DURF Report
Optimization of NYU Shanghai Shuttle Bus Schedule

Yuhan Yao yy2564@nyu.edu
Xue Bai xb437@nyu.edu

1. Introduction

Campus and corporate shuttle bus services are essential to the everyday life of many students, faculty, and workers. Take NYU Shanghai for example, many students choose shuttle buses to commute between residential halls and Pudong campus. Maintaining the shuttle bus service is also a crucial part of the school budget. As convenient as the shuttle bus service may be, there still exist unsatisfactory incidents and a waste of university resources. For example, students find themselves waiting in line to get on the last shuttle in the morning just to be informed there are no more seats; on the other hand, some shuttles may operate with only a few students and a lot of empty seats.

We believe that many problems regarding the shuttle bus service can be resolved by optimizing the shuttle bus schedule. Because the buses are leaving from an assigned depot and returning to the same depot and have fixed stops, this can be seen as a vehicle scheduling problem (VSD) [1]. Our case, where the shuttle buses need to go back and forth between residential halls and campus, is a special case of VSD. Furthermore, since our black box problem formulation cannot delineate a closed-form objective function, we resort to heuristic algorithms to solve the problem. Genetic Algorithm (GA) is the most suitable for our formulation. While research has been made on Genetic Algorithm and scheduling problems, we find them inappropriate for us. For example, D. Tan [2] uses Genetic Algorithm to obtain the frequency of the bus departure, where the bus routes are all unidirectional. For unidirectional models, each single 0/1 represents one bus where 0 means no bus departs and 1 otherwise. A series of 0/1 can thus show a daily workflow. However, for our vehicle scheduling problem, we need to not only know whether the bus is going to depart, but also whether it is going to return. Therefore, a tailored design of the GA is necessary.

In this report, we are hoping to tackle the common pains in students' life due to a suboptimal shuttle timetable. However, increasing bus frequency to shorten waiting time will cause the service cost to skyrocket. Therefore, we aim to construct a mathematical model to optimize NYU Shanghai shuttle bus timetable which balances the trade-off between fulfilling students' demand as much as possible and reducing service costs for the school. We will start with a baseline problem. The baseline model is a simplified case where there is only one residential hall. We will then move on to the extended problem, which is closer to reality. The extended model includes all three residential halls to the campus, represented by multi-node. The genetic algorithm is then applied in a suitable way to solve both problems.

2. Problem Description

Before COVID-19, NYU Shanghai had one academic building on Century Avenue and three residence halls in Jinqiao/Green Court, Jinyang, and Pusan. In the past year, because of the Go Local program, we had an extra academic building in Shinmei and all go local students have been allotted to stay in the Pusan residence hall. Given that the Go Local program only existed because of COVID-19, this DURF project focuses on the original and more generalizable situation with one academic building (AB) and three residence halls as transportation depots for NYU Shanghai students only. Our research will start with a simplified baseline problem where we only consider Jinqiao residence hall and then extend to the full picture where all three residence halls are considered.

2.1 Baseline Problem

The baseline problem isolates the shuttle buses commuting between AB and Jinqiao. Even though the bus schedule may be different during midterms, finals, and weekends, to simplify our model, we make the following assumptions:

1. All shuttle buses commute only between Jinqiao and AB, not other residential halls.
2. All shuttle buses leave from Jinqiao or wait at Jinqiao for the first dispatch, which means no shuttle bus leaves from AB during the first dispatch in the morning. It is consistent with the real-world operation.
3. Shuttle bus services start from 7:00 in the morning and end at 24:00 at night (17 hours per day) on weekdays with no schedule changes on special occasions.
4. All shuttle buses can only travel from AB to Jinqiao or travel from Jinqiao to AB once in an interval of 30 minutes. The one and only exception of the assumption above is that during rush hours, all shuttle buses require 2 intervals (1 hour) to travel to their destinations.
5. Every shuttle bus has one bus driver to operate. He or she cannot work more than a maximum working hour of 4 hours consecutively.

We set the time interval T to be 30 minutes because it is the shortest interval in the shuttle timetable before COVID-19 (See Appendix A) and it is enough for one commute according to Baidu Map data (See Appendix B). Therefore, we have 34 intervals in total for the whole operation period. However, empirically speaking, there is no guarantee that Baidu Map is always accurate, so we are going to consider the rush hour constraint in assumption 4 from 7:30 to 8:30 and from 17:30 to 18:30.

2.2 Extended Problem

In the extended problem, we take Jinyang and Pusan residence hall into consideration on top of the baseline problem. Since Jinqiao and Jinyang residence halls are only one block away from each other and shuttle buses are parked in Jinqiao parking lot, buses will leave Jinqiao first and go to Jinyang to pick up Jinyang students and finally go to AB (See Appendix C). If buses

depart from AB, they still go to Jinqiao first and then drop off the Jinyang students. Here, we highlight the assumptions different from the baseline problem:

1. All shuttle buses may commute between Jinqiao/Jinyang and AB or Pusan and AB on any day if student demand requires.
2. All shuttle buses leave from Jinqiao or Pusan or wait at Jinqiao or Pusan for the first dispatch, which means no shuttle bus leaves from AB during the first dispatch in the morning.
3. All shuttle buses can only travel from AB to Jinqiao/Jinyang or Pusan or travel from Jinqiao/Jinyang or Pusan to AB once in an interval T of 30 minutes. The one and only exception of the assumption above is that during rush hours, all shuttle buses require 2 intervals (1 hour) to travel to their destinations.

We keep the interval T to be 30 minutes because it only takes 3 to 5 minutes to drive from Jinqiao to Jinyang, so 30 minutes are more than enough for one Jinqiao-Jinyang-AB trip. Similarly, a Pusan-AB or AB-Pusan trip is also within 30 minutes according to Baidu Map. Therefore, the 30-minute interval persists and again, we have 34 intervals in total. Still, empirically speaking, there is no guarantee that Baidu Map is always accurate, so we are going to consider the rush hour constraint in assumption 6 from 7:30 to 8:30 and from 17:30 to 18:30.

3. Model Formulation

We adopted the spatio-temporal networks model, or more specifically, time expanded networks for our project. The spatio-temporal networks model is a graphic model that captures changes across time and space. It is widely used in time series analysis in computer science and is gaining increasing popularity in transportation science as vehicle scheduling problems become more important. Time expanded graphs, in particular, "can be used to represent time dependent graphs" [4]. As shown in Fig. 1, a time expanded graph splits time into several intervals and duplicates the nodes of the network for every interval. The graph is connected by "holdover edges", which connect the same node across different time periods, and "transfer edges", which connect different nodes across different time intervals. This graph allows us to track the behavior of transportations (in our case shuttle buses) for each interval and see whether they are staying at the same place or commuting to another. For example, all N1 nodes can represent Jinqiao and all N2 nodes can represent AB. An arrow pointing from N1 to N1 means that the shuttle bus stays at Jinqiao during an interval; An arrow pointing from N1 to N2 means that the shuttle bus travels from Jinqiao to AB during an interval.

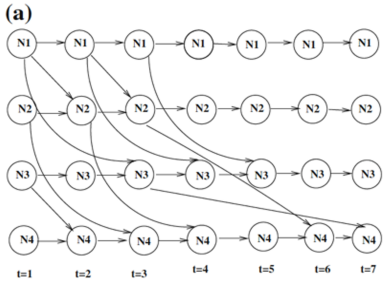


Figure 1: Example of Time Expanded Graph [4]

3.1 Baseline Problem

Starting from version 1.0, we do not consider the rush hour constraint just yet. Apart from the first dispatch, the spatio-temporal network is a repetition of the same network structure shown in Fig. 2. Every shuttle bus can travel from node n to node n+1 or node (n+1)' within one interval T = 30 minutes.

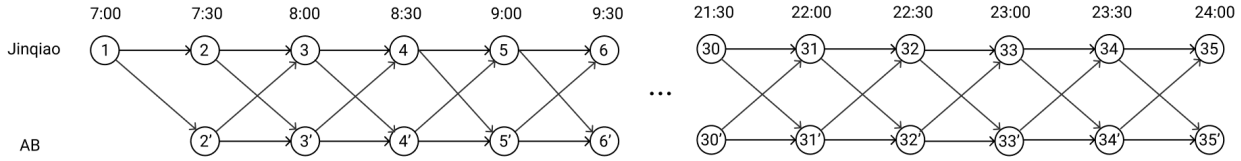


Figure 2: Baseline Problem Network Structure v1.0

Version 2.0 adds the rush hour constraint and changes the network structure from 7:30 to 8:30 and from 17:30 to 18:30, as shown in Fig. 3. An example of one of the changes is that there is no longer an arrow pointing from node 2 to node 3'. Instead, node 2 is pointing to node 4', because it takes more than 30 minutes to go from Jinqiao to AB during rush hours. Everywhere else remains the same as version 1.0.

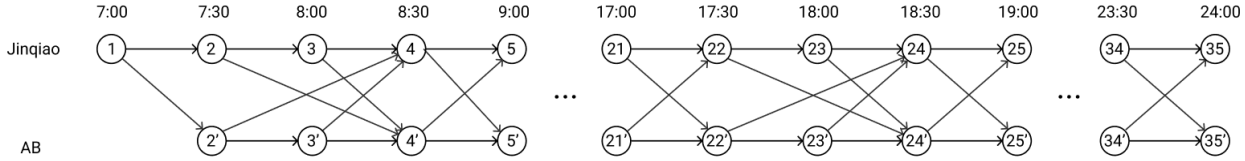


Figure 3: Baseline Problem Network Structure v2.0

3.2 Extended Problem

In the extended problem, we are going to add Jinyang and Pusan nodes to the baseline problem model formulation, as Fig. 4 shows below. Jinyang nodes are all slightly shifted to the right because it takes 3 to 5 minutes to go from Jinqiao to Jinyang, but the Jinqiao-Jinyang-AB commute is still within one interval T of 30 minutes. The Pusan nodes are very similar to Jinqiao, except that Pusan is far away from Jinqiao and Jinyang, so Pusan is treated as separate nodes.

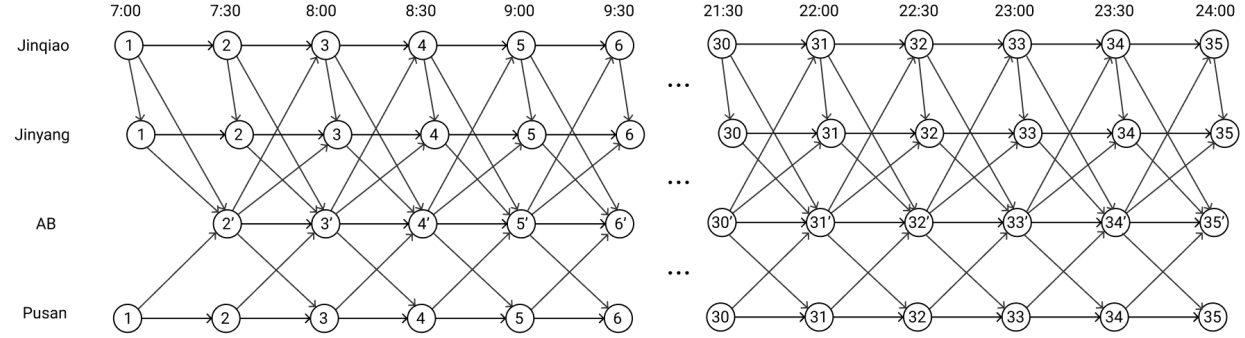


Figure 4: Extended Problem Full Network Structure

In reality, skipping Jinyang or not does not make any difference in terms of shuttle bus scheduling. Bus drivers who arrive at Jinyang first will message the other drivers whether they

need to drive to Jinyang to pick up more students. Since Jinyang has no effect on bus schedules, we can merge Jinqiao nodes and Jinyang nodes to formulate the problem in a simpler way. Version 1.0 is without the rush hour constraint, as Fig. 5 shows below.

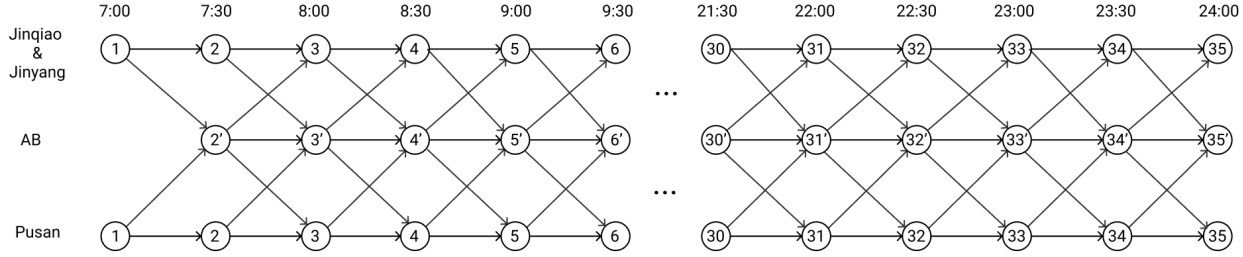


Figure 5: Extended Problem Network Structure v1.0

Similarly, version 2.0 adds the rush hour constraint and changes the network structure from 7:30 to 8:30 and from 17:30 to 18:30. The network is shown in Fig. 6.

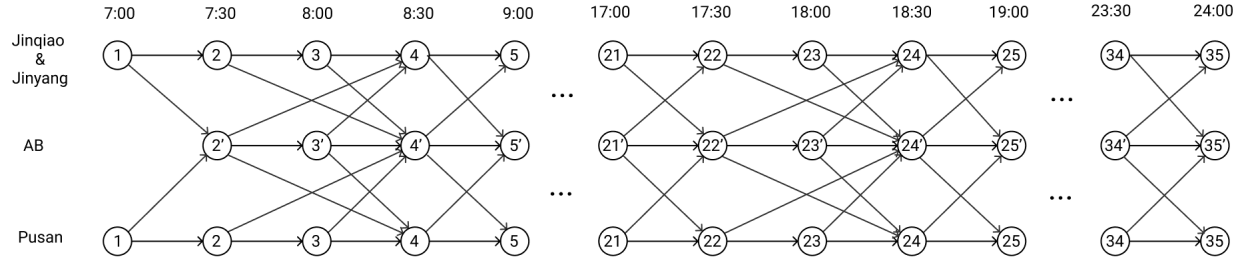


Figure 6: Extended Problem Network Structure v2.0

3.3 Black Box Formulation

Our objective is to minimize the total cost of the NYU Shanghai shuttle bus fleet. Depending on the different versions of network structure, constraints are susceptible to changes. The union of all constraints includes:

1. Demand Constraint:
   The shuttle bus fleet needs to satisfy the demand of students as much as possible with a small tolerance to spare. An example is that each bus holds 50 students, and the tolerance is 3. Even if there are 3 students waiting for a trip, the bus will not leave. The tolerance depends on how much the school values students' demands.
2. Rush Hour Constraint:
   During rush hours in the morning and in the afternoon, no bus can make a trip within an interval of 30 minutes.
3. Max Working Hour Constraint:
   Each bus is operated by one bus driver only and he or she cannot work more than 4 hours consecutively.

In an ideal case, an optimization problem has an explicit objective function that we can express in the form of $f(x)$. However, in our case, formulating the problem using an explicit form is incredibly complicated and somewhat gratuitous. The main reason is that the cost of each bus cannot be written in a closed form. According to Public Safety, the price of each bus is negotiated with the third-party shuttle bus company, because each bus operates at different hours

throughout the day [5]. There is no closed form function that dictates the relationship between operation hours and price. Therefore, we formulate the objective as a black box, meaning we do not attempt to write an explicit objective function. Similarly, there is no need to explicitly express our constraints.

4. Solution Algorithm

Given our path-based problem formulation, we use enhanced Genetic Algorithm to solve the problem. Genetic Algorithm mimics the process of natural selection. Typically, a solution can be represented in strings of 0's and 1's called chromosomes. The 0's and 1's are called genes. For example, a solution can look like 0110001110, whose meaning changes depending on the problem formulation. A population is a set of chromosomes and pairs of chromosomes can crossover and mutate to form new chromosomes or solutions in our case. Crossover means cutting two chromosomes at the same gene and switching the following sequence of genes to generate a new pair of chromosomes. The visualization is shown in Fig.7. Mutation of a gene is simply changing 0 into 1 or 1 into 0. [6]

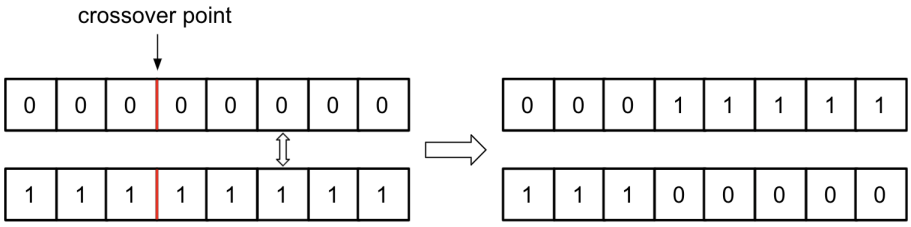


Figure 7: Genetic Algorithm Crossover

Genetic Algorithm starts with a randomized initial population of solutions, or chromosomes, that are most likely suboptimal. Then, each chromosome in the population is given a fitness score according to a fitness function. Based on these fitness scores, the elites survive and go on to the next generation. The non-elites die. Children of the first generation, very likely the children of the elites, populate the rest of the spots in the next generation. These children are produced by crossing over and mutating the genes of their parents. This process of creating new generations loops for many times until convergence or manual termination. Generally speaking, each generation is better than the previous one, so by the end of the evolution, we will have a much better solution than in the beginning. However, there is no guarantee that the best solution in the end is the optimal solution out of all. There is a chance that the algorithm will get stuck in suboptimality. [6]

The pseudocode for Genetic Algorithm is:

1. Generate initial population
2. Compute fitness scores
3. Repeat:
    a. Generate new generation by selection, crossover, and mutation
    b. Compute fitness scores
4. Stop until convergence or manual termination

Genetic Algorithm beats its counterparts in our case because of its unique advantages. Since the cost of a bus can only be determined if its path is enumerated, we tried the method of Column Generation first. Column Generation enumerates all possible paths in the solution set and uses brute force search to find the optimal solution. It is like finding a needle in a haystack by checking the hay one by one until we find the needle. However, the baseline case alone already has 2 ** 34 = 17,179,869,184 paths in total and no computer has the hardware capacity to store all this information. We can only see a random part of the haystack, with no idea if the needle is really in this part or not, so Column Generation becomes computationally infeasible. Therefore, we turned to Genetic Algorithm, a heuristic algorithm. It starts with a randomized suboptimal solution and then evolves to get better solutions that are closer to the optimal solution over time. Genetic Algorithm also does not require an explicit objective function, so it is well-suited for our black box formulation.

In order to use Genetic Algorithm, one small change needs to be applied to the problem formulation. That is, all constraints are converted into penalty and added onto the objective function to create the fitness function [6]. Genetic Algorithm may get stuck in the beginning and loop indefinitely if constraints are imposed. If the algorithm does not allow moving on to the next generation until all constraints are met, since Genetic Algorithm usually starts with randomized no-so-good solutions that do not satisfy all constraints, it may forever be stuck in the initialization stage and never move on. Therefore, converting constraints into penalty allows the algorithm to move on. It can still generate the next generation even if the previous one does not satisfy all constraints. Thanks to the penalty, the solutions that violate constraints are going to have a greater fitness score, so they are more likely to be eliminated. Therefore, over time, the ones that do not violate constraints and have a smaller fitness score survive.

4.1 Baseline Problem

For the baseline problem, one path in a solution looks like 011001010001011111000010 1110001011. It has 34 genes of 0 or 1 because there are 34 intervals in total. 0 represents a bus waiting still during an interval and does not carry any students; 1 represents a bus taking a trip either from Jinqiao to AB or from AB to Jinqiao. Please note that a gene of 1 does not necessarily mean that the bus carries students. The bus might go to Jinqiao empty in order to fulfill the huge demand at the beginning of the next interval. One solution has a couple of buses, so a solution will be several path chromosomes concatenated together. Fig.8 visualizes the terms.

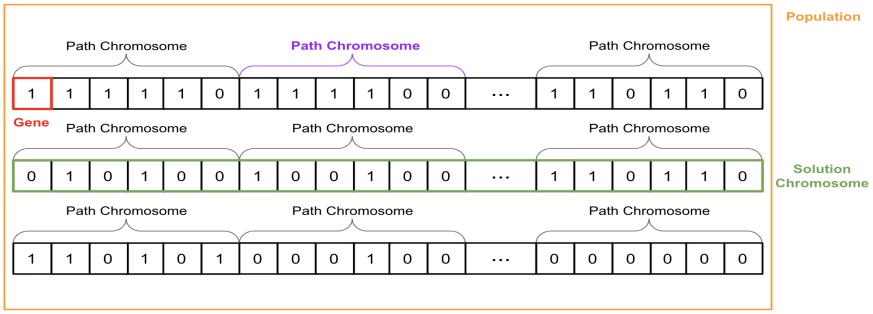


Figure 8: Visualization of GA Terms

After a solution chromosome is generated, a fitness score is calculated according to a fitness function. A fitness function has two components, a total cost and a penalty cost. For the first component, according to NYU Shanghai Public Safety, although the price of each bus needs to be negotiated with the shuttle bus company, the prices generally depend on the duration starting from the beginning to the end of bus operation. Please note that as long as a driver starts his day of work and hasn't clocked out, all intervals marked 0 still cost money even though the bus stayed still in one place. Therefore, we used the following mathematical formula:

$$path\ cost = \begin{cases} 0 & if\ x = 0 \\ 90 & if\ 0 < x \leq 5 \\ 180 & if\ 5 < x \leq 7.5 \\ 20x & if\ x > 7.5 \end{cases}$$

where x is the operation duration of a bus (unit: hours). This formula is counterintuitive in many ways, but it is by far the simplest formula to approximate the current prices of our shuttle buses. For example, the path chromosome 011001010001011111000010111000101 1 operates 33 intervals, which is 16.5 hours, so the cost for this bus is 20 * 16.5 = 330RMB. Then, the total cost is the sum of the cost of each bus. The second component of the fitness score is the penalty for violating constraints. Depending on which version of the base function you are considering, you may have one penalty where you only enforce the demand constraint, or two constraints, or all three penalties of the demand constraint, the rush hour constraint and the max working hour constraint. A different amount of small penalty is added to the fitness score every time a student does not get on to the bus, or a driver finishes a trip in one interval during rush hours, or a driver works longer than 4 hours consecutively.

Specifically, here is how constraint violations are detected. First, for the demand constraint, the chromosome paths need to be encoded to an array of size 4 * 34 which specifies what each 0 and 1 means. Fig. 9 shows an example. What helps us calculate the total bus capacity is the second and third row of the array. We can then add up all the Jinqiao to AB buses and AB to Jinqiao buses and then times the bus capacity to compare with the demand.

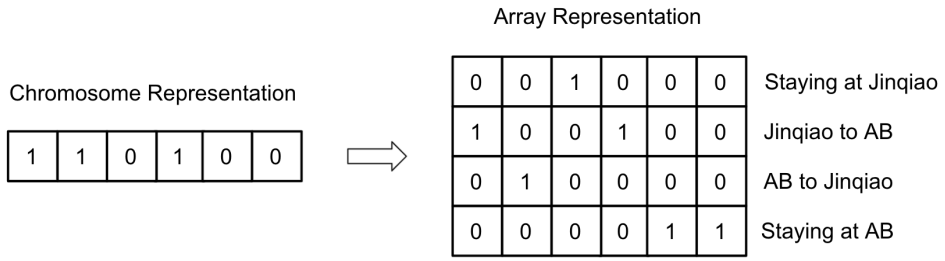


Figure 9: Example of Chromosome Encoding

Second, for the rush hour constraints, we need to check each chromosome for consecutive 1's from 7:30 to 8:30 and from 17:30 to 18:30. 10, 01 and 00 are valid during rush hours, but not 11. Third, for the max working hour constraint, we need to keep track of the number of 1's in a row. Also, in cases of rush hours, 0 can also mean that a bus is running, so we need to take care of this special case as well. If the working duration is greater than 4 hours, it causes a penalty.

Please see Appendix D for detailed Python code implementation.

4.2 Extended Problem

The extended problem is more complicated than the baseline problem in that we have three choices to go instead of two choices 0 and 1. For example, a shuttle bus at AB can choose to park at AB for an interval's time, or go to Jinqiao/Jinyang, or go to Pusan. Then two numbers 0 and 1 are not enough to represent three choices. Therefore, the genes become 00, 10 and 01 in the extended problem. A path chromosome now looks like 00101000001010001000000101101 01010100000001000000001011010101010001010. It still has 34 genes but in 68 digits. 00 means that the bus stays still in the same location; 10 means that the bus travels between AB and Jinqiao/Jinyang; 01 means that the bus travels between AB and Pusan. Similarly, a gene of 10 or 01 does not necessarily mean that the bus carries students. The bus might go to Jinqiao/Jinqiao or Pusan empty in order to fulfill the huge demand at the beginning of the next interval. The rest is the same as the baseline problem.

We consider 00 to be equivalent to 0 and 10 and 01 to be equivalent to 1, so the calculation of the cost and penalties is the same as the baseline problem. Please see Appendix E for detailed Python code implementation.

5. Numerical Example

We are going to use the NYU Shanghai shuttle bus fleet as our numerical example. We will try to incorporate as much authentic information as possible.

5.1 Parameters

The solution parameters include those for the Genetic Algorithm and those for the network formulation. Table 1 is a list of all parameters. All parameters below with multiple values are tuned and tested in our research.

Category	Parameter Name	Parameter Description	Value(s)
GA Parameters	initial_prob	When initializing GA, we want the probability of randomizing 1's to be greater than 0.5 so that the solutions can start with less penalty on the demand constraint. But this doesn't matter much because the algorithm is going to learn better in the end.	0.8
	pusan_prob	The probability of splitting shuttle buses between Jinqiao/Jinyang and Pusan	0.2
	population_size	The number of chromosomes in a population	20
	elitism_cutoff	When generating a new generation, we select a certain number of elites into the next generation.	2
	mutation_num	Number of single mutation (See 5.2)	1-5
	mutation_prob	Probability above which a gene is going	0.95

		to mutate (See 5.2)	
	evolution_depth	The number of generations to produce until termination.	3,000-50,000
	loop_limit	Some chromosomes are not feasible after crossover or mutation, so we set a loop_limit to avoid looping forever.	100
Shuttle Parameters	N	Number of shuttle buses	15-30
	D	Max number of students on a bus	50
	tolerance	If the number of students is less than or equal to the tolerance value, the bus will not travel.	0-3 but here we always use 0 until 5.4
	intervalDuration	Duration of an interval	0.5
	intervalNum	Number of intervals in total	34
	maxWorkingHour	Maximum consecutive working hours	4
	alpha	Coefficient for demand penalty. Its value depends on how much the school values students' demand.	(0, 1] but here we always use 1 until 5.4
	demandViolationPenalty	Penalty to add to the fitness score every time a student's demand is not met	10 for baseline 20 for extension
	rushHourViolationPenalty	Penalty to add to the fitness score every time the rush hour constraint is violated	7 for baseline 17 for extension
	maxWorkingHourViolationPenalty	Penalty to add to the fitness score every time the max working hour constraint is violated	5 for baseline 15 for extension
	demand	An array of the numbers of students waiting to commute at the beginning of each interval	See Table 2

Table 1: List of All Parameters

There is no existing data on how much the students' demand is, so we conducted a survey and collected valid information and opinions from 56 NYU Shanghai students (See Appendix F). We augmented the number of students based on the approximate number of residents in Jinqiao/Jinyang and Pusan and got the following results.

	7:00	7:30	8:00	8:30	9:00	9:30	10:00	10:30	11:00	11:30	12:00	12:30	13:00	13:30	14:00	14:30	15:00
Jinqiao/Jinyang to AB	114	105	132	132	117	83	57	52	13	8	18	13	26	3	13	10	0
AB to Jinqiao/Jinyang	0	0	0	0	0	0	14	2	0	7	12	7	9	5	7	7	12
	15:30	16:00	16:30	17:00	17:30	18:00	18:30	19:00	19:30	20:00	20:30	21:00	21:30	22:00	22:30	23:00	23:30
Jinqiao/Jinyang to AB	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
AB to Jinqiao/Jinyang	9	32	39	53	35	30	18	60	44	60	53	90	58	78	71	35	55

	7:00	7:30	8:00	8:30	9:00	9:30	10:00	10:30	11:00	11:30	12:00	12:30	13:00	13:30	14:00	14:30	15:00
Pusan to AB	13	12	15	15	13	9	6	6	1	1	2	1	3	0	1	1	0
AB to Pusan	0	0	0	0	0	0	2	0	0	1	1	1	1	1	1	1	1
	15:30	16:00	16:30	17:00	17:30	18:00	18:30	19:00	19:30	20:00	20:30	21:00	21:30	22:00	22:30	23:00	23:30
Pusan to AB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AB to Pusan	1	4	4	6	4	3	2	7	5	7	6	10	6	9	8	4	6

Table 2: Demand Table

There are many things to change in the Python codes, plenty of parameters to tune, and multiple versions of the problem to solve, so here we list a few interesting findings one by one to illustrate the outcomes. Please note that Genetic Algorithm uses randomization at its core, so all tests below might not be reproducible, and results are not definitive. Tests in different tables with the same index number are the same tests.

5.2 Mutation Implementation

We tested two implementations of the mutation step in the Genetic Algorithm. The first one is the conventional implementation where each gene might mutate given a probability mutation_prob. There is no guarantee if a chromosome will be mutated or how many places a chromosome will be mutated. The second one is randomly picking a gene out of a solution chromosome to mute. The mutation is guaranteed and there could only be one mutation. We tried the second implementation in an attempt to converge the algorithm faster. And test results in Table 3 show that the second one is indeed better than the conventional one in many ways. Also, the second implementation is also faster because it does not need to loop through every gene and determine if it would be mutated.

#	Problem Version	Mutation Version	evolution_depth	N	checkDemandFlag checkRushHourFlag checkMaxWorkingHourFlag	Constraints*	Final Cost**
1	Baseline	1	3,000	20	All True	r6	5,610
2		2				r2	3,770
3		1	10,000			r6	5,620
4		2				satisfied	3,490
5		1	20,000			r5	5,500

* r6 means there are 6 violations of the rush hour constraint. Similarly, d2 would mean 2 violations of the demand constraint and w4 would mean 4 violations of the max working hour constraint, there are any.
** Final costs are penalty-free. They are not fitness scores.

Table 3: Mutation Implementation Comparison Tests

As you can see in Table 3, comparing test 1&2 and 3&4, given a fixed evoluation_depth, mutation version 2 has fewer constraint violations and lower final costs. Comparing test 1&3&5, mutation version 1 does not seem to converge very well even after creating 20,000 generations. It's performance plateaus early on and is far from satisfactory compared to test 4. Therefore, our self-designed implementation of the mutation function, rather than the conventional method, performs better. And this superiority persists no matter how much one tunes the parameter mutation_prob. We suspect that it is an algorithmic advance in our case rather than a fluke.

Now, let's further investigate the self-designed mutation implementation method. The one tested above guarantees one mutation in each solution chromosome, where the parameter mutation_num equals 1. What if there are more mutations in one chromosome?

#	Problem Version	Mutation Version	mutation_num	evolution_depth	N	checkDemandFlag checkRushHourFlag checkMaxWorkingHourFlag	Constraints	Final Cost
4	Baseline	Self-Design Mutation	1	10,000	20	All True	satisfied	3,490
6			2				r2	3,490
7			3				r2	3,660
8			4				d5r1	3,470

Table 4: Self-designed Mutation Parameter Tests

Table 4 showcases that mutation_num equals 1 seems to be the best. Tuning the parameter higher results in more constraint violations or higher final cost. We will adopt the self-designed mutation implementation with mutation_num set to 1 for the rest of the report.

5.3 Different Versions

As mentioned in Section 3, there are the baseline problem and the extended problem, each having two versions of network structure given if the rush hour constraint is considered. We here test out all four different versions.

#	Problem Version	evolution_depth	N	checkDemandFlag	checkRushHourFlag	checkMaxWorkingHourFlag	Constraints	Final Cost
4	Baseline	10,000	20	True	True	True	satisfied	3,490
9				True	False	True	satisfied	3,580
10		50,000		True	True	True	satisfied	3,090
11				True	False	True	satisfied	3,230
12	Extended	10,000	25*	True	True	True	d7r10w6	5,710
13				True	False	True	d6w7	5,660
14		50,000		True	True	True	d4r9w8	5,630
15				True	False	True	d2w2	5,380

* The extended problem has more demand than the baseline problem and therefore requires more buses.

Table 5: Self-designed Mutation Parameter Tests

According to Table 5, having the rush hour constraint tends to result in a lower final cost if all constraints are satisfied, but the algorithm is more prone to violating constraints. Both problems seem to be much easier and faster to yield a constraint-satisfying solution. Since tests 12 to 15 do not satisfy constraints, the final costs are not conclusive.

Test 10 makes one wonder if for the baseline problem, we can reduce N and still yield a solution that meets all constraints.

#	Problem Version	evolution_depth	N	checkDemandFlag checkRushHourFlag checkMaxWorkingHourFlag			Constraints	Final Cost
10	Baseline	50,000	20	True	True	True	satisfied	3,090
11				True	False	True	satisfied	3,230
16			19	True	True	True	satisfied	3,040
17				True	False	True	satisfied	3,230
18			18	True	True	True	satisfied	2,950
19				True	False	True	satisfied	2,940
20			17	True	True	True	satisfied	2,900
21				True	False	True	satisfied	3,030
22			16	True	True	True	r1	2,617
23				True	False	True	satisfied	2,700
24			15	True	True	True	r1	2,630
25				True	False	True	satisfied	2,840

Table 6: Baseline Problem Feasibility Tests

The version differences mentioned above are generally consistent with results in Table 6. Here, we take a closer look at test 20, because it considers the rush hour constraint and has minimum final costs (constraint satisfied) in our test so far. Test 20 is the closest to the actual NYU Shanghai shuttle bus scenario. Fig. 10 shows the details of test 20. The best fitness score with penalties drops from 5858 to 2900; the best fitness score without penalties drops from 5620 to 2900. The improvement rate over 50,000 evolutions of the fitness scores without penalties is about 50%.

```
**********************************************************************
Progress_with_penalty of improvement: 5858.0 to 2900.0
Progress of improvement: 5620.0 to 2900.0
Improvement Rate of progress: 0.48398576512455516
**********************************************************************
```

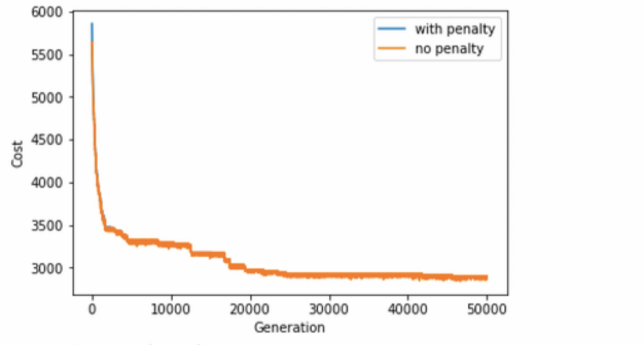


Figure 10: Detailed Results of Test 20

An interesting observation from all the tests above is that our cost function, which approximates the real-life scenario, hardly makes sense. Intuitively, the following formula

$$path\ cost = \begin{cases} 0 & if\ x = 0 \\ 90 & if\ 0 < x \leq 5 \\ 180 & if\ 5 < x \leq 7.5 \\ 20x & if\ x > 7.5 \end{cases}$$

where x is the operation duration of a bus (unit: hours)

gives us a path cost of 160 if x = 8 and yet 180 if x = 7.5. We will expect a bus to cost more money if it operates for a longer period. Another point is that two buses can actually be cheaper than one according to this formula. For example, the first two paths in Fig. 11 can easily be fit into one. That is, one bus can easily do the job of two, but the total cost of two buses is 90 * 2 = 180RMB while the cost of one bus is 20 * 33 hours= 660RMB. There could be a huge difference in the costs. Therefore, our algorithm is more prone to picking two buses instead of one. These two points are very counterintuitive, which means the cost function, or what the shuttle bus companies are proposing now, might be fundamentally flawed.

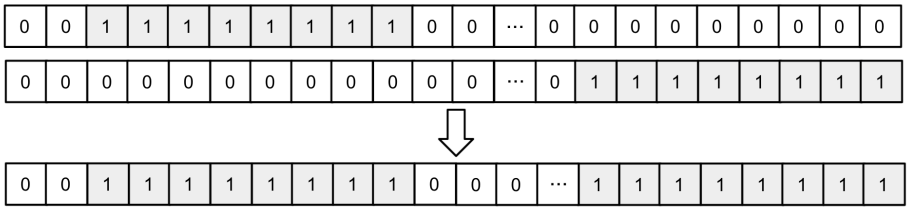


Figure 11: Example of Cost Function

The ideal cost function would enable the algorithm to start from a bigger number of buses N and then reach a best solution where a lot of paths are all 0's. That is, these buses with all 0's do not operate during the day. Therefore, we can start with a bigger N and then slowly reduce the actual number of buses needed. Theoretically, no matter how big N is, the optimal solution in the end should be the same and the actual number of buses needed should also be the same.

Similarly, we test the extended problem.

#	Problem Version	evolution_depth	N	checkDemandFlag checkRushHourFlag checkMaxWorkingHourFlag			Constraints	Final Cost
26	Extended	50,000	24	True	True	True	d1r9w3	5,550
27				True	False	True	d4w6	5,540
14			25	True	True	True	d4r9w8	5,630
15				True	False	True	d2w2	5,380
28			26	True	True	True	d3r6w3	5,470
29				True	False	True	d6w7	5,700
30				True	True	True	d4r15w5	5,620

31			27	True	False	True	d6w5	5,820
32		100,000	25	True	True	True	d3r11w3	5,440
33				True	False	True	d3w7	5,580

Table 7: Extended Problem Feasibility Tests

Unfortunately, after more tests, we still have not been able to pinpoint a feasible N. Iterating more does not seem to help convergence. We might need to try a much bigger N and a larger evolution_depth to find a feasible solution. But in Table 6, test 28 seems to yield the best solution among tests of extended problem with considering the rush hour constraint, with a relatively low final cost and fewer constraint violations. The details of the outcome are in Figure 12. The best fitness score with penalties drops from 10421 to 5677; the best fitness score without penalties drops from 8590 to 5470. The improvement rate over 50,000 evolutions of the fitness scores without penalties is about 36%.

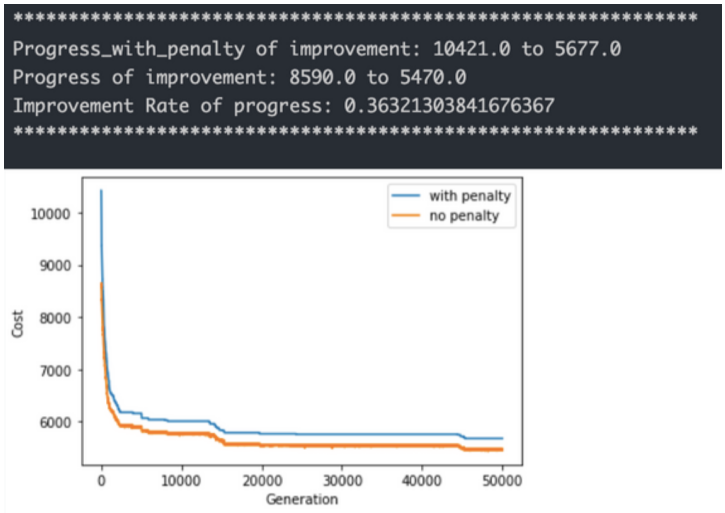


Figure 12: Detailed Results of Test 28

5.4 Sensitivity Analysis

Parameters change from time to time, so we conduct a sensitivity analysis to see if small changes in data result in the same optimal solution or not. In our case, we will adjust the parameter tolerance, which depends on how much the school values students' demand.

#	Problem Version	evolution _depth	N	checkDemandFlag checkRushHourFlag checkMaxWorkingHourFlag	Tolerance	Constraints	Final Cost
20	Baseline	50,000	17	All True	0	satisfied	2,900
34					1	r2	2,924
35					2	satisfied	3,010
36					3	r2	2,620
14					0	d4r9w8	5,630

37	Extended		25		1	r12w6	5,300
38					2	r10w2	5,600
39					3	r9w6	5,100

Table 8: Sensitivity Analysis

The test results are very unexpected. We would expect a bigger tolerance to make convergence easier and faster, but tests are pointing to an opposite direction. We actually tested the baseline problem with tolerance = 3 five times, but none of them yielded a feasible solution. This could be extremely bad luck because of randomization, or there might be a reason to investigate. And again, we have not found a feasible solution for the extended problem. After all, the entire problem is formulated as a black box.

6. Conclusion

From all tests stated above, we select the best solutions for the baseline problem with rush hour constraint and the extended problem with rush hour constraint. Test 20 yields the best solution so far for the baseline problem; Test 28 yields the best solution so far for the extended problem. Compared to the worst case, test 4, the best baseline solution improves by (3490 - 2900) / 2900 = 20.34%. Similarly, test 28 is better than test 12 by (5710 - 5470) / 5470 = 4.39%. The modified schedules are in Figure 13.

Baseline Problem			
Jinqiao to AB		AB to Jinqiao	
7:00	3		
7:30	3	7:30	2
8:00	3	8:00	1
8:30	3	8:30	4
9:00	3	9:00	3
9:30	3	9:30	2
10:00	5	10:00	6
10:30	3	10:30	1
11:00	5	11:00	4
11:30	3	11:30	2
12:00	4	12:00	5
12:30	6	12:30	2
13:00	3	13:00	4
13:30	3	13:30	5
14:00	5	14:00	3
14:30	4	14:30	6
15:00	5	15:00	3
15:30	2	15:30	4
16:00	4	16:00	1
16:30	2	16:30	3
17:00	4	17:00	3
17:30	1	17:30	1
18:00	1	18:00	1
18:30	2	18:30	2
19:00	1	19:00	4
19:30	4	19:30	1
20:00	2	20:00	2
20:30	2	20:30	2
21:00	1	21:00	2
21:30	2	21:30	2
22:00	2	22:00	2
22:30	2	22:30	2
23:00	1	23:00	1
		23:30	2

Extended Problem							
Jinqiao/Jinyang to AB		AB to Jinqiao/Jinyang		Pusan to AB		AB to Pusan	
7:00	6			7:00	3		
7:30	3	7:30	4	7:30	1	7:30	1
8:00	7	8:00	4	8:00	2		
8:30	5	8:30	7	8:30	1	8:30	1
9:00	5	9:00	6	9:00	1	9:00	1
9:30	7	9:30	2	9:30	2	9:30	1
10:00	2	10:00	8	10:00	3	10:00	2
10:30	4	10:30	7	10:30	2		
11:00	11	11:00	6	11:00	1		
11:30	6	11:30	7				
12:00	7	12:00	7	12:00	1		
12:30	6	12:30	7	12:30	1	12:30	1
13:00	8	13:00	8	13:00	1	13:00	1
13:30	6	13:30	8	13:30	3	13:30	1
14:00	9	14:00	8	14:00	1	14:00	1
14:30	9	14:30	8	14:30	1	14:30	1
15:00	4	15:00	10			15:00	1
15:30	13	15:30	2			15:30	1
16:00	4	16:00	9			16:00	1
16:30	6	16:30	3	16:30	1	16:30	1
17:00	3	17:00	7			17:00	2
17:30	5	17:30	1	17:30	1	17:30	1
18:00	2	18:00	1			18:00	1
18:30	2	18:30	7			18:30	1
19:00	6	19:00	2			19:00	1
19:30	3	19:30	4	19:30	1	19:30	1
20:00	2	20:00	2			20:00	1
20:30	4	20:30	2	20:30	1	20:30	1
21:00	4	21:00	2			21:00	1
21:30	2	21:30	2	21:30	1	21:30	1
22:00	1	22:00	2			22:00	1
22:30	2	22:30	2			22:30	1
23:00	1	23:00	1	23:00	1	23:00	1
		23:30	2			23:30	1

Figure 13: Baseline Problem Schedule

In conclusion, we formulated the NYU Shanghai shuttle bus schedule optimization problem into a path-based time-space network and implemented a modified Genetic Algorithm

to solve it. We ran many tests and devised the adjusted shuttle bus schedule from the best test results. The important factors that change the network structure are which residence halls to consider and whether or not we consider the rush hour constraint. Tolerance is one major factor that affects the sensitivity of the models. For future work, there are still several places that we can improve. As mentioned in 5.2, we can use a proper cost function to yield more reasonable results. There are also many parameters that we have not tuned. The penalties, for one, may have a significant impact on our solutions. We can try using a random search, or a grid search if time allows, to tune all the parameters and see if the results will be better. Also, due to the inherent randomization property of Genetic Algorithm, it would be advisable to run every test multiple times and check the average performance.

Acknowledgements

This research is supported by Dean's Undergraduate Research Fund and is carried out under the supervision of professor Zhibin Chen. Special thanks to Yaming Zhou from NYU Shanghai Public Safety for providing crucial shuttle information.

References

[1] Daduna J.R.. *Vehicle Scheduling. In: Floudas C., Pardalos P. (eds) Encyclopedia of Optimization.* Springer, Boston, MA, 2008.

[2] D. Tan, J. Wang, H. Liu, and X. Wang. *The optimization of bus scheduling based on genetic algorithm.* Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE), 2011.

[3]"Campus Transportation." *NYU Shanghai*, 19 Jan. 2021, https://shanghai.nyu.edu/public-safety/transportation.

[4] Betsy, George, and Sangho Kim. *Spatio-Temporal Networks Modeling and Algorithms*. Springer New York, 2013.

[5] NYU Shanghai Department of Public Safety, 2021.

[6] Sandou, Guillaume, et al. "Enhanced genetic algorithm with guarantee of feasibility for the Unit Commitment problem." *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, Berlin, Heidelberg, 2007.

Appendix A: Existing Shuttle Bus Schedule

Since we are modeling the problem before COVID-19, we referenced the Fall 2019 schedule.

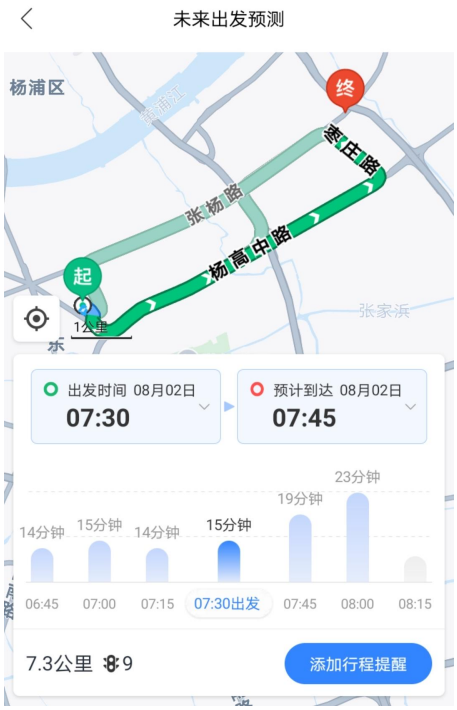
Bus Schedule for JINQIAO Residence Halls Fall 2019

Monday to Friday/周一至周五	
JQ/JY Residence to Pudong Campus 宿舍至浦东校园	Pudong Campus to JQ/JY Residence 浦东校园至宿舍
7:15	-
8:45	-
-	9:45
10:30	-
11:45	11:45
12:45	12:45
-	13:45
14:15	14:45
15:15	15:15
16:15	16:15
-	16:45
-	17:15
-	17:45
-	18:15
-	18:45
-	19:15 *
19:30	-
-	19:45
-	20:15 *
-	20:45
-	21:15 *
-	21:45
-	22:15 *
-	22:45
	23:15 *
-	23:45 *

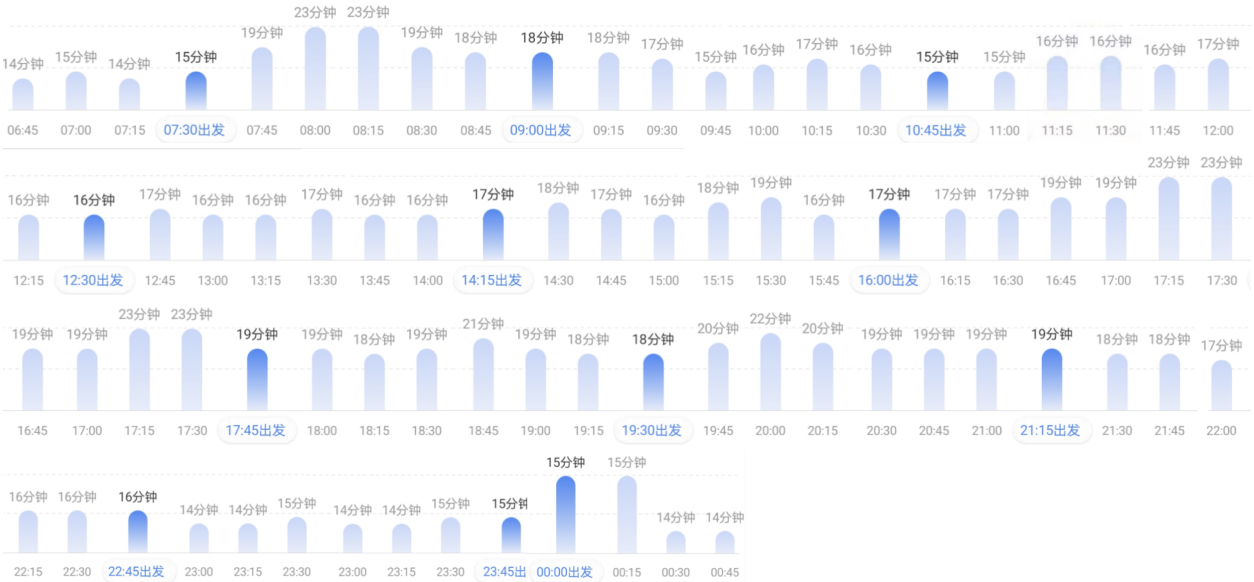
*** For Monday to Thursday only. The last bus on Friday departs on 22:45, going from Campus to Dorm**

Appendix B: Baidu Map Time Estimation

Baidu Map app provides estimated time for traveling between AB and Jinqiao. When given the departure time, it has a nice visualization of how long it is estimated to take to arrive at the destination. For example, departing from ABat 7:30am is estimated to take 15 minutes to arrive at AB. Here is a screenshot of the user interface.

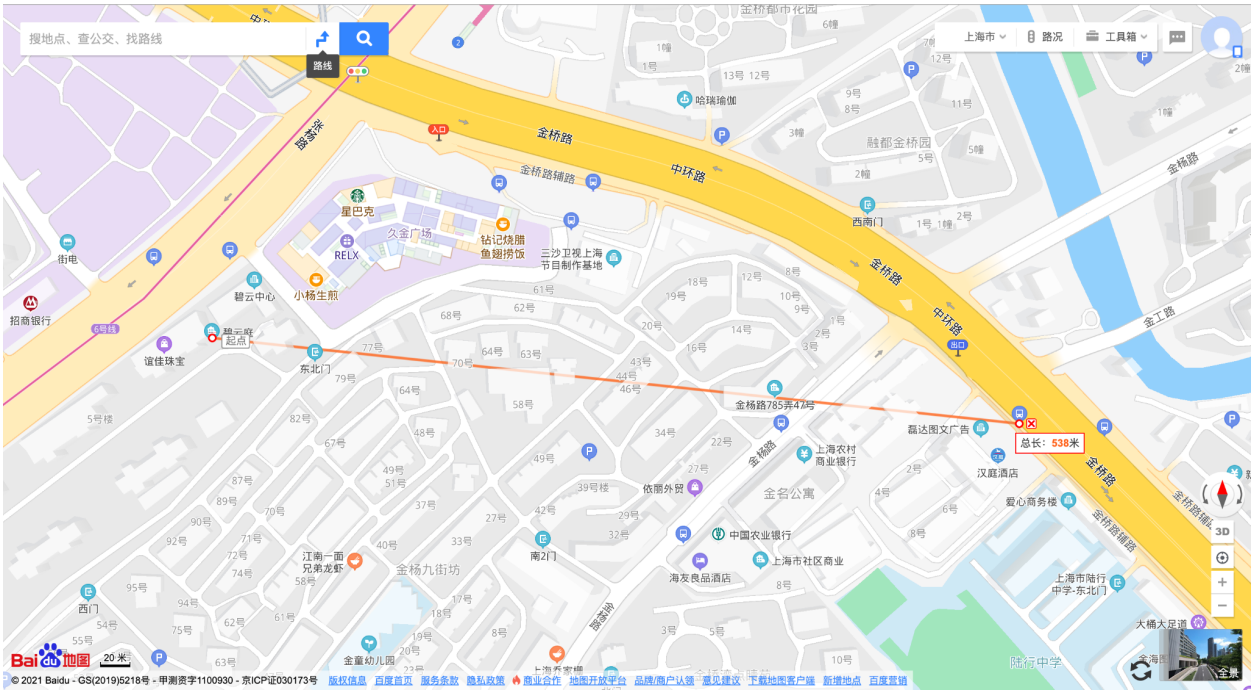


The visualization from 6:45am to 00:45am is shown below. Our time interval of interest is from 7:00am to 00:00 midnight. All estimated durations are within the 30-minute interval.



Appendix C: Jinqiao & Jinyang Map [3]

According to the Baidu Map app, Jinqiao and Jinyang pickup places are only around 500 meters apart.



Appendix D: Python Code for Baseline Problem Solution

```python
import random
import numpy as np
import time
import matplotlib.pyplot as plt


def generate_random_N_paths(N, path_length):
    '''
    Randomize N paths (1 path is like 010101010101) to generate one solution
    '''
    one_solution = []
    for _ in range(N):
        # set the weights to initialize feasible solution faster
        one_path = random.choices(population=[0, 1], weights=[1-initial_prob, initial_prob], k=path_length)
        one_solution.append(one_path)
    return np.array(one_solution)

def decode_one_path(one_path):
    decoded = []
    i, previous_node = None, None
    for j, current_node in enumerate(one_path):
        # first node
        if i == previous_node == None:
            if current_node == 0:
                decoded.append([1, 0, 0, 0])
            else:
                decoded.append([0, 1, 0, 0])
        # all nodes after first node
        else:
            previous_path = decoded[i]
            assert sum(previous_path) == 1
            if previous_path[0] == 1: # A
                if current_node == 0: # A
                    decoded.append([1, 0, 0, 0])
                else: # B
                    decoded.append([0, 1, 0, 0])
            elif previous_path[1] == 1: # B
                if current_node == 0: # D
                    decoded.append([0, 0, 0, 1])
                else: # C
                    decoded.append([0, 0, 1, 0])
            elif previous_path[2] == 1: # C
                if current_node == 0: # A
                    decoded.append([1, 0, 0, 0])
                else: # B
                    decoded.append([0, 1, 0, 0])
            else:
                if current_node == 0: # D
                    decoded.append([0, 0, 0, 1])
                else: # C
                    decoded.append([0, 0, 1, 0])
        i, previous_node = j, current_node
    return np.array(decoded).T

def demand_constraint(solution_chromosome, tolerance):
    '''
    make sure the demand is met
    '''
    # get the link representation first
    directional_N_paths = [decode_one_path(one_path) for one_path in solution_chromosome]
    link = sum(directional_N_paths)
    supplyDemandDifference = np.greater_equal(demand - tolerance, link[1:3, :] * D)
    mask = (demand - tolerance) - (link[1:3, :] * D)
    missedDemandNum = np.sum(supplyDemandDifference * mask)
    return int(missedDemandNum) == 0, int(missedDemandNum)

def rush_hour_constraint(solution_chromosome):
    '''
    during rush hours, one interval is not enough time to commute
    '''
    violationCount = 0
    for one_path in solution_chromosome:
        # morning rush hour
        if one_path[1] + one_path[2] == 2:
            violationCount += 1
        # evening rush hour
        if one_path[21] + one_path[22] == 2:
            violationCount += 1
    return int(violationCount) == 0, int(violationCount)

```

```python
def max_working_hour_constraint(solution_chromosome):
    '''
    make sure that no driver works more than a few hours continuously
    '''
    violationCount = 0
    for one_path in solution_chromosome:
        num, num_list = 0, []
        one_path_copy = one_path.copy()
        # first check if rush hour 10 or 01 actually is 11
        if checkRushHourFlag:
            if one_path_copy[1] == 1 and one_path_copy[2] == 0:
                one_path_copy[2] = 1
            if one_path_copy[21] == 1 and one_path_copy[22] == 0:
                one_path_copy[22] = 1
        for i, node in enumerate(one_path_copy):
            num += node
            if i+1 == len(one_path_copy):
                num_list.append(num)
                continue
            if node == 1 and one_path_copy[i+1] == 0:
                num_list.append(num)
                num = 0
        violationCount += sum(np.array(num_list) > maxWorkingHour / intervalDuration)
    return int(violationCount) == 0, int(violationCount)

def check_feasibility(solution_chromosome, checkDemand=True, checkRushHour=False, checkMaxWorkingHour=False):
    '''
    s.t. constraints (make sure initial paths & crossover paths & mutated paths are feasible)
    constraint1: meet demand
    constraint2: during rush hours, one interval is not enough time to commute (optional)
    constraint3: make sure that no driver works more than a few hours continuously
    '''
    demandFlag, rushHour, maxWorkingHour = True, True, True
    if checkDemand:
        demandFlag, demandViolationNum = demand_constraint(solution_chromosome, tolerance)
    if checkRushHour:
        rushHour, rushHourViolationNum = rush_hour_constraint(solution_chromosome)
    if checkMaxWorkingHour:
        maxWorkingHour, maxWorkingHourViolationNum = max_working_hour_constraint(solution_chromosome)
    if not demandFlag:
        print("d"+str(demandViolationNum), end="")
    if not rushHour:
        print("r"+str(rushHourViolationNum), end="")
    if not maxWorkingHour:
        print("w"+str(maxWorkingHourViolationNum), end="")
    return demandFlag and rushHour and maxWorkingHour

def fitness(solution_chromosome, addPenalty=False):
    """
    objective function ish -> natural selection to pick the good ones
    the lower the better!!
    """
    total_cost = 0
    # basic cost
    for one_path in solution_chromosome:
        target_indices = np.where(one_path == 1)[0]
        if len(target_indices) == 0:
            duration_interval_num = 0
        else:
            duration_interval_num = int(target_indices[-1] - target_indices[0] + 1)
        if duration_interval_num == 0:
            total_cost += 0
        elif duration_interval_num * intervalDuration <= 5:
            total_cost += 90
        elif duration_interval_num * intervalDuration <= 7.5:
            total_cost += 180
        else:
            total_cost += (20 * intervalDuration) * duration_interval_num
    # add penalty
    if addPenalty:
        demandFlag, demandViolationNum = demand_constraint(solution_chromosome, tolerance)
        rushHour, rushHourViolatonNum = rush_hour_constraint(solution_chromosome)
        maxWorkingHour, maxWorkingHourViolationNum = max_working_hour_constraint(solution_chromosome)
        if checkDemandFlag:
            total_cost += alpha * demandViolationNum * demandViolationPenalty
        if checkRushHourFlag:
            total_cost += rushHourViolatonNum * rushHourViolationPenalty
        if maxWorkingHourViolationPenalty:
            total_cost += maxWorkingHourViolationNum * maxWorkingHourViolationPenalty
    return total_cost

def generate_population(population_size):
    population, fitness_scores_add_penalty = [], []
    for _ in range(population_size):
        solution_chromosome = generate_random_N_paths(N, intervalNum)
        population.append(solution_chromosome)
        fitness_score_add_penalty = fitness(solution_chromosome, addPenalty=True)
        fitness_scores_add_penalty.append(fitness_score_add_penalty)
    return np.array(population), np.array(fitness_scores_add_penalty)

def elitism(population, fitness_scores, elitism_cutoff=2):
    elite_indices = np.argpartition(np.array(fitness_scores), elitism_cutoff)[:elitism_cutoff]
    return population[elite_indices, :]

```

```python
def create_next_generation(population, fitness_scores, population_size, elitism_cutoff):
    """
    Randomly pick the good ones and cross them over
    """
    children = []
    while True:
        parents = random.choices(
            population=population,
            weights=[(max(fitness_scores) - score + 1)/(max(fitness_scores) * len(fitness_scores) - sum(fitness_scores) + len(fitness_scores)) for score in fitness_scores],
            k=2
        )
        kid1, kid2 = single_point_crossover(parents[0], parents[1])
        for _ in range(mutation_num):
            kid1 = mutation(kid1)
        children.append(kid1)
        if len(children) == population_size - elitism_cutoff:
            return np.array(children)
        for _ in range(mutation_num):
            kid2 = mutation(kid2)
        children.append(kid2)
        if len(children) == population_size - elitism_cutoff:
            return np.array(children)


def single_point_crossover(parent1, parent2):
    """
    Randomly pick the good ones and cross them over
    The crossover point is ideally NOT going to disrupt a path.
    """
    assert parent1.size == parent2.size
    length = len(parent1)
    if length < 2:
        return parent1, parent2
    cut = random.randint(1, length - 1)
    kid1 = np.append(parent1[0:cut, :], parent2[cut:, :]).reshape((N, intervalNum))
    kid2 = np.append(parent2[0:cut, :], parent1[cut:, :]).reshape((N, intervalNum))
    return kid1, kid2

def mutation(solution_chromosome):
    """
    Mutate only one node in one path for now
    """
    # case 1: concentional mutation implementation
    if mutationType == 'Conv':
        path_num, node_num = solution_chromosome.shape
        for k in range(path_num):
            for i in range(node_num):
                solution_chromosome[k, i] = solution_chromosome[k, i] if random.random() > mutation_prob else abs(solution_chromosome[k, i] - 1)
    # case 2: self-designed mutation implementation
    else:
        mutate_path = np.random.randint(0, N)
        mutate_node = np.random.randint(0, intervalNum)
        solution_chromosome[mutate_path][mutate_node] = abs(1 - solution_chromosome[mutate_path][mutate_node])
    return solution_chromosome

def result_stats(progress_with_penalty, progress):
    """
    print important stats & visulize progress_with_penalty
    """
    print('************************************************************')
    print(f"Progress_with_penalty of improvement: {progress_with_penalty[0]} to {progress_with_penalty[-1]}" )
    print(f"Progress of improvement: {progress[0]} to {progress[-1]}")
    print("Improvement Rate of progress:", abs(progress[-1] - progress[0])/progress[0])
    print('************************************************************')
    plt.plot(progress_with_penalty, data=progress_with_penalty, label='with penalty')
    plt.plot(progress, data=progress, label='no penalty')
    plt.xlabel("Generation")
    plt.ylabel("Cost")
    plt.legend()
    plt.show()

```

```python
def run_evolution(population_size, evolution_depth, elitism_cutoff):
    '''
    Main function of Genetic Algorithm
    '''
    tic = time.time()

    # first initialize a population
    population, population_fitnesses_add_penalty = generate_population(population_size)
    initialization_end = time.time()
    print('\nInitialization Done!', initialization_end - tic)
    population_fitnesses = [fitness(solution_chromosome) for solution_chromosome in population]
    print(f'Initial Min Cost: {min(population_fitnesses_add_penalty)} -> {min(population_fitnesses)}')
    # keep track of improvement
    progress_with_penalty, progress = [], []

    # start evolving :)
    for i in range(evolution_depth):
        progress_with_penalty.append(min(population_fitnesses_add_penalty))
        progress.append(min(population_fitnesses))
        print(f'----------------------------- generation {i + 1} Start! -----------------------------')
        elitism_begin = time.time()
        elites = elitism(population, population_fitnesses_add_penalty, elitism_cutoff)
        print('Elites selected!')
        children = create_next_generation(population, population_fitnesses_add_penalty, population_size, elitism_cutoff)
        print('Children created!')
        population = np.concatenate([elites, children])
        population_fitnesses_add_penalty = [fitness(solution_chromosome, addPenalty=True) for solution_chromosome in population]
        population_fitnesses = [fitness(solution_chromosome) for solution_chromosome in population]

        evol_end = time.time()
        print(f"Min Cost: {min(population_fitnesses_add_penalty)} -> {min(population_fitnesses)}")
        # check best solution feasibility
        minIndex = population_fitnesses_add_penalty.index(min(population_fitnesses_add_penalty))
        best_solution = population[minIndex]
        allFeasibilityFlag = check_feasibility(best_solution, checkDemand=checkDemandFlag, checkRushHour=checkRushHourFlag, checkMaxWorkingHour=checkMaxWorkingHourFlag)
        print("\nAll constraints met?", allFeasibilityFlag)

        # print best solution
        print('best solution (path):\n', best_solution)
        directional_N_paths = [decode_one_path(one_path) for one_path in population[minIndex]]
        link = sum(directional_N_paths)
        print('best solution (link): \n', link)

        print(f'---------------------- generation {i + 1} evolved! Time: {evol_end - elitism_begin:.4f}s ----------------------\n')

    # plot results
    result_stats(progress_with_penalty, progress)

    # print best solution
    minIndex = population_fitnesses_add_penalty.index(min(population_fitnesses_add_penalty))
    best_solution = population[minIndex]
    print('best solution (path):\n', best_solution)

    # check if all constraints are met (ideally True)
    print("\nAll constraints met?", check_feasibility(best_solution, checkDemand=checkDemandFlag, checkRushHour=checkRushHourFlag, checkMaxWorkingHour=checkMaxWorkingHourFlag))
    directional_N_paths = [decode_one_path(one_path) for one_path in population[minIndex]]
    link = sum(directional_N_paths)
    print('best solution (link): \n', link)


if __name__ == "__main__":

    """initialization for genetic algo"""
    initial_prob = 0.8
    population_size = 20
    elitism_cutoff = 2
    mutationType = 'New' # Conv
    mutation_prob = 0.95
    mutation_num = 1 if mutationType == 'Conv' else 1
    evolution_depth = 50000

    """initialization for buses"""
    # number of buses
    N = 16
    # number of seats on each bus
    D = 50
    tolerance = 0
    intervalDuration = 0.5
    # numerical example
    demand = np.array([
        [114,106,132,132,117,83,57,52,13,8,18,13,26,3,13,10,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,14,2,0,7,12,7,9,5,7,7,12,9,32,39,53,35,30,18,60,44,60,53,90,58,78,71,35,55]
    ])

    intervalNum = demand.shape[-1]
    maxWorkingHour = 4
    checkDemandFlag, checkRushHourFlag, checkMaxWorkingHourFlag = True, True, True
    alpha, demandViolationPenalty, rushHourViolationPenalty, maxWorkingHourViolationPenalty = 1, 10, 7, 5

    # run main function
    run_evolution(population_size, evolution_depth, elitism_cutoff)
```

Appendix E: Python Code for Extended Problem Solution

```python
import random
import numpy as np
import time
import matplotlib.pyplot as plt


def generate_random_N_paths(N, path_length):
    '''
    Randomize N paths where 1 path is like 00 01 00 01 01 01
    '''
    one_solution = []
    while len(one_solution) < N:
        one_path_single_digit = random.choices(population=[0, 1], weights=[1-initial_prob, initial_prob], k=path_length)
        one_path_double_digit = ''
        for i in one_path_single_digit:
            if i == 0:
                one_path_double_digit += '00'
            elif i == 1:
                one_path_double_digit += random.choices(population=['10', '01'], weights=[1-pusan_prob, pusan_prob])[0]
        if check_path_integrity(one_path_double_digit):
            one_solution.append(one_path_double_digit)
    return one_solution

def check_solution_integrity(solution):
    for one_path_double_digit in solution:
        if not check_path_integrity(one_path_double_digit):
            return False
    return True

def check_path_integrity(one_path_double_digit):
    last_visited = None
    for i in range(len(one_path_double_digit)):
        if i % 2 == 0:
            two_digits = one_path_double_digit[i:i+2]
            if two_digits != '00':
                # first time going to AB
                if last_visited is None:
                    last_visited = 'AB'
                # following times
                elif last_visited == 'JQJY':
                    if two_digits == '01':
                        return False
                    else: # '10'
                        last_visited = 'AB'
                elif last_visited == 'PS':
                    if two_digits == '10':
                        return False
                    else: # '01'
                        last_visited = 'AB'
                else:
                    if two_digits == '10':
                        last_visited = 'JQJY'
                    else: # '01'
                        last_visited = 'PS'
    return True

def decode_one_path(one_path_double_digit):
    decoded, initial_node, last_visited = [], None, None
    for i in range(len(one_path_double_digit)):
        if i % 2 == 0:
            two_digits = one_path_double_digit[i:i+2]
            if two_digits == '00':
                if last_visited is None:
                    decoded.append([0, 0, 0, 0, 0, 0, 0])
                elif last_visited == 'JQJY':
                    decoded.append([1, 0, 0, 0, 0, 0, 0])
                elif last_visited == 'AB':
                    decoded.append([0, 0, 0, 1, 0, 0, 0])
                else: # PS
                    decoded.append([0, 0, 0, 0, 0, 0, 1])
            elif two_digits == '10':
                if last_visited is None:
                    initial_node = 0
                    last_visited = 'AB'
                    decoded.append([0, 1, 0, 0, 0, 0, 0])
                elif last_visited == 'AB':
                    last_visited = 'JQJY'
                    decoded.append([0, 0, 1, 0, 0, 0, 0])
                elif last_visited == 'JQJY':
                    last_visited = 'AB'
                    decoded.append([0, 1, 0, 0, 0, 0, 0])
                else:
                    print('SOMETHING IS WRONG1!!!')
            elif two_digits == '01':
                if last_visited is None:
                    initial_node = -1
                    last_visited = 'AB'
                    decoded.append([0, 0, 0, 0, 0, 1, 0])
                elif last_visited == 'AB':
                    last_visited = 'PS'
                    decoded.append([0, 0, 0, 0, 1, 0, 0])
                elif last_visited == 'PS':
                    last_visited = 'AB'
                    decoded.append([0, 0, 0, 0, 0, 1, 0])
                else:
                    print('SOMETHING IS WRONG2!!!')
    decoded = np.array(decoded).T
    decoded_sum = decoded.sum(axis=0)
    if sum(decoded_sum) == 0:
        if random.random() <= pusan_prob:
            decoded[0, :] = 0
        else:
            decoded[0, :] = 1
        return decoded
    k = 0
    while decoded_sum[k] == 0:
        decoded[initial_node, k] = 1
        k += 1
    return decoded

```

```python
def demand_constraint(binary_N_paths, tolerance):
    '''
    make sure the demand is met
    '''
    directional_N_paths = [decode_one_path(one_path) for one_path in binary_N_paths]
    link = sum(directional_N_paths)
    link_JQJY = link[:4, :]
    link_PS = link[-1:2:-1, :]
    JQJY_supply_demand_difference = np.greater_equal(demand_JQJY - tolerance, link_JQJY[1:3, :] * D)
    JQJY_mask = (demand_JQJY - tolerance) - (link_JQJY[1:3, :] * D)
    PS_supply_demand_difference = np.greater_equal(demand_PS - tolerance, link_PS[1:3, :] * D)
    PS_mask = (demand_PS - tolerance) - (link_PS[1:3, :] * D)
    missedDemandNumJQJY = np.sum(JQJY_supply_demand_difference * JQJY_mask)
    missedDemandNumPS = np.sum(PS_supply_demand_difference * PS_mask)
    return int(missedDemandNumJQJY + missedDemandNumPS) == 0, int(missedDemandNumJQJY + missedDemandNumPS)

def rush_hour_constraint(binary_N_paths):
    '''
    during rush hours, one interval is not enough time to commute
    '''
    violationCount = 0
    for one_path_double_digit in binary_N_paths:
        one_path_single_digit_list = []
        one_path_double_digit_list = list(one_path_double_digit)
        for i in range(len(one_path_double_digit_list)):
            if i % 2 == 0:
                one_path_single_digit_list.append(int(one_path_double_digit_list[i]) + int(one_path_double_digit_list[i+1]))
        # morning rush hour
        if one_path_single_digit_list[1] + one_path_single_digit_list[2] == 2:
            violationCount += 1
        # evening rush hour
        if one_path_single_digit_list[21] + one_path_single_digit_list[22] == 2:
            violationCount += 1
    return int(violationCount) == 0, int(violationCount)

def max_working_hour_constraint(binary_N_paths):
    '''
    make sure that no driver works more than a few hours continuously
    '''
    violationCount = 0
    for one_path_double_digit in binary_N_paths:
        one_path_single_digit_list = []
        one_path_double_digit_list = list(one_path_double_digit)
        for i in range(len(one_path_double_digit_list)):
            if i % 2 == 0:
                one_path_single_digit_list.append(int(one_path_double_digit_list[i]) + int(one_path_double_digit_list[i+1]))
        num, num_list = 0, []
        one_path_copy = one_path_single_digit_list.copy()
        # first check if rush hour 10 actually is 11.
        if checkRushHourFlag:
            if one_path_copy[1] == 1 and one_path_copy[2] == 0:
                one_path_copy[2] = 1
            if one_path_copy[21] == 1 and one_path_copy[22] == 0:
                one_path_copy[22] = 1
        for i, node in enumerate(one_path_copy):
            num += node
            if i+1 == len(one_path_copy):
                num_list.append(num)
                continue
            if node == 1 and one_path_copy[i+1] == 0:
                num_list.append(num)
                num = 0
        violationCount += sum(np.array(num_list) > maxWorkingHour / intervalDuration)
    return int(violationCount) == 0, int(violationCount)

def check_feasibility(binary_N_paths, checkDemand=True, checkRushHour=False, checkMaxWorkingHour=False):
    '''
    s.t. constraints (make sure initial paths & crossover paths & mutated paths are feasible)
    constraint1: meet demand
    constraint2: during rush hours, one interval is not enough time to commute (optional)
    constraint3: make sure that no driver works more than a few hours continuously
    '''
    demandFlag, rushHour, maxWorkingHour = True, True, True
    if checkDemand:
        demandFlag, demandViolationNum = demand_constraint(binary_N_paths, tolerance)
    if checkRushHour:
        rushHour, rushHourViolationNum = rush_hour_constraint(binary_N_paths)
    if checkMaxWorkingHour:
        maxWorkingHour, maxWorkingHourViolationNum = max_working_hour_constraint(binary_N_paths)
    if not demandFlag:
        print("d"+str(demandViolationNum), end="")
    if not rushHour:
        print("r"+str(rushHourViolationNum), end="")
    if not maxWorkingHour:
        print("w"+str(maxWorkingHourViolationNum), end="")
    return demandFlag and rushHour and maxWorkingHour

```

```python
def fitness(binary_N_paths, addPenalty=False):
    """
    objective function ish -> natural selection to pick the good ones
    the lower the better!!
    """
    total_cost = 0
    # basic cost
    for one_path_double_digit in binary_N_paths:
        one_path_single_digit_list = []
        one_path_double_digit_list = list(one_path_double_digit)
        for i in range(len(one_path_double_digit_list)):
            if i % 2 == 0:
                one_path_single_digit_list.append(int(one_path_double_digit_list[i]) + int(one_path_double_digit_list[i+1]))
        one_path_single_digit_np = np.array(one_path_single_digit_list)
        target_indices = np.where(one_path_single_digit_np == 1)[0]
        if len(target_indices) == 0:
            duration_interval_num = 0
        else:
            duration_interval_num = int(target_indices[-1] - target_indices[0] + 1)
        if duration_interval_num == 0:
            total_cost += 0
        elif duration_interval_num * intervalDuration <= 5:
            total_cost += 90
        elif duration_interval_num * intervalDuration <= 7.5:
            total_cost += 180
        else:
            total_cost += (20 * intervalDuration) * duration_interval_num
    # add penalty
    if addPenalty:
        demandFlag, demandViolationNum = demand_constraint(binary_N_paths, tolerance)
        rushHour, rushHourViolatonNum = rush_hour_constraint(binary_N_paths)
        maxWorkingHour, maxWorkingHourViolationNum = max_working_hour_constraint(binary_N_paths)
        if checkDemandFlag:
            total_cost += alpha * demandViolationNum * demandViolationPenalty
        if checkRushHourFlag:
            total_cost += rushHourViolatonNum * rushHourViolationPenalty
        if maxWorkingHourViolationPenalty:
            total_cost += maxWorkingHourViolationNum * maxWorkingHourViolationPenalty
    return total_cost

def generate_population(population_size):
    population, fitness_scores_add_penalty = [], []
    for _ in range(population_size):
        binary_N_paths = generate_random_N_paths(N, intervalNum)
        population.append(binary_N_paths)
        fitness_score_add_penalty = fitness(binary_N_paths, addPenalty=True)
        fitness_scores_add_penalty.append(fitness_score_add_penalty)
    return np.array(population), np.array(fitness_scores_add_penalty)

def elitism(population, fitness_scores, elitism_cutoff=2):
    elite_indices = np.argpartition(np.array(fitness_scores), elitism_cutoff)[:elitism_cutoff]
    return population[elite_indices, :]

def create_next_generation(population, population_fitnesses_add_penalty, population_size, elitism_cutoff):
    """
    Randomly pick the good ones and cross them over
    """
    children = []
    while True:
        parents = random.choices(
            population=population,
            weights=[(max(population_fitnesses_add_penalty) - score + 1)/(max(population_fitnesses_add_penalty) * len(population_fitnesses_add_penalty) - sum(population_fitnesses_add_penalty) + len(population_fitnesses_add_penalty)) for score in population_fitnesses_add_penalty],
            k=2
        )
        kid1, kid2 = single_point_crossover(parents[0], parents[1])
        for _ in range(mutation_num):
            kid1 = single_mutation(kid1)
        children.append(kid1)
        if len(children) == population_size - elitism_cutoff:
            return np.array(children)
        for _ in range(mutation_num):
            kid2 = single_mutation(kid2)
        children.append(kid2)
        if len(children) == population_size - elitism_cutoff:
            return np.array(children)
```

```python
def single_point_crossover(parent1, parent2):
    """
    Randomly pick the good ones and cross them over
    """
    assert parent1.size == parent2.size
    length = len(parent1)
    if length < 2:
        return parent1, parent2
    count = 0
    while count <= loop_limit:
        cut = random.randint(1, length - 1) * 2
        kid1 = np.array(list(parent1)[:cut] + list(parent2)[cut:])
        kid2 = np.array(list(parent2)[:cut] + list(parent1)[cut:])
        if check_solution_integrity(kid1) and check_solution_integrity(kid2):
            return kid1, kid2
        elif check_solution_integrity(kid1) and not check_solution_integrity(kid2):
            return kid1, None
        elif not check_solution_integrity(kid1) and check_solution_integrity(kid2):
            return None, kid2
        count += 1
    return parent1, parent2

def single_mutation(binary_N_paths):
    """
    Mutate only one node in one path for now
    """
    count = 0
    binary_N_paths_copy = binary_N_paths.copy()
    while count <= loop_limit:
        mutate_path = np.random.randint(0, N)
        mutate_index = np.random.randint(0, intervalNum) * 2
        double_digits_to_mutate = binary_N_paths_copy[mutate_path][mutate_index:mutate_index+2]
        pool = ['00', '01', '10']
        pool.remove(double_digits_to_mutate)
        mutated_double_digits = random.choices(population=pool)[0]
        original_string = binary_N_paths_copy[mutate_path]
        mutated_string = original_string[:mutate_index] + mutated_double_digits + original_string[mutate_index+2:]
        if check_path_integrity(mutated_string):
            binary_N_paths_copy[mutate_path] = mutated_string
            return binary_N_paths_copy
        count += 1
    return binary_N_paths
```

```python
def result_stats(progress_with_penalty, progress):
    """
    print important stats & visulize progress_with_penalty
    """
    print('************************************************************')
    print(f"Progress_with_penalty of improvement: {progress_with_penalty[0]} to {progress_with_penalty[-1]}" )
    print(f"Progress of improvement: {progress[0]} to {progress[-1]}")
    print("Improvement Rate of progress:", abs(progress[-1] - progress[0])/progress[0])
    print('************************************************************')
    plt.plot(progress_with_penalty, data=progress_with_penalty, label='with penalty')
    plt.plot(progress, data=progress, label='no penalty')
    plt.xlabel("Generation")
    plt.ylabel("Cost")
    plt.legend()
    plt.show()

def run_evolution(population_size, evolution_depth, elitism_cutoff):
    '''
    Main function of Genetic Algorithm
    '''
    tic = time.time()

    # first initialize a population
    population, population_fitnesses_add_penalty = generate_population(population_size)
    initialization_end = time.time()
    print(f'\nInitialization Done! Time: {initialization_end - tic:.6f}s')
    population_fitnesses = [fitness(binary_N_paths) for binary_N_paths in population]
    print(f'Initial Min Cost: {min(population_fitnesses_add_penalty)} -> {min(population_fitnesses)}')
    # keep track of improvement
    progress_with_penalty, progress = [], []

    # start evolving :)
    for i in range(evolution_depth):
        progress_with_penalty.append(min(population_fitnesses_add_penalty))
        progress.append(min(population_fitnesses))
        print(f'------------------------------ generation {i + 1} Start! ------------------------------')
        elitism_begin = time.time()
        elites = elitism(population, population_fitnesses_add_penalty, elitism_cutoff)
        print('Elites selected!')
        children = create_next_generation(population, population_fitnesses_add_penalty, population_size, elitism_cutoff)
        print('Children created!')
        population = np.concatenate([elites, children])
        population_fitnesses_add_penalty = [fitness(binary_N_paths, addPenalty=True) for binary_N_paths in population]
        population_fitnesses = [fitness(binary_N_paths) for binary_N_paths in population]

        evol_end = time.time()
        print(f"Min Cost: {min(population_fitnesses_add_penalty)} -> {min(population_fitnesses)}")
        # check best solution feasibility
        minIndex = population_fitnesses_add_penalty.index(min(population_fitnesses_add_penalty))
        best_solution = population[minIndex]
        allFeasibilityFlag = check_feasibility(best_solution, checkRushHour=checkRushHourFlag, checkMaxWorkingHour=checkMaxWorkingHourFlag)
        print("\nAll constraints met?", allFeasibilityFlag)

        # print best solution
        print('best solution (path):\n', best_solution)
        directional_N_paths = [decode_one_path(one_path) for one_path in population[minIndex]]
        link = sum(directional_N_paths)
        print('best solution (link): \n', link)

        print(f'---------------------- generation {i + 1} evolved! Time: {evol_end - elitism_begin:.4f}s ----------------------\n')

    # plot results
    result_stats(progress_with_penalty, progress)

    # print best solution
    minIndex = population_fitnesses_add_penalty.index(min(population_fitnesses_add_penalty))
    best_solution = population[minIndex]
    print('best solution (path):\n', best_solution)

    # check if all constraints are met (ideally True)
    print("\nAll constraints met?", check_feasibility(best_solution, checkDemand=checkDemandFlag, checkRushHour=checkRushHourFlag, checkMaxWorkingHour=checkMaxWorkingHourFlag))
    directional_N_paths = [decode_one_path(one_path) for one_path in population[minIndex]]
    link = sum(directional_N_paths)
    print('best solution (link): \n', link)

```

```python
if __name__ == "__main__":

    """initialization for genetic algo"""
    initial_prob = 0.8
    pusan_prob = 0.2
    population_size = 20
    elitism_cutoff = 2
    mutation_num = 1
    loop_limit = 100
    evolution_depth = 50000

    """initialization for buses"""
    # # of buses
    N = 23
    # #seats on each bus
    D = 50
    tolerance = 0
    intervalDuration = 0.5
    # numerical example
    demand = np.array([
        [114,106,132,132,117,83,57,52,13,8,18,13,26,3,13,10,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,14,2,0,7,12,7,9,5,7,7,12,9,32,39,53,35,30,18,60,44,60,53,90,58,78,71,35,55]
    ])
    demand_JQJY = demand
    demand_JQJY = demand_JQJY.astype(int)
    demand_PS = np.around(demand / 9)
    demand_PS = demand_PS.astype(int)

    intervalNum = demand.shape[-1]
    maxWorkingHour = 4
    checkDemandFlag, checkRushHourFlag, checkMaxWorkingHourFlag = True, True, True
    alpha, demandViolationPenalty, rushHourViolationPenalty, maxWorkingHourViolationPenalty = 1, 20, 17, 15

    # run main function
    run_evolution(population_size, evolution_depth, elitism_cutoff)
```

Appendix F: Survey Data on Students' Preference on Bus Schedule

	7:00	7:30	8:00	8:30	9:00	9:30	10:00	10:30	11:00	11:30	12:00	12:30	13:00	13:30	14:00	14:30	15:00
JinqiaotoAB	44	41	51	51	45	32	22	20	5	3	7	5	10	1	5	4	0
ABtoJinqiao	0	0	0	0	0	0	6	1	0	3	5	3	4	2	3	3	5
	15:30	16:00	16:30	17:00	17:30	18:00	18:30	19:00	19:30	20:00	20:30	21:00	21:30	22:00	22:30	23:00	23:30
JinqiaotoAB	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
ABtoJinqiao	4	14	17	23	15	13	8	26	19	26	23	39	25	34	31	15	24