Paper SAS1758-2018

Using SAS/OR® to Optimize Scheduling and Routing of Service Vehicles

Rob Pratt, SAS Institute Inc.

ABSTRACT

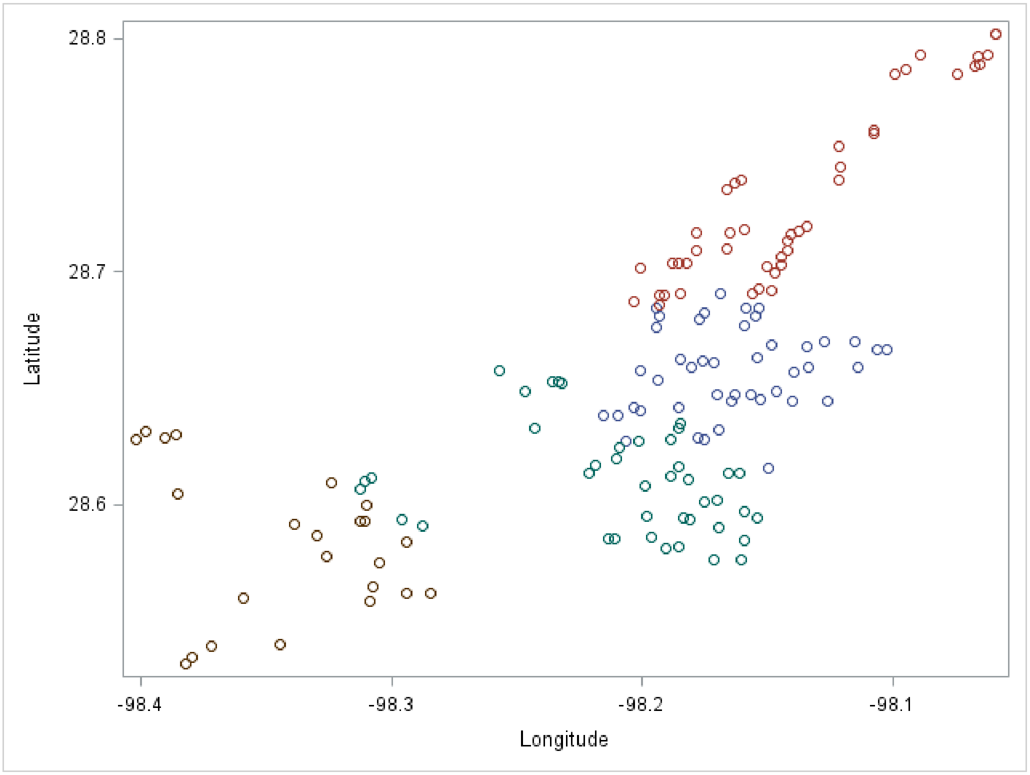
An oil company has a set of wells and a set of well operators. Each well has an established amount of time required for servicing. For a given planning horizon, the company wants to determine which operator should perform service on which wells, on which days, and in which order, with the goal of minimizing service time plus travel time. A frequency constraint for each well restricts the number of days between visits. The solution approach that is presented in this paper uses several features in the OPTMODEL procedure in SAS/OR® software. A simple idea and a small change in the code reduced the run time from one hour to one minute.

INTRODUCTION

The problem that this paper addresses comes from Paal Navestad, a data scientist and SAS® user at ConocoPhillips, the world's largest independent exploration and production company. ConocoPhillips has approximately 850 oil wells in South Texas, with 20 well operators, or service technicians, who perform regular maintenance on them. Over a planning horizon of at least 10 days, the problem is to determine which operator should visit which wells on which days and in which order, so the problem involves both scheduling and routing. A frequency requirement for each well restricts the number of days between visits, and there is an upper limit on the service time plus travel time per operator per day. The goal is to minimize the total operator time over the planning horizon. This problem is a variant of what is known in the optimization literature as a periodic vehicle routing problem (Campbell and Wilson 2014).

Figure 1 shows the locations of wells that are assigned to four operators. The color of each well indicates which operator is currently assigned to it. You can see that the assignments are clustered geographically. The rest of this paper focuses on only the red set of wells in the northeast region.

Figure 1 Well Locations for Four Operators



ConocoPhillips does not want to reassign the operators to different wells very often, so as an initial simplification assume that the assignments are fixed. In that case, the problem decomposes into a separate problem for each operator.

The solution approach that this paper illustrates is common for vehicle routing problems. The main idea is to relax the connectivity requirements and dynamically generate constraints to enforce them as needed. This approach uses several features of the OPTMODEL procedure in SAS/OR:

- the mixed integer linear programming (MILP) solver
- the network solver (in particular, to find connected components and solve various instances of the traveling salesman problem)
- the COFOR loop, to solve independent problems concurrently
- the SUBMIT block, to call PROC SGPLOT from within PROC OPTMODEL
- the programming language functionality, to implement a customized algorithm that invokes these solvers

DATA

This section describes the input data for the problem. The number of days in the planning horizon is 10, and the upper limit on the operator time per day is 10 hours. The solution approach that is illustrated in this paper models the problem in terms of a network in which each well corresponds to a node and each pair of wells corresponds to an edge. For each well, the following inputs are required:

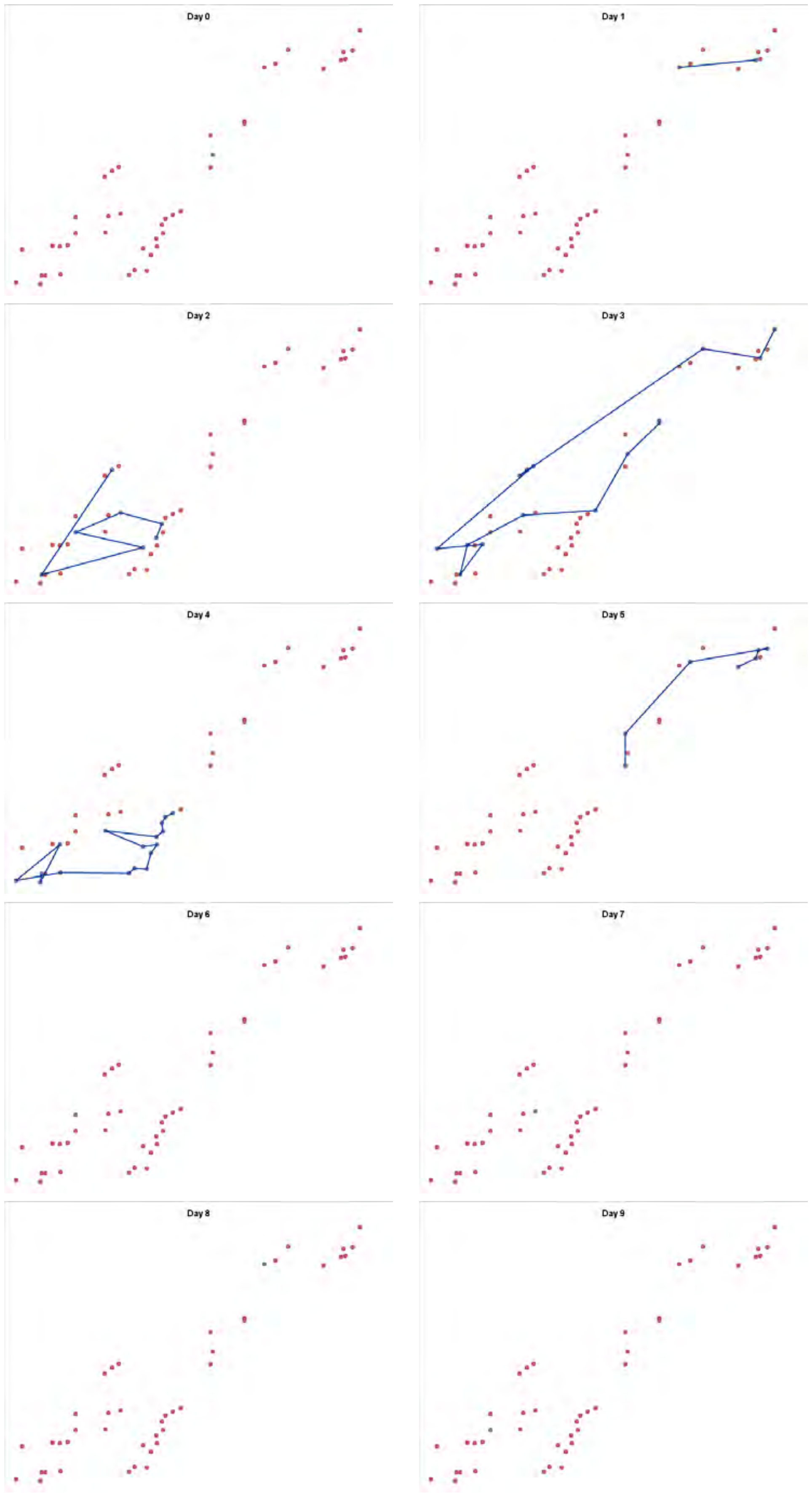
- which operator is currently assigned to it
- its location (only for plotting purposes)
- the time required to service the well each time the operator visits it
- the maximum number of days between visits
- the date of the last visit

The required input also includes the travel time between each pair of wells.

EXAMPLE SOLUTION

As a preview, Figure 2 shows a plot of a solution, which has a separate route for each day in the planning horizon. For each day, blue markers indicate wells that are visited, and red markers indicate wells that are not visited. Note that five of the days have only one well visited.

Figure 2 Example Solution



PROC OPTMODEL

This paper uses the OPTMODEL procedure, which contains an algebraic modeling language that enables you to define optimization problems by declaring decision variables, constraints, bounds, and objectives algebraically. It also supports numeric or character parameters, arrays, and index sets. As a SAS procedure, PROC OPTMODEL is closely integrated with the SAS programming environment. You can read and write as many data sets as you want. Like most modeling languages, the algebraic modeling language in PROC OPTMODEL supports separation between model and data so that you can solve other instances merely by changing the data and not changing the model. PROC OPTMODEL is interactive and provides direct access to the linear, mixed integer linear, quadratic, nonlinear, network, and constraint programming solvers. As the paper illustrates, you can use this procedure to write customized algorithms. PROC OPTMODEL includes almost all the standard SAS functions, and you can also write your own functions by using PROC FCMP. For more information, see the OPTMODEL procedure chapter in *SAS/OR User's Guide: Mathematical Programming*.

OPTIMIZATION MODEL

This section describes the optimization model in terms of decision variables, an objective function to be optimized, and constraints that the variables must satisfy.

VARIABLES

The following PROC OPTMODEL statements declare two sets of variables:

```
/* UseNode[i,d] = 1 if node i is visited on day d; 0 otherwise */
var UseNode {NODES, DAYS} binary;

/* UseEdge[i,j,d] = 1 if edge <i,j> is traversed on day d; 0 otherwise */
/* Can equal 2 if edge incident with depot */
var UseEdge {<i,j> in EDGES, DAYS} integer >= 0 <= (if depot in {i,j} then 2 else 1);
```

The first VAR statement declares a binary decision variable, **UseNode[i,d]**, that indicates whether node i is visited on day d. The **UseEdge** variable has a similar interpretation for the edges between pairs of nodes. As is common in vehicle routing problems, the network includes a dummy depot node in addition to the nodes for the actual wells. On each day, the operator must start and end at the depot, and the travel time to and from the depot is 0. If the value of a **UseEdge** variable is 2, the interpretation is that the operator leaves the depot, visits only one well, and returns to the depot.

The following IMPVAR statement declares an implicit variable, **TimePerDay**, that computes the service time plus travel time for a given day d as a linear combination of the **UseNode** and **UseEdge** variables:

```
/* implicit variable to be used in multiple places */
impvar TimePerDay {d in DAYS} =
   sum {i     in NODES} service_time[i]  * UseNode[i,d]
 + sum {<i,j> in EDGES} travel_time[i,j] * UseEdge[i,j,d];
```

This same expression appears in multiple places, so it is convenient to give it a name.

OBJECTIVE

The following MIN statement declares the objective to minimize the total time as a sum across all the days in the planning horizon:

```
/* objective */
min TotalTime = sum {d in DAYS} TimePerDay[d];
```

CONSTRAINTS

The following CON statement declares TwoMatching constraints that link the **UseNode** and **UseEdge** variables by forcing every node in the solution to use either zero edges (meaning that the node is not visited on that day) or two edges (in which case the operator enters the node along one edge and exits along another edge):

```
/* each node uses zero or two edges */
con TwoMatching {i in NODES, d in DAYS}:
   sum {<(i),j> in EDGES} UseEdge[i,j,d] + sum {<j,(i)> in EDGES} UseEdge[j,i,d]
 = 2 * UseNode[i,d];
```

The following TimeBudgetPerDay constraints put an upper limit on the implicit variable **TimePerDay**:

```
/* at most time_budget minutes per day */
con TimeBudgetPerDay {d in DAYS}:
   TimePerDay[d] <= &time_budget * UseNode[depot,d];
```

Here, the value of the macro variable *time_budget* is 600, the number of minutes in 10 hours, which is the upper limit on the operator time per day.

The following VisitDepot constraints enforce a logical condition that, if well i is visited on day d, then the depot must also be visited on that day d:

```
/* if operator visits any well, must visit depot */
con VisitDepot {i in NODES diff {depot}, d in DAYS}:
   UseNode[i,d] <= UseNode[depot,d];
```

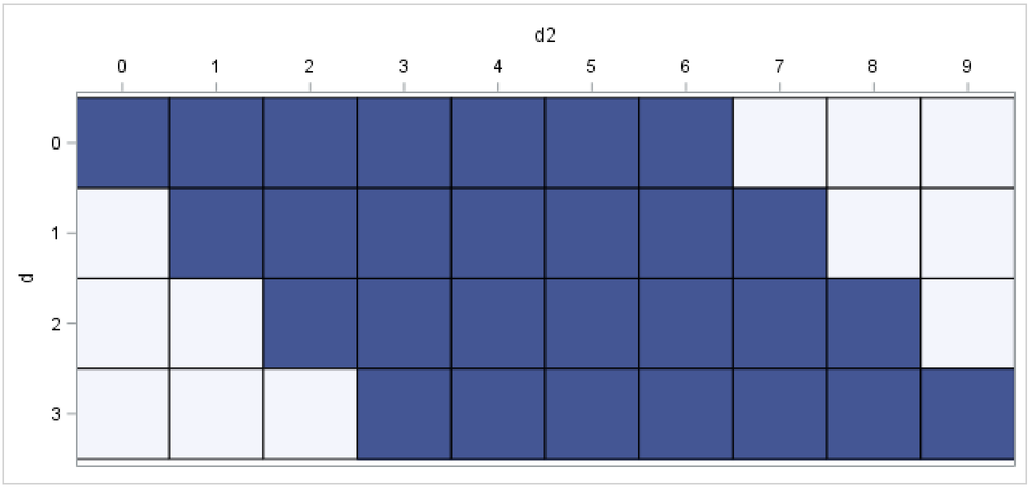
A less-than-or-equal constraint between these two binary variables is the way to model this if-then relationship linearly.

The following Cover constraints enforce the upper limit on the frequency between visits to each well:

```
/* cover each interval of consecutive days */
con Cover {i in NODES diff {depot}, d in DAYS: d+max_freq[i]-1 in DAYS}:
   sum {d2 in d..d+max_freq[i]-1} UseNode[i,d2] >= 1;
```

A sum of binary variables is constrained to be greater than or equal to 1, which means that at least one of them must equal 1. Figure 3 shows the days that are involved in each constraint if the maximum frequency of well i is 7: each consecutive 7-day interval within the 10-day planning horizon must be covered.

Figure 3 Cover Constraints

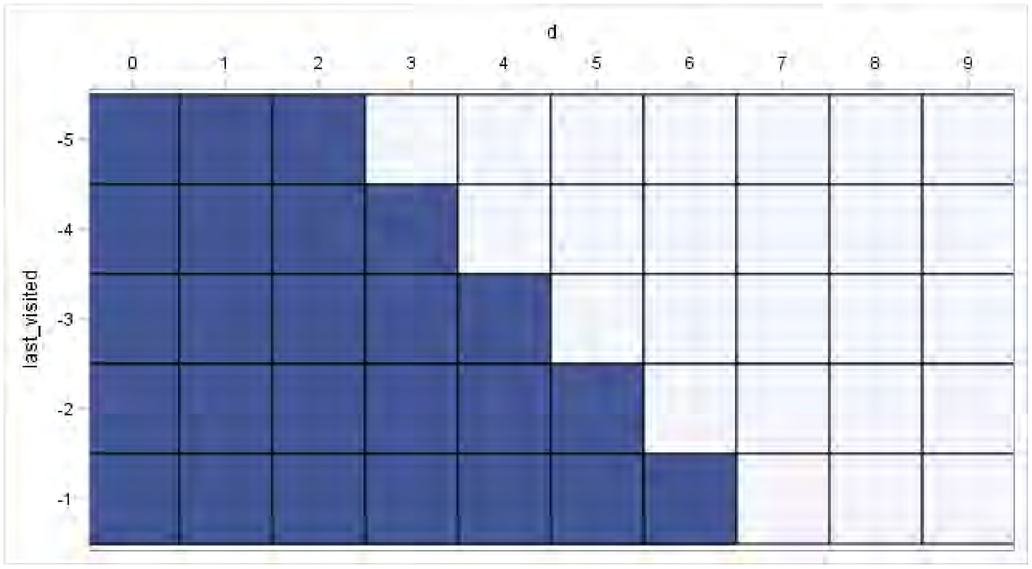


The following FirstVisit constraints are similar but enforce that the first visit to each well must be early enough in the planning horizon:

```
/* first visit to each well must be early enough */
con FirstVisit {i in NODES diff {depot}}:
   sum {d in DAYS inter last_visited[i]+1..last_visited[i]+max_freq[i]} UseNode[i,d] >= 1;
```

For example, if well i was last visited five days ago, then it must be visited again on day 0, 1, or 2 to satisfy the maximum frequency requirement, as illustrated in the first row of Figure 4.

Figure 4 FirstVisit Constraints



SUBTOUR ELIMINATION

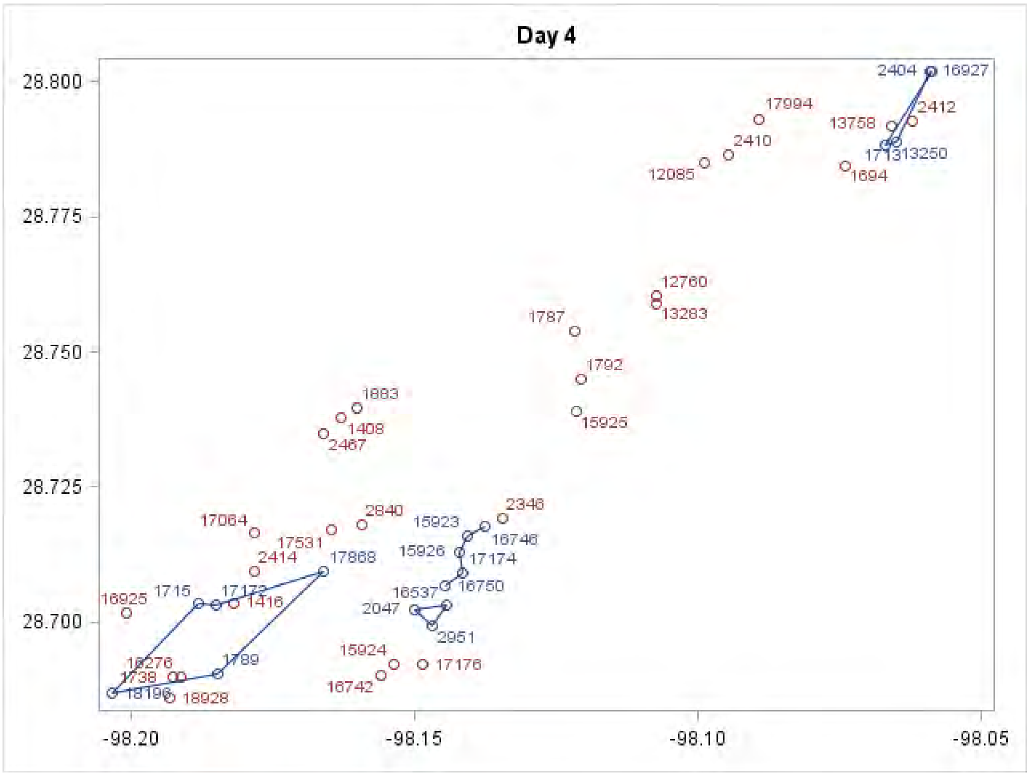
A *subtour* is a connected subgraph of nodes and edges for which the corresponding **UseNode** and **UseEdge** variables satisfy the TwoMatching constraints. As illustrated in Figure 2, a solution to the business problem consists of, for each day in the planning horizon, at most one subtour. If you solve the optimization problem that contains only the constraints declared so far, the resulting solution could instead contain multiple subtours per day, and any such solution is physically impossible to implement because no operator can instantaneously jump from one well to another without incurring any travel time. Because there are an exponential number of subtours that do not contain the depot, you should eliminate them by dynamically generating constraints as needed. The following statements declare the SubtourElimination constraints:

```
num num_subtours init 0;

/* subset of nodes not containing depot node */
set <str> SUBTOUR {1..num_subtours};

/* if node k in SUBTOUR[s] is used on day d,
   then must use at least two edges across partition induced by SUBTOUR[s] */
con SubtourElimination {s in 1..num_subtours, k in SUBTOUR[s], d in DAYS}:
   sum {i in NODES diff SUBTOUR[s], j in SUBTOUR[s]: <i,j> in EDGES} UseEdge[i,j,d]
 + sum {i in SUBTOUR[s], j in NODES diff SUBTOUR[s]: <i,j> in EDGES} UseEdge[i,j,d]
>= 2 * UseNode[k,d];
```

Initially, the number of eliminated subtours is 0, and a set is declared to contain the nodes in each subtour. The SubtourElimination constraints are declared only once and are automatically updated when *num_subtours* and *SUBTOUR[s]* change.

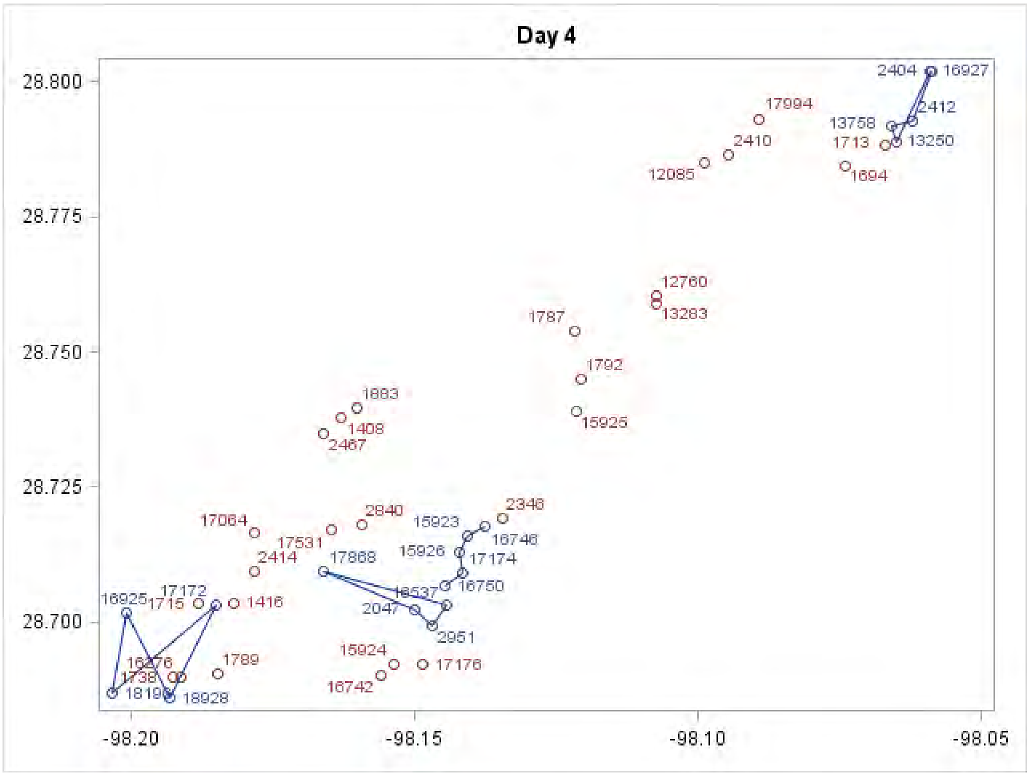
Figure 5 shows the MILP solution without any SubtourElimination constraints for one particular day (day 4).

Figure 5 Subtours



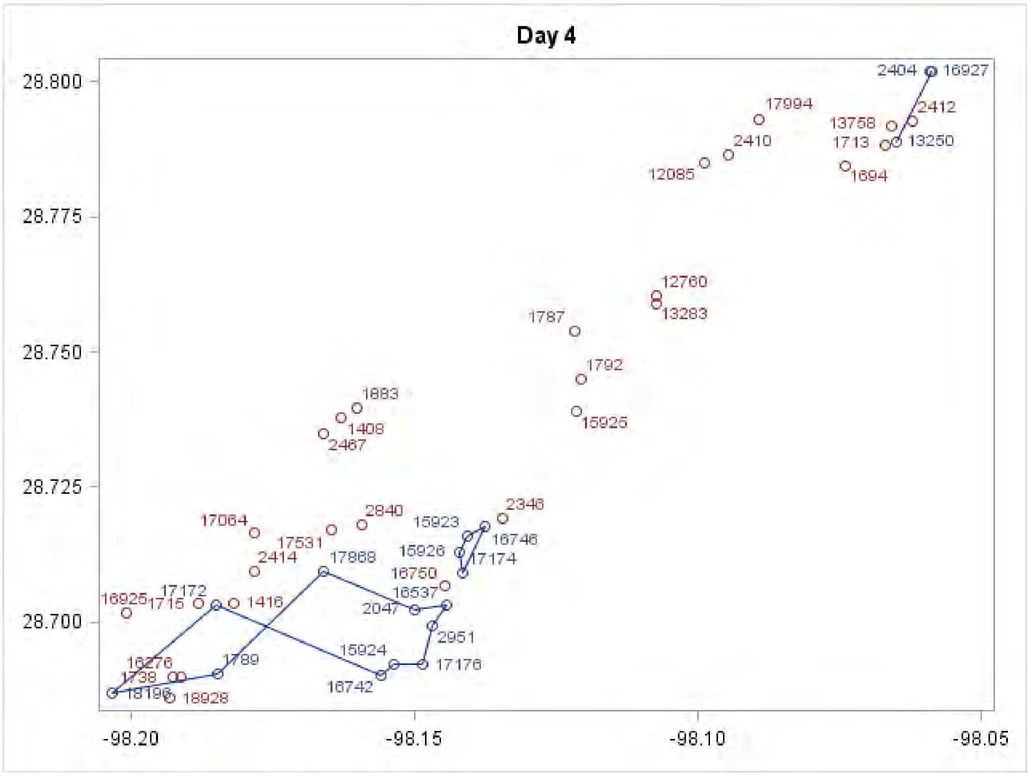
As before, blue markers indicate wells that are visited, and red markers indicate wells that are not visited. You can see that the solution is disconnected, with four connected components. The dummy depot node (which is not shown) has no physical location but belongs to the component in the middle that looks like a path. You need SubtourElimination constraints for the three subtours that do not contain the depot. After you add those constraints and call the MILP solver again, you get a different solution that has new subtours, as shown in Figure 6.

Figure 6 More Subtours



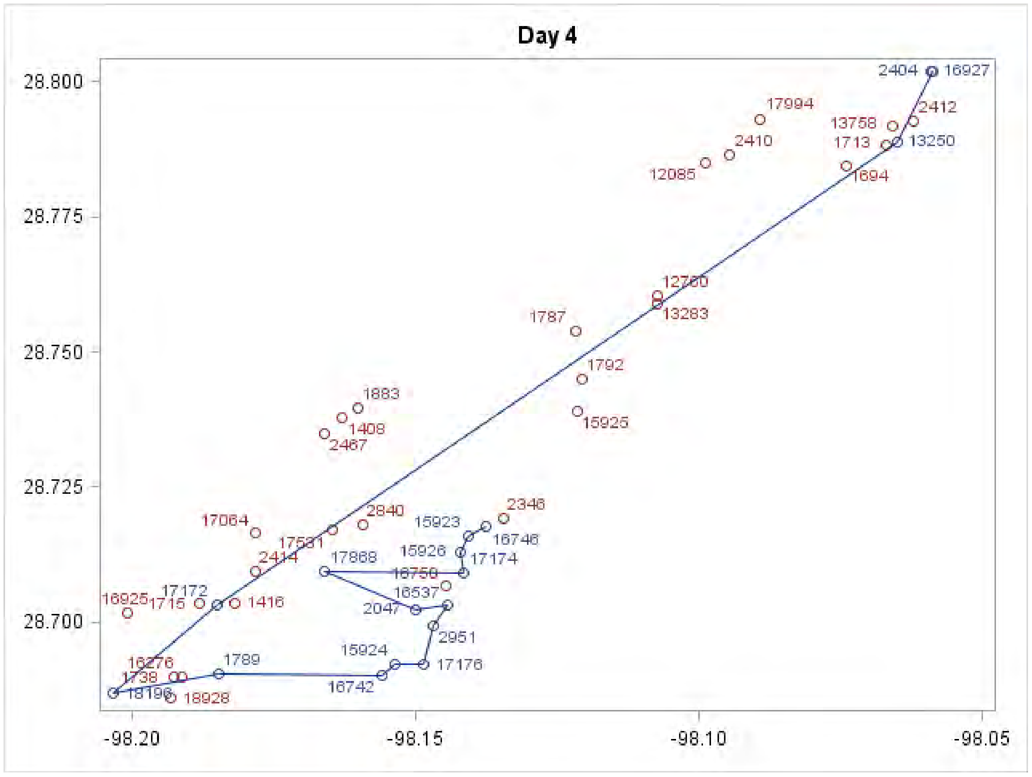
If you solve again with the additional SubtourElimination constraints, you get more subtours, as shown in Figure 7.

Figure 7 Even More Subtours



If you solve again with the additional SubtourElimination constraints, you get no subtours, as shown in Figure 8.

Figure 8 No Subtours



When the solution is connected for each day, the algorithm terminates.

The following DO UNTIL loop invokes the MILP solver, records the best lower bound, and calls two macros to look for connected components and check feasibility, as shown in the next section:

```
/* loop until optimality gap is small enough */
do until (gap ne . and gap <= &relobjgap);
   solve with MILP / relobjgap=&relobjgap target=(best_lower_bound);
   lower_bound = _OROPTMODEL_NUM_['BEST_BOUND'];
   best_lower_bound = max(best_lower_bound, lower_bound);
   %findConnectedComponents;
   %checkFeasibility;
end;
```

Note that the MILP solver TARGET= option is used to terminate each solver call early if the solver finds an integer feasible solution whose objective matches the best lower bound so far.

NETWORK ALGORITHMS

SAS/OR provides access to a number of network algorithms:

- connected components
- biconnected components and articulation points
- maximal cliques
- cycles
- transitive closure
- linear assignment problem
- shortest path problem

- minimum-cost network flow problem
- minimum spanning tree problem
- minimum cut problem
- traveling salesman problem

Beginning in SAS/OR 12.1, you can access these network algorithms by using PROC OPTNET, a specialized procedure that accepts nodes and links data sets, or you can access them from within PROC OPTMODEL by using a SUBMIT block. Beginning in SAS/OR 13.1, you have more direct access to these algorithms in PROC OPTMODEL via the SOLVE WITH NETWORK statement. Some of the network algorithms, such as connected components and maximal cliques, are diagnostic. Others solve classical network optimization problems, including minimum-cost network flow, the traveling salesman problem (TSP), and linear assignment. For more information, see *SAS/OR User's Guide: Network Optimization Algorithms* and the network solver chapter in *SAS/OR User's Guide: Mathematical Programming*. In particular, the example in this paper uses connected components and TSP.

The following statements declare the %findConnectedComponents macro:

```
%macro findConnectedComponents;
   for {d in DAYS} do;
      NODES_SOL = {i     in NODES: UseNode[i,d].sol   > 0.5};
      EDGES_SOL = {<i,j> in EDGES: UseEdge[i,j,d].sol > 0.5};
      solve with network / concomp
         links=(include=EDGES_SOL) out=(concomp=component);
      COMPONENTS = setof {i in NODES_SOL} component[i];
      for {c in COMPONENTS} NODES_c[c] = {};
      for {i in NODES_SOL} do;
         ci = component[i];
         NODES_c[ci] = NODES_c[ci] union {i};
      end;
      num_components[d] = _OROPTMODEL_NUM_['NUM_COMPONENTS'];
      /* create subtour from each component not containing depot node */
      for {c in COMPONENTS: depot not in NODES_c[c]} do;
         num_subtours = num_subtours + 1;
         SUBTOUR[num_subtours] = NODES_c[c];
      end;
   end;
%mend findConnectedComponents;
```

This macro looks at the nodes and edges in the MILP solution for each day and uses the network solver together with the CONCOMP= option to find the connected components. For each subtour that does not include the depot, the *num_subtours* parameter is incremented by 1, and the nodes in that component are recorded in the *SUBTOUR* index set to be used in the SubtourElimination constraints declared earlier.

The following statements declare the %checkFeasibility macro:

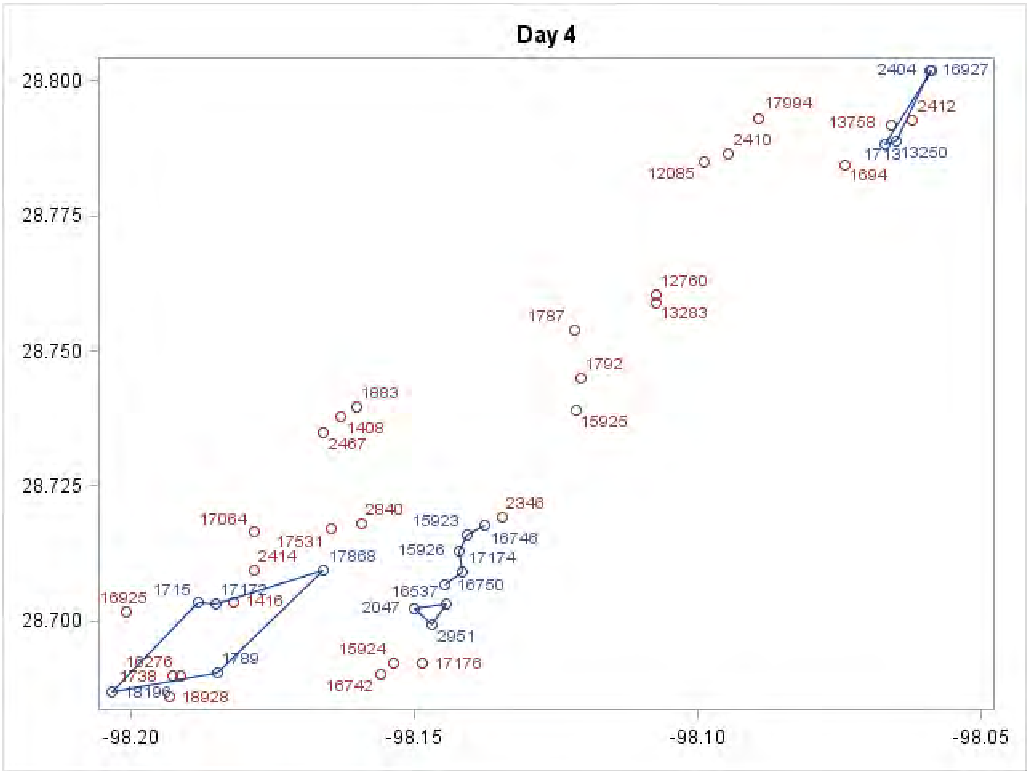
```
%macro checkFeasibility;
   if (and {d in DAYS} (num_components[d] <= 1)) then do;
      upper_bound = _OBJ_.sol;
      if best_upper_bound > upper_bound then do;
         best_upper_bound = upper_bound;
      end;
   end;
%mend checkFeasibility;
```

This macro uses the logical AND operator to check whether the solution is connected for each day. If the condition is true, then the solution is feasible, and the best upper bound is recorded.

REPAIR HEURISTIC

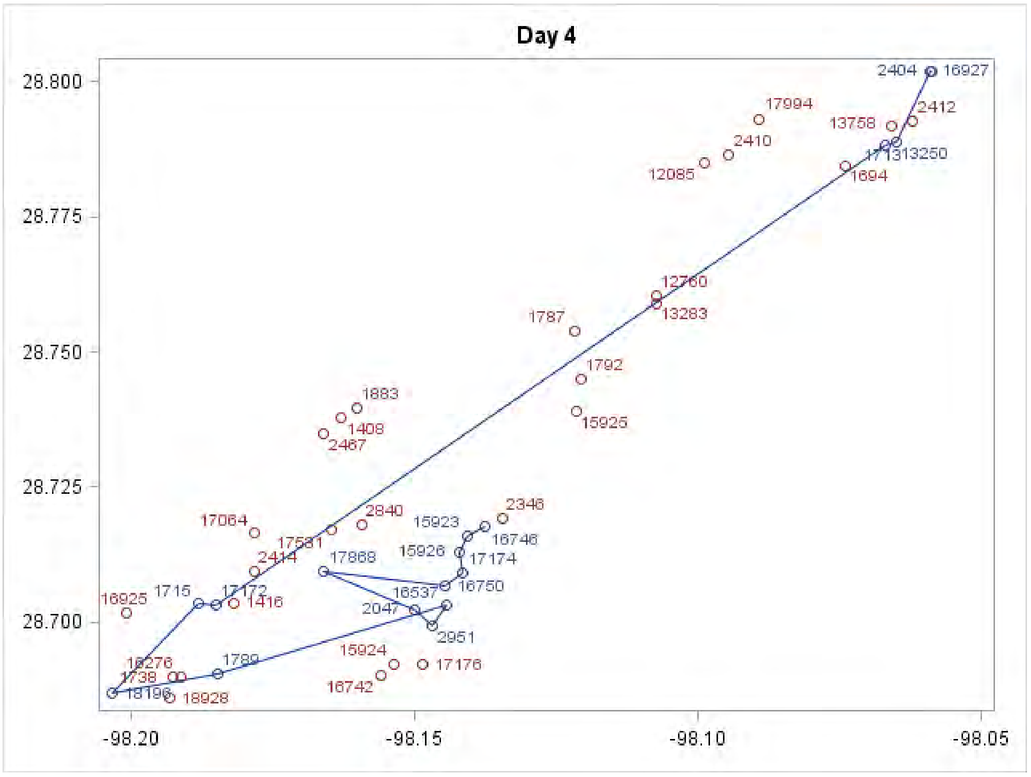
The approach that was just described is a complete algorithm similar to the Milk Collection example in *SAS/OR User's Guide: Mathematical Programming Examples*. This algorithm can solve the problem with the ConocoPhillips data in about an hour. By introducing a simple repair heuristic, you can reduce the run time to one minute. Each MILP solution satisfies all the requirements except possibly connectivity. If you leave the **UseNode** variables alone, the scheduling requirements are still satisfied. The repair heuristic modifies the **UseEdge** variables by solving a TSP for each day. The resulting solution satisfies all requirements except possibly the time budget per day. For example, suppose that the MILP solution for day 4 is as shown in Figure 9.

Figure 9 Before Repair: TimePerDay$[4] = 566.235 \leq 600$



The service plus travel time is at most 600 minutes because of the constraints imposed, but the solution is disconnected. The repair heuristic solves a TSP through the blue nodes, yielding a connected solution, as shown in Figure 10. The time per day increases, but it is still at most 600 minutes.

Figure 10 After Repair: TimePerDay[4] $= 583.267 \leq 600$



The following example shows what can go wrong. Before repair, the service plus travel time for day 5 is at most 600, as shown in Figure 11. After repair, the solution is connected, as shown in Figure 12. But the time per day exceeds 600, so the heuristic has failed. As long as the heuristic sometimes succeeds, it can reduce the overall solution time.

Figure 11 Before Repair: TimePerDay[5] $= 587.698 \leq 600$

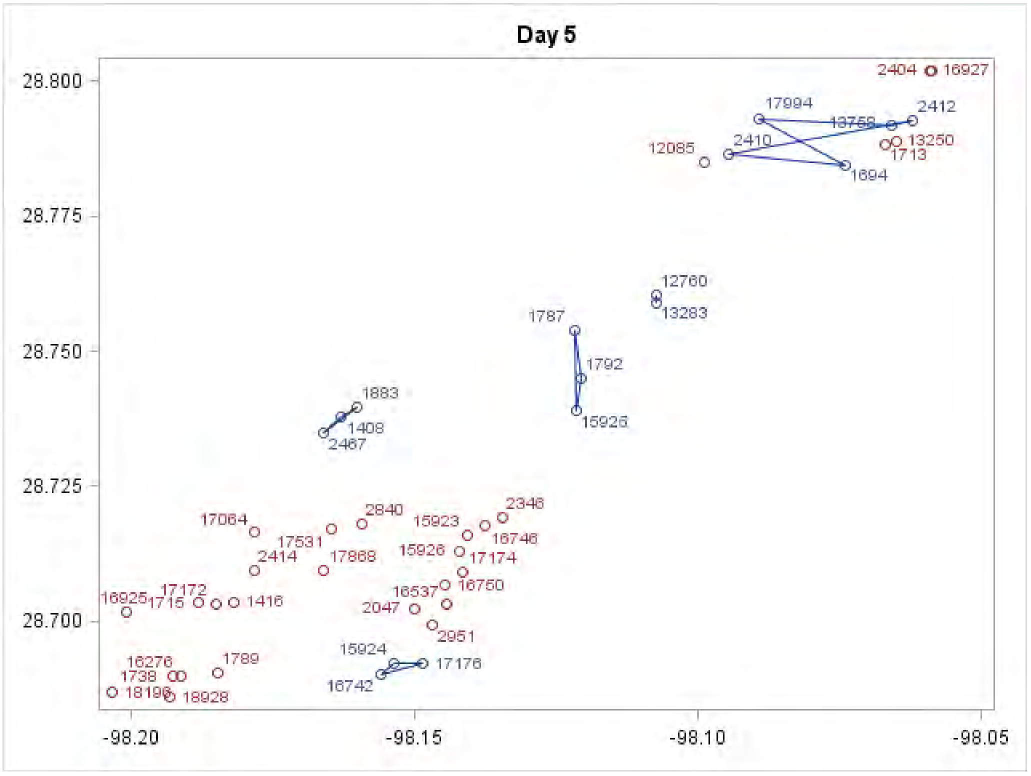
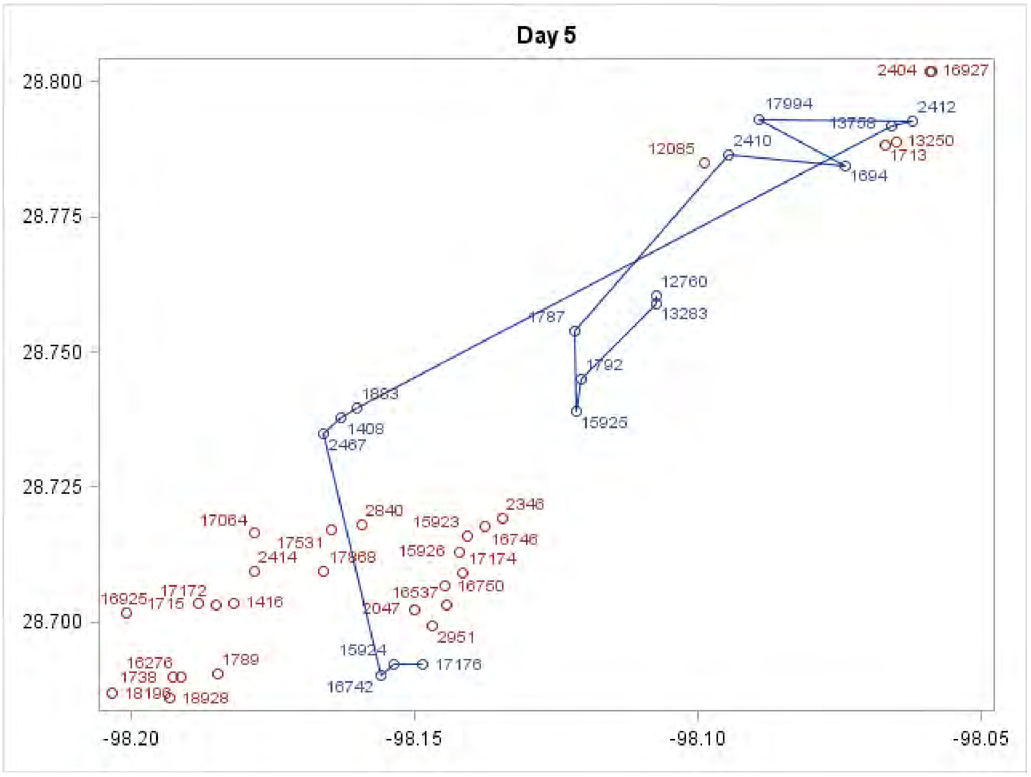


Figure 12 After Repair: TimePerDay[5] $= 612.657 > 600$



The TSPs are independent across days and can potentially benefit from the COFOR statement, which enables you to solve independent optimization problems concurrently by using multiple threads. The COFOR syntax is very similar to the serial FOR loop syntax, with just one keyword change, from FOR to COFOR:

```
cofor {...} do;
   ...
   solve ...;
   ...
end;
```

A best practice is to first develop your code by using a FOR loop, and then when everything is working correctly, switch the keyword FOR to COFOR to boost the performance. Beginning in SAS/OR 14.1, the COFOR statement also supports running in a distributed environment across multiple machines.

The following %repairSolution macro implements the repair heuristic:

```
%macro repairSolution;
   for {<i,j> in EDGES, d in DAYS} UseEdge[i,j,d] = 0;
   /* exactly two nodes */
   for {d in DAYS: num_nodes[d] = 2} do;
      put d=;
      TSP_NODES = {i in NODES: UseNode[i,d].sol > 0.5};
      for {i in TSP_NODES, j in TSP_NODES: <i,j> in EDGES} UseEdge[i,j,d] = 2;
   end;
   /* more than two nodes */
   cofor {d in DAYS: num_nodes[d] > 2} do;
      put d=;
      TSP_NODES = {i in NODES: UseNode[i,d].sol > 0.5};
      solve with network / tsp
         links=(weight=distance) subgraph=(nodes=TSP_NODES) out=(tour=TOUR);
      for {<i,j> in TOUR} UseEdge[i,j,d] = 1;
   end;
%mend repairSolution;
```

The **UseEdge** variables are initialized to 0. If only two nodes (the depot and one well) are used in a day, there is no TSP to be solved, and the **UseEdge** variable is set to 2. For days that use more than two nodes, the COFOR loop calls the network solver along with the TSP= option, and the **UseEdge** variables are set to 1 for the edges in the resulting tour.

Only a few lines change in the %checkFeasibility macro:

```
%macro checkFeasibility;
   if (and {d in DAYS} (TimePerDay[d].sol <= &time_budget)) then do;
      upper_bound = _OBJ_.sol;
      if best_upper_bound > upper_bound then do;
         best_upper_bound = upper_bound;
         INCUMBENT_ID = {i in NODES, d in DAYS: UseNode[i,d].sol > 0.5};
         for {k in 1..2}
            INCUMBENT_IJD[k] = {<i,j> in EDGES, d in DAYS: round(UseEdge[i,j,d].sol) = k};
      end;
   end;
%mend checkFeasibility;
```

After repair, you know that the solution is connected, so the IF condition changes to check the TimeBudgetPerDay constraint satisfaction instead. Two additional statements store the best feasible solution found so far. The following %recoverIncumbentSolution macro recovers this solution:

```
%macro recoverIncumbentSolution;
   for {i     in NODES, d in DAYS} UseNode[i,d]   = (<i,d> in INCUMBENT_ID);
   for {<i,j> in EDGES, d in DAYS} UseEdge[i,j,d] =
      if      <i,j,d> in INCUMBENT_IJD[1] then 1
      else if <i,j,d> in INCUMBENT_IJD[2] then 2
      else 0;
%mend recoverIncumbentSolution;
```

The main DO UNTIL loop changes very little:

```
/* loop until optimality gap is small enough */
do until (gap ne . and gap <= &relobjgap);
   %recoverIncumbentSolution;
   solve with MILP / relobjgap=&relobjgap target=(best_lower_bound) primalin;
   lower_bound = _OROPTMODEL_NUM_['BEST_BOUND'];
   best_lower_bound = max(best_lower_bound, lower_bound);
   %findConnectedComponents;
   %repairSolution;
   %checkFeasibility;
end;
```

The %recoverIncumbentSolution macro is called, and the PRIMALIN option for the MILP solver is used to warm start with this solution. The final change is that the %repairSolution macro is called.

ILLUSTRATION OF ALGORITHM

This section illustrates the entire algorithm, with *relobjgap* set to 0.01 so that the algorithm terminates with a solution that is within 1% of optimal. To conserve space, only days 2 through 5 out of the 10 days in the planning horizon are shown. Nothing interesting happens on the other days.

In the first iteration, the MILP solver is called, yielding a lower bound of 1916 and an infinite upper bound because there is not yet a feasible solution, as shown in Figure 13. The solution is disconnected on all four of these days. The repair heuristic solves a TSP for each day and is successful. The new upper bound is 1979.6, with an optimality gap of 3.3%, as shown in Figure 14.

Figure 13 Iteration 1: MILP Solver Call
Bounds $= [1916.0, \infty)$, Gap $=$.

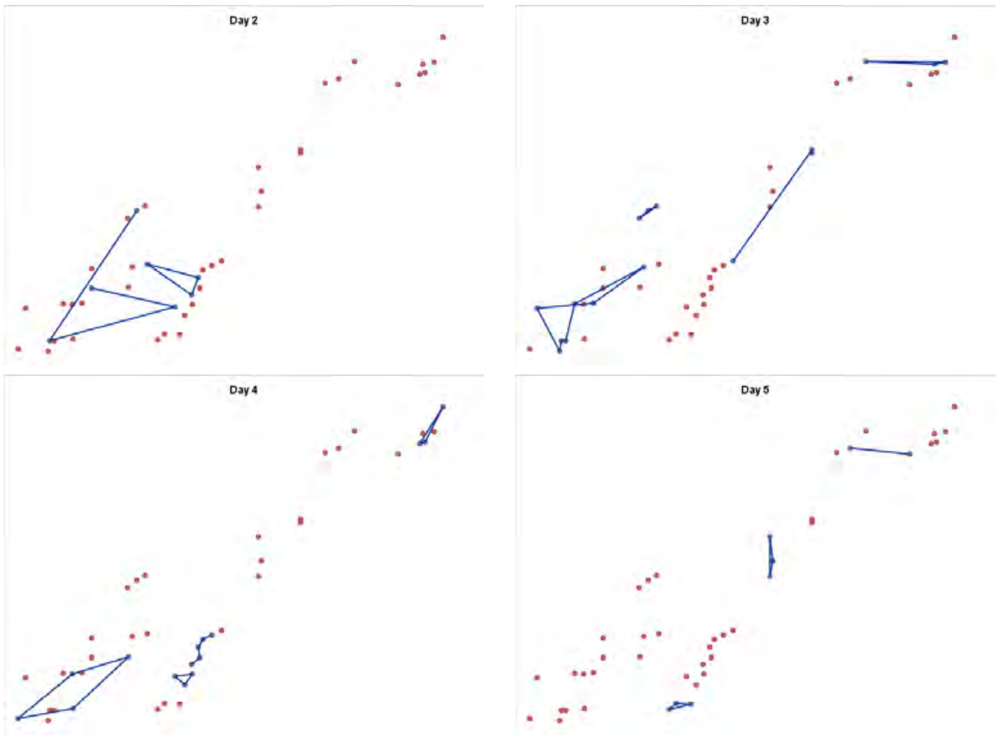
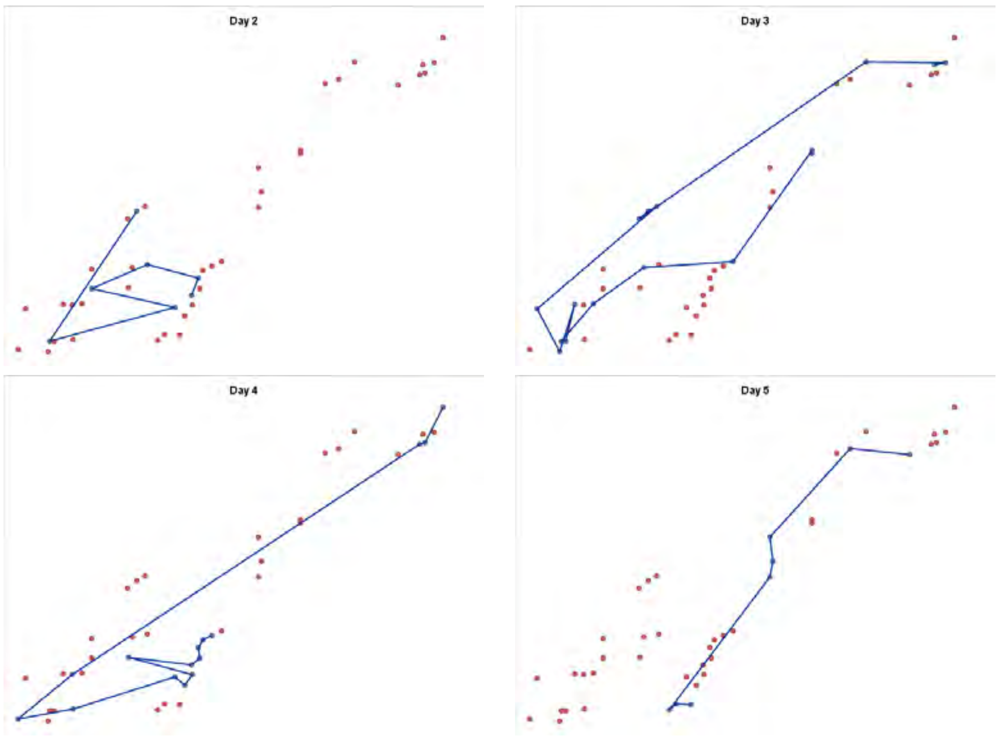


Figure 14 Iteration 1: Repair Heuristic Succeeds
Bounds $= [1916.0, 1979.6]$, Gap $= 0.033$



With the new SubtourElimination constraints added, the MILP solver returns a new solution, with a higher lower bound of 1924.2, as shown in Figure 15. The repair heuristic solves a TSP for each day but fails because the TimeBudgetPerDay constraint is violated on days 3 and 4, so the upper bound does not change, as shown in Figure 16.

Figure 15 Iteration 2: MILP Solver Call
Bounds $= [1924.2, 1979.6]$, Gap $= 0.029$

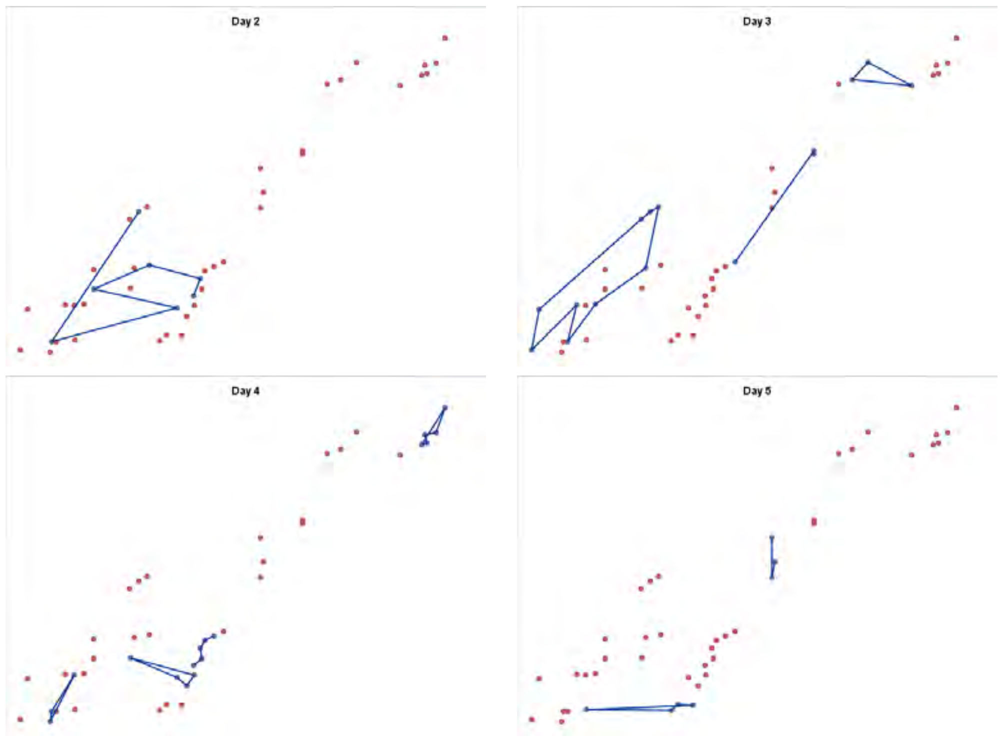
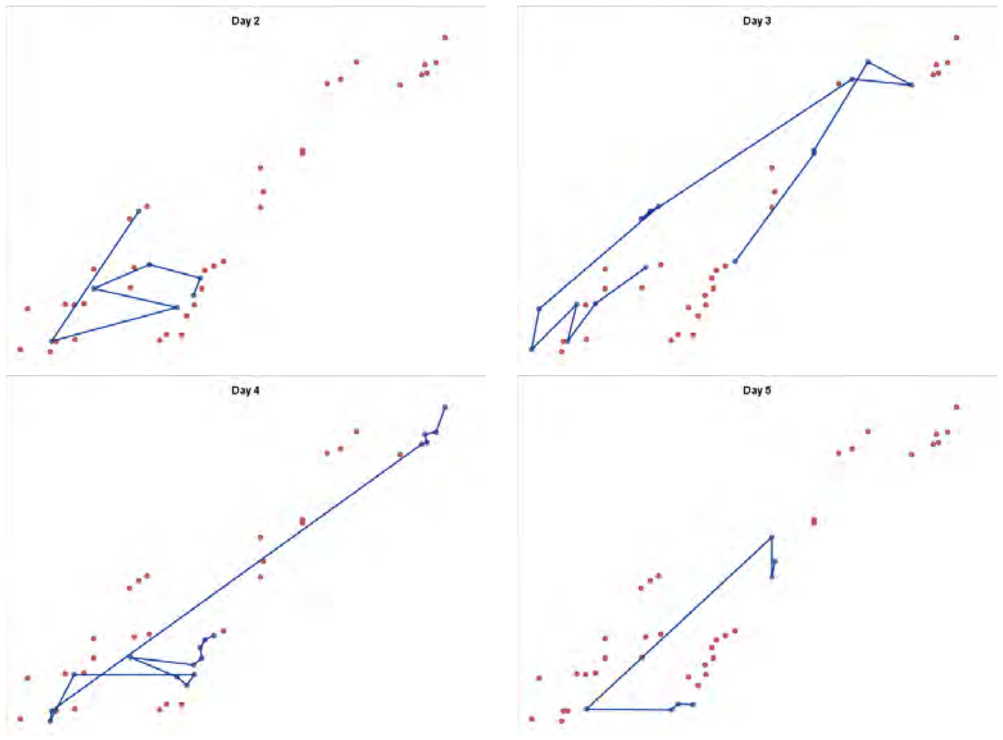


Figure 16 Iteration 2: Repair Heuristic Fails
Bounds $= [1924.2, 1979.6]$, Gap $= 0.029$



After the next MILP solver call, the lower bound increases, and the repair heuristic fails again, as shown in Figure 17 and Figure 18.

Figure 17 Iteration 3: MILP Solver Call
Bounds $= [1924.8, 1979.6]$, Gap $= 0.028$

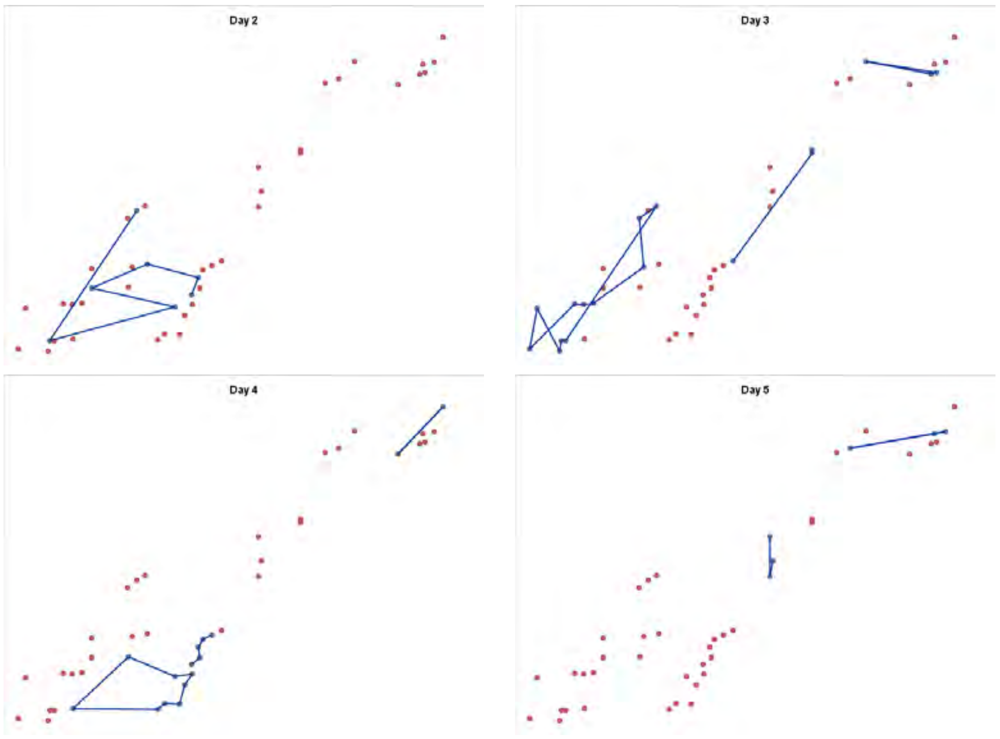
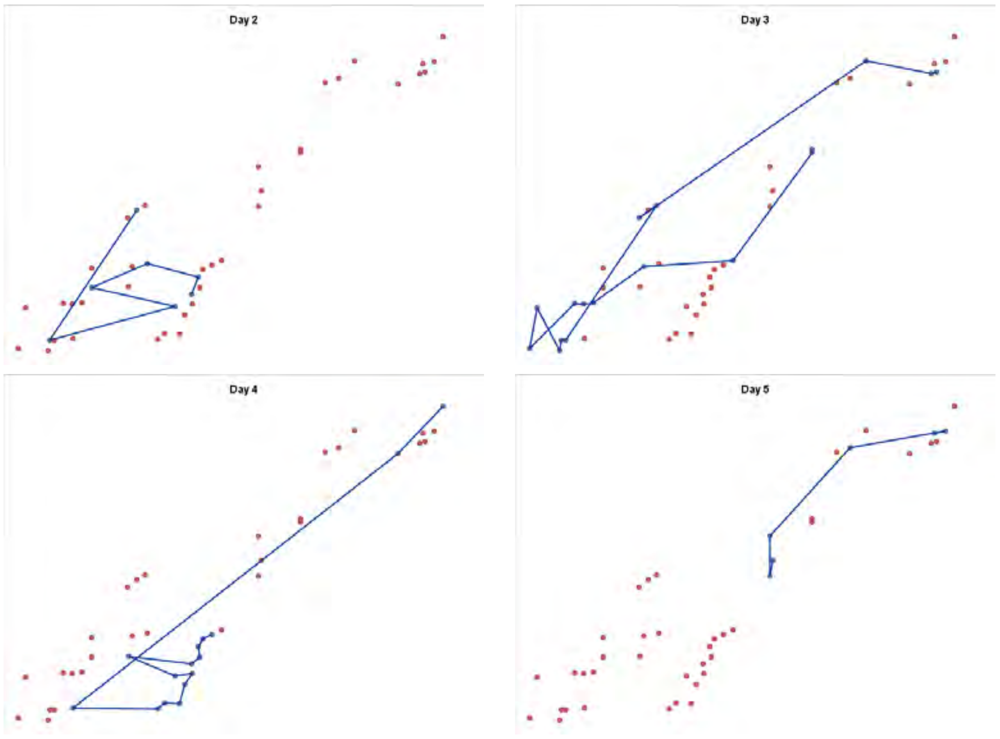


Figure 18 Iteration 3: Repair Heuristic Fails
Bounds $= [1924.8, 1979.6]$, Gap $= 0.028$



After the next MILP solver call, the repair heuristic succeeds, and both bounds are updated, as shown in Figure 19 and Figure 20.

Figure 19 Iteration 4: MILP Solver Call
Bounds $= [1926.5, 1979.6]$, Gap $= 0.028$

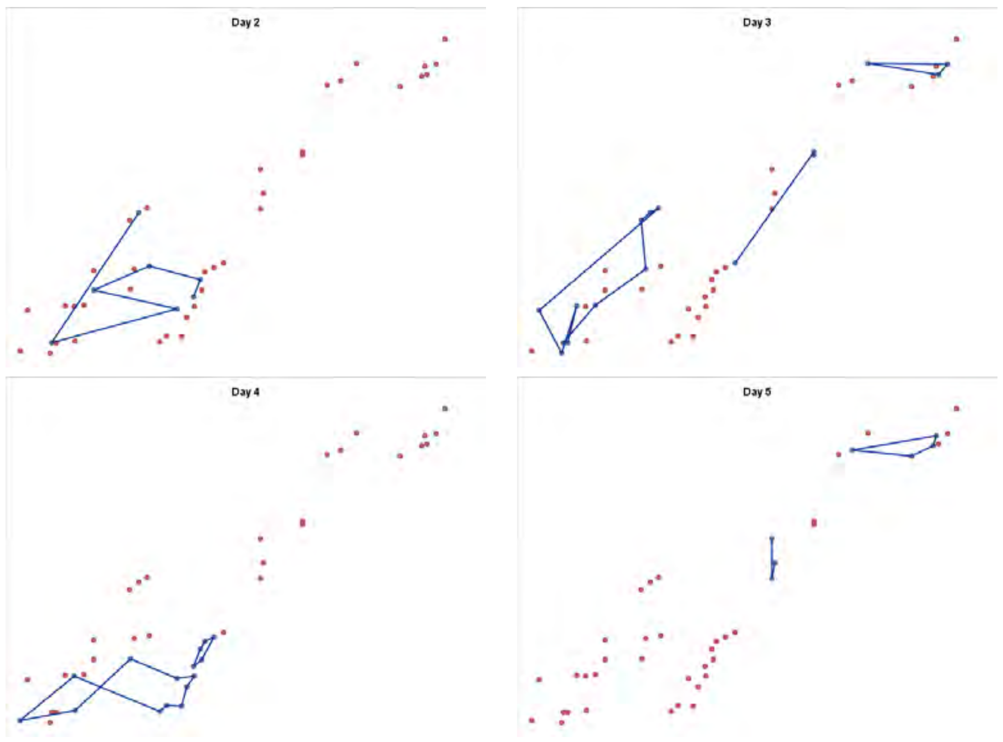
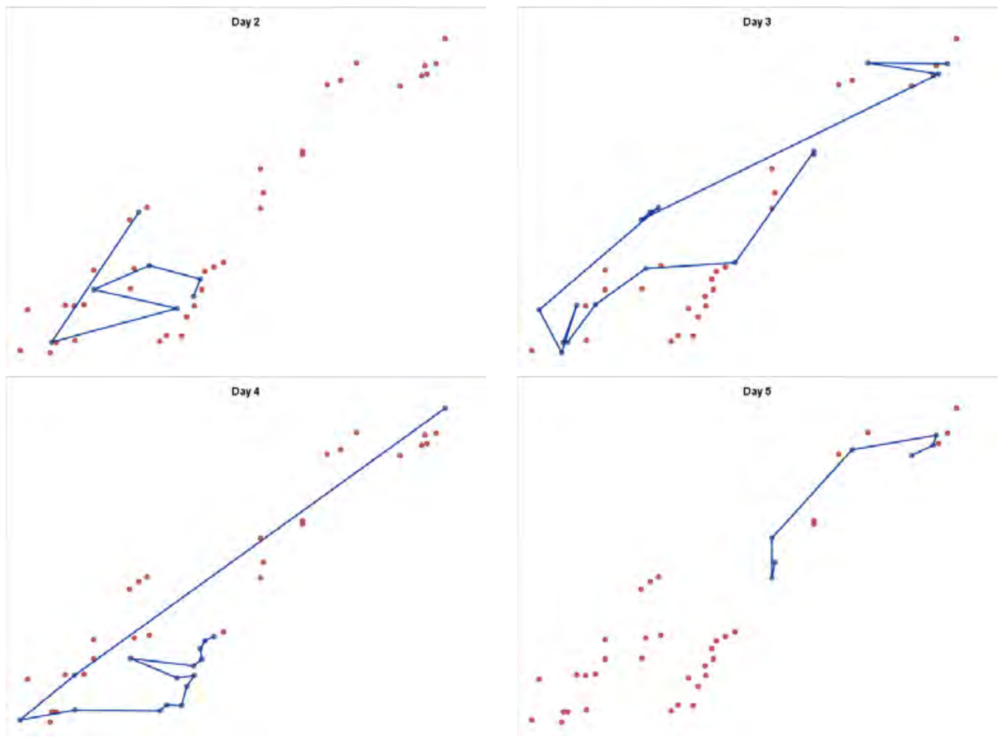


Figure 20 Iteration 4: Repair Heuristic Succeeds
Bounds $= [1926.5, 1966.8]$, Gap $= 0.021$



After the next MILP solver call, the repair heuristic succeeds, but the resulting solution does not improve the upper bound, as shown in Figure 21 and Figure 22.

Figure 21 Iteration 5: MILP Solver Call
Bounds $= [1931.7, 1966.8]$, Gap $= 0.018$

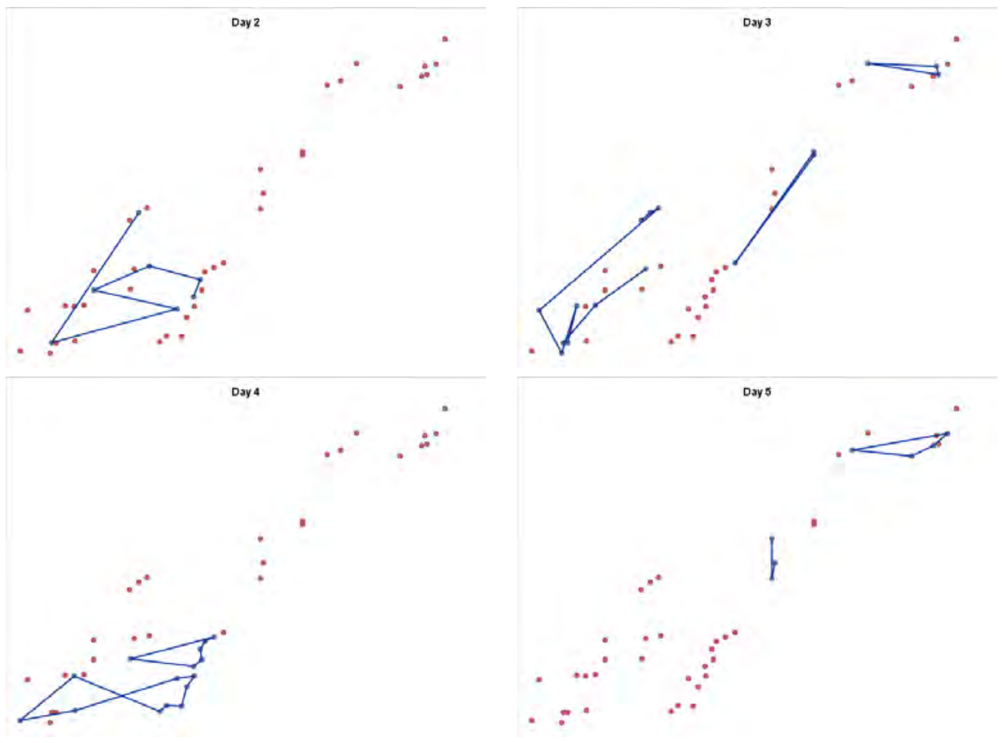
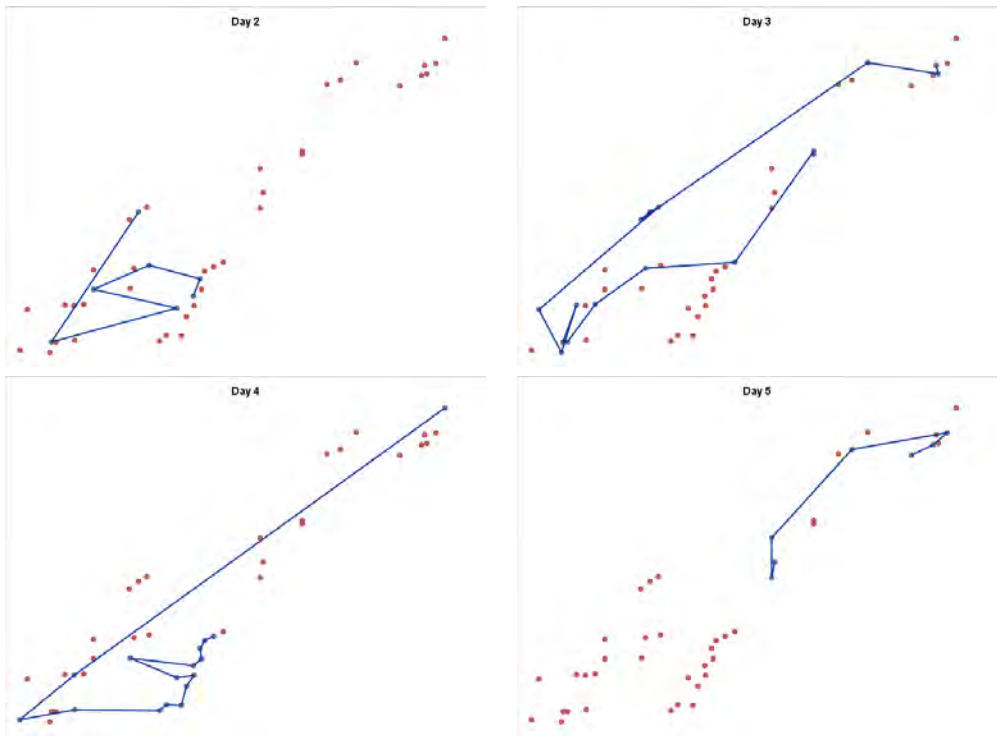


Figure 22 Iteration 5: Repair Heuristic Succeeds but No Improvement
Bounds $= [1931.7, 1966.8]$, Gap $= 0.018$



After the next MILP solver call, the repair heuristic fails, as shown in Figure 23 and Figure 24.

Figure 23 Iteration 6: MILP Solver Call
Bounds $= [1932.7, 1966.8]$, Gap $= 0.018$

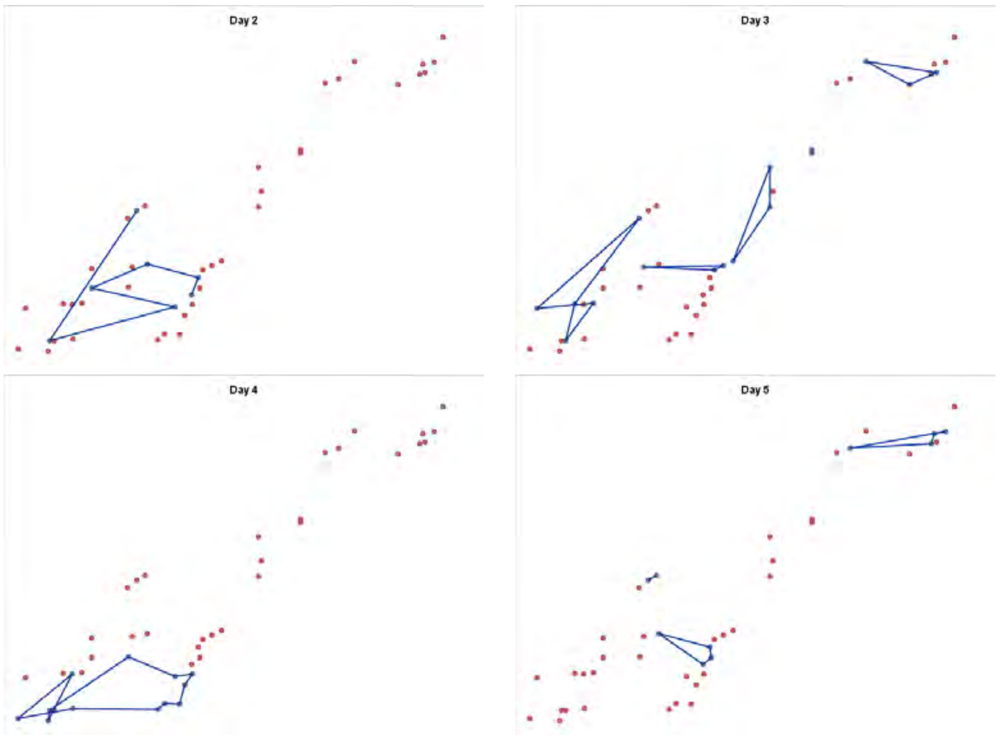
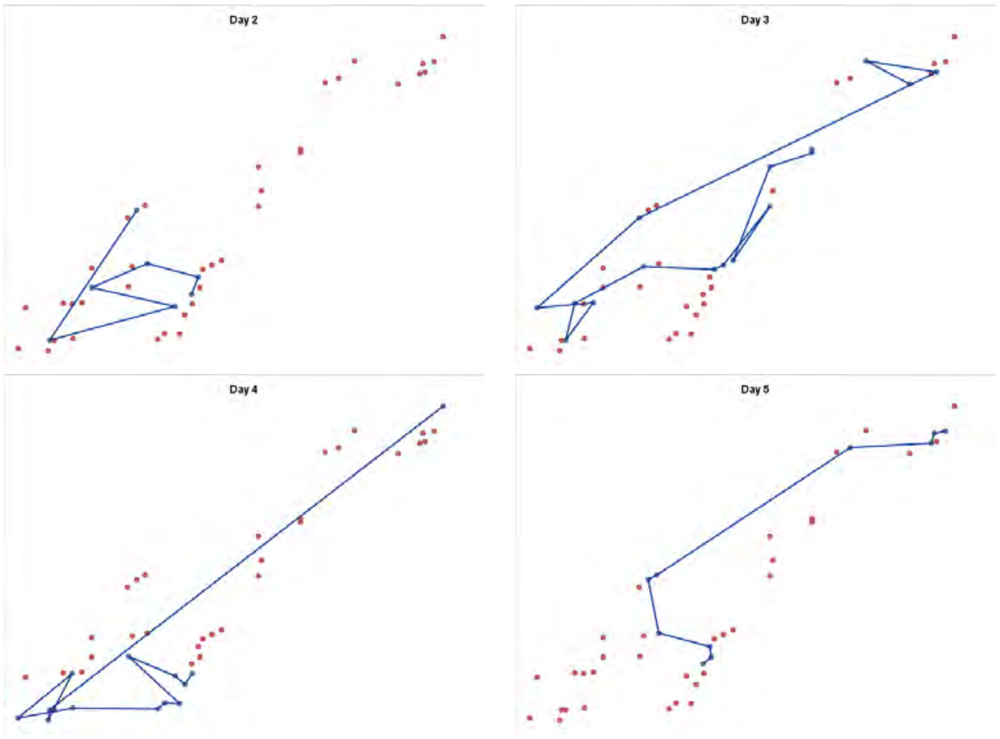


Figure 24 Iteration 6: Repair Heuristic Fails
Bounds $= [1932.7, 1966.8]$, Gap $= 0.018$



After the next MILP solver call, the repair heuristic succeeds, as shown in Figure 25 and Figure 26. Because the optimality gap is 1%, the DO UNTIL loop terminates.

Figure 25 Iteration 7: MILP Solver Call
Bounds $= [1934.2, 1966.8]$, Gap $= 0.017$

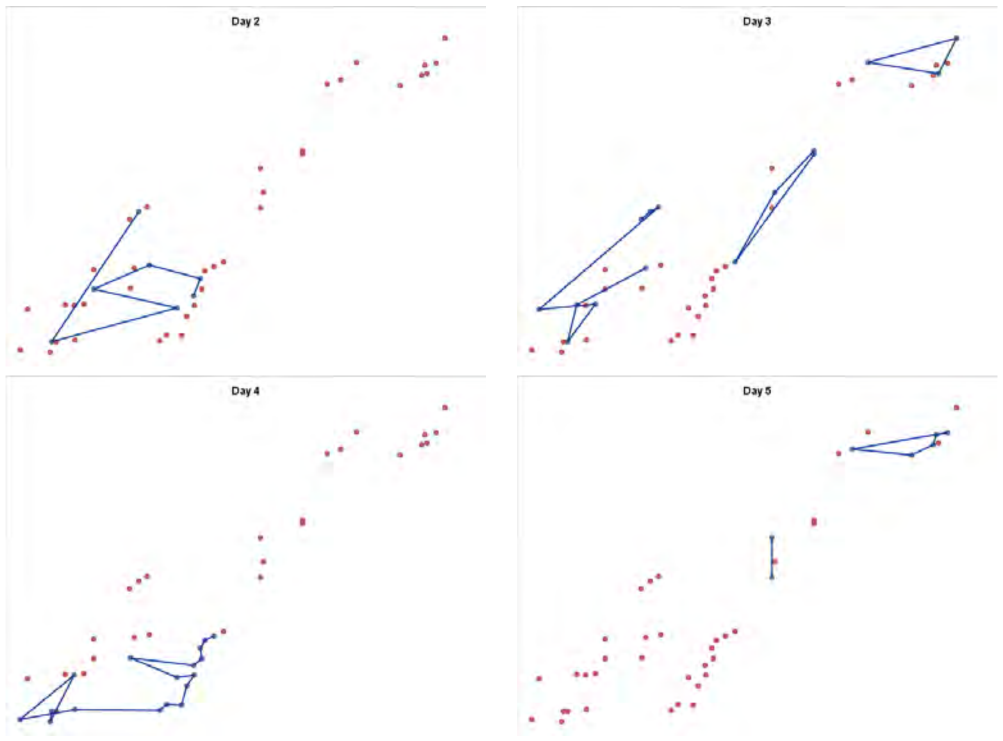


Figure 26 Iteration 7: Repair Heuristic Succeeds
Bounds $= [1934.2, 1953.2]$, Gap $= 0.010$

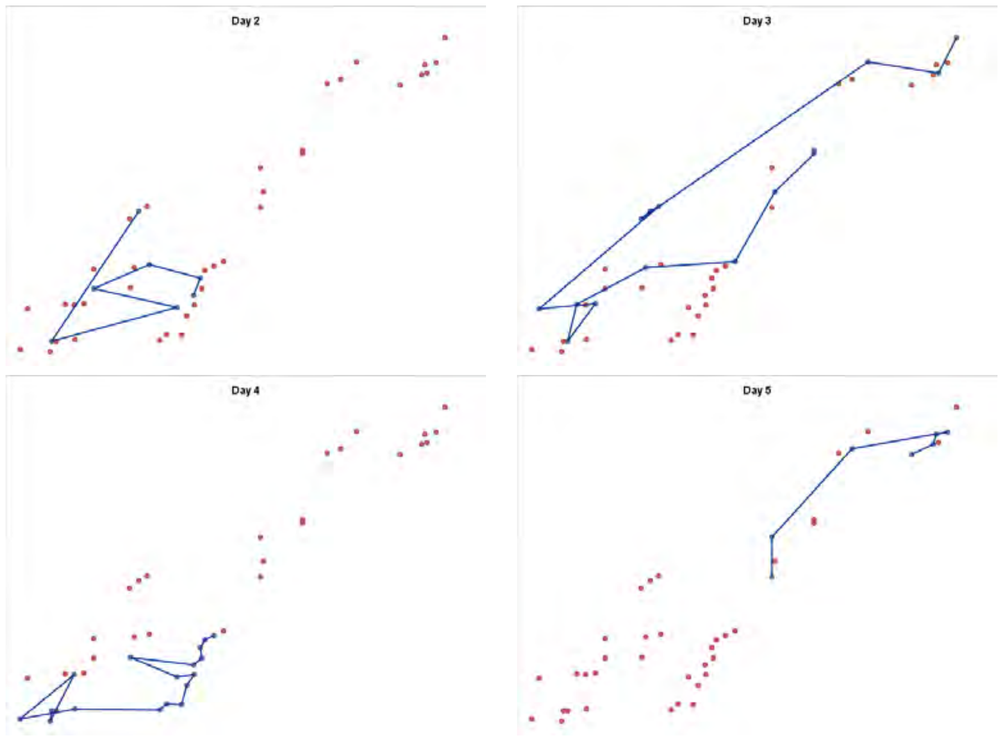


Figure 27 shows the resulting optimal[1] schedule as a heat map, with the nodes along the left and the days across the top. You can see that the depot is visited each day, and every seven-day interval is covered for each well, but on

[1] For the solutions shown in this paper, the word *optimal* means within 1% of optimal.

several days only one other node is visited. The stacked bar chart in Figure 28 shows that the TimeBudgetPerDay constraints are satisfied. On days when only one well is visited, the travel time is 0.

Figure 27 Optimal Schedule

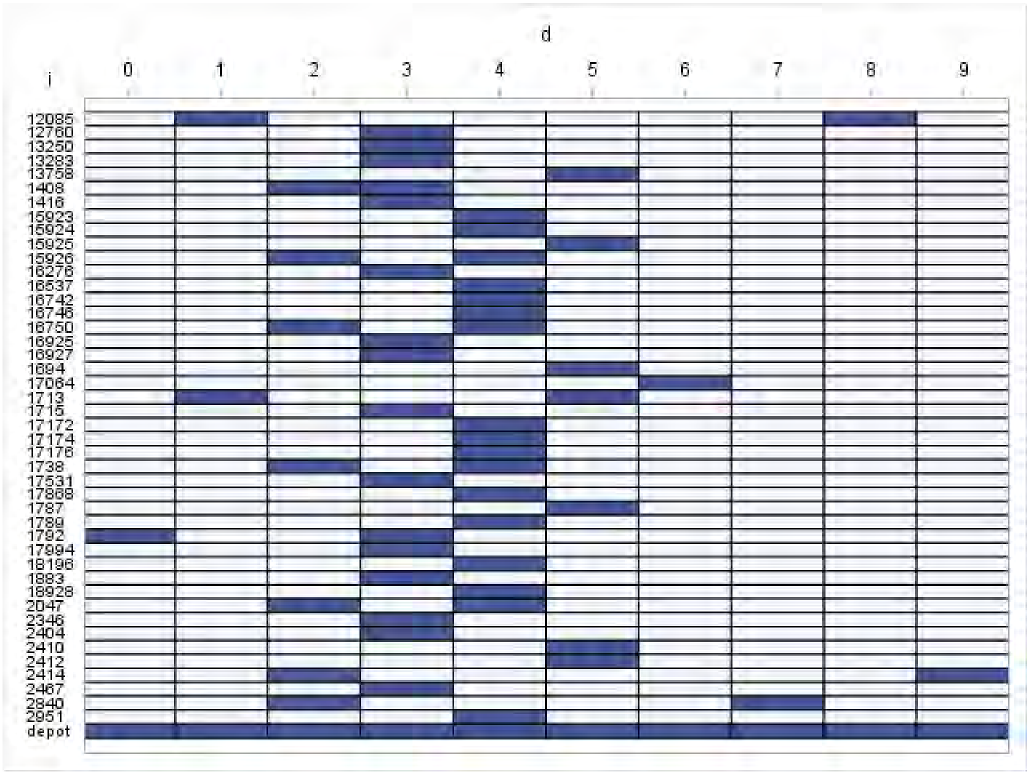
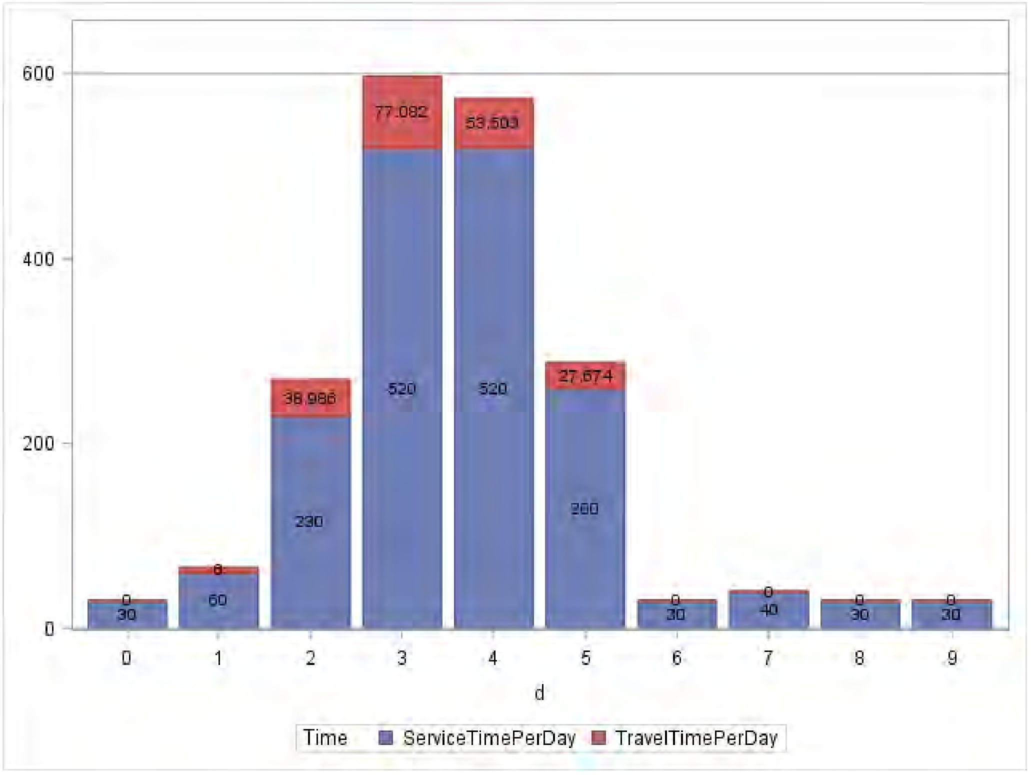


Figure 28 Time per Day



REDUCING IMBALANCE

The optimization model encourages this imbalance because travel to and from the depot does not count in the operator's workday. It is never optimal to have an empty day. To avoid this imbalance, you can introduce the following Threshold constraints that require either 0 or some minimum number of wells to be visited per day:

```
/* if operator visits depot, must visit at least five wells */
con Threshold {d in DAYS}:
   5 * UseNode[depot,d] <= sum {i in NODES diff {depot}} UseNode[i,d];
```

If the depot is visited on day d, the left-hand side of the constraint is 5, and the constraint forces at least five wells to be visited on that day. If the depot is not visited, the left-hand side is 0, and the constraint is redundant. The new optimal schedule shown in Figure 29 prescribes visits on only six days in the planning horizon. Figure 30 shows that the TimeBudgetPerDay constraints are still satisfied, but the total time increases, as expected because the optimization model contains additional constraints.

Figure 29 Optimal Schedule

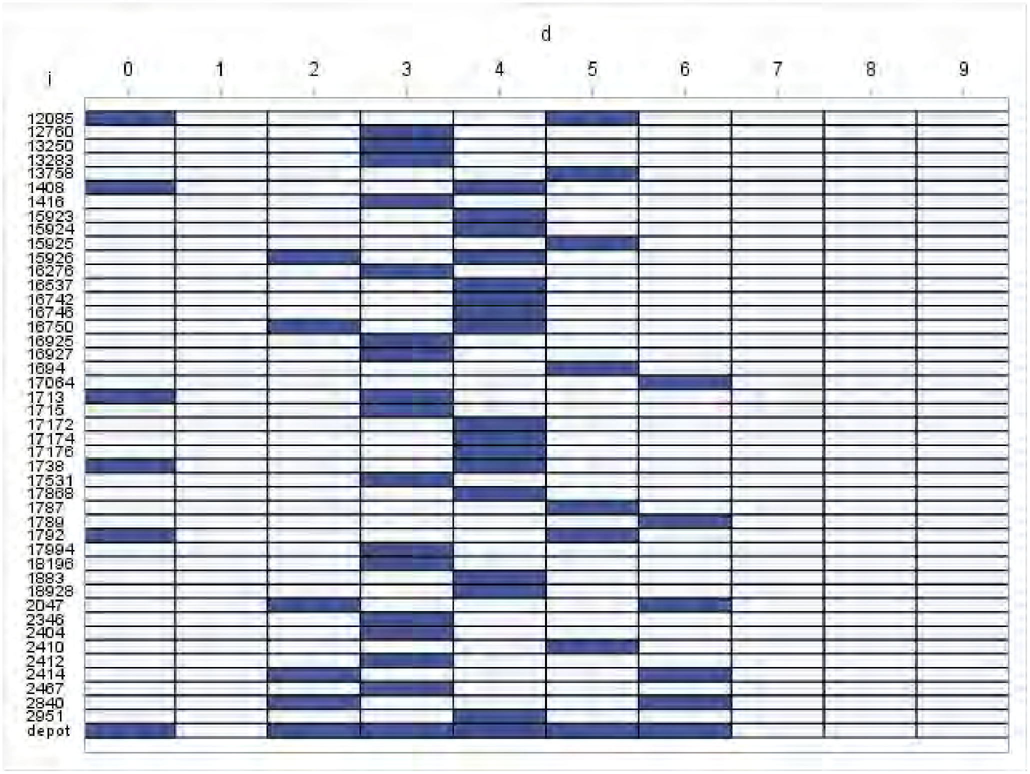
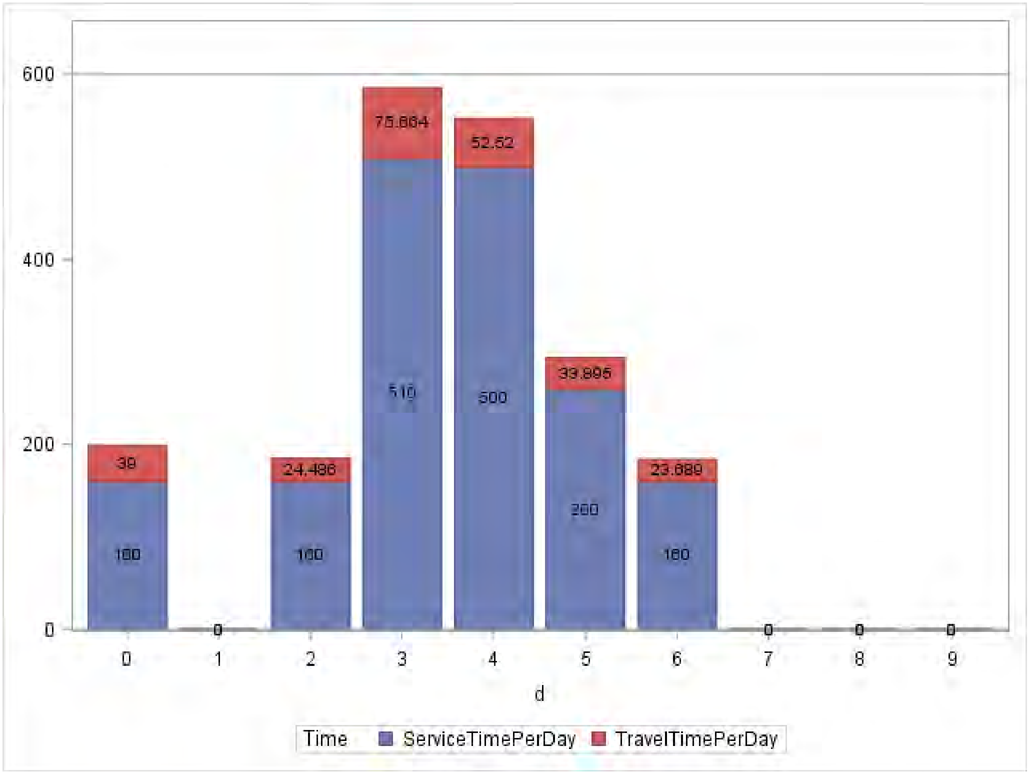


Figure 30 Time per Day (TotalTime Increases from 1953.2 to 1999.3)



An alternative way to discourage imbalance is to limit the number of workdays in the planning horizon. For example, the following Cardinality constraint limits the number of workdays to four:

```
/* work at most four days */
con Cardinality:
   sum {d in DAYS} UseNode[depot,d] <= 4;
```

Figure 31 shows the new optimal schedule, and Figure 32 shows the new bar chart for time per day.

Figure 31 Optimal Schedule

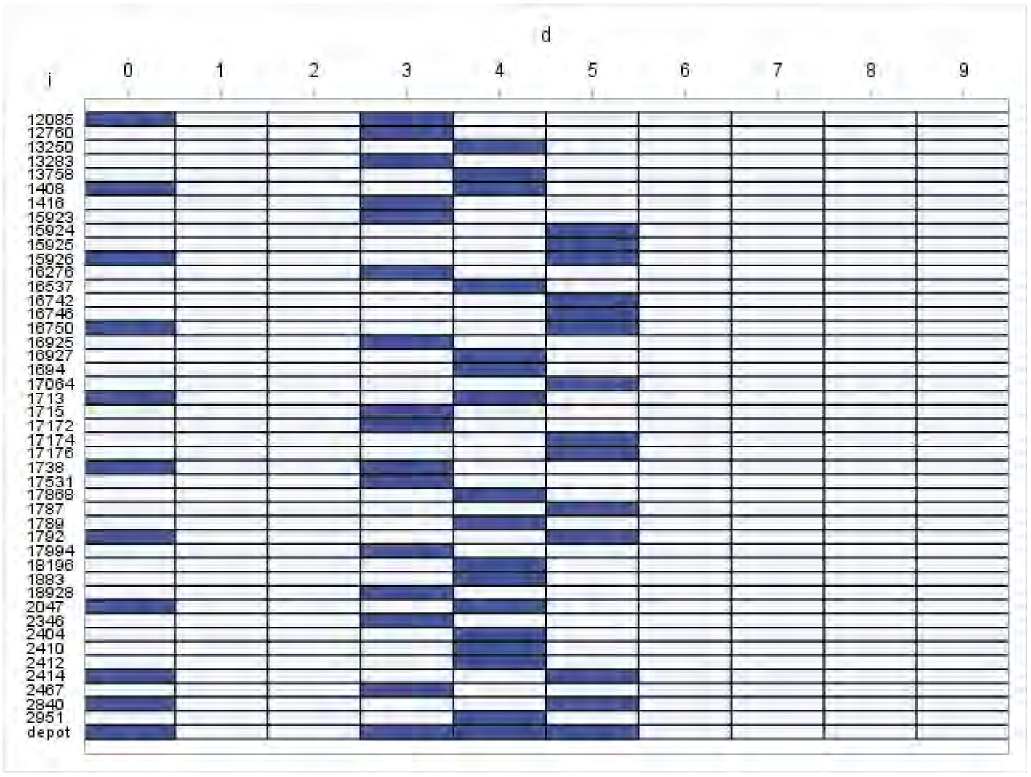
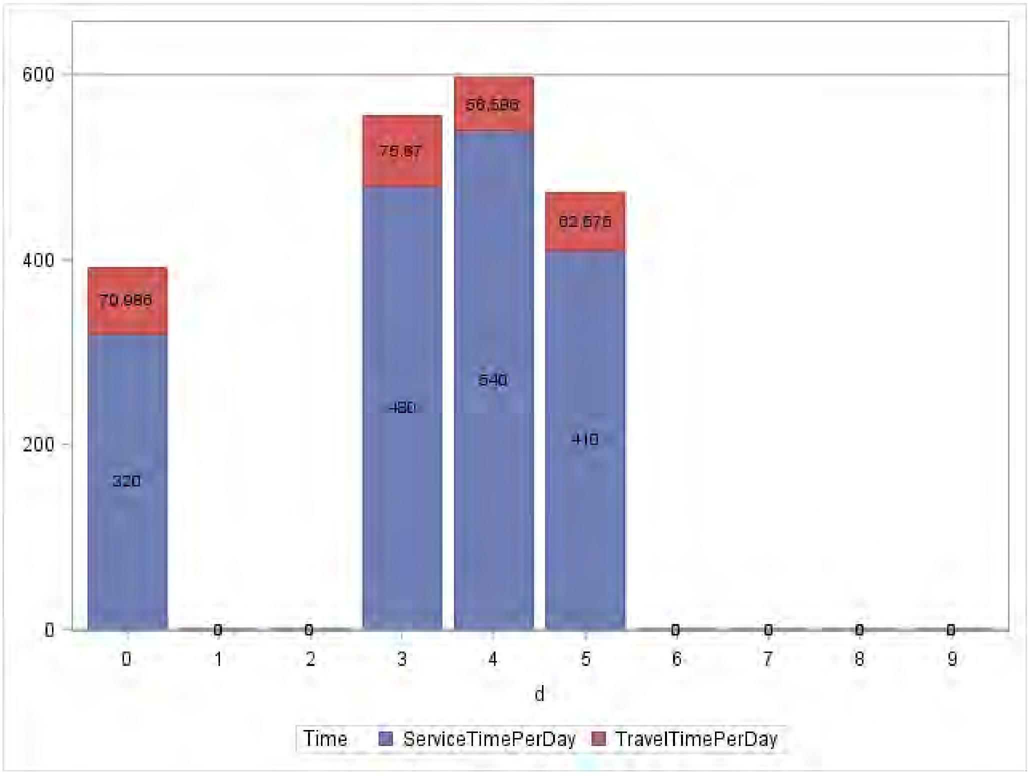


Figure 32 Time per Day (TotalTime Increases from 1953.2 to 2015.8)



OTHER EXTENSIONS

Besides the additional constraints to reduce imbalance, this section suggests a few other extensions to the problem.

PACKING CONSTRAINTS

The optimization model uses covering constraints to force at least one visit to each well over each seven-day interval. You can similarly use packing constraints to prohibit more than one visit to the same well over a short interval.

OPERATOR SCHEDULING RULES

You can include other constraints on the **UseNode** variables to capture rules associated with the operators, such as minimum or maximum consecutive days off, or scheduling conflicts due to operator vacations.

ASYMMETRIC TRAVEL TIMES

If the travel times are asymmetric in the sense that the time to travel from well i to well j is different from the travel time from j to i, minor modifications to the optimization model can also handle that, and the repair heuristic would then instead use the asymmetric TSP solver that was introduced in SAS/OR 14.1.

OPERATOR REASSIGNMENT

This paper solves the problem for one operator at a time, but you could also reassign operators to wells in a larger optimization model that considers all operators and wells simultaneously. This larger problem would probably be suitable for the decomposition algorithm, with each block corresponding to an operator.

UNCERTAINTY

Finally, the service and travel times here are treated as deterministic. If the data change, you can reoptimize the problem, but you could also handle uncertainty by using robust optimization, a methodology in which input data are replaced with confidence intervals.

CONCLUSION

This paper demonstrates the power and flexibility of the OPTMODEL procedure in SAS/OR to solve mathematical optimization problems, with a scheduling and routing problem as an illustrative example. The rich and expressive algebraic modeling language in PROC OPTMODEL enables you to easily formulate problems and access multiple solvers. You can also use the programming language provided by PROC OPTMODEL to write customized algorithms that call the solvers as subroutines. The COFOR statement offers a simple way to exploit parallel processing by solving independent problems concurrently, on either one machine or a grid.

REFERENCES

Campbell, A. M., and Wilson, J. H. (2014). "Forty Years of Periodic Vehicle Routing." *Networks* 63:2–15.

SAS Institute Inc. (2017a). *SAS/OR 14.3 User's Guide: Mathematical Programming*. Cary, NC: SAS Institute Inc. `http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=titlepage.htm&docsetVersion=14.3&locale=en`.

SAS Institute Inc. (2017b). *SAS/OR 14.3 User's Guide: Mathematical Programming Examples*. Cary, NC: SAS Institute Inc. `http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=titlepage.htm&docsetVersion=14.3&locale=en`.

SAS Institute Inc. (2017c). *SAS/OR 14.3 User's Guide: Network Optimization Algorithms*. Cary, NC: SAS Institute Inc. `http://go.documentation.sas.com/?docsetId=ornoaug&docsetTarget=titlepage.htm&docsetVersion=14.3&locale=en`.

ACKNOWLEDGMENT

Thanks to Paal Navestad of ConocoPhillips for introducing the author to this problem and for providing the data.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Rob Pratt
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-1099
Rob.Pratt@sas.com