Technical Disclosure Commons

Defensive Publications Series

February 2023

Improved Time-related Checking in Routing Solvers

Steven Gay

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Gay, Steven, "Improved Time-related Checking in Routing Solvers", Technical Disclosure Commons, (February 08, 2023)
https://www.tdcommons.org/dpubs_series/5671

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Improved Time-related Checking in Routing Solvers

ABSTRACT

To solve routing problems with given constraints, an initial routing solution is obtained in a construction phase and iteratively improved in a subsequent improvement phase. With O(1) time complexity to check for constraint satisfaction per change with p being the total length of all paths changed between a known solution and a new candidate solution, current approaches have $O(p^2)$ time and space complexity when replacing the current solution with a new one. This does not scale when routing problems involve paths that have hundreds of nodes. This disclosure describes techniques to perform efficient pre-computations to reduce the time and space complexity to O(p log p), thus significantly saving time and resources. The pre-computations check constraint satisfaction for new candidate solutions generated with small variations from a current solution that is known to respect the constraints. The new candidate solutions are described by the chains of their paths that are changed in comparison to the current solution. The checking and pre-computations are performed with a general algorithm and associated query scheme for each subproblem, and run in linear time in the number of chains. As a result, the time and space complexity of the operation is reduced to O(p log p), where p is the total length of all changed paths in the new solution.

KEYWORDS

- Vehicle fleet
- Routing problem
- Local search
- Route optimization
- Dimensional constraint
- Time complexity
- Space complexity
- Constraint satisfaction

BACKGROUND

Operators of vehicle fleets need to determine optimal route assignments for vehicles in the fleet based on the allocated tasks and time constraints for the fleet as a whole and/or each individual vehicle within the fleet. For example, an operator of a fleet of vehicles used for delivering packages would need to assign routes to each vehicle within the fleet based on the start and end time of the delivery operation, delivery addresses associated with each package, the earliest and latest times by which the packages must be delivered, etc.

A typical vehicle routing problem has a fixed number of nodes, N, that must be visited by a fleet of m vehicles. Each vehicle has distinct and fixed start and end nodes such that there are $2m$ different start and end nodes that are inputs to the problem. A solution to the problem is an assignment of each vehicle to a path p, which is a sequence of distinct nodes $n_1 \ldots n_k$, where n_1 and n_k are the start and end nodes of the vehicle, respectively.

Apart from path constraints, vehicle routing problems may include additional contractions, such as dimension constraints that represent permissible values for various physical parameters (e.g., weight, volume, number of items, distance, time, etc.). Such constraints can be associated with nodes, paths, or vehicles involved in the problem. For instance, each node n_i may have a maximum permissible cumulative interval for a parameter, $Cumul(n_i)$; each pair of nodes may have a permissible transition interval for another parameter, $Transit(n_i, n_{i+1})$; each path p may have a limited capacity for a parameter, $Capacity(p)$; etc. One or both bounds of intervals for a dimensional constraint may be infinite. A permissible solution to the vehicle routing problem requires computing the values for the dimensional parameters for nodes, node pairs, paths, and vehicles as appropriate and ensuring that the values are within permissible bounds.

The routing for all vehicles within a fleet under the given path and dimensional constraints can be determined via various approaches, such as local search. Local search is typically split into at least two phases: construction and improvement, respectively. An initial routing solution is obtained in the construction phase and iteratively improved in the subsequent improvement phase. For instance, the initial candidate solution candidate solution for each path $v_1 \ldots v_m$ can be represented as sequences of chains:

$$v_1 \ldots v_m = (v^1{}_1 v^1{}_2 \ldots v^1{}_{m1})(v^2{}_1 v^2{}_2 \ldots v^2{}_{m2}) \ldots (v^n{}_1 v^n{}_2 \ldots v^n{}_{mn}) = c_1 c_2 \ldots c_n$$

where each chain $c_j = v^j{}_1 \ldots v^j{}_{mj}$ is a subpath of some path of the solution.

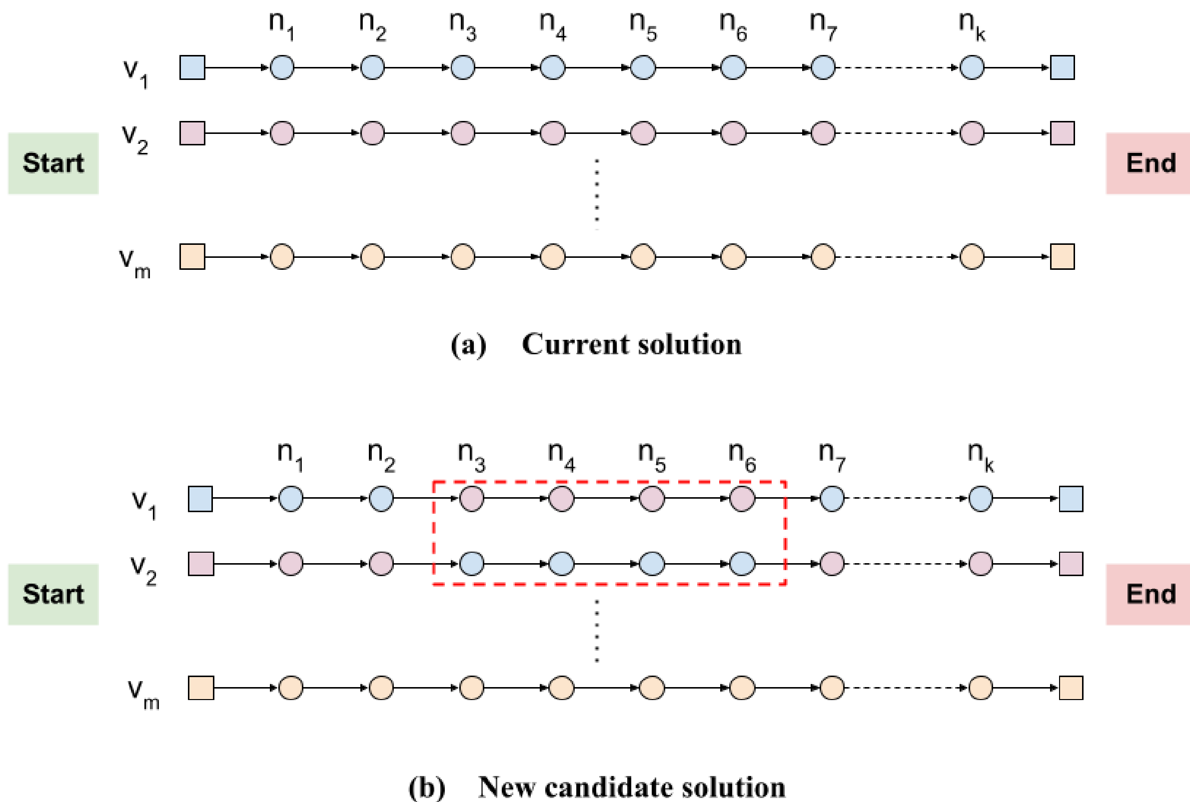

Fig. 1: Generating a new candidate solution with small changes to a current solution

As Fig. 1 shows, each iteration in the improvement phase involves generating a new candidate solution with a small change to the current solution. Fig. 1(a) shows a current solution

that depicts the nodes $n_1 \ldots n_k$ in the respective route for each of the vehicles $v_1 \ldots v_m$ in a fleet. The new candidate solution shown in Fig. 1(b) differs from the current solution of Fig. 1(a) only in terms of four of the nodes being swapped between v_1 *and* v_2 (as indicated by the dotted red rectangle), with the rest of the routing remaining identical to the current solution. Therefore, each new solution can be represented in terms of the set of paths that differ from the current candidate solution, with each changed path in turn described as a sequence of chains as above. If the new solution satisfies the routing constraints better than the current solution, it can replace the current solution.

Typically, most of the new solutions generated during the improvement phase do not satisfy the constraints and/or improve the current solution. As a result, there are usually 1,000 to 1,000,000 times more checks with the current solution that are discarded compared to the ones that result in the current solution being replaced with a new solution.

A similar iterative process of checking and replacing the current solution with a more optimal solution may be employed to find the initial solution during the construction phase. Employing the approach during the construction phase results in several orders of magnitudes more comparisons without solution replacement than those in which a new solution replaces an existing one, as in the improvement phase. However, the difference between the two is typically substantially less than that in the improvement phase.

Given the large number of comparisons among solutions that are involved in the application of local search for determining optimal routing solutions, the performance of the process is dependent on the time O(p) taken for performing each comparison, where p is the total length of the paths that are changed from the current solution to form the new solution. For some families of routing problems, the performance of performing an O(p) time procedure for each

comparison might be acceptable. However, the approach does not scale when routing problems involve paths that have hundreds of nodes.

Given that the new solutions being compared with the current solutions are small variations of the current solution, the complexity of performing future comparisons can be improved by performing relevant pre-computation whenever a current solution is replaced with a current one. A common approach is to pre-compute the relevant characteristics (e.g., travel time) of all subpaths involved in the new solution with which the current solution is replaced. Such pre-computation can typically be performed in $O(p^2)$ time and space, with the characteristics of all subpaths stored in a matrix indexed by the start and end of the subpath.

DESCRIPTION

This disclosure describes techniques to perform efficient pre-computations to check constraint satisfaction for new candidate solutions to routing problems that are generated with small variations from a current solution that is known to respect the required constraints. The new candidate solutions can be described by the chains of their paths that are changed in comparison to the current solution.

A general algorithm that can run in linear time in the number of chains of the path can be applied to propagate the given constraints from the first node of the first chain to the last node of the last chain. For each subproblem connected respectively to each constraint, a query scheme that can run in linear time can be used to propagate the given constraints from the first to the last nodes of a specific chain. Assuming O(1) time complexity for comparing relevant values for each change between a new candidate solution and the current solution, the algorithm and associated query scheme enable pre-computations for a new candidate solution to be performed

with O(p log p) complexity, where *p* is the total length of all paths that changed in the new solution compared to those in the current solution.

For instance, the satisfaction of the cumulative, transit, and capacity dimensional constraints for each of the chains within the solution being examined can be checked by iteratively applying the general algorithm as follows:

```
feasible_interval = Cumul(start(path)) ∩ Capacity(p);
previous_node = start(path);
for chain in path:
      feasible_interval += Transit(previous_node, chain[0]);
      feasible_interval ∩= Cumul(chain[0]) ∩ Capacity(path);
      feasible_interval ∩= FirstNodeCapacityIntersection(chain)
      if feasible_interval is empty: return Infeasible

      feasible_interval += TransitSum(chain)
      feasible_interval ∩= PathCapacityIntersection(p, chain)
      feasible_interval ∩= LastNodeCapacityIntersection(chain)
      if feasible_interval is empty: return Infeasible

      previous_node = chain[-1]
return Feasible
```

Specifically, the algorithm involves preserving the feasible interval of the first and last nodes of the current chain and returning `Feasible` or `Infeasible` depending on whether the given constraints can be satisfied or not, respectively.

To deal with infinities the algorithm can employ extended interval structure that contains four fields: `min`, `max`, `num_pos_inf`, `num_neg_inf`, with the first two being finite real value and the latter two being nonnegative integers. The relevant checking operations can then be performed in O(1) time based on intersection, addition, and emptiness tests defined as follows:

```
I is empty iff I.num_pos_inf == I.num_neg_inf == 0 and I.min > I.max.

a ∩ b = c, with fields
      c.min = max(a.min if a.num_neg_inf = 0 else -inf,
            b.min if b.num_neg_inf = 0 else -inf);
      if c.min = -inf, c.min = 0;
      c.max = min(a.max if a.num_pos_inf = 0 else +inf,
            b.max if b.num_pos_inf = 0 else +inf);
      if c.max == +inf, c.max = 0;
      c.num_neg_inf = min(a.num_neg_inf, b.num_neg_inf)
      c.num_pos_inf = min(a.num_pos_inf, b.num_pos_inf)

a + b = c, with
      c.min = a.min + b.min
      c.max = a.max + b.max
      c.num_neg_inf = a.num_neg_inf + b.num_neg_inf
      c.num_pos_inf = a.num_pos_inf + b.num_pos_inf

FromTo(a, b) = c, with
      c.min = b.min - a.min
      c.max = b.max - a.max
      c.num_neg_inf = b.num_neg_inf - a.num_neg_inf
      c.num_pos_inf = b.num_pos_inf - a.num_pos_inf
```

The query schemes associated with the algorithm can be similarly computed in O(1) time, thus ensuring that the entire procedure to check for constraint satisfaction can run in time that is linear in terms of the number of chains of the path. For instance, satisfaction of constraints related to Transit and Capacity as mentioned above can be checked by computing relevant values via corresponding functions as illustrated by the following examples:

1. **TransitSum**: The TransitSum function is a well-known range sum query scheme that returns the sum of Transit(n_i, n_{i+1}) for i in [1, c) when applied on a chain $n_1 \ldots n_c$. The values sum_transits of a path $n_1 \ldots n_k$ are pre-computed with:

   sum_transits(n_i) = 0 if i = k; OR

   sum_transits(n_i) = transition(n_i, n_{i+1}) + sum_transits(n_{i+1}) if i ≉ k

Information about the first and last node of a chain is sufficient for computing the answer for each subsequent query on the chain. Therefore, TransitSum($n_1 \dots n_c$) returns FromTo(sum_transits(n_c), sum_transits(n_1)). Such an extension of the range sum query scheme can be used to deal correctly with infinities.

2. **PathCapacityIntersection**: PathCapacityIntersection(p, $n_1 \dots n_c$) can be defined as $\bigcap_{i \text{ in } [1,c]}$ Capacity(p) + TransitSum($n_i \dots n_c$). The i-th term of this intersection is the interval of feasible values of cumul(n_c) based on the path capacity constraint on n_i and the transition constraints between nodes n_i and n_c. The commutation properties of sum and intersection can be used to compute the equivalent efficiently:

$$\text{Capacity(p)} + \text{FromTo(sum_transits}(n_c), \bigcap_{i \text{ in } [1,c]} \text{sum_transits}(n_i))$$

The above computation enables decoupling the path capacity of the chain in the current solution from that of the candidate. The intersection $\bigcap_{i \text{ in } [1,c]}$ sum_transits(n_i) can be computed efficiently by adapting the conventional range max query scheme with O(n log n) time and space pre-computation and O(1) time query such that is takes the form of a range intersection query on extended intervals. When the current solution is replaced by a more optimal candidate solution, the values for transit_sum_riq are pre-computed for each changed path $n_1 \dots n_k$ as transit_sum_riq[0][n_i] = sum_transits[n_i] with layer = 1 .. floor($\log_2$(k)), w = 2^{layer}, for all i:

transit_sum_riq[layer][n_i] = transit_sum_riq[layer-1][n_i] ∩ transit_sum_riq[layer-1][n_{i+w}]

if $i \leq k - w$; OR

transit_sum_riq[layer][n_i] = transit_sum_riq[layer-1][n_i] if $i > k - w$

When checking for constraint satisfaction, the query $\cap_{i \text{ in } [a,b]}$ sum_transits(n_i) when a < b can be answered by computing layer = floor($\log_2$(b - a)), w = 2^{layer} and returning transit_sum_riq[layer][n_a] ∩ transit_sum_riq[layer][n_{b-w+1}].

3. **NodeCapacityIntersection**: LastNodeCapacityIntersection($n_i \ldots n_j$) can be defined as $\cap_{i \text{ in } [1,c]}$ Cumul(ni) + TransitSum($n_i \ldots n_c$). The i-th term of this intersection is the interval of feasible values of cumul(n_c) based on the node capacity constraint on n_i and transition constraints between n_i and n_c. When the current solution is replaced by a more optimal candidate solution, the values for node_capacity_riq are pre-computed for each changed path $n_1 \ldots n_k$ as node_capacity_riq[0][n_i] = Cumul(n_i) with layer = 1 .. floor($\log_2$(k)), w = 2^{layer}, for all i:

   node_capacity_riq[layer][i] = node_capacity_riq[layer-1][i - w] + TransitSum($n_{i-w} \ldots n_i$) ∩ node_capacity_riq[layer-1][i] if i ≤ w; OR

   node_capacity_riq[layer][i] = node_capacity_riq[layer-1][i] if i > w

When checking for constraint satisfaction, the query LastNodeCapacityIntersection($n_a \ldots n_b$) can be answered by computing layer = floor($\log_2$(b - a)), w = 2^{layer} and returning node_capacity_riq[layer][a+w-1] + TransitSum(n_{a+w-1}, n_b) ∩ node_capacity_riq[layer][b]. FirstNodeCapacityIntersection is defined and computed in a symmetrical manner.

As the above examples illustrate, the query schemes can be used not only to check for constraint satisfaction, but also to compute the minimum feasible value of *cumul* at the end of a given path. Moreover, the same schemes with mirror values can compute the maximum feasible value of *cumul* at the beginning of the path. These two values can in turn be used to compute with the same time and space complexity characteristics the values of several parameters related

to paths, such as the cost of being early at the beginning of a path, the cost of being late at the end of a path, the minimum span of a path, the cost of the minimum span, overtime cost, etc.

The techniques described in this disclosure can be provided as a basic building block via any suitable mechanism, such as a code library, an application programming interface (API), etc. The techniques can be implemented when using local search to determine solutions to routing problems within any relevant application, platform, or service, such as digital maps, vehicle fleet management, etc. Further, the techniques can be used for performing relevant pre-computations when solving other types of problems, such as scheduling, as well as within other types of problem-solving approaches, such as evolutionary algorithms.

Assuming O(1) time complexity to check for constraint satisfaction per change with *p* being the total length of all paths changed between a known solution and a new candidate solution, current approaches provide $O(p^2)$ time and space complexity when replacing the current solution with a new one. Implementation of the techniques can reduce the time and space complexity to O(p log p), thus significantly saving the time and resources required to determine optimal routes, especially in scenarios that involve long routes and hundreds to thousands of nodes with numerous constraints on various factors, such as time, capacity, etc.

CONCLUSION

This disclosure describes techniques to perform efficient pre-computations to reduce the time and space complexity to O(p log p), thus significantly saving time and resources. The pre-computations check constraint satisfaction for new candidate solutions generated with small variations from a current solution that is known to respect the constraints. The new candidate solutions are described by the chains of their paths that are changed in comparison to the current solution. The checking and pre-computations are performed with a general algorithm and

associated query scheme for each subproblem, and run in linear time in the number of chains. As a result, the time and space complexity of the operation is reduced to O(p log p), where p is the total length of all changed paths in the new solution.

REFERENCES

1. Kindervater, Gerard AP, and Martin WP Savelsbergh. "10. Vehicle routing: handling edge exchanges." in *Local search in combinatorial optimization*, pp. 337-360. Princeton University Press, 2018.
2. “OR-Tools | Google Developers” available online at https://developers.google.com/optimization accessed January 23, 2023.