The National Basketball Association Scheduling Problem

Course Project, 21-393 Operations Research II

Renee Chen, Teddy Ding, Emily Hu, Alyssa Wang

Professor David Offner

December 16, 2016

Abstract

This paper discusses the National Basketball Association scheduling problem using the local search algorithm and simulated annealing in particular. Starting with the NBA 15-16 regular season schedule, our method tests its optimality and searches for a better schedule, if possible, under our evaluating system.

Introduction

The NBA, short for the National Basketball Association, is the pre-eminent men's professional basketball league in North America. Originated in 1946, the NBA currently has 30 teams, with two conferences of three divisions. It is also the fourth profitable sports league in the world. We are interested in creating an efficient NBA schedule which not only ensures the competitive fairness, but also raises the commercial value and lowers the travel costs. Since the current schedule is mostly done by hand, we are also interested in using a computer program to check if there exist better schedules.

Every year, the NBA league office establishes a regular season playing schedule of 82 games for each team, including 41 home games and 41 away games. Each team must play 4 games against other 4 division opponents, 4 games against 6 out-of-division conference opponents, 3 games against the remaining 4 conference teams, and 2 games against teams in the opposing conference. In our research, we take these general rules as our constraints. To ensure fairness, number of back-to-back games and 4-in-5s for each team are limited. To create greater

commercial value, popular teams play each other on popular dates. In our research, we only take travel distance, back-to-back games, and team popularity as objectives for simplicity.

We researched into several algorithms and decided to use local search. Local search is a heuristic method to find a solution maximizing a criterion among several candidate solutions. The general idea is to repeatedly make small changes to a given schedule until stop condition is met. To prevent cycling, we use simulated annealing, which evaluates the new schedule and reject with higher probability if it is a worse schedule.

Current Scheduling Method

Matt Winick, the vice president of scheduling and game operations for the NBA, was the NBA schedule maker for 30 years. Due to the complexity of the scheduling problem, most of the work was done by hand. The scheduling method is kept secret. However, we do know that before the end of the preceding regular season, all teams are asked to submit a list of at least 50 dates on which their home court will be available, including 4 Mondays and 4 Thursdays. Christmas eve, the all-star game, and the NCAA championship game are the only official breaks. Games can be moved to satisfy the NBA's broadcasting partners.

According to Winick, the NBA schedule is constructed to be "efficient from a competitive standpoint with an indirect consideration of travel costs"[1]. The balance between competitive fairness and travel costs contributes to the numerous constraints and the complexity of scheduling. In the 16-17 season, no team plays more than 18 back-to-back games, with a league average of 17.8. There are 10 teams that don't have to play 4 games in 5 nights; no team plays more than one 4-in-5's, with a league average of 0.7. Generally, each team plays 3.5 games in a week, and 82 games take roughly 165 days through the end of the regular season[2].

[1] Winick, Matt. Telephone interview, December 12, 2007.
[2] USA Today. NBA Schedule More Player Friendly with Fewer Back-to-Back Games. http://www.usatoday.com/story/sports/

Preliminary Research and Possible Algorithms

Integer programming is often used in solving sports scheduling problems and can come up with optimal solutions for smaller leagues. We can use a variable x_{ijt} that equals 1 if team x (the home team) is playing team y (the away team) on day t and 0 otherwise. The constraints used for our problem are:

- $\sum_{t=1} x_{ijt} = 1$ for certain i and j to ensure the conditions for each specific game in the 82 games are met.
- $\sum_{t=1} x_{ijt} = 41$ for each i = 1 to n and for each j = 1 to n to ensure that a proper number of home and away games are played.
- $\sum_{t=1} x_{ijt} \leq 1$ for each i = 1 to n, t = 1 to n to ensure that teams are not playing more than once for each time.

We would also like to optimize certain values such as $\sum_{t=1} x_{ijt} * x_{i+1jt} = 1$ which counts the number of back to back games, and other functions for counting 4 out of 5 games or travel distance. We would like to minimize the value for number of back to back games.

However, the basketball scheduling program simply involves too many variables and constraints to be solved within a reasonable time with integer programming. While it is infeasible to use pure integer programming for large problems, there are aspects which can be used in conjunction with other methods to obtain a solution. The optimization constraints can be adapted into a scoring function such as the one we use.

Another method to approach scheduling is decomposition, which is usually done with one of two methods. In the first, the pairings that need to be played are determined for a section of schedule, and then a home-away pattern is calculated for the section. In the second, a home-away pattern is first calculated, with the teams playing chosen afterward. This is a good method to develop a schedule that can easily be improved upon. Another way is to use a greedy algorithm that assigns teams to games in a schedule.

A common approach to problems with a huge amount of constraints and considerations such as in sports scheduling is to use different methods of heuristics searches and metaheuristics. Because these problems cannot be solved completely optimally in a reasonable amount of time, we often resort to algorithms that can find solutions that are likely sub-optimal but obtainable in a reasonable runtime. There are many approaches to developing such algorithms, such as tabu search, greedy randomized searches, genetic algorithms, and many more. These methods have many similarities mostly have a focus on different ways to search for an optimal solution using a given one with methods to not get stuck in local minima. The method we primarily look at is simulated annealing.

Proposed Scheduling Method

Formulation of the Problem

When creating the NBA schedule, we take the following objectives into consideration: the number of back-to-back games, the total distance each team travels, the occurrence of popular games on important dates, and the fairness of the schedule regarding each team. We assign weights to each objective and use them to calculate a score for the schedule. The weight assignment process translates the NBA scheduling problem into a mathematical problem of finding the schedule with the highest possible score. Our scheduling method follows all constraints of a valid NBA schedule, and works under certain assumptions.

Starting with a past schedule, we use the local search algorithm and simulated annealing, in particular, to test the optimality of the schedule and search for a better schedule, if possible, under our scoring system.

The Evaluator

The evaluator is a function that takes in a schedule and returns a score. In other words, it evaluates a given schedule. The evaluation is based on three factors – travel distance, number of back-to-back games, and team popularity.

The biggest complaint about the NBA schedule has been too many back-to-back games. Therefore, our most important assumption is that penalty for number of back-to-back games increases quadratically. Under this assumption, the number of back-to-back games has a significant impact on the score of a schedule. We count the number of back-to-back games for each team and for the whole season by iterating through the schedule. We then assign the quadratic penalty and add it into the total score.

As for travel distance, we simply assume that the team does not return to its home city between two consecutive games. That is, if a team has two consecutive away games at city A and B, then the team travels from A to B directly. We assign the distance factor a softer weight with linear penalty. Similarly, we iterate through the current schedule to get the total distance for each team and for the whole season. Then we assign the linear penalty and add it into the total score.

We simplify the calculation of fairness by considering only number of back-to-back games and the travel distance. Our goal is to have all teams play roughly the same number of back-to-back games and total distance. Since standard deviation is an indication of variation among a data set, we use the standard deviation of the number of back-to-back games and the total distance for each team as evaluation of fairness. We assign penalty for both standard deviations, and add it into the total score.

The last factor, team popularity, is a key factor of commercial value creation. Ticket price, TV rating and viewership, sponsorships, and media coverage are all affected by team popularity. Another assumption is that there would be significantly more audience if two popular teams are playing on a popular date. We assume that Fridays and Christmas day are the popular dates, and the team popularity could be rated based on a team's performance, fan base, and commercial value.

We rate the 30 teams by popularity considering overall performance rating, number of Twitter and Facebook followers, and current value. We weight each factor according to its significance. For example, overall performance rating is significant for the schedule since it is more up-to-date, and the majority of audience is not from the fan base. Therefore, we scale up overall performance rating by $rating + |lowest\ rating| * 1.5$. Meanwhile, since the number of followers on Twitter is generally less than that on Facebook, we scale up Twitter followers by 4 times. The final popularity rating is calculated by summing up all weighted factors. The table below shows the top and bottom 3 teams by popularity. (See full table in Appendix)

City	Team	Overall[3]	Overall*	Twitter[4]	Twitter*	Facebook[5]	Value[6]	Popularity
Golden States	Warriors	10.49	31.00	2.44	9.76	9.15	1.9	**51.82**
Miami	Heat	1.52	17.55	3.56	14.24	16.21	1.3	**49.30**
Los Angeles	Lakers	-9.43	1.13	5.22	20.88	21.85	2.7	**46.56**
Brooklyn	Nets	-7.61	3.86	0.68	2.72	2.79	1.7	**11.07**
Phoenix	Suns	-6.50	5.52	0.58	2.32	1.89	1	**10.73**
Philadelphia	76ers	-10.18	0	0.71	2.84	1.52	0.7	**5.06**

Table 1: Popularity Rating

Most teams' popularity ratings lie between 20 and 30. The 5 most popular teams are rated more than 40. We define the popularity score for a game as the sum of the team popularity ratings. Then it is fair to assume that a popular game has a score greater than 60. Under this assumption, we derive the method as follows: for example, a game is played on a Friday with a total score of K larger than 60. Then, we add $K^2 - 60^2$ to the raw score for popularity (unweighted). By squaring K, an increase in K would have an enlarged effect on the final popularity score.

The Local Search Algorithm: Simulated Annealing

After examining an array of possible approaches to the problem, we decided to apply the local search algorithm, in particularly simulated annealing for this problem. The simulated annealing algorithm will be described as applied to a combinational optimization problem. Many combinatorial optimization problem have been shown to be NP-hard, which means the running time for any algorithms currently known to guarantee an optimal solution is an exponential function of the size of the problem[7]. Simulated annealing works particularly well in our case, for a couple of reasons specific to the NBA scheduling problem:

---

[3] Basketball Reference. *2015-2016 NBA Team Ratings.* http://www.basketball-reference.com/leagues/NBA/
[4] Statista. *Twitter Followers of NBA Teams (in 1,000s), September 2016.* https://www.statista.com/statistics/240386/
[5] Statista. *Facebook Fans of NBA Teams in September 2016 (in millions).*https://www.statista.com/statistics/240382/
[6] Forbes. *The Business of Basketball (2016 Ranking).* http://www.forbes.com/nba-valuations/list/
[7] Eglese, R.W. Simulated Annealing: A Tool for Operational Research. http://www.sciencedirect.com/science/

1) The total number of games in a season is $\frac{82\ games/team * 30\ teams}{2\ teams/game} = 1230$, and the 1230 games span 161 different game dates. The search space is impossibly large for finding a global optimal solution. Simulated annealing is designed for this type of problem as it searches for global maximum while allowing travelling to worse situations, thus expanding its search range to find a "better" local maxima.
2) The simulated annealing algorithm focuses on finding neighboring viable solutions in the search space. This works well with the NBA scheduling problem. We note that for any schedule that is a feasible solution, by identifying a rule to switch games or a bundle of games, we can easily find a new feasible solution that is a neighbor of our previous solution. We experimented with two different ways of finding a new schedule/neighbor, which will be elaborated later.
3) Our goal is to "find better schedules". An NBA schedule has a wide array of metrics, as described above, and there is no obvious way to compare any two schedules. Thus, the best way to evaluate is to assign numerical scores to each schedule. The evaluation or scoring process is the foundation of a local search algorithm, since it evaluates the generated schedule on any iteration and make local decisions based on its improvement from the previous solution.
4) A local search algorithm is convenient if you have a viable solution to start with. This is the case in our research, since we (or any official in the real world) have access to the schedule from previous years. In fact, these are already good viable solutions to the problem. This provides an easy and favorable starting point for our algorithm to search for neighboring maxima.

A Breakdown of the Algorithm

A general idea of the simulated annealing approach has been described above. A pseudocode algorithm is provided below:

1. $i := 0$
2. Generate initial solution $s \in \mathcal{S}$
3. $best := f(s)$
4. REPEAT
    1. Generate randomly a solution $s' \in \mathcal{N}(s)$
    2. IF Rand(0,1) $< \min\{1, e^{\frac{f(s')-f(s)}{t_i}}\}$ THEN
        1. $s := s'$
        2. IF $f(s') < best$ THEN
        3. $s^* := s$
        4. $best := f(s')$
5. $i := i + 1$
6. UNTIL some stopping condition is satisfied

- Step 2 of the algorithm is simple: we take a working NBA schedule and feed it to our schedule searcher. In particular, we chose NBA's 2015-2016 Season Regular season schedule as a starting point for our search.

- For the first step in the REPEAT loop, our algorithm generates a random neighboring solution by following a switch rule as mentioned before. Specifically, we tried the following two different methodology:

    1. Switch games: We randomly find two games on different dates, and switch them if doing so don't break the constraints. In particular, all constraints related to number of games are maintained, and we only have to check if switching the selected games leads to one team playing two games on the same day. This switch method maintains the number of games played on each date, which is important since we don't want a viable solution that aggregates too many games on some dates (e.g., naively put all the games in week on Friday, which is a desirable dates)

    2. Switch dates: We randomly find two different dates, and simply switch them. This also maintains all the important invariants as in the first method, but its flexibility since it always moves a bunch games together making it hard for the algorithm to try to maximize the benefit of single games, or, extracting games from on date, and distributing them to dates that improve the schedule.

Another key concept of the algorithm lies in the second step in the REPEAT loop, where we simulate the "cooling" process of this algorithm. To put simply, the condition to move to a new schedule cools down as the algorithm goes further. This is achieved by the denominator t_i in the formula, which is called a "control value sequence". This sequence converges to zero, to polarize the difference of $f(s') - f(s)$, thus gradually decreasing (cooling) the probability of moving a worse-off solution in the long run[8].

Score and Statistics of Initial Schedule

[Total iterations = 0, Updated iterations = 0]
s_score = -86.8582,
btbNum = 532,
btbStdev = 1.8607,
distanceSum = 815636.0000,
distanceStdev = 7816.9061,
tvRatingScore = 429735.7629
christmasDayGames: set([('New Orleans', 'Miami'), ('Cleveland', 'Golden State'), ('Los Angeles Clippers', 'Los Angeles Lakers'), ('Chicago', 'Oklahoma City'), ('San Antonio', 'Houston')])

The measured correspond to the metrics given in the previous section on evaluator's methodology:

"btbNum": Number of back-to-back games in total
"btbStdev": Standard deviation of the number of back-to-back games of each team
"distanceSum": Total travel distance of the teams
"distanceStdev": Standard deviation of the travel distance among teams
"tvRatingScore": Score for popular games on popular dates
"christmasDayGames": The games scheduled on Christmas Day, namely 12/25/15

[8] In our implementation, we maintain a minimum threshold value for t_i THRESHOLD = 0.0001, given the fact that too small denominator makes too large exponent for *e* and causes arithmetic overflow in python.

Here it's worth noting that our schedule *f(s)* normalizes each factor into a scale of 0 to 100, and then sums them up with respective weights that sum to 1. This helps to understand our schedule score on a 0-to-100 scale. The initial schedule, as given above, is assigned a score of -86.8582.

Applications and Results

Through playing with the factors of each metrics in our evaluator, we tested out our algorithm given a few different configurations: different ways to find neighboring schedules as well as different allowance for worse-off positions. First, while fixing the control value sequence t_i, we ran the algorithm with two different ways to generate neighboring schedule. The results are illustrated below.

Method A: Switch Two Games

[Total Iterations = 20000, updated iterations = 517]
s_score = -45.8878
btbNum = 460
btbStdev = 1.01105
distanceSum = 546274.0000
distanceStdev = 4412.3043
tvRatingScore = 601925.9572
Christmas Day Games = set([('Chicago', 'Oklahoma City'), ('Cleveland', 'Golden State'), ('Boston', 'Miami'), ('Dallas', 'Brooklyn'), ('San Antonio', 'Los Angeles Lakers')])

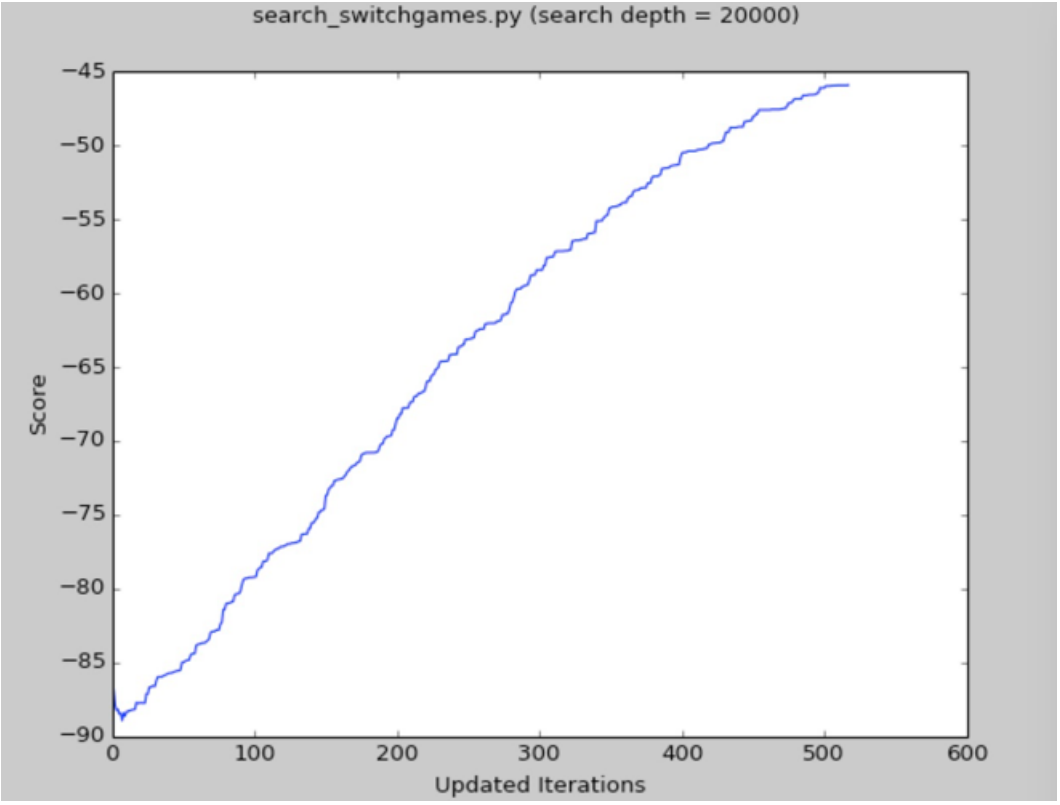


Figure 1: Graph for Method A Score vs. Updates Iterations

Method B: Switch Two Dates

[Total iterations = 19999, Updated iterations = 66]

s_score = -91.2688

btbNum = 549

btbStdev = 1.7156

distanceSum = 730097.0000

distanceStdev = 6960.0307

tvRatingScore = 417638.0188

christmasDayGames: set([('San Antonio', 'Chicago'), ('Oklahoma City', 'Atlanta'), ('Boston', 'Miami'), ('Houston', 'Detroit'), ('Portland', 'Los Angeles Clippers'), ('Golden State', 'Utah'), ('Denver', 'Milwaukee'), ('Dallas', 'Sacramento')])

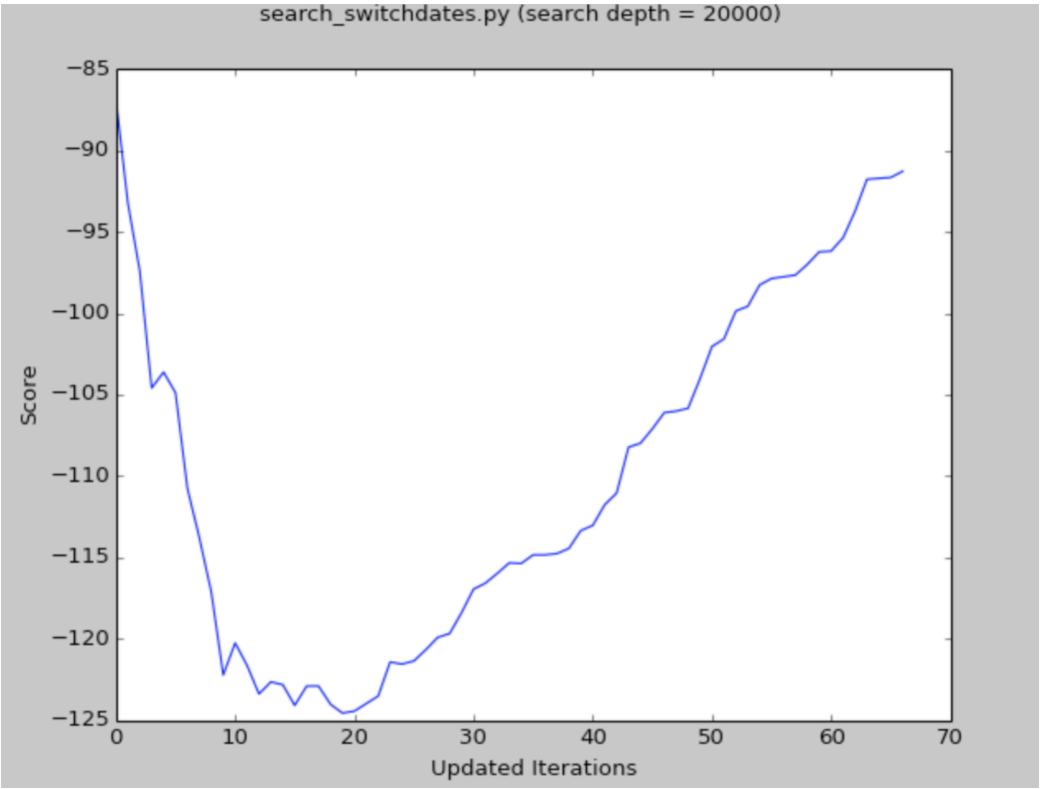


Figure 2: Method B Score vs. Updated Iterations

Method C: Switch Games, Strictly Increasing Score Sequence (No simulated Annealing)

[Total iterations = 19999, Updated iterations = 555]

s_score = -47.9032

btbNum = 462, btbStdev = 1.0832

distanceSum = 547030.0000

distanceStdev = 3826.4762

tvRatingScore = 511570.0289

christmasDayGames:set([('San Antonio', 'Houston'), ('Cleveland', 'Golden State'), ('Atlanta', 'Miami'), ('Oklahoma City', 'Chicago'), ('Los Angeles Clippers', 'Los Angeles Lakers')])

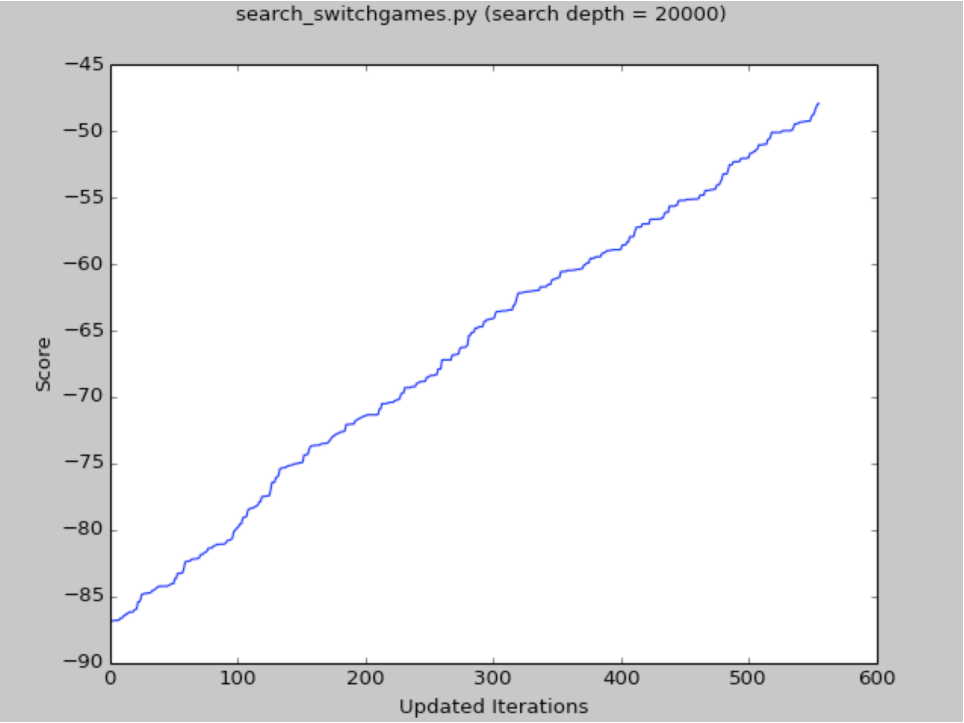


Figure 3: Method C Score vs. Updated Iterations

Discussion and Reflections

Performance of the Methods

We first discussed the first two methods where we adopted the simulated annealing process. As we can see from the graph and numerical results, on both graph A and B, the search process displays a drop to lower scores initially (in method A, this initial drop is hard to see given the scale) followed by a flatter process of gradual increase. This is expected from the algorithm, given the initially large control value t_i which allows high probability of moving to worse-of schedules. As t_i converges to zero, we see that the program becomes less and less inclined to accept a worse-of position and roughly follows the pattern of a non-decreasing sequence.

To compare the results from two methods, we see that method A that switches two games produces a much more optimal result. It's also obvious that number of actual "updated" iterations from method B is far smaller than that of method A. Note that while the search depth of both methods is 20000, the updated iterations where the search "accepts" a new schedule differs far from the search depth. So, a graph that plots score against iterations will look far flatter. We believe these distinctions is related to the flexible nature of these two methods: switching by games generates far more possible valid solutions and thus a far larger search space, while switching games by dates limits the flexibility of optimizing the utility of each game within that date. By contrast, switch individual games does well in putting to positions that generates utility (separating games to reduce back-to-back games, put particularly favorable games on better dates, etc.). The idea to switch games by dates sources from the notion of "moving bundle of games", which attempts recognize a couple of games that are already "bundled" together, for instance, some team making an away-game trip that involves playing Houston, San Antonio and Dallas in a row. Switching dates fells short at recognizing this local optimality.

We also ran a third method of switching game, except this time forcing searcher to reject all worse-off situations. In theory, this strictly gives us the local maximum of closest to the initialization. As the test result turns out, method A gives us a final score of *-47.9032* while method C gives a schedule with score *-45.8878.* This does not display obvious difference, and possible explanation is the limited depth we applied. Given the fact that our evaluator is non-trivial, we decided that 20000 iterations would be adequate depth. In theory, we should be able to run more iterations, or set terminate conditions requiring convergence (e.g., the average speed of increase falls below a threshold), and see a larger difference between the simulated annealing searcher and the non-increasing searcher.

Weight of Metrics and Reasoning

One of the major problem was to decide on an appropriate set of factor weights for the evaluator. This was a long process involving a lot of guessing and testing, the final set of weights we arrived on is as follow:

$W_{btbNum} = \%60$ $W_{distanceStdev} = \%2.5$

$W_{btbStdev} = \%2.5$ $W_{distanceSum} = \%5$

$W_{distanceSum} = \%30$

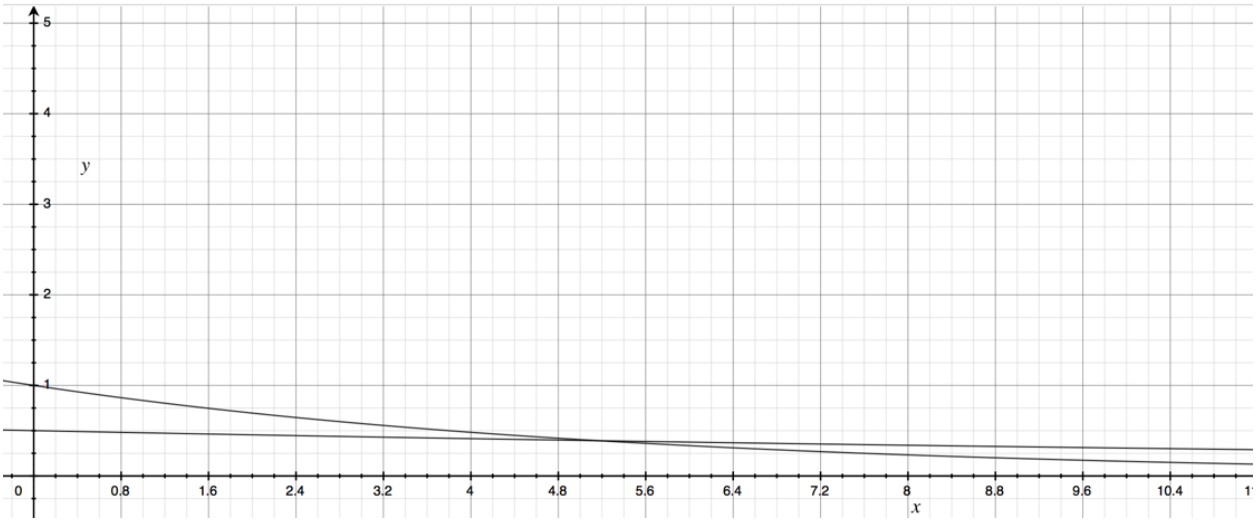
To account for the sharp difference between weights for standard deviation and the value of variables, we note that the actual values of standard deviation (*btbStdev* ≈ *1.9*) is far smaller than the actual variable values (*btbNum* ≈ *20.0*). One issue to overcome throughout our testing was to force the searcher to suppress the number of back-to-back games, as it's relatively easy to generate a new schedule that increases the number of back-to-back. Therefore, giving *btbNum* the most weight, combined with a cubic function that further penalizes high *btbNum*, ended up working for us. Another interesting point to add is that penalizing standard deviation helps limits the number of total back-to-back games, since initially all teams have around *teamBtbNum = 20*, and as we evaluate the schedule again after any switch of two games, the evaluator suppresses any single team from becoming an outlier by increasing *teamBtbNum*, which helps keeping *btbNum* down.

Choices for the Control Sequence t_i

While the only requirements on control value sequence t_i is that it converges to zero, its initial value as well as the speed of converge changes things a lot. Here highlighted are two better choices that we came up with.

Choice 1: $y = \max(0.0001, 1.2^{-x})$ starts at 1 but steeper, so algorithm initially accepts worse schedules easily but quickly starts to accept better schedules only.

Choice 2: $y = \max(0.0001, {}^{1.05^{-x}}/_{2})$ starts at 2 but flatter, so algorithm initially accept worse schedules with moderate possibility and gradually decline the possibility. This approach is more moderate and makes the algorithm more flexible and gives wider search range. In the end, we decided to adopt this sequence, since this roughly mimics the cooling process we have in our mind.



Possible Directions of Further Research

Throughout our research process, one challenge we face was finding a "clever" way to generate new schedules that boosts the speed of our algorithm. In the end, we stick the randomized game-switching algorithm, which gives sufficient result. But this could definitely be improved by finding other switch algorithms that "consciously" looks to decrease back-to-back games while minimizing travel distance. One approach for this, as some other sports scheduling algorithms have attempted, is to identify "bundles" of games that work well together on subsequent days, and switch these together to a better position in the schedule. This identification process is tricky, but that's one direction of improvement

One other we could take to better understand the algorithm in this particular case, is to analyze the development of search process, possibly visualizing the trend of different factors over the course of improving total score. This could give us insight into what the searcher prioritizes initially, and what it favors in the long run.

Acknowledgements

We thank Professor Offner who provided insights and expertise that greatly assisted the research.

Bibliography

Basketball Reference. *2015-2016 NBA Team Ratings.* http://www.basketball-reference.com/leagues/NBA_2016_ratings.html

Eglese, R.W. *Simulated Annealing: A Tool for Operational Research*. http://www.sciencedirect.com/science/article/pii/037722179090001R

Forbes. *The Business of Basketball.* http://www.forbes.com/nba-valuations/list/#tab:overall

Nieberg, Tim. *"Metaheuristics in Scheduling, Local Search and Genetic Algorithms"*.

Statista. *Twitter Followers of NBA Teams (in 1,000s), September 2016.* https://www.statista.com/statistics/240386/twitter-followers-of-national-basketball-association-teams/

Statista. *Facebook Fans of NBA Teams in September 2016 (in millions).* https://www.statista.com/statistics/240382/facebook-fans-of-national-basketball-association-teams/

USA Today. *NBA Schedule More Player Friendly with Fewer Back-to-Back Games.* http://www.usatoday.com/story/sports/news/1075796-nba-limits-back-to-backs-4-in-5s-on-2016-17-schedule

Winick, Matt. Telephone interview, December 12, 2007.

Appendix I: Team Popularity Table

City	Team	Overall	Twitter	Facebook	Value	Overall*	Twitter*	Popularity
Golden State	Warriors	10.49	2.44	9.15	1.9	31.00	9.76	**51.81**
Miami	Heat	1.52	3.56	16.21	1.3	17.55	14.24	**49.30**
Los Angeles	Lakers	-9.43	5.22	21.85	2.7	1.12	20.88	**46.55**
San Antonio	Spurs	11.1	1.58	6.95	1.15	31.92	6.32	**46.34**
Chicago	Bulls	-1.42	2.87	18.76	2.3	13.14	11.48	**45.68**
Oklahoma City	Thunder	7.35	1.42	6.69	0.95	26.29	5.68	**39.61**
Cleveland	Cavaliers	5.9	1.5	7.15	1.1	24.12	6	**38.37**
Boston	Celtics	2.81	1.96	8.78	2.1	19.48	7.84	**38.20**
Los Angeles	Clippers	4.31	0.99	3.84	2	21.73	3.96	**31.53**
Toronto	Raptors	4.5	1.23	2.16	0.98	22.02	4.92	**30.08**
New York	Knicks	-3.02	1.52	6.1	3	10.74	6.08	**25.92**
Houston	Rockets	0.4	1.16	3.81	1.5	15.87	4.64	**25.82**
Atlanta	Hawks	3.62	0.66	1.59	0.825	20.70	2.64	**25.75**
Dallas	Mavericks	0.11	0.99	4.52	1.4	15.43	3.96	**25.31**
Indiana	Pacers	1.68	0.79	3.31	0.84	17.79	3.16	**25.10**
Charlotte	Hornets	2.34	0.58	1.73	0.75	18.78	2.32	**23.58**
Portland	Trail Blazers	1.03	0.63	2.39	0.975	16.81	2.52	**22.70**
Utah	Jazz	2.09	0.51	1.16	0.875	18.40	2.04	**22.48**
Orlando	Magic	-1.8	1.39	2.76	0.9	12.57	5.56	**21.79**
Detroit	Pistons	0.4	0.58	1.85	0.85	15.87	2.32	**20.89**
Washington	Wizards	-0.39	0.53	1.52	0.96	14.68	2.12	**19.28**
Memphis	Grizzlies	-2.22	0.61	1.81	0.78	11.94	2.44	**16.97**
Sacramento	Kings	-2.33	0.51	1.75	0.925	11.77	2.04	**16.49**
Denver	Nuggets	-2.94	0.52	1.96	0.855	10.86	2.08	**15.75**
Minnesota	Timberwolves	-3.68	0.51	1.83	0.72	9.75	2.04	**14.34**
New Orleans	Pelicans	-3.63	0.52	1.63	0.65	9.82	2.08	**14.18**
Milwaukee	Bucks	-4.37	0.52	1.41	0.675	8.71	2.08	**12.88**
Brooklyn	Nets	-7.61	0.68	2.79	1.7	3.85	2.72	**11.06**
Phoenix	Suns	-6.5	0.58	1.89	1	5.52	2.32	**10.73**
Philadelphia	76ers	-10.18	0.71	1.52	0.7	0	2.84	**5.06**

Appendix II: Codes

(Github Repo link: https://github.com/kn1ghtted/393Project.git)

Searcher Template Class (searcher.py):

```
GRAPH_UPATED_ITERATIONS_ONLY = False
ONLY_ACCEPT_BETTER = False

class searcher:
 def __init__(self):
   return
 # given a schedule file,
 # return a dictionary type of the schedule:
 # 'mm/dd/yy' -> set([(away1, home1), (away2, home2), ...])
 def readSchedule(self, filename):
   reader = CsvReader(filename)
   cal = reader.data
   attributes = reader.attributes
   # change datetime column to only datetime
   schedule = MyOrderedDict()
   for gameEntry in reader.data:
     date = gameEntry[DATE].split(" ")[DATE]
     epoch = timeUtil.dateToEpoch(date)
     standardDate = timeUtil.epochToDate(epoch)
     gameEntry[DATE] = standardDate
     awayTeam = gameEntry[AWAY]
     homeTeam = gameEntry[HOME]
     game = (awayTeam, homeTeam)
     # if this date already stored
     if standardDate in schedule:
       schedule[standardDate].add(game)
```

```
    else:
      schedule[standardDate] = set([game])
  self.schedule = schedule

# return True if team not relevant in games
def teamNoConflict(self, team, games):
  for game in games:
    if team in game:
      return False
  return True

def switchGames(self, schedule, date1, date2, game1, game2):
  assert(game1 in schedule[date1])
  assert(game2 in schedule[date2])
  schedule[date1].remove(game1)
  schedule[date2].remove(game2)
  schedule[date1].add(game2)
  schedule[date2].add(game1)
def generateNewSchedule(self):
  pass

def switchBack(self):
  pass
# uses simulated Annealing from page 16 of pdf
def searchSchedule(self):
  schedule = self.schedule
  scaleFactor = None
  self.best = evaluate(schedule)["score"]
  depth = 0
  update = 0
```

```
   if (PLOT):
     if (GRAPH_UPATED_ITERATIONS_ONLY):
       scorePlot = Plot(sys.argv[0], "Updated Iterations")
     else:
       scorePlot = Plot(sys.argv[0], "Iterations")

   # choose a solution s' from S randomly
   # by selecting a game randomly and swithing it with
   # another game, making sure that all four teams involved
   # don't have games on the same day
   while (depth < SEARCH_DEPTH):
     retObject = evaluate(schedule)
     s_score = retObject["score"]
     btbNum, btbStdev, distanceSum, distanceStdev, popularityScore =
retObject["btbNum"], \
       retObject["btbStdev"], retObject["distanceSum"],
retObject["distanceStdev"], retObject["tvRatingScore"]
     totalBtbs = 1
     print ("[Total iterations = %d, Updated iterations = %d]\n
s_score = %.04f, btbNum = %d, btbStdev = %.04f, distanceSum = %.04f,
distanceStdev = %.04f, tvRatingScore = %0.04f" % (depth, update,
s_score, btbNum, btbStdev, distanceSum, distanceStdev,
popularityScore))
     print ("christmasDayGames:" + str(schedule["12/25/15"]))
     print
     if (PLOT):
       if (GRAPH_UPATED_ITERATIONS_ONLY):
         scorePlot.update(update, s_score)

       else:
         scorePlot.update(depth, s_score)
```

```
      self.generateNewSchedule()
      # use randomness to decide with move to s'
      randNum = random.uniform(0.0, 1.0)
      s1_object = evaluate(schedule)
      s1_score = s1_object["score"]
      if (scaleFactor == None):
        scaleFactor = abs(s1_score - s_score)
      delta = s1_score - s_score
      # condition = min(1,
math.exp((delta*1.0/scaleFactor)*1.0/controlValues[update]))
      exponent = min(0,
(delta*1.0/scaleFactor)*1.0/controlValues[update])
      condition = math.exp(exponent)
      # print "delta = %.04f, randNum = %.04f, condition = %.04f" %
(delta, randNum, condition)
      if (randNum >= condition):
        # switch back
        self.switchBack()
      else:
        if (s1_score >= self.best):
          self.best = s1_score
        # this means we only update schedule when it's going in a
better direction
        else:
          if (ONLY_ACCEPT_BETTER):
            self.switchBack()
        update += 1
      depth += 1
```

Switch Game Method Implementation (search_switchgames.py):

```
class searcherSwitchGames(searcher):
 def switchBack(self):
     self.switchGames(self.schedule, self.date2, self.date1,
self.game1, self.game2)

 def generateNewSchedule(self):
   schedule = self.schedule
   self.date1 = random.choice(schedule.keys())
   self.game1 = random.choice(list(schedule[self.date1]))
   self.date2 = self.date1
   # choose the target game to switch
   # not on same day, all four games don't have
   # same day matches after switch
   date2Valid = False
   while ((not date2Valid)):
   # ???? should we limit the range of difference between the
   # dates to switch with?
     self.date2 = random.choice(schedule.keys())
     self.game2 = random.choice(list(schedule[self.date2]))
     if (self.date1 == self.date2):
       continue
     else:
       date1Games = copy.deepcopy(schedule[self.date1])
       date2Games = copy.deepcopy(schedule[self.date2])
       date1Games.remove(self.game1)
       date2Games.remove(self.game2)
       (teamA, teamB) = self.game1
       (teamC, teamD) = self.game2
       if ((self.teamNoConflict(teamA, date2Games)) and \
```

```
        (self.teamNoConflict(teamB, date2Games)) and \
        (self.teamNoConflict(teamC, date1Games)) and \
        (self.teamNoConflict(teamD, date1Games))):
          date2Valid = True
    # switch games, move to s'
    self.switchGames(schedule, self.date1, self.date2, self.game1,
self.game2)


S = searcherSwitchGames()
S.readSchedule("nba_games_2015-2016.txt")
S.searchSchedule()
```

Switch Dates Method Implementation (search_switchdates.py):

```
class searcherSwitchDates(searcher):
 def switchBack(self):
   schedule = self.schedule
   temp = schedule[self.date1]
   schedule[self.date1] = schedule[self.date2]
   schedule[self.date2] = temp

 def generateNewSchedule(self):
   schedule = self.schedule
   self.date1 = random.choice(schedule.keys())
   self.date2 = self.date1
   while (self.date1 == self.date2):
   # ???? should we limit the range of difference between the
   # dates to switch with?
     self.date2 = random.choice(schedule.keys())
   # print self.date1, self.date2
   # print schedule[self.date1], schedule[self.date2]
   temp = schedule[self.date1]
   schedule[self.date1] = schedule[self.date2]
   schedule[self.date2] = temp
   # print schedule[self.date1], schedule[self.date2]


S = searcherSwitchDates()
S.readSchedule("nba_games_2015-2016.txt")
S.searchSchedule()
```

Evaluator (evaluator.py)

```
GAME_SCORE_THRESHOLD = 4000

total = 0
teams = set()
teamScores = dict()
teamDistance = dict()
# to be determined
btbOnTotal = -10
btbOnTeam = -10
weightBtb = 0.3
weightFairness = 0.4
weightDistance = 0.3
distanceReader = DistanceReader("distances.csv")
distance = distanceReader.distanceDict
popularityReader = PopularityReader("Popularity new.csv")
popularityDict = popularityReader.popularityDict


def allTeams(calDict):
 for date in calDict:
   s = calDict[date]
   for game in s:
     for team in game:
       teams.add(team)

def initialTeamSocores(calDict):
 for eachTeam in teams:
   teamScores[eachTeam] = 0
   teamDistance[eachTeam] = (0,None)
```

```
# return True if team not relevant in games
def inGame(team, games):
 for game in games:
   if team in game:
     return True
 return None

def backToback(calDict,team):
 totalPanelty = 0
 totalDistance = 0
 counter = 0
 for date in calDict:
   nextDate = nextDay(date)
   games = calDict[date]
   if inGame(team,games):
     if (nextDate in calDict) and inGame(team,calDict[nextDate]):
       counter += 1
       totalPanelty += btbOnTeam
 return (totalPanelty,counter)


def getStdDev(teamScores):
 total = 0
 for team in teamScores:
   total += teamScores[team]
 mean = total * 1.0 /len(teams)
 variance = 0
 for team in teamScores:
   variance += (teamScores[team] - mean)**2
 variance = variance * 1.0 / len(teams)
```

```
 return math.sqrt(variance)

def popularity(calDict):
 popularityPoint = 0
 for date in calDict:
   month, day, year = (int(x) for x in date.split('/'))
   year = 2000 + year
   ans = datetime.date(year, month, day)
   weekday = ans.strftime("%A")
   games = calDict[date]
   if weekday == "Friday":
       for game in games:
           totalScore = popularityDict[game[0]] +
popularityDict[game[1]]
           if (totalScore ** 2) > GAME_SCORE_THRESHOLD:
               popularityPoint += (totalScore) ** 2 -
GAME_SCORE_THRESHOLD
   # Christmas day
   if date == "12/25/15":
       for game in games:
           totalScore = popularityDict[game[0]] +
popularityDict[game[1]]
           if (totalScore ** 2) > GAME_SCORE_THRESHOLD:
               popularityPoint += CHRISTMAS_MULTIPLIER * (totalScore
** 2 - GAME_SCORE_THRESHOLD)
 return popularityPoint


def totalDistance(calDict,teams):
 total = 0
 for date in calDict:
```

```
   games = calDict[date]
   for game in games:
     homeTeam = game[1]
     awayTeam = game[0]
     hDistance = teamDistance[homeTeam][0]
     aDistance = teamDistance[awayTeam][0]
     if (teamDistance[awayTeam][1] == None):
       teamDistance[awayTeam] = (distance[awayTeam,homeTeam] +
aDistance, homeTeam)
     else:
       previous = teamDistance[awayTeam][1]
       teamDistance[awayTeam] = (distance[awayTeam,previous] +
aDistance, homeTeam)
     if (teamDistance[homeTeam][1] == None):
       teamDistance[homeTeam] = (hDistance, None)
     else:
       previous = teamDistance[awayTeam][1]
       teamDistance[homeTeam] = (distance[homeTeam,previous] +
hDistance, homeTeam)
 for each in teamDistance:
   total += teamDistance[each][0]
 return total



def evaluate(calDict):
 allTeams(calDict)
 initialTeamSocores(calDict)
 btbNum = 0
 distanceSum = totalDistance(calDict, teams)
 for team in teamScores:
```

```
    teamScoreDelta, btbNumDelta = backToback(calDict,team)
    teamScores[team] += btbNumDelta
    btbNum += btbNumDelta
  btbStdev = getStdDev(teamScores)
  teamD = dict()
  for each in teamDistance:
    teamD[each] = teamDistance[each][0]
  distanceStdev = getStdDev(teamD)
  popularityScore = popularity(calDict)
  totalScore = (- 0.025) * btbStdev * 50 + (-0.6) * (btbNum**4) /
819247506.25 + (- 0.025) * distanceStdev / 80.0 + (- 0.3) *
(distanceSum/8000.0) \
  + (0.05) * popularityScore / 3000.0
  retObject = {}
  retObject["score"] = totalScore
  retObject["btbNum"] = btbNum
  retObject["btbStdev"] = btbStdev
  retObject["distanceSum"] = distanceSum
  retObject["distanceStdev"] = distanceStdev
  retObject["tvRatingScore"] = popularityScore
  # print "popularityScore = ", popularityScore
  return retObject
```