CS5234 Project
Airport Gate Scheduling

Choo XianJun, Davin
Lee Yik Jiun

Contents

Chapter 1

Introduction

1.1 Background

Scheduling is an interesting class of combinatorial optimization problems that have very practical use. One such scheduling problem is the **airport gate scheduling problem**.

In this problem, we have n_1 arriving flights and n_2 departure flights. Associated with each of these flights is a timing x_i, where $i \in \{1, ..., n_1 + n_2\}$, indicating the time of arrival or departure of the respective flight. Depending on the type of flight, different amount of buffer times are allocated, resulting in an interval of time which the plane needs to use a gate. Departing flight i can be represented as the interval $[90 - x_i, x_i + 30)$ while arriving flight j can be represented as the interval $[x_j, x_j + 45)$. The problem then requires us to minimise the number of gates needed to satisfy a given list of flights so that no intervals collide.

Beyond the above vanilla formulation of the airport gate scheduling problem, there are many variations and modifications to the problem. Life is hardly predictable and so are scheduled flight timings. Airports encounter several flight delays[1] daily. This presents an interesting variant of the problem which considers delays in the system.

1.2 Organisation of report

We first present some literature review and plausible approaches to the problem. After tackling the vanilla airport gate scheduling problem without delays, we delve into the more interesting variant which considers delays in the system. Along the way, we will provide sufficient mathematical rigour and point out interesting patterns and observations that we have.

[1] Delays can be negative, which simply means the plane took off/landed earlier than expected!

Chapter 2

Vanilla Airport Gate Scheduling

2.1 Interval scheduling

2.1.1 Problem description

Given a set of jobs which must take place between their respective interval periods of time, what is the maximum number of jobs that we can fulfill without any interval overlaps? In interval scheduling, we are restricted to only 1 processor. That is to say, everything must be scheduled on a single timeline. This is also known as the "Interval Scheduling Maximizatin Problem (ISMP)".

There are several other variants to this problem where the set of jobs is described as a union of different groups of jobs. The problem of "Group Interval Scheduling Maximization Problem (GISMP)" is then concerned about maximizing the number of groups such that at least one job in these groups is being executed.

In the context of interval scheduling airport gate scheduling can then be seen as interval scheduling with multiple processors, where we aim to minimize the number of processers needed. We illustrate this in Figure 2.1.

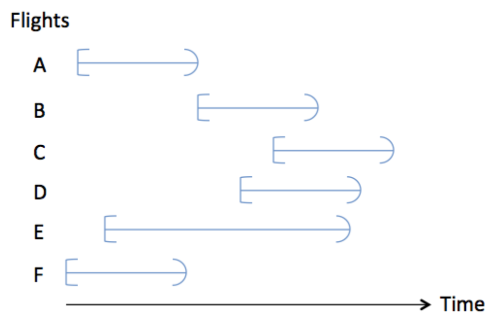


Figure 2.1: Illustration of flights as time intervals

2.1.2 Optimal solution

It turns out the interval scheduling problem has a greedy optimal algorithm (Algorithm 1) of packing as many earliest ending jobs as possible.

Algorithm 1 GreedyIntervalScheduling(Jobs)

```
1: $J \leftarrow$ SortByEarliestFinishingTime(Jobs)
2: $Chosen \leftarrow \emptyset$
3: while $J$ not empty do
4:     $x \leftarrow J.pop()$
5:     $Chosen \leftarrow Chosen \cup \{x\}$
6:     for $y \leftarrow J \setminus \{x\}$ do
7:         if $y$ overlaps with $x$ then
8:             $J \leftarrow J \setminus \{y\}$
9:         end if
10:    end for
11: end while
12: return $Chosen$
```

Proof of correctness

It is obvious that if k jobs overlap at any point in time, we can at most execute one of them.

Claim 1. *Suppose k jobs overlap at time x. Let t be the earliest start time of these k jobs. Assume that from time t to x, there are no overlaps with jobs outside of these k jobs. Then, compared to picking the earliest ending job j, we can do no better by picking any other job j' that ends later.*

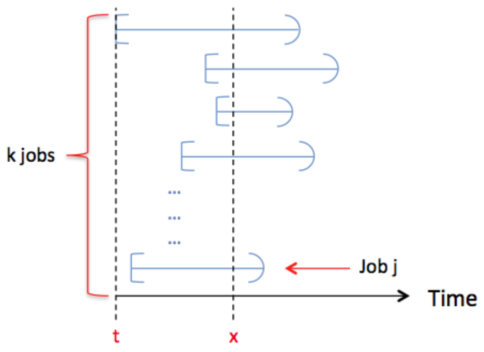


Figure 2.2: Our best option is to pick Job j

Proof. By assumption, from time t to x, the choice of job within the k jobs will not affect any other jobs outside of themselves. So, picking j over j' will not result in additional overlaps beyond the k jobs. Now, suppose we construct an optimal solution by picking

j' instead of j. Then, we can obtain a feasible solution by replacing j' by j. Hence, the optimal solution involving j is at least as good as the solution generated by picking j' out of the k jobs. $\square$

Let `Chosen` $= \{x_1, x_2, ..., x_m\}$ and $e_i =$ end time of job $x_i, \forall i \in \{1, 2, ..., m\}$. Then we can repeatedly apply the claim's argument at times $t = \{0, e_1, e_2, ..., e_{m-1}\}$ to show that the greedy algorithm is optimal.

Algorithmic analysis

Let n be the number of jobs.
Then, the greedy algorithm takes $O(n \log_2 n + n^2) = O(n^2)$.
$O(n \log_2 n)$ comes from the sorting while $O(n^2)$ is due to $O(n)$ pairwise job comparisons to check for overlaps, each taking $O(n)$ time.

2.1.3 Attempt to adapt to airport scheduling

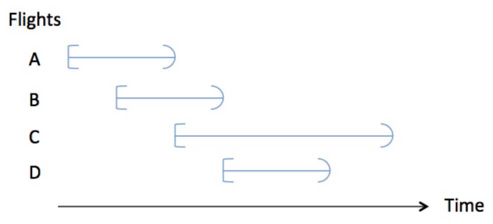
Motivated by what we learnt in interval scheduling, we came up with a naïve solution to airport gate scheduling. We attempted to modify the greedy interval scheduling algorithm as follows:

Algorithm 2 GreedyAirportScheduling(Flights)

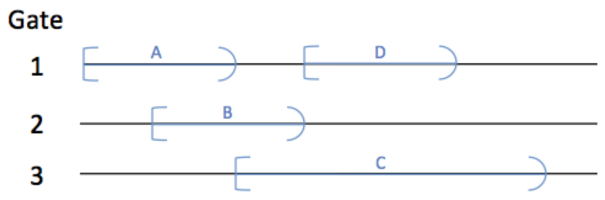
```
1: $Gates \leftarrow \emptyset$
2: while $Flights$ not empty do
3:     $New\_Gate \leftarrow$ GreedyIntervalScheduling(Flights)
4:     $Flights \leftarrow Flights \setminus New\_Gate$
5:     $Gates \leftarrow Gates \cup New\_Gate$
6: end while
7: return $Gates$
```

Counter example

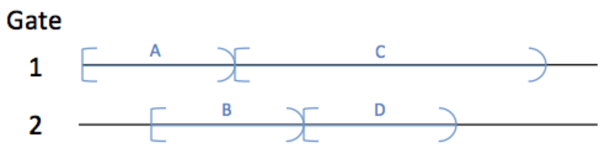
Unfortunately, not surprisingly, greedy algorithms do not always produce optimal solutions. During our analysis of our naïve attempt, we found the following counter example, as shown in Figure 2.3:



(a) Counter example



(b) The naïve attempt uses 3 gates.

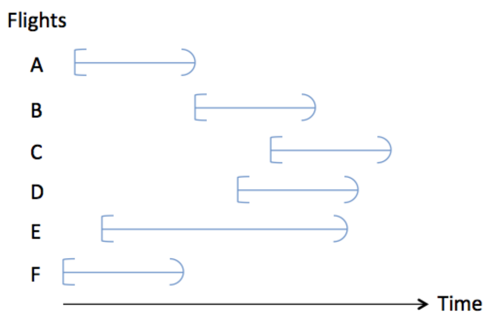


(c) The optimal gate usage is 2.

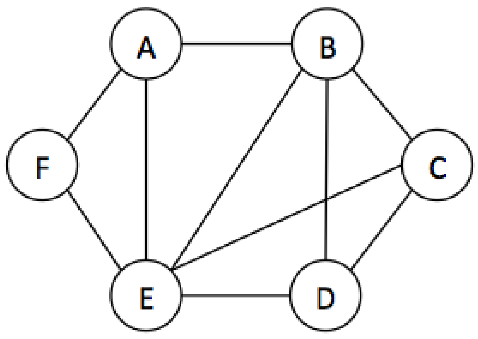
Figure 2.3: Counter example to optimality of naïve attempt

2.2 Scheduling as a Graph Colouring problem

Brushing aside our small setback, we took another look at the problem and realised that we can model our airport scheduling problem as a graph colouring problem. Let $G = (V, E)$ be the graph, where V is the set of flights, and $(u, v) \in E$ means flights u and v have a collision in their schedules. In this formulation, finding a minimum colouring on the resultant graph G will provide us an optimal solution to our original airport scheduling problem.



(a) Airport gate scheduling problem



(b) Graph representation

Figure 2.4: Modelling airport gate scheduling as a graph

The problem of graph colouring is well explored in the Graph Theory branch of mathematics. In graph theory notation, **max degree vertex** $\Delta(G) = \max_{v \in V} deg(v)$ represents the highest degree of any vertex in the graph G, and **chromatic number** $\omega(G)$ means the minimum number of colours needed to colour the graph G. Clearly, $\omega(G) \leq \Delta(G) + 1$. Algorithm 3 presents a simple method to colour the graph in at most $\Delta(G) + 1$ colours.

Algorithm 3 GreedyColour(V, E)

```
1:  $C \leftarrow$ Array of 0's of size $|V|$
2:  for $v \leftarrow V$ do
3:      $N \leftarrow$ GetNeighbours(v, E)
4:      $c \leftarrow 1$
5:      for $n \leftarrow N$ do
6:          $c \leftarrow \min(\{1, ..., |N| + 1\} \setminus \{c[n] : n \in N\})$
7:      end for
8:      $C[v] \leftarrow c$
9:  end for
10: return $C$
```

2.2.1 Proof of correctness

Observe that we always try to assign the smallest 'free'/unused colour (from 1 to $|N| + 1$). We claim that throughout the entire algorithm, the following invariant holds.

Claim 2. *For any vertex v,* $C[v] = c \leq deg(v) + 1$.

Proof. In the worst case, all neighbours of v use different colours, from 1 up to $deg(v)$. So, v has to use the colour $deg(v) + 1$. □

Since this holds for all vertices, it holds for the vertex with degree $\Delta(G)$. Hence, the algorithm gives us a colouring with at most $\Delta(G) + 1$ colours.

2.2.2 Proof of optimality

Here, we claim that after transforming our problem into a graph, Algorithm 3 provides us with an optimal schedule.

Observe that in interval scheduling, if there are k overlaps at any time t, then we can at most include one of the k flights. In a graph, these overlaps would be represented by a k-clique. It is easy to see that $\Delta(G) + 1$ will simply be the maximum number of overlaps. Since we need at least $\Delta(G) + 1$ colours to resolve this maximum overlap, and our algorithm provides such a colouring, it is optimal.

2.2.3 Algorithmic analysis

The greedy algorithm takes $O(|V| + |E|)$. This is because we considered each vertex and edge once in the rest of the entire algorithm[1].

2.3 Interval partitioning

Despite having an optimal solution, it can be time/space consuming to store the data in a graph before we can actually execute the above algorithm. So, we looked for alternative solutions.

In our search, we came across a problem known as “Resource Allocation” or “Interval Partitioning” which describes exactly the airport gate scheduling problem[2], but in a different context.

The problem can be formulated as follows: Given a set of n jobs with their start and finish times, find a partition of size k such that each set of jobs in the partitions do not overlap and k is minimized. In our case, “jobs” refer to flights and “partitions” refer to “gates”.

Algorithm 4 is the greedy, yet optimal, algorithm for the interval partitioning problem, which minimizes |`Partition`| while satisfying the condition that all jobs in the partitions do not overlap. We pass in an empty list as `Partition`. The reason for using it as a parameter is to allow us to pass in non-empty lists as initial partitions[3].

Algorithm 4 GreedyIntervalPartitioning(Jobs, Partiton)

```
1: $J \leftarrow$ SortByEarliestStartTime(Jobs)
2: $count \leftarrow 0$
3: for $j \in J$ do
4:     if $j$ does not overlap with all existing jobs in some Partition $k$ then
5:         Partition$[k] \leftarrow$ Partition$[k] \cup \{j\}$
6:     else
7:         $count \leftarrow count + 1$
8:         Partition.$append(\emptyset)$
9:         Partition$[count] \leftarrow$ Partition$[count] \cup \{j\}$
10:    end if
11: end for
12: return Partition
```

For our purposes, when given a choice of possible partitions in Line 4, we choose the one with the most slack. This allows us to extend our algorithm to the case when we are given extra gates to begin with.

[1] We can efficiently check the neighbours of a vertex if we store the graph in an adjacency list.

[2] References:
`http://www.cs.rit.edu/~zjb/courses/800/lec8.pdf`
`http://courses.cs.vt.edu/cs5114/spring2009/lectures/lecture04-greedy-scheduling.pdf`

[3] This allows us to schedule with extra gates during the experiments in Section 4.3.

2.3.1 Proof of correctness

By construction, the check in Line 4 of the algorithm will ensure that the resultant partitions do not have overlapping jobs.

2.3.2 Proof of optimality

Let d be the maximum number of overlaps at any point in time. i.e. $d = \Delta(G) - 1$. Since we know that $|\texttt{Partition}| \geq d$, if we can show that $|\texttt{Partition}| = d$, then the algorithm is optimal.

Claim 3. $|\textit{Partition}| = d$

Proof.
Recall that $|\texttt{Partition}| \geq d$ and $|\texttt{Partition}| = \textit{count}$ within the algorithm.
Suppose not. Then let us assume that $|\texttt{Partition}| > d$.
Then at some point in our algorithm, $d = \textit{count}$. Consider the job j that caused us to increment *count* (i.e. We create a $(\textit{count} + 1)^{th}$ partition for j).
By the algorithm, we only do so when all the current partitions $\texttt{Partition}[1]$ up to $\texttt{Partition}[\textit{count}]$ all have overlaps with job j. This implies that at the start time of job j, we have $(\textit{count} + 1) \geq d + 1 > d$ number of overlaps.
But this contradicts that d is the maximum number of overlaps at any point in time!
So, $|\texttt{Partition}| > d$ is impossible. Hence, we have that $|\texttt{Partition}| = d$. □

2.3.3 Algorithmic analysis

Let n be the number of jobs and d be the maximum number of overlaps at any point in time. Then, the algorithm takes $O(n \log_2 n + n \cdot d) = O(n \cdot (\log_2 n + d))$ time to run.
This is because we take $O(n \log_2 n)$ time to sort the jobs. Then, for every job, we compare with each existing partition, which takes $O(d)$ time each.

2.4 Interesting observations

The most interesting observation is probably that the number of gates required is exactly d, where d is the maximum number of overlaps at any point in time. As we discussed before, it is an obvious lower bound since we need at least one gate per plane when we incur d overlaps. However, we were initially pleasantly surprised that this lower bound was actually attainable.

On hindsight, if we formulated the problem in to graph colouring problem, then the result would not have been so surprising after all. Graph colouring gives us an obvious upper bound of $\Delta(G) + 1$ that actually equals d. It was nice to see different fields of study come together.

Chapter 3

Airport Gate Scheduling with Delays

In real world situations, delays are usually inevitable. In our context, a delay happens when the actual flight arrival or departure time differs from the scheduled one. It can be a positive or negative amount of delay (Negative delay indicates earlier arrival/departure than expected). In this chapter, we consider the same problem of airport gate scheduling and explore how delay affects our scheduling.

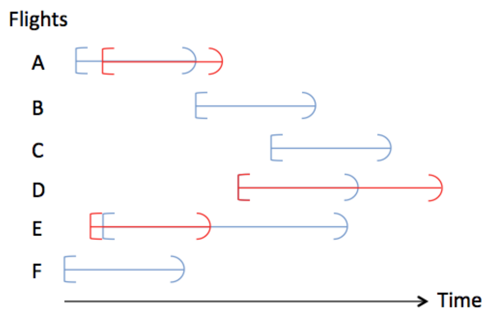


Figure 3.1: Illustration of delays:
Flights A and D have positive delay while Flight E has negative delay.

3.1 Definitions

Before we begin, let us first lay out some definitions.

3.1.1 Delays, Gate Collision and Re-assignments

Delay (Measured in minutes)

A delay is an integer $d \in \mathbb{Z}$, which can be either positive or negative. It is the time difference between the scheduled flight arrival/departure and the actual one.

We usually have to plan based on the scheduled timings and then make impromptu changes to the schedule as delays occur. A plane that has non-zero delay is also called a delayed plane.

Gate Collision

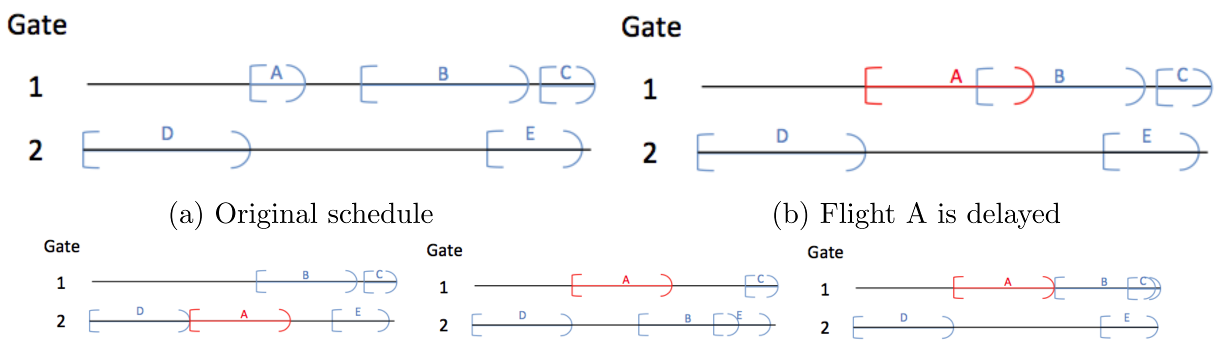
A gate collision happens when using the original schedule and the delayed planes, two planes are assigned to the same gate at the same time.

Re-assignment

A re-assignment occurs when we move a Plane p from some Gate i to some Gate j. We need to do this when a gate collision happens when we take into account the delay of Plane p.

Relation between gate collisions and re-assignments

While it is clear that gate collisions are bound to cause re-assignments, it is perhaps less obvious that re-assignments can also cause gate collisions. Consider the following scenario where we put into effect Plane i's delay, which caused a collision with Plane j:



(a) Original schedule (b) Flight A is delayed

(c) Shift Plane A (d) Shift Plane B (e) Delay Plane B

Figure 3.2: Illustration of the various options of resolving gate collisions

Assuming we have sufficient resources, one method that always works is to just open a new gate and assign either Plane i or Plane j there. That will definitely solve the issue. However, resource is usually tight, so we consider the the following options:

(a) Shift Plane i to another existing gate

(b) Shift Plane j to another existing gate

(c) Keep both planes in the same gate. Push Plane j's usage of the gate backwards (push back the take-off time or ask it to circle around in the air for a bit before landing)

Each of these options are recursive in nature, where we keep picking an option until we finally manage to get rid of all gate collisions due to Plane i's delay.

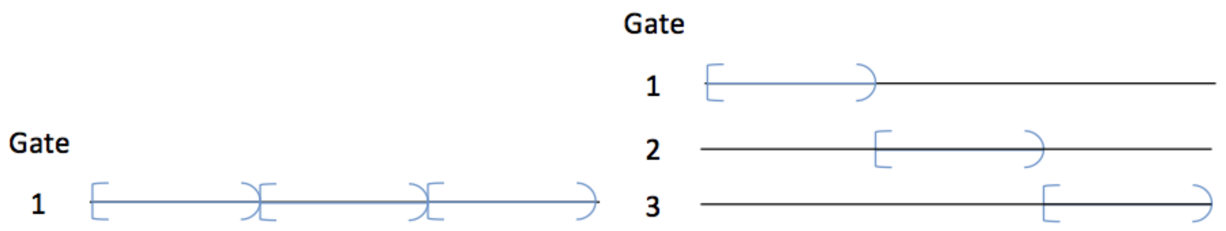
There is no simple method to determine before hand which of the above options is optimal since future planes can be delayed as well. In the worst case scenario, while resolving the single gate collision due to Plane i's delay, we could potentially cause a whole lot more gate collisions between other planes. Unfortunately, it is non-trivial to predict beforehand which of the options is the best.

3.1.2 Slack

The key concern when scheduling with delays is to minimise the number of gate collisions and re-assignments. Unfortunately, unless we have an oracle, we cannot measure or predict beforehand. The next best thing we can do is to employ an auxiliary/intermediary variable and optimize over that.

Slack is a useful notion that serves this purpose. Intuitively, a schedule with more slack would be more robust to perturbation in the flight schedules since the slack will likely 'absorb' delays and avoid gate collisions.

For illustration purposes, let us consider the extreme cases in Figure 3.3. When all planes and gates are packed like sardines in a can, a slight amount of delay in any of the planes would cause a re-scheduling catastrophe. On the other hand, if we assign a gate to every single plane, then we are guaranteed that no amount of delay will cause any form of gate collisions.



(a) 0 slack: No free space between planes (b) Maximum slack: 1 gate per plane

Figure 3.3: Illustration of the extreme slack cases

While it is easy to convince ourselves that slack could be useful for our purposes, there are 2 more tasks at hand: (a) Proving that maximizing slack guarantees minimising gate collisions/re-assignment; and (b) Giving a working definition of slack.

For (a), we are unable to formalize the problem hence we did not manage to tackle it. We are also unable to find any work that seem to have made progress on this. So, for now, let us just trust that our intuition that "greater slack implies lower number of gate collisions" is reasonable.

As for (b), we employ a rather strict working definition of slack, and we aim to maximize it: Slack is the minimum free time between any pair of flights across all gates. We show a graphical example in Figure 3.4.

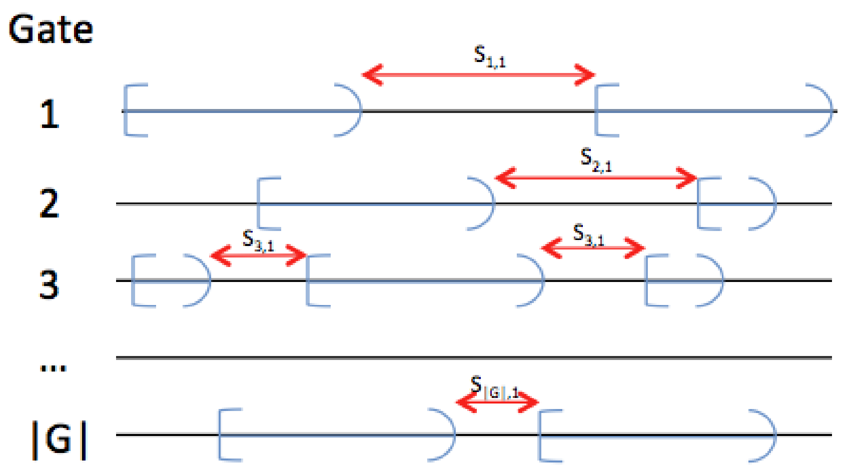


Figure 3.4: Slack $= \min x_{i,j}, \forall i, j$

Although in this project we aim to maximise slack, it is perhaps important to point out that reality is much more complicated. Companies often want to minimize slack in order to reduce wastage of resources such as idle gate time. At the same time, they wish to minimize gate collisions. This leads to an interesting trade-off that we will not explore in this project.

3.2 Our method

3.2.1 Handling gate re-assignments

We first calculate the optimal schedule based on "scheduled arrival/departures". Then, we consider the following scenarios:

1. With random probability p, add random delay in the range [`minD`, `maxD`] to each plane

2. Use the "actual arrival/departures" from the data

Algorithm 5 describes our approach for rescheduling. We will explore the above two situations in Section 4.3.

Algorithm 5 RESCHEDULING(Schedule, Planes)

```
1: $P \leftarrow$ SORTBYEARLIESTDELAYEDSTARTTIME(PLANES)
2: Overflow $\leftarrow$ []
3: for $p \in P$ do
4:     if $p$ collides in current schedule then
5:         Free Gate with most slack $\leftarrow$ Schedule $\cup$ Overflow
6:         if No such free gates then
7:             Assign $p$ to a new gate
8:             Assign newly created gate to Overflow
9:         else
10:            Reassign $p$ to free gate
11:        end if
12:    end if
13: end for
14: return Schedule $\cup$ Overflow
```

3.2.2 Scheduling with extra gates before hand

Suppose we are now given G gates to begin our assignment. We simply pass in a list of G empty gates instead of an empty list to the parameter `Partition` in our Algorithm 4.

Chapter 4

Implementation and experiments

4.1 Dataset

We obtain our dataset from the "Research and Innovative Technology Administration Bureau of Transportation Statistics" (RITA)[1]. The database provides us with 471950 flight details for the month of January 2014. Table 7 explains the attributes which we extracted from the database.

Some flight entries have missing fields so we omit those in our computation. We also filter away airports that are not busy (< 100 flights) since they usually produce anomalous results in our experiments. As a result, we only look at a total of 736601 flights[2] and 1788 airports across the entire month of January 2014[3].

Header in dataset	Attribute
`FL_DATE`	Date of flight
`ORIGIN_AIRPORT_ID`	Identifier of airport which flight took off from
`DEST_AIRPORT_ID`	Identifier of airport which flight landed in
`CRS_DEP_TIME`	Scheduled departure time
`DEP_TIME`	Actual departure time
`DEP_DELAY`	(Actual - Scheduled) departure time[4]
`CRS_ARR_TIME`	Scheduled arrival time
`ARR_TIME`	Actual arrival time
`ARR_DELAY`	(Actual - Scheduled) arrival time[5]

Table 4.1: Flight Attributes

[1] Database download:
`http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236`.
Attribute description:
`http://www.transtats.bts.gov/TableInfo.asp?Table_ID=236`

[2] Flights are double counted: once for departure and once for arrival.

[3] Without filtering airports to have at least 100 flights, we would have been considering a total of 880906 flights and 9111 airports. But we feel that the filtered dataset is still reasonably large enough.

We actually do not make use of the attributes `DEP_DELAY` and `ARR_DELAY`. We just pulled them out of the database for referencing purposes.

4.2 Implementation of algorithms

We provide the source code of our implementation and experiments in the Github repository here[6]. In the repository, besides our report, you will find the following source programs:

Name	Purpose
data.py	Processes data from RITA
experiments.py	Script that runs our experiments
schedule.py	Assigns and re-assigns flights to gates
util.py	Helper functions (e.g. time manipulation)
Gate.py	Gate class
FlightInfo.py	Flight info class
Interval.py	Interval class
test_data.csv	Data pulled from RITA

Table 4.2: Project Files

In our entire project, we treat time in terms of minutes within a day. That is to say, time will be represented by a natural number from 0 to 1439.

4.3 Experiments

From empirical data, we observe that the probability of an actual delay[7] p is approximately 0.948 (rounded to 3 decimal places). So, if we want to introduce random delays to the data, we would delay each plane with a random delay[8] in the range `[minDelay, maxDelay]`[9] with probability p. In our experiments, we use a symmetric range for random delay. That is to say, we set `|minDelay| = |maxDelay|`.

4.3.1 Scenario 1 - Establishing a basic relationship

In this scenario, we investigate the relationship between the number of flights and the number of gates needed.

[4] Negative delay means earlier departure.
[5] Negative delay means earlier arrival.
[6] `https://github.com/sozos/RAS`
[7] The fraction of flights that have differing scheduled and actual departure/arrival times
[8] In minutes
[9] `minDelay` can be negative

While our optimal algorithm (Algorithm 4) tells us that the number of gates is exactly the highest number of overlaps in any point in time, we cannot use that to answer this query since we do not know the distribution of overlaps in flights.

Therefore, we compute the number of flights and gates needed for all the airports, across all days, within the month of January.

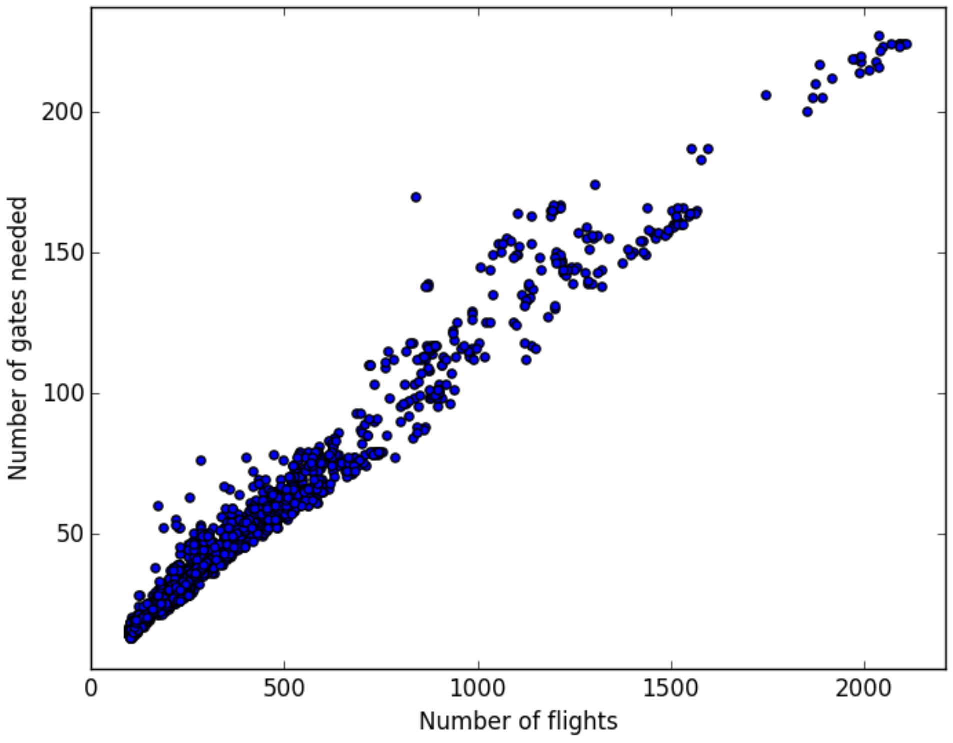


Figure 4.1: Relationship between number of flights and number of gates needed

Our findings show that there is a somewhat linear relationship, which makes intuitive sense.

4.3.2 Scenario 2 - Artificial random delays

In this scenario, given an optimal schedule using the scheduled times of flights, we wonder how much random delay is needed before we observe a collision.

To do so, we introduce random delay to perturb the flight timings. Clearly, the amount of delay that can be tolerated would depend somewhat on how packed the schedule is. Hence, we compare this value against our definition of slack (as per Subsection 3.1.1) across all airports, across all days, within the month of January.

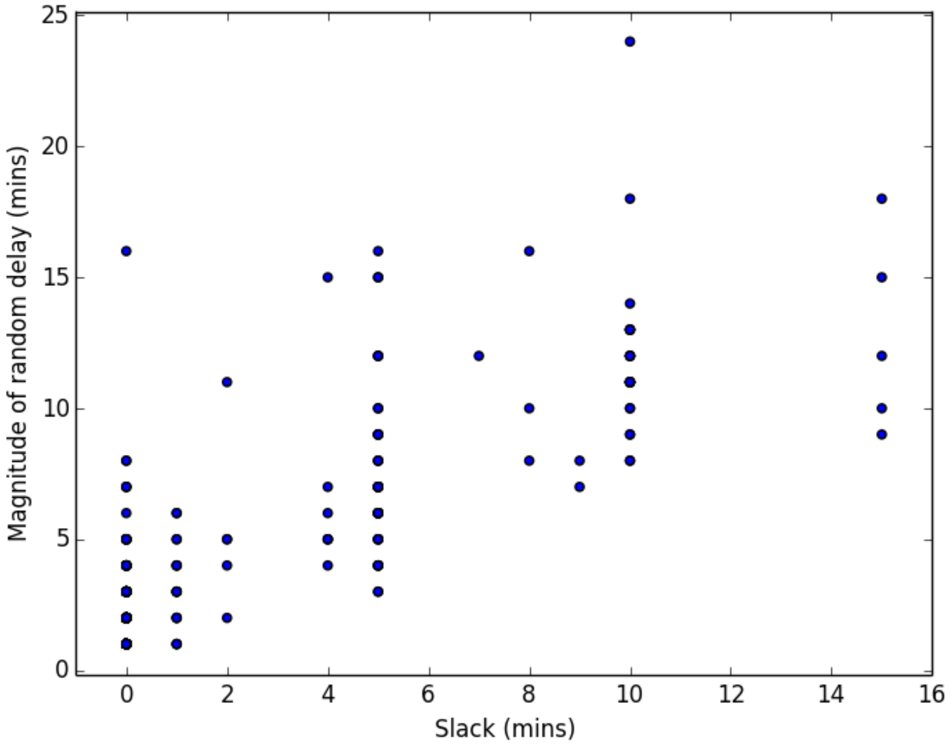


Figure 4.2: Smallest magnitude of random delay before first collision, with respect to slack at airport

Notice that the bottom right section of Figure 4.2 is devoid of data points. That tells us empirically that as minimum slack increases, we have some sort of guarantee on the minimum magnitude of random delay that is needed before we see any collisions.

Now, we explore a scenario similar to the previous, but from a slightly different point of view. Instead of increasing the magnitude of random delay until we observe a collision, we consider the number of gate collisions given a fixed level of random delay.

From the dataset, we observe that the mean magnitude of delay is 23.68091 minutes (rounded to 5 decimal places), so we introduce random delays in the range `[-24, 24]` across all airports, across all days, within the month of January.

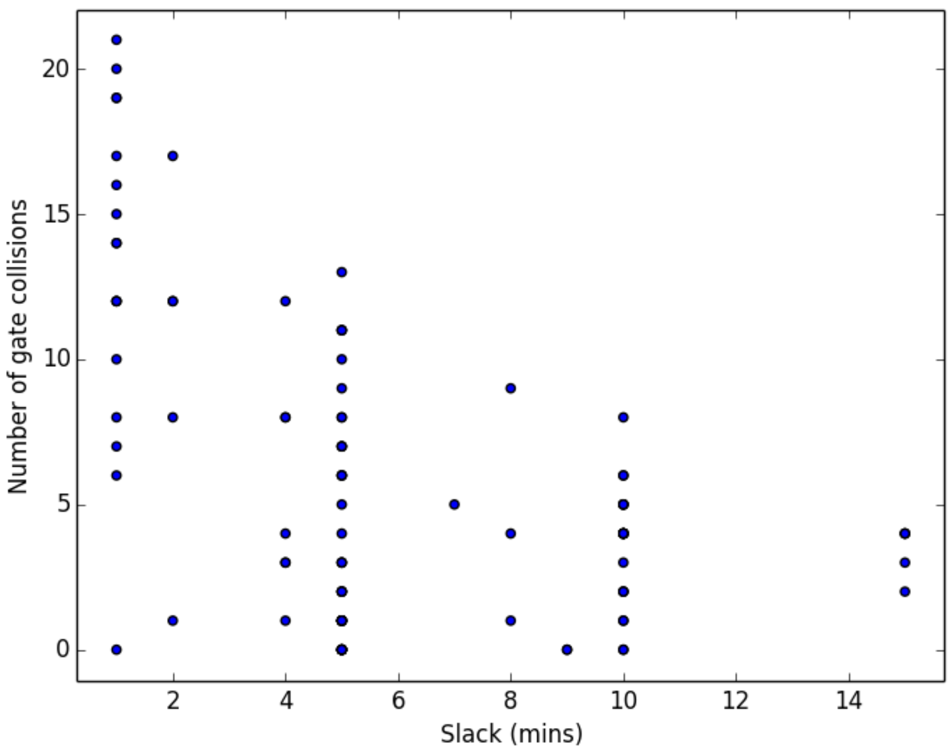


Figure 4.3: Number of gate collisions when random delay is introduced, with respect to slack at airport

Obviously when we have no slack, the number of collisions are astronomical. There are 1683 such cases with the maximum number of gate collisions being 702. Thankfully, we see that the number of collisions drops drastically when we have just 1 minute of slack, as shown in Figure 4.3. We can vaguely see the outline of a reciprocal shape in the figure as well.

4.3.3 Scenario 3 - Investigating actual delays

Having played with artificial delays, we now turn our attention to the actual delays from the real dataset. We first look at the number of gate collisions that occur in reality.

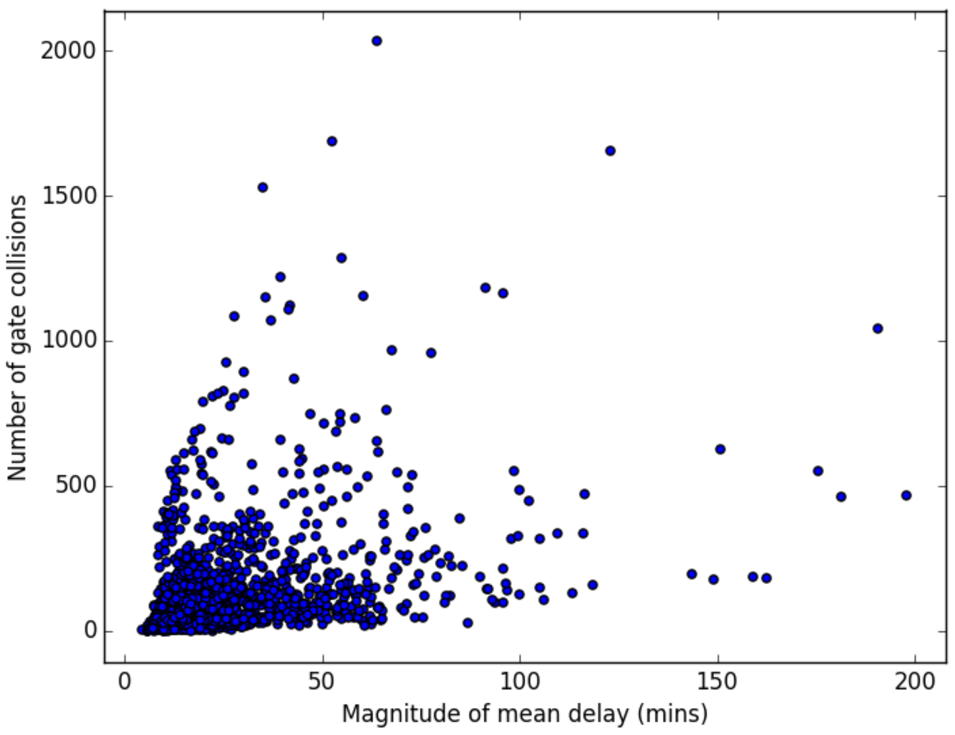


Figure 4.4: Number of collisions w.r.t. the magnitude of mean delay at the airport [10]

Observe that there is no clear trend in Figure 4.4 except that most airports cluster in the bottom left portion of the graph. Perhaps mean delay is not indicative of the number of gate collisions that can happen at an airport. We looked at the raw data and there are certain cases where planes are delayed for hours. Situations like those could result in tons of gate collisions, especially if the airport is packed with low amount of slack.

Next, we run our rescheduling algorithm and compare it against the ideal case (where an oracle provides us the actual departure/arrival time and we just run our optimal scheduling algorithm on that). Below, we show our results with relation to the mean amount of delay for each airport, across all days, within the month of January.

For Figures 4.5 and 4.6, we overlay 3 different sets of data for visualization purposes. On each graph, we show: (i) Result from our rescheduling algorithm; (ii) Result from the oracle scheduling; (iii) The difference between our result and the oracle's. As we can see, our rescheduling algorithm fares pretty well.

Note that for many situations with low magnitude of mean delay, the optimal algorithm actually took less gates than the optimal gates required in the original schedule (hence "negative extra gates"). In those situations, a "difference" comparison is pretty meaningless.

[10] Magnitude of mean delay of an airport $= \frac{\sum |\texttt{actual flight time}-\texttt{scheduled flight time}|}{\texttt{Number of flights at airport}}$

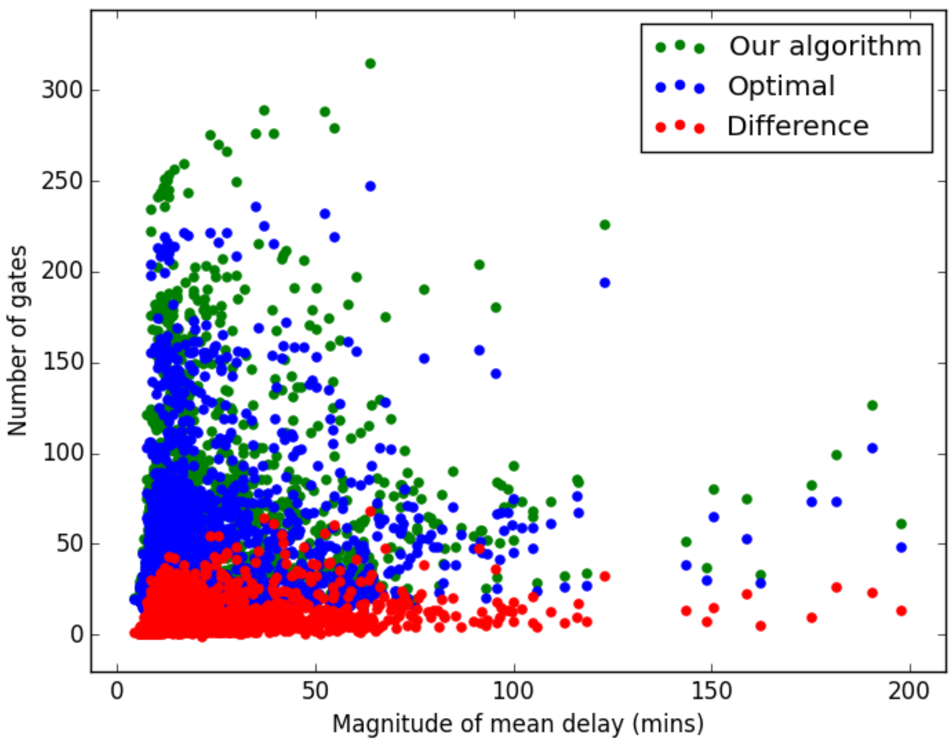


Figure 4.5: Number of gates w.r.t. the magnitude of mean delay at the airport

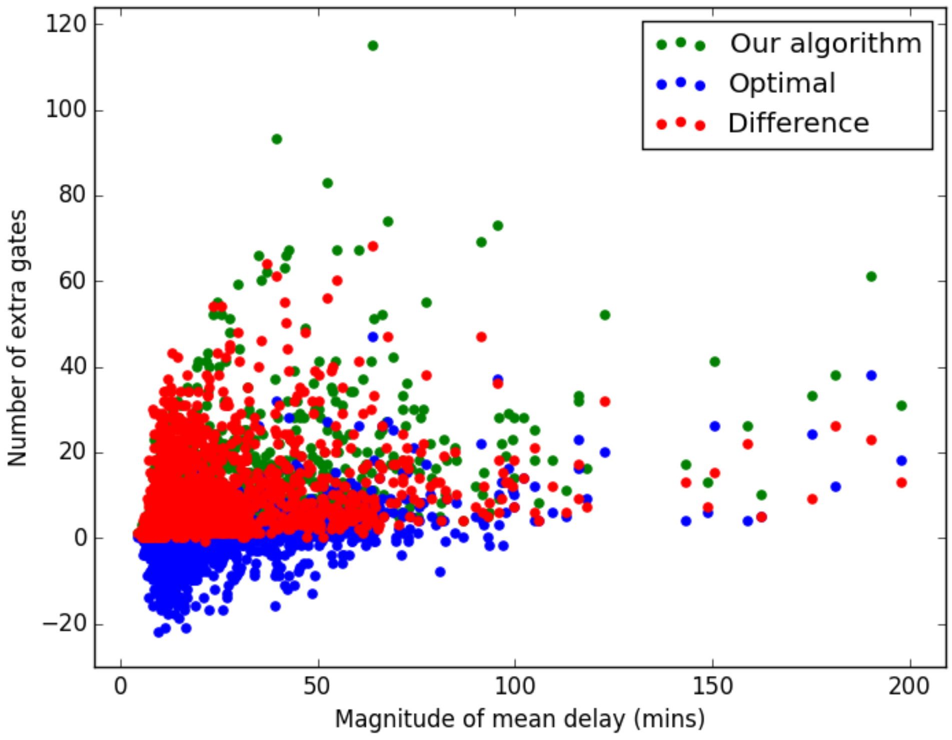


Figure 4.6: Number of extra gates w.r.t. the magnitude of mean delay at the airport

4.3.4 Scenario 4 - Minimizing number of gate re-assignments

Lastly, we consider the situation where we are given E extra gates on top of the minimum number of gates required to plan the scheduled timings.

As a basis for comparision, we consider the range of extra gates from none, to having an extra gate per plane[11]. We denote this as the "percentage of extra gates". Note that the number of extra gates given at the start is not a fixed constant but will vary based on the number of flights at the airport.

Figures 4.7 and 4.8 compare the number of reassignments and percentage of success across all airports in our dataset[12] against the percentage of extra gates.

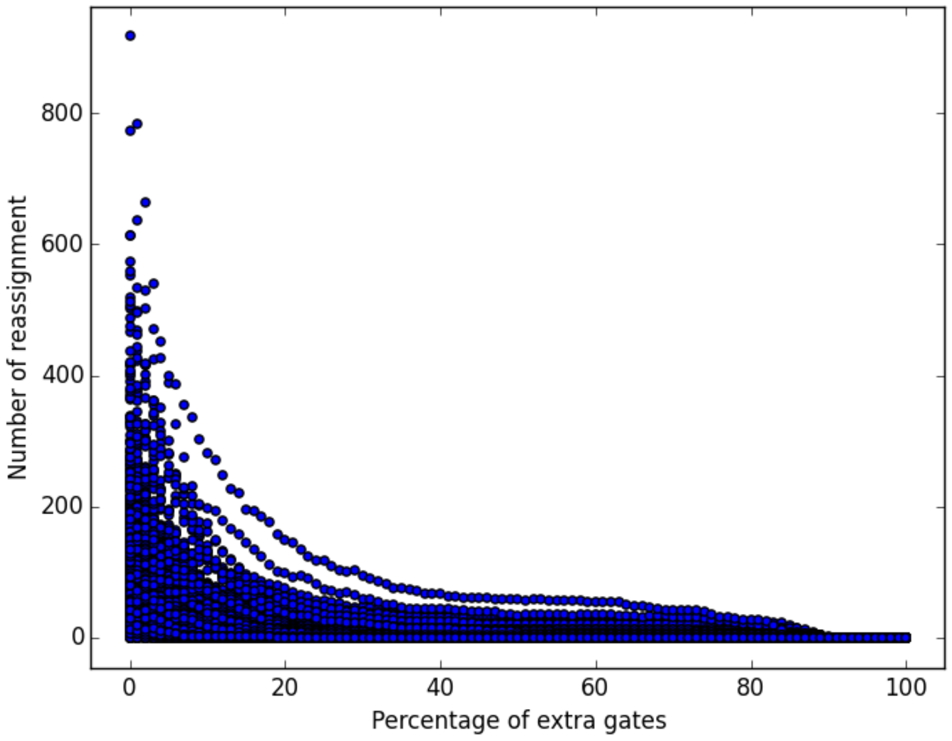


Figure 4.7: Number of gate assignments w.r.t. Percentage of extra gates

[11] Recall that when we have 1 gate per plane, then we will not have any collisions.

[12] We consider a success for an airport when we are able to reschedule all flights without employing any overflow gates.

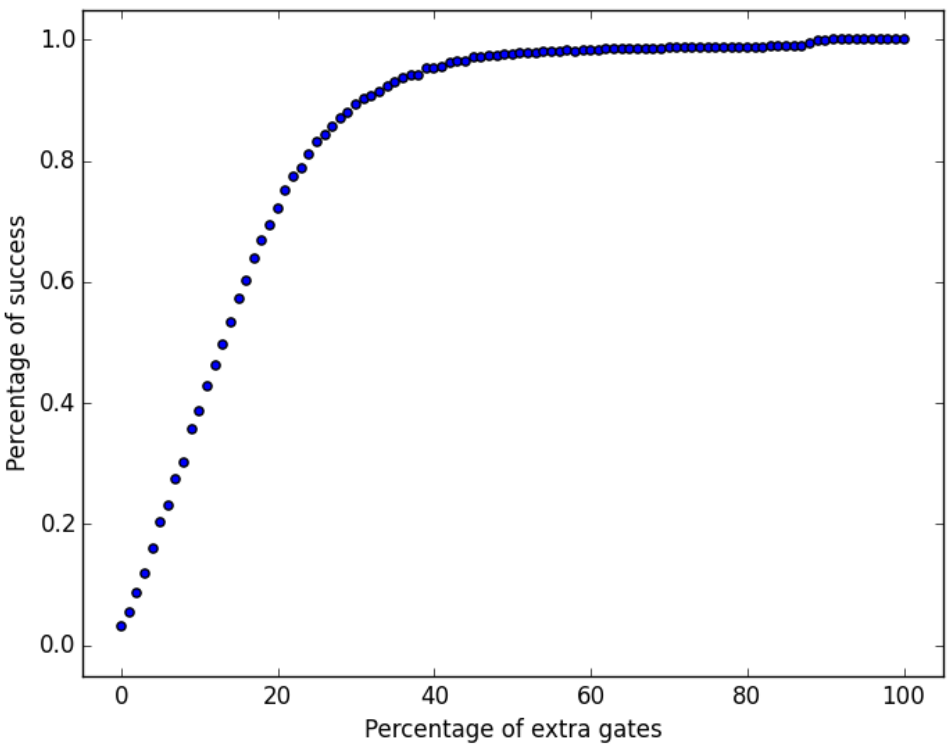


Figure 4.8: Number of extra gate assignments w.r.t. Percentage of extra gates

As expected, Figures 4.7 and 4.8 display a reciprocal shape and logarithmic shape respectively. From Figure 4.8, we can state with relative confidence that “using extra gates with at least 40% of the total number of flights, we can attain 95% success rate in dealing with delays”.

Chapter 5

Conclusion

In this project, we investigated the problem of robust airport scheduling.

To solve the vanilla airport scheduling problem, we first looked at the literature of interval scheduling and graph colouring before we settled on using the optimal interval partitioning algorithm. In this portion, we learn that by formulating the problem into different fields, we can obtain both obvious upper and lower bounds for our problem.

While we optimally solve the vanilla airport scheduling problem, things get hairy when we consider a more realistic situation where flights get delayed. For this problem, we first came up with ways to reschedule flights that cause collisions with a given original schedule. To aid us, we investigated “slack” as an auxiliary measurable variable in place of the vague notion of “delay”.

Lastly, we conducted computational experiments based on our findings and explored a few interesting scenarios using data from RITA.