DM204 – Spring 2011

Scheduling, Timetabling and Routing

Lecture 14
Vehicle Routing
Local Search based Metaheuristics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Improvement Heuristics

2. Metaheuristics

3. Constraint Programming for VRP

Course Overview

- ✔ Scheduling
  - ✔ Classification
  - ✔ Complexity issues
  - ✔ Single Machine
  - ✔ Parallel Machine
  - ✔ Flow Shop and Job Shop
  - ✔ Resource Constrained Project Scheduling Model

- Timetabling
  - ✔ Sport Timetabling
  - ✔ Reservations and Education
  - ✔ University Timetabling
  - ✔ Crew Scheduling
  - ✔ Public Transports

- Vechicle Routing
  - ✔ MIP Approaches
  - Construction Heuristics
  - Local Search Algorithms

Outline

1. Improvement Heuristics

2. Metaheuristics

3. Constraint Programming for VRP

Outline

Local Search for CVRP and VRPTW

- Neighborhood structures:
  - Intra-route: 2-opt, 3-opt, Lin-Kernighan (not very well suited), Or-opt (2H-opt)
  - Inter-routes: λ-interchange, relocate, exchange, cross, 2-opt*, b-cyclic k-transfer (ejection chains), GENI

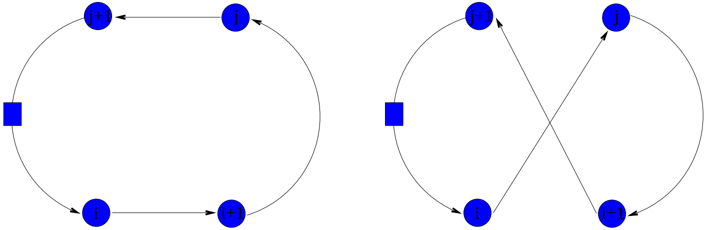
Local Search for CVRP and VRPTW

- Neighborhood structures:
  - Intra-route: 2-opt, 3-opt, Lin-Kernighan (not very well suited), Or-opt (2H-opt)
  - Inter-routes: λ-interchange, relocate, exchange, cross, 2-opt*, b-cyclic k-transfer (ejection chains), GENI
- Solution representation and data structures
  - They depend on the neighborhood.
  - It can be advantageous to change them from one stage to another of the heuristic

Intra-route Neighborhoods

2-opt

$$\{i, i+1\}\{j, j+1\} \longrightarrow \{i, j\}\{i+1, j+1\}$$

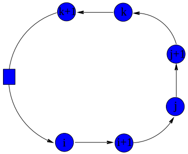


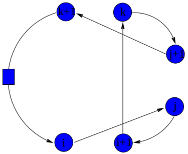
$O(n^2)$ possible exchanges
One path is reversed

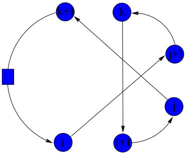
Intra-route Neighborhoods

3-opt

$$\{i, i+1\}\{j, j+1\}\{k, k+1\} \longrightarrow \ldots$$





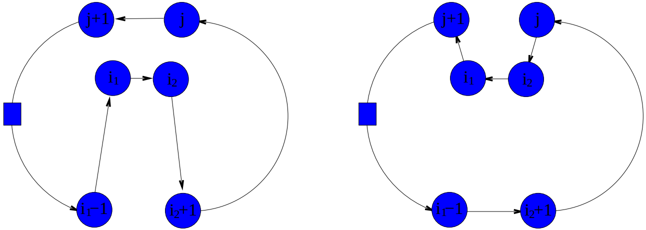


$O(n^3)$ possible exchanges
Paths can be reversed

Intra-route Neighborhoods

Or-opt [Or (1976)]

$$\{i_1-1, i_1\}\{i_2, i_2+1\}\{j, j+1\} \longrightarrow \{i_1-1, i_2+1\}\{j, i_1\}\{i_2, j+1\}$$



sequences of one, two, three consecutive vertices relocated

$O(n^2)$ possible exchanges — No paths reversed

Inter-route Neighborhoods

[Savelsbergh, ORSA (1992)]

Figure 6. The exchange neighborhood.

Inter-route Neighborhoods

[Savelsbergh, ORSA (1992)]

Figure 5. The relocate neighborhood.

Inter-route Neighborhoods

[Savelsbergh, ORSA (1992)]

Figure 7. The cross neighborhood.

GENI: generalized insertion [Gendreau, Hertz, Laporte, Oper. Res. (1992)]

- select the insertion restricted to the neighborhood of the vertex to be added (not necessarily between consecutive vertices)
- perform the best 3- or 4-opt restricted to reconnecting arc links that are close to one another.

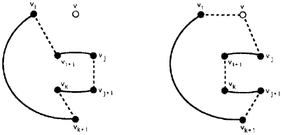


Figure 1. Type I insertion of vertex v between v_i and v_j.

GENI: generalized insertion [Gendreau, Hertz, Laporte, Oper. Res. (1992)]

- select the insertion restricted to the neighborhood of the vertex to be added (not necessarily between consecutive vertices)
- perform the best 3- or 4-opt restricted to reconnecting arc links that are close to one another.

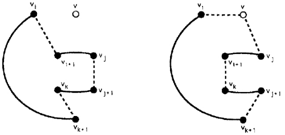


Figure 1. Type I insertion of vertex v between v_i and v_j.

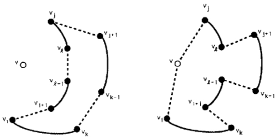


Figure 2. Type II insertion of vertex v between v_i and v_j.

Efficient Implementation
Intra-route

Time windows: Feasibility check

In TSP verifying k-optimality requires $O(n^k)$ time
In TSPTW feasibility has to be tested then $O(n^{k+1})$ time

Efficient Implementation
Intra-route

Time windows: Feasibility check

In TSP verifying k-optimality requires $O(n^k)$ time
In TSPTW feasibility has to be tested then $O(n^{k+1})$ time

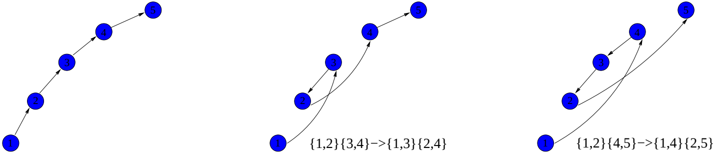
(Savelsbergh 1985) shows how to verify constraints in constant time
Search strategy + Global variables

$\Downarrow$

$O(n^k)$ for k-optimality in TSPTW

Search Strategy

- Lexicographic search, for 2-exchange:
  - $i = 1, 2, \ldots, n-2$ (outer loop)
  - $j = i+2, i+3, \ldots, n$ (inner loop)



Previous path is expanded by the edge $\{j-1, j\}$

Global variables (auxiliary data structure)

- Maintain auxiliary data such that it is possible to:
  - handle single move in constant time
  - update their values in constant time

Global variables (auxiliary data structure)

- Maintain auxiliary data such that it is possible to:
    - handle single move in constant time
    - update their values in constant time

Ex.: in case of time windows:

- total travel time of a path
- earliest departure time of a path
- latest arrival time of a path

Efficient Local Search

[Irnich (2008)] uniform model

Outline

1. Improvement Heuristics

2. Metaheuristics

3. Constraint Programming for VRP

Metaheuristics

Many and fancy examples, but first thing to try:

- Variable Neighborhood Search + Iterated greedy

Basic Variable Neighborhood Descent (BVND)

Procedure VND
input : $\mathcal{N}_k$, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for $\mathcal{N}_k$, $k = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
repeat
  $s' \leftarrow$ FindBestNeighbor(s,$\mathcal{N}_k$)
  if $g(s') < g(s)$ **then**
    $s \leftarrow s'$
    $k \leftarrow 1$
  else
    $k \leftarrow k + 1$
until $k = k_{max}$;

Variable Neighborhood Descent (VND)

Procedure VND
input : $\mathcal{N}_k$, $k = 1, 2, \ldots, k_{max}$, and an initial solution s
output: a local optimum s for $\mathcal{N}_k$, $k = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
repeat
  $s' \leftarrow$ IterativeImprovement(s,$\mathcal{N}_k$)
  if $g(s') < g(s)$ **then**
    $s \leftarrow s'$
    $k \leftarrow 1$
  else
    $k \leftarrow k + 1$
until $k = k_{max}$;

- Final solution is locally optimal w.r.t. all neighborhoods
- First improvement may be applied instead of best improvement
- Typically, order neighborhoods from smallest to largest
- If iterative improvement algorithms II_k, $k = 1, \ldots, k_{max}$ are available as black-box procedures:
  - order black-boxes
  - apply them in the given order
  - possibly iterate starting from the first one
  - order chosen by: *solution quality* and *speed*

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

However,

- Designing a local search algorithm is an engineering process in which learnings from other courses in CS might become important.

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

However,

- Designing a local search algorithm is an engineering process in which learnings from other courses in CS might become important.

- It is important to make such algorithms as much efficient as possible.

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

However,

- Designing a local search algorithm is an engineering process in which learnings from other courses in CS might become important.
- It is important to make such algorithms as much efficient as possible.
- Many choices are to be taken (search strategy, order, auxiliary data structures, etc.) and they may interact with instance features. Often a trade-off between examination cost and solution quality must be decided.

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

However,

- Designing a local search algorithm is an engineering process in which learnings from other courses in CS might become important.
- It is important to make such algorithms as much efficient as possible.
- Many choices are to be taken (search strategy, order, auxiliary data structures, etc.) and they may interact with instance features. Often a trade-off between examination cost and solution quality must be decided.
- The assessment is conducted through:

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

However,

- Designing a local search algorithm is an engineering process in which learnings from other courses in CS might become important.

- It is important to make such algorithms as much efficient as possible.

- Many choices are to be taken (search strategy, order, auxiliary data structures, etc.) and they may interact with instance features. Often a trade-off between examination cost and solution quality must be decided.

- The assessment is conducted through:
  - analytical analysis (computational complexity)

General recommendation: use a combination of 2-opt* + or-opt
[Potvin, Rousseau, (1995)]

However,

- Designing a local search algorithm is an engineering process in which learnings from other courses in CS might become important.

- It is important to make such algorithms as much efficient as possible.

- Many choices are to be taken (search strategy, order, auxiliary data structures, etc.) and they may interact with instance features. Often a trade-off between examination cost and solution quality must be decided.

- The assessment is conducted through:
  - analytical analysis (computational complexity)
  - experimental analysis

Table 5.6. *The effect of 3-opt on the Clarke and Wright algorithm.*

	Sequential				Parallel			
Problem	No 3-opt[1]	+ 3-opt FI[2]	+ 3-opt BI[3]	K^4	No 3-opt[5]	+ 3-opt FI[6]	+ 3-opt BI[7]	K^8
E051-05e	625.56	624.20	624.20	5	584.64	578.56	578.56	6
E076-10e	1005.25	991.94	991.94	10	900.26	888.04	888.04	10
E101-08e	982.48	980.93	980.93	8	886.83	878.70	878.70	8
E101-10c	939.99	930.78	928.64	10	833.51	824.42	824.42	10
E121-07c	1291.33	1232.90	1237.26	7	1071.07	1049.43	1048.53	7
E151-12c	1299.39	1270.34	1270.34	12	1133.43	1128.24	1128.24	12
E200-17c	1708.00	1667.65	1669.74	16	1395.74	1386.84	1386.84	17
D051-06c	670.01	663.59	663.59	6	618.40	616.66	616.66	6
D076-11c	989.42	988.74	988.74	12	975.46	974.79	974.79	12
D101-09c	1054.70	1046.69	1046.69	10	973.94	968.73	968.73	9
D101-11c	952.53	943.79	943.79	11	875.75	868.50	868.50	11
D121-11c	1646.60	1638.39	1637.07	11	1596.72	1587.93	1587.93	11
D151-14c	1383.87	1374.15	1374.15	15	1287.64	1284.63	1284.63	15
D200-18c	1671.29	1652.58	1652.58	20	1538.66	1523.24	1521.94	19

[1] Sequential savings.

[2] Sequential savings + 3-opt and first improvement.

[3] Sequential savings + 3-opt and best improvement.

[4] Sequential savings: number of vehicles in solution.

[5] Parallel savings.

[6] Parallel savings + 3-opt and first improvement.

[7] Parallel savings + 3-opt and best improvement.

[8] Parallel savings: number of vehicles in solution.

Table 5.6. *The effect of 3-opt on the Clarke and Wright algorithm.*

	Sequential				Parallel			
Problem	No 3-opt[1]	+ 3-opt FI[2]	+ 3-opt BI[3]	K^4	No 3-opt[5]	+ 3-opt FI[6]	+ 3-opt BI[7]	K^8
E051-05e	625.56	624.20	624.20	5	584.64	578.56	578.56	6
E076-10e	1005.25	991.94	991.94	10	900.26	888.04	888.04	10
E101-08e	982.48	980.93	980.93	8	886.83	878.70	878.70	8
E101-10c	939.99	930.78	928.64	10	833.51	824.42	824.42	10
E121-07c	1291.33	1232.90	1237.26	7	1071.07	1049.43	1048.53	7
E151-12c	1299.39	1270.34	1270.34	12	1133.43	1128.24	1128.24	12
E200-17c	1708.00	1667.65	1669.74	16	1395.74	1386.84	1386.84	17
D051-06c	670.01	663.59	663.59	6	618.40	616.66	616.66	6
D076-11c	989.42	988.74	988.74	12	975.46	974.79	974.79	12
D101-09c	1054.70	1046.69	1046.69	10	973.94	968.73	968.73	9
D101-11c	952.53	943.79	943.79	11	875.75	868.50	868.50	11
D121-11c	1646.60	1638.39	1637.07	11	1596.72	1587.93	1587.93	11
D151-14c	1383.87	1374.15	1374.15	15	1287.64	1284.63	1284.63	15
D200-18c	1671.29	1652.58	1652.58	20	1538.66	1523.24	1521.94	19

[1] Sequential savings.

[2] Sequential savings + 3-opt and first improvement.

[3] Sequential savings + 3-opt and best improvement.

[4] Sequential savings: number of vehicles in solution.

[5] Parallel savings.

[6] Parallel savings + 3-opt and first improvement.

[7] Parallel savings + 3-opt and best improvement.

[8] Parallel savings: number of vehicles in solution.

What is best?

Iterated Greedy

Key idea: use the VRP cosntruction heuristics

- alternation of Construction and Deconstruction phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):

determine initial candidate solution s
while termination criterion is not satisfied **do**
    $r := s$
    greedily destruct part of s
    greedily reconstruct the missing part of s
    apply subsidiary iterative improvement procedure (eg, VNS)
    based on acceptance criterion,
        keep s or revert to $s := r$

In the literature, the overall heuristic idea received different names:

- Removal and reinsertion
- Ruin and repair
- Iterated greedy
- Fix and re-optimize

Removal procedures

Remove some related customers
(their re-insertion is likely to change something, if independent would be reinserted in same place)

Relatedness measure r_{ij}

- belong to same route
- geographical
- temporal and load based
- cluster removal
- history based

Dispersion sub-problem:
choose q customers to remove with minimal r_{ij}

$$\begin{aligned} \min \quad & \textstyle\sum_{ij} r_{ij} x_i x_j \\ & \textstyle\sum_j x_j = q \\ & x_j \in \{0, 1\} \end{aligned}$$

Heuristic stochastic procedure:

- select i at random and find j that minimizes r_{ij}
- Kruskal like, plus some randomization
- history based
- random

Reinsertion procedures

- Greedy (cheapest insertion)
- Max regret:

  Δf_i^q due to insert i into its best position in its q^{th} best route

  $i = \arg\max(\Delta f_i^2 - \Delta f_i^1)$
- Constraint programming (max 20 costumers)

Advantages of remove-reinsert procedure with many side constraints:

- the search space in local search may become disconnected
- it is easier to implement feasibility checks
- no need of computing delta functions in the objective function

Further ideas

- Adaptive removal: start by removing 1 pair and increase after a certain number of iterations
- use of roulette wheel to decide which removal and reinsertion heuristic to use (π past contribution)

$$p_i = \frac{\pi_i}{\sum \pi_i} \qquad \text{for each heuristic } i$$

- SA as accepting criterion after each reconstruction

Outline

Performance of exact methods

Current limits of exact methods [Ropke, Pisinger (2007)]:

CVRP: up to 135 customers by branch and cut and price

VRPTW: 50 customers (but 1000 customers can be solved if the instance has some structure)

CP can handle easily side constraints but hardly solve VRPs with more than 30 customers.

Large Neighborhood Search

Other approach with CP: [Shaw, 1998]

- Use an over all local search scheme

Large Neighborhood Search

Other approach with CP: [Shaw, 1998]

- Use an over all local search scheme
- Moves change a large portion of the solution

Large Neighborhood Search

Other approach with CP: [Shaw, 1998]

- Use an over all local search scheme
- Moves change a large portion of the solution
- CP system is used in the exploration of such moves.

Large Neighborhood Search

Other approach with CP: [Shaw, 1998]

- Use an over all local search scheme
- Moves change a large portion of the solution
- CP system is used in the exploration of such moves.
- CP used to check the validity of moves and determine the values of constrained variables

Large Neighborhood Search

Other approach with CP: [Shaw, 1998]

- Use an over all local search scheme
- Moves change a large portion of the solution
- CP system is used in the exploration of such moves.
- CP used to check the validity of moves and determine the values of constrained variables
- As a part of checking, constraint propagation takes place. Later, iterative improvement can take advantage of the reduced domains to speed up search by performing fast legality checks.

Solution representation:

- Handled by local search:
  Next pointers: A variable n_i for every customer i representing the next visit performed by the same vehicle

  $$n_i \in N \cup S \cup E$$

  where $S = \bigcup S_k$ and $E = \bigcup E_k$ are additional visits for each vehicle k marking the start and the end of the route for vehicle k

- Handled by the CP system: time and capacity variables.

Insertion

by CP:

- constraint propagation rules: time windows, load and bound considerations
- search heuristic most constrained variable + least constrained valued (for each v find cheapest insertion and choose v with largest such cost)
- Complete search: ok for 15 visits (25 for VRPTW) but with heavy tails
- Limited discrepancy search

[Shaw, 1998]

```
Reinsert(RoutingPlan plan, VisitSet visits, integer discrep)
    if |visits| = 0 then
        if Cost(plan) < Cost(bestplan) then
            bestplan := plan
        end if
    else
        Visit v := ChooseFarthestVisit(visits)
        integer i := 0
        for p in rankedPositions(v) and i ≤ discrep do
            Store(plan) // Preserve plan on stack
            InsertVisit(plan, v, p)
            Reinsert(plan, visits - v, discrep - i)
            Restore(plan) // Restore plan from stack
            i := i + 1
        end for
    end if
end Reinsert
```