UNIVERSIDADE DE SÃO PAULO
ESCOLA POLTÉCNICA
PMR - ENGENHARIA MECATRÔNICA

EDUARDO FELIPE NUNES FUNÇÃO, JOSÉ RAFAEL SOUZA DO NASCIMENTO

ESTUDO E OTIMIZAÇÃO DA LOCALIZAÇÃO DE BICICLETAS COMPARTILHADAS NA UNIVERSIDADE DE SÃO PAULO

TRABALHO DE CONCLUSÃO DE CURSO

SÃO PAULO
2021

EDUARDO FELIPE NUNES FUNÇÃO, JOSÉ RAFAEL SOUZA DO NASCIMENTO

ESTUDO E OTIMIZAÇÃO DA LOCALIZAÇÃO DE BICICLETAS COMPARTILHADAS NA UNIVERSIDADE DE SÃO PAULO

Trabalho de Conclusão de Curso apresentado ao PMR - Engenharia Mecatrônica da Universidade de São Paulo

Orientador: Marcos S. G. Tsuzuki
Escola politécnica da USP

SÃO PAULO
2021

AGRADECIMENTOS

Aos nossos pais, irmãos e familiares, que nos incentivaram nos momentos difíceis e nos apoiaram nos tempos de ausência.

Aos amigos e colegas de curso, que sempre estiveram presentes, apoiando e trabalhando junto ao longo de toda a graduação.

Agradecemos ao nosso orientador, o Professor Marcos Tzusuki por ter aceitado nos acompanhar neste projeto. A sua ajuda e orientação foram essenciais para a nossa motivação à medida que as dificuldades iam surgindo.

À Escola Politécnica da USP e a todos profissionais do departamento de Engenharia Mecatrônica, essenciais no nosso processo de formação profissional, que nos deram toda a base para realização desse trabalho.

Para ser grande, sê inteiro: nada Teu exagera ou exclui. Sê todo em cada coisa. Põe quanto és No mínimo que fazes. Assim em cada lago a lua toda Brilha, porque alta vive.

- Fernando Pessoa

RESUMO

NASCIMENTO, José, FUNÇÃO, Eduardo. Estudo e otimização da localização de bicicletas compartilhadas na Universidade de São Paulo. 2021. 74 f. Trabalho de Conclusão de Curso – PMR - Engenharia Mecatrônica, Universidade de São Paulo. São Paulo, 2021.

Estações de compartilhamento de bicicletas estão sendo cada vez mais adotadas em pequenas e grandes cidades ao redor do mundo, melhorando a mobilidade urbana de forma sustentável e reduzindo a emissão de gases de efeito estufa e garantindo uma vida mais saudável aos seus usuários. Um dos grandes problemas associados ao seu uso está na falta de estações perto de regiões com alta demanda e elevada distância entre elas. Em um primeiro estágio foram utilizados dois algoritmos metaheurísticos, *Particle Swarm Optimization* e *Simulated Annealing* para obter os pontos ideais das estações e, em seguida, foram simuladas as interações dos usuários com as novas estações, utilizando o software gráfico Processing. Os dados utilizados para guiar os algoritmos foram obtidos da base de Nova York.

Palavras-chave: Compartilhamento de Bicicletas. Otimização por Enxame de Partículas. Recozimento Simulado.

ABSTRACT

NASCIMENTO, José, FUNÇÃO, Eduardo. Shared Bicycles Location Study and Optimization in the University of São Paulo: Users approach. 2021. 74 f. Trabalho de Conclusão de Curso – PMR - Engenharia Mecatrônica, Universidade de São Paulo. São Paulo, 2021.

Bike-sharing stations are increasingly being adopted around the world, improving urban mobility in a sustainable way, reducing greenhouse gas emissions, and ensuring a healthier life for their users. One of the major problems preventing further use is the lack of stations close to regions with high demand and the long distances between them. In a first stage, two metaheuristic algorithms, *Particle Swarm Optimization* and *Simulated Annealing* were used to obtain the ideal points of the stations, and then the interactions of users with the new stations were simulated using a graphic software named Processing. The data used to guide the algorithms were obtained from the New York City base.

Keywords: bike-sharing. Particle Swarm Optimization. Simulated Annealing. Facility Location Problem.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A mobilidade urbana é de importância ímpar para o entendimento do desenvolvimento das cidades, principalmente quando se trata de grandes centros urbanos. Especificamente no Brasil, a falta de planejamento urbano (com desenvolvimento voltado principalmente para o transporte rodoviário) contribuiu para que um problema crônico se revelasse no transporte do país. Trânsito problemático, engarrafamentos constantes, transporte público sucateado, falta de meios alternativos para locomoção e descaso com sistemas multimodais são alguns dos fatores que abrem a discussão, clamando por perguntas e para a procura de soluções.

Uma das formas de transporte que ganha popularidade desde a última década são as bicicletas compartilhadas. No Brasil, foram introduzidas em dezembro de 2008 com o "Pedala Rio", como uma forma de teste do modal na região sul carioca. Em São Paulo, foram várias as tentativas de introduzir o modal na cidade, como por exemplo o caso das bicicletas individuais da Yellow e outra da empresa Tembici em parceria com o banco Itaú (TEMBICI, 2020).

A abordagem da Yellow se baseava em um sistema de bicicletas que poderiam ser retiradas e estacionadas em qualquer lugar da cidade mediante o uso de uma trava para desbloqueá-las. Tal modelo apresenta alto custo de manutenção uma vez que muitas são jogadas ao chão ou mesmo roubadas. O modelo que será abordado neste relatório é o da empresa "Tembici" em que as bicicletas só podem ser retiradas e colocadas em estações espalhadas pela cidade. As suas principais vantagens são um menor custo de manutenção por conta da menor taxa de avaria feita pelos usuários e a facilidade de recolha das bicicletas para manutenção ou rearranjo, uma vez que se sabe exatamente a sua localização.

Apesar dos modelos de negócio para as bicicletas compartilhadas nem sempre triunfarem, é inquestionável o poder do modal como alternativa de desafogamento do trânsito em grandes centros urbanos, funcionando especialmente bem para distâncias curtas em percursos majoritariamente planos.

A partir disso, a questão que surge e o problema o qual se pretende tratar aparecem naturalmente: como otimizar a localização de estações de bicicletas compartilhadas, ou a localização das oficinas onde será feita a manutenção das bicicletas. Esse problema, em resumo, está atrelado diretamente ao fluxo de pessoas dentro de uma cidade. A ida e a volta do trabalho e assim como os demais afazeres das pessoas devem gerar um fluxo relativamente repetitivo e previsível, devendo ser essencial para a escolha dos locais das estações de bicicletas, assim como a sua quantidade necessária.

2 OBJETIVO E MOTIVAÇÃO

Estima-se que o mercado mundial de bicicletas compartilhadas está crescendo 20% ao ano de acordo com uma pesquisa realizada pela Roland Berger (BERGER, 2008). O principal mercado é o Asiático, sendo a China o principal expoente com mais de 2 milhões de bicicletas compartilhadas apenas em Pequim (BERGER, 2008) e em crescimento.

A expansão do mercado corrobora com a visão atual de ESG empresarial (Environmental, Social and Governance) de redução das emissões de carbono e melhora da qualidade de vida alinhado com o crescimento de faixas reservadas e sinalização para bicicletas em grandes metrópoles. Com o crescimento do mercado teremos o desenvolvimento de serviços e sistemas mais inteligentes de compartilhamento de bicicletas e também da sua reorganização e manutenção.

A otimização das estações de bicicletas compartilhadas visa garantir um conforto ao usuário que necessita do meio de transporte principalmente nos horários de maior demanda e em localidades específicas. Além de com isso reduzir os custos de transporte, rearranjo e manutenção da empresa num mercado em expansão. Desse modo, garantindo melhorias sociais para a população.

O problema é relevante e pode ser abordado no curso de Engenharia Mecatrônica pelo desenvolvimento de análises e utilização de algoritmos por programação para estudar o caso. Projetos de criação de softwares, utilizando-se algoritmos methaeurísticos para análise de grandes volumes de dados são muito relevantes na atualidade.

3 ESTADO DA ARTE

Vogel, Greiser e Mattfeld (2011) exploraram os padrões de atividade de sistemas de bicicletas compartilhadas. Por meio de *Data Mining*, este complexo sistema pode ser entendido de uma forma melhor, revelando também um dos principais problemas no compartilhamento de bicicletas: o desbalanceamento na distribuição das bicicletas, gerado justamente por esse fluxo desigual da atividade nas cidades durante o dia.

Outra forma de abordar o problema é mudando o foco para o reabastecimento das estações de bicicletas. Papazek et al. (2013) propõem um sistema para otimizar as rotas que os veículos de reabastecimento devem seguir para que as estações com mais demanda nos diferentes períodos do dia estejam abastecidas.

Seguindo a mesma linha de estudo, foi, também, proposto por Schuijbroek, Hampshire e van Hoeve (2017) um modelo para determinar a necessidade de rebalanceamento de bicicletas entre as estações e qual seria a melhor rota do veículo responsável pelo rebalanceamento.

A questão de definir uma localização de estações de bicicletas compartilhadas sempre esteve no cerne da problemática em vista da implementação de tais sistemas. Kloimüllner e Raidl (2017) discutem esse problema na fase do planejamento. Com informações reduzidas propõe-se um sistema que irá escolher onde serão construídas tais estações e de qual tamanho devem ser. Fatores como usuários em potencial, demanda local e *Budget* da empresa são levados em consideração. A principal vantagem de se utilizar um sistema computacional para essa decisão é que, dessa forma, a escolha manual das primeiras estações a serem construídas pode ser evitada, diminuindo-se os possíveis erros e perdas.

Caggiani et al. (2019) propõem um sistema para otimização baseado na satisfação do usuário. Nesse modelo, pressupõe-se que já exista um sistema em funcionamento na cidade. A partir daí, dados como desistência de usuários, além de informações sobre o abastecimento das estações (por exemplo, estações cheias ou vazias) podem ser usados para definir a alocação de recursos para esse sistema, procurando maximizar a satisfação do usuário.

Mais recentemente, algumas pesquisas têm considerado outras áreas influenciadas por sistemas de bicicletas compartilhadas. Yang, Jiang e Zhang (2021) investigam como esses sistemas podem ser responsáveis por aumentar a demanda em atrações e pontos turísticos da cidade.

Ter uma visão que procura por soluções ecológicas e sustentáveis tornou-se indispensável na gestão de uma cidade como um todo. Não diferente deve ser a forma como é tratada a questão da mobilidade urbana. Numa pesquisa de ABDELLAOUI ALAOUI e KOUMETIO TEKOUABOU (2021), foi proposto um sistema que utiliza internet das coisas e *Machine Learning* para facilitar a gestão, além de potencializar a disponibilidade de bicicletas e aumentar a lucratividade de sistemas de compartilhamento. O modelo também é capaz de prever o número de bicicletas utilizadas durante algum período de tempo com base em diversos parâmetros

dinâmicos. Dados reais sobre o sistema de compartilhamento de bicicletas de Londres foram utilizados para comprovar a eficácia do modelo.

Após a instalação de um serviço de bicicletas compartilhadas, são necessários aprimoramentos, visando a sobrevivência financeira da empresa. Dokuz (2021) propõe um modo de identificação espaciotemporal de estações-chave de bicicleta em uma cidade. Para isso são propostos dois algoritmos que devem identificar tais estações chave. Provando sua eficácia, os algoritmos podem ser bons aliados para a pesquisa do comportamento da mobilidade urbana em termos do uso de bicicletas, além de ser fonte de informações valiosas a respeito da satisfação do usuário.

Os efeitos observados na mobilidade urbana estão fortemente ligados com fenômenos observáveis sociais e econômicos. Em um artigo de Chibwe et al. (2021), pretendeu-se relacionar a taxa de desemprego com a demanda de bicicletas compartilhadas. Usando dados do sistema de Londres, como era previsto, observou-se forte relação entre as duas situações.

De forma mais prática Cintrano, Chicano e Alba (2020) documentam a utilização de diferentes algoritmos metaheurísticos para a definição de localização de estações de bicicletas compartilhadas. Aplicando-se a um caso real, o artigo busca definir as melhores localizações para as estações do sistema de bicicletas compartilhadas da cidade de Málaga, na Espanha. Os algoritmos metaheurísticos são uteis para solucionar o problema das p-medianas e se aplicam bem no caso da otimização da localização de estações de bicicletas compartilhadas.

4 BASE TEÓRICA

Revisando a pesquisa feita acerca do tema, verifica-se que o desafio de otimizar a localização de estações de bicicletas compartilhadas se enquadra bem como um problema das p-medianas. Nesse tipo de problema, deve-se buscar por escolher localizações ótimas de acordo com pontos de demanda já especificados.

4.1 PROBLEMA DAS P-MEDIANAS (*P-MEDIAN PROBLEM*)

Desenvolvido a partir das pesquisas de Hakimi (1964), o problema p-mediana tem como alvo encontrar um conjunto 'p' de facilidades que devem atender a outro conjunto 'n' de pontos de demanda. As facilidades devem ser escolhidas de tal forma que as distâncias entre elas e os pontos de demanda sejam minimizadas (SILVA; MESTRIA, 2019).

É possível dividir o problema em dois tipos: capacitado e não-capacitado. Neste último, cada localização candidata possui uma capacidade não limitada. No primeiro, as localizações em potencial possuem capacidade finita (TRAGANTALERNGSAK; RöNNQVIST, 2000).

Os problemas p-mediana são *NP-hard*. NP é a sigla em inglês para tempo polinomial não determinístico (*Non-Deterministic Polynomial time*), definindo a classe de problemas que podem ser resolvidos em tempo polinomial por uma máquina de Turing não-determinística. *NP-hard*, em complemento, define a classe de problemas que são pelo menos mais difíceis que problemas NP.

Além disso, esses problemas são de natureza combinatória. Existem várias formas para sua solução: para problemas de pequena dimensão, podem ser utilizados métodos exatos, os que buscam o ponto ótimo; para problemas de grandes dimensões, no entanto, não é viável que se promova uma busca por soluções exatas devido ao tempo de processamento que tais métodos iriam requerer (STEINER, 2003).

4.2 OUTROS ALGORITMOS ESTUDADOS

Para problemas de grande porte, devem-se buscar por soluções aproximadas. Métodos que forneçam esses tipos de solução (que se aproximem da solução ótima) em um tempo praticável. Mais especificamente esses problemas de grande porte podem ser solucionados pela heurística relaxação Lagrangeana. No entanto, este método funciona bem somente para problemas específicos, não demonstrando grande eficiência para casos genéricos (COSTA, 2005).

Outra tentativa para a solução de tais problemas encontra-se na metaheurística. As principais técnicas da metaheurística são: Algoritmos Genéticos, *Simulated Annealing*, *Particle Swarm Optimization Variable Neighborhood Search* (Busca em vizinhança variada),*Chemical Reaction Optimization(CRO)*, *Iterated Local Search*, entre outros (SILVA; MESTRIA, 2019).

4.3 SIMULATED ANNEALING

O primeiro algoritmo implementado é o *Simulated Annealing* (SA), ou Recozimento Simulado, utilizado com o objetivo de se encontrar a localização ótima global das estações. O algoritmo traz o seu nome de uma metáfora do processo de recozimento de uma peça de metal que no início se encontra a uma Temperatura elevada fazendo com que os átomos tenham maior liberdade e com o tempo, conforme esfriam acabam por encontrar regiões com menor energia (TROSSET, 2001).

Nessa técnica, escolhe-se uma solução inicial aleatória dentro do domínio. Em seguida, dentro de um laço, o resultado é incrementado e comparado com novos valores obtidos da função objetiva. Dessa forma, caso o valor seja menor do que o anterior e, portanto, melhor posicionado no domínio, a nova solução é aceita. Quando um novo resultado não é menor do que o ótimo local, ele não necessariamente será descartado. No SA, pode-se aceitar o valor quando um número aleatório for inferior ao fator de $e^{\frac{-\triangle}{T}}$. Esse fator depende do delta entre as soluções que estão sendo comparadas e da Temperatura do sistema, a qual diminui a cada iteração.

Dessa forma a probabilidade de se aceitarem soluções piores é maior no início das iterações, propiciando ao código a possibilidade de rejeitar pontos de mínimo local, atingindo-se com o tempo pontos de mínimo global. O *loop* do programa termina quando, após um determinado número de iterações, não forem encontradas novas soluções melhores do que a corrente. Vide pseudocódigo abaixo.

Algorítimo "Simulated Annealing"
N ⟵ numero-estacoes-inicias
X ← posicoes-iniciais (N)
X ← objetiva(X)
ENQUANTO num-aceitos $>$ 0 FAÇA
    num-aceitos ← 0
    num-testados ← 0
    ENQUANTO $num - aceitos < iter$ E $num - testados < iter * 2$ FAÇA
        X ′ ← incrementa(X)
        num-testados ← num-testados $+1$
        SE $objetiva(X') < objetiva(X)$ OU $random(0{,}1) < ex(-\triangle objetiva()/T)$
ENTÃO
            X ← X'
            num-aceitos ← num-aceitos $+1$
RETORNA X

4.4 PARTICLE SWARM OPTIMIZATION

O *Particle Swarm Optimization* (PSO) é um algoritmo metaheurístico que consiste na obtenção de um ponto correspondente à optimização da função objetiva por meio do uso de um sistema de *swarm* (enxame) inteligente (HUDAIB; HWAITAT, 2017). Originalmente foi desenvolvido para estudar graficamente o movimento aleatório de pássaros, criando um enxame de partículas que se relacionam entre si e com o ambiente (KENNEDY; EBERHART, 2001).

Inicialmente determina-se o número de partículas que vão estar presentes no enxame e as suas respectivas posições e velocidades iniciais. Define-se como critério de parada um número máxima de iterações ou com um critério de erro mínimo da solução final. Para cada partícula presente no enxame, determina-se o erro local, assim como se avalia se essa partícula, é a melhor global, e atualiza o valor global. Após isso, percorre-se o enxame novamente, atualizando-se as velocidades e as posições. Vide pseudocódigo abaixo baseado em (HUDAIB; HWAITAT, 2017).

A seleção da próxima posição de cada partícula leva em consideração a sua velocidade que é alterada, considerando-se a inércia da partícula para alterar a velocidade anterior, a velocidade cognitiva correspondendo ao mínimo local para aquela partícula e uma terceira velocidade chamada de velocidade social responsável pelo encaminhamento da partícula ao ponto de mínimo global obtido até o momento (KENNEDY; EBERHART, 2001).

```
Algorítimo "Particle Swarm Optimization"
N ⟵ numero-estacoes-inicias
X ← posicoes-iniciais (N)
ENQUANTO i < Maxiterações FAÇA
        PARA cada partícula FAÇA
            objetiva(X)
            SE objetiva(X)<best.global ENTÂO
              best.global ← objetiva(X)
        PARA cada partícula FAÇA
            atualiza.velocidade
            atualiza.posição
RETORNA X
```

4.5 FUNÇÃO OBJETIVA

O objetivo do programa é minimizar a distância euclidiana (di(X)) entre os pontos de demanda e as estações (j) de forma que se leve em consideração a demanda (p) de cada região. A localização das possíveis estações é dada pelo vetor X =(X1,..., Xp) e cada estação Xj = (xj, yj). Levando isso em consideração estabeleceu-se a função objetiva representada pela

seguinte equação (Eq. 1) baseada pelo artigo de Drezner et al. (2015).

$$F(X) = \sum_{i=1}^{n} p * min_{1 \leq j \leq p}\{di(X)\} \tag{1}$$

4.6 Processos Estocásticos

Processos estocásticos podem ser definidos como uma série de variáveis aleatórias que são modificadas de acordo com a passagem do tempo. A variável definida convencionalmente por $X(t)$ representa alguma característica mensurável em um sistema em operação ao longo do tempo. Por exemplo, pode representar o número de bicicletas em uma estação de compartilhamento de bicicletas a cada momento, que pode ser dimensionado de acordo com chegadas e saídas de bicicletas, de forma aleatória (NOGUEIRA, 2017).

Os valores que X(t) pode assumir são denominados estados e o conjunto de estados possíveis são determinados por espaço de estados. A mudança de estados é um processo estocástico, portanto, acontece de forma aleatória de acordo com as Probabilidades de Transição, que definem qual é a chance de mudança de um estado para outro de acordo com o tempo. Um processo estocástico pode ser classificado de diferentes maneiras:

- Quanto ao tempo: Caso o tempo dentro de um processo varie dentro dos números reais, ele é chamado de processo estocástico de tempo contínuo. Da mesma forma, se o tempo for uma variável contável, dá-se o nome de processo estocástico em tempo discreto;
- Quanto ao espaço de estados: que, da mesma maneira, pode ser contínuo ou discreto (IBE, 2013).

4.7 Processos de Markov

Um processo Markoviano (ou Cadeia de Markov) representa um processo estocástico no qual a mudança para um estado futuro depende somente do estado presente. Em outros termos, trata-se de um processo estocástico sem memória.

Usualmente, um processo de Markov é definido com base em um diagrama de estados. A Figura 1 abaixo demonstra um exemplo de uma cadeia de Markov representado um processo com dois estados. As setas de transição indicam a possibilidade de mudança de estado, sendo que a soma das probabilidades de transição saindo de cada estágio deve ter soma 1 (IBE, 2013).

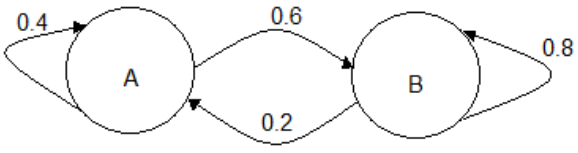

Figura 1 – Diagrama exemplo de uma cadeia de Markov com dois estados.

4.8 Filas M/M/1

Filas M/M/1 é a denominação dada para filas que possuem somente um ponto de atendimento, onde as chegadas são descritas por um processo de Poisson que ocorrem com uma taxa λ; e atendimentos são descritos de acordo com uma distribuição exponencial de taxa μ.

Essas filas são do tipo FIFO (*first in, first out* em inglês), que possuem a característica de que o primeiro cliente a chegar na fila será o primeiro a ser servido. Em outras palavras, os clientes são atendidos em ordem nas filas FIFO.

As filas M/M/1 podem ser modeladas como uma cadeia de Markov de acordo com o seguinte diagrama de estados descrito na Figura 2 (NOGUEIRA, 2017).

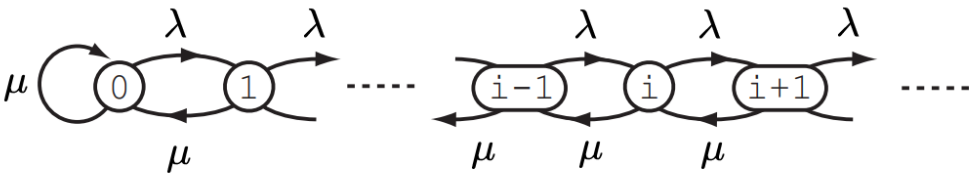

Figura 2 – Diagrama de Markov de uma fila M/M/1 (NORRIS, 2021)

Este sistema é, portanto, descrito por uma variável aleatória X(t) que representa o número de clientes na fila ao longo do tempo. Caso $\lambda < \mu$, o sistema é considerado estável. Isso significa que o serviço dessa fila atende numa taxa maior do que a taxa com a qual chegam os clientes.

No estado estacionário, sendo $\rho = \lambda/\mu$ a probabilidade de que uma fila M/M/1 possua i clientes é descrita por:

$$P(i) = (1 - \rho)\rho^i \tag{2}$$

Portanto, é possível observar que as probabilidades desse tipo de fila estão distribuídas de forma geométrica, com parâmetro $1 - \rho$. Assim, tira-se que o número médio de clientes dentro desse tipo de filas em estado estacionário é $\rho/(1 - \rho)$, com variância de $\rho/(1 - \rho)^2$ (LIBERMAN, 2006).

5 METODOLOGIA

O problema a ser considerado visa otimizar a localização de estações de bicicletas compartilhadas, tendo como base a demanda de usuários em certa região. Tais informações precisam ser obtidas nas bases públicas. Comparando-se as diferentes bases de dados, deve-se escolher uma base de dados atualizada, relevante e com qualidade boa de dados. A escolha deve ser crítica e, para tal, deve-se explorar as bases por meio de gráficos e visualizações. Problemas iniciais podem ser identificados dessa forma. A partir disso, serão utilizados diferentes algoritmos para se resolver o problema das p-medianas descrito no capítulo 4. Os resultados, então, precisam ser revistos e os algoritmos, comparados entre si.

5.1 OBTENÇÃO DE DADOS NYC

As bases de dados utilizadas para treinar o modelo foram retiradas do portal de bicicletas da empresa Citibike de Nova York (NYC, 2013) e do NYC Open Data (OPEN-DATA, 2020). Do primeiro portal, foram retirados os dados referentes as estações de início e fim do trajeto, como localização das estações por nome, latitude e longitude, duração das viagens entre duas estações, identificação da bicicleta que realizou o trajeto, data e hora da viagem, idade do usuário e gênero. Do segundo, é possível obter dados referentes à população em cada bairro a fim de calcular a demanda por região.

5.1.1 COLETA E TRATAMENTO DE DADOS DA CIDADE DE NOVA IORQUE

Em uma análise inicial da base, foram feitos os seguintes tratamentos de dados da cidade de Nova York.

Os dados foram inicialmente filtrados retirando os *outliers*. Para o caso em estudo o campo referente à duração da viagem (*tripduration*) apresentava alguns valores muito distantes da média padrão e por isso foram excluídos, como pode ser observado na Figura 3 em que se compara a duração da viagem pela idade do usuário.

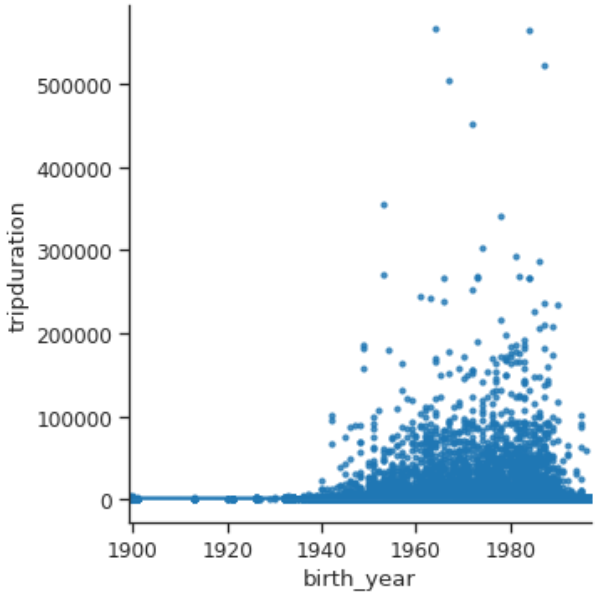

Figura 3 – Dados de duração de viagem na cidade de Nova York por idade.

A idade do usuário acaba por não ser tão representativa na análise, uma vez que existem muitos usuários não cadastrados utilizando o serviço, deixando os dados incompletos. Dessa forma, tal dado apenas será utilizado comparativamente.

Os dados foram analisados para verificar a completude da base (Figura 4). Pode-se identificar que os campos mais incompletos da base são do ano de nascimento do usuário e os campos relacionados com a estação final, a sua localização e o seu nome. Tal problema se deve principalmente a viagens não terminadas em que aconteceram problemas de funcionamento durante o percurso, o que acaba por afetar o campo *tripduration*.

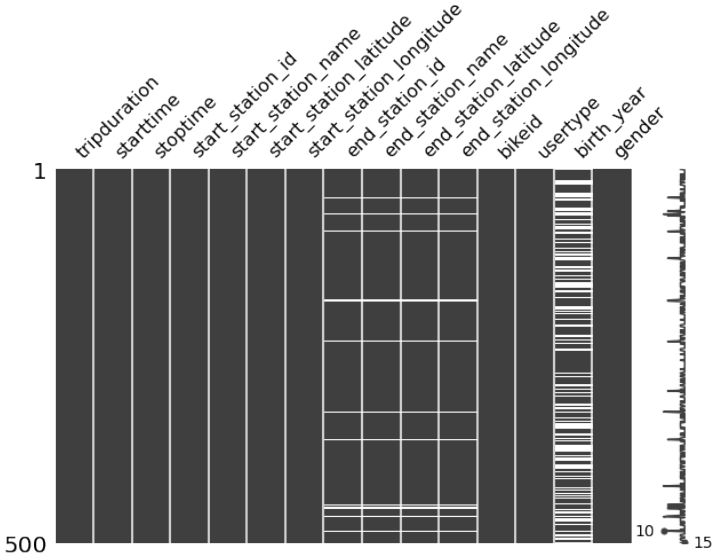

Figura 4 – Completude dos dados

Os campos numéricos da base foram normalizados em um segundo momento ficando

entre um intervalo de 0 a 1. Dessa forma é possível realizar uma análise de correlação entre os campos, observada na imagem 5 com um gráfico de *heatmap* (mapa de calor).

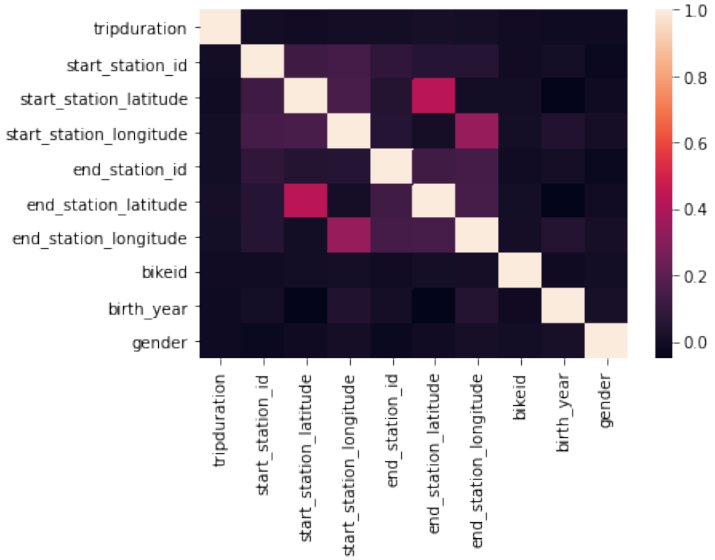

Figura 5 – Heatmap - Correlação dos dados

Os dados sobre a latitude e longitude das estações iniciais e finais, mostram que a maioria das viagens são realizadas em trajetos curtos.

Utilizando-se os pontos de latitude e longitude das estações iniciais e finais é possível desenhar em um gráfico real da cidade de Nova York as localizações das estações utilizadas durante o período estudado (Figura 6). No eixo x encontra-se a Latitude e o eixo y a Longitude. É fácil perceber que regiões perto das linhas de metrô possuem um maior número de estações, assim como em avenidas principais.

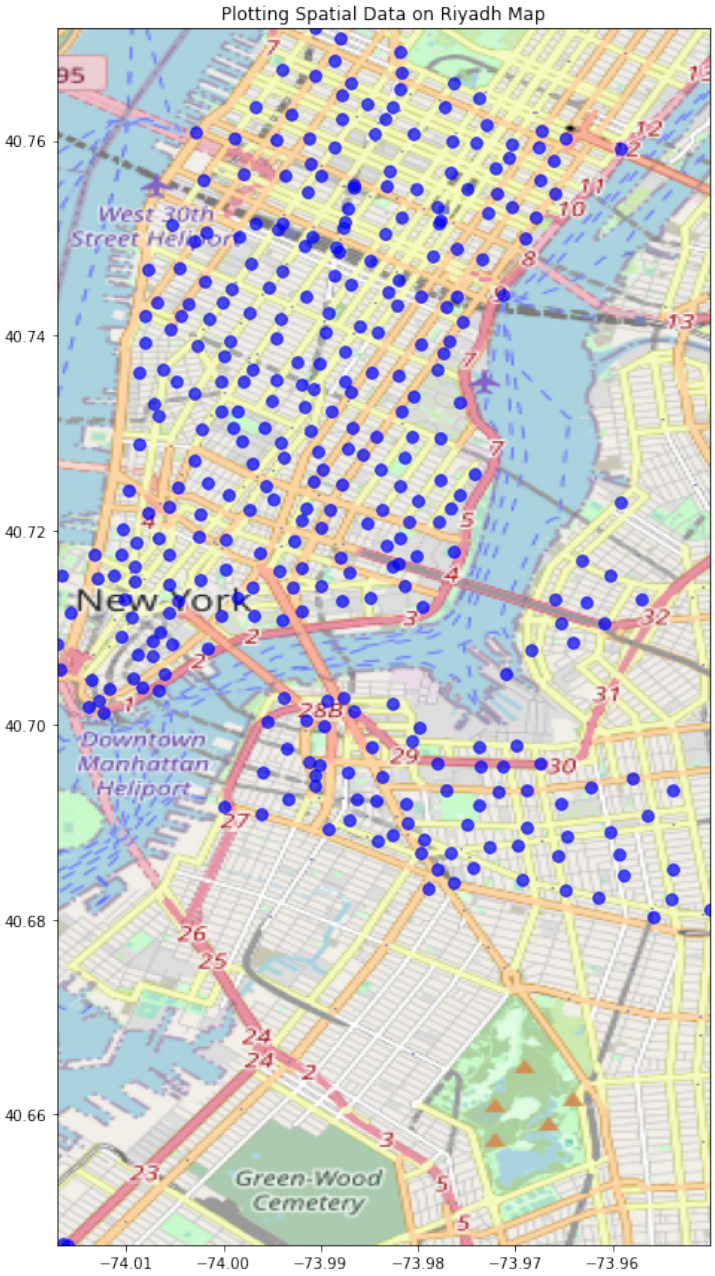

Figura 6 – Estações de bicicletas compartilhadas na cidade de Nova York.

Uma análise inicial também nos trouxe os principais horários de utilização das bicicletas Figura 7. Observa-se que os principais horários de pico ocorrem no final do dia entre às 17:00 e 19:00 horas. E o início das atividades se inicia entre às 7:00 e 8:00 horas da manhã.

Figura 7 – Histograma dos horários de utilização das bicicletas.

5.1.2 FREQUÊNCIAS DE UTILIZAÇÃO

Após o tratamento dos dados foi possível realizar a análise de frequências de saída e chegada das bicicletas em cada estação, assim como a velocidade média das viagens.

Para obtenção das frequências, foram utilizados dados de 30 dias da cidade de Nova York, observando-se, em um intervalo de uma hora, a frequência esperada de chegada e saída de bicicletas em cada estação. A fim de melhor observar a movimentação de bicicletas ao longo do dia alguns *hubs* foram retirados, restando apenas os que estivessem a pelo menos 1 km de distância entre-si, sobrando apenas 28 estações. Elas estão marcadas marcadas com pontos azuis e podem ser observadas na Figura 8.

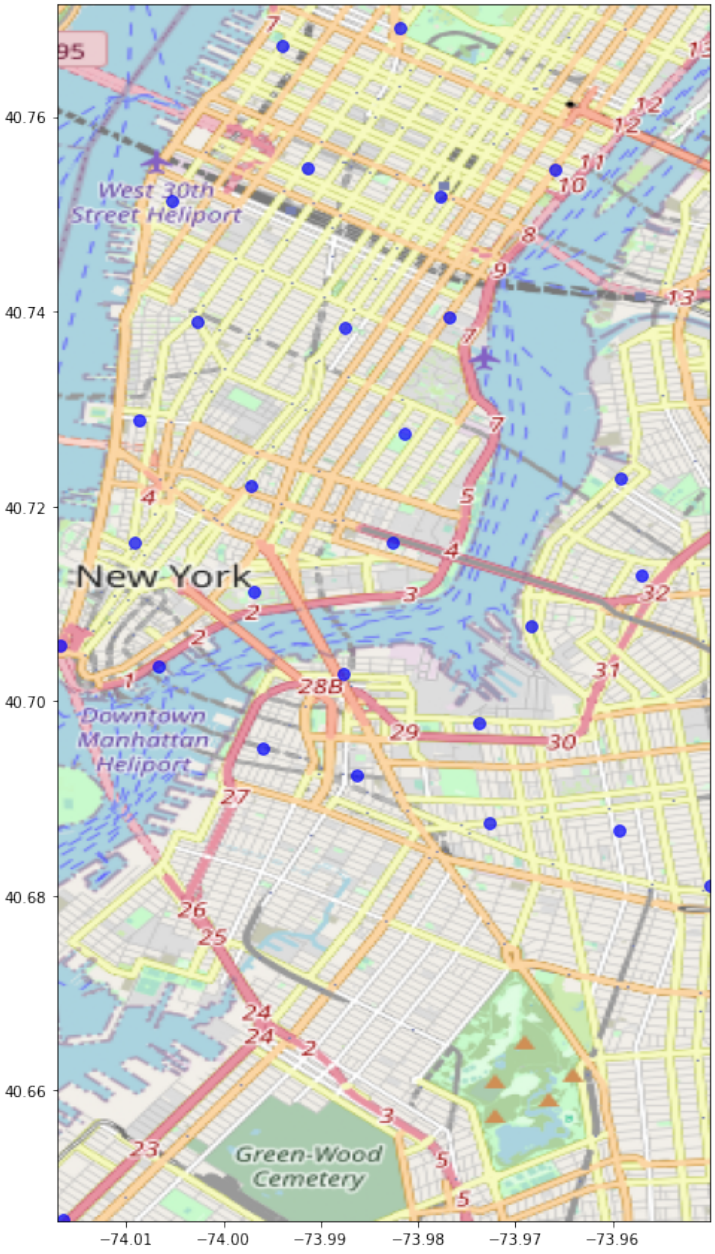

Figura 8 – Síntese das estações (em azul).

Com a finalidade de se obter a velocidade média de cada viagem, considerou-se que a movimentação ocorreu em linha reta, e a distância foi calculada com base na diferenças de latitude e longitude das estações de início e fim (Eq. 3 e 4). Em que lat_2 e lat_1 correspondem as latitudes dos dois pontos e lon_2 e lon_1 as respectivas longitudes, R representa o raio da terra em km.

$$\alpha = \sin^2(\frac{(lat_2 - lat_1)\pi}{180}) + \cos(\frac{lat_1\pi}{180}) \times \cos(\frac{lat_2\pi}{180}) \times \sin^2(\frac{(lon_2 - lon_1)\pi}{180}) \tag{3}$$

$$Distancia(m) = 2R \times 1000 \arctan(\sqrt{\alpha}, \sqrt{1-\alpha}); R = 6378.137(km) \tag{4}$$

Os resultados das distâncias médias percorridas, duração e velocidade média em cada viagem, podem ser observados nas Figuras 9, 10 e 11. Dos gráficos podemos observar que distâncias de até 3 km (80% da distribuição) e, consequentemente, tempos de viagem de até 30 minutos (80%) são preferíveis entre os usuários. Da mesma forma, conforme a Tabela 1, vemos que, embora o desvio padrão do tempo de viagens seja elevado, a média da velocidade se manteve em 2.47 m/s com um desvio padrão de 1.74 m/s. Esses valores foram utilizados em uma distribuição normal para fazer previsões de viagens na cidade de Nova Iorque que serão apresentadas na seção 6.1.

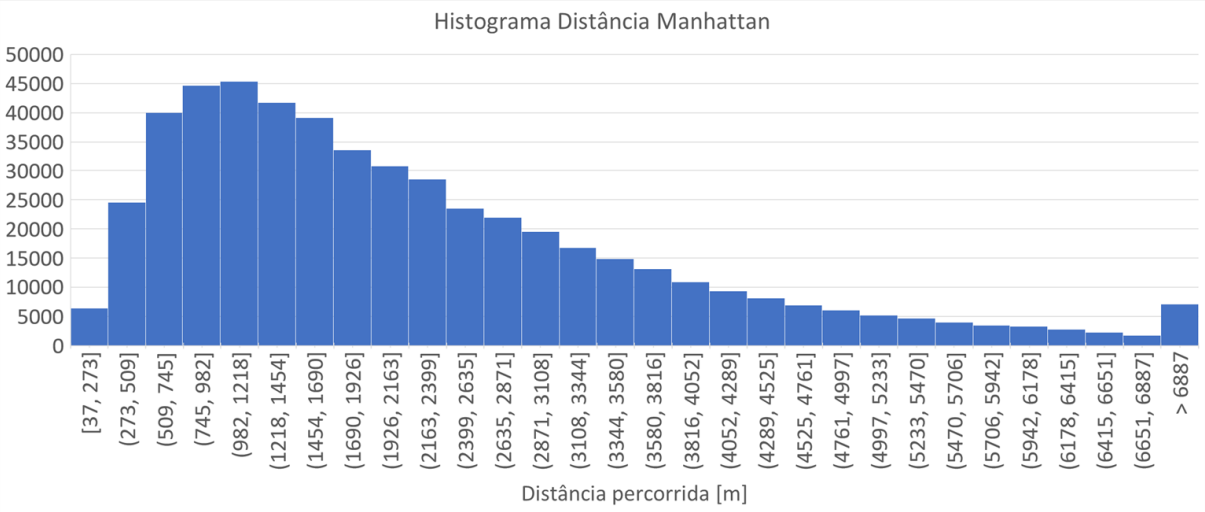

Figura 9 – Histograma das distâncias médias das viagens na ilha de Manhattan.

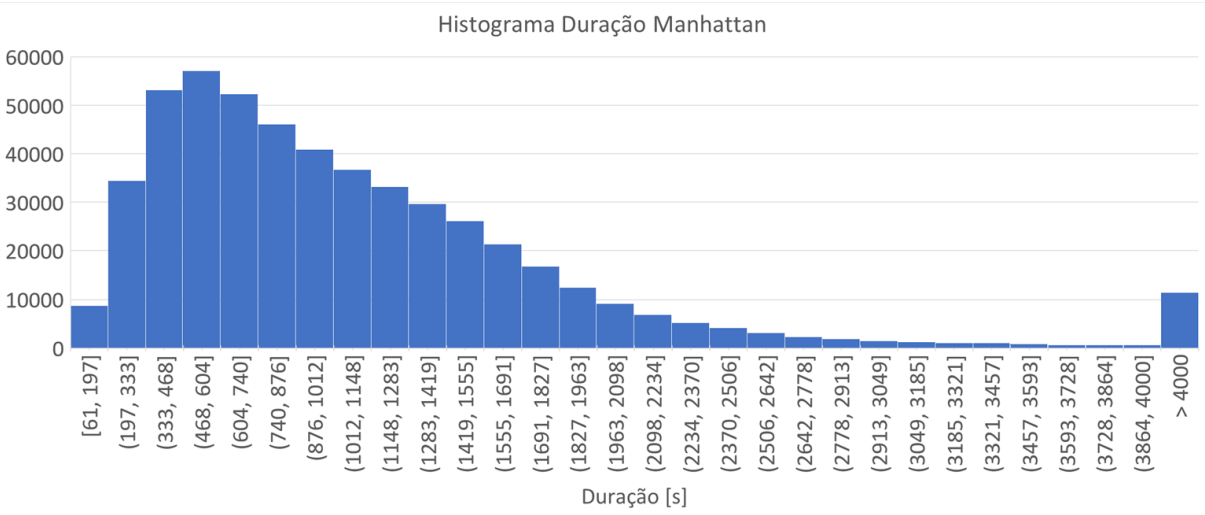

Figura 10 – Histograma das durações das viagens na ilha de Manhattan.

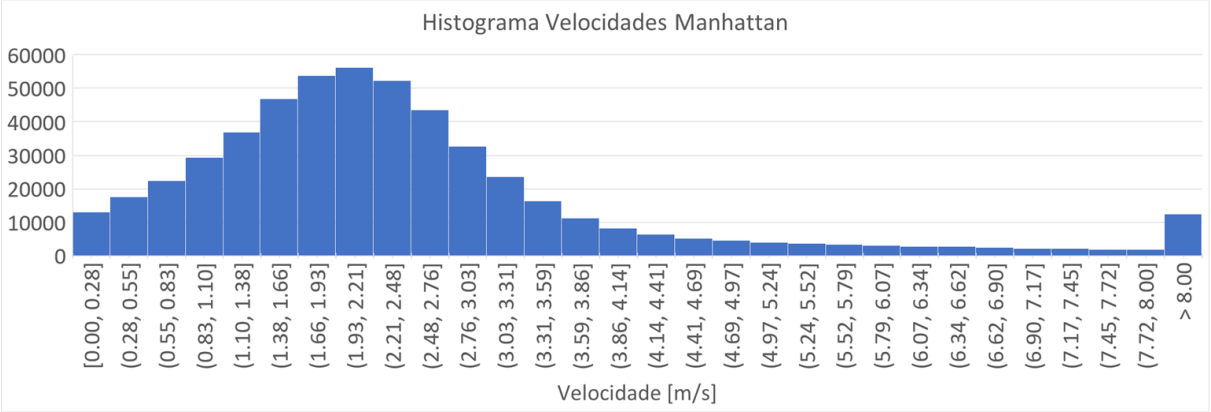

Figura 11 – Histograma das velocidades médias das viagens na ilha de Manhattan.

Tabela 1: Médias e Desvio Padrão para as grandezas observadas.

	Média	Desvio Padrão
Distância (m)	2216.2315	1556.9304
Duração (s)	1395.3479	9424.6224
Velocidade (m/s)	2.4666	1.7354

5.1.3 MODELAGEM DAS ESTAÇÕES COMO FILAS M/M/1

As estações de bicicleta podem ser modeladas como filas M/M/1. de acordo com o modelo descrito no capítulo 4.8, as frequências de chegada e de saída de bicicletas das estações podem ser representadas pelas taxas de chegada de clientes e de serviço nas filas: λ e μ, respectivamente.

Caso a frequência de chegada seja maior do que a frequência de saída, a fila é considerada instável, e, para valores de tempo muito grandes, ela crescerá indefinidamente. Em contrapartida, para filas estáveis (onde a frequência de saída é maior do que a frequência de chegada), pode ser calculado o valor médio de clientes dentro delas, no estado estacionário.

O gráfico 12 pontua qual é a quantidade de estações estáveis e instáveis. Observa-se que $45{,}95\%$ das estações são estáveis.

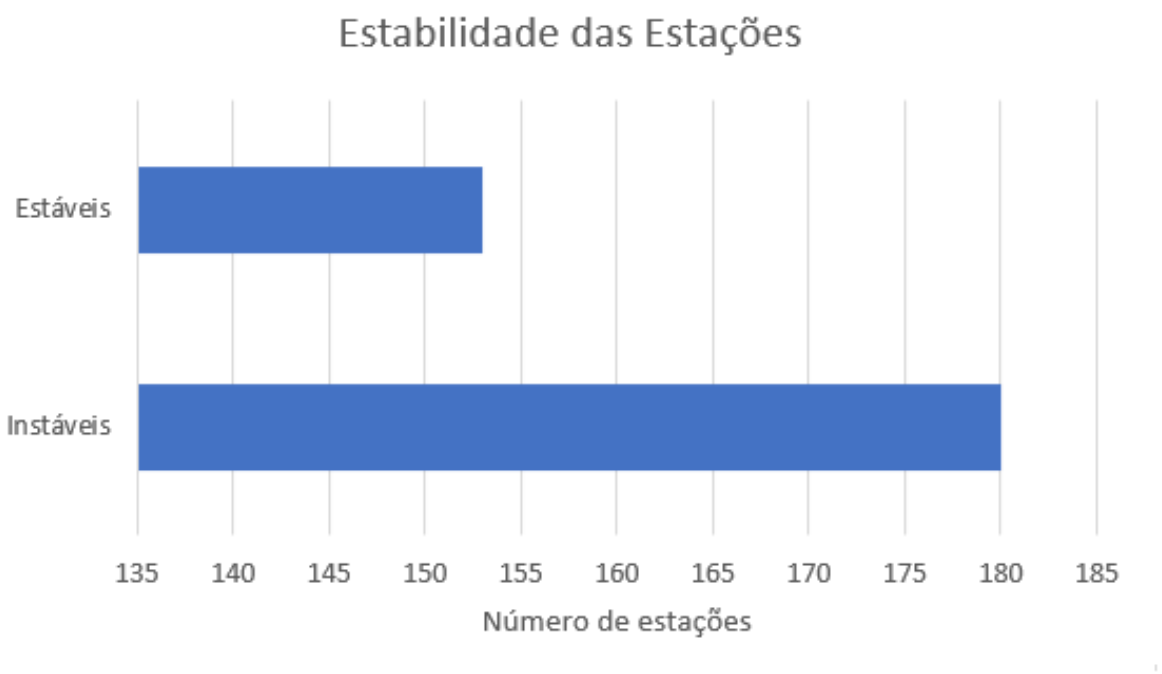

Figura 12 – Estabilidade das estações.

5.1.4 COLETA DE DADOS DA UNIVERSIDADE DE SÃO PAULO

A fim de estabelecer as estações de bicicletas e simular a movimentação dos usuários dentro do campus da cidade de São Paulo utilizou-se o Anuário da USP do ano de 2020 (USP, 2020) para se obter os principais pontos de demanda da universidade, conforme observado na Tabela 2.

Além dos dados da universidade também foram retiradas as posições de latitude e longitude de cada unidade para estabelecimento dos pontos de demanda dentro do campus.

Tabela 2: Comunidade USP segmentada por Unidade no ano de 2020.(USP, 2020)

Cidade Universitária "Armando de Salles Oliveira"	Recursos Humanos	Graduação	Pós-Graduação	Total de Pessoas
CEPEUSP	121	0	0	121
ECA	359	2256	1213	3828
EDUSP	52	0	0	52
EEFE	132	501	167	800
EP	802	5246	2584	8632
FAU	243	1336	880	2459
FCF	217	926	406	1549
FE	244	946	720	1910
FEA	239	3013	817	4069
FFLCH	705	9131	3661	13497
FMVZ	351	477	636	1464
FO	301	760	395	1456
HU	1331	0	0	1331
IAG	181	302	281	764
IB	284	763	515	1562
ICB	419	189	650	1258
IEA	26	0	0	26
IEB	54	0	118	172
IEE	146	0	310	456
IF	373	1294	312	1979
IGc	160	439	285	884
IME	278	1633	928	2839
IO	163	220	162	545
IP	198	413	971	1582
IPEN	0	0	805	805
IQ	300	758	488	1546
IRI	45	308	148	501
MAC	88	0	0	88
MAE	65	0	131	196
NAIPE	0	0	0	0
PUSP-C	159	0	0	159
PUSP-CL	7	0	0	7
RUSP	1170	0	0	1170

5.2 ALGORITMOS UTILIZADOS

Foram utilizados para análise os algoritmos *Particle Swarm Optimization* (KENNEDY, 1995) e *Simulated Annealing* (KIRKPATRICK; JR.; VECCHI, 1983) determinando a eficiência de cada algoritmo.

Com isso pretendeu-se obter novas localizações de estações de bicicletas compartilhadas nas regiões estudadas e analisar a eficácia delas.

5.3 VALIDAÇÃO DO ALGORITMO

A fim de validar o algoritmo, retirou-se dados públicos da cidade de Nova York, assim como os utilizados anteriormente só que dessa vez relacionados com a população da Ilha de Manhattan. O estudo dividiu inicialmente a ilha em 29 Neighborhood Tabulation Areas (NTA, 2010) e a respectiva população de cada área. Com isso, foi possível modelar 29 pontos de demanda com as localizações determinadas por latitude e longitude e também um fator de peso para cada ponto com base na quantidade de residentes por região. Os pontos selecionados da cidade podem ser observados na Figura 13. Uma segunda abordagem utilizando os dados do Census de 2020 (CENSUS, 2020) dividindo a ilha em 283 segmentos pode ser observada na Figura 14. Ambos segmentos foram utilizados no teste do algoritmo e na obtenção das novas posições de estações.

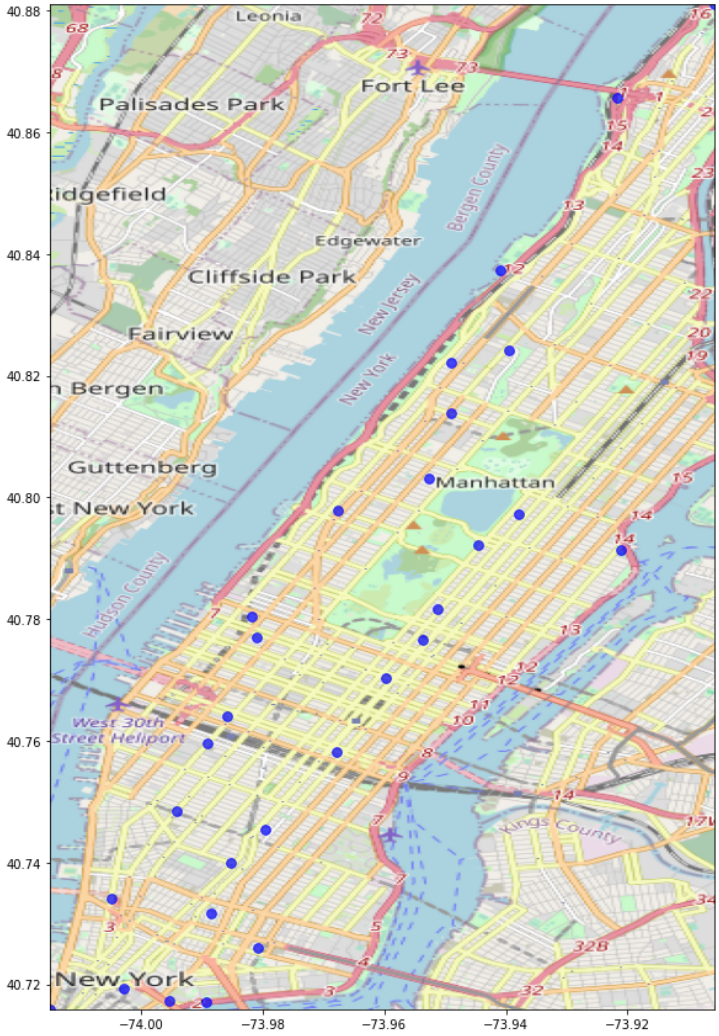

Figura 13 – Áreas populacionais da cidade de Nova York (NTA).

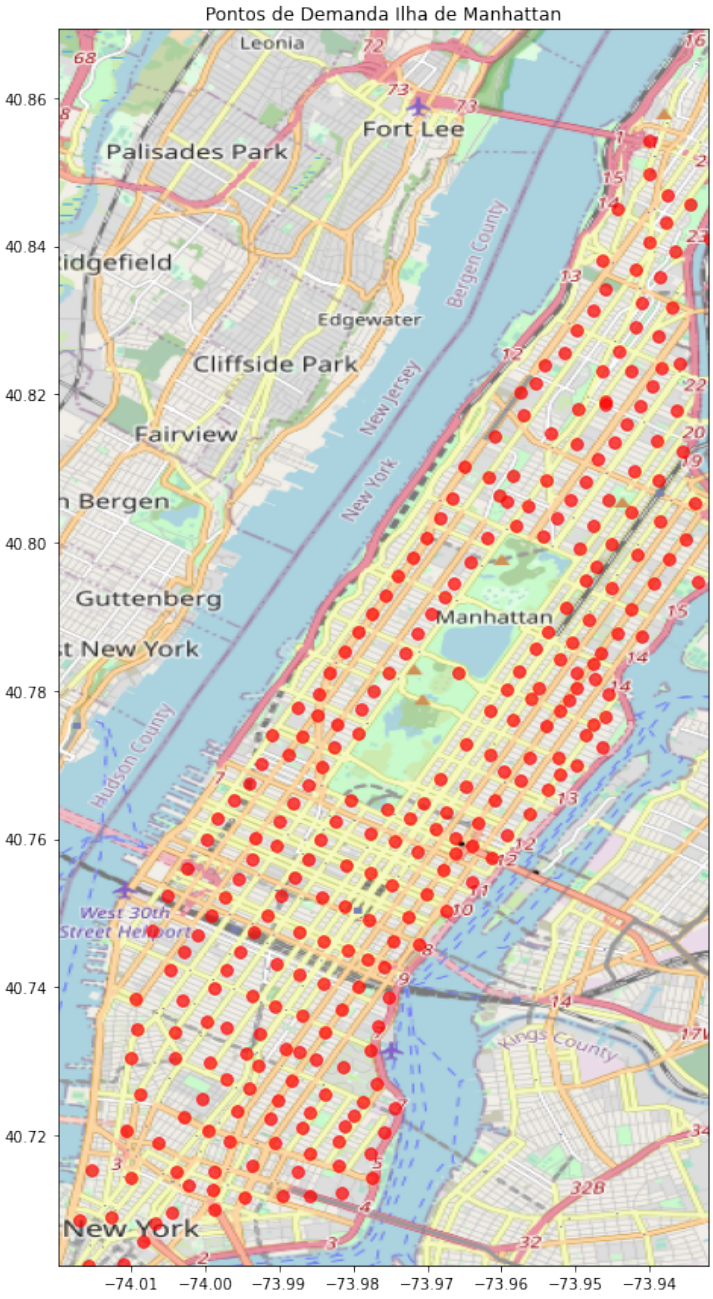

Figura 14 – Áreas populacionais da cidade de Nova York (Census 202).

5.4 PROCESSING

Processing é um software *open source* criado em 2001, que utiliza um ambiente de desenvolvimento integrado (IDE) direcionado à criação de artes visuais em duas ou três dimensões. Uma de suas grandes vantagens é a utilização de linguagens de programação já conhecidas e largamente difundidas, tais como JavaScript e Python. A utilização do programa consiste em duas funções principais, uma chamada setup(), em que se inicializam as variáveis, assim como o tamanho da janela gráfica e o *framerate* da animação. Na outra função draw(), colocam-se os códigos responsáveis por realizar os desenhos e cores da animação a cada *framerate*. Nas simulações utilizou-se o Processing 3.5.4 no ambiente de programação em Python 2.7 (FRY; REAS, 2001).

6 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Serão tratados nesse capítulo os resultados referentes às localizações das estações obtidas pelos algoritmos *Simulated Annealing* e *Particle Swarm Optimization* tanto na ilha de Manhattan quanto na cidade universitária "Armando de Salles Oliveira". As entradas dos dois algoritmos são os pontos de demanda respectivos estabelecidos no Capítulo 5. Alinhado com a criação das estações também foram realizadas simulações do deslocamento de usuários em cada região.

6.1 SIMULATED ANNEALING E PARTICLE SWARM OPTIMIZATION PARA MANHATTAN

Tabela 3: Comparação dos métodos utilizados (SA e PSO).

	SA	PSO
Entradas		
Nº de estações	75	75
Nº iteração Máxima	-	10000
Nº Aceites Máximo	3000	-
Nº Avaliados Máximo	6000	-
Fator de Cristalização máximo	40	-
Temperatura Inicial	0.1	-
Nº de partículas	-	1000
W (Constante de inércia)	-	0.1
c1 (Constante cognitiva)	-	2.5
c2 (Constante social)	-	3
Critério de parada	Nº Aceites $= 0$	Nº iteração Máxima OU Atual/Melhor < 0.99
Comparações		
Tempo de processamento (s)	180205.97	9457,84
Nº de iterações	804	109
Solução função objetiva	0.0367898	0.0505179
Δ Resíduo	0.0137281	

Utilizando-se os pontos de demanda da seção 5.3, mostrados na Figura 13 e os inputs dos programas apresentados na Tabela 3 conseguimos utilizar os códigos SA e PSO para o caso da ilha de Manhattan. Considerando-se que a área total da ilha de Manhattan corresponde à 59.1 km^2 e que, de acordo com o observado na seção 5.1.2, a moda da distância está aproximadamente em 1km podemos calcular o número de estações necessárias supondo uma

área circular de 1 km de diâmetro de atuação de cada estação. Dessa forma, obtemos que seriam necessárias, aproximadamente, 75 estações. A fim de confirmar essa hipótese inicial foram feitas diversas iterações com quantidades diferentes de estações, de 1 a 75, onde foram calculadas as distâncias máximas a serem percorrida por uma pessoa (Figura 15). Pode-se observar que posicionando estações de forma inteligente a distância cai rapidamente. Com 75 estações obteve-se uma distância máxima de 1006.49 metros.

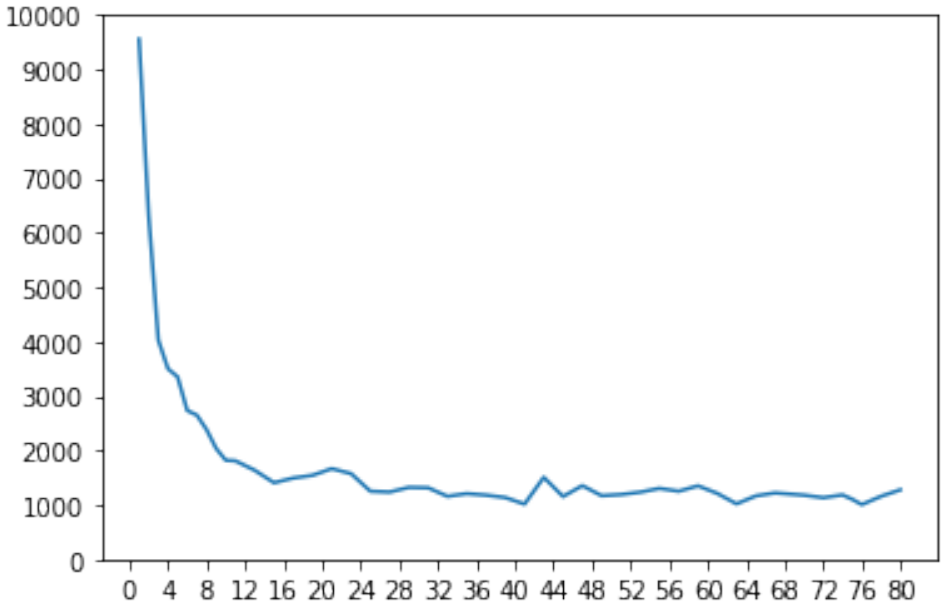

Figura 15 – Distância máxima percorrida por um usuário por quantidade de estações.

A Figura 16 a seguir representa as posições das estações no SA e no PSO. Em vermelho estão os pontos criados pelo PSO e em azul as estações criadas pelo SA. Como era de se esperar as áreas mais povoadas apresentaram uma concentração maior de estações.

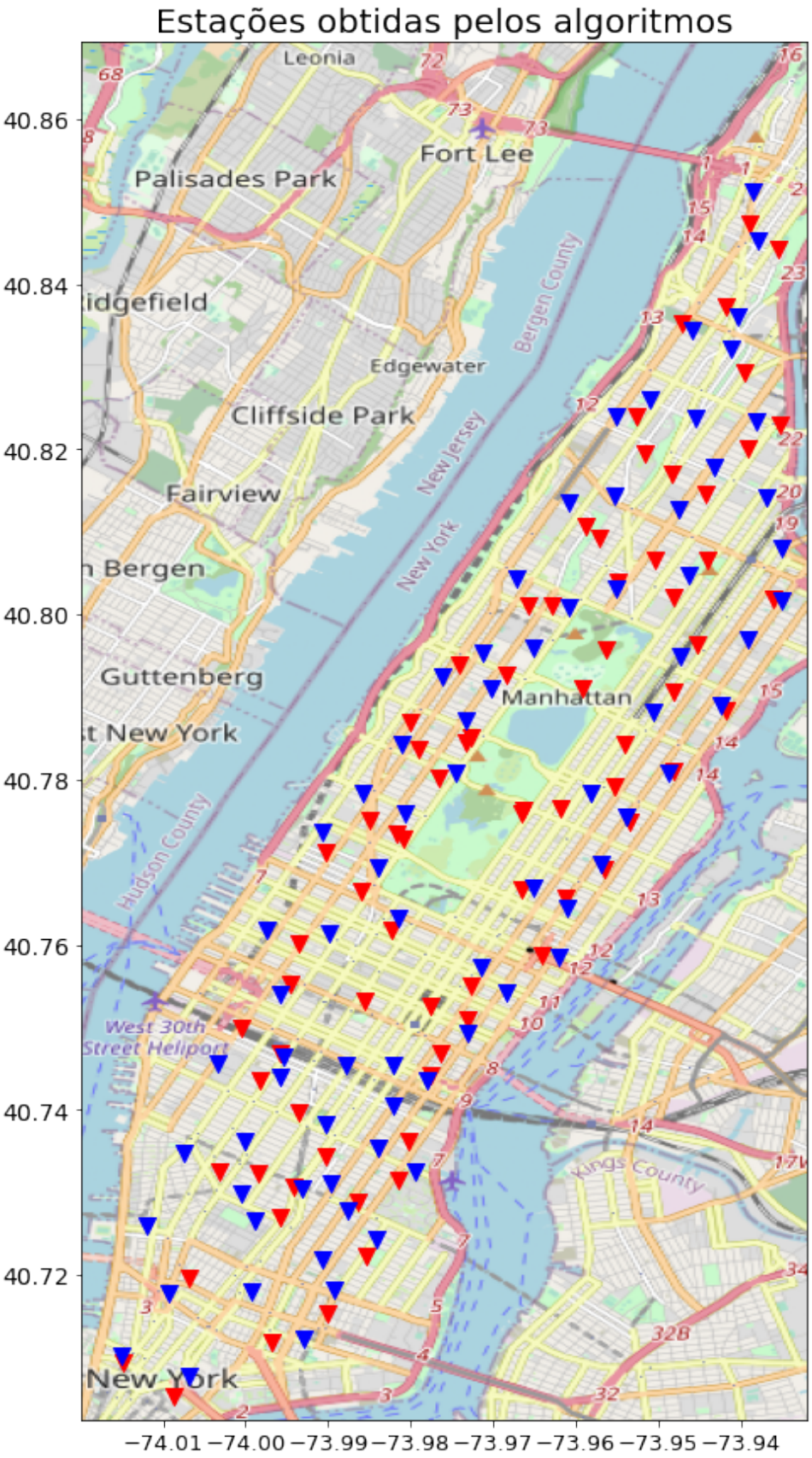

Figura 16 – Estações obtidas pelos métodos SA (em azul) e PSO (em vermelho).

Além das posições finais, é interessante estudar a procura dos melhores pontos realizada

pelo algoritmo. Conforme a Figura 17, pode-se ver como que as coordenadas de uma estação se alteram ao longo das iterações. No início do recozimento, enquanto a temperatura ainda está elevada, as estações costumam explorar todo domínio das abscissas e ordenadas de forma a minimizar o resultado da função objetiva e por conta do fator de aleatoriedade contido no algoritmo do SA em que para temperaturas elevadas existe uma chance maior de aceitar um resultado superior ao anterior (Apêndice B). À medida que a temperatura diminui, os pontos da estação deixam de explorar e passam a refinar as respectivas posições, até encontrarem por fim o mínimo global.

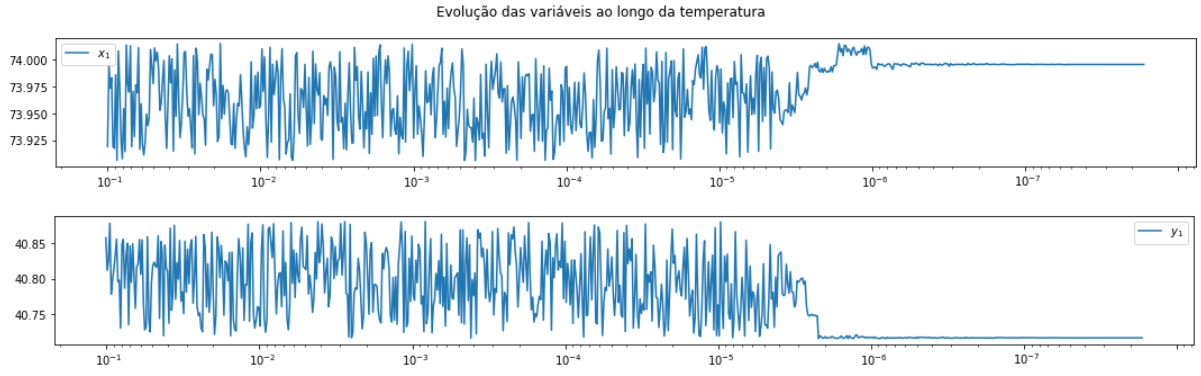

Figura 17 – Posição nas abscissas e ordenadas de uma estação a cada temperatura (escala logarítmica).

Um resultado semelhante acontece para os valores da função objetiva apresentados nas Figuras 18 e 19. Enquanto a temperatura está elevada é normal que soluções não ótimas sejam aceitas, ocasionando os pontos de máximo da figura. Conforme a temperatura diminui, a função converge para o ponto de mínimo.

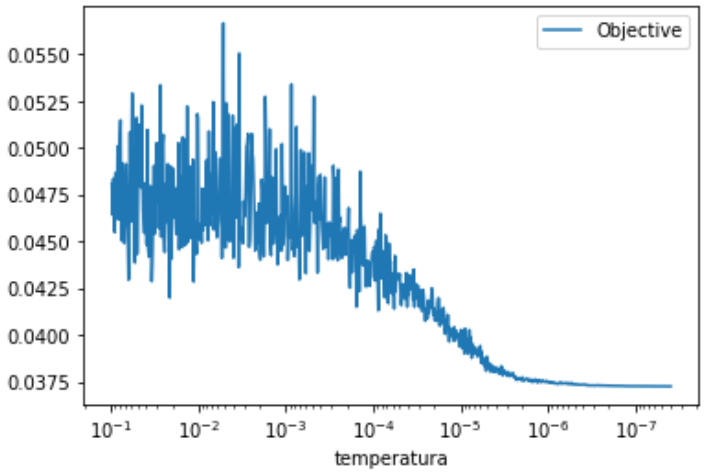

Figura 18 – Valores da função objetiva pela temperatura (log) .

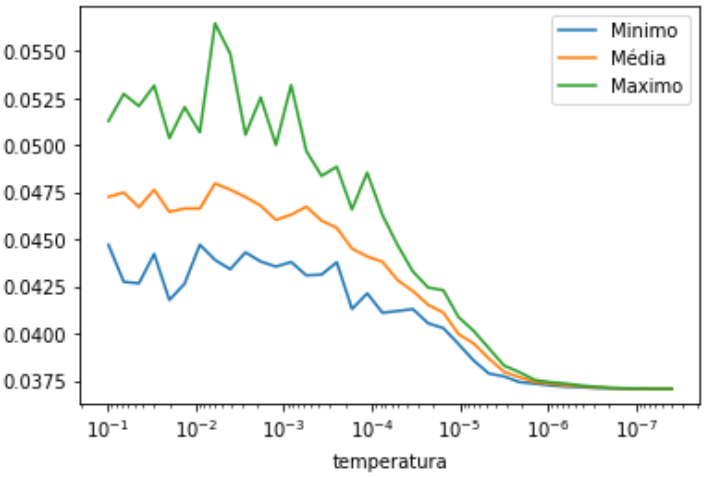

Figura 19 – Máximo, mínimo e média da função objetiva pela temperatura

O fator de cristalização também pode ser observado ao longo das iterações nas Figuras 20 e 21. Conforme o número de rejeitados aumenta, o fator de cristalização em cada eixo tende a aumentar, garantindo um refino maior da solução final.

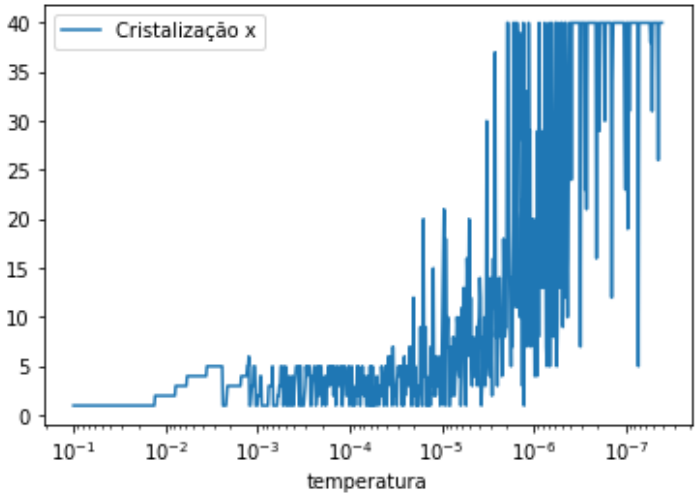

Figura 20 – Fator de cristalização para o eixo x da estação.

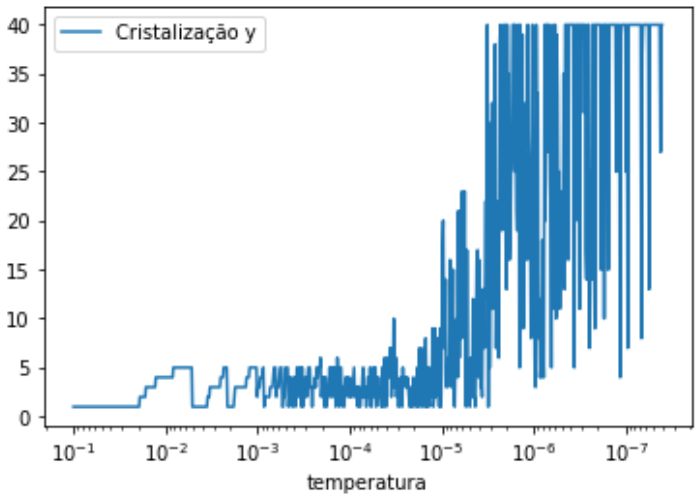

Figura 21 – Fator de cristalização para o eixo y da estação.

O critério de parada para o SA ocorre quando o número de opções aceitas chega a zero dentro de um espaço de opções avaliadas, portanto, espera-se que o algoritmo tenha atingido o mínimo global nesse momento. As Figuras 22 e 23 retratam que, no início, praticamente todas soluções são aceitas e conforme a temperatura diminui algumas soluções são rejeitadas, aumentando a quantidade de soluções avaliadas.

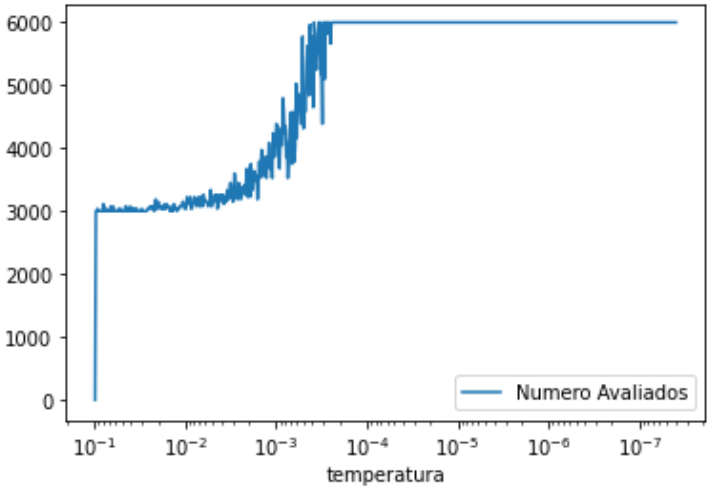

Figura 22 – Quantidade de soluções avaliadas ao longo das iterações.

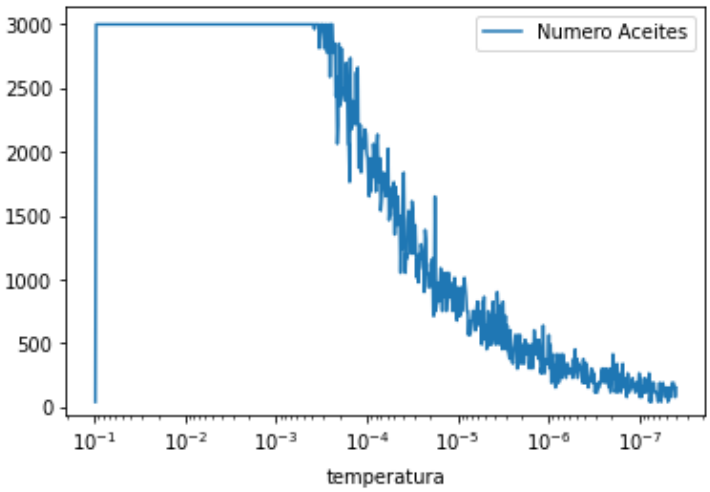

Figura 23 – Quantidade de soluções aceitas ao longo das iterações.

No caso do PSO, a única forma de se aceitar um novo resultado é quando o resultado testado possui um valor inferior ao global. Por conta disso o gráfico da função objetiva é decrescente (Figura 24) e tende a convergir de forma mais rápida do que o SA.

Uma consequência desse algoritmo é a tendência de serem aceitos mínimos locais, ao invés de mínimos globais. No problema em questão, podemos observar que os valores da função objetiva acabam por ser ligeiramente diferentes e a posição das estações estão mais distantes umas das outras em alguns casos.

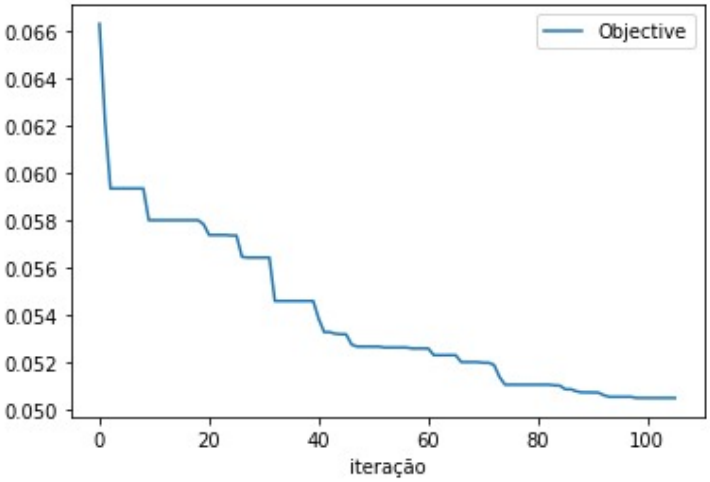

Figura 24 – Valores da função objetiva para cada iteração.

Essa característica pouco exploratória do PSO se reflete nas posições das possíveis estações ao longo das iterações do programa. Como podemos observar na Figura 25, as coordenadas dessa estação em cada iteração alteraram pouco quando comparadas com as do método SA (Figura 17) e também convergiram a um ponto de maneira muito mais rápida.

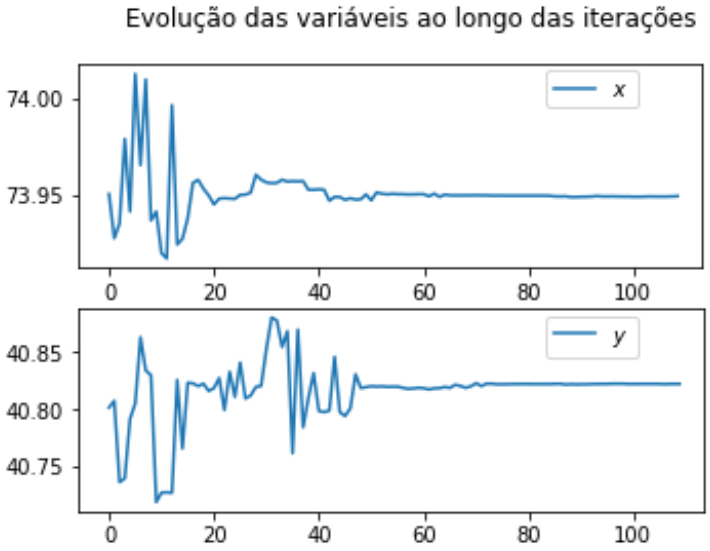

Figura 25 – Posição nas abscissas e ordenadas de uma estação ao longo das iterações do programa PSO.

6.2 NÚMERO DE BICICLETAS POR ESTAÇÃO

Por meio dos dados de frequência de utilização da ilha de Manhattan foi possível fazer simulações a fim de obter a quantidade inicial ideal de bicicletas em cada estação, assim como a quantidade total de *slots* para um período de utilização de 24 horas.

Por meio de um processo iterativo, estações onde haviam lotações ou falta de bicicletas tiveram seu número inicial incrementado ou reduzido em lotes de 5 bicicletas. Esse processo foi então repetido até que as estações pudessem suportar um dia em operação sem que houvesse problemas de falta ou sobrecarga. Com isso, numa situação prática, as estações somente precisariam ser reabastecidas ou remanejadas uma vez por dia, por volta das 05:00 da manhã o qual é um horário de pouquíssimo uso (Figura 7). Os resultados para cada estação podem ser observados nas tabelas abaixo para Manhattan (Tabela 4) e para USP (Tabela 5).

Tabela 4: Disposição das bicicletas por estação em Manhattan.

Nº Estação	Qtd. Mínima	Qtd. Total	Nº Estação	Qtd. Mínima	Qtd. Total
1	15	25	39	20	30
2	20	30	40	25	35
3	20	30	41	20	30
4	20	30	42	20	30
5	25	35	43	25	35
6	20	30	44	25	35
7	20	30	45	25	35
8	15	25	46	15	25
9	20	30	47	25	35
10	25	35	48	25	35
11	20	30	49	15	25
12	25	35	50	25	35
13	20	30	51	15	25
14	25	35	52	25	35
15	25	35	53	25	35
16	25	35	54	25	35
17	15	25	55	20	30
18	25	35	56	30	55
19	30	40	57	15	25
20	25	35	58	10	20
21	25	35	59	15	25
22	15	25	60	15	25
23	20	30	61	20	30
24	30	40	62	25	35
25	25	35	63	15	25
26	25	35	64	30	50
27	20	30	65	20	30
28	20	30	66	20	30
29	20	30	67	15	25
30	25	35	68	20	30
31	20	30	69	25	35
32	20	30	70	25	35
33	10	20	71	20	30
34	15	25	72	25	35
35	25	35	73	30	40
36	20	30	74	20	30
37	25	35	75	20	30
38	20	30			

Tabela 5: Disposição das bicicletas por estação na USP.

Nº Estação	Qtd. Mínima	Qtd. Total
1	25	35
2	25	35
3	30	40
4	35	45
5	20	30
6	10	20
7	25	35
8	20	30
9	15	25
10	30	40
11	20	30
12	10	20
13	20	30
14	10	20
15	30	40
16	30	40
17	35	45
18	15	25

6.3 SIMULAÇÃO PROCESSING PARA MANHATTAN

Possuindo as posições das estações obtidas com os métodos anteriores, as frequências de uso nas estações e a distribuição normal de velocidade das viagens, é possível simular as viagens dentro da ilha de Manhattan, como pode ser observado na Figura 26 abaixo.

Utilizaram-se as estações obtidas pelo SA para o caso. Inicialmente foram feitas simulações com quantidades infinitas de bicicletas por estação e capacidade infinita. Ao longo das simulações, foi possível refinar o número de bicicletas e a capacidade para cada estação.

Cada círculo vermelho, preenchido estático no mapa, representa uma estação, e o número logo acima, a quantidade de bicicletas na estação a cada momento. Os círculos menores representam bicicletas em viagem.

Na Figura 26, está representada uma simulação de uma hora, em que cada estação tem uma probabilidade de enviar uma bicicleta a cada minuto de acordo com a frequência obtida. Existem, portanto, momentos em que nenhuma bicicleta parte da estação, sendo cada simulação única. Como podem sair bicicletas até o último minuto, a simulação pode ultrapassar o tempo de uma hora. Também foram feitas simulações de 10 horas, sabendo que existissem bicicletas em todas estações a todo momento. Para visualizar clique aqui, ou acesse pelo seguinte link: <https://youtu.be/dMB6ZW36Esc>.

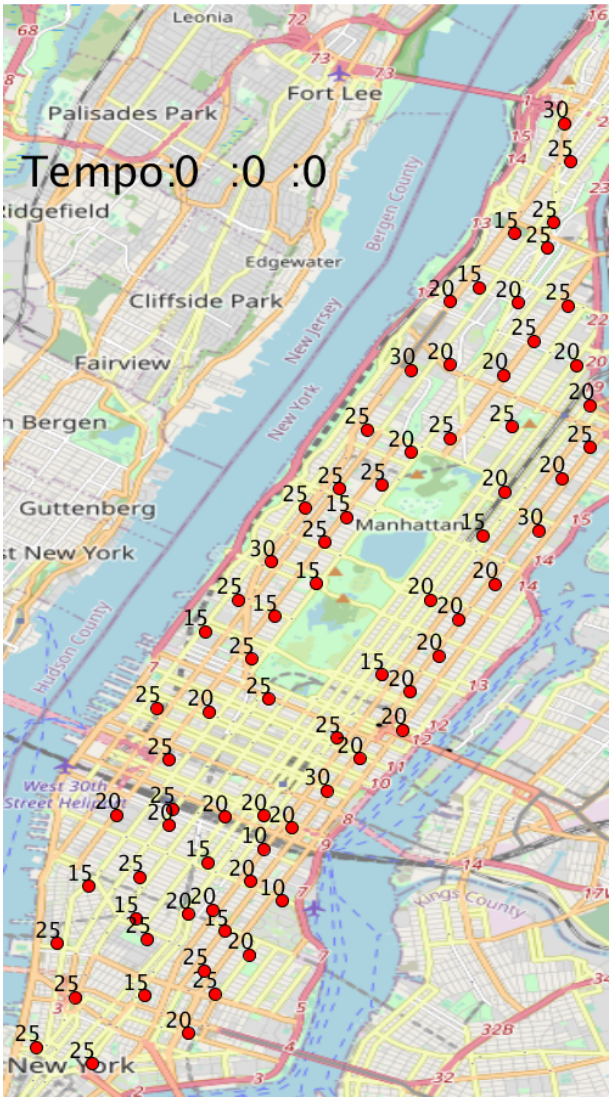

Figura 26 – Início da Simulação de uma hora utilizando o Processing.

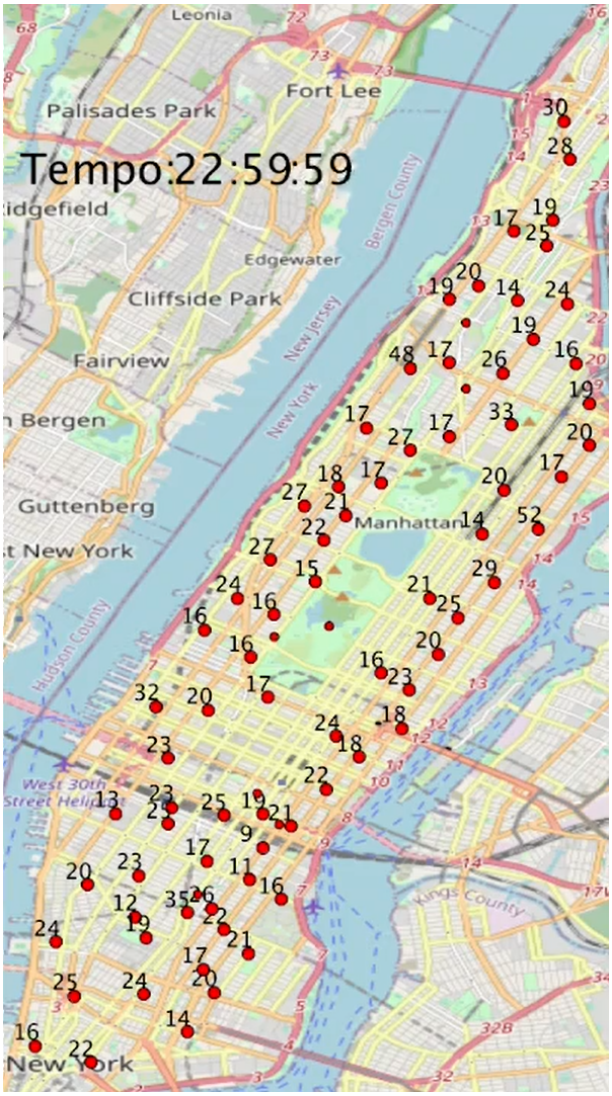

Figura 27 – Fim da Simulação de uma hora utilizando o Processing.

6.4 SIMULATED ANNEALING E PARTICLE SWARM OPTIMIZATION PARA USP

No caso da cidade universitária, foram feitas as simulações de acordo com os pontos de demanda da Tabela 2, para pontos com número relevantes de usuários. Foram, igualmente, adicionados pontos de demanda (Figura 28) na entrada da CPTM, que se encontra perto do Portão 1, e no bandejão central devido à demanda em horários específicos de entrada, e de saída e de refeições, respectivamente.

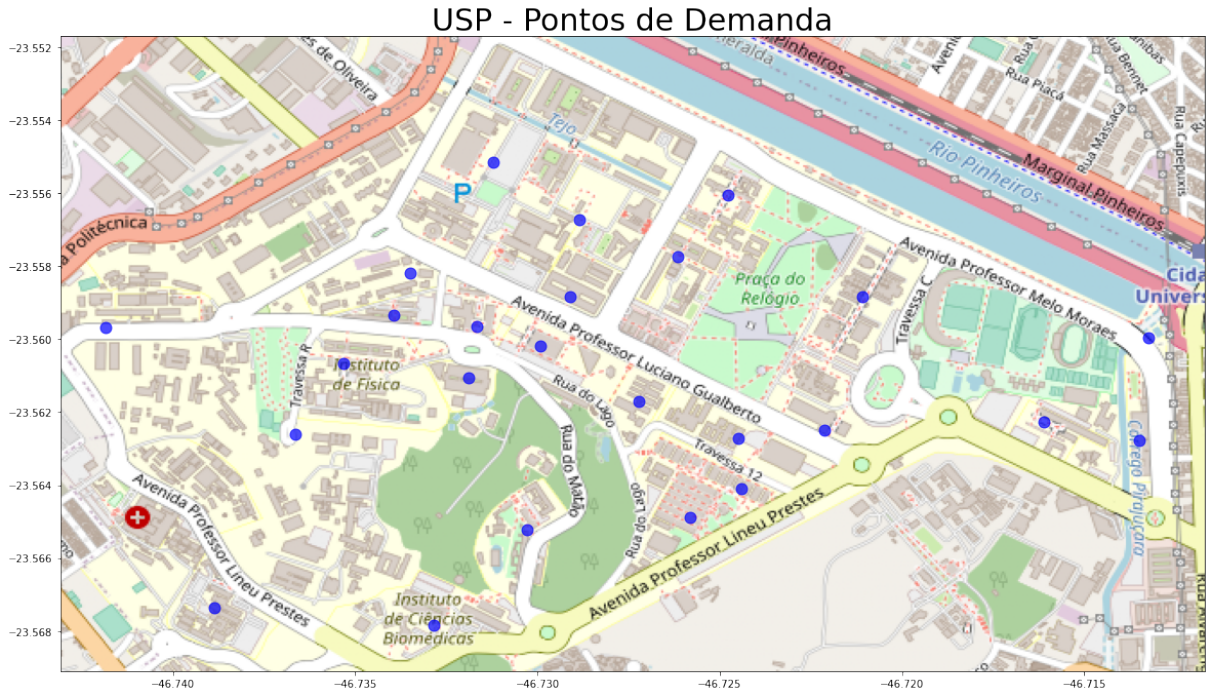

Figura 28 – Pontos de demanda estabelecidos para o campus da USP em azul.

O projeto da USP é diferente do de Nova York, pois deseja-se obter um número de estações suficientes para a alta demanda da USP concentrada principalmente nos horários de pico, e em regiões específicas. Foram utilizadas 18 estações para as simulações dos algoritmos, o qual é o mesmo número de estações já presentes no campus.

Observando o gráfico que adicionando 18 estações obtém-se uma distância de 673 metros (Figura 29). Isso quer dizer que no máximo uma pessoa terá que andar 673 metros para chegar em uma estação. Portanto optou-se por manter 18 estações garantindo a otimização de quantidade de estações pela região, evitando um acúmulo de estações de forma desnecessária.

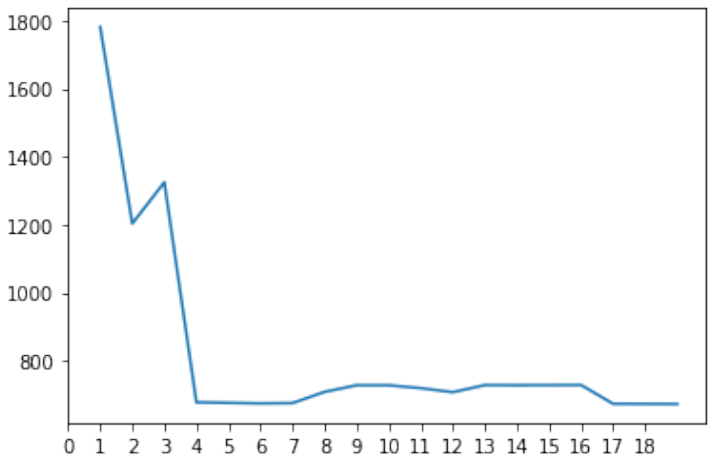

Figura 29 – Distância máxima percorrida por quantidade de estações.

Tabela 6: Comparação dos métodos utilizados para USP (SA e PSO).

	SA	PSO
Entradas		
Nº de estações	18	18
Nº iteração Máxima	-	1000
Nº Aceites Máximo	720	-
Nº Avaliados Máximo	1440	-
Fator de Cristalização máximo	40	-
Temperatura Inicial	10	-
Nº de partículas	-	3000
W (Constante de inércia)	-	0.1
c1 (Constante cognitiva)	-	2.5
c2 (Constante social)	-	3
Critério de parada	Nº Aceites $= 0$	Nº iteração Máxima OU Atual/Melhor < 0.99
Comparações		
Tempo de processamento (s)	1351.63	304.25
Nº de iterações	931	90
Solução função objetiva	0.00089187	0.00181476
Δ Resíduo	0.00092289	

Podemos observar pela Tabela 6 e pela Figura 30 que os resultados, utilizando-se o método do SA, foram mais precisos do que quando empregando o método do PSO, mesmo que esse seja mais rápido e menos computacionalmente custoso. Os gráficos de exploração e refino mostrados para Manhattan são muito semelhantes aos obtidos para USP, uma vez que retratam o funcionamento dos algoritmos e por isso não foram adicionados novamente nesta seção.

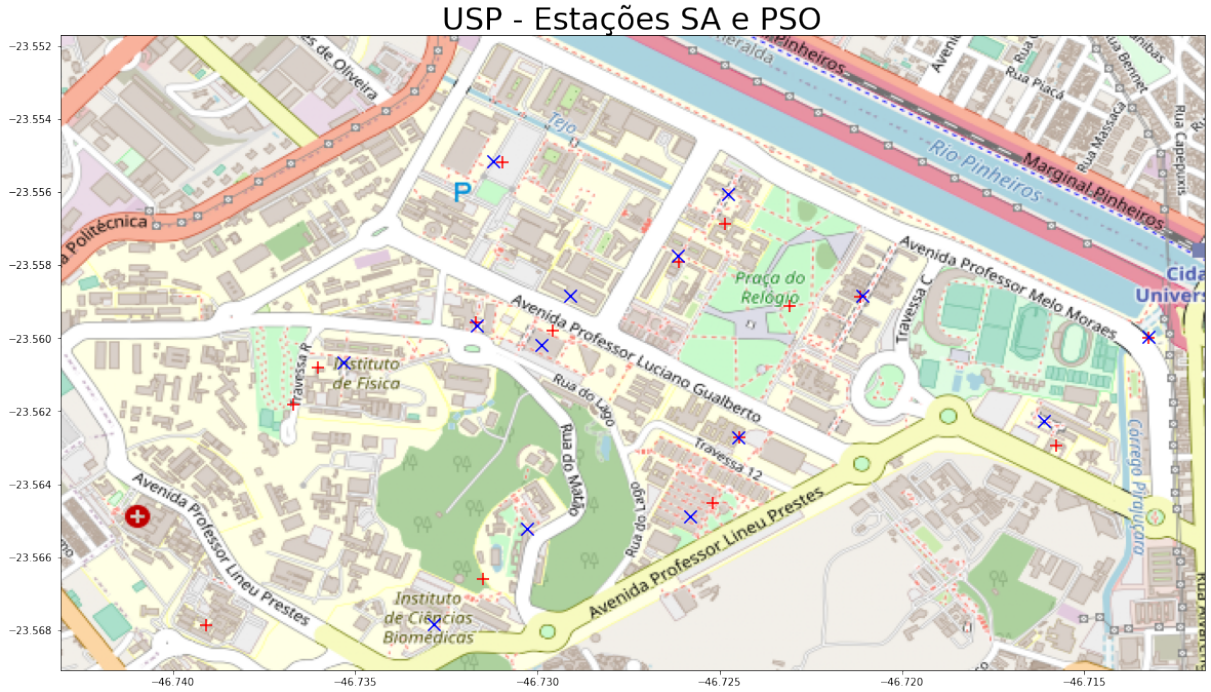

Figura 30 – Estações definidas pelo SA (azul) e PSO (vermelho).

6.5 SIMULAÇÃO PROCESSING PARA USP

A simulação foi feita com base em um dia útil da universidade com início às 6 horas da manhã e término às 20 horas, considerando-se que nesse instante quase todos cursos já encerraram as suas atividades e a demanda pelas bicicletas após esse horário é bem escassa. Para ver o vídeo clique aqui, ou acesse pelo seguinte link: <https://youtu.be/4OJPrSjMzlo>.

As frequências de utilização em cada estação foram retiradas da ilha de Manhattan, uma vez que foi feito uma aproximação da demanda considerando a quantidade de pessoas nos pontos de demanda comparativamente com os das bases de estudo.

Utilizando a função objetiva especificada anteriormente (Eq. 1), com as estações da Tembici obtivemos um resultado de 0.00467201, um valor aproximadamente cinco vezes maior do que pelo SA. Comparando-se as localizações das estações (Figura 31) podemos ver que nas regiões com maior quantidade de alunos, como a Avenida Luciano Gualberto, existem mais estações. Nas regiões situadas nas extremidades do campus, como na Avenida Professor Lineu Prestes, o número de estações diminui consoante a redução de cursos universitários situados próximos a esse local.

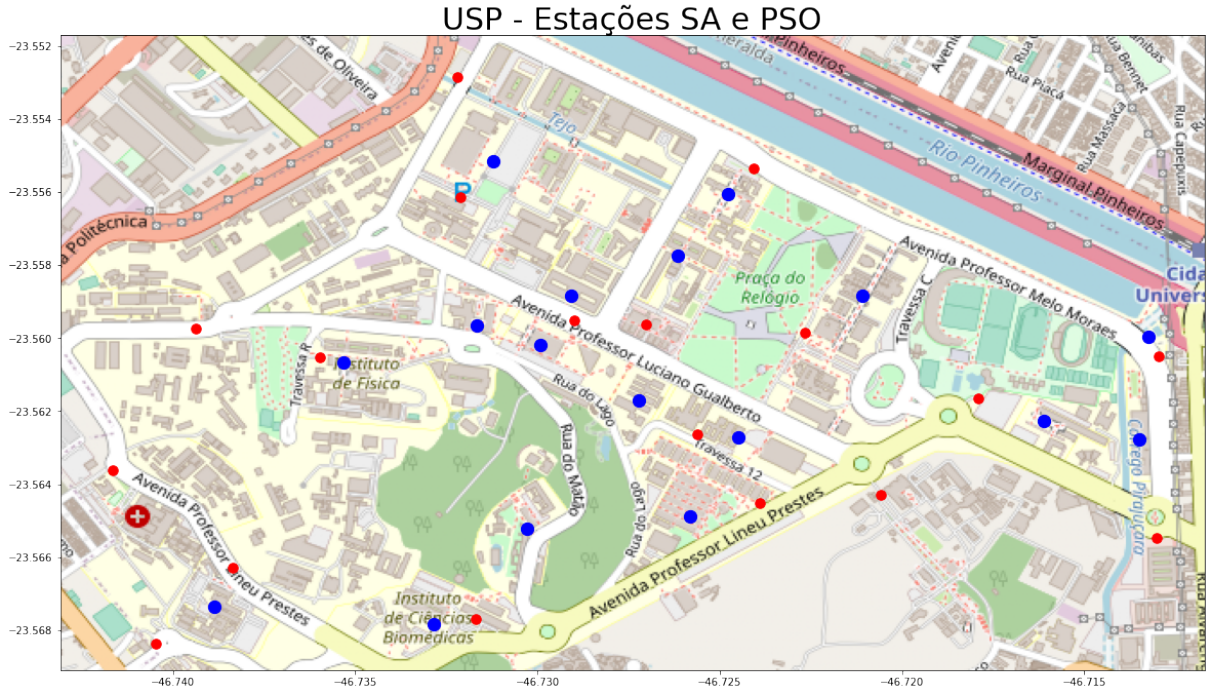

Figura 31 – Estações definidas pelo SA (azul) e estações da Tembici (vermelho).

Realizando uma comparação com os números de bicicletas em cada estação ao final do período de utilização, com as mesmas frequências utilizadas para as estações mais próximas obtidas no SA anteriormente e mantendo o limite máximo de *slots* presentes nas estações da Tembici obtivemos o resultado mostrado na Figura 32. Pode-se perceber que as estações de maiores demanda, como as entradas da universidade apresentam valores negativos ou nulos, mostrando que para essa simulação faltaram bicicletas livres nas respectivas estações.

Figura 32 – Resultado final da simulação para a Tembici.

7 CONCLUSÃO

O problema de localização de estações de bicicletas compartilhadas é de grande relevância para o transporte urbano de grandes cidades e está em linha com as medidas sustentáveis adotadas em diversos países, diminuindo a lotação de ruas, trazendo maior rapidez para transporte de pequenas distâncias, reduzindo a emissão de gases de efeito estufa, reduzindo a poluição sonora e disponibilizando um elemento de lazer individual e coletivo.

Em um primeiro momento, obtiveram-se dados abertos referentes ao sistema de transporte de bicicletas da cidade de Nova York, assim como os dados sobre a população dos bairros da cidade. Com isso, já foi possível modelar o problema, utilizando-se os algoritmos mencionados anteriormente, de resolução do problema de p-mediana.

Foram utilizados dois algoritmos para obtenção das localizações das estações. O SA apresentou resultados superiores ao PSO, com um valor de resíduo inferior e com melhor distribuição das estações tanto para o caso de Manhattan quanto para a USP. Entretanto, ele teve oito vezes mais iterações e um tempo vinte vezes superior ao PSO, mostrando um custo computacional elevado. Tendo em vista os altos níveis de exploração mostrados no SA e a habilidade de evitar pontos de mínimo locais esse foi o escolhido, obtendo-se 75 estações para a ilha de Manhattan e 18 para a cidade universitária da USP.

Após termos obtido as localizações das estações, foi feita uma simulação gráfica de uma possível utilização das bicicletas em cada estação, por meio das frequências extraídas das bases de dados. Com essa simulação, foi possível observar e corrigir estações que poderiam ficar sem bicicletas e outras que, ao contrário, estariam superlotadas impedindo a devolução.

Foram realizadas simulações de Manhattan de uma e de dezessete horas, enquanto que para a USP tivemos simulações de uma e catorze horas, além da simulação das estações já existentes da Tembici.

Analisando os resultados das simulações realizadas, também foi possível fazer uma estimativa do número de bicicletas ideal para cada uma das estações. Em estações com problemas de lotação ou de falta de bicicletas, seus números de bicicletas foram atualizados em um processo iterativo, a fim de reduzir tais dificuldades.

Os algoritmos testados poderiam ser utilizados na implementação de novos sistemas de bicicletas compartilhadas. Aliados a pesquisa e a uma sólida base de dados, eles poderiam ser essenciais para redução de estações inutilizadas e redundantes, ou na previsão de estações com problemas de superlotação. Mais do que corrigir erros encontrados, esse sistema de otimização visa a prevenção de erros na implementação de sistemas similares em outras regiões ou cidades.

Referências

ABDELLAOUI ALAOUI, E. A.; KOUMETIO TEKOUABOU, S. C. Intelligent management of bike sharing in smart cities using machine learning and internet of things. **Sustainable Cities and Society**, v. 67, p. 102702, 2021. ISSN 2210-6707. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2210670720309161>. Citado na página 3.

BERGER, R. **Bike Sharing: Cornerstone of future urban mobility**. [S.l.], 2008. Citado na página 2.

CAGGIANI, L. et al. User satisfaction based model for resource allocation in bike-sharing systems. **Transport Policy**, v. 80, p. 117–126, 2019. ISSN 0967-070X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0967070X17305322>. Citado na página 3.

CENSUS. **New York City 2020 CENSUS**. 2020. Disponível em: <https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.2020.html>. Acesso em: 20 de julho de 2021. Citado na página 20.

CHIBWE, J. et al. An exploratory analysis of the trend in the demand for the london bike-sharing system: From london olympics to covid-19 pandemic. **Sustainable Cities and Society**, v. 69, p. 102871, 2021. ISSN 2210-6707. Disponível em: <https://www.sciencedirect.com/science/article/pii/S221067072100161X>. Citado na página 4.

CINTRANO, C.; CHICANO, F.; ALBA, E. Using metaheuristics for the location of bicycle stations. **Expert Systems with Applications**, v. 161, p. 113684, 2020. ISSN 0957-4174. Disponível em: <https://www.sciencedirect.com/science/article/pii/S095741742030508X>. Citado na página 4.

COSTA, C. E. d. S. Aplicação de técnicas de pesquisa operacional na determinação de setores de atendimento em uma concessionária de energia. p. 146 f., 2005. Citado na página 5.

DOKUZ, A. S. Fast and efficient discovery of key bike stations in bike sharing systems big datasets. **Expert Systems with Applications**, v. 172, p. 114659, 2021. ISSN 0957-4174. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0957417421001007>. Citado na página 4.

DREZNER, Z. et al. New heuristic algorithms for solving the planar p-median problem. **Computers and Operations Research**, Elsevier, v. 62, p. 296–304, October 2015. Full text complies with journal regulations. Disponível em: <https://kar.kent.ac.uk/51102/>. Citado na página 8.

FRY, B.; REAS, C. **Processing Environment Description**. 2001. Disponível em: <https://processing.org/environment>. Acesso em: 20 de novembro de 2021. Citado na página 23.

HAKIMI, S. L. Optimum locations of switching centers and the absolute centers and medians of a graph. **Operations Research**, INFORMS, v. 12, n. 3, p. 450–459, 1964. ISSN 0030364X, 15265463. Disponível em: <http://www.jstor.org/stable/168125>. Citado na página 5.

HUDAIB, A.; HWAITAT, A. A. Movement particle swarm optimization algorithm. **Modern Applied Science**, v. 12, p. 148, 12 2017. Citado na página 7.

IBE, O. C. 2 - basic concepts in stochastic processes. In: IBE, O. C. (Ed.). **Markov Processes for Stochastic Modeling (Second Edition)**. Second edition. Oxford: Elsevier, 2013. p. 29–48. ISBN 978-0-12-407795-9. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780124077959000025>. Citado na página 8.

KENNEDY, J.; EBERHART, R. **Swarm Intelligence**. San Mateo, CA: Morgan Kaufmann., 2001. Citado na página 7.

KENNEDY, R. E. J. Particle swarm optimization. in proceedings of icnn'95-international conference on neural networks. **IEEE**, v. 4, p. 1942–1948, 1995. Citado na página 20.

KIRKPATRICK, S.; JR., C. D. G.; VECCHI, M. P. Optimization by simulated annealing. **Science**, v. 220, n. 4598, p. 671–680, 1983. Cited By :27951. Disponível em: <www.scopus.com>. Citado na página 20.

KLOIMüLLNER, C.; RAIDL, G. Hierarchical clustering and multilevel refinement for the bike-sharing station planning problem. In: . [S.l.: s.n.], 2017. p. 150–165. ISBN 978-3-319-69403-0. Citado na página 3.

LIBERMAN, R. H. . **Introdução à Pesquisa Operacional**. [S.l.]: McGraw Hill, 2006. Citado na página 9.

NOGUEIRA, F. **Modelagem e Simulação - Cadeias de Markov**. 2017. Disponível em: <https://www.ufjf.br/epd042/files/2009/02/cadeiaMarkov1.pdf>. Acesso em: 26 de novembro de 2021. Citado 2 vezes nas páginas 8 e 9.

NORRIS, J. **Markov Chains**. 2021. Disponível em: <http://www.statslab.cam.ac.uk/~rrw1/markov/M.pdf>. Acesso em: 5 de dezembro de 2021. Citado 2 vezes nas páginas e 9.

NTA. **New York City 2010 Neighborhood Tabulation Areas**. 2010. Disponível em: <http://www.infoshare.org/main/Neighborhood%20Tabulation%20Areas%20(NTAs).pdf>. Acesso em: 9 de julho de 2021. Citado na página 20.

NYC, C. **CitiBike New York City Data**. 2013. Disponível em: <https://www.citibikenyc.com/system-data>. Acesso em: 28 de abril de 2021. Citado na página 10.

OPEN-DATA, N. **NYC Open Data**. 2020. Disponível em: <https://data.cityofnewyork.us/City-Government/New-York-City-Population-By-Neighborhood-Tabulatio/swpk-hqdp>. Acesso em: 28 de maio de 2021. Citado na página 10.

PAPAZEK, P. et al. A pilot/vnd/grasp hybrid for the static balancing of public bicycle sharing systems. In: . [S.l.: s.n.], 2013. p. 372–379. ISBN 978-3-642-53855-1. Citado na página 3.

SCHUIJBROEK, J.; HAMPSHIRE, R.; van Hoeve, W.-J. Inventory rebalancing and vehicle routing in bike sharing systems. **European Journal of Operational Research**, v. 257, n. 3, p. 992–1004, 2017. ISSN 0377-2217. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0377221716306658>. Citado na página 3.

SILVA, Y.; MESTRIA, M. Algoritmos para o problema de localização de estações de carregamento de veículos elétricos. **Revista Produção Online**, v. 19, p. 290–320, 03 2019. Citado na página 5.

STEINER, M. T. A. **Programa de PósGraduação em Métodos Numéricos em Engenharia**. [S.l.], 2003. Citado na página 5.

TEMBICI. **Bicicletas compartilhadas**. 2020. Disponível em: <https://tembici.com.br/bicicletas-compartilhadas>. Acesso em: 9 de junho de 2021. Citado na página 1.

TRAGANTALERNGSAK, S.; RöNNQVIST, M. Exact method for the two-echelon, single-source, capacitated facility location problem. **European Journal of Operational Research**, v. 123, p. 473–489, 06 2000. Citado na página 5.

TROSSET, M. What is simulated annealing? **Optimization and Engineering**, v. 2, p. 201–, 06 2001. Citado na página 6.

USP. **Anuário Estatístico USP**. VREA/USP, 2020. Disponível em: <https://uspdigital.usp.br/anuario/br/acervo/AnuarioUSP_2020.pdf>. Acesso em: 10 de outubro de 2021. Citado 3 vezes nas páginas , 18 e 19.

VOGEL, P.; GREISER, T.; MATTFELD, D. C. Understanding bike-sharing systems using data mining: Exploring activity patterns. **Procedia - Social and Behavioral Sciences**, v. 20, p. 514–523, 2011. ISSN 1877-0428. The State of the Art in the European Quantitative Oriented Transportation and Logistics Research – 14th Euro Working Group on Transportation & 26th Mini Euro Conference & 1st European Scientific Conference on Air Transport. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877042811014388>. Citado na página 3.

YANG, Y.; JIANG, L.; ZHANG, Z. Tourists on shared bikes: Can bike-sharing boost attraction demand? **Tourism Management**, v. 86, p. 104328, 2021. ISSN 0261-5177. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0261517721000479>. Citado na página 3.

Apêndices

APÊNDICE A – Tratamento Inicial dos Dados

Código referente à análise e tratamento inicial dos dados retirados da cidade de Nova Iorque.

```
import pandas as pd
import matplotlib.pyplot as plt
import os
import io
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import missingno as mn
import seaborn as sns
from sklearn import preprocessing

#Data from NYC
df2 = pd.read_csv("dados.csv")


columns=['tripduration','start_station_id','start_station_latitude',
'start_station_longitude','end_station_id','end_station_latitude',
'end_station_longitude','bikeid','birth_year','gender']

x = df2[columns] #returns a numpy array

min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
df3 = pd.DataFrame(x_scaled,columns=columns)
df3.head()

# checking for incompletness of data
mn.matrix(df2.sample(500), figsize=(10,6))

#Dendrogram
mn.dendrogram(df2, figsize=(10,6))
mn.bar(df2, figsize=(10,6))

#histograms
```

```
df2.hist(bins=50, figsize=(20,15))

#heatmap and correlation
sns.heatmap(df3.corr());
df2.corr()

#Finding Outliers
sns.set_context("notebook", font_scale=1.1)
sns.set_style("ticks")
sns.lmplot('birth_year', 'tripduration',
           data=df2,
           fit_reg=True,
           scatter_kws={"marker": "D",
                        "s": 10})
plt.show()

def get_iqr_values(df_in, col_name):
    median = df_in[col_name].median()
    q1 = df_in[col_name].quantile(0.25) # 25th percentile / 1st quartile
    q3 = df_in[col_name].quantile(0.75) # 7th percentile / 3rd quartile
    iqr = q3-q1 #Interquartile range
    minimum  = q1-1.5*iqr
    # The minimum value or the |- marker in the box plot
    maximum = q3+1.5*iqr
    # The maximum value or the -| marker in the box plot
    return median, q1, q3, iqr, minimum, maximum

def get_iqr_text(df_in, col_name):
    median, q1, q3, iqr, minimum, maximum = get_iqr_values(df_in, col_name)
    text = f"median={median:.2f}, q1={q1:.2f}, q3={q3:.2f},
    iqr={iqr:.2f}, minimum={minimum:.2f}, maximum={maximum:.2f}"
    return text

def remove_outliers(df_in, col_name):
    _, _, _, _, minimum, maximum = get_iqr_values(df_in, col_name)
    df_out = df_in.loc[(df_in[col_name] > minimum) & (df_in[col_name] < maximum)]
    return df_out

def count_outliers(df_in, col_name):
```

```
    _, _, _, _, minimum, maximum = get_iqr_values(df_in, col_name)
    df_outliers = df_in.loc[(df_in[col_name] <= minimum) |
    (df_in[col_name] >= maximum)]
    return df_outliers.shape[0]

def box_and_whisker(df_in, col_name):
    title = get_iqr_text(df_in, col_name)
    sns.boxplot(df_in[col_name])
    plt.title(title)
    plt.show()

# removendo outliers segundo a regra dos quartis
box_and_whisker(df2, 'tripduration')
print(f"tripduration has {count_outliers(df2, 'tripduration')} outliers")

df2 = remove_outliers(df2, 'tripduration')
box_and_whisker(df2, 'tripduration')

sns.set_context("notebook", font_scale=1.1)
sns.set_style("ticks")
sns.lmplot('birth_year', 'tripduration',
           data=df2,
           fit_reg=True,
           scatter_kws={"marker": "D",
                        "s": 10})
plt.show()
```

APÊNDICE B – Código Simulated Annealing

Código possui duas entradas diferentes, uma para o caso da ilha de Manhattan e outra para a USP.

```
#Módulos importados

import pandas as pd
from numpy import exp
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed
from matplotlib import pyplot as plt
import numpy as np
import random
import time
from __future__ import division
import math
import copy    # array-copying convenience
import sys     # max float

#Funções auxiliares
# objective function
def objective(x, y, pontos_demanda):
  ''' Função objetiva do algoritmo de SA. Recebe três vetores contendo os
  pontos de demanda da região que está sendo estudada,
  as posiçoes x e y de cada ponto que foi determinado pelo algoritmo de SA.
  Ela retorna a soma que corresponde à soma do mínimo entre às distâncias dos
  pontos do algoritmo e dos pontos de demanda.
  Também é retornado o vetor de todas distâncias para análise.
  '''
  soma=0
  dist2=[]
  for n in range(len(pontos_demanda)):
  #para cada ponto de demanda verifca-se todos pontos criados pelo algoritmo
    dist=[]
    for i in range(len(x)):
      dist.append(euclidean(x[i],y[i],pontos_demanda[n][0],pontos_demanda[n][1]))
    soma+=min(dist)*pontos_demanda[n][2]
```

```
    dist2.append(dist)

  return soma,dist2

def euclidean(ai,aj,bi,bj):
  '''
  Função que retorna a distância euclidiana entre dois pontos em um espaço 2d.
  As entradas são as posições x e y (i e j) de dois pontos a e b.
  '''
  return np.sqrt((ai-bi)**2 + (aj-bj)**2)

#Rever se precisa

def normalize(x):
  ''' função para normalizar os pesos dos pontos de demanda.
  '''
  novo=0
  for i in range(len(x)):
    novo+=x[i]**2
  xnovo=x/np.sqrt(novo)
  return xnovo



def valores_iniciais(N_stations,bounds,best):
  '''Função responsável por gerar de forma aleatória o vetor com
  as estimativas dos pontos iniciais do SA.
  Recebe o número de estações que serão geradas, um vetor com os limites
  em latitude e longitude da região estudada e um vetor vazio best.
  '''
  best2=best.copy()

  for p in range(N_stations):
    #escolhe um x e m y de maneira randomica
    #desde que esteja dentro dos limites de latitude e longitude
    best2[0][p]=np.float64(random.randrange(round(bounds[0][0]*1000000),
    round(bounds[0][-1]*1000000))/1000000).item()
    best2[1][p]=np.float64(random.randrange(round(bounds[1][0]*1000000),
    round(bounds[1][-1]*1000000))/1000000).item()
```

```
  return best2
def normalize(x):
  ''' Função para normalizar os pesos em cada ponto de demanda.
  '''
  novo=0
  for i in range(len(x)):
    novo+=x[i]**2
  xnovo=x/np.sqrt(novo)
  return xnovo
def soma_valores(best,step_size,bounds,c):
  '''
  Nessa função acontece a atualização dos valores obtidos no algoritmo.
  recebemos o vetor best com as posições atuais do SA que foram aceitas,
  o step_size para determinar o acrescimo na posição de cada ponto,
  vetor com os limites da região, bound
  e o vetor c com o fator de cristalização.
  Retorna o novo vetor best2 com os novos valores a serem testados pelo algoritmo,
  p indicando qual foi alterado e axis indicando se foi no eixo x ou y.
  '''

  best2=np.copy(best)

  Nx=0.0
  Ny=0.0

  if random.random() < 0.5:
  #Se for menor do que 0.5 escolhemos atualizar o eixo x.
    axis = 0 #indicação do eixo x.
    p = random.randrange(len(best[0])) #escolha do valor a ser atualizado

    while True:
      Nx = 0
      for i in range(c[2*p]):
        Nx += random.uniform(-1,1)
      best2[0][p]=np.float64(best[0][p]+Nx*step_size[0]/c[2*p]).item()
      #atualização do valor.

      if best2[0][p] >= bounds[0][0] and best2[0][p] <= bounds[0][-1]:
        #caso esteja fora dos limites refaz a atualização.
```

```
        break

  else: #se random>0.5 esoclhemos atualizar o eixo y.
    axis = 1
    p = random.randrange(len(best[1]))
    while True:
      Ny = 0
      for i in range(c[2*p+1]):
        Ny += random.uniform(-1,1)
      best2[1][p]=np.float64(best[1][p]+Ny*step_size[1]/c[2*p+1]).item()
      if best2[1][p] >= bounds[1][0] and best2[1][p] <= bounds[1][-1]:
        break

  return best2, p, axis

#função principal

def simulated_annealing(objective, bounds, step_size, temp,N_stations,best):
  ''' Função principal do simulated_annealing onde acontecem as inicializações
  dos pontos, atualizações e determina os melhores pontos.
  Recebe os limites da região, o step_size do incremento, a temperatura do
  recozimento, número de estações e um vetor vazio best.
  Retorna principalmente os melhores pontos correntes "current" e a soma da função
  objetiva desses pontos "current_eval",
  além de demais vetores que são utilizados para plotagem de gráficos para
  verificar o funcionamento do algoritmo.
  '''
  #valores iniciais e inicialização de vetores para extração de dados.
  num_accepted = 1
  num_evaluated = 0
  x = []
  f = []
  dist2=[]
  #vetores para análise do funcionamento do código
  temp2=[]
  num_accepted2=[]
  num_evaluated2=[]
  c2=[]
```

```
  # gera o ponto inicial
  best = valores_iniciais(N_stations,bounds,best)
  # inicializa o vetor com fator de cristalização com uns.
  c=2*N_stations*[1]
# avalia o ponto inicial
  best_eval,dist = objective(best[0], best[1], pontos_demanda)
# determina a solução corrente
  curr, curr_eval = best, best_eval

  # Roda o código. Termina quando não encontra nenhuma outra solução aceita.
  while num_accepted > 0:
    temp2.append(temp)
    num_accepted2.append(num_accepted)
    num_evaluated2.append(num_evaluated)

    num_accepted  = 0
    num_evaluated = 0

    #Analisa no máxima N_stations*40 resultados aceites ou
    #N_stations*80 resultados avaliados.
    while num_accepted < N_stations*20*2 and num_evaluated < N_stations*40*2:
      # Aualização do ponto anterior
      candidate, p, axis = soma_valores(curr,step_size,bounds,c)
      num_evaluated += 1
      # Obtém a distância do ponto atualizado.
      candidate_eval,dist = objective(candidate[0],candidate[1],pontos_demanda)

      # Determina a diferença entre o ponto candidato e o ponto corrente.
      diff = (candidate_eval - curr_eval)

      # Calcula a temperatura
      t = temp
      # calcula o fator de metropolis.
      metropolis = exp(-diff / t)
      # Verifica se devemos aceitar o novo ponto
      if diff < 0 or rand() < metropolis:
        num_accepted += 1

        if c[2*p + axis]>5:
```

```
          c[2*p + axis] = round(c[2*p + axis]/10)

        c3=np.copy(c)

        # Aceita o novo ponto.
        curr, curr_eval = candidate, candidate_eval
      else:
        #caso não aceite, aumenta o fator de cristalização do ponto.
        if c[2*p + axis] < 40:
          c[2*p + axis] += 1
          c3=np.copy(c)

    c2.append(c3)

    f.append(curr_eval)
    #atualização da temperatura do recozimento
    temp = temp*0.98
    x.append(curr)

  return [curr, curr_eval, x, f,temp2,c2,num_accepted2,num_evaluated2]

#condições iniciais para funcionamento do SA.

#numero de estações
N_stations= 18

#Vetor vazio para armanezar os pontos obtidos do SA.
best= [[0 for col in range(N_stations)] for row in range(2)]

#cria pontos de demanda da USP com base no arquivo csv inputado.

#Transforma o dataframe em lista e insere em um novo vetor
#pontos_demanda para realizar uma cópia.
demanda_lat=df['latitude'].to_list()
demanda_long=df['longitude'].to_list()
demanda_pop=normalize(df['Population'].to_list())
pontos_demanda=np.zeros((len(demanda_long),3))

for j in range(len(demanda_long)):
```

```
  pontos_demanda[j][1]=demanda_lat[j]
  pontos_demanda[j][0]=-demanda_long[j]
  pontos_demanda[j][2]=demanda_pop[j]

# seed para o gerador de números pseudoaleatórios.
seed(1)

# define o limite da região que está sendo tratada.
bounds=[[-df.longitude.max(),-df.longitude.min()],
[df.latitude.min(),df.latitude.max()]]

# definição do step_size máximo
step_size = [bounds[0][-1]-bounds[0][0],bounds[1][-1]-bounds[1][0]]

# temperatura inicial.
temp = 10

# performa-se o algoritmo de SA, iniciando o tempo para análise futura.
start_time = time.time()
best, score,x,f_SA,temp,crist,NA,NE =
simulated_annealing(objective, bounds, step_size, temp,N_stations,best)
tempo_SA= (time.time() - start_time)
print('Done!')
print('f(%s) = %f' % (best, score))

#plota-se um gráfico com os pontos de demanda e
#os pontos obtidos pelo SA para análise.
best2 = [i * -1 for i in best[0]]
plt.scatter(best2,best[1],marker="+")
pd_x=[]
pd_y=[]
for i in range(len(pontos_demanda)):
  pd_x.append(-pontos_demanda[i][0])
  pd_y.append(pontos_demanda[i][1])
plt.scatter(pd_x,pd_y,marker="o")
plt.show()

#Entrada para USP
#importação da base de pontos de demanda da usp.
```

```
df = pd.read_csv("usp.csv",names=
['id','Name','Population','latitude','longitude'])

#estabelece os limites do gráfico em latitude e longitude.
BBox = ((df.longitude.min(),   df.longitude.max(),
         df.latitude.min(), df.latitude.max()))
df.head()
BBox

#Entrada para Manhattan
df = pd.read_csv("manhattan_centros_local.csv",
names=['id','Name','Population','latitude','longitude'])


BBox = ((df.longitude.min(),   df.longitude.max(),
         df.latitude.min(), df.latitude.max()))
df.head()
BBox

#Análise dos valores obtidos no código

#Evolução varíaveis x e y para cada ponto
x = np.array(x)
n = len(x[0][0])
fig, axs = plt.subplots(n,2,figsize=(2.5*n,2.5*n))


for i in range(n):
  axs[i][0].plot(temp,x[:,0][:,i])
  axs[i][0].set_xscale("log")
  axs[i][0].invert_xaxis()
  axs[i][1].plot(temp,x[:,1][:,i])
  axs[i][1].set_xscale("log")
  axs[i][1].invert_xaxis()
  axs[i][0].legend(['$x_{%d}$'%(i+1)])
  axs[i][1].legend(['$y_{%d}$'%(i+1)])

fig.suptitle("Evolução das variáveis ao longo da temperatura")
```

```
# Curva do número de aceites e de avaliados
plt.figure(2)
plt.plot(temp,NE)
plt.xscale("log")
plt.gca().invert_xaxis()
plt.legend(["Numero Avaliados"])
plt.xlabel("temperatura")
plt.figure(1)
plt.plot(temp,NA)
plt.xscale("log")
plt.gca().invert_xaxis()
plt.legend(["Numero Aceites"])
plt.xlabel("temperatura")

#Curva do fator de cristalização

xcrist=[]
ycrist=[]
x2crist=[]
y2crist=[]
for i in range(len(crist)):
  xcrist.append(crist[i][0])
  ycrist.append(crist[i][1])
  x2crist.append(crist[i][2])
  y2crist.append(crist[i][3])

plt.figure(3)
plt.plot(temp,xcrist)
plt.legend(["Cristalização x"])
plt.gca().invert_xaxis()
plt.xscale("log")
plt.xlabel("temperatura")

plt.figure(2)
plt.plot(temp,ycrist)
plt.legend(["Cristalização y"])
plt.gca().invert_xaxis()
plt.xscale("log")
plt.xlabel("temperatura")
```

```
plt.figure(4)
plt.plot(temp,x2crist)
plt.legend(["Cristalização x"])
plt.gca().invert_xaxis()
plt.xscale("log")
plt.xlabel("temperatura")

plt.figure(5)
plt.plot(temp,y2crist)
plt.legend(["Cristalização y"])
plt.gca().invert_xaxis()
plt.xscale("log")
plt.xlabel("temperatura")

# Curva da função objetiva
f_SA = np.array(f_SA)

plt.figure()
plt.plot(temp,f_SA)
plt.legend(["Objective"])
plt.gca().invert_xaxis()
plt.xscale("log")
plt.xlabel("temperatura")

Disc=20
temp2=[]
for i in range(0,len(temp)-Disc,Disc):
  temp2.append((sum(temp[i:i+Disc-1]))/len(temp[i:i+Disc-1]))


Media=[]

for i in range(0,len(f_SA)-Disc,Disc):
  Media.append((sum(f_SA[i:i+Disc-1]))/len(f_SA[i:i+Disc-1]))
Media = np.array(Media)

Maxi=[]
```

```
for i in range(0,len(f_SA)-Disc,Disc):
  Maxi.append(max(f_SA[i:i+Disc-1]))
Minim=[]

for i in range(0,len(f_SA)-Disc,Disc):
  Minim.append(min(f_SA[i:i+Disc-1]))
Minim = np.array(Minim)
plt.figure(3)
plt.plot(temp2,Minim)
plt.plot(temp2,Media)
plt.plot(temp2,Maxi)
plt.xscale("log")
plt.gca().invert_xaxis()
plt.legend(["Minimo", "Média", "Maximo"])
plt.xlabel("temperatura")
```

APÊNDICE C – Código Particle Swarm Optimization

```
# Algoritmo do Particle Swarm Optimization

# função objetiva que estamos tentando minimizar.
def func1(x):
  '''
  Função objetiva do algoritmo do PSO. Recebe o vetor x com as posiçoes x e y
  de cada ponto que foi determinado pelo algoritmo.
  Nesse caso o vetor de pontos_demanda foi estabelecido como variável global.
  Ela retorna a soma que corresponde à soma do mínimo entre às distâncias
  dos pontos do algoritmo e dos pontos de demanda.
  '''
  soma = 0.0

  for n in range(len(pontos_demanda)):
  #para cada ponto de demanda verifca-se todos pontos criados pelo algoritmo
    dist = []

    for i in range(round(len(x)/2)):
      x1 = x[2*i]
      y1 = x[2*i+1]
      dist.append(euclidean(x1,y1,pontos_demanda[n][0],pontos_demanda[n][1]))

    soma+=min(dist)*pontos_demanda[n][2]

  return soma


class Particle:
    def __init__(self,x0):
        self.position_i=[]          # posição da partícula
        self.velocity_i=[]          # velocidade da partícula
        self.pos_best_i=[]          # melhor posição individual
        self.err_best_i=-1          # melhor erro individual
        self.err_i=-1               # erro individual

        for i in range(0,num_dimensions):
```

```
            self.velocity_i.append(random.uniform(-1,1))

            self.position_i.append(x0[i])

    # Avalia a posição
    def evaluate(self,costFunc):

        self.err_i=costFunc(self.position_i)

        # Verifica se a posição corrente é uma melhor posição individual
        if self.err_i < self.err_best_i or self.err_best_i==-1:
            self.pos_best_i=self.position_i
            self.err_best_i=self.err_i


    # Atualiza a velocidade da particula
    def update_velocity(self,pos_best_g,w,c1,c2):

        for i in range(0,num_dimensions):
            r1=random.random()
            r2=random.random()

            vel_cognitive=c1*r1*(self.pos_best_i[i]-self.position_i[i])
            vel_social=c2*r2*(pos_best_g[i]-self.position_i[i])
            self.velocity_i[i]=w*self.velocity_i[i]+vel_cognitive+vel_social

    # Atualiza a posição da particula de acordo com a velocidade atualizada.
    def update_position(self,bounds):
        for i in range(0,num_dimensions):
            self.position_i[i]=self.position_i[i]+self.velocity_i[i]

            if i%2 ==0:
            #verifica se os novos valores de posição estão dentro dos limites,
            senão determina nova posição aleatória.
              if self.position_i[i]>bounds[0][1] or
              self.position_i[i] < bounds[0][0] :
                 self.position_i[i]=random.random()*(bounds[0][1]-bounds[0][0])
                 +bounds[0][0]
```

```
            else:
              if self.position_i[i]>bounds[1][1] or
              self.position_i[i] < bounds[1][0]:
                  self.position_i[i]=random.random()*(bounds[1][1]-bounds[1][0])
                  +bounds[1][0]

class PSO():
    def __init__(self,costFunc,x0,bounds,num_particles,
    maxiter,pontos_demanda,w,c1,c2):
        global num_dimensions
        global pos_best_g
        global itera
        global err_best_g
        global list_err_best_g
        list_err_best_g=[]
        global swarm_iterations
        swarm_iterations=[]

        num_dimensions=len(x0)
        err_best_g=-1                   # melhor erro para o grupo
        pos_best_g=[]                   # melhor posição para o grupo

        # estabelece o swarm
        swarm=[]
        for i in range(0,num_particles):
            swarm.append(Particle(x0))

        # começa a optimização do loop
        i=0
        l=0
        stop = False
        best_stop = err_best_g

        while i < maxiter and stop == False:

            # Percorre as particulas no swarm e avalia os pontos
            for j in range(0,num_particles):
                swarm[j].evaluate(costFunc)
```

```
                # Determina se a particula corrente é a melhor global.
                if swarm[j].err_i < err_best_g or err_best_g == -1:
                    pos_best_g=list(swarm[j].position_i)
                    err_best_g=float(swarm[j].err_i)
                    itera=i

            # percorre o swarm e atualiza a velocidade e posição
            for j in range(0,num_particles):
                swarm[j].update_velocity(pos_best_g,w,c1,c2)
                swarm[j].update_position(bounds)
            list_err_best_g.append(err_best_g)

            list_swarm = []


            for k in range(len(swarm)):
              list_swarm.append(swarm[k].pos_best_i.copy())
            list_swarm2 = list_swarm.copy()
            swarm_iterations.append(list_swarm2)

            i+=1
            l+=1

            if err_best_g > 1.01*best_stop or err_best_g < 0.99*best_stop:
              l=0
              best_stop = err_best_g
            if l >=20:
              stop = True

        # printa o resultado final
        print ('FINAL:')
        print (pos_best_g)
        print (err_best_g)

if __name__ == "__PSO__":
    main()

#Run
```

```
global pontos_demanda

#Determina condições iniciais
n_estacoes = 18
num_particles=3000
maxiter=1000

vx=[]
vy=[]
for i in range(len(pontos_demanda)):
  vx.append(pontos_demanda[i][0])
  vy.append(pontos_demanda[i][1])
bounds=[[min(vx),max(vx)],[min(vy),max(vy)]]

pos_glob=[]
itera_g=[]
err_best_g_list=[]
swarm_iterations_g=[]
err_best=[]
w=0.1
# Constante de inercia (peso), quanto pesa a velocidade anterior
c1=2.5        # constante cognitiva
c2=3      #constante social
a=0

#início do programa.
start_time = time.time()
#caso queira fazer mais de uma vez o programa e retirar o melhor
for p in range(0,1):
  a+=1
  initial=[]
  for i in range(0,n_estacoes):
    pos_x=random.random()*(bounds[0][1]-bounds[0][0])+bounds[0][0]
    pos_y=random.random()*(bounds[1][1]-bounds[1][0])+bounds[1][0]
    initial.append(pos_x)
    initial.append(pos_y)

  PSO(func1,initial,bounds,num_particles,maxiter,pontos_demanda,w,c1,c2)
  pos_glob.append(copy.copy(pos_best_g))
```

```
  itera_g.append(copy.copy(itera))
  err_best_g_list.append(copy.copy(list_err_best_g))
  swarm_iterations_g.append(copy.copy(swarm_iterations))
  err_best.append(copy.copy(err_best_g))
tempo_PSO= (time.time() - start_time)
#Plot de resultados com os pontos de demanda, SA e PSO.
min_value = min(err_best)
min_index = err_best.index(min_value)
err1=[]
err2=[]
for i in range(round(len(pos_glob[min_index])/2)):
  err1.append(-pos_glob[min_index][2*i])
  err2.append(pos_glob[min_index][2*i+1])
plt.scatter(err1,err2,marker="+",s=55) #PSO
plt.scatter(best2,best[1],marker="v",s=55) #SA
pd_x=[]
pd_y=[]
for i in range(len(pontos_demanda)):
  pd_x.append(-pontos_demanda[i][0])
  pd_y.append(pontos_demanda[i][1])
plt.scatter(pd_x,pd_y,marker="o",s=20) #Pontos de demanda
plt.show()

#Tabela comparação dos algoritmos
from tabulate import tabulate
table = [['Método', 'Nº Estações', 'Resíduo', 'Tempo','Iterações'],
['SA', N_stations, score,tempo_SA,len(f_SA)],
['PSO',n_estacoes, min_value,tempo_PSO,len(err_best_g_list[min_index])]]
print(tabulate(table))
print('Delta Resíduo:   ', (score-min_value))

#Funções para verificação do código

#Função objetiva e evolução das variáveis dos pontos
min_value = min(err_best)
min_index = err_best.index(min_value)

f = np.array(err_best_g_list[min_index])
#f=f[:5]
```

```
plt.figure(1)
plt.plot(f)
plt.legend(["Objective"])
plt.xlabel("iteração")



f2 = np.array(swarm_iterations_g[min_index])

posi=np.around(pos_glob[min_index],4)
f3=np.around(f2[itera_g[min_index]-1],4)

for j in range(len(f3)):

  if bool(set(posi).intersection(f3[j])):
    vector=j

posicao=np.zeros((len(f2[0][0]), itera_g[min_index]))


for i in range(itera_g[min_index]):
  for j in range(0,len(f2[0][0])-1,2):
    posicao[j][i]=(f2[i][vector][j])
    posicao[j+1][i]=(f2[i][vector][j+1])


n = len(posicao)
fig, axs = plt.subplots(round(n/2),2,figsize=(2.5*n,2.5*n))
print(posicao)

for i in range(round(n/2)):
  axs[i][0].plot(posicao[2*i][:])
  axs[i][1].plot(posicao[2*i+1][:])
  axs[i][0].legend(['$x_{%d}$'%(i+1)])
  axs[i][1].legend(['$y_{%d}$'%(i+1)])

fig.suptitle("Evolução das variáveis ao longo das iterações")
```

APÊNDICE D – Código Processing e auxiliares

Código utilizado no Processing e também código utilizado para preparar os dados inseridos.

```
#Código Auxiliar

import numpy as np
import random
from matplotlib import pyplot as plt
import math
import csv
import pandas as pd

# Funções auxiliares
def dist(a, b):
  # distância entre dois pontos do tipo [xi,yi]
  return ((a[0] - b[0])**2 + (a[1] - b[1])**2)**0.5
def measure(a,b):  # transforma distancia de latitude/longitude para metro
  lat1=a[0]        #é importante quando utilizamos a vel em m/s
  lat2=b[0]
  lon1=a[1]
  lon2=b[1]
  R = 6378.137; # Raio da terra em km.
  dLat = lat2 * math.pi / 180 - lat1 * math.pi / 180;
  dLon = lon2 * math.pi / 180 - lon1 * math.pi / 180;
  a = math.sin(dLat/2) * math.sin(dLat/2) + math.cos(lat1 * math.pi / 180) *
  math.cos(lat2 * math.pi / 180) *math.sin(dLon/2) *
  math.sin(dLon/2);
  c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a));
  d = R * c;
  return d * 1000; # metros
def chooseDestination(n, num_trips, stations_s, tempo_atual,trip_queue,weights):

  for i in range(num_trips):
    stations2 = stations_s.copy()

    choice = random.choices(range(0,len(stations2)),weights=weights,k=1)
    while choice[0] == n: # não pode escolher própria estação como destino
```

```
      choice = random.choices(range(0,len(stations2)),weights=weights,k=1)

    vel=np.random.normal(2.46661500206781, 1.73536134461819)*60
    while vel<0.005:
      vel=np.random.normal(2.46661500206781, 1.73536134461819)*60

    trip_queue.append([stations2[n][0],stations2[choice[0]][0],
    tempo_atual,tempo_atual +
    round(measure(stations2[choice[0]][0],stations2[n][0])/vel)])
    # insere na fila
  return trip_queue

#importar as frequencias e as estações da cidade todas as 326.

df = pd.read_csv("frequencias_manh.csv",
names=['id','Saida','Chegada','latitude','longitude'])

freq=[]
freq=df.values.tolist()

stations3=[]
for i in range(0,len(freq)):
  stations3.append([[float(freq[i][4]),float(freq[i][3])],
  float(freq[i][2]),int(math.ceil(float(freq[i][1]))),
  round(np.random.normal(10,3))])
stations3=np.array(stations3)
print(stations3)

#Código para gerar as filas
stations4=stations3.copy()
stations=stations4.copy()

trip_queue = [] # fila de viagens [[destino1,tempo de chegada1],
#[destino2,tempo de chegada2],...]
tempo=60*14
for i in range(tempo): # iterações - tempo rodando modelo
  if i%60==0:
      stations=stations3.copy()
  peso = stations[:,2].copy()
```

```
  weights = stations[:,1].copy()

  for n in range(len(stations)): # cada estação

      choices_l=list(range(0,int(peso[n])+1))
      weights_l=[1/60]*(int(peso[n])+1)
      weights_l[0]=1-peso[n]/60

      num_trips=random.choices(choices_l,weights_l,k=1)

      if num_trips[0] > stations[n][3]:
      # não podem sair mais bicicletas do q tem na estação
        num_trips[0] = stations[n][3]

      stations[n][2] -= num_trips[0]
      stations[n][3] -= num_trips[0]
      #bicicletas saem da estação de origem

      trip_queue=chooseDestination(n, num_trips[0], stations, i,
      trip_queue,weights)
      #função q escolhe destino e tempo de viagem e insere na
      fila de viagens

  # Visualização simplificada
  print("Minuto %d" %(i))
  print('Fila de viagens:', trip_queue)

#Alterando o formato do vetor trip_queue
queue_stations=[]

for k in range(len(trip_queue)):
  queue_stations.append(trip_queue[k][0])
  queue_stations.append(trip_queue[k][1])

remove_duplicate = []
for i in queue_stations:
    if i not in remove_duplicate:
        remove_duplicate.append(i)
```

```
print(len(queue_stations))
print(len(remove_duplicate))

N_bikes=[]
for i in range(len(stations3)):
  for k in range(len(remove_duplicate)):
    if remove_duplicate[k]==stations3[i][0]:
      N_bikes.append(stations3[i][3])

print(remove_duplicate)
#retorna lista das estações para colocar no processing
print(N_bikes)
trip_queue2=[]
trip_queue2=trip_queue.copy()

for i in range(len(trip_queue2)):
  trip_queue2[i][2]=float(trip_queue2[i][2])
  trip_queue2[i][3]=float(trip_queue2[i][3])
  for k in range(len(remove_duplicate)):
    if remove_duplicate[k]==trip_queue2[i][0]:
      trip_queue2[i][0]=k
    elif remove_duplicate[k]==trip_queue2[i][1]:
      trip_queue2[i][1]=k
print(trip_queue2)
print(len(trip_queue2))

with open('trip_queue.csv', 'w') as myfile:
  wr = csv.writer(myfile,quoting=csv.QUOTE_ALL)
  wr.writerow(trip_queue2)
  #retorna um csv com as coordenadas de início, fim,
  tempo de inicio e tempo de fim em uma lista.


#Código Processing


import random
from Trip import *
import math
```

```
def changeCoordinates(p):

    # converte x
    OldMin = -74.015
    OldMax = -73.905
    NewMin = 0.0
    NewMax = 500
    OldRange = (OldMax - OldMin)
    NewRange = (NewMax - NewMin)
    NewValuex = (((p[0] - OldMin) * NewRange) / OldRange) + NewMin

    # converte y
    OldMin = 40.715
    OldMax = 40.885
    NewMin = 900
    NewMax = 0.0
    OldRange = (OldMax - OldMin)
    NewRange = (NewMax - NewMin)
    NewValuey = (((p[1] - OldMin) * NewRange) / OldRange) + NewMin

    return [NewValuex,NewValuey]
class Trip:
  def __init__(self, position =0, origin = 0, destination = 0,
  startTime = 0, totalTime = 0):

    self.origin = origin
    self.destination = destination
    position2=[]
    for i in range(len(stations[position])):
        position2.append(stations[position][i])
    #arrayCopy(stations[position],position2)
    self.position=position2

    self.startTime = startTime*60
    self.totalTime= totalTime*60
 def update(self):
        if frameCount> self.startTime:
            self.position[0] += (stations[self.destination][0] -
            stations[self.origin][0])/
```

```
            (self.totalTime-self.startTime)
            self.position[1] += (stations[self.destination][1] -
            stations[self.origin][1])/
            (self.totalTime-self.startTime)


  def drawBike(self,i):
      if frameCount > self.startTime:
          if frameCount < self.totalTime:
              #text(i,self.position[0], self.position[1])
              circle(self.position[0], self.position[1], 7)
          elif frameCount == self.totalTime:
              N_bikes[(self.destination)]+=1
      elif frameCount == self.startTime:
          N_bikes[(self.origin)]-=1
#inputs
stations = [[-73.9678664378652, 40.75829727], [-73.95274565689759, 40.8030079]]
N_bikes=[37, 31, 20, 14]
trips_csv=[[[-74,40.7],[-73.98,40.68],0,20],
[[-73.95,40.76],[-73.96,40.65],30,50]]
#exemplos de valores
for i in range(len(stations)):
    stations[i] = changeCoordinates(stations[i])

trips = []
for i in range(len(trips_csv)):
    position1 = trips_csv[i][0]
    origin1 = position1
    destination1 = trips_csv[i][1]
    startTime1 = trips_csv[i][2]
    endTime1 = trips_csv[i][3]
    print([position1,origin1,destination1,startTime1, endTime1])
    trips.append(Trip(position1,origin1,destination1,startTime1, endTime1))
print(trips)

def setup():
    #size(313, 777)
    size(500,900)
    frameRate(60)
```

```
    #noStroke()


def draw():
    background(0,0,0)
    img = loadImage("map (10).png")
    img.resize(500,900);
    image(img,0,0)
    global stations
    global trips
    global N_bikes
    textSize(35)
    fill(0, 0, 0)
    text('Minutos:',40, 150)
    text(frameCount/60,185, 150)
    text(':',225, 150)
    text(frameCount%60,235, 150)
    # draw stations
    for i in range(len(stations)):
        fill(255, 0, 0)
        circle(stations[i][0], stations[i][1],10)
        textSize(18)
        fill(0, 0, 0)
        text(N_bikes[i],stations[i][0]-19, stations[i][1]-5)
        #text(i,stations[i][0]+5, stations[i][1]+5)
    fill(255, 0, 0)
    for i in range(len(trips)):
        trips[i].update()
        trips[i].drawBike(i)
```