node $v \in T - S'$, add v to S', and remove neighbors of v from S'. In the latter case, pick a node $v \in S'$, $v \notin T$ and remove v from S'. Let $N(v)$ denote the set of neighbors of v. Then we iterate over nodes u in $N(v) \cap T$. If $N(u) \cap S' = \emptyset$, we add u to S'.

For large graphs, finding good solutions is expensive. Instead of combining two good solutions, we apply path-relinking to combine the randomized greedy solution S_G with the current best solution S^*, which is locally optimal. While S^* is a good solution, S_G may not be good, and the solutions on the path far from S^* are usually not good either. We modify path-relinking so that it examines only a prefix of the path close to S^*. The prefix is small enough so that the solution quality remains good, yet big enough so that the subsequent local search will not end up with a locally optimal solution equivalent to S^*. This an adaptive variant of the *truncated greedy path-relinking* described in [19].

The first modification is to choose the node x to add to S or to remove from S greedily. We pick a node that maximizes the weight of the solution we get after the move. A second modification is to do a truncated path-relinking: we stop the process after a certain number of steps, which we adjust adaptively. We start with a small limit on the number of steps and increase the limit if the algorithm gets stuck in a local optimum of weight $w(S^*)$.

3 Data Structures and Optimizations

For large graphs, the choice of data structures is important for the efficiency of the algorithm. When making trade-offs between performance on sparse and dense graphs we favor the former because our motivating application yields relatively sparse graphs.

Several of our data structures use sets of objects. We use a representation of sets based on hashing. This representation allows constant time addition, deletion, and membership query, and linear time iteration over all set elements. We also assume that if we add an element to the set that already contains the element, the set does not change. Similarly, if we delete an element not in the set, the set does not change.

3.1 Input Graph

The input graph is static: it does not change throughout the execution. We assign to the nodes of the graph integer IDs from $[0, \ldots, n-1]$ and place them in an array, with node i in position i. Each node has an array of edges incident to it. This places the edges incident to a node in contiguous memory locations, assuring that a common operation of scanning an edge list has a good memory locality. We sort edges by IDs of the head node. This allows us to do neighborhood queries (e.g., "Is v in $N(u)$?") in time logarithmic in the degree of u using binary search.

Note that using sets to represent neighborhoods would give constant neighborhood queries and linear time edge list scan. However, the constant factors, both in terms of running time and memory consumption, associated with hashing are large. In addition, we lose the locality in edge list scans. For graphs arising from our motivating application, the array-based implementation is significantly faster than the one based on sets.

3.2 Interstate Graph

The *interstate graph* makes the local search operations more efficient. To describe this graph, we need a few definitions.

For a node $u \in S, (u, v) \in E$, we say that v is a *1-tight neighbor* of u if $N(v) \cap S = \{u\}$ [1]. Note that if we remove u from S, we can add to S any 1-tight neighbor of u.

Two nodes $u, v \in S$ are *mates* if for at least one node $w \notin S$, w has exactly two neighbors in S: $N(w) \cap S = \{u, v\}$. We call the node w a *2-tight neighbor* of u and v. We say that w is a 2-tight neighbor of u if u has a mate v such that w is a 2-tight neighbor of u and v. If we delete u and v from S, we can replace them by an independent set of the union of three sets: the set of the 1-tight neighbors of u, the set of 1-tight neighbors of v, and the set of the shared 2-tight neighbors of u and v.

Our main data structure is the *interstate graph* $G_{IS} = (V, E_{IS}, w)$. For G_{IS}, the nodes and node weights are the same as in the input graph G. The edge set E_{IS} is changed dynamically depending on the nodes in the current independent set S. E_{IS} has three types of edges:

1. $e = (u, v) \in E$, where $u \in S$ and v is a *1-tight neighbor* of u;
2. $e = (u, w) \in E$, where $u \in S$ and w is a *2-tight neighbor* of u;
3. $e = (u, v)$, where $u, v \in S$ are *mates*.

We represent the three edge types separately.

1. For every $u \in S$, we represent its 1-tight neighbors as sets. For $v \notin S$ that is a 1-tight neighbor of u we add the *1-tight* edge (v, u).
2. For every pair of mates u and v, we maintain a set of 2-tight neighbors of u and v. For every 2-tight neighbor $w \notin S$, we add the pair of *2-tight* edges (w, u) and (w, v).
3. For every node v in S, we maintain a set M_v of its mates. Every mate $w \in M_v$ corresponds to a mate edge (v, w).

3.3 Efficient Implementation of (x, y) Moves

In this section we show how to efficiently implement (x, y) moves using the interstate graph and two additional optimizations, one for the $(1, *)$ moves and another for $(*, y)$ moves. We discuss maintenance of the interstate graph in Section 3.2.

To implement $(*, 1)$ operations efficiently, we use an idea from [17]. For every $u \notin S$, we maintain a value

$$\Delta(u) = w(u) - \sum_{v \in S \cap N(u)} w(v)$$

to speed up the $(*, 1)$ moves. Such a move is an improving move when $\Delta(u) > 0$. We keep a set S^+ of the nodes u with $\Delta(u) > 0$. Note that for an efficient implementation of $(*, 1)$ moves, we need to update the vector $\Delta(\cdot)$ and the set S^+. We do this as follows. Every time we add a node u to S, we remove u from S^+. Then for each $v \in N(u)$, $v \notin S$, we set $\Delta(v) = \Delta(v) - w(u)$. Every time we remove u from S, we scan the edge list of u and compute $\Delta(u)$. If $\Delta(u) > 0$, we add u to S^+. Also during the scan, for every neighbor v of u such that $v \notin S$, we increase $\Delta(v)$ by $w(u)$, and if $\Delta(v)$ becomes positive, we add v to S^+. We have an improving $(*, 1)$ move if and only if S^+ is non-empty. In this case, we can pick a node u from S^+ and apply the $(*, 1)$ move to it.

Since for every $u \in S$ we maintain a set of its 1-tight neighbors as a hash set, we can efficiently run the recursive or the greedy algorithm described in Section 2 on this set. Similarly, since for every $u \in S$ we maintain the set of its mates, we can iterate over all mates of u. Furthermore, for a pair of mates u and v, we have the set of the common 2-tight neighbors, and we can apply the randomized algorithm to this set.

Next we describe an optimization that prunes $(1, *)$ and $(2, *)$ moves that are unlikely to improve the solution. For the $(1, *)$ move that removes u, we evaluate the move only if the 1-tight neighborhood of u changed since the last time we evaluated the move but failed to improve the solution. We say that the neighborhood changed if we add u to S and u has a non-trivial 1-tight neighborhood. Since our implementation of the $(1, *)$ move is deterministic and depends only on the 1-tight neighborhood, we know that the move will fail. We maintain the set S_1 of nodes $u \in S$ whose 1-tight neighborhood changed but is not empty. We pick nodes for $(1, *)$ moves from S_1. While initializing G_{IN}, we initialize S_1 to include all nodes with non-trivial 1-tight neighborhoods. When we update G_{IN}, we also update S_1 (see Section 3.5).

For the $(2, *)$ move, we maintain a set S_2 of mate pairs $\{u, v\}$ which are eligible for the move. We delete a pair from S_2 and evaluate the move that removes this pair from S. We add a pair $\{u, v\}$ to S_2 when they become 2-tight mates, or when $\{u, v\}$ are 2-tight mates and their 2-tight neighborhood changes, or when they are 2-tight mates and the 1-tight neighborhood of either u or v changes. Our implementation of the $(2, *)$ move depends only on the 2-tight neighborhood of the mates. However, the implementation is randomized. Although it is possible that one evaluation of the move succeeds and another fails when the 2-tight neighborhood stays the same, we assume this is unlikely and prune the move. We maintain the set S_2 of mates whose 2-tight neighborhood changed. We pick mates for $(2, *)$ moves from S_2. While initializing G_{IN}, we initialize S_2 to all pairs of mates. When we update G_{IN}, we update S_2 as well.

3.4 AAP Moves

For efficiency, we only look for alternating augmenting paths (AAPs) in the interstate graph. The only edges on any AAP are either edges from members of S to their 1-tight and 2-tight neighbors (as edges between 2-tight mates would not yield an alternating path). To limit the number of AAP move evaluations, we start a search for an AAP from a 1-tight neighbor of $v \in S_1$ (S_1 was introduced in Section 3.3). This way we guarantee that the move will not decrease the cardinality of S, making the move more likely to succeed. The alternating path initially contains v and its single neighbor $u \in S$. We grow the path as follows. Let $u \in S$ be the last node on the current AAP, and let U be the set of nodes on the AAP that are in S and $\bar{U}$ be the set of nodes on the AAP that are not in S. We pick a mate w and a 2-tight neighbor x of u such that

- x is not a neighbor of any node of $\bar{U}$ in the input graph (so that the extended path will be an AAP),
- neither x nor w are already in AAP,
- the gain of flipping the extended path is maximized.

If we succeed in finding such a $\{w, x\}$ pair, we add w and x to the path. Then we redefine u to be x and continue growing the path. To introduce additional randomness, we increase the gain for every $\{w, x\}$ pair by a random real number $\epsilon \in [-\delta,\ \delta]$ and maximize the perturbed gains. We use $\delta = 50$ in our experiments. We terminate the search if the length of the path exceeds a threshold or the gain of flipping the path falls below a (negative) threshold. We then perform the highest positive gain move that flips a prefix of the final path. If no positive gain move is encountered, we do nothing (the move fails).

3.5 Maintaining the Interstate Graph

The vast majority of the local search moves we evaluate do not improve the solution and G_{IN} does not change. We need to update the graph only when a move succeeds, which happens rarely. Our data structures speed up move evaluations and support move pruning. The added overhead is in data structure initialization and updates. The update complexity is non-trivial, but for sparse graphs the complexity is much smaller than the time we save due to the improved move efficiency and pruning.

Let $\rho(u) = |N(u) \cap S|$ denote the number of the neighbors of u in S. Note that for nodes $u \in S$, $\rho(u) = 0$. We maintain $\rho(u)$ for all nodes $u \in V$.

Given an initial solution S, we build G_{IN}, S_1, and S_2 as follows. We process all nodes $u \notin S$. For each u, we scan its edge list in G and initialize $\rho(u)$. If $\rho(u) = 1$, we let $N(u) \cap S = \{v\}$, add the 1-tight edge (u, v) to the edge list of u in G_{IN}, and add u to the set of 1-tight neighbors of v. If $\rho(u) = 2$, we let $N(u) \cap S = \{v, w\}$, add v to the set of mates of w and add w to the set of mates of v. We also add the pair of 2-tight edges (u, v) and (u, w) to G_{IN}. Finally, we add u to the set of 2-tight neighbors of the mates $\{v, w\}$. We initialize S_1 to the set of all nodes $u \in S$ with non-empty set of 1-tight neighbors. We initialize S_2 to the set of all mate pairs $\{u, v\}$. The initialization takes linear time.

Our algorithm updates S by removing a set of nodes S^- and adding a set of nodes S^+. We break the update into a sequence of single-node updates: first we remove nodes of S^- one by one, then we add nodes of S^+ one by one. We update G_{IN} after each individual update of S.

After removing a node u from S, we empty its set of 1-tight neighbors and remove u from S_1. For each mate v of u, we set the corresponding set of 2-tight neighbors to empty and remove u from the set of mates of v. We also remove the pair $\{u, v\}$ from S_2. Afterwards, we empty the set of mates of u. We then visit its neighbors $v \in V \setminus S$. For each neighbor v, we decrement $\rho(v)$. We need to update G_{IN} if $\rho(v)$ becomes zero, one, or two.

Cases for zero and two are simpler. If the value is zero, we set the 1-tight neighbor of v to null. If the value is two, let $N(v) \cap S = \{a, b\}$. We can find a and b by scanning the edge list of v in G. We add a to the set of mates of b and vice versa. We also add v to the set of 2-tight neighbors of $\{a, b\}$. Finally, we add the 2-tight pair of edges (v, a) and (v, b) to G_{IN}.

If the value is one, we have to update both the old 2-tight neighborhood and the new 1-tight neighborhood. For the latter, we set the 1-tight neighbor of v to the unique neighbor $w \in S$, and add v to the 1-tight neighbor set of v. For the former update, note that v was a 2-tight neighbor for mates $\{v, w\}$ for some $w \in S$ before the removal of v. We remove v from the set of 2-tight neighbors of w and delete the 2-tight edge pair (v, u) and (v, w) from G_{IN}.

Now consider the addition of a node u to S that maintains the independence of S. We scan the edge list of u and for all neighbors v (guaranteed not to be in S) and increment $\rho(v)$. We need to update G_{IN} if $\rho(v)$ becomes one, two, or three.

Cases for one and three are simpler. If the value is one, we add the 1-tight edge (v, u) to G_{IN}, add v to the set of 1-tight neighbors of u, and add u to S_1. If the value is three, v has a pair of 2-tight edges (v, a) and (v, b), where a and b are mates. We delete (v, a) and (v, b) from G_{IN}. Then we remove v from the set of 2-tight neighbors of a and b. If the set becomes empty, a and b are no longer mates, so we remove a from the set of mates of b, remove b from the list of mates of a, and remove $\{a, b\}$ from S_2.

If the value is two, we have to update both the old 1-tight neighborhood and the new 2-tight neighborhood. For the former, let (v, w) be the 1-tight edge. We remove the edge and remove v from the set of 1-tight neighbors of w. If the set becomes empty, we remove w from S_1. In the latter case, $N(v) \cap S = \{v, w\}$ for some $w \in S$. We add u to the set of mates of w and vice versa. We also add v to the set of 2-tight neighbors of v and w. Finally, we add $\{a, b\}$ to S_2.

Note that since when we add or remove u to or from S, we may need to scan edge lists of multiple neighbors of u, updating G_{IN} when G is dense may be expensive.

4 Experimental results

4.1 Algorithms and Computational Environment

We implemented our algorithm, which we call *METAMIS*, in Java because it is used in a production system at Amazon and Java is a requirement. For the same reason, we use doubles for node weights. Furthermore, due to licensing restrictions, we use only standard Java libraries. We compiled our code using Java 8.

Although one can tune our algorithm for specific problem families, we use fixed parameter settings in all experiments.

We compare our implementation to the *ILSVND* algorithm of [17]. The publicly available code of [17] is implemented in C++ and represents weights using integers. We made one modification to ILSVND: added the ability to warm start from an initial solution. Given a solution in the input, we initialize the current solution of ILSVND to the input solution. We compiled ILSVND using full optimization (`-O3`).

For a given instance, algorithm time-quality plots give a lot of information about relative performance of the algorithms. For example, one algorithm may dominate another, or one can converge to a better solution but take longer to converge, etc. The algorithms we compare are stochastic and algorithm performance depends on the pseudo-random seed we use. Furthermore, the algorithms we compare do not know if and when they reach an optimal solution. Usually there is a chance that a solution may improve. However, the algorithms *converge* in a sense that it may reach a point of diminishing returns when a substantial improvement becomes unlikely. To compare the two algorithms, we put a time limit T on their executions. For different problem families, the limit may be different. We run each instance with five different pseudo-random seeds and report the best solution value the algorithm finds. In many cases the algorithms converge. However, for harder problems this may take too long, and the algorithms do not converge within the time limit.

For representative instances, we give the time-quality plots, but we have too many instances to give all the plots. Therefore, we report solution quality at times $T/10$ and $T/2$. In addition, we report the time t^* defined as follows. For a given problem instance, consider the set of final solution values over all algorithms and seed values. Let s be the smallest one of these values. For a given algorithm, consider the run producing the best final solution value. For this algorithm, we define t^* to be the earliest time this run reaches the value of s or higher, Intuitively, we are comparing best runs of the algorithms being evaluated.

For graph algorithms, C++ is usually faster than Java by a factor from three to six. We expect this to hold for our algorithm as well, especially since we make heavy use of standard Java hash set library, which incurs significant overhead compared to C++. Although we do not adjust the runtimes we report, one has to keep this in mind that if re-implemented in C++, our algorithm would be faster.

We run our experiments on an AWS r3.4xlarge instance with 122GiB RAM and 16 virtual CPUs on Intel Xeon Ivy Bridge processors.

4.2 Computational Results

Our full study [4] uses three benchmark families, but due to the page limit we focus on the benchmark from our motivating application, vehicle routing [3]. In this application, the MWIS problem comes up in several contexts, and we have several instances for each of these contexts.

Tables and plots appear in the appendix. Table 1 lists the *VR instances* with their sizes. The number of nodes in these instances ranges from 979 to 883,238; the number of edges ranges from 3,140 to 389,304,424. The instances are moderately sparse, but the density tends to grow with the problem size. The average degree is below 4 on some small instances and over 400 on some large ones.

Table 1 has additional information: values for the initial solutions we use and upper bounds on optimal solution values. We obtain the upper bounds by solving the corresponding LP relaxation problems to optimality. The initial solution are good: their values are close to the upper bound. Note that an optimal solution may not achieve the upper bound.

For VR instances, we have additional information: relaxed LP solutions and initial solutions. We use this information in practice as it yields better results. In our experiments, we give results both for runs with and runs without initial solutions. We also run our algorithm with initial solutions but without the relaxed solutions to see how much a good initial solution matters, and to have an apples to apples comparison with ILSVND, which does not use this information.

In this section we discuss *VR Instances* [3], which motivated our work. Plot for 2-hour runs of all algorithms on one of the largest instances, CR-S-L-4, given in Figure 1, provides insight into relative algorithm performance. All codes converge, and METAMIS dominates corresponding ILSVND runs. Without warm start, ILSVND solution is worse than the initial solution while METAMIS finds a better solution. With warm start, both algorithms find better solutions. Although plots for METAMIS with and without LP data look very close. However, Table 3 shows that the best solution value with LP was 1% better than without LP: $5,775,704$ vs. $5,715,256$.

Next we discuss performance of VR instances in detail. Here we set the time limit T = 3 600 seconds. Table 2 gives results for MWIS with no additional data. For each instance in the table, column $w_{10\%}$ shows the best solution value found at time point $T/10$, column $w_{50\%}$ shows the best solution value found at time point $T/2$, and column w shows the best solution value found when the process is finished at time T. METAMIS finds better solutions than ILSVND except for three instances. For two instances, MT-D-01 and MT-W-01, solution quality is the same. On MW-W-01, the ILSVND solution is better, but only by 0.8%. All three exceptions happen on smaller instances and both algorithms converge quickly. There is no improvement after time $T/10$.

An interesting observation is that on MT-D-01 and MT-W-01, solution values match the corresponding upper bounds given in Table 1, so the solutions are optimal. Since the upper bound need not be tight, it is possible that we solve other instances to optimality, but do not have a proof.

On larger instances, METAMIS has better final values as well as better values at times $T/10$ and $T/2$. On the problem with the highest number of nodes, CR-S-L-3, the difference in the final values is 2.1%. Note that on large instances, neither algorithms converged in time T.

Table 3 shows results for the VR instances for METAMIS+LP, METAMIS, and ILSVND. Note that on three instances, `MT-D-FN`, `MW-D-FN`, and `MW-W-FN`, ILSVND fails to improve the initial solution and t^* is undefined. METAMIS improves the solution on these instances, probably due to a more sophisticated set of local search operations. While both algorithms allow a warm start from a given solution, the METAMIS+LP version of our algorithm uses clique information to compute the relaxed LP solution, and uses it to guide local search., We evaluate both versions of METAMIS to see how much the LP relaxation helps. As in the case of no initial solution, the algorithms converge on most of the small instances and do not converge on larger instances.

Recall that with no initial solution, we found optimal solutions for MT-D-01 and MT-W-01. With the initial solution, METAMIS+LP finds an optimal solution for two more instances, MT-W-FN and MR-W-FN. METAMIS finds an optimal solution for the latter instance, but not for the former. ILSVND does not find any new optimal solutions.

Next we discuss the effect of a good initial solution, comparing results for METAMIS and ILSVND from Tables 2 and 3. Comparing initial solution values from Table 1 with solutions obtained by solving the problems from scratch, we see that in many cases, the initial solution is better than the solution computed from scratch. In fact, for ILSVND, most solutions are worse than the corresponding initial solution. This confirms that our initial solutions are good.

With the warm start, both variants of our algorithm, METAMIS and METAMIS+LP, dominate ILSVND, producing same or (in most cases) better quality solutions. ILSVND is also slower on all instances except one.

To evaluate the benefit of using LP relaxation, we compare METAMIS+LP with METAMIS. On most instances, METAMIS+LP dominates METAMIS. The latter never finds a better solution. For about 1/3 of the instances, solution quality is the same, and for the remaining 2/3, METAMIS+LP performs better. The same holds for intermediate times $T/10$ and $T/2$ except for one instance at $T/2$ where METAMIS solution value is slightly better.

5 Concluding remarks

We developed METAMIS for a real-world VR application for which even a small improvement in solution quality yields substantial cost reduction. Our study is the first to include the benchmark of VR instances [3]. We show that METAMIS works well on the VR instances. We also observed that the VR instances have a structure that is different from that of computer, road, and social network (CRS) instances [16]. The main result of Lamm [16] are local transformations which reduce a MWIS problem to an equivalent problem that is much smaller. On the VR instances, the transformations failed to reduce the problem size significantly.

Our full paper shows that METAMIS works well of the CRS instances. The algorithm of Lamm [16] solves these instances to optimality. Instances of Lamm [16] are hard to reproduce due to weight randomization. In the full paper, we define weights so that they are easy to reproduce. It would be interesting to run the algorithm of Lamm [16] and compare the results.

METAMIS uses a more sophisticated set of local search moves and introduces data structures and lazy evaluation techniques that facilitate efficient implementation of these moves. We also introduce a new variation of path-relinking tailored to large problems. In addition, we show how to use a good relaxed solution to guide local search. These techniques add to the metaheuristic design toolset. We hope that our ideas will lead to even more efficient MWIS algorithms. The ideas may also prove useful in metaheuristic algorithms for other problems.

References

1 D.V. Andrade, M.G.C. Resende, and R.F. Werneck. Fast local search for the maximum independent set problem. *J. of Heuristics*, 18:525–547, 2012.

2 S. Butenko. *Maximum independent set and related problems with applications*. PhD thesis, U. of Florida, Gainesville, Florida, 2003.

3 Y. Dong, A.V. Goldberg, A. Noe, N. Parotsidis, M.G.C. Resende, and Q. Spaen. New instances for maximum weight independent set from a vehicle routing application. *Operations Research Forum*, 2(48), 2021. doi:10.1007/s43069-021-00084-x.

4 Y. Dong, A.V. Goldberg, A. Noe, N. Parotsidis, M.G.C. Resende, and Q. Spaen. A Metaheuristic Algorithm for Large Maximum Weight Independent Set Problems. Technical Report arXiv:2203.15805, arXiv.org, 2022.

5 J. Edmonds. Paths, trees, and flowers. *Can. J. Math.*, 17:449–467, 1965.

6 T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

7 T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

8 T.A. Feo, M.G.C. Resende, and S.H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994.

9 C. Friden, A. Hertz, and D. de Werra. STABULUS: A technique for finding stable sets in large graphs with tabu search. *Computing*, 42:35–44, 1989.

10 M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.

11 J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

12 R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

13 A.F. Kummer. Personal communication, 2020.

14 A.F. Kummer, M.G.C. Resende, and M. Souto. Automatic algorithm configuration and selection of MetaMIS for maximum independent set. Technical report, Amazon MMPROS, Seattle, 2020.

15 M. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

16 S. Lamm, C. Schulz, D. Strash, R. Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019*, pages 144–158. SIAM, 2019. doi:10.1137/1.9781611975499.12.

17 B. Nogueira, R.G.S. Pinheiro, and A. Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, 12(3):567–583, 2018.

18 M. Pelillo. Heuristics for maximum clique and independent set. In C.A. Floudas and P.M. Pardalos, editors, *Encyclopedia of Optimization*, pages 1508–1520. Springer US, Boston, MA, 2009.

19 M.G.C. Resende, R. Martí, M. Gallego, and A. Duarte. GRASP and path relinking for the max-min diversity problem. *Computers & Operations Research*, 37:498–508, 2010.

20 M.G.C. Resende and C.C. Ribeiro. *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures.* Springer, New York, 2016.

A Appendix: Tables and Plots

Table 1 VR instances.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Initial Sol.	LP bound
MT-D-01	979	3 841	228 874 404	238 166 485
MT-D-200	10 880	547 529	286 750 411	287 228 467
MT-D-FN	10 880	645 026	290 723 959	290 881 566
MT-W-01	1 006	3 140	299 132 358	312 121 568
MT-W-200	12 320	554 288	383 620 215	384 099 118
MT-W-FN	12 320	593 328	390 596 383	390 869 891
MW-D-01	3 988	19 522	465 730 126	477 563 775
MW-D-20	10 790	718 152	522 485 254	531 510 712
MW-D-40	33 563	2 169 909	533 938 531	543 396 252
MW-D-FN	47 504	4 577 834	542 182 073	549 872 520
MW-W-01	3 079	48 386	1 268 370 807	1 270 311 626
MW-W-05	10 790	789 733	1 328 552 109	1 334 413 294
MW-W-10	18 023	2 257 068	1 342 415 152	1 360 791 627
MW-W-FN	22 316	3 495 108	1 350 675 180	1 373 020 454
MR-D-01	14 058	60 738	1 664 446 852	1 695 332 636
MR-D-03	21 499	168 504	1 739 544 141	1 763 685 757
MR-D-05	27 621	295 700	1 775 123 794	1 796 703 313
MR-D-FN	30 467	367 408	1 794 070 793	1 809 854 459
MR-W-FN	15 639	267 908	5 386 472 651	5 386 842 781
CW-T-C-1	266 403	162 263 516	1 298 968	1 353 493
CW-T-C-2	194 413	125 379 039	933 792	957 291
CW-T-D-4	83 091	43 680 759	457 715	463 672
CW-T-D-6	83 758	44 702 150	457 605	463 946
CW-S-L-1	411 950	316 124 758	1 622 723	1 677 563
CW-S-L-2	443 404	350 841 894	1 692 255	1 759 158
CW-S-L-4	430 379	340 297 828	1 709 043	1 778 589
CW-S-L-6	267 698	191 469 063	1 159 946	1 192 899
CW-S-L-7	127 871	89 873 520	589 723	599 271
CR-T-C-1	602 472	216 862 225	4 605 156	4 801 515
CR-T-C-2	652 497	240 045 639	4 844 852	5 032 895
CR-T-D-4	651 861	245 316 530	4 789 561	4 977 981
CR-T-D-6	381 380	128 658 070	2 953 177	3 056 284
CR-T-D-7	163 809	49 945 719	1 451 562	1 469 259
CR-S-L-1	863 368	368 431 905	5 548 904	5 768 579
CR-S-L-2	880 974	380 666 488	5 617 351	5 867 579
CR-S-L-4	881 910	383 405 545	5 629 351	5 869 439
CR-S-L-6	578 244	245 739 404	3 841 538	3 990 563
CR-S-L-7	270 067	108 503 583	1 969 254	2 041 822

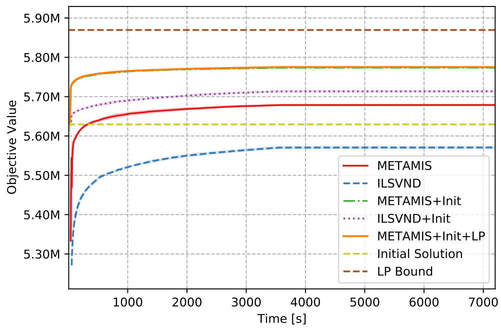

Figure 1 Time-quality plot for CR-S-L-4. Note that the plots for METAMIS+Init and METAMIS+Init+LP are very close.

Table 2 Results on VR instances with no additional information.

	METAMIS				ILSVND			
Name	$w_{10\%}$	$w_{50\%}$	w	$t^*[s]$	$w_{10\%}$	$w_{50\%}$	w	$t^*[s]$
MT-D-01	**238 166 485**	**238 166 485**	**238 166 485**	**0.948**	**238 166 485**	**238 166 485**	**238 166 485**	1.290
MT-D-200	**286 976 422**	**287 048 909**	**287 048 909**	**188.1**	286 838 210	286 838 210	286 943 799	2 276
MT-D-FN	**290 866 943**	**290 866 943**	**290 866 943**	**104.4**	290 393 532	290 666 380	290 666 380	561.6
MT-W-01	**312 121 568**	**312 121 568**	**312 121 568**	0.278	**312 121 568**	**312 121 568**	**312 121 568**	**0.080**
MT-W-200	383 818 136	**383 961 099**	**383 961 323**	1 433	**383 865 836**	383 896 403	383 896 403	**1 036**
MT-W-FN	390 688 944	**390 830 057**	**390 854 593**	**568.1**	**390 715 890**	390 798 842	390 798 842	709.2
MW-D-01	**476 099 262**	**476 164 209**	**476 334 711**	**267.9**	475 653 439	475 906 790	475 906 790	1 173
MW-D-20	**524 255 389**	**525 036 493**	**525 124 575**	**85.40**	520 854 115	522 415 092	523 138 978	2 685
MW-D-40	**533 934 442**	**535 707 479**	**536 520 199**	**81.36**	530 227 261	532 272 896	532 400 878	1 830
MW-D-FN	**539 754 400**	**541 372 345**	**541 918 916**	**98.34**	532 663 872	537 238 784	537 674 129	2 466
MW-W-01	**1 270 305 952**	**1 270 305 952**	**1 270 305 952**	**0.500**	1 246 949 460	1 246 949 460	1 246 949 460	23.66
MW-W-05	**1 328 958 047**	**1 328 958 047**	**1 328 958 047**	**19.96**	1 327 687 399	1 328 707 787	1 328 707 787	984.8
MW-W-10	**1 340 878 388**	**1 342 899 725**	**1 342 899 725**	**1 204**	1 331 002 512	1 341 482 310	1 342 067 985	1 876
MW-W-FN	**1 349 369 736**	**1 350 818 543**	**1 350 818 543**	**527.7**	1 334 835 589	1 348 128 240	1 350 159 705	3 584
MR-D-01	**1 689 074 331**	**1 689 520 690**	**1 689 781 114**	**15.52**	1 683 529 331	1 686 091 786	1 687 842 856	2 906
MR-D-03	**1 753 188 475**	**1 753 968 167**	**1 754 110 286**	**20.34**	1 743 429 914	1 747 269 072	1 749 972 580	3 257
MR-D-05	**1 784 519 403**	**1 785 664 042**	**1 786 342 921**	**19.56**	1 770 832 093	1 774 407 092	1 777 876 780	3 595
MR-D-FN	**1 795 912 642**	**1 797 284 091**	**1 797 573 192**	**22.65**	1 779 897 201	1 785 545 729	1 788 331 878	3 388
MR-W-FN	**5 357 026 363**	5 358 386 615	5 358 386 615	1 442	5 352 347 338	**5 370 471 580**	**5 371 649 721**	**461.6**
CW-T-C-1	**1 310 223**	**1 315 122**	**1 317 775**	**94.52**	1 290 974	1 299 279	1 302 478	3 585
CW-T-C-2	**924 664**	**929 626**	**931 802**	**189.7**	914 736	921 021	922 858	3 599
CW-T-C-4	**454 769**	**456 565**	**457 185**	**324.4**	452 035	453 741	454 544	2 365
CW-T-D-6	**455 823**	**457 382**	**457 790**	**70.48**	452 366	454 254	454 254	1 582
CW-S-L-1	**1 623 280**	**1 630 417**	**1 634 950**	**261.9**	1 603 051	1 615 247	1 620 756	3 597
CW-S-L-2	**1 695 131**	**1 704 424**	**1 708 820**	**225.3**	1 670 836	1 685 870	1 690 536	3 596
CW-S-L-4	**1 712 553**	**1 722 542**	**1 725 591**	**173.7**	1 689 318	1 701 309	1 706 264	3 599
CW-S-L-6	**1 150 229**	**1 156 916**	**1 158 925**	**138.4**	1 136 356	1 142 720	1 145 694	3 086
CW-S-L-7	**582 925**	**585 929**	**587 288**	**125.2**	577 087	581 583	581 583	1 278
CR-T-C-1	**4 617 204**	**4 644 635**	**4 654 419**	**58.16**	4 508 901	4 558 780	4 576 695	3 598
CR-T-C-2	**4 834 040**	**4 863 054**	**4 874 346**	**62.29**	4 715 023	4 772 847	4 789 909	3 600
CR-T-D-4	**4 778 868**	**4 808 490**	**4 817 281**	**56.91**	4 663 588	4 716 258	4 734 674	3 598
CR-T-D-6	**2 945 721**	**2 964 007**	**2 970 011**	**94.09**	2 896 260	2 921 540	2 929 671	3 574
CR-T-D-7	**1 431 915**	**1 438 896**	**1 440 281**	**148.4**	1 411 061	1 423 279	1 426 400	3 581
CR-S-L-1	**5 547 038**	**5 575 602**	**5 588 489**	**72.42**	5 400 658	5 464 532	5 487 254	3 595
CR-S-L-2	**5 652 928**	**5 680 688**	**5 691 892**	**57.91**	5 491 814	5 561 766	5 586 973	3 580
CR-S-L-4	**5 634 886**	**5 671 369**	**5 681 336**	**65.09**	5 477 340	5 550 943	5 572 856	3 573
CR-S-L-6	**3 833 391**	**3 851 432**	**3 859 513**	**92.45**	3 751 019	3 793 995	3 808 314	3 599
CR-S-L-7	**1 977 161**	**1 986 354**	**1 989 879**	**90.90**	1 940 573	1 957 872	1 963 579	3 584

Table 3 METAMIS+LP, METAMIS, ILSVND results on VR instances with start solution.

Name	METAMIS+LP $w_{10\%}$	METAMIS+LP $w_{50\%}$	METAMIS+LP w	METAMIS+LP $t^*[s]$	METAMIS $w_{10\%}$	METAMIS $w_{50\%}$	METAMIS w	METAMIS $t^*[s]$	ILSVND $w_{10\%}$	ILSVND $w_{50\%}$	ILSVND w	ILSVND $t^*[s]$
MT-D-01	**238 166 485**	**238 166 485**	**238 166 485**	**0.109**	**238 166 485**	**238 166 485**	**238 166 485**	0.373	**238 166 485**	**238 166 485**	**238 166 485**	1.473
MT-D-200	**287 038 328**	**287 048 081**	**287 048 081**	**69.51**	287 010 847	287 018 324	287 036 715	122.6	286 949 274	286 973 561	286 973 561	363.1
MT-D-FN	**290 771 450**	**290 771 450**	**290 771 450**	–	290 752 054	**290 771 450**	**290 771 450**	–	290 723 959	290 723 959	290 723 959	–
MT-W-01	**312 121 568**	**312 121 568**	**312 121 568**	0.122	**312 121 568**	**312 121 568**	**312 121 568**	0.320	**312 121 568**	**312 121 568**	**312 121 568**	**0.063**
MT-W-200	**383 971 124**	383 985 408	383 985 408	**893.0**	383 804 298	**383 986 483**	**383 986 483**	1 343	383 808 376	383 979 962	383 979 962	1 721
MT-W-FN	**390 828 160**	**390 869 891**	**390 869 891**	**139.9**	390 787 880	390 848 998	390 856 179	710.1	390 805 960	390 805 960	390 805 960	196.2
MW-D-01	**475 886 356**	**475 987 082**	**475 987 082**	**270.2**	475 549 969	475 814 986	475 955 989	2 278	475 523 699	475 732 519	475 825 497	2 134
MW-D-20	**525 052 532**	**525 402 318**	**525 486 034**	8.694	524 574 519	525 068 939	525 192 291	**7.699**	523 248 884	523 248 884	523 248 884	26.98
MW-D-40	**535 705 687**	**536 210 247**	**536 735 155**	**0.434**	535 436 892	535 711 417	536 092 070	0.474	534 040 009	534 040 009	534 040 009	7.797
MW-D-FN	**543 098 071**	**543 622 238**	**543 857 187**	–	542 740 347	543 253 226	543 374 394	–	542 182 073	542 182 073	542 182 073	–
MW-W-01	1 269 314 742	**1 269 344 846**	**1 269 344 846**	672.0	**1 269 344 846**	**1 269 344 846**	**1 269 344 846**	**0.603**	**1 269 344 846**	**1 269 344 846**	**1 269 344 846**	1.247
MW-W-05	**1 328 958 047**	**1 328 958 047**	**1 328 958 047**	**0.431**	**1 328 958 047**	**1 328 958 047**	**1 328 958 047**	0.447	1 328 955 871	1 328 955 871	1 328 955 871	4.266
MW-W-10	**1 342 915 691**	**1 342 915 691**	**1 342 915 691**	**0.511**	**1 342 915 691**	**1 342 915 691**	**1 342 915 691**	1.255	1 342 847 887	1 342 847 887	1 342 847 887	19.39
MW-W-FN	**1 350 814 699**	1 350 814 699	**1 350 818 543**	–	1 350 771 010	**1 350 818 542**	**1 350 818 543**	–	1 350 675 180	1 350 675 180	1 350 675 180	–
MR-D-01	**1 688 024 106**	**1 688 777 944**	**1 689 278 470**	**7.245**	1 687 486 503	1 687 807 619	1 688 118 984	16.84	1 684 211 854	1 686 046 636	1 686 452 467	2 763
MR-D-03	**1 756 186 736**	**1 756 989 875**	**1 757 227 519**	**5.123**	1 755 768 835	1 756 154 528	1 756 337 669	12.31	1 751 006 933	1 752 345 436	1 752 769 459	3 305
MR-D-05	**1 787 220 357**	**1 787 666 207**	**1 787 849 777**	**19.91**	1 786 084 687	1 786 734 327	1 786 755 817	73.22	1 782 046 226	1 782 560 957	1 783 836 981	3 525
MR-D-FN	**1 798 215 807**	**1 798 926 794**	**1 799 452 160**	**17.40**	1 798 075 911	1 798 571 155	1 798 661 823	38.60	1 794 949 819	1 794 949 819	1 796 037 791	3 564
MR-W-FN	**5 386 842 781**	**5 386 842 781**	**5 386 842 781**	**0.503**	**5 386 842 781**	**5 386 842 781**	**5 386 842 781**	0.855	5 386 838 669	5 386 838 669	5 386 838 669	10.01
CW-T-C-1	**1 334 884**	**1 336 953**	**1 338 064**	30.69	1 333 129	1 335 297	1 336 563	**22.44**	1 322 410	1 326 551	1 327 556	3 501
CW-T-C-2	**944 404**	**945 748**	**945 886**	**25.86**	943 366	944 785	945 565	27.72	939 568	940 356	940 356	701.3
CW-T-D-4	**460 643**	**461 027**	**461 056**	2.000	460 554	460 852	461 025	**1.828**	458 360	458 360	458 360	48.65
CW-T-D-6	**460 982**	**461 223**	**461 312**	2.717	460 815	461 057	461 174	**2.706**	459 096	459 096	459 096	80.73
CW-S-L-1	**1 656 404**	**1 660 475**	**1 660 815**	**46.27**	**1 656 404**	**1 660 475**	**1 660 815**	90.11	1 644 241	1 649 006	1 651 483	3 585
CW-S-L-2	**1 731 077**	**1 735 964**	**1 738 128**	**85.11**	1 730 208	1 734 736	1 736 245	109.3	1 714 923	1 722 672	1 724 930	3 452
CW-S-L-4	**1 748 029**	**1 752 354**	**1 753 803**	91.08	1 746 941	1 751 474	1 751 988	**84.05**	1 733 007	1 739 992	1 742 459	3 553
CW-S-L-6	1 174 005	**1 175 931**	**1 177 156**	**27.48**	**1 174 169**	1 175 886	1 176 233	33.79	1 167 611	1 169 914	1 170 096	1 886
CW-S-L-7	593 045	**593 744**	593 891	**4.825**	**593 077**	**593 744**	**593 947**	6.622	591 398	591 398	591 398	161.2
CR-T-C-1	**4 730 533**	**4 739 684**	**4 743 040**	**17.92**	4 725 855	4 735 644	4 738 289	18.10	4 665 849	4 687 422	4 696 568	3 591
CR-T-C-2	**4 954 613**	**4 966 121**	**4 968 952**	25.80	4 950 818	4 962 045	4 964 446	**19.83**	4 891 697	4 912 140	4 920 058	3 585
CR-T-D-4	**4 898 377**	**4 908 285**	**4 911 646**	19.69	4 896 504	4 906 792	4 909 999	**17.86**	4 836 312	4 859 311	4 867 272	3 597
CR-T-D-6	**3 017 902**	**3 022 448**	**3 024 523**	**28.67**	3 016 890	3 022 046	3 023 349	40.35	2 990 174	2 999 852	3 004 067	3 593
CR-T-D-7	**1 458 949**	**1 459 958**	**1 460 240**	16.88	1 458 571	1 459 653	1 459 948	**8.115**	1 455 226	1 456 048	1 456 048	752.0
CR-S-L-1	**5 672 398**	**5 686 939**	**5 692 891**	36.79	5 668 764	5 685 884	5 690 515	**21.79**	5 590 089	5 617 916	5 630 437	3 596
CR-S-L-2	**5 763 866**	**5 780 859**	**5 784 034**	24.90	5 759 512	5 775 002	5 780 449	**22.53**	5 670 522	5 701 371	5 715 430	3 589
CR-S-L-4	**5 756 016**	**5 771 410**	**5 777 081**	**24.08**	5 755 282	5 771 391	5 775 704	24.18	5 676 163	5 701 271	5 715 256	3 598
CR-S-L-6	**3 926 517**	**3 933 476**	**3 936 137**	22.24	3 923 574	3 932 059	3 935 089	**19.00**	3 883 092	3 898 898	3 905 831	3 597
CR-S-L-7	**2 014 584**	**2 018 371**	**2 019 428**	**34.09**	2 013 466	2 017 034	2 017 836	40.24	1 998 320	2 006 129	2 007 794	3 488