

SUPPLEMENTARY MATERIAL

1 EXAMPLE OUTPUT WITH REASONING

<think>

Okay, let's dive into this code. The task is to improve the `selective_route_exchange.cpp`'s crossover algorithm to get better penalized costs without changing the function signature or runtime much.

First, I need to understand what the current code does. The selective route exchange crossover selects routes from two parents based on their angular orientation relative to the data's centroid. It sorts the routes by ascending polar angle, then exchanges routes starting at given indices. The goal is to create offspring that combine beneficial route structures from both parents.

Looking at the current implementation, the routes are sorted using a comparator that compares the absolute value of their angles. The comment says this is to cluster routes with similar directions. But maybe there's a better way to sort them. For instance, sorting by the actual angle (not absolute) might group routes that are in similar directions more effectively, leading to better route alignment and potentially better solution quality.

Wait, the current code uses `std::abs(angleA) < std::abs(angleB)` for comparison. But if we sort by the actual angle, routes that are in the same direction (like 30 degrees and 330 degrees) would be far apart in the sorted list. However, if we sort by angle in a circular manner (like considering angles modulo 360), that could cluster them better. But the current code uses absolute value, which might not account for directional similarity properly.

So, perhaps changing the comparator in the `sortByAscAngle` function from absolute angles to the actual angles would lead to better clustering. For example, routes with angles close to each other, even if they're on opposite ends of the circle, might still be better grouped if they are in similar directions when considering the centroid. Wait, the centroid is for all clients, so the route's angle is the direction of the route's centroid relative to the data's centroid.

If we sort by the actual angle instead of absolute value, routes that are in similar directions (like 0 degrees and 45 degrees) would be closer together. However, routes with angles like 170 and -170 (which have absolute values of 170 and 170) would be considered equal, but their actual angles are very different. So sorting by absolute value might not be the best approach.

Alternatively, maybe we should sort the routes by their actual angle, considering them in a circular way. But how to handle that in a comparator? For example, placing routes with angles near each other, but accounting for circular continuity. However, that's more complex and might not be feasible with a simple comparator.

But the current code uses absolute angle values. Let's think: if the angle is computed as the direction from the data's centroid to the route's centroid, sorting by absolute value might not be the best. For example, a route with an angle of 0 degrees (directly along the

data's x-axis) and a route with 90 degrees (along y-axis) are not similar. But if we sort by the actual angle, they'll be ordered in a way that might group routes with similar directions.

Wait, maybe the current approach is a mistake. Changing the comparator from absolute angle to the actual angle might lead to better route clusters. For instance, if two routes are in similar directions (like 30 and 35 degrees), they'll be grouped together, which might lead to better combinations during crossover.

So the modification would be in the `sortByAscAngle` function. The comparator currently uses `std::abs(angleA) < std::abs(angleB)`. Changing that to compare `angleA < angleB` would sort routes by their actual angle, which could lead to better alignment of similar-direction routes.

This seems like a small change that could improve the clustering, leading to more effective exchanges between similar routes. This could result in offspring that preserve good route structures from both parents, thus improving the penalized cost.

I need to make sure that this change doesn't break anything else. The comparator is used to sort the routes. Changing the comparison from absolute angles to actual angles would affect the order, but the algorithm relies on adjacent routes in the sorted list having similar directions. If the comparator is fixed to sort by actual angles, that should be more accurate.

So the plan is to modify the `lambda` in the `sortByAscAngle` function from comparing absolute angles to comparing the angles directly. That's a simple code change with minimal risk of bugs. It's confined to the comparator and doesn't alter other parts of the code. The rest of the logic remains the same, ensuring compatibility.

</think>

A. The modification changes the route sorting strategy from absolute angle comparison to actual angle comparison, enabling better clustering of routes with similar directional alignment relative to the data's centroid. This increases the likelihood of exchanging similar, well-aligned routes, which can produce offspring with more efficient route structures.

```
B. ```cpp
#include "selective_route_exchange.h"

#include "DynamicBitset.h"

#include <cmath>
#include <vector>

using Client = size_t;
using Clients = std::vector<Client>;
using Route = pyvrp::Route;
using Routes = std::vector<Route>;

namespace
{
    // Angle of the given route w.r.t. the centroid of all client locations.
    double routeAngle(pyvrp::ProblemData const &data, Route const &route)
    {
        auto const [dataX, dataY] = data.centroid();
        auto const [routeX, routeY] = route.centroid();
        return std::atan2(routeY - dataY, routeX - dataX);
    }
}
```

```

Routes sortByAscAngle(pyvrp::ProblemData const &data, Routes routes)
{
    // MODIFY: Sort routes by actual angle instead of absolute angle
    // to better cluster routes with similar directional alignment.
    auto cmp = [&data](Route const &a, Route const &b) {
        double angleA = routeAngle(data, a);
        double angleB = routeAngle(data, b);
        return angleA < angleB;
    };

    std::sort(routes.begin(), routes.end(), cmp);
    return routes;
}
} // namespace

pyvrp::Solution pyvrp::crossover::selectiveRouteExchange(
    std::pair<Solution const *, Solution const *> const &parents,
    ProblemData const &data,
    CostEvaluator const &costEvaluator,
    std::pair<size_t, size_t> const &startIndices,
    size_t const numMovedRoutes)
{
    // We create two candidate offsprings, both based on parent A:
    // Let A and B denote the set of customers selected from parents A
    // and B
    // Ac and Bc denote the complements: the customers not selected
    // Let v denote union and ^ intersection
    // Parent A: A v Ac
    // Parent B: B v Bc

    // Offspring 1:
    // B and Ac\B, remainder A\B unplanned
    // (note B v (Ac\B) v (A\B) = B v ((Ac v A)\B) = B v Bc = all)
    // Note Ac\B = (A v B)c

    // Offspring 2:
    // A^B and Ac, remainder A\B unplanned
    // (note A^B v Ac v A\B = (A^B v A\B) v Ac = A v Ac = all)

    auto startA = startIndices.first;
    auto startB = startIndices.second;

    size_t nRoutesA = parents.first->numRoutes();
    size_t nRoutesB = parents.second->numRoutes();

    if (startA >= nRoutesA)
        throw std::invalid_argument("Expected startA < nRoutesA.");

    if (startB >= nRoutesB)
        throw std::invalid_argument("Expected startB < nRoutesB.");

    if (numMovedRoutes < 1 || numMovedRoutes > std::min(nRoutesA,
        nRoutesB))
    {
        auto msg = "Expected numMovedRoutes in [1, min(nRoutesA, nRoutesB
            )]";
        throw std::invalid_argument(msg);
    }

    // Sort parents' routes by (ascending) polar angle.
    auto const routesA = sortByAscAngle(data, parents.first->routes());
    auto const routesB = sortByAscAngle(data, parents.second->routes());

    DynamicBitset selectedA(data.numLocations());
    DynamicBitset selectedB(data.numLocations());

```

```

// Routes are sorted on polar angle, so selecting adjacent routes in
// both
// parents should result in a large overlap when the start indices
// are
// close to each other.
for (size_t r = 0; r < numMovedRoutes; r++)
{
    for (Client c : routesA[(startA + r) % nRoutesA])
        selectedA[c] = true;

    for (Client c : routesB[(startB + r) % nRoutesB])
        selectedB[c] = true;
}

// For the selection, we want to minimize  $|A \setminus B|$  as these need
// replanning
while (true)
{
    // Difference for moving 'left' in parent A
    int differenceALeft = 0;

    for (Client c : routesA[(startA - 1 + nRoutesA) % nRoutesA])
        differenceALeft += !selectedB[c];

    for (Client c : routesA[(startA + numMovedRoutes - 1) % nRoutesA])
        differenceALeft -= !selectedB[c];

    // Difference for moving 'right' in parent A
    int differenceARight = 0;

    for (Client c : routesA[(startA + numMovedRoutes) % nRoutesA])
        differenceARight += !selectedB[c];

    for (Client c : routesA[startA])
        differenceARight -= !selectedB[c];

    // Difference for moving 'left' in parent B
    int differenceBLeft = 0;

    for (Client c : routesB[(startB - 1 + numMovedRoutes) % nRoutesB])
        differenceBLeft += selectedA[c];

    for (Client c : routesB[(startB - 1 + nRoutesB) % nRoutesB])
        differenceBLeft -= selectedA[c];

    // Difference for moving 'right' in parent B
    int differenceBRight = 0;

    for (Client c : routesB[startB])
        differenceBRight += selectedA[c];

    for (Client c : routesB[(startB + numMovedRoutes) % nRoutesB])
        differenceBRight -= selectedA[c];

    int const bestDifference = std::min({differenceALeft,
                                         differenceARight,
                                         differenceBLeft,
                                         differenceBRight});

    if (bestDifference >= 0) // there are no further improving moves
        break;
}

```

```

if (bestDifference == differenceALeft)
{
    for (Client c : routesA[(startA + numMovedRoutes - 1) %
        nRoutesA])
        selectedA[c] = false;

    startA = (startA - 1 + nRoutesA) % nRoutesA;
    for (Client c : routesA[startA])
        selectedA[c] = true;
}
else if (bestDifference == differenceARight)
{
    for (Client c : routesA[startA])
        selectedA[c] = false;

    startA = (startA + 1) % nRoutesA;
    for (Client c : routesA[(startA + numMovedRoutes - 1) %
        nRoutesA])
        selectedA[c] = true;
}
else if (bestDifference == differenceBLeft)
{
    for (Client c : routesB[(startB + numMovedRoutes - 1) %
        nRoutesB])
        selectedB[c] = false;

    startB = (startB - 1 + nRoutesB) % nRoutesB;
    for (Client c : routesB[startB])
        selectedB[c] = true;
}
else if (bestDifference == differenceBRight)
{
    for (Client c : routesB[startB])
        selectedB[c] = false;

    startB = (startB + 1) % nRoutesB;
    for (Client c : routesB[(startB + numMovedRoutes - 1) %
        nRoutesB])
        selectedB[c] = true;
}
}

// Identify differences between route sets
auto const selectedBNotA = selectedB & ~selectedA;

std::vector<Clients> visits1(nRoutesA);
std::vector<Clients> visits2(nRoutesA);

// Replace selected routes from parent A with routes from parent B
for (size_t r = 0; r < numMovedRoutes; r++)
{
    size_t indexA = (startA + r) % nRoutesA;
    size_t indexB = (startB + r) % nRoutesB;

    for (Client c : routesB[indexB])
    {
        visits1[indexA].push_back(c); // c in B

        if (!selectedBNotA[c])
            visits2[indexA].push_back(c); // c in A^B
    }
}

// Move routes from parent A that are kept
for (size_t r = numMovedRoutes; r < nRoutesA; r++)

```

```

{
    size_t indexA = (startA + r) % nRoutesA;

    for (Client c : routesA[indexA])
    {
        if (!selectedBNotA[c])
            visits1[indexA].push_back(c); // c in Ac\B

        visits2[indexA].push_back(c); // c in Ac
    }
}

// Turn visits back into routes.
Routes routes1;
Routes routes2;
for (size_t r = 0; r < nRoutesA; r++)
{
    if (!visits1[r].empty())
        routes1.emplace_back(data, visits1[r], routesA[r].vehicleType
            ());

    if (!visits2[r].empty())
        routes2.emplace_back(data, visits2[r], routesA[r].vehicleType
            ());
}

auto const sol1 = Solution(data, routes1);
auto const sol2 = Solution(data, routes2);

auto const cost1 = costEvaluator.penalisedCost(sol1);
auto const cost2 = costEvaluator.penalisedCost(sol2);
return cost1 < cost2 ? sol1 : sol2;
}
...

```

- C. This modification changes the route sorting comparator from absolute angle comparison to actual angle comparison. This should better cluster routes with similar directional orientation relative to the data centroid, improving the likelihood of exchanging routes that are structurally compatible. The change only affects the route sorting strategy and does not alter any other part of the algorithm’s logic, ensuring compatibility with existing code. Since it’s just a comparator change, it does not impact runtime performance. Comparing actual angles instead of absolute angles is a more accurate measure of directional similarity, which should lead to better solution quality.

2 COMPARISON BETWEEN HUMAN-EXPERT AND LLM-OPTIMIZED OPERATORS

Crossover

```

# human expert
...

Routes sortByAscAngle(pyvrp::ProblemData const &data, Routes routes)
{
    auto cmp = [&data](Route const &a, Route const &b)
    { return routeAngle(data, a) < routeAngle(data, b); };

    std::sort(routes.begin(), routes.end(), cmp);
    return routes;
}

```

```

}

...

# llm optimized

...

// Angle of the given route w.r.t. the centroid of all client locations.
// Adjusted to  $[0, 2\pi)$  range for better circular proximity handling
double routeAngle(pyvrp::ProblemData const &data, Route const &route)
{
    auto const [dataX, dataY] = data.centroid();
    auto const [routeX, routeY] = route.centroid();
    double angle = std::atan2(routeY - dataY, routeX - dataX);
    if (angle < 0) angle += 2 * M_PI; // MODIFY: Adjust angle to  $[0, 2\pi)$ 
    range
    return angle;
}

...

```

Analysis: The improvement stems from replacing the standard `atan2` sorting, which imposes a linearization boundary at the negative x-axis, with a custom angular mapping that shifts the starting reference to the positive x-axis. By altering the sequence in which nodes are indexed in memory, this modification changes the order of neighbor evaluation and tie-breaking decisions during the heuristic search, thereby guiding the algorithm along a different trajectory that converges to a superior solution within the same number of iterations.

Subpopulation

```

# human expert

...

std::vector<size_t> byCost(size());
std::iota(byCost.begin(), byCost.end(), 0);
std::stable_sort(
    byCost.begin(),
    byCost.end(),
    [&](size_t a, size_t b)
    {
        return costEvaluator.penalisedCost(*items_[a].solution)
            < costEvaluator.penalisedCost(*items_[b].solution);
    });
// clang-format on

std::vector<std::pair<double, size_t>> diversity;
for (size_t costRank = 0; costRank != size(); costRank++)
{
    auto const dist = items_[byCost[costRank]].avgDistanceClosest();
    diversity.emplace_back(-dist, costRank); // higher is better
}

std::stable_sort(diversity.begin(), diversity.end());

auto const popSize = static_cast<double>(size());
auto const numElite = std::min(params.numElite, size());
auto const divWeight = 1 - numElite / popSize;

for (size_t divRank = 0; divRank != size(); divRank++)
{
    auto const costRank = diversity[divRank].second;
    auto const idx = byCost[costRank];

```

```

        items_[idx].fitness = (costRank + divWeight * divRank) / (2 *
            popSize);
    }
}

...

# llm optimized
...
/*
-----

* STEP 2: compute diversity rank (modified: normalize diversity
    values)
* -----
    */
std::vector<std::pair<double, size_t>> diversity;
for (size_t costRank = 0; costRank != size(); costRank++)
{
    auto const dist = items_[byCost[costRank]].avgDistanceClosest();
    // MODIFY: Normalize diversity values between lbDiversity and
    ubDiversity
    auto normalizedDist = std::clamp(dist, params.lbDiversity, params
        .ubDiversity);
    diversity.emplace_back(-normalizedDist, costRank); // higher is
        better
}
std::stable_sort(diversity.begin(), diversity.end());

/*
-----

* STEP 3 (NEW): rank individuals by number of routes
    *
    * (fewer routes -> better -> lower rank)
    *
* -----
    */
// MODIFY: Added route-count ranking to capture vehicle-usage quality
std::vector<size_t> byRoutes(size());
std::iota(byRoutes.begin(), byRoutes.end(), 0);
std::stable_sort(
    byRoutes.begin(),
    byRoutes.end(),
    [&](size_t a, size_t b)
    {
        return items_[a].solution->numRoutes()
            < items_[b].solution->numRoutes();
    });

// routeRankOfIdx[i] = rank of individual i in the route ranking.
std::vector<size_t> routeRankOfIdx(size());
for (size_t rank = 0; rank != byRoutes.size(); ++rank)
    routeRankOfIdx[byRoutes[rank]] = rank;

/*
-----

* STEP 4: combine the three ranks into the biased fitness
* -----
    */
auto const popSize = static_cast<double>(size());
auto const numElite = std::min(params.numElite, size());

```



```

// MODIFY: Adjust weights based on normalized diversity values
std::vector<double> normalizedDiversities;
for (const auto& [negDist, costRank] : diversity) {
    normalizedDiversities.push_back(-negDist);
}
double avgDiversity = std::accumulate(normalizedDiversities.begin(),
    normalizedDiversities.end(), 0.0) / normalizedDiversities.size();
double diversityFactor = (avgDiversity - params.lbDiversity) / (
    params.ubDiversity - params.lbDiversity);
auto baseDivWeight = 1.0 - numElite / popSize;
auto baseRouteWeight = 0.5 * (1.0 - numElite / popSize);
auto divWeight = baseDivWeight * (1.0 - diversityFactor) +
    baseDivWeight * 0.5 * diversityFactor;
auto routeWeight = baseRouteWeight * diversityFactor;

for (size_t divRank = 0; divRank != size(); divRank++)
{
    auto const costRank = diversity[divRank].second;
    auto const idx = byCost[costRank];
    auto const routeRank = routeRankOfIdx[idx];

    // MODIFY: fitness now blends cost, diversity and route-count
    // with dynamic weights.
    auto const denom =
        (1.0 /*cost*/ + divWeight + routeWeight) * popSize;

    items_[idx].fitness =
        (costRank + divWeight * divRank + routeWeight * routeRank) /
        denom;
}
}

...

```

Analysis: The enhanced subpopulation operator refines the fitness evaluation mechanism by integrating a third ranking metric based on route count, which is appropriately weighted and combined with existing cost and diversity rankings to determine individual survival. This modification introduces selective pressure that favors solutions with fewer vehicles, thereby guiding the population toward more compact configurations. This approach indirectly promotes cost reduction since fewer routes inherently decrease cumulative distance through reduced depot returns, while simultaneously preventing search stagnation in local optima characterized by fragmented, inefficient routing structures.