C++ for Combinatorial Optimization

From Exact Solvers to Metaheuristics

Luka Matijević

Mathematical Institute of the Serbian Academy of Sciences and Arts

Računarski fakultet, 12.06.2026.

Outline

About Me

- Research Assistant Professor at MI SANU
- Assistant Professor at Metropolitan University
- Research interests: operations research and metaheuristics
- Author of 17 papers — 5 in international journals, 12 in conference proceedings
- Freelance developer: web applications and information systems
- Primary tools: C++ and Python

What Is Combinatorial Optimization?

Combinatorial optimization: finding the best object from a finite, but typically enormous, set of candidates.

- Decision variables are *discrete*
- Feasible region grows combinatorially with problem size
- Most interesting problems are NP-hard

Classical examples:

- Traveling Salesman Problem (TSP)
- Vehicle Routing Problem (VRP)
- Knapsack, Bin Packing, Graph Coloring
- Scheduling and Assignment problems

Why It Matters

Applications

- Logistics and supply chain
- Energy systems
- Manufacturing and scheduling
- Telecommunications
- Healthcare operations
- Finance and portfolio design

The challenge

- Real instances are large
- Solutions are needed fast
- Constraints keep changing
- Performance directly translates to cost savings

Why C++?

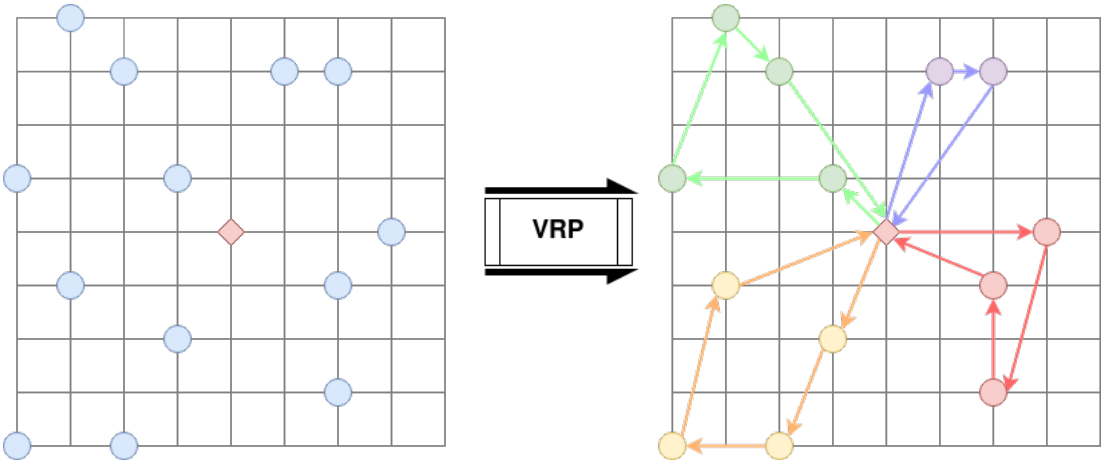
- **Performance**: low-level control, predictable memory layout, minimal runtime overhead
- **Abstraction without cost**: templates, constexpr, RAII
- **Ecosystem**: most commercial and open-source solvers expose C or C++ APIs
- **Maturity**: numerical libraries, profilers, debuggers, build systems
- **Reality**: tight inner loops in metaheuristics benefit directly from cache-friendly C++ code

Python and other languages are great for prototyping. When the algorithm matters and instances are large, C++ remains the workhorse.

Roadmap

1. EVRP with time-dependent speeds and soft time windows
2. A MILP formulation (TD-EVRP-STW)
3. Solving small instances with GLPK in C++
4. Wrapping a C API in idiomatic C++
5. Why exact methods are not enough
6. General Variable Neighborhood Search for TD-EVRP-STW
7. Experimental results vs. CPLEX and ALNS

Vehicle Routing Problem



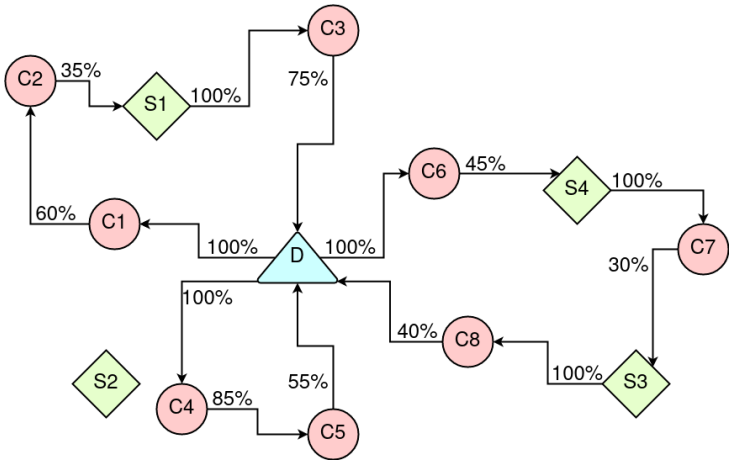
From VRP to EVRP

Vehicle Routing Problem (VRP): route a fleet of vehicles to serve customers at minimum cost. **Electric VRP**: vehicles are battery-powered.

- Limited driving range
- Must visit recharging stations during the route
- Recharging takes time (not instantaneous like refueling)
- Recharging station selection becomes part of the decision

Practically motivated by last-mile delivery with electric fleets.

EVRP


C3
C2
35%
S1
100%
75%
C6
45%
S4
100%
60%
C1
100%
D
100%
C7
30%
100%
40%
C8
S2
55%
100%
S3
C4
85%
C5

TD-EVRP-STW: The Problem We Solve

Time-dependent speeds and soft time windows

- Vehicle speed depends on the time of day (rush hours, congestion)
- Travel time on an arc is computed dynamically based on departure time and which time intervals it spans
- Customers have preferred time windows — arriving early or late incurs a penalty
- Penalties are added to the objective, not enforced as hard constraints

Real-world inspired: last-mile delivery in urban areas where waiting at a customer is expensive and traffic is non-uniform.

Assumptions

- Homogeneous fleet: same capacity, consumption rate, base speed
- Full recharge at each visit to a station (no partial recharges)
- Recharging time is linear in the missing battery level
- Energy consumption is linear in distance
- Recharging stations have unlimited capacity
- Each customer has demand, service time, and a soft time window

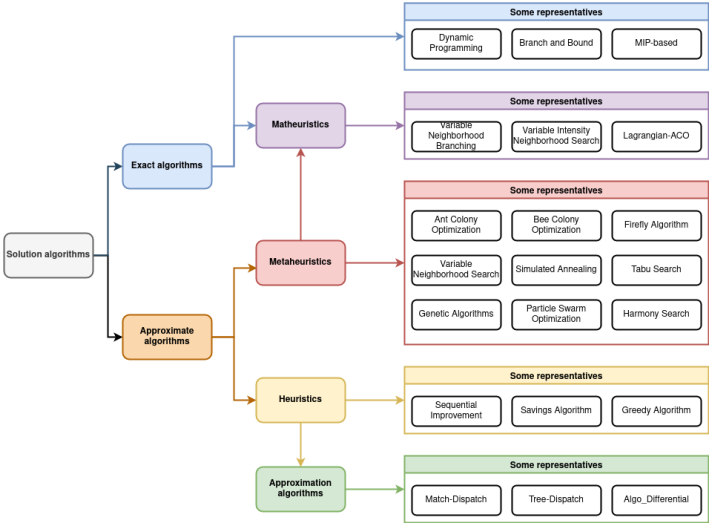
Nonlinear consumption and charging models can be added to the metaheuristic without changing the algorithm.

Why TD-EVRP-STW Is Hard

- VRP alone is NP-hard
- Battery constraints introduce a continuous state along each route
- Recharging station decisions add a combinatorial layer
- Time-dependent speeds break the simple ”distance/speed” model
- Soft time windows expand the feasible region but add objective complexity

Even instances with 15 customers push CPLEX to its limits.

Solution approaches



Graph and Sets

Directed complete graph $G = (N, A)$:

- $0, N{+}1$ — depot copies (origin and destination)
- $V = \{1, \ldots, N\}$ — customers
- F — recharging stations; F' — replicated dummy copies of F (each station may be visited multiple times)

Useful unions:

$$V' = V \cup F'$$
$$V'_0 = V' \cup \{0\}, \qquad V'_{N+1} = V' \cup \{N{+}1\}$$

Time horizon is split into intervals $k \in K$, each with start ψ_k, end υ_k, and speed L_k.

Decision Variables

$x_{ij} \in \{0,1\}$	arc (i,j) used
$z_{ijk} \in \{0,1\}$	arc (i,j) traversed in interval k
$\tau_i \geq 0$	arrival time at i
$\phi_i \geq 0$	departure time from i
$u_i \geq 0$	remaining cargo on arrival at i
$y_i \geq 0$	battery level on arrival at i
$\varphi_{ijk} \geq 0$	distance on arc (i,j) in interval k
$\mu_i, \eta_i \geq 0$	early / late penalty at i
$p_i \geq 0$	total penalty at i

Objective

Minimize total distance plus time-window penalty:

$$\min \left(c_1 \sum_{i \in V'_0} \sum_{\substack{j \in V'_{N+1} \\ i \neq j}} d_{ij} x_{ij} \; + \; c_2 \sum_{i \in V} p_i \right)$$

- c_1, c_2 control the trade-off between distance and penalty
- In our experiments: $c_1 = c_2 = 1$
- Soft windows are reflected purely through p_i in the objective

Routing and Flow Constraints

Each customer visited exactly once:

$$\sum_{\substack{j \in V'_{N+1} \\ i \neq j}} x_{ij} = 1, \qquad \forall i \in V$$

Each dummy station visited at most once:

$$\sum_{\substack{j \in V'_{N+1} \\ i \neq j}} x_{ij} \leq 1, \qquad \forall i \in F'$$

Flow conservation:

$$\sum_{\substack{j \in V'_{N+1} \\ i \neq j}} x_{ji} - \sum_{\substack{j \in V'_0 \\ i \neq j}} x_{ij} = 0, \qquad \forall j \in V'$$

Time Propagation

Arrivals from customers and depot:

$$\tau_i + \sum_{k \in K} t_{ijk} + s_i - l_0(1 - x_{ij}) \leq \tau_j$$

Arrivals from recharging stations (full recharge time included):

$$\tau_i + \sum_{k \in K} t_{ijk} + g(Q - y_i) - (l_0 + gQ)(1 - x_{ij}) \leq \tau_j$$

- g — recharging rate
- Q — battery capacity
- s_i — service time at customer i

Soft Time Window Penalty

Early and late deviation:

$$\mu_i = \max(0, e_i - \tau_i), \qquad \eta_i = \max(0, \tau_i - l_i)$$

Total penalty at customer i:

$$p_i = \alpha\mu_i + \beta\eta_i$$

Because p_i is being minimized, the $\max$ can be linearized by two inequalities each, without auxiliary binary variables:

$$\mu_i \geq e_i - \tau_i, \quad \mu_i \geq 0$$

Capacity and Battery

Capacity (Miller–Tucker–Zemlin):

$$0 \leq u_j \leq u_i - q_i x_{ij} + C(1 - x_{ij})$$

Battery from customers:

$$0 \leq y_j \leq y_i - h\, d_{ij}\, x_{ij} + Q(1 - x_{ij})$$

Battery from depot/stations (full charge):

$$0 \leq y_j \leq Q - h\, d_{ij}\, x_{ij}$$

C — vehicle capacity, h — consumption rate per unit distance.

Time-Dependent Travel

Distance on (i,j) split across time intervals:

$$d_{ij} \cdot x_{ij} = \sum_{k \in K} \varphi_{ijk}$$

Link distance and indicator:

$$\varphi_{ijk} \leq d_{ij} z_{ijk}, \qquad \sum_{k \in K} z_{ijk} \geq x_{ij}$$

Travel time per interval:

$$t_{ijk} = \frac{\varphi_{ijk}}{L_k}$$

Each segment cannot exceed its interval's length:

$$t_{ijk} \leq \upsilon_k - \psi_k$$

FIFO Property

Departure time consistency:

$$\phi_i \geq \tau_i + s_i, \qquad \phi_i \geq \tau_i + g(Q - y_i) \text{ for } i \in F'$$

Interval boundaries are respected:

$$\phi_i + t_{ijk} \leq \upsilon_k + \upsilon_{\text{last}}(1 - z_{ijk})$$

$$\psi_k + t_{ijk} \leq \tau_j + \upsilon_{\text{last}}(1 - z_{ijk})$$

This formulation correctly handles speed changes at interval boundaries — avoiding the step-function pitfalls noted by Ichoua et al. (2003).

Strengths and Limitations of MILP

What you gain

- Provably optimal solutions on small instances
- A clean mathematical statement of the problem
- A baseline to compare metaheuristics against

What you lose

- Scalability — runtime explodes past $\sim$15 customers
- Modelling multiple visits requires dummy nodes, which inflates the model
- Nonlinear consumption / charging cannot be handled directly

Open-Source MIP Solvers

Solver	Language	Notes
GLPK	C	Small, simple, GNU project
CBC	C++	COIN-OR, mature
HiGHS	C++	Modern, actively developed
SCIP	C	Strong on MIP, academic license
OR-Tools	C++	Google, broad toolkit

Today we use **GLPK**: a small C library, easy to call from C++, perfect for teaching the underlying mechanics.

GLPK in a Nutshell

- Pure C library, header `<glpk.h>`, link with `-lglpk`
- Problem is built up imperatively: rows, columns, coefficients
- Solvers: `glp_simplex` for LP, `glp_intopt` for MIP
- Memory is managed manually
- API is low-level — a perfect opportunity for a C++ wrapper

Creating the Problem

```c
#include <glpk.h>

glp_prob *lp = glp_create_prob();
glp_set_prob_name(lp, "evrp");
glp_set_obj_dir(lp, GLP_MIN);

// Add rows (constraints) and columns (variables)
int n_vars = ...;
int n_cons = ...;
glp_add_cols(lp, n_vars);
glp_add_rows(lp, n_cons);
```

Each variable and each constraint must be configured separately with its bounds, type, and objective coefficient.

Defining Variables

```
// x_ij is a binary variable
for (int k = 1; k <= n_arcs; ++k) {
    glp_set_col_name(lp, k, ("x_" + name(k)).c_str());
    glp_set_col_kind(lp, k, GLP_BV);          // binary
    glp_set_obj_coef(lp, k, distance[k]);     // c_ij
}

// y_i: continuous, 0 <= y_i <= Y
for (int k = n_arcs + 1; k <= n_arcs + n_nodes; ++k) {
    glp_set_col_kind(lp, k, GLP_CV);
    glp_set_col_bnds(lp, k, GLP_DB, 0.0, Y);
}
```

Loading the Constraint Matrix

```cpp
// GLPK uses 1-indexed arrays of (row, col, value) triplets
std::vector<int>    ia{0}, ja{0};
std::vector<double> ar{0.0};

for (auto const& [i, j, v] : nonzeros) {
    ia.push_back(i);
    ja.push_back(j);
    ar.push_back(v);
}

glp_load_matrix(lp,
                static_cast<int>(ia.size()) - 1,
                ia.data(), ja.data(), ar.data());
```

Notice: GLPK expects *1-indexed* arrays. The leading dummy element is the canonical way to handle this in C++.

Solving and Reading Results

```
glp_iocp params;
glp_init_iocp(&params);
params.presolve = GLP_ON;
params.tm_lim   = 60 * 1000;        // 60-second time limit

int status = glp_intopt(lp, &params);

if (status == 0) {
    double obj = glp_mip_obj_val(lp);
    for (int k = 1; k <= n_vars; ++k) {
        double v = glp_mip_col_val(lp, k);
        // extract solution
    }
}

glp_delete_prob(lp);   // do not forget!
```

Pain Points with the Raw API

- Manual memory management (`glp_delete_prob`)
- 1-indexed arrays mixing with C++ 0-indexed containers
- C-style `int` constants instead of typed enums
- Strings passed as `const char*`
- Errors reported through return codes, not exceptions
- Builders are imperative and verbose

These are real issues — and an opportunity to apply modern C++.

RAII for the Problem Handle

```
class GlpkProblem {
public:
    GlpkProblem() : prob_(glp_create_prob()) {}

    ~GlpkProblem() { if (prob_) glp_delete_prob(prob_); }

    GlpkProblem(GlpkProblem const&) = delete;
    GlpkProblem& operator=(GlpkProblem const&) = delete;

    GlpkProblem(GlpkProblem&& other) noexcept
        : prob_(std::exchange(other.prob_, nullptr)) {}

    glp_prob* get() const noexcept { return prob_; }

private:
    glp_prob* prob_;
};
```

Scoped Enums Instead of Macros

```
enum class VarKind  { Continuous, Integer, Binary };
enum class Sense    { Minimize, Maximize };
enum class BoundKind { Free, Lower, Upper, Double, Fixed };

constexpr int to_glpk(VarKind k) {
    switch (k) {
        case VarKind::Continuous: return GLP_CV;
        case VarKind::Integer:    return GLP_IV;
        case VarKind::Binary:     return GLP_BV;
    }
}
```

Type-safe, self-documenting, and the compiler catches typos.

A Small Builder for Constraints

```
class ConstraintBuilder {
public:
    ConstraintBuilder& term(int var_idx, double coef) {
        rows_.push_back(idx_);
        cols_.push_back(var_idx);
        vals_.push_back(coef);
        return *this;
    }
    void le(double rhs) { /* GLP_UP */ }
    void ge(double rhs) { /* GLP_LO */ }
    void eq(double rhs) { /* GLP_FX */ }
private:
    int idx_;
    std::vector<int>    rows_, cols_;
    std::vector<double> vals_;
};
```

What CPLEX Can and Cannot Do

On TD-EVRP-STW with our settings:

- Instances up to $\sim$10 customers: optimality in seconds to minutes
- Instances with 15 customers: typically reports a feasible solution but does not close the gap within 60 min
- Larger instances (100 customers): no feasible solution found in 60 min
- Dummy nodes for station replication inflate the model substantially

The same C++ code base with GLPK shows the same qualitative pattern — with even tighter scalability limits.

Exact vs. Heuristic Methods

Exact methods

- Guarantee optimality (given enough time)
- Provide dual bounds
- Scale poorly with instance size

Heuristic and metaheuristic methods

- No optimality guarantees
- Find very good solutions in reasonable time
- Scale to industrial instances
- Flexible: easy to add new constraints

The Metaheuristic Landscape

Trajectory-based

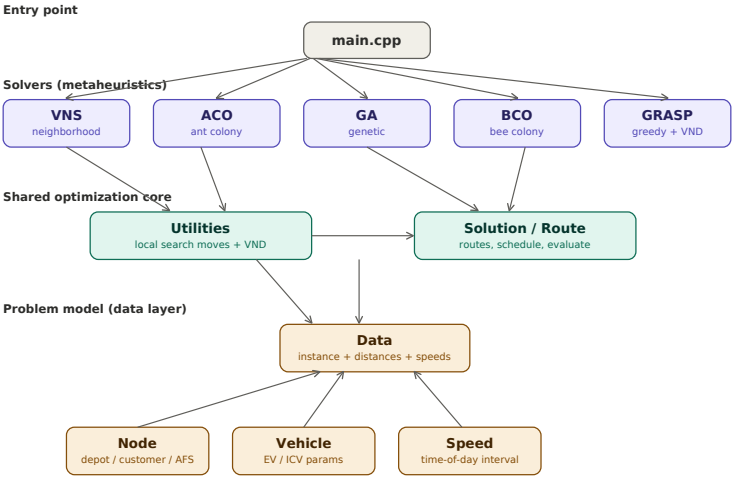
- Local Search
- Simulated Annealing
- Tabu Search
- Variable Neighborhood Search
- Iterated Local Search

Population-based

- Genetic Algorithms
- Evolutionary Strategies
- Particle Swarm
- Ant Colony Optimization
- Memetic Algorithms

Most modern solvers are hybrids: heuristic search combined with exact subproblems.

EVRP Solver High-Level Architecture



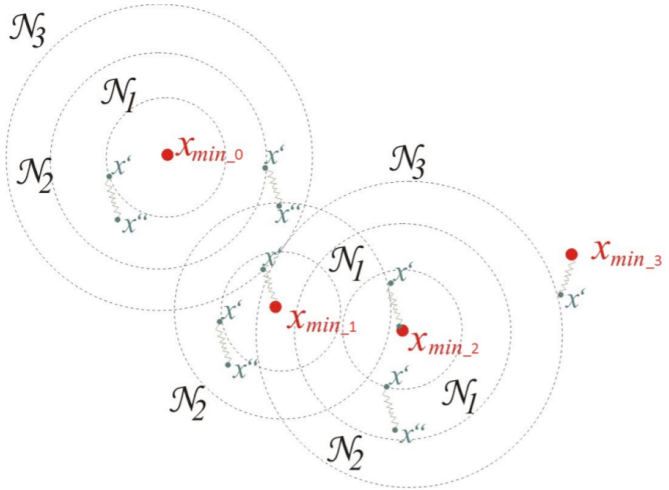
General Variable Neighborhood Search

VNS (Mladenović and Hansen, 1997): single-solution metaheuristic that systematically changes neighborhood structures. Three core steps:

- **Shaking**: jump to a random neighbor at distance k
- **Local search**: improve the shaken solution
- **Neighborhood change**: accept or move to a larger neighborhood

GVNS: VNS where the local search itself is a Variable Neighborhood Descent (VND).

General Variable Neighborhood Search



Solution Representation

```
class Route {
public:
    int vehicleId;
    deque<int>     customers;     // node indices in visit order
    vector<double> arrivals;      // arrival[j]   at customers[j]
    vector<double> departure;     // departure[j] from customers[j]
    // ...
};

class Solution {
public:
    vector<Route> routes;          // pre-allocated, MAX_TRUCKS slots
    int           trucksUsed;
    double        value = 0.0;
    // ...
};
```

- deque<int> gives O(1) push/pop *and* random access — `list` cannot do random access, `vector` cannot do O(1) push_front
- Schedules live next to the route, not in a separate map
- Routes are pre-allocated; `trucksUsed` tracks the live count

Ten Neighborhood Structures

$N_1(S, \varepsilon)$	Shift departure time of a customer by $\pm\varepsilon$
$N_2(S)$	Move a customer to another position in same route
$N_3(S)$	Move a customer to a different route
$N_4(S)$	Swap two customers within the same route
$N_5(S)$	Swap two customers between routes
$N_6(S)$	Remove a recharging station from a route
$N_7(S)$	Move a recharging station to another position
$N_8(S)$	Replace a recharging station with another
$N_9(S, k)$	Move k consecutive nodes within the same route
$N_{10}(S, k)$	Move k consecutive nodes to another route

and N_{10} are used in shaking.

N_1–N_8 are used in local search; N_9

Time-Dependent Travel

```
double Data::travelTime(int i, int j, double start, int vehicle) {
    double dist     = getNode(i).distance(getNode(j));
    size_t interval = getSpeedInterval(start);
    double speed    = speeds[interval].value;
    double time     = 0;

    while (dist > 0) {
        double a = dist / speed;                // time to finish arc
        double b = speeds[interval].end - time; // time until next int.
        double c = min(a, b);                   // whichever comes first
        dist -= c * speed;
        time += c;
        interval = min(interval + 1, speeds.size() - 1);
        speed    = speeds[interval].value;
    }
    return time;
}
```

A compact loop that correctly handles arcs that cross multiple speed intervals — the FIFO property of Ichoua et al. (2003).

Battery Feasibility: Repair by Insertion

```
for (size_t j = 1; j < route.customers.size(); ++j) {
    double spent = dist(j, j-1) / consumptionRate;

    if (current_battery >= spent) {
        current_battery -= spent;       // ok, advance
    } else {
        // Scan backward to find a station insertion point
        for (size_t k = j; k > 1; --k) {
            int best = find_best_station_at(k);
            if (best != -1) {
                route.customers.insert(begin + k, best);
                break;
            }
            current_battery += dist(k, k-1) / consumptionRate;
        }
        current_battery = fullCapacity;
        j--;                            // retry from same step
    }
}
```

Repair in Words

Time windows are soft, so they cannot make a solution infeasible. The two hard constraints are:

- **Capacity**: removing customers always restores feasibility
- **Battery depletion**: requires inserting a recharging station in the right place

The repair walks each route, tracking battery. When a step would deplete it, scan backward and try inserting the best feasible station. If none works, walk further back. **Real-world detail**: the repair has a hard 0.2-second budget per call. Pathological inputs can lock it up otherwise.

VND in Practice

```
void Utilities::VND(Data &d, Solution &sol,
                    clock_t begin, int VND_LIMIT)
{
    bool ret = true;
    clock_t vnd_begin = clock();

    while (ret && (clock() - vnd_begin)/CLOCKS_PER_SEC <= VND_LIMIT) {
        ret = moveCustomer(d, sol);     if (ret) continue;
        ret = relocateCustomer(d, sol); if (ret) continue;
        ret = swapCustomer(d, sol);     if (ret) continue;
        ret = exchangeCustomer(d, sol); if (ret) continue;
        ret = removeStation(d, sol);    if (ret) continue;
        ret = shiftStation(d, sol);     if (ret) continue;
        ret = replaceStation(d, sol);
    }
}
```

- First-improvement: as soon as one neighborhood succeeds, restart
- Order of neighborhoods matters — cheapest, highest-impact first
- Time-bounded: even VND has a budget

The Try-Revert Neighborhood Pattern

```
bool Utilities::moveCustomer(Data &d, Solution &sol) {
    Solution temp;
    sol.copySolution(temp);              // snapshot

    for (int i = 0; i < sol.trucksUsed; ++i) {
      for (size_t j = 0; j < sol.routes[i].customers.size(); ++j) {
        if (d.getNode(sol[i][j]).getType() != CUSTOMER) continue;
        for (size_t k = 1; k < sz; ++k) {
            /* apply move in-place */
            sol.calculateSchedule(d);
            sol.optimizeSchedule(d);
            double val = sol.evaluate(d);
            if (val < sol.value && val != -1) {
                sol.setValue(val);
                return true;             // first-improvement
            }
            temp.copySolution(sol);      // revert
        }
      }
    }
    return false;
}
```

Try-Revert: The Real Inner Loop

The pattern

1. Snapshot the current solution
2. Apply a move in place
3. Recompute schedule, evaluate
4. Accept and return, or restore the snapshot

Performance implications

- Each candidate move costs at least one full copy
- Most moves are rejected — so we mostly pay for copies
- The whole schedule is recomputed, not the delta
- Clear opportunity: replace with O(1) delta evaluation + local rollback

Honest takeaway: the simple version got the paper out; the fast version is the natural next step.

Which Neighborhoods Actually Matter?

From the experimental analysis (gradient boosting on solution quality):

- N_2, N_3: the largest contributors — customer moves within and between routes
- N_4, N_6, N_8: moderate impact
- N_5: lower than expected (subsumed by N_3)
- N_7: low impact, often creates infeasibility
- N_1: virtually no impact, can hurt on large instances

Lesson: even a clean design can carry dead weight. Profile not only the code, but the algorithm itself.

Benchmark Setup

- Instances adapted from Schneider et al. (2014)
- Two sets: small (≤ 15 customers) and large (100 customers)
- Hardware: Intel i7-10750H, 32 GB RAM, Ubuntu 20.04
- GCC 9.4 with `-03`
- Exact solver: CPLEX 20.1.0 with 60-minute time limit
- GVNS and ALNS: 10-minute (small) / 20-minute (large) limit, 30 / 10 independent runs

Case Study I: Small Instances vs. CPLEX

Out of 35 small instances:

- GVNS matched CPLEX in 23 instances
- GVNS strictly better than CPLEX in 10 instances (avg. improvement 7.2%, max 23.8%)
- CPLEX strictly better in 2 instances (avg. gap 0.5%, max 0.9%)

Even on instances where CPLEX terminates, GVNS reaches the same quality in a small fraction of the time — typically seconds vs. minutes or hours.

Case Study II: Large Instances (100 customers)

CPLEX cannot find feasible solutions in reasonable time — so the comparison is GVNS vs. ALNS over 56 instances.

- Best solution: GVNS better in 52, tied in 4
- Average solution: GVNS better in 55, ALNS better in 1
- Mean improvement in best solution: 3.3%
- Maximum improvement: 17.9%
- Wilcoxon signed-rank test: $p < 10^{-9}$ in both cases

GVNS is also faster on average to reach its best-known solution.

One Codebase, Many Metaheuristics

```
// main.cpp
Data D = Data(argv[1]);

// VNS V = VNS(D, kmin, kstep, kmax, e);
// ACO A = ACO(D, num_ants, alpha, beta, rho);
// GA  G = GA (D, pop_size, offspring_size);
// BCO B = BCO(D, max_moves, num_of_bees);
// GRASP G = GRASP(D, RCL_LIMIT);
```

Once `Data`, `Solution`, `Route`, and the neighborhood pack exist, plugging in a new metaheuristic is almost mechanical:

- All of them share `evaluate`, `fixSolution`, `calculateSchedule`, `optimizeSchedule`
- Each algorithm contributes its own *search strategy*, not its own problem encoding
- This is where the C++ design effort pays off: not the first paper, but the fifth

What I Learned Building This

- Start with a clean mathematical model, even if you do not plan to solve it exactly
- Wrap C libraries early — RAII pays for itself fast
- Treat the inner loop as a first-class design concern
- Templates and concepts are worth the learning curve for algorithmic code
- Profile before optimizing, but design for performance from the start
- Reproducibility matters: seed everything, log everything

Takeaways for C++ Developers

- C++ shines when algorithms meet performance constraints
- Wrapping a C API is a clean way to apply modern C++
- Generic programming makes algorithms reusable across problems
- Combinatorial optimization is a rich playground for C++ design choices

Resources

This work

- Matijević, L. (2023). General variable neighborhood search for electric vehicle routing problem with time-dependent speeds and soft time windows. *IJIEC* 14, 275–292.

Books

- Talbi, *Metaheuristics: From Design to Implementation*
- Toth and Vigo, *Vehicle Routing: Problems, Methods, and Applications*

Software

- GLPK, HiGHS, OR-Tools

Thank you!

Questions?