Final project – NTHU bikes

By 108021121 王俊皓

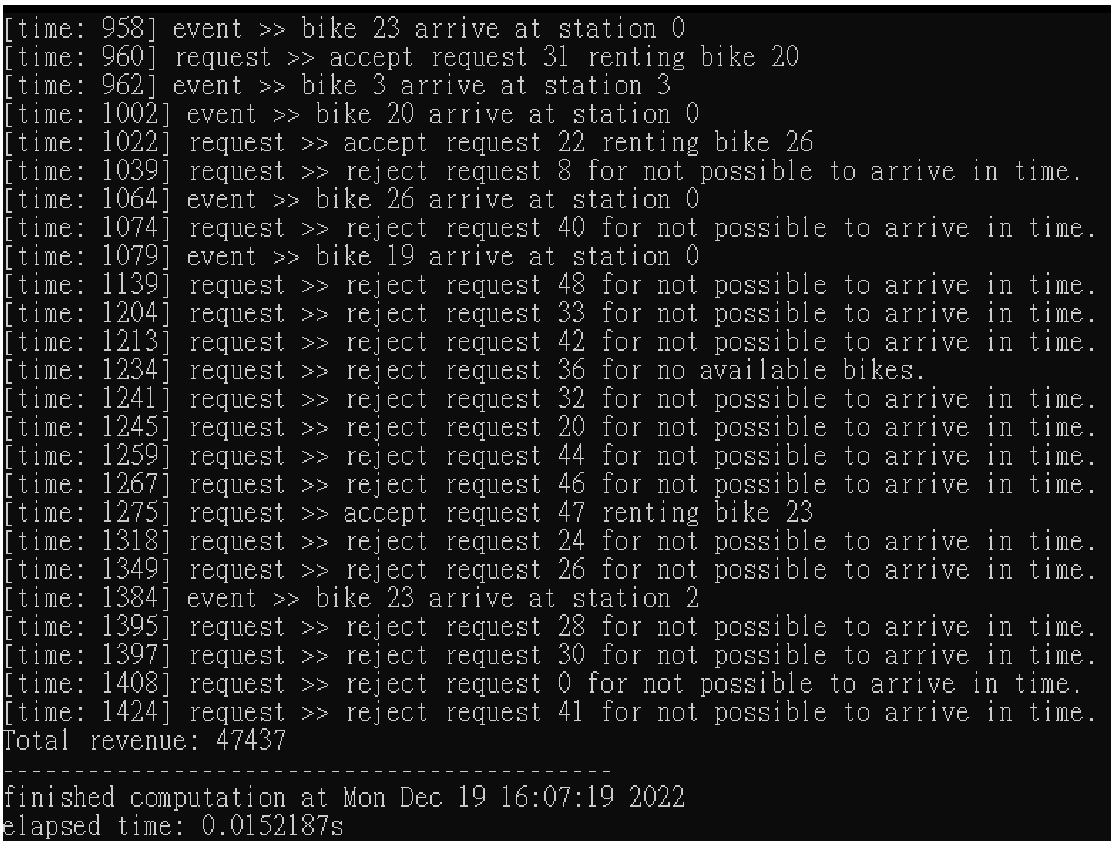
The problem will not be addressed here.

1. Compile and Run Example

For both versions, run `./bin/main <case> <version>` when in the folder 108021121_proj. For the advanced version, please use "advanced" instead of "advance" for `<version>`.

Then, the program will generate the .txt files required in folder "result/<case>".

There is an option in advanced.cpp for logs. Change "debug" to true to see rough logs of what happens in the command line. Screenshot below shows an example of the logs.



```
[time: 958] event >> bike 23 arrive at station 0
[time: 960] request >> accept request 31 renting bike 20
[time: 962] event >> bike 3 arrive at station 3
[time: 1002] event >> bike 20 arrive at station 0
[time: 1022] request >> accept request 22 renting bike 26
[time: 1039] request >> reject request 8 for not possible to arrive in time.
[time: 1064] event >> bike 26 arrive at station 0
[time: 1074] request >> reject request 40 for not possible to arrive in time.
[time: 1079] event >> bike 19 arrive at station 0
[time: 1139] request >> reject request 48 for not possible to arrive in time.
[time: 1204] request >> reject request 33 for not possible to arrive in time.
[time: 1213] request >> reject request 42 for not possible to arrive in time.
[time: 1234] request >> reject request 36 for no available bikes.
[time: 1241] request >> reject request 32 for not possible to arrive in time.
[time: 1245] request >> reject request 20 for not possible to arrive in time.
[time: 1259] request >> reject request 44 for not possible to arrive in time.
[time: 1267] request >> reject request 46 for not possible to arrive in time.
[time: 1275] request >> accept request 47 renting bike 23
[time: 1318] request >> reject request 24 for not possible to arrive in time.
[time: 1349] request >> reject request 26 for not possible to arrive in time.
[time: 1384] event >> bike 23 arrive at station 2
[time: 1395] request >> reject request 28 for not possible to arrive in time.
[time: 1397] request >> reject request 30 for not possible to arrive in time.
[time: 1408] request >> reject request 0 for not possible to arrive in time.
[time: 1424] request >> reject request 41 for not possible to arrive in time.
Total revenue: 47437
-------------------------------------------
finished computation at Mon Dec 19 16:07:19 2022
elapsed time: 0.0152187s
```

2. Used Data Structures

In this section, Data structures of all used objects will be introduced. Note that only the functions that is actually used will be introduced.

- Individual **bike**

  The way to store bike data is rather simple. Each bike contains:

  - **id** (int): The bike id.
  - **type** (int): The type of the bike.

- **station** (int): The station id that the bike is in. updates when arriving a station.
- **is_available** (boolean): whether it is available for rent.
- **price** (double): The current price of the bike.
- **rental_count** (int): The number of times that the bike has been rented.

- **request**

  A request has elements as followed:
  - **user_id** (int): the id of the user that made the request.
  - **rejected** (boolean): Whether it is rejected.
  - **accept_bike_type** (**myintarray**): the type(s) that the user requested.
  - **start_time** (int): the start time the rent started.
  - **arrive_time** (int): the deadline time the rent must end.
  - **start_station** (int): the starting station the rent wanted.
  - **arrive_station** (int): the destination station the rent wanted.
  - **start_deadline** (int): Advanced. The latest start time that make this request completable.

  With a custom comparison:
  - Request r1 is **less than** request r2 if r1 has strictly lower start_time or if r1 has a lower user_id when the two have the same start_time.

- **result_log**

  One result log contains:
  - **user_id** (int): the **id** of the user that is involved.
  - **bike_id** (int): the **id** of the **bike** that is involved. Not specified if the request is rejected.
  - **accept** (boolean): whether the request is accepted.
  - **revenue** (int): the money the request has earned. Not specified if the request is rejected.
  - **start_time** (int): the time that the user receives the rented bike. Not specified if the request is rejected.
  - **arrive_time** (int): the time that the user returns the rented bike. Not specified if the request is rejected.
  - **start_station** (int): the station that the user receives the rented bike. Not specified if the request is rejected.
  - **arrive_station** (int): the station that the user returns the rented bike. Not specified if the request is rejected.

  In implementation, they are stored as a result log array with ascending **user_id**.

  This data structure is used to get sorted output files. (user_result.txt and transfer_log.txt)

- **myintarray**

  This is a custom array structure dedicated for storing unspecified number of integers. It will double the memory every time it ran out of space. Mainly used for storing accept_bike_type from requests. Some of the request can accept multiple types.

  It has elements of the following:

  - **size** (int): the number of integers is stored.
  - **max_size** (int): the current maximum number of integers that it can store.
  - **data** (int*): the array that stores the actual data.

  It has functions of the following:

  - .**append**(int): add input integer to the last of **data**. Reallocate memory if needed.
  - .**search**(int): search if a certain integer exists in the array. Since the array will be mostly very small, there is no need to implement binary search. This function uses linear search. Returns true if such integer exists and return false otherwise.

  Constructor defaults **max_size** to 1. This can be set to any positive integer via calling the constructor with input.

- **bikeNode** and **station**

  station structure stores all information needed of a station for the program.

  station has elements of the following:

  - **id** (int): the id of the **station**.
  - **bike_count** (int): the number of bikes at this station. Exclude retired **bike**s.
  - **root** (**bikeNode***): the start of the **bike**s at this station. **bike**s are stored as a linked list.

  bikeNode contains:

  - **Bike** (**bike**): the bike data.
  - **next** (**bikeNode***): pointer to the next **bikeNode**.

  stations have behaviors of the following:

  - .**rent**(**myintarray**): according to the input array, choose the appropriate bike to rent and remove it from the station. Returns the **id** of the selected **bike**.
  - .**arrive**(**bike**): append the input bike to the linked list and other variable records to the station.
  - .**sort**(): sort all **bike**s in the station in ascending order with respect to

each bike's **id**. Uses insertion sort. Other sort methods are not better for linked lists because of the limitations to linked list having $O(n)$ entry access. This is for the file output station_status.txt.
- **.rent_advanced**(**request**, **bike***): Advanced. rent a specific bike from the station and returns the revenue, also go through all the process of renting. This is implemented for efficiency and is conceptually bad, could seek for better intuition.

- **event** and **event_heap**
  This data structure is for handling bike arrives. Using a min-heap structure, we can get every event that happened in the period of the last request to this request, while not having to check for every bike's status and figure out which ones are arriving.
  Expandable for other types of events, as for this project, only arrive type is implemented.
  An **event** structure consists of:
  - **type** (string): indicates which type of event this is. Possible types are: "arrive".
  - **occur_time** (int): the time the event happens.
  - **Bike** (**bike***): The pointer of the **bike** that is involved in this event.
  - **Station** (**station***): The pointer of the **station** that is involved in this event.

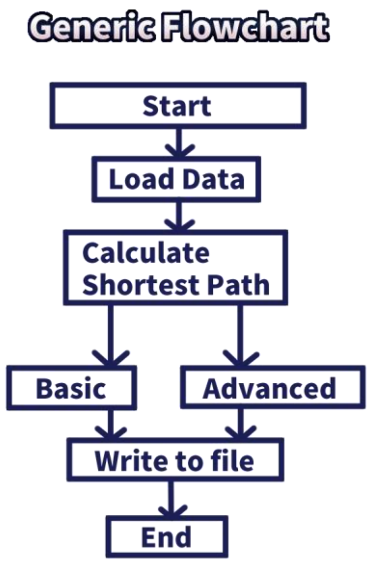
  An **event_heap** has the following elements:
  - **size** (int): The number of **event**s that is in the heap.
  - **max_size** (int): The maximum number of events the heap can handle. Doubles every time **events** ran out of space.
  - **events** (event*): The actual storage for events. Is always in the state of a minimum heap. Reallocate memory every time it ran out of space.

  It can have behaviors of the followings:
  - **.execute**: Let the next **event** happen.
  - **.schedule**(**event**): Schedule an **event** into the heap.
  - **.realloc**: Doubles the memory for the heap to store more **event**.
  - **.nextdate**: Get the **occur_time** of the next **event**.
  - **.is_empty**: Whether there are no more events in the heap.
  - **.what_to_do**(**event**): Advanced. In the advanced version of the code, immediate decision is not required since requests can wait. This function is an algorithmic core of the advanced implementation.

3. Algorithms

- **Generic**

The following is the general flowchart of the algorithm.



The data loaded in is stored in arrays of a size 10 initially. Once reached, the memory is reallocated twice as big. Once the load is complete, arrays are allocated again to be exactly what size needed.

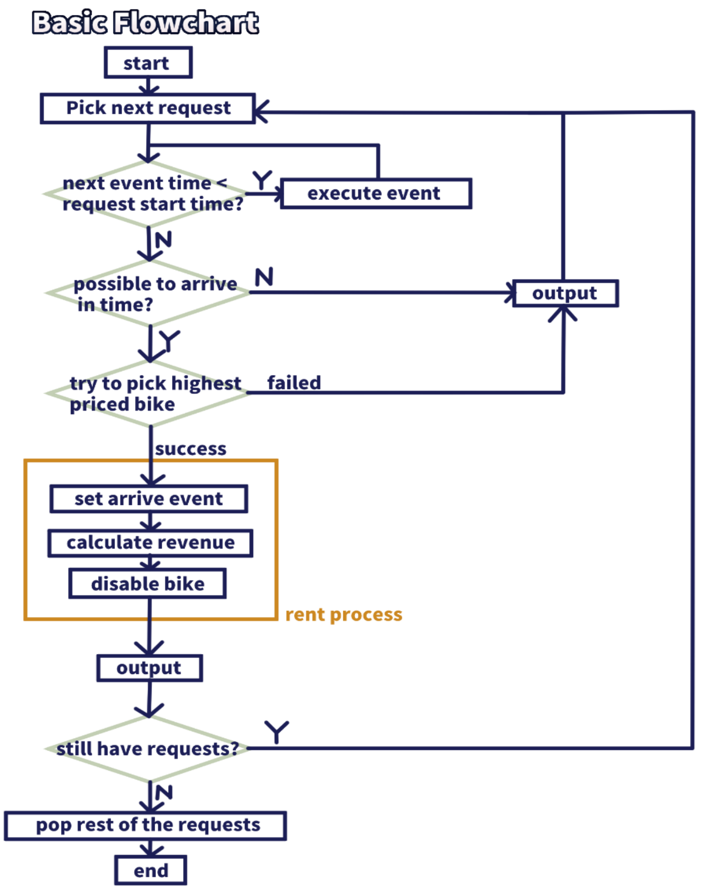
The requests are sorted using heap sort in this stage.

The map is stored in the form of adjacency array. Uses the Floyd-Warshall algorithm to obtain the all-pairs shortest path.

The stations are generated after.

- **Basic**

  Below shows the flowchart of the basic version.



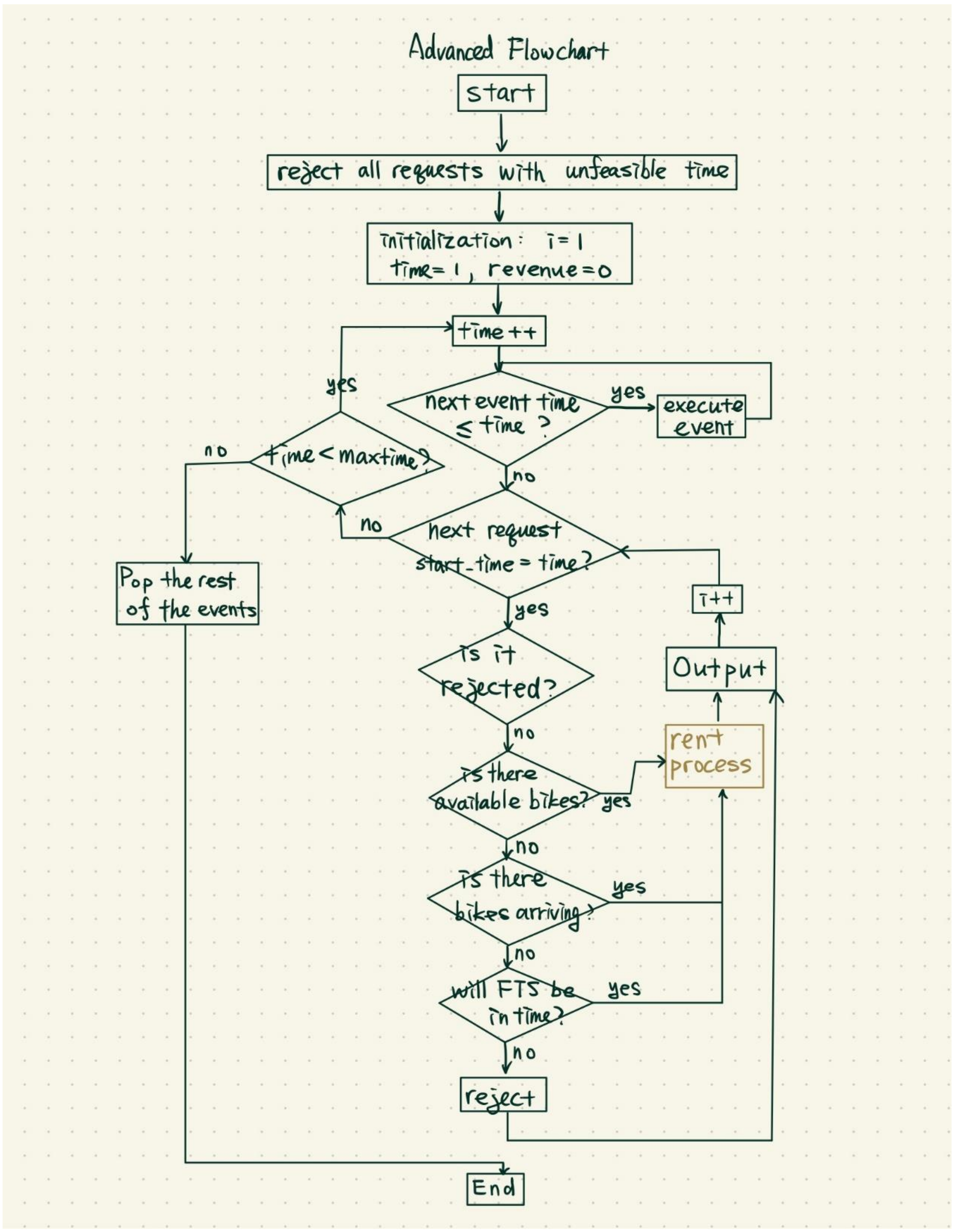
Typo: pop rest of the "events".

The basic section loops on requests. At first, the event_heap will be empty.

Then, we check if we want to accept the request. The criteria are shown above. Afterwards, if it turns out possible, we rent the bike. The rental count increases right after the revenue is calculated. This will also schedule an arrive event into the event_heap. Then, record this to the output array.

Before each event onwards, any events that has lower occur_time than the request's start_time will be executed in order.
It keeps looping until there are no requests left. At last, all ongoing bikes will arrive in order, to finish the algorithm.

- **Advanced**



Error: go to output if the request is rejected.

The advanced version will be a bit more complicated. First, we reject all requests that is not possible. That is, there is no way to arrive in time.

This time, the iterator is changed to time. Like basic algorithm, we check for events first, then check requests after. For each check of request, if there are bikes that fits, still rent the highest cost one.

If there is none fits, however, since we are allowed to wait, first check if there are bikes incoming to the station. If so, we wait for that, “merge” the request with the arriving one by extending the arrive timer and change the end station.
If not, check if it is possible to use the free transfer service to send some bikes here and then rent it.
If all the above are not the case, finally reject the request.

4. Test cases revenue results

Test cases	Basic	Advanced
1	47437	64674
2	926832	1092605
3	26425651	27897419

Not a significant change. There are some more potential changes can be done. First is what we originally tried. Make it so we don’t decide whether to reject or not until the request “expires.” Rent the bike accordingly by price to requests’ distance. This is to maximize higher priced bikes to be rent by people going further. This quickly became complicated since we have to save requests in a lot of places. In stations, in event_heap and so on. Also, a good portion of requests expires really long after start, which means by that time, they could have made it and transferred back.
Secondly, make it so that the free transfer service forever running to keep, as much as it can, all stations have all bikes. Again, this will need to change the past whenever a request rent the last bike. These technics have great complexity in greedy algorithms and can be very tricky to deal with.