Finding the Optimal “Fast Pass” Proportion for an Amusement Park

using Discrete Event Simulation Optimization

Lauren Alley, Lauren Slade,

Anthony Khoudari, Joshua Linneburg

Oakland University

MIS/POM 4410, Fall 2018

**Introduction**

When we were presented with the challenge of designing and building a model to simulate the flow of customers through an amusement park, it quickly became apparent which parts of the assignment were more crucial than others. For example, we realized most of the decision-making in the model would take place at the central node, in the problem referred to as the “Big Board”. Developing sufficient logic to handle these decisions was our primary concern. Another such concern was to model the operation of both the Rollercoaster and Ferris Wheel as close to real-life as possible. From there, we felt that only once those two concerns were met; a sufficiently effective Big Board decision-making Add-On Process and a realistic representation of both rides could we worry about measuring satisfaction levels of customers based on wait times. In fact, measuring the wait time of customers at various points throughout the model was one of the last aspects of the model to find its way into our Simio file.

Our group, being comprised of two MIS majors, a Marketing major and a Finance major had a much-different approach to designing, developing and testing the model compared to other groups. For instance, we employed the “walking skeleton” approach to great effect. Our initial steps were to build a version of the model that was exceedingly simple: no TimePaths, Servers instead of Combiners and Separators, no load-in or unload times, and so forth. From there, we built the model iteratively, starting with the most-crucial aspects before continuing to the less-important ones. We decided from the beginning that it would be impossible to build the entire model with all the features we wanted in one-go. It would be much-easier to get the model working at a basic level first and add features as we became more comfortable with the assignment and the inner-workings of Simio itself.

**The Big Board Decision**

Based on the description provided in the assignment, our group sought to create a dynamic decision-making process at the Big Board which would adjust the list of possible outcomes based on their availability. A die whose numbers of sides varied with the number of viable options, if you will. Our group identified 8 possible outcomes from the decision at the *BigBoard*: they could go to any one of the six activity areas, be sent out of the park, or be asked to wait until one of the six activities becomes available if none are a viable path.

To accomplish this, our group created six Model Entity States in the form of *[AreaName]Viable*. Each time the *BigBoardDecision* Add-On Process is executed, the entity receives a value of 1 or 0 for each of these six Model Entity States, based on that specific area’s availability as laid out in the problem. At first, we had a rather simple implementation of this approach which used the Random.Discrete function in Simio, where all six areas had a 16.67% chance of being selected as the Model Entity’s *NextStop*, another Model Entity State created. The issue we encountered is this did not dynamically-change the list of possibilities based on their availability. In effect, it would roll a six-sided die and *then* decide if the choice was viable. If not, the entity would wait and roll again, and again, and again. When entities only had one or two possible options, we would oftentimes have them waiting at the *BigBoard* for 15+ minutes at a time before they finally “rolled” a valid choice. This was close, but not quite what the problem described.

A new approach was developed. Another Model Entity State was created, *NumberViable*, which was the summation of the 1s and 0s assigned in the first step of the Add-On Process. Six viable options meant a *NumberViable* of 6. The Random.Discrete function was used again, but in a slightly different form. The probability of going to a specific activity area, instead of being 16.67%, is expressed as:

Areas 1 through 6 were assigned to the rollercoaster, Ferris Wheel, Carnival, Arcade, Concessions and Bathroom, respectively. For example, the probability of going to the Arcade, , is:

This approach, while a rather long expression for an Assign step in our Add-On Process, guaranteed entities would only be able to be assigned a *NextStop* from the list of viable stops. Entities are guaranteed to be sent somewhere, if at least one area is a viable stop. We introduced an exception-handling step that first verifies this, and if then we ask the entity to wait 1 minute and re-run the process. With this working, entities are assigned a truly-random *NextStop* once they re-enter the *BigBoard*. Based on the value of *NextStop,* the entity is routed to that corresponding location. An additional step was added to the beginning of this process to verify for that entity. If it evaluates to False, the entity is sent to the park’s Exit, our Sink. Otherwise, the Add-On Process continues to the next step. This necessitated the creation of a *LengthOfStay* Model Entity State, which we will discuss further in the next section.

**Entering the Park**

Upon creation, customer entities are assigned another few key Model Entity States to determine how they behave when they enter the park. These are: *ArcadeInterest, MaxBathroom, LengthOfStay.* For debugging purposes, we also assign entities an *EnteredPark* State to verify in the Facility view when an entity should be routed to the exit. Entities assigned an *ArcadeInterest* of “False” will never go to the Arcade. And the *MaxBathroom* state is used to control the number of times an entity can use the Bathroom, alongside another state called *BathroomCounter.*  This is handled in the *CreateEntity* Process. The randomness of each of these Model Entity States is taken straight from the problem, each are assigned using the Random.Discrete function with appropriate cumulative probabilities.

Customers arrive to the park using a Time Varying Arrival Rate, as specified in the Rate Table *CustArrival.* Customers are assigned a type: Regular or Fast Pass, according to proportions specified in the *CustData.CustMix* table and column. This proportion is a Model Property that can be controlled in the Experiments window called *PctFastPass.* All values used in these tables were taken straight from the problem description.

**Tracking Food and Restroom Breaks**

Whenever a customer goes to the Bathroom or Concessions Servers, they invoke an Add-On Process that increments the *BathroomCounter* and *FoodCounter* Model Entity States by 1, respectively. Tracking these counters and their value relative to the maximum specified number of times an entity goes to that location is done in the first Assign step of our *BigBoardDecision* Process.

**Tracking Satisfaction**

Entities are assigned an *EnteredLine* State based on the *TimeNow* when they enter the FerrisLine or CoasterLine nodes. And because we have assumed the load-in time for the Rollercoaster is a constant, instead of the triangularly-distributed random variable as specified in the problem, we can measure the *ExitedRide* as the *TimeNow* when the customer enters the MemberOutputBuffer of the Rollercoaster or Ferris Wheel separators. These are subtracted to net us the *WaitingTime.* We compare this value to the *SatisfactionThreshold* for both Regular and Fast Pass entities, another Model Property, and increment the respective Model State by 1 based on the entity type and whether they were satisfied or not. This is converted to a percentage in our Experiments window.

All other metrics we were instructed to track can be found in the Experiments window. Six Model Tally Statistics were used to track the average and maximum waiting time for Regular and Fast Pass customers at the coaster, Ferris Wheel and both rides combined. These Tally Statistics are also updated in the Add-On Processes *CoasterSatisfaction and FerrisSatisfaction.*

**Model Assumptions, Shortcomings and Validation**

In our model, we assumed the Rollercoaster can support both cars on the track at the same time. We also assumed the Ferris Wheel is half unloaded and half loaded once every 5 minutes, instead of fully unloaded and loaded once every 10 minutes. We also assumed that Fast Pass customers do not have an absolute line priority over Regular customers. What we mean by this, is that there is a certain amount of time a Regular customer can stand in line which gives them priority over a just-entered Fast Pass customer. This is a Model Property known as RegularOffset, and we assign the Model Entity State *LinePriority* when we measure *EnteredLine.* We take the smallest value of *LinePriority* first in both ride queues, effectively mashing a First-In-First-Out queue with some priority aspects to it. We think management should explore how “powerful” they want the Fast Pass to be, right now a Regular customer who has been waiting at least .875 of an hour (52.5 minutes) will have a queue priority over a Fast Pass customer that just entered the queue. This is adjustable in the Experiments window.

One shortcoming we experienced was the inability to measure the wait time of entities once they entered the Combiners. So, we were forced to make the load-in time for the Rollercoaster a constant 9 minutes. Otherwise, our model works as advertised. Average wait time without any Fast Pass customers is roughly equivalent to what is described in the problem; 17 minutes in our model vs 20 minutes as described. Our Bathroom wait time is a fair bit lower than described in the problem; 6.8 minutes vs 9 minutes as described. The total number of entities created lines up with the expected value of 800, we are creating just over or under 800 entities on each simulation run. By our estimation, we should expect to have just over 225 entities in the park, on average. When our simulation is run, the NumberInSystem came out to roughly that, somewhere in the 230-240 range. Entities spend on average 206 minutes in the park, a little higher than the expected 195 minutes, but considering that we do not immediately kick entities out once their *LengthOfStay* has been exceeded (they are permitted to finish their activity) the higher values for NumberInSystem and TimeInSystem than their expected values make sense. Satisfaction decreases for both Regular and Fast Pass customers as the PctFasPass increases, as we expected.

Because it is possible for a Server or Combiner’s InputBuffer to have capacity when a customer evaluates their options at the *BigBoard* and to then have that capacity used up by an entity that was already in route when the evaluation was done, we added Balking mechanisms to each of the six activity areas to send an entity back so they do not have to wait for that activity to open up. This happens in our model rather infrequently but is still possible because we are not keeping track of the capacity of each queue once *NextStop* is assigned, but rather using the updated capacity of the queue once an entity reaches there. This might be possible but would add a great deal of complexity since we would effectively have to maintain a running total of the InputBuffer capacity for each area, when that functionality is already built into the program itself. We decided the added work and the convoluted nature of this process was not worth the time, and simply added a Balking mechanism to each area to serve as our backup.

**Model Results**

As far as our results go, we are confident they reflect an accurate representation of the problem which was provided to us. The statistics we were asked to track can be found in the table below, along with what value we were benchmarking our results against based on the problem description or our estimates, if we were able to make one. This table shows the results from what we deemed the optimal level of *PctFastPass* and *RegularOffset*, and we will discuss this more in the next section.

|  |  |  |
| --- | --- | --- |
| Statistics (*PctFastPass = .175, RegularOffset = .875)* | Value | Benchmark Value |
| Fast Pass Average Time in Coaster Queue (Minutes) | 5.75 | <10 |
| Fast Pass Average Time in Ferris Queue (Minutes) | 4.20 | <10 |
| Regular Average Time in Coaster Queue (Minutes) | 40.2 | <60 |
| Regular Average Time in Ferris Queue (Minutes) | 14.0 | <60 |
| Fast Pass Maximum Time in Coaster Queue (Minutes) | 13.8 | 10 |
| Fast Pass Maximum Time in Ferris Queue (Minutes) | 7.20 | 10 |
| Regular Maximum Time in Coaster Queue (Minutes) | 66.6 | 60 |
| Regular Maximum Time in Ferris Queue (Minutes) | 36.0 | 60 |
| Fast Pass Average NumberInSystem (Customers) | 39.5 | 39.4 |
| Regular Average NumberInSystem (Customers) | 197 | 185 |
| Fast Pass Average Time in Queue (Minutes) | 5.10 | 10 |
| Regular Average Time in Queue (Minutes) | 29.7 | 60 |
| Bathroom Average Time in Queue (Minutes) | 4.60 | 9 |
| Bathroom Average Number in Queue (Customers) | 5.50 | N/A |
| Fast Pass Satisfaction | 90.6% | 90% |
| Regular Satisfaction | 90.7% | 90% |

Clearly, our results line up favorably with the benchmark values we wanted to either meet or get close to. Our maximum wait time on the Rollercoaster exceeds the *SatisfactionThreshold* for both Fast Pass and Regular customers, we would recommend that management looks at this as the largest bottleneck in the entire park. Other groups had experienced a bottleneck at the Ferris Wheel, but given our different processing logic, the Ferris Wheel completes half a cycle every 5 minutes instead of a full cycle every 10, we were able to unload and load entities much quicker than other groups, perhaps a bit too quickly. We might want to investigate delaying the entire Ferris Wheel TimePath when the other *FerrisBucket* is being loaded; right now, the first bucket does not wait until the second bucket enters the TimePath. This might make a huge difference but would make it even more realistic.

**Conclusion**

The overall task of this assignment was to find what we found to be the optimal mix of Regular and Fast Pass customers in the park to be 82.5% Regular, 17.5% Fast Pass with Regular customers gaining priority over Fast Pass customers that just entered the ride queue if they’ve waited .875 of an hour (52.5 minutes). We wanted a combination that netted us a Satisfaction rating for both Regular and Fast Pass customers of at least 90%. This level of Fast Pass customers with this offset gets us almost exactly that, as specified in the table above.

Management is free to adjust the *PctFastPass* and *RegularOffset* to tune the model to their liking, and if they are comfortable with more unsatisfied customers, should feel free to increase the *PctFastPass* accordingly. They should be warned, however, that Satisfaction decreases at an increasing rate. So, a 1% increase in *PctFastPass* might result in a 2% reduction in Satisfaction across the board. We’d recommend no higher than 25%, at this level of *PctFastPass,* customers were only satisfied ~75% of the time. We deemed this far too low and used 25% as an absolute maximum value for *PctFasPass* in our testing.

This was both a challenging and rewarding assignment. And as a group, we all were pleased with our efforts to come together as a team and create something as interesting and useful as this. There are many ways to approach a project like this but we were fortunate our first approach ended up working as intended. Of course, there were challenges involved in creating the model.

The biggest challenge we faced was the concept modeling a real life example and having to reflect that details in a simulation. The biggest challenge we faced was knowing at what point our model was sufficiently complex to model the problem we were posed with. Identifying the point where we had done enough work to make the model close to real-life, and to then start work on validating and applying the model to the problem, was the biggest issue we faced. We are all sure that, if given enough time, we could make the model infinitely more complex and close to real-life. An example would be the *RegularOffset* which was introduced to mix the FIFO and priority queues. A real-life implementation of such a system would be much more complex to model here, but we thought what we had was sufficient to capture the required details of the model, and nothing more. At a certain point, we would encounter diminishing marginal returns that would greatly increase the complexity of the model at no added benefit to the analysis it is able to provide for the amusement park manager. Overall, we did the best we could with this assignment given our individual abilities, the time constraints we faced and the nature of the problem set before us.