

The Traveling Baseball Fan Problem and the OPTMODEL Procedure

Tonya Chapman, Matt Galati, and Rob Pratt, SAS Institute Inc.

ABSTRACT

In the traveling salesman problem, a salesman must minimize travel distance while visiting each of a given set of cities exactly once. This paper uses the SAS/OR® OPTMODEL procedure to formulate and solve the traveling baseball fan problem, which complicates the traveling salesman problem by incorporating scheduling constraints: a baseball fan must visit each of the 30 Major League ballparks exactly once, and each visit must include watching a scheduled Major League game. The objective is to minimize the time between the start of the first attended game and the end of the last attended game. One natural integer programming formulation involves a binary decision variable for each scheduled game, indicating whether the fan attends. But a reformulation as a side-constrained network flow problem yields much better solver performance.

INTRODUCTION

Cleary et al. (2000) introduced the traveling baseball fan problem and described a greedy heuristic that produced a schedule for the 2000 Major League season that enabled a fan to complete the task in 41 days. More recently, Chuck Booth (Booth, Landgren, and Lee 2011) worked out a 24-day schedule by hand for the 2009 season and actually attended the games, breaking a Guinness world record. In 2012, he broke his own record by completing a 23-day schedule. In both cases, Booth used a variety of modes of transportation, including airplanes, trains, rental cars, subways, and taxicabs. In 2008, Josh Robbins (Major League Baseball 2008) completed a 26-day schedule, traveling only by car.

Instead of using a greedy heuristic or hand calculations, a more rigorous approach to finding an optimal schedule is to formulate the problem in terms of mixed integer linear programming and call an optimization solver. The OPTMODEL procedure in SAS/OR provides an algebraic modeling language and access to several optimization solvers. This paper explores two main mathematical programming formulations, and in doing so it illustrates the flexibility of PROC OPTMODEL to declare and modify formulations to improve solver performance or incorporate more sophistication into the mathematical model.

ASSUMPTIONS

This paper initially makes the following assumptions:

- Every game has a known duration, which includes buffer time for extra innings, rain delays, and so on.
- No game is canceled.
- Travel times between stadiums are known and do not depend on time of day.
- There are no traffic delays.

In the section “[EXTENSIONS](#)” on page 25, several problem extensions address how you can handle violations of these assumptions.

DATA

Major League Baseball has 30 teams, with stadiums throughout the United States plus one in Canada (Toronto). During the regular season (usually April through September¹), each team plays 162 games, for a total of $30 \cdot 162 / 2 = 2,430$ games.² The input data that you need for the problem are date, starting time, location, and duration of each game, and travel times between stadiums.

Figure 1 shows the first few observations from the Game_Data data set, which contains game-related input data for the games in the 2014 regular season.

Figure 1 Input Data: Game_Data

START_DATE	START_TIME_ ET	LOCATION	AWAY_TEAM	HOME_ TEAM
03/30/2014	20:05:00.000	Petco Park	Dodgers	Padres
03/31/2014	13:05:00.000	PNC Park	Cubs	Pirates
03/31/2014	13:08:00.000	Comerica Park	Royals	Tigers
03/31/2014	13:10:00.000	Citi Field	Nationals	Mets
.

The first several PROC OPTMODEL statements declare an index set and several numeric and string parameters for the games and then read the Game_Data data set:

```
proc optmodel;
  set GAMES;
  num start_date {GAMES};
  num start_time {GAMES};
  str location {GAMES};
  str away_team {GAMES};
  str home_team {GAMES};
  read data Game_Data into GAMES=[_N_]
    start_date
    start_time=start_time_et
    location
    away_team
    home_team;
```

The following statements declare and populate parameters that are associated with the start and duration of each game, with time measured in days and time 0 corresponding to the start time of the earliest possible game:

```
num dhms {g in GAMES} = dhms(start_date[g], 0, 0, start_time[g]);
num min_dhms = min {g in GAMES} dhms[g];
num seconds_per_day = 60 * 60 * 24;
num start_datetime {g in GAMES} = (dhms[g] - min_dhms) / seconds_per_day;
num hours_per_game = 4;
num duration {GAMES} = hours_per_game / 24;
```

Note that, unlike many other professional sports, baseball does not use a game clock. Although the game duration is assumed to be constant here (a conservative estimate of four hours for each game), the optimization models that this paper presents do not require this assumption. You could instead read game-dependent durations from a data set if you have more accurate predictions, perhaps based on previous games. For example, games between two teams that have strong offenses might take longer because more pitches are required to complete each inning.

Figure 2 shows the first few observations from the Stadium_Data data set, which contains stadium-related input data for the 2014 regular season.

¹In 2014, the regular season starts on March 22 in Sydney, Australia, and ends on September 28. For obvious reasons, this paper ignores the two games in Australia.

²As of February 20, 2014, 24 games have undetermined starting times and are ignored in this paper.

Figure 2 Input Data: Stadium_Data

location	latitude	longitude	city	st
AT&T Park	37.7783	-122.389	San Francisco	CA
Angel Stadium of Anaheim	33.8003	-117.883	Anaheim	CA
Busch Stadium	38.6225	-90.193	St. Louis	MO
Chase Field	33.4453	-112.067	Phoenix	AZ
.

The following statements declare an index set and parameters for the stadiums and then read the Stadium_Data data set:

```
set <str> STADIUMS;  
num stadium_id {STADIUMS};  
num latitude {STADIUMS};  
num longitude {STADIUMS};  
str city {STADIUMS};  
str state {STADIUMS};  
read data Stadium_Data into STADIUMS=[location]  
    stadium_id=_N_ latitude longitude city state=st;
```

A simple way to get approximate travel times between stadiums is to use the GEODIST function to compute great-circle distances and then convert them to travel times by assuming a constant driving speed, such as 60 miles per hour, as in the following statements (which are not run but are shown for illustration):

```
num miles_per_hour = 60;  
set STADIUM_PAIRS = {s1 in STADIUMS, s2 in STADIUMS: s1 ne s2};  
num miles_between_stadiums {<s1,s2> in STADIUM_PAIRS} =  
    geodist(latitude[s1],longitude[s1],latitude[s2],longitude[s2], 'M');  
num time_between_stadiums {<s1,s2> in STADIUM_PAIRS} =  
    miles_between_stadiums[s1,s2] / (miles_per_hour * 24);
```

A more accurate approach, described in a SAS® Global Forum paper by Zdeb (2010), is to use Google Maps, which accounts for road networks and speed limits.

Figure 3 shows the first few observations from the Travel_Time_Data data set, which contains the travel-time data from Google Maps, with time measured in days.

Figure 3 Input Data: Travel_Time_Data

s1	s2	miles_ between_ stadiums	time_between_ stadiums
AT&T Park	Angel Stadium of Anaheim	408	0.25486
AT&T Park	Busch Stadium	2051	1.20833
AT&T Park	Chase Field	751	0.44861
AT&T Park	Citi Field	2918	1.75000
.	.	.	.

The following statements declare and read the travel-time data:

```
set STADIUM_PAIRS = {s1 in STADIUMS, s2 in STADIUMS: s1 ne s2};  
num miles_between_stadiums {STADIUM_PAIRS};  
num time_between_stadiums {STADIUM_PAIRS};  
read data Travel_Time_Data into [s1 s2] miles_between_stadiums time_between_stadiums;
```

INITIAL FORMULATION

This section describes the initial mixed integer linear programming (MILP) formulation.

VARIABLES, OBJECTIVE, AND CONSTRAINTS

Because the traveling baseball fan will attend a subset of the 2,430 games, it is natural to define a binary decision variable for each game g :

$$\text{Attend}[g] = \begin{cases} 1 & \text{if the baseball fan attends game } g \\ 0 & \text{otherwise} \end{cases}$$

One set of constraints enforces the rule that the fan visits each stadium s exactly once:

$$\sum_{\substack{g \in \text{GAMES}: \\ \text{location}[g] = s}} \text{Attend}[g] = 1$$

Another set of constraints prevents the fan from attending any pair (g_1, g_2) of games whose schedules conflict:

$$\text{Attend}[g_1] + \text{Attend}[g_2] \leq 1$$

The following statements declare these variables and constraints:

```
/* Attend[g] = 1 if attend game g, 0 otherwise */
var Attend {GAMES} binary;

/* visit every stadium exactly once */
con Visit_Once {s in STADIUMS}:
    sum {g in GAMES: location[g] = s} Attend[g] = 1;

/* do not attend games that conflict */
set CONFLICTS = {g1 in GAMES, g2 in GAMES:
    location[g1] ne location[g2]
    and start_datetime[g1] <= start_datetime[g2]
    < start_datetime[g1] + duration[g1]
    + time_between_stadiums[location[g1], location[g2]]};
con Conflict {<g1,g2> in CONFLICTS}:
    Attend[g1] + Attend[g2] <= 1;
```

Because the objective is to minimize the total time between the start of the first attended game and the end of the last attended game, it is convenient to introduce two additional decision variables, **Start** and **End**, which have the following interpretations:

$$\begin{aligned} \text{Start} &= \min_{\substack{g \in \text{GAMES}: \\ \text{Attend}[g] = 1}} \text{start_datetime}[g] \\ \text{End} &= \max_{\substack{g \in \text{GAMES}: \\ \text{Attend}[g] = 1}} (\text{start_datetime}[g] + \text{duration}[g]) \end{aligned}$$

The standard way to use linear constraints to express these relationships is to use so-called big-M constraints. In particular, if $\text{Attend}[g] = 1$, the constraints should enforce $\text{Start} \leq \text{start_datetime}[g]$. Similarly, if $\text{Attend}[g] = 1$, the constraints should enforce $\text{End} \geq \text{start_datetime}[g] + \text{duration}[g]$.

The following statements declare these variables, objective, and constraints, along with a named problem (InitialFormulation), because this same PROC OPTMODEL session will contain another formulation that is described in the section “[NETWORK FORMULATION](#)” on page 12:

```

/* declare start of first game and end of last game */
var Start
    >= min {g in GAMES} start_datetime[g]
    <= max {g in GAMES} start_datetime[g];
var End
    >= min {g in GAMES} (start_datetime[g] + duration[g])
    <= max {g in GAMES} (start_datetime[g] + duration[g]);

/* minimize total time between start of first game and end of last game (in days) */
min TotalTime = End - Start;

/* if Attend[g] = 1 then Start <= start_datetime[g] */
con Start_def {g in GAMES}:
    Start - start_datetime[g]
<= (Start.ub - start_datetime[g]) * (1 - Attend[g]);

/* if Attend[g] = 1 then End >= start_datetime[g] + duration[g] */
con End_def {g in GAMES}:
    -End + start_datetime[g] + duration[g]
<= (-End.lb + start_datetime[g] + duration[g]) * (1 - Attend[g]);

problem InitialFormulation include
    Attend Conflict Visit_Once Start End TotalTime Start_def End_def;
use problem InitialFormulation;

```

The following statement calls the MILP solver and imposes a one-hour time limit:

```

solve with MILP / logfreq=100000 maxtime=3600;

```

Figure 4 shows the log that results from calling the MILP solver for this initial big-M formulation.

Figure 4 MILP Solver Log for Initial Big-M Formulation

```

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2406 variables (0 free, 0 fixed).
NOTE: The problem has 2404 binary and 0 integer variables.
NOTE: The problem has 35999 linear constraints (35969 LE, 30 EQ, 0 GE, 0 range).
NOTE: The problem has 74338 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 1922 constraints.
NOTE: The MILP presolver removed 2265 constraint coefficients.
NOTE: The MILP presolver modified 6526 constraint coefficients.
NOTE: The presolved problem has 2406 variables, 34077 constraints, and 72073
      constraint coefficients.
NOTE: The MILP solver is called.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	-178.1539881	.	5
0	1	0	.	-177.9400607	.	12
0	1	0	.	-177.9400607	.	18
0	1	0	.	-177.9400607	.	23
0	1	0	.	-177.9400607	.	29
0	1	0	.	-177.9400607	.	47

```

NOTE: The MILP solver added 574 cuts with 17302 cut coefficients at the root.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
3	3	1	162.1250000	-177.9400607	191.11%	57
70	36	2	161.0416667	-175.9994973	191.50%	157
106	54	3	120.8784722	-174.9358673	169.10%	198
194	98	4	109.0416667	-159.6774080	168.29%	280
228	115	5	77.4583333	-123.9041373	162.51%	297
745	297	6	73.9201389	0.1666667	44252.1%	314
842	76	7	39.1631944	0.1666667	23397.9%	372
869	76	8	37.1631944	0.1666667	22197.9%	425
5936	3665	9	35.1631944	0.1666667	20997.9%	482
8806	4510	10	35.0381944	0.1666667	20922.9%	521
10615	4526	11	34.1458333	0.1666667	20387.5%	549
18717	5249	12	33.4166667	0.1666667	19950.0%	722
18787	5175	13	32.9131944	0.1666667	19647.9%	725
47269	7579	14	32.0381944	1.1215278	2756.66%	1297
48292	7666	15	31.9166667	1.1666667	2635.71%	1326
100000	29846	15	31.9166667	2.3958333	1232.17%	2062
116365	39839	16	30.7881944	2.9166667	955.60%	2223
200000	99341	16	30.7881944	3.1215278	886.32%	2861
300000	174572	16	30.7881944	3.4166667	801.12%	3564
303185	177087	16	30.7881944	3.4201389	800.20%	3585

```

NOTE: CPU time limit reached.
NOTE: Objective of the best integer solution found = 30.788194444.

```

As the log shows, the optimality gap is still large after one hour. It turns out that the MILP solver would eventually run out of memory for this instance. Furthermore, it takes a significant amount of time even to deduce a positive lower bound.

AVOIDING THE BIG-M CONSTRAINTS

Big-M constraints are notorious for yielding weak linear programming relaxations. In the presence of the Visit_Once constraints, you can express the same logical relationships among the **Attend**_[g], **Start**, and **End** variables by instead using two smaller (and stronger) sets of constraints. The following statements drop the big-M constraints, replace them with these stronger constraints, and call the MILP solver:

```

drop Start_def End_def;

/* if Attend[g] = 1 then Start <= start_datetime[g] */
con Start_def2 {s in STADIUMS}:
    Start <= sum {g in GAMES: location[g] = s} start_datetime[g] * Attend[g];

/* if Attend[g] = 1 then End >= start_datetime[g] + duration[g] */
con End_def2 {s in STADIUMS}:
    End >= sum {g in GAMES: location[g] = s} (start_datetime[g] + duration[g]) * Attend[g];

solve with MILP / logfreq=100000 maxtime=3600;

```

Figure 5 shows the log that results from calling the MILP solver for this improved initial formulation.

Figure 5 MILP Solver Log for Improved Initial Formulation

```

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2406 variables (0 free, 0 fixed).
NOTE: The problem has 2404 binary and 0 integer variables.
NOTE: The problem has 31251 linear constraints (31191 LE, 30 EQ, 30 GE, 0
range).
NOTE: The problem has 69593 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 4808 dropped constraints.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 1918 constraints.
NOTE: The MILP presolver removed 2261 constraint coefficients.
NOTE: The MILP presolver modified 6526 constraint coefficients.
NOTE: The presolved problem has 2406 variables, 29333 constraints, and 67332
constraint coefficients.
NOTE: The MILP solver is called.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	0.1666667	.	2
0	1	1	37.1284722	0.1666667	22177.1%	3
0	1	1	37.1284722	0.1666667	22177.1%	3
0	1	1	37.1284722	0.1666667	22177.1%	3
0	1	1	37.1284722	0.1666667	22177.1%	6

```

NOTE: The MILP solver added 36 cuts with 240 cut coefficients at the root.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
181	94	2	35.1666667	0.1666667	21000.0%	24
227	117	3	34.8750000	0.1666667	20825.0%	25
11923	6606	4	34.1250000	0.1666667	20375.0%	167
44211	19225	5	33.1284722	0.1666667	19777.1%	760
69020	16237	6	33.1250000	0.1666667	19775.0%	1993
100000	23495	6	33.1250000	0.1666667	19775.0%	2509
146898	38239	7	32.1909722	0.1666667	19214.6%	3212
173247	47049	7	32.1909722	0.1666667	19214.6%	3599

```

NOTE: CPU time limit reached.
NOTE: Objective of the best integer solution found = 32.190972222.

```

As the log shows, the optimality gap is still large after one hour, but the MILP solver finds a positive lower bound right away.

A LOWER BOUND

You can improve the formulation by including one optional cut constraint that captures the fact that the total time must include watching a game at each stadium:

$$\text{End} - \text{Start} \geq \sum_{s \in \text{STADIUMS}} \min_{\substack{g \in \text{GAMES:} \\ \text{location}[g] = s}} \text{duration}[g]$$

The following statements declare this optional cut and call the MILP solver:

```
con Cut:
    End - Start >= sum {s in STADIUMS} min {g in GAMES: location[g] = s} duration[g];

solve with MILP / logfreq=100000 maxtime=3600;
```

Figure 6 shows the log that results from calling the MILP solver and including this optional cut.

Figure 6 MILP Solver Log for Initial Formulation with Optional Cut

NOTE: Problem generation will use 4 threads.							
NOTE: The problem has 2406 variables (0 free, 0 fixed).							
NOTE: The problem has 2404 binary and 0 integer variables.							
NOTE: The problem has 31252 linear constraints (31191 LE, 30 EQ, 31 GE, 0 range).							
NOTE: The problem has 69595 linear constraint coefficients.							
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).							
NOTE: The problem has 4808 dropped constraints.							
NOTE: The MILP presolver value AUTOMATIC is applied.							
NOTE: The MILP presolver removed 0 variables and 1918 constraints.							
NOTE: The MILP presolver removed 2261 constraint coefficients.							
NOTE: The MILP presolver modified 6599 constraint coefficients.							
NOTE: The presolved problem has 2406 variables, 29334 constraints, and 67334 constraint coefficients.							
NOTE: The MILP solver is called.							
Node	Active	Sols	BestInteger	BestBound	Gap	Time	
0	1	0	.	5.0000000	.	2	
0	1	1	39.2534722	5.0000000	685.07%	3	
0	1	1	39.2534722	5.0000000	685.07%	3	
0	1	1	39.2534722	5.0000000	685.07%	3	
0	1	2	36.9131944	5.0000000	638.26%	3	
0	1	2	36.9131944	5.0000000	638.26%	5	
NOTE: The MILP solver added 43 cuts with 587 cut coefficients at the root.							
3285	1883	4	33.8958333	5.0000000	577.92%	54	
100000	79795	4	33.8958333	5.0000000	577.92%	740	
200000	158418	4	33.8958333	5.0000000	577.92%	1429	
275992	210547	5	33.1284722	5.0000000	562.57%	2035	
300000	228983	5	33.1284722	5.0000000	562.57%	2173	
334064	254912	6	32.3784722	5.0000000	547.57%	2459	
400000	307987	6	32.3784722	5.0000000	547.57%	2907	
423595	325770	7	32.2916667	5.0000000	545.83%	3111	
438732	335411	8	31.1284722	5.0000000	522.57%	3242	
456722	348813	9	31.0416667	5.0000000	520.83%	3347	
482881	366978	9	31.0416667	5.0000000	520.83%	3599	
NOTE: CPU time limit reached.							
NOTE: Objective of the best integer solution found = 31.041666667.							

Again, the optimality gap is still large after one hour. But the MILP solver finds a better lower bound right away.

A BETTER LOWER BOUND FROM TSP

You can push this idea further by also including a lower bound on the total travel time between stadiums. To obtain such a bound, you can solve the traveling salesman problem (TSP) on a graph that contains one node per stadium, as well as a dummy node (with zero-cost edges to the other nodes) that represents the start and end of the tour. This approach is a standard way to solve a traveling salesman problem variant in which the traveler is allowed to start and end in different cities.

The following statements create this graph and solve the TSP by using the TSP option in the new SOLVE WITH NETWORK statement available in SAS/OR 13.1:


```

/* solve TSP with dummy node to get lower bound on travel time */
set TSP_NODES = {'dummy'} union STADIUMS;
set TSP_EDGES = {i in TSP_NODES, j in TSP_NODES: i ne j};
num tsp_weight {<i,j> in TSP_EDGES} =
    if 'dummy' in {i,j} then 0
    else min(time_between_stadiums[i,j],time_between_stadiums[j,i]);
set <str,str> TOUR;
solve with NETWORK / links=(weight=tsp_weight) tsp out=(tour=TOUR);
put TOUR=;
num tsp_lower_bound = sum {<i,j> in TOUR} tsp_weight[i,j];
num tsp_miles = sum {<i,j> in TOUR: 'dummy' not in {i,j}}
    min(miles_between_stadiums[i,j],miles_between_stadiums[j,i]);
put 'miles: ' tsp_miles;

```

Figure 7 shows the log that results from solving the TSP.

Figure 7 TSP Log

```

NOTE: The experimental Network solver is used.
NOTE: The number of nodes in the input graph is 31.
NOTE: The number of links in the input graph is 465.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 5.5159722222 using 0.17
      (cpu: 0.05) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The MILP solver added 11 cuts with 462 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 5.5159722222.
NOTE: Processing the traveling salesman problem used 0.19 (cpu: 0.06) seconds.
TOUR={<'dummy', 'Marlins Park'>,<'Marlins Park', 'Tropicana Field'>,<
'Tropicana Field', 'Turner Field'>,<'Turner Field', 'Great American Ball Park'>,<
'Great American Ball Park', 'Progressive Field'>,<'Progressive Field', 'PNC Park'>
,<'PNC Park', 'Nationals Park'>,<'Nationals Park', 'Oriole Park at Camden Yards'>,
<'Oriole Park at Camden Yards', 'Citizens Bank Park'>,<'Citizens Bank Park',
'Yankee Stadium'>,<'Yankee Stadium', 'Citi Field'>,<'Citi Field', 'Fenway Park'>,<
'Fenway Park', 'Rogers Centre'>,<'Rogers Centre', 'Comerica Park'>,<
'Comerica Park', 'U.S. Cellular Field'>,<'U.S. Cellular Field', 'Wrigley Field'>,<
'Wrigley Field', 'Miller Park'>,<'Miller Park', 'Target Field'>,<'Target Field',
'Kauffman Stadium'>,<'Kauffman Stadium', 'Busch Stadium'>,<'Busch Stadium',
'Minute Maid Park'>,<'Minute Maid Park', 'Globe Life Park in Arlington'>,<
'Globe Life Park in Arlington', 'Coors Field'>,<'Coors Field', 'Chase Field'>,<
'Chase Field', 'Petco Park'>,<'Petco Park', 'Angel Stadium of Anaheim'>,<
'Angel Stadium of Anaheim', 'Dodger Stadium'>,<'Dodger Stadium', 'O.co Coliseum'>,
<'O.co Coliseum', 'AT&T Park'>,<'AT&T Park', 'Safeco Field'>,<'Safeco Field',
'dummy'>}
miles: 8756.5

```

The following statements first strengthen the original cut by increasing the right-hand side by the TSP lower bound and then call the MILP solver:

```

Cut.lb = Cut.lb + tsp_lower_bound;

solve with MILP / logfreq=100000 maxtime=3600;

```

Figure 8 shows the log that results from calling the MILP solver and including this strengthened cut.

Figure 8 MILP Solver Log for Initial Formulation with Strengthened Cut

```

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2406 variables (0 free, 0 fixed).
NOTE: The problem has 2404 binary and 0 integer variables.
NOTE: The problem has 31252 linear constraints (31191 LE, 30 EQ, 31 GE, 0
range).
NOTE: The problem has 69595 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 4808 dropped constraints.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 1918 constraints.
NOTE: The MILP presolver removed 2261 constraint coefficients.
NOTE: The MILP presolver modified 6675 constraint coefficients.
NOTE: The presolved problem has 2406 variables, 29334 constraints, and 67334
constraint coefficients.
NOTE: The MILP solver is called.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	10.5159722	.	2
0	1	1	36.2500000	10.5159722	244.71%	3
0	1	1	36.2500000	10.5159722	244.71%	3
0	1	1	36.2500000	10.5159722	244.71%	3
0	1	2	33.1284722	10.5159722	215.03%	3
0	1	2	33.1284722	10.5159722	215.03%	5

```

NOTE: The MILP solver added 42 cuts with 656 cut coefficients at the root.
78503 70026 3 33.1236111 10.5159722 214.98% 285
100000 89874 3 33.1236111 10.5159722 214.98% 349
200000 178584 3 33.1236111 10.5159722 214.98% 653
260577 232560 4 32.0437500 10.5159722 204.72% 844
300000 268520 4 32.0437500 10.5159722 204.72% 964
400000 353880 4 32.0437500 10.5159722 204.72% 1290
500000 445483 4 32.0437500 10.5159722 204.72% 1585
600000 535926 4 32.0437500 10.5159722 204.72% 1963
700000 627542 4 32.0437500 10.5159722 204.72% 2336
800000 719088 4 32.0437500 10.5159722 204.72% 2705
900000 810134 4 32.0437500 10.5159722 204.72% 3085
1000000 898860 4 32.0437500 10.5159722 204.72% 3500
1024356 920765 4 32.0437500 10.5159722 204.72% 3599
NOTE: CPU time limit reached.
NOTE: Objective of the best integer solution found = 32.04375.

```

Again, the optimality gap is still large after one hour. But the MILP solver finds a better lower bound right away.

CLIQUEES

Another way to strengthen this initial formulation is to tighten the Conflict constraints by using cliques. For example, if x is a collection of binary variables, you can replace conflict constraints such as

$$x_i + x_j \leq 1, \quad x_i + x_k \leq 1, \quad x_j + x_k \leq 1$$

with one (stronger) clique constraint:

$$x_i + x_j + x_k \leq 1$$

The MILP solver automatically looks for these “clique cuts” during the branch-and-cut algorithm, but sometimes you can reduce the solve time by explicitly including all such cuts a priori in the formulation.

The following statements drop the Conflict constraints, find the maximal cliques by calling the network solver available in SAS/OR 13.1, declare the Clique constraints (one per maximal clique), and call the MILP solver:

```

drop Conflict;
set <num,num> ID_NODE;
solve with NETWORK / links=(include=CONFLICTS) clique out=(cliques=ID_NODE);
set CLIQUES init {};
set GAMES_c {CLIQUES} init {};
for {<c,g> in ID_NODE} do;
    CLIQUES = CLIQUES union {c};
    GAMES_c[c] = GAMES_c[c] union {g};
end;
con Clique {c in CLIQUES}:
    sum {g in GAMES_c[c]} Attend[g] <= 1;

solve with MILP / logfreq=100000 maxtime=3600;

```

Figure 9 shows the log that results from calling the MILP solver and including the strengthened cut and cliques.

Figure 9 MILP Solver Log for Initial Formulation with Strengthened Cut and Cliques

```

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2406 variables (0 free, 0 fixed).
NOTE: The problem has 2404 binary and 0 integer variables.
NOTE: The problem has 4858 linear constraints (4797 LE, 30 EQ, 31 GE, 0 range).
NOTE: The problem has 48773 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 35969 dropped constraints.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 10 constraints.
NOTE: The MILP presolver added 70 constraint coefficients.
NOTE: The MILP presolver modified 336 constraint coefficients.
NOTE: The presolved problem has 2406 variables, 4848 constraints, and 48843
      constraint coefficients.
NOTE: The MILP solver is called.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	1	43.4201389	-160.9715278	126.97%	0
0	1	1	43.4201389	10.5159722	312.90%	0
0	1	2	36.2500000	10.5159722	244.71%	2
0	1	2	36.2500000	10.5159722	244.71%	2
0	1	2	36.2500000	10.5159722	244.71%	2
0	1	2	36.2500000	10.5159722	244.71%	3

```

NOTE: The MILP solver added 23 cuts with 1250 cut coefficients at the root.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
1000	248	3	35.5451389	10.5159722	238.01%	25
2040	1263	4	35.1701389	10.5159722	234.44%	36
4543	3708	5	34.1284722	10.5159722	224.54%	55
5038	4181	6	29.1666667	10.5159722	177.36%	62
23227	21900	7	28.0833333	10.5159722	167.05%	184
28111	26616	8	27.8763889	10.5159722	165.09%	209
100000	94531	8	27.8763889	10.5159722	165.09%	551
200000	189289	8	27.8763889	10.5159722	165.09%	1366
214032	202341	9	27.1250000	10.5159722	157.94%	1406
300000	283048	9	27.1250000	10.5159722	157.94%	1645
400000	378370	9	27.1250000	10.5159722	157.94%	1791
500000	470490	9	27.1250000	10.5159722	157.94%	2131
600000	562517	9	27.1250000	10.5159722	157.94%	2443
700000	656979	9	27.1250000	10.5159722	157.94%	2623
800000	750384	9	27.1250000	10.5159722	157.94%	2781
900000	843916	9	27.1250000	10.5159722	157.94%	2919
1000000	937607	9	27.1250000	10.5159722	157.94%	3087
1100000	1033437	9	27.1250000	10.5159722	157.94%	3204
1200000	1124936	9	27.1250000	10.5159722	157.94%	3479
1285109	1204501	9	27.1250000	10.5159722	157.94%	3599

```

NOTE: CPU time limit reached.
NOTE: Objective of the best integer solution found = 27.125.

```

Again, the optimality gap is still large after one hour. Because this formulation is tighter and has fewer constraints, the node throughput is higher. But it turns out that the MILP solver would eventually run out of memory for this instance. It seems that a completely different approach is needed.

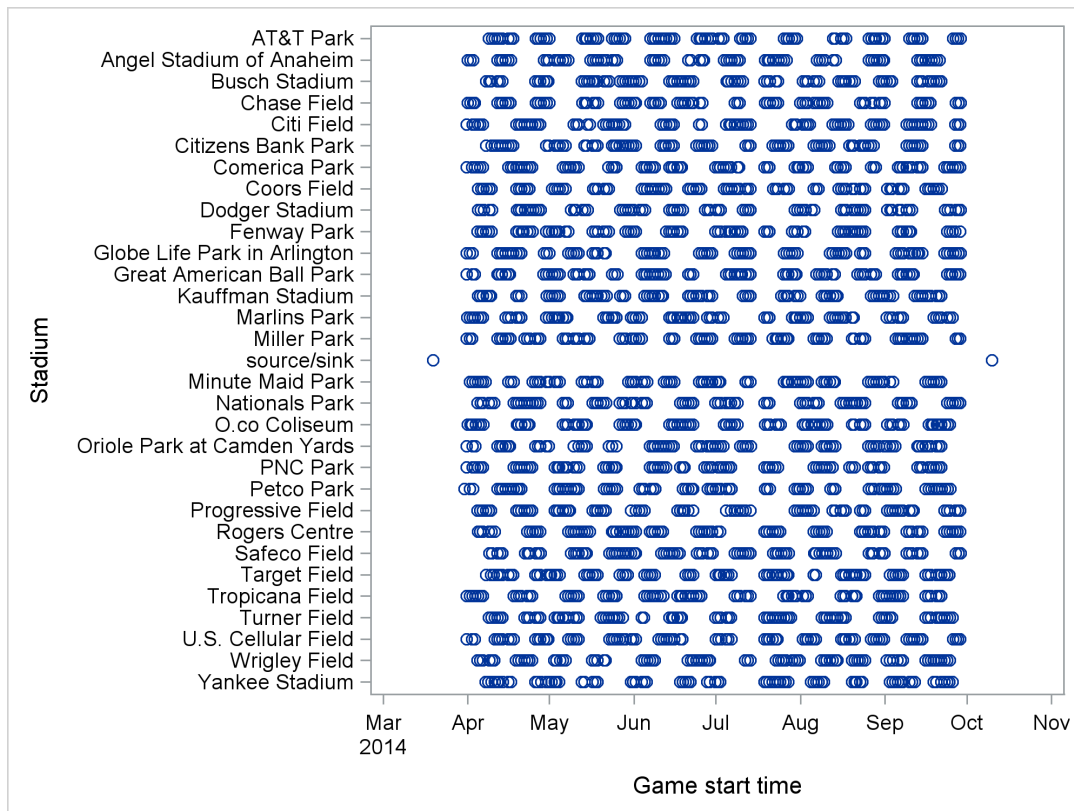
NETWORK FORMULATION

This section describes a network-based integer programming formulation that performs much better than the initial formulation discussed in the previous section.

NETWORK DEFINITION

The network contains one node per game, plus a dummy source node and a dummy sink node to represent the start and end of the schedule, respectively. Figure 10 shows the nodes in the network, with game start time along the horizontal axis and stadium along the vertical axis.

Figure 10 Network-Based Formulation: Nodes



The (directed) arcs in the network are of three types:

- from the source node to each game node
- from each game node to the sink node
- from game g_1 to game g_2 if it is possible to attend game g_1 and then game g_2

The cost for each arc is the additional time incurred from the end of game g_1 to the end of game g_2 . The goal is to find a shortest path that starts at the source, visits every stadium, and ends at the sink.

The following statements declare the source, sink, and set of nodes:

```
num source = 0;
num sink = 1 + max {g in GAMES} g;
set NODES = GAMES union {source,sink};
```

The following statement (which is not run but is shown for illustration) is a simple way to declare the set of arcs, but the resulting optimization model would be unnecessarily large and highly degenerate:

```
set ARCS init {g1 in GAMES, g2 in GAMES: location[g1] ne location[g2] and
    start_datetime[g1] + duration[g1]
    + seconds_between_stadiums[location[g1],location[g2]]
    <= start_datetime[g2]};
```

Without loss of optimality, for a given node you can instead consider only the shortest feasible arc to each stadium. The following statements declare this smaller set of arcs and the cost per arc:

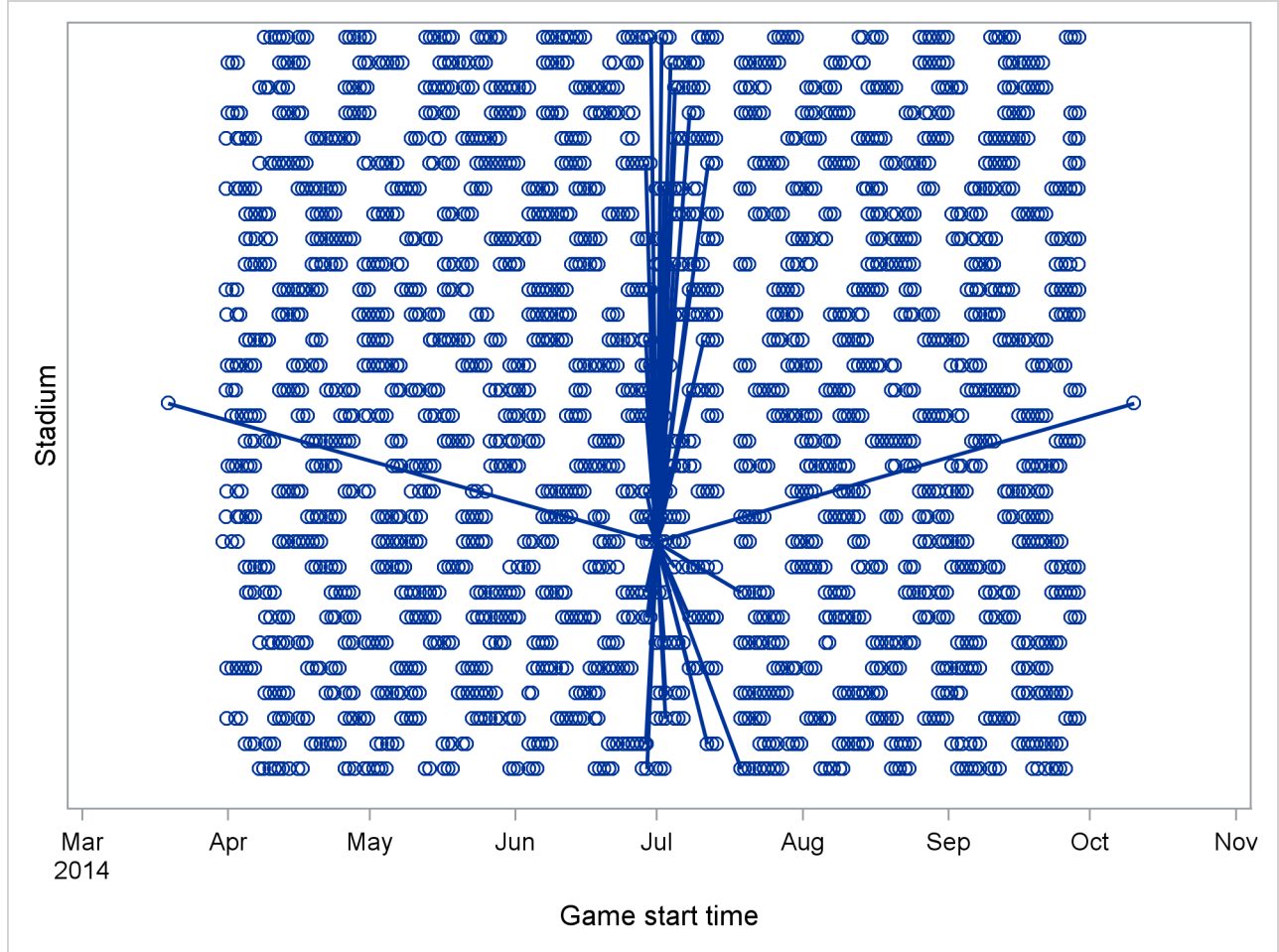
```

/* for each game and stadium, include only the shortest feasible arc */
num infinity = constant('BIG');
set <num,num> ARCS init {};
num min {STADIUMS};
num argmin {STADIUMS};
str loc_g1, loc_g2;
for {g1 in GAMES} do;
    loc_g1 = location[g1];
    for {s in STADIUMS} do;
        min[s] = infinity;
        argmin[s] = -1;
    end;
    for {g2 in GAMES} do;
        loc_g2 = location[g2];
        if loc_g1 ne loc_g2
            and start_datetime[g1] + duration[g1]
                + time_between_stadiums[loc_g1,loc_g2]
                <= start_datetime[g2]
            and min[loc_g2] > start_datetime[g2] then do;
                min[loc_g2] = start_datetime[g2];
                argmin[loc_g2] = g2;
            end;
        end;
    ARCS = ARCS union (setof {s in STADIUMS: argmin[s] ne -1} <g1,argmin[s]>);
end;
/* include source and sink */
ARCS = ARCS union ({source} cross GAMES) union (GAMES cross {sink});
/* cost = start2 - end1 + duration2 */
num cost {<g1,g2> in ARCS} =
    (if g1 ne source and g2 ne sink then start_datetime[g2]
        - (start_datetime[g1] + duration[g1]))
    +
    (if g2 ne sink then duration[g2]);

```

Figure 11 shows the arcs to and from an arbitrary game node in the network.

Figure 11 Network-Based Formulation: Arcs to and from an Arbitrary Game



VARIABLES, OBJECTIVE, AND CONSTRAINTS

In this network formulation, you have a binary decision variable for each arc:

$$\text{UseArc}[g_1, g_2] = \begin{cases} 1 & \text{if the fan attends games } g_1 \text{ and } g_2 \text{ and no game in between} \\ 0 & \text{otherwise} \end{cases}$$

As before, the objective is to minimize the total time, which is expressed here as a sum of the arc costs:

$$\sum_{(g_1, g_2) \in \text{ARCS}} \text{cost}[g_1, g_2] \text{UseArc}[g_1, g_2]$$

One set of constraints enforces flow balance at every node g :

$$\sum_{(g, g_2) \in \text{ARCS}} \text{UseArc}[g, g_2] - \sum_{(g_1, g) \in \text{ARCS}} \text{UseArc}[g_1, g] = \begin{cases} 1 & g = \text{source} \\ -1 & g = \text{sink} \\ 0 & \text{otherwise} \end{cases}$$

Another set of constraints enforces the rule that the fan visits each stadium s exactly once:

$$\sum_{\substack{(g_1, g_2) \in \text{ARCS}: \\ g_2 \neq \text{sink and location}[g_2] = s}} \text{UseArc}[g_1, g_2] = 1$$

The following statements declare these variables, objective, and constraints, as well as a named problem (NetworkFormulation):

```

/* UseArc[g1,g2] = 1 if attend games g1 and g2 and no game in between, 0 otherwise */
var UseArc {ARCS} binary;

/* minimize total time between start of first game and end of last game */
min TotalTime_Network = sum {<g1,g2> in ARCS} cost[g1,g2] * UseArc[g1,g2];

/* flow balance at every node */
con Balance {g in NODES}:
    sum {<(g),g2> in ARCS} UseArc[g,g2] - sum {<g1,(g)> in ARCS} UseArc[g1,g]
    = (if g = source then 1 else if g = sink then -1 else 0);

/* visit every stadium exactly once */
con Visit_Once_Network {s in STADIUMS}:
    sum {<g1,g2> in ARCS: g2 ne sink and location[g2] = s} UseArc[g1,g2] = 1;

problem NetworkFormulation include
    UseArc TotalTime_Network Balance Visit_Once_Network;
use problem NetworkFormulation;

```

This network-based formulation models the problem as an integer network flow problem (shortest path in a directed acyclic network) that has relatively few side constraints. The Conflict constraints in the initial formulation correspond to a removal of arcs (variables) in this formulation. Although the number of variables increases from $O(|\text{GAMES}|)$ to $O(|\text{GAMES}| \cdot |\text{STADIUMS}|)$, the network-based formulation is solved successfully without running out of memory.

The following statement calls the MILP solver to minimize **TotalTime_Network**:

```
solve with MILP / logfreq=100000;
```

Figure 12 shows the log that results from calling the MILP solver for the network formulation.

Figure 12 MILP Solver Log for Network Formulation

```

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 72747 variables (0 free, 0 fixed).
NOTE: The problem has 72747 binary and 0 integer variables.
NOTE: The problem has 2436 linear constraints (0 LE, 2436 EQ, 0 GE, 0 range).
NOTE: The problem has 215837 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 72747 variables, 2436 constraints, and 215837
      constraint coefficients.
NOTE: The MILP solver is called.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	0	.	23.2615355	.	20
0	1	1	29.2500000	23.2615355	25.74%	44
0	1	1	29.2500000	23.2615355	25.74%	60
256	27	2	24.1631944	23.6264785	2.27%	2699
381	0	2	24.1631944	24.1631944	0.00%	2893

```

NOTE: Optimal.
NOTE: Objective = 24.163194444.

```

The solver returns an optimal solution in about 50 minutes. The resulting schedule completes the 30 games in 24.2 days of real time, or 25 calendar days.

By using the PARALLEL=1 option (experimental in SAS/OR 13.1), you can run a threaded version of the MILP solver to reduce the run time:


```
solve with MILP / logfreq=100000 parallel=1;
```

Figure 13 shows the log that results from calling the MILP solver for the network formulation and using the PARALLEL=1 option.

Figure 13 MILP Solver Log for Network Formulation with PARALLEL=1 Option

NOTE: The experimental parallel Branch and Cut algorithm is used.							
NOTE: The Branch and Cut algorithm is using up to 4 threads.							
Node	Active	Sols	BestInteger	BestBound	Gap	Time	
0	1	0	.	23.2615355	.	23	
0	1	1	29.2500000	23.2615355	25.74%	46	
477	406	2	25.9145833	23.5790094	9.91%	1202	
523	327	3	24.1631944	23.5790094	2.48%	1284	
718	0	3	24.1631944	24.1631944	0.00%	1685	
NOTE: Optimal.							
NOTE: Objective = 24.163194444.							

The solver returns an optimal solution in about 28 minutes. For another way to use parallel processing to solve this problem faster in SAS/OR 13.1, see the section “[COFOR STATEMENT](#)” on page 24.

The following statements declare sets and create the output data set Schedule1, which corresponds to the optimal solution that the solver returns:

```
set PATH = {<g1,g2> in ARCS: UseArc[g1,g2].sol > 0.5};
set SOLUTION = {<g1,g2> in PATH: g2 ne sink};

create data Schedule1(drop=g1) from [g1 g]=SOLUTION
  location[g] away_team[g] home_team[g] city[location[g]] state[location[g]]
  start_datetime=dhms[g]/format=datetime14.
  latitude[location[g]] longitude[location[g]];
```

Figure 14 shows the resulting optimal path in the network, Figure 15 shows the resulting optimal schedule, and Figure 16 shows the resulting optimal schedule geographically.

Figure 14 Network-Based Formulation: Optimal Path

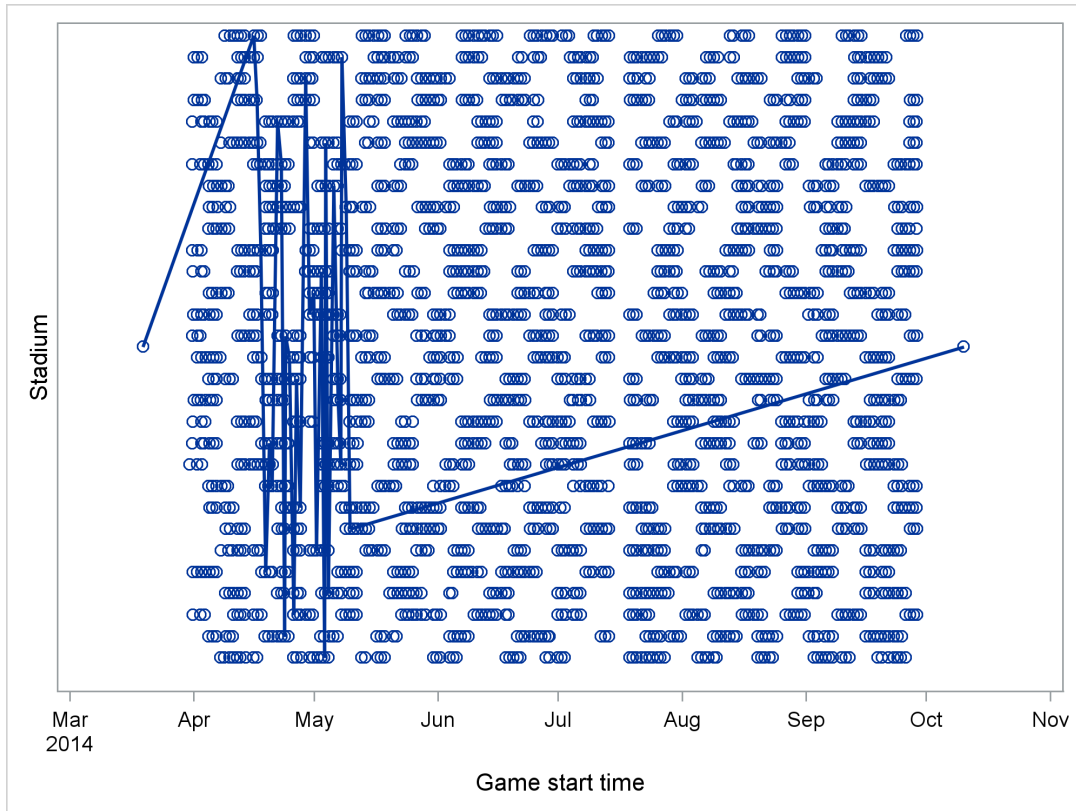
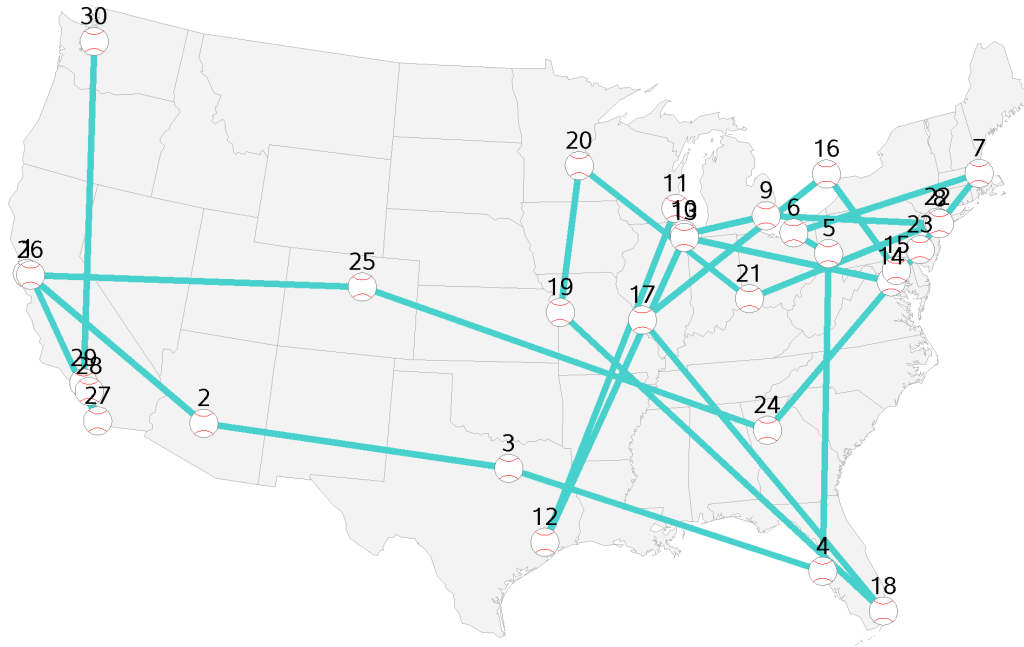


Figure 15 Network-Based Formulation: Optimal Schedule

Obs	location	away_team	home_team	city	state	start_datetime
1	AT&T Park	Dodgers	Giants	San Francisco	CA	15APR14:22:15
2	Chase Field	Mets	D-backs	Phoenix	AZ	16APR14:15:40
3	Globe Life Park in Arlington	Mariners	Rangers	Arlington	TX	17APR14:14:05
4	Tropicana Field	Yankees	Rays	St. Petersburg	FL	18APR14:19:10
5	PNC Park	Brewers	Pirates	Pittsburgh	PA	19APR14:19:05
6	Progressive Field	Blue Jays	Indians	Cleveland	OH	20APR14:13:05
7	Fenway Park	Orioles	Red Sox	Boston	MA	21APR14:11:05
8	Citi Field	Cardinals	Mets	Flushing	NY	21APR14:19:10
9	Comerica Park	White Sox	Tigers	Detroit	MI	22APR14:19:08
10	Wrigley Field	D-backs	Cubs	Chicago	IL	23APR14:14:20
11	Miller Park	Padres	Brewers	Milwaukee	WI	23APR14:20:10
12	Minute Maid Park	Athletics	Astros	Houston	TX	24APR14:20:10
13	U.S. Cellular Field	Rays	White Sox	Chicago	IL	25APR14:20:10
14	Nationals Park	Padres	Nationals	Washington	DC	26APR14:13:05
15	Oriole Park at Camden Yards	Royals	Orioles	Baltimore	MD	26APR14:19:05
16	Rogers Centre	Red Sox	Blue Jays	Toronto	ON	27APR14:13:07
17	Busch Stadium	Brewers	Cardinals	St. Louis	MO	28APR14:20:15
18	Marlins Park	Braves	Marlins	Miami	FL	29APR14:19:10
19	Kauffman Stadium	Blue Jays	Royals	Kansas City	KS	30APR14:20:10
20	Target Field	Dodgers	Twins	Minneapolis	MN	01MAY14:13:10
21	Great American Ball Park	Brewers	Reds	Cincinnati	OH	02MAY14:19:10
22	Yankee Stadium	Rays	Yankees	Bronx	NY	03MAY14:13:05
23	Citizens Bank Park	Nationals	Phillies	Philadelphia	PA	03MAY14:19:05
24	Turner Field	Giants	Braves	Atlanta	GA	04MAY14:13:35
25	Coors Field	Rangers	Rockies	Denver	CO	05MAY14:20:40
26	O.co Coliseum	Mariners	Athletics	Oakland	CA	06MAY14:22:05
27	Petco Park	Royals	Padres	San Diego	CA	07MAY14:15:40
28	Angel Stadium of Anaheim	Yankees	Angels	Anaheim	CA	07MAY14:22:05
29	Dodger Stadium	Giants	Dodgers	Los Angeles	CA	08MAY14:22:10
30	Safeco Field	Royals	Mariners	Seattle	WA	09MAY14:22:10

Figure 16 Network-Based Formulation: Map of Optimal Schedule



A SECONDARY OBJECTIVE

Because the total time is determined by only the first and last games attended, the problem usually has multiple optimal solutions. A natural way to break ties among solutions that have the same minimum total time is to introduce a secondary objective to minimize the total distance traveled.

The following statements declare a constraint on the primary objective and declare **TotalDistance** as a secondary objective:

```
num minTotalTime;
minTotalTime = TotalTime_Network.sol;
con TotalTime_con:
    TotalTime_Network <= minTotalTime;

num distance {<g1,g2> in ARCS} =
    (if g1 = source or g2 = sink then 0
     else miles_between_stadiums[location[g1],location[g2]]);
min TotalDistance = sum {<g1,g2> in ARCS} distance[g1,g2] * UseArc[g1,g2];
```

The following statement calls the MILP solver to minimize the secondary objective and uses the PRIMALIN option because the solution from the previous solver call is a good starting solution:

```
solve with MILP / logfreq=100000 parallel=1 primalin;
```

Figure 17 shows the log that results from calling the MILP solver for the network formulation to minimize the secondary objective.

Figure 17 MILP Solver Log for Network Formulation with Secondary Objective

```
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 72747 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 72747 binary and 0 integer variables.
NOTE: The problem has 2437 linear constraints (1 LE, 2436 EQ, 0 GE, 0 range).
NOTE: The problem has 286180 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 72747 variables, 2437 constraints, and 286180
constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The experimental parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	1	20373.2000000	9892.8041634	105.94%	16
281	109	2	19099.2000000	12734.5262553	49.98%	506
424	0	2	19099.2000000	19099.2000000	0.00%	581

```
NOTE: Optimal.
NOTE: Objective = 19099.2.
```

You can see that the solver finds a solution that, while minimizing total time, also reduces the total distance traveled by about 1,300 miles.

The following statement creates the output data set Schedule2, which corresponds to this new optimal solution:

```
create data Schedule2(drop=g1) from [g1 g]=SOLUTION
  location[g] away_team[g] home_team[g] city[location[g]] state[location[g]]
  start_datetime=dhms[g]/format=datetime14.
  latitude[location[g]] longitude[location[g]];
quit;
```

Figure 18 shows the new resulting optimal path in the network, Figure 19 shows the new resulting optimal schedule, and Figure 20 shows the new resulting optimal schedule geographically.

Figure 18 Network-Based Formulation: Optimal Path with Secondary Objective

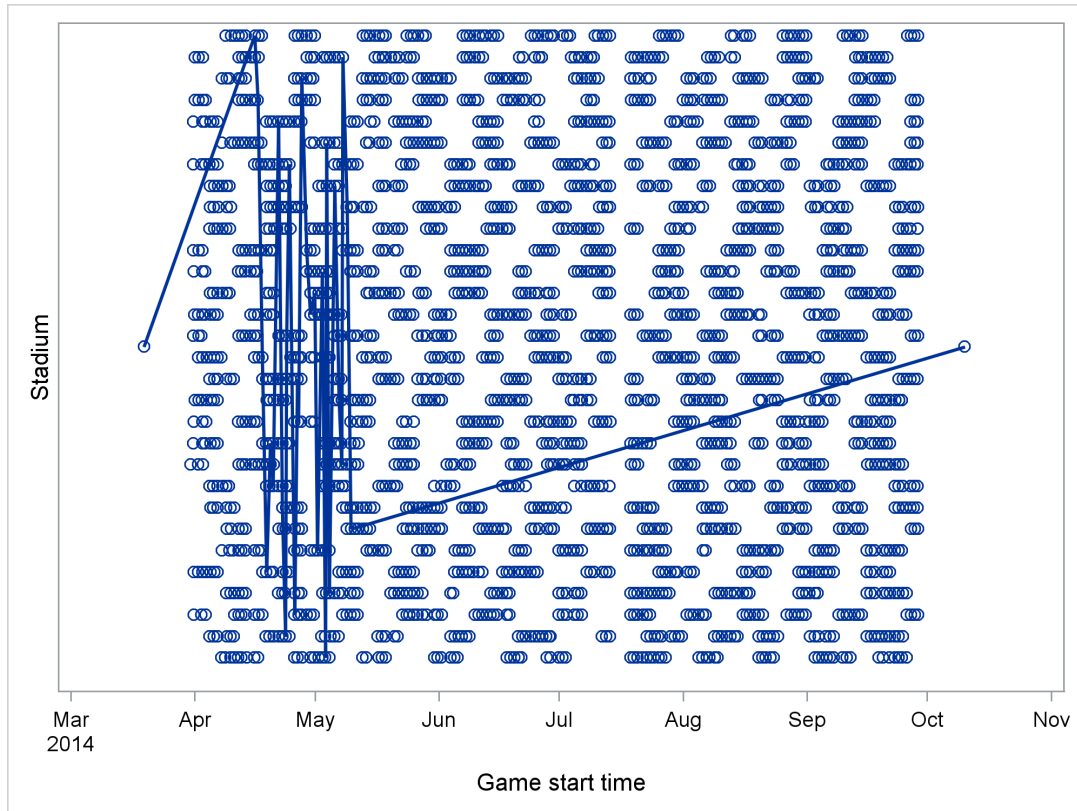
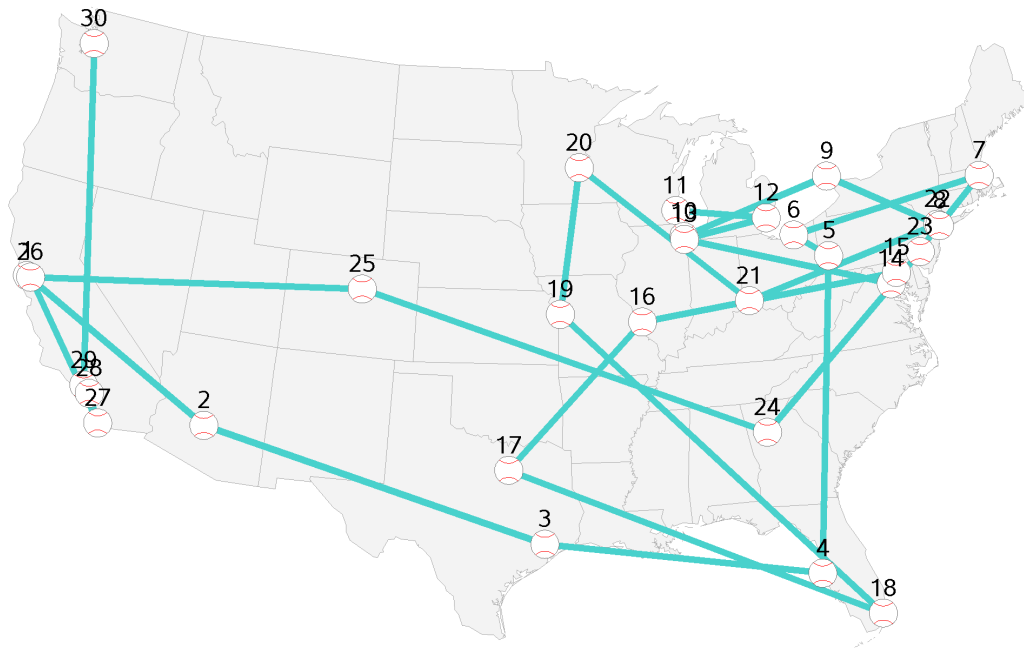


Figure 19 Network-Based Formulation: Optimal Schedule with Secondary Objective

Obs	location	away_team	home_team	city	state	start_datetime
1	AT&T Park	Dodgers	Giants	San Francisco	CA	15APR14:22:15
2	Chase Field	Mets	D-backs	Phoenix	AZ	16APR14:15:40
3	Minute Maid Park	Royals	Astros	Houston	TX	17APR14:20:10
4	Tropicana Field	Yankees	Rays	St. Petersburg	FL	18APR14:19:10
5	PNC Park	Brewers	Pirates	Pittsburgh	PA	19APR14:19:05
6	Progressive Field	Blue Jays	Indians	Cleveland	OH	20APR14:13:05
7	Fenway Park	Orioles	Red Sox	Boston	MA	21APR14:11:05
8	Citi Field	Cardinals	Mets	Flushing	NY	21APR14:19:10
9	Rogers Centre	Orioles	Blue Jays	Toronto	ON	22APR14:19:07
10	Wrigley Field	D-backs	Cubs	Chicago	IL	23APR14:14:20
11	Miller Park	Padres	Brewers	Milwaukee	WI	23APR14:20:10
12	Comerica Park	White Sox	Tigers	Detroit	MI	24APR14:13:08
13	U.S. Cellular Field	Rays	White Sox	Chicago	IL	25APR14:20:10
14	Nationals Park	Padres	Nationals	Washington	DC	26APR14:13:05
15	Oriole Park at Camden Yards	Royals	Orioles	Baltimore	MD	26APR14:19:05
16	Busch Stadium	Pirates	Cardinals	St. Louis	MO	27APR14:14:15
17	Globe Life Park in Arlington	Athletics	Rangers	Arlington	TX	28APR14:20:05
18	Marlins Park	Braves	Marlins	Miami	FL	29APR14:19:10
19	Kauffman Stadium	Blue Jays	Royals	Kansas City	KS	30APR14:20:10
20	Target Field	Dodgers	Twins	Minneapolis	MN	01MAY14:13:10
21	Great American Ball Park	Brewers	Reds	Cincinnati	OH	02MAY14:19:10
22	Yankee Stadium	Rays	Yankees	Bronx	NY	03MAY14:13:05
23	Citizens Bank Park	Nationals	Phillies	Philadelphia	PA	03MAY14:19:05
24	Turner Field	Giants	Braves	Atlanta	GA	04MAY14:13:35
25	Coors Field	Rangers	Rockies	Denver	CO	05MAY14:20:40
26	O.co Coliseum	Mariners	Athletics	Oakland	CA	06MAY14:22:05
27	Petco Park	Royals	Padres	San Diego	CA	07MAY14:15:40
28	Angel Stadium of Anaheim	Yankees	Angels	Anaheim	CA	07MAY14:22:05
29	Dodger Stadium	Giants	Dodgers	Los Angeles	CA	08MAY14:22:10
30	Safeco Field	Royals	Mariners	Seattle	WA	09MAY14:22:10

Figure 20 Network-Based Formulation: Map of Optimal Schedule with Secondary Objective



COFOR STATEMENT

Besides the MILP solver `PARALLEL=1` option, another way to use parallel processing to solve the problem faster is to use the COFOR statement, which is new in SAS/OR 13.1. The COFOR statement operates in the same manner as the FOR statement, except that with the COFOR statement PROC OPTMODEL can execute the SOLVE statement concurrently with other statements.

Suppose that you know an upper bound n on the number of days in an optimal schedule. Then you can divide and conquer the problem by considering each possible interval of n consecutive days as a separate subproblem. Each of these subproblems (179, in 2014) has the same structure as the original problem but considers a much smaller subset of games, and an optimal solution to the original problem is among the optimal solutions to the subproblems.

The following statements declare some numeric parameters and sets used for this approach, with $n = 25$ because you know that a 25-day schedule exists for the 2014 season:

```
num n = 25;
num num_solved init 0;
num bestObj init infinity;
set <num,num> INCUMBENT;
set START_DATES = setof {g in GAMES} start_date[g];
num cutoff {START_DATES} init infinity;
set GAMES_d {d in START_DATES} = {g in GAMES: start_date[g] in d..d+n-1};
set ARCS_d {d in START_DATES} =
    {<g1,g2> in ARCS: (g1 = source and g2 in GAMES_d[d])
                     or (g1 in GAMES_d[d] and g2 = sink)
                     or (g1 in GAMES_d[d] and g2 in GAMES_d[d])};
```

You can now use a FOR statement to loop through the start dates, fix the **UseArc** variables to 0 for arcs that include a game outside the current n -day subproblem, call the MILP solver, and unfix the variables that

were fixed. But because the subproblems are independent of one another, you can solve them in parallel by simply changing the FOR statement to a COFOR statement instead, as in the following:

```

cofor {d in START_DATES} do;
  put d=date9.;
  for {<g1,g2> in ARCS diff ARCS_d[d]} fix UseArc[g1,g2] = 0;
  cutoff[d] = bestObj;
  solve with MILP / cutoff=(cutoff[d]);
  for {<g1,g2> in ARCS diff ARCS_d[d]} unfix UseArc[g1,g2];
  num_solved = num_solved + 1;
  if substr(_solution_status_,1,7) = 'OPTIMAL' and bestObj > _OBJ_ then do;
    bestObj = _OBJ_;
    INCUMBENT = {<g1,g2> in ARCS: UseArc[g1,g2].sol > 0.5};
  end;
  put num_solved= bestObj=;
end;
for {<g1,g2> in ARCS} UseArc[g1,g2] = (<g1,g2> in INCUMBENT);

```

Note that the MILP solver CUTOFF= option is used to terminate each subproblem early if the solver determines that the current subproblem cannot yield a better solution than the current best. [Figure 21](#) shows the log for the COFOR loop iteration that yields an optimal solution.

Figure 21 Log for One Iteration of COFOR Loop

```

d=15APR2014
NOTE: Problem generation will use 2 threads.
NOTE: The problem has 72747 variables (0 free, 63617 fixed).
NOTE: The problem has 72747 binary and 0 integer variables.
NOTE: The problem has 2436 linear constraints (0 LE, 2436 EQ, 0 GE, 0 range).
NOTE: The problem has 215837 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 63617 variables and 2063 constraints.
NOTE: The MILP presolver removed 188788 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 9130 variables, 373 constraints, and 27049
      constraint coefficients.
NOTE: The MILP solver is called.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	1	24.1631944	24.1631944	0.00%	3
0	0	1	24.1631944	24.1631944	0.00%	3

```

NOTE: Optimal.
NOTE: Objective = 24.163194444.
num_solved=17 bestObj=24.163194444

```

It turns out that this entire COFOR loop finishes running in six minutes, yielding a provably optimal schedule. Furthermore, you can apply this same approach to the weaker initial formulation by instead fixing and unfixing the **Attend** variables.

EXTENSIONS

Because PROC OPTMODEL provides a flexible modeling language, you can easily modify the optimization model to increase accuracy or to account for additional rules.

Two natural refinements that require essentially no changes to the optimization model are as follows:

- game-dependent game durations
- time-dependent travel times

The first refinement was already addressed in the section “[DATA](#)” on page 2. The second refinement can be handled by replacing *times_between_stadiums[location[g₁], location[g₂]]* with *times_between_games[g₁, g₂]* values read from a data set.

You can account for the following extensions by adding constraints to the optimization model:³

- See each team exactly twice.
- See your favorite team at least *n* times.
- See a particular matchup at least *n* times.
- No repeat matchups.
- No consecutive games with the same team.
- Do not visit cold cities in April or September.
- Force a game to coincide with your existing travel plans to a city.
- Fix variables and reoptimize if your plan is disrupted (perhaps because of a game cancellation, extra innings, or traffic delays).

Although you can express many of these constraints more naturally by using the **Attend[g]** variables from the initial formulation rather than the **UseArc[g₁, g₂]** variables from the network formulation, you can rewrite any constraints that involve **Attend[g]** by using the correspondence

$$\text{Attend}[g] = \sum_{(g_1, g) \in \text{ARCS}} \text{UseArc}[g_1, g]$$

either as an explicit constraint,

```
con Link {g in GAMES}:
    Attend[g] = sum {<g1, (g)> in ARCS} UseArc[g1, g];
```

or as an implicit variable,

```
impvar Attend {g in GAMES} =
    sum {<g1, (g)> in ARCS} UseArc[g1, g];
```

For example, because the existing constraints already enforce seeing every team *t* once at its home stadium, you can write the first side constraint (“see each team exactly twice”) for each *t* as follows:

$$\sum_{\substack{g \in \text{GAMES}: \\ \text{away_team}[g] = t}} \text{Attend}[g] = 1$$

The following statement declares this set of constraints:

```
con See_away_once {t in TEAMS}:
    sum {g in GAMES: away_team[g] = t} Attend[g] = 1;
```

You can handle the remaining extensions similarly.

CONCLUSION

This paper demonstrates the power and flexibility of the OPTMODEL procedure in SAS/OR to formulate and solve a routing and scheduling optimization problem. The rich and expressive algebraic modeling language enables you to easily explore multiple mathematical programming formulations and access multiple optimization solvers, all within one PROC OPTMODEL session. When a natural formulation turns out to be weak, sometimes a reformulation that uses more decision variables turns out to be easier to solve. PROC OPTMODEL also offers ways to exploit parallel processing, including solver options and the COFOR statement, which is new in SAS/OR 13.1. When you have a tractable approach to solve a basic version of the problem, it is often easy to model several side constraints.

³Including any of these side constraints would require the full arc set rather than the smaller arc set used to generate the results in this paper.

REFERENCES

- Booth, D., Landgren, C. B., and Lee, K. A. (2011), *The Fastest Thirty Ballgames: A Ballpark Chasers World Record Story*, Bloomington, IN: AuthorHouse.
- Cleary, R., Faga, D., Lui, A., and Topel, J. (2000), "The Traveling Baseball Fan," *Math Horizons*, 8, 18–22.
URL <http://www.jstor.org/stable/25678277>
- Major League Baseball (2008), "TWIB: 30 Ballparks," online, video.
URL http://wapc.mlb.com/play/?content_id=3295802
- Zdeb, M. (2010), "Driving Distances and Times Using SAS and Google Maps," in *Proceedings of the SAS Global Forum 2010 Conference*, Cary, NC: SAS Institute Inc.
URL <http://support.sas.com/resources/papers/proceedings10/050-2010.pdf>

ACKNOWLEDGMENTS

Thanks to Phil Gibbs for introducing the authors to this problem and to Robert Allison for providing the SAS code to generate the maps.

CONTACT INFORMATION

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

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

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.