PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME DE MELLO MATTOS TASCHETTO
E PEDRO PILLON VANZELLA

SIMULAÇÃO E PLANNING APLICADOS A ELEVADORES

Porto Alegre
2016

GUILHERME DE MELLO MATTOS TASCHETTO E PEDRO PILLON VANZELLA

SIMULAÇÃO E PLANNING APLICADOS A ELEVADORES

Trabalho de Conclusão II apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. João Batista de Oliveira

Porto Alegre
2016

AGRADECIMENTOS

Ao Professor João Batista pelas inesquecíveis aulas de ALPRO 3; pelas orientações instigantes e certeiras; por sua paixão pela computação e intelecto inspirador.

A meus pais Erli e Elisabeth e o restante da família. Nada seria possível sem o seu eterno e infinito amor, admiração e incentivo.

À minha esposa Lívia, pelo amor, compreensão, paciência e pelos lanchinhos com achocolatado quentinho servidos durante as muitas horas dedicadas à este trabalho.

E a todos amigos que participaram, de alguma forma, da minha formação.

Guilherme Taschetto

À minha mãe, que sempre me apoiou.

Ao meu pai, que sempre me mostrou o caminho certo.

E à minha esposa Vitória, que esteve ao meu lado por tudo isto.

Pedro Vanzella

“Beware of bugs in the above code; I have only proved it correct, not tried it.”
(Donald E. Knuth)

RESUMO

Elevadores são um meio de transporte utilizado por milhões de pessoas no mundo inteiro. Este trabalho apresenta de um simulador como ferramenta para analisar diferentes políticas e técnicas de *scheduling* de elevadores, de modo a reduzir o tempo que passageiros despendem em função destes. A modelagem destas políticas e técnicas é detalhada e elas são simuladas em diferentes cenários. Por fim, os resultados obtidos são analisados e comparados, buscando orientar quais técnicas e políticas são mais adequadas para cada cenário.

Palavras-Chave: elevadores, algoritmos, simulação, agendamento, planejamento.

ABSTRACT

Elevators are a mode of transportation used by millions of people around the world. This project presents a simulator as a tool for analyzing scheduling techniques and policies, in order to reduce users' waiting times. The techniques and policy models are detailed and they are simulated under differentes scenarios. Results are then analyzed and compared, trying to show which techniques and policies are better for each scenario.

Keywords: elevators, algorithms, simulation, scheduling, planning.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Em 2014, 54% da população mundial vivia em áreas urbanas, de acordo com a Organização das Nações Unidas [17]. A expectativa é que esta proporção aumente para 66% até o ano 2050. Em números absolutos isto representa um acréscimo de 2,5 bilhões de pessoas à população urbana mundial nos próximos 35 anos. Uma das consequências da alta densidade populacional em regiões geográficas limitadas é o crescimento do modelo de verticalização na construção civil. Neste cenário, onde prédios de diversos andares se tornam presentes no cotidiano da maioria da população, os elevadores[1] passam a um papel de destaque.

Uma pesquisa realizada pela IBM (Figura 1.1) no ano de 2010 em 16 cidades norte-americanas constatou que, durante 12 meses, o tempo estimado no qual trabalhadores de escritórios[2] aguardaram por elevadores foi de 92 anos [6]. Em uma economia onde o salário horário médio de um trabalhador é de US$ 24,99, o tempo de espera por elevadores representa custos de mais de US$ 20 bilhões em média por ano [3] somente nas cidades listadas.

Além do impacto econômico existe o impacto psicológico. Trabalhadores em centros metropolitanos empreendem uma parcela significativa da sua rotina no deslocamento entre residência e local de trabalho e no caminho inverso ao final do dia. Além de gastar uma quantidade significativa de tempo no trânsito das ruas, em carros, ônibus, bicicletas e metrôs, o tempo compreendido entre aguardar o elevador e desembarcar no andar desejado está longe de ser desprezível. De acordo com reportagem da revista Time, o *everyday commute*, ou *translado casa-trabalho e trabalho-casa* em uma tradução livre, pode causar efeitos físicos, como aumento nos níveis de açúcar, colesterol e dores nas costas a psicológicos0 como aumento na ansiedade e depressão [11].

Neste contexto global, a indústria de elevadores possui alguns desafios: primeiro, lidar com a pressão para a redução de custos na construção civil, construindo sistemas de grupos de elevadores mais baratos e eficientes, com melhorias no desempenho de transporte; segundo, competir no mercado oferecendo serviços novos, personalizados e com garantia de qualidade, visando revolucionar a maneira como elevadores interagem e servem passageiros [8]. Já sob o ponto de vista dos passageiros, estes esperam que suas chamadas sejam atendidas imediatamente e que sejam levados ao seu destino o mais rápido possível. Portanto, melhorias no desempenho destes sistemas traduzem-se em alto valor tanto para os fabricantes quanto para os usuários de elevadores.

[1] Dispositivos de transporte vertical que movimentam pessoas ou cargas entre andares ou níveis de um prédio ou estrutura.

[2] Em uma força de trabalho total de 51 milhões de trabalhadores, dos quais 12,7 milhões são usuários de elevadores diariamente [6].

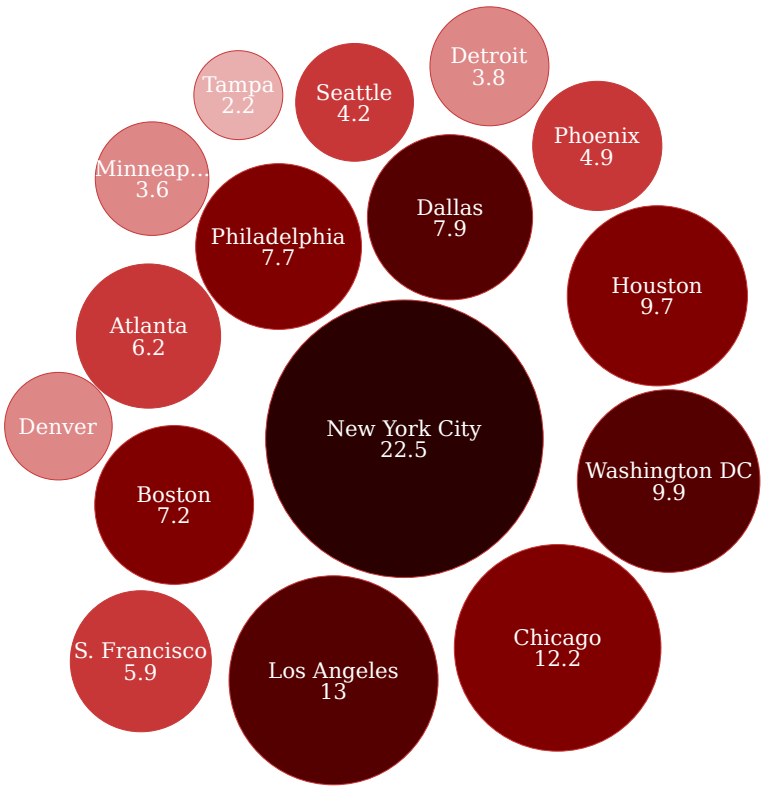

Figura 1.1 – Tempo de espera acumulado (em anos) por elevadores durante 12 meses em 16 cidades norte-americanas. Fonte: [6]

Existem diversas abordagens que os fabricantes de elevadores podem usar para tornar o sistema mais eficiente. Por exemplo, projetar o sistema com um número maior de elevadores ou optar por elevadores com maior capacidade de carga. Entretanto, este tipo de alteração não é sempre realizável em função de limitações na estrutura do prédio ou inviabilidade financeira. Uma solução de mais fácil aplicação é otimizar o sistema de controle dos elevadores.

A função deste sistema é resolver um problema de otimização: atribuir elevadores para atender chamadas feitas pelos passageiros minimizando alguma métrica. Entretanto, este problema encontra-se no conjunto de problemas NP-difícil (ou NP-hard, ou NP-completo) [15]. Portanto, uma solução ótima, computável em tempo polinomial, ainda não é conhecida para este problema.

Desde meados dos anos 1980 a indústria de elevadores vem estudando e implementando estratégias para encontrar soluções sub-ótimas para o problema de otimização. Diversas técnicas de Inteligência Artificial foram adotadas, como redes neurais, algoritmos genéticos, lógicas *fuzzy* e, mais recentemente, sistemas multi-agentes, planejamento e aprendizado de máquina [8]. Porém, melhorias significativas apenas são encontradas nos modelos de ponta, que adotam novos paradigmas de utilização e interface com usuários, e acabam ausentes dos modelos mais simples e legados, i.e., que já estão instalados nos prédios.

Ao analisar a tecnologia atual, percebe-se que houve pouca evolução nos elevadores desde sua concepção - sobretudo nos sistemas mais simples e mais baratos. De fato, o mecanismo que os faz mover evoluiu drasticamente, evidenciado ao comparar máquinas de tração manual com máquinas elétricas modernas. No entanto, essas evoluções são de difícil percepção aos usuários de elevadores. O sistema de controle dos elevadores ainda é simples, utilizando-se muito pouco de técnicas de algorítmicas para alterar seu comportamento em tempo de execução, apesar dos esforços da indústria ao longo das últimas décadas.

Acredita-se que tais técnicas permitam trazer uma evolução grande para os sistemas de controle de grupos de elevadores, assim como os motores elétricos foram para a tração. Por isto, o objetivo desde trabalho é comparar, através de simulações, diferentes estratégias de controle de elevadores em alguns cenários. Assim, espera-se ser possível avaliar, dentre as opções possíveis, quais combinações resultam em um melhor desempenho no transporte de passageiros para cada cenário.

1.1 Motivação prática

O uso de elevadores é presente na vida de grande parte dos habitantes de grandes metrópoles. Uma parcela significativa do dia-a-dia desta população é passada em grandes caixas metálicas, ou esperando pelas mesmas. A possibilidade de aplicar conhecimentos em algoritmos para melhorar o desempenho deste meio de transporte e, por consequência, contribuir com um aumento na qualidade de vida de seus passageiros é um grande atrativo para o estudo deste assunto.

Além disso, a algoritmos são uma área de conhecimento de grande interesse dos autores, junto dos problemas relacionados a simulações, às ferramentas de programação e à engenharia de software. O problema de otimização na alocação de elevadores apresenta um campo de pesquisa único para a aplicação destas ferramentas em busca da prática destes conhecimentos.

2 DESCRIÇÃO DO PROBLEMA

A maior parte dos prédios comerciais possui instalações de grupos com 2 a 8 elevadores [8]. Em muitos prédios, esses são o meio de transporte primário entre andares, visto que escadas são menos práticas e, muitas vezes, menos acessíveis ou exclusivas para situações de emergência. Dado o seu uso em larga escala, a ineficiência dos sistemas de controle de elevadores é percebida diariamente por seus usuários sob a forma de tempo despendido. Neste cenário, deseja-se encontrar formas de otimizar estes sistemas de controle e refletir positivamente no desempenho geral do sistema e na percepção da qualidade por seus usuários.

O problema estudado neste trabalho é modelado da seguinte forma: seja um prédio com **F** andares e **E** elevadores, cada um com capacidade para transportar **C** pessoas simultaneamente; sabe-se que a chegada de passageiros, em cada andar, obedece uma função de distribuição de probabilidade **D**, distinta das demais. De que forma o sistema pode atender os passageiros de modo a minimizar o tempo médio de atendimento das pessoas em um dado intervalo de tempo?

2.1 Sistema de controle de grupo de elevadores

Um *Elevator Group Control System* (**EGCS**), ou sistema de controle de grupo de elevadores, é responsável por coordenar as ações dos elevadores do prédio [10]. Esta coordenação visa atender a todas as **chamadas de corredor**[1] e **chamadas de cabine**[2] em um dado instante. Para isto, utiliza como entrada um conjunto de dados que modelam o **estado atual do sistema**. Este conjunto de dados pode ser modelado, minimamente, pelas seguintes informações:

- Para cada andar:
  - Se existe uma **chamada de corredor** a partir deste andar e o sentido (subir, descer ou ambos);
- Para cada elevador:
  - Um conjunto de **chamadas de cabine** solicitadas pelos passageiros embarcados;
  - O andar em que se encontra;

[1] Passageiros estão fora dos elevadores e realizam uma chamada para subir ou descer a partir do andar em que se encontram.

[2] Passageiros estão dentro do elevador e realizam uma chamada para desembarcar em um andar destino.

  - Se está parado, subindo ou descendo;
  - Sua lotação[3] atual estimada.

Com base nestes dados o EGCS utiliza-se de algoritmos e técnicas para fornecer como saída:

- Para uma nova **chamada de corredor**:
  - Qual elevador deverá atender esta chamada de modo a minimizar o tempo de espera.

2.2 Aquisição de dados e métricas

Embora a aquisição dos dados sobre o estado atual do sistema[4] seja trivial para a tecnologia atual, através dos sensores e atuadores distribuídos ao longo da instalação do sistema, obter dados mais complexos sobre o tráfego de um sistema de elevadores ainda é um problema em aberto na indústria [8]. Em casos simples é possível designar pessoas para observar e contar passageiros entrando e saindo dos elevadores. Já em casos mais complexos foram aplicadas soluções mais engenhosas, como contagem de pessoas aplicando algoritmos de visão computacional em câmeras de segurança e sensores de carga. Estas abordagens possuem fatores complicadores - como, por exemplo, a dificuldade em lidar com as diferenças de iluminação, baixa qualidade de vídeo, baixa precisão de sensores, etc - e o resultado obtido não compensa o custo [8].

A simulação de sistemas de elevadores torna-se uma alternativa atraente para obter métricas e avaliar o desempenho de um sistema na fase de projeto. Por exemplo, supondo que em um prédio de 10 andares com 2 elevadores exista uma fila de 20 pessoas que desejam ir a andares variados do prédio; cada uma destas pessoas chegou à fila em um momento distinto; porém, a chamada de corredor foi realizada apenas pela primeira pessoa da fila. Como medir o tempo que cada passageiro ficou esperando pelo elevador chegar, se somente a primeira pessoa da fila interagiu diretamente com o sistema? Ou como medir o tempo que cada pessoa levou para chegar ao seu destino? A modelagem do problema e simulação em ambiente computacional permitem obter dados que em um ambiente real seriam impossíveis ou muito caros de se conseguir. Tais dados podem compreender:

- Para cada andar:
  - Uma fila dos passageiros que encontram-se esperando naquele andar e querem subir;

[3] A capacidade máxima **C** é uma propriedade do problema e é igual para todos os elevadores que compõem o sistema.

[4] Conforme visto na seção 2.1.

  - Uma fila dos passageiros que encontram-se esperando naquele andar e querem descer;

- Para cada passageiro:
  - A hora de sua chegada na fila;
  - O seu andar de origem;
  - O seu andar de destino;
  - A hora de embarque no elevador;
  - A hora de desembarque no andar de destino;

- Para cada elevador:
  - O conjunto dos passageiros que encontram-se dentro daquele elevador e suas informações associadas (vide item anterior).

A partir destes dados, algumas das principais métricas utilizadas pela indústria de elevadores para avaliar o desempenho de seus sistemas podem ser calculadas:

$HC_{5\%}$ Percentual da população total do prédio que um sistema de elevadores consegue transportar em um intervalo de 5 minutos. Um $HC_{5\%}$ aceitável é de no mínimo 14% [8]. Por exemplo, em um prédio cuja população é de 600 pessoas, este índice representa o transporte de no mínimo 84 pessoas em 5 minutos.

WT *Waiting Time*, ou tempo de espera; está compreendido entre a requisição por passageiro, ou sua chegada na fila, e o seu embarque em um elevador.

JT *Journey Time*, ou tempo de jornada; está compreendido entre o embarque de um passageiro em um elevador e o posterior desembarque em seu destino.

ST *System Time*, ou tempo de sistema; está compreendido entre a chegada de um passageiro e o desembarque em seu destino, ou seja, é a soma do tempo de espera com o tempo de jornada.

AWT *Average Waiting Time*, ou tempo médio de espera.

AJT *Average Journey Time*, ou tempo médio de jornada.

AST *Average System Time*, ou tempo médio de sistema.

RTT *Round-trip Time*, ou tempo de ida e volta em uma tradução livre; é o tempo médio de uma viagem de um elevador partindo do lobby, indo até todos os andares do prédio e de volta ao lobby em horário de pico.

O tempo médio de sistema (AST) define a qualidade do serviço, já que está ligado diretamente à percepção que os passageiros possuem do sistema. É correto afirmar que o desejo de um passageiro é chegar no seu destino o mais rápido possível - ou seja, com o

menor tempo de sistema possível. Normalmente, tempos de sistema menores relacionam-se com um alto $HC_{5\%}$; porém, passageiros tendem a dar maior importância a um baixo tempo de espera (WT) do que a um baixo tempo de jornada (JT) [8]. Isto por que, uma vez dentro do elevador, o passageiro não se sente mais esperando: ele sente que já está sendo servido. Ainda assim, embora uma redução de 32 para 28 segundos de espera não seja considerada uma grande melhoria, é psicologicamente importante evitar esperas longas.

2.3 Situações fora do escopo deste trabalho

Existem fatores humanos cujas ocorrências geram situações especiais que aumentam a complexidade do problema e podem prejudicar o desempenho geral do sistema. Por exemplo, podem ser destacadas as seguintes situações:

- Um passageiro segura a porta do elevador aberta para alguém atrasado - deste modo, o passageiro que segura a porta está sendo altruísta com o atrasado, porém egoísta em relação aos outros passageiros que estão aguardando nos demais andares;
- Um passageiros acidentalmente faz uma chamada de corredor no sentido errado;
- Um passageiro faz uma chamada de corredor, desiste de aguardar pelo elevador e utiliza as escadas - ou seja, não embarca no elevador;
- Um passageiro acidentalmente faz uma chamada de cabine para o andar errado;

Estes são apenas alguns exemplos de situações que ocorrem diariamente. Alguns destes problemas podem ser mitigados através de aprimoramentos da interface dos elevadores. Por exemplo, seleções acidentais de andares ou desistências poderiam ser desfeitas, caso houvesse uma interface para cancelar chamadas de corredor ou de cabine.

No entanto, propor este tipo de análise e alteração de interfaces e comportamentos não está no escopo deste estudo. O objetivo aqui é analisar e propor alterações somente nos sistemas de controle que regem o comportamento dos elevadores da maneira com que estão atualmente instalados na vasta maioria dos prédios.

3 OBJETIVO DO TRABALHO

O objetivo deste trabalho é comparar, através de simulações, diferentes estratégias de controle de elevadores em cenários distintos. Os resultados das simulações são avaliados e, dentre as opções possíveis, são verificadas quais estratégias resultam em melhor desempenho no transporte de passageiros para cada cenário. Tal melhora se reflete em duas métricas principais: *Waiting Time* e *Journey Time*. Entretanto, a meta principal é a redução do *Average Waiting Time*, ou tempo médio de espera. A preferência por esta métrica, em detrimento do *Average Journey Time*, ou tempo médio de jornada, se dá pois, uma vez dentro do elevador, o passageiro não se sente mais esperando: ele sente que já está sendo servido, conforme constatado na seção 2.2.

O simulador carrega uma lista de cenários a partir de um arquivo de configuração e realiza a simulação de cada cenário. É possível comparar várias estratégias em um mesmo cenário. Para isto, foram implementados dois algoritmos de agendamento - *simple* e *planning* (Seção 6.8) e 4 funções de custo - *random*, *nearest neighbour*, *better nearest neighbour* e *weighted* (Seção 6.9).

Após as simulações, o simulador fornece um relatório apresentando métricas[1] de desempenho para sistemas de controle de grupo de elevadores. Além do relatório, o simulador é capaz de gerar, a partir das métricas obtidas, os seguintes gráficos: *Clientes por Elevador*, *Chegadas por Andar*, *Desembarques por Andar* e *Tempo de Jornada por Andar*.

Clientes por Elevador Este gráfico mostra a ocupação total de cada elevador, durante o período de simulação, em um formato de barras.

Chegadas por Andar Este gráfico em barras ilustra a quantidade de clientes que foram gerados pelo simulador em cada andar.

Desembarques por Andar O gráfico de Desembarques por Andar plota, através de barras verticais, a quantidade de clientes que foram desembarcados em cada andar.

Tempo de Jornada por Andar Esta matriz de gráficos mostra, utilizando barras verticais, o tempo médio de jornada entre cada par de andares.

Os relatórios e gráficos dão base para análise e proposta de uma estratégia (ou um conjunto de estratégias) para serem implementados em prédios já existentes.

[1] Seção 2.2.

3.1 Cenários de testes

Pode-se dizer que cada prédio é um cenário em potencial no contexto deste trabalho. Os atributos que podem ser utilizados para definir um cenário são:

F Número total de andares do prédio.

E Número total de elevadores que compõem o sistema.

C Capacidade[2] (em número de pessoas[3]) máxima de passageiros que cada elevador é capaz de transportar.

D Conjunto[4] de distribuições de probabilidade de chegada de passageiros.

Logo, há tantos cenários possíveis quanto há prédios ao redor do mundo. Os cenários de testes serão limitados em algumas categorias. A escolha destas divisões segue a classificação [1] sugerida pela empresa **Emporis GmbH**, uma empresa de mineração de dados sobre imóveis com sede em Frankfurt. O limite inferior de 4 andares é devido à exigência legal do município de Porto Alegre[5] onde prédios deste tamanho ou maiores (mas não menores que isto) devem ser construídos com elevadores. O limite superior é de 163 andares[6].

Os cenários são definidos na tabela 3.1. Serão simulados os tamanhos de prédios limítrofes superiores de cada categoria combinados com as quantidades de elevadores limítrofes superiores estabelecidas de forma arbitrária.

Tabela 3.1 – Categorias de cenários de testes.

Cenário	Altura	F	E	C
Low-rise	menor que 35 m	4 a 11	1 a 2	6
High-rise	entre 35 e 100 m	12 a 39	5 a 8	10
Skyscraper	maior que 100 m	40 a 163	10 a 16	12

3.1.1 Parâmetros não utilizados

Em virtude da limitação de tempo para a elaboração deste trabalho, dois parâmetros foram removidos do escopo da simulação:

[2]Como efeito de redução de escopo, consideram-se todos os elevadores de um mesmo sistema como tendo a mesma capacidade.

[3]Considerando uma pessoa com peso médio de 70 kg.

[4]Cada andar do prédio possui uma distribuição distinta.

[5]http://www2.portoalegre.rs.gov.br/netahtml /sirel/atos/Lei%201344

[6]O maior arranha-céu do mundo em 2015 é o Burj Khalif, localizado em Dubai, com mais de 800m de altura distribuídos em 163 andares habitáveis.

P População total do prédio. Com a remoção deste parâmetro considera-se que a população do prédio é infinita - ou seja, enquanto a simulação estiver sendo executada, novos passageiros continuarão a surgir - respeitando a distribuição de probabilidade.

Pu Propósito do prédio: *residencial* (com baixo fluxo entre os andares superiores), *comercial com múltiplas empresas* (com médio fluxo entre andares superiores) ou *comercial com única empresa* (com alto fluxo entre andares superiores)[7]. Com a remoção deste parâmetro considera-se que a probabilidade de um passageiro chegar e ir para qualquer andar do prédio é sempre a mesma.

Acredita-se que a simplificação resultante destas remoções não vá impactar os resultados obtidos de forma negativa.

[7] O propósito do prédio é diretamente relacionado com **D**.

4 ALGORITMOS DE AGENDAMENTO PARA ELEVADORES

A busca pela solução do problema de atribuir elevadores para atender chamadas feitas pelos passageiros, minimizando alguma métrica, é apresentada pela literatura pesquisada na forma de algoritmos [8]. Tais algoritmos possuem complexidades distintas, indo desde algoritmos triviais - mas ainda interessantes para fins de comparação -, até soluções mais complexas onde mais dados são utilizados de modo a tomar decisões mais complexas.

Em um hipotético cenário ideal, ter-se-iam todos os dados de cada passageiro - *i.e.* cada pessoa que chegasse ao andar informaria de antemão para qual andar deseja ir antes mesmo de entrar no elevador. No entanto, isto não é realista no contexto dos sistemas de elevadores instalados atualmente, onde cada pessoa apenas informa se deseja subir ou descer. Portanto, os algoritmos aqui descritos tentam fazer inferências[1] a respeito de dados que não possuem, quando relevante, ou tentam tomar decisões ignorando os dados que não estão disponíveis.

A seguir são descritos, do mais simples ao mais complexo, os algoritmos selecionados para o desenvolvimento.

4.1 Nearest neighbour

O algoritmo de *Nearest Neighbour* é o mais ingênuo de todos, e servirá de base para a avaliação dos demais algoritmos. Seu funcionamento é trivial: o elevador mais próximo da chamada sempre atenderá esta chamada [4]. Um dos problemas deste algoritmo é que ele pode causar muitas mudanças de direção de um elevador, o que acarreta um tempo de espera maior para os passageiros que estão dentro dele.

Considere, por exemplo, o cenário da Figura 4.1. Este cenário é composto por um prédio de 8 andares com 3 elevadores. A situação dos elevadores é a seguinte:

- $E1$ no sexto andar, com ocupação[2] de 20% e como destino[3] o segundo andar;
- $E2$ no primeiro andar, com ocupação 10% e como destino o sexto andar;
- $E3$ no quarto andar, com ocupação 90% e como destino o sétimo andar.

Neste instante, uma nova chamada de corredor é originada no sétimo andar.

[1] *e.g.* A lotação do elevador pode ser estimada com base no peso reportado pela balança interna do elevador, que já se encontra nele por motivos de segurança, ou abstrair a capacidade e lotação em número de pessoas.

[2] A ocupação do elevador é representada pelo percentual dentro do círculo.

[3] O destino do elevador é representado pela seta.

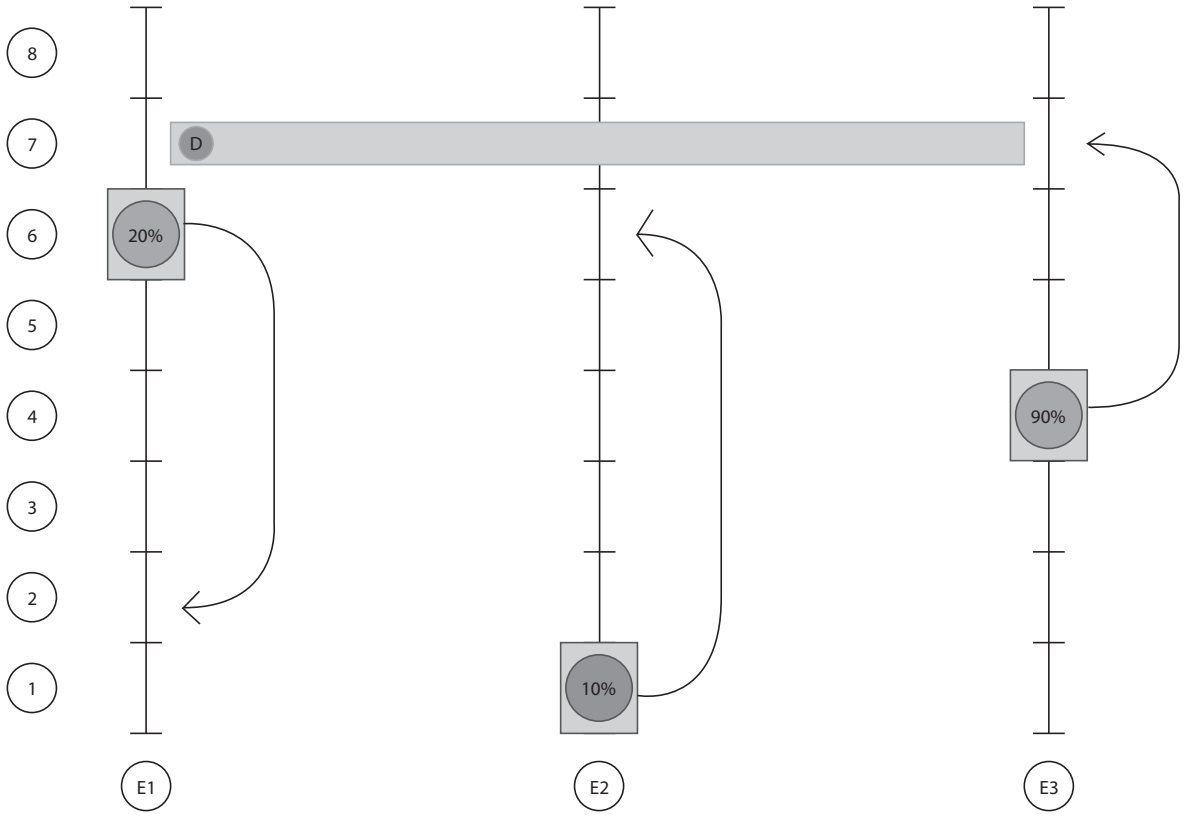

Figura 4.1 – Cenário exemplo #1 de prédio com 8 andares e 3 elevadores.

Caso o algoritmo de *Nearest Neighbour* seja utilizado, o elevador $E1$ seria selecionado para atender a chamada no sétimo andar. Isto é claramente ruim para os passageiros deste elevador. Além disto, é possível notar que o elevador $E3$ já tinha uma parada programada no sétimo andar e seria possível atender a chamada sem alterar a agenda de nenhum elevador. O algoritmo de *Nearest Neighbour*, no entanto, não leva em consideração estas informações.

O único propósito deste algoritmo é servir de base de comparação com outros algoritmos propostos, de modo a validarmos o simulador. Espera-se que uma melhora clara de desempenho seja notada ao comparar-se este com o próximo trivial, o *Nearest Neighbour Melhorado*.

4.2 Nearest neighbour melhorado

Uma melhoria que pode ser feita ao algoritmo de *Nearest Neighbour* é considerar o sentido em que o elevador está indo para atender a chamada [4]. Isto implica em considerar-se agora a informação de sentido das chamadas. É importante notar que ainda não se considera quantas pessoas fizeram uma chamada - apenas sabe-se que há chamadas no andar, e como destinos tem-se “subir”, “descer” ou “ambos”.

Este algoritmo resolve o problema de mudanças de direção que o algoritmo de *Nearest Neighbour* sofre.

Considere, novamente, o caso ilustrado pela Figura 4.1. Este algoritmo considerará apenas os elevadores que estão parados ou indo no sentido de onde a chamada foi originada. Neste caso, apenas os elevadores $E2$ e $E3$ serão considerados. O elevador $E3$ está mais próximo da chamada, então será escolhido.

4.3 Simple Scheduler

Podemos definir estes algoritmos, de forma mais genérica, como funções de custo. Estas funções são inerentes a cada elevador, descrevendo quão custoso é atender uma chamada [4]. A decisão de qual elevador é escolhido para atender a chamada é feita com base em qual deles terá o menor valor da função de custo.

Um exemplo de função de custo é:

$$J(e, l, p) = \lambda l(|p - e|)$$

Onde:

e Número do andar onde o elevador se encontra

l Percentual de ocupação do elevador

p Número do andar onde a chamada foi originada

λ Fator de multiplicação, que assume os seguintes valores:

- 0, caso o programa atual do elevador faça com que ele passe por aquele andar;
- 1, caso o elevador não tenha um programa (*i.e.*, ele esteja ocioso) ou o elevador esteja indo na direção da chamada;
- 2, caso ele mude de direção para atender esta chamada[4].

Utilizando como exemplo o cenário da Figura 4.2, onde uma chamada de corredor para descer[5] é originada no oitavo andar. A situação de cada elevador é a seguinte:

- $E1$ no sétimo andar, com ocupação de 20% e como destino o segundo andar;
- $E2$ no primeiro andar, com ocupação 10% e como destino o sexto andar;
- $E3$ no sétimo andar, com ocupação 90% e como destino o primeiro andar.

[4]Multiplica-se a distância por dois pois é necessário ir até o andar da chamada e então voltar para o andar onde se estava anteriormente, para só então atender a chamada.

[5]O sentido da chamada é representado pelo círculo com um **D**, de *Down*.

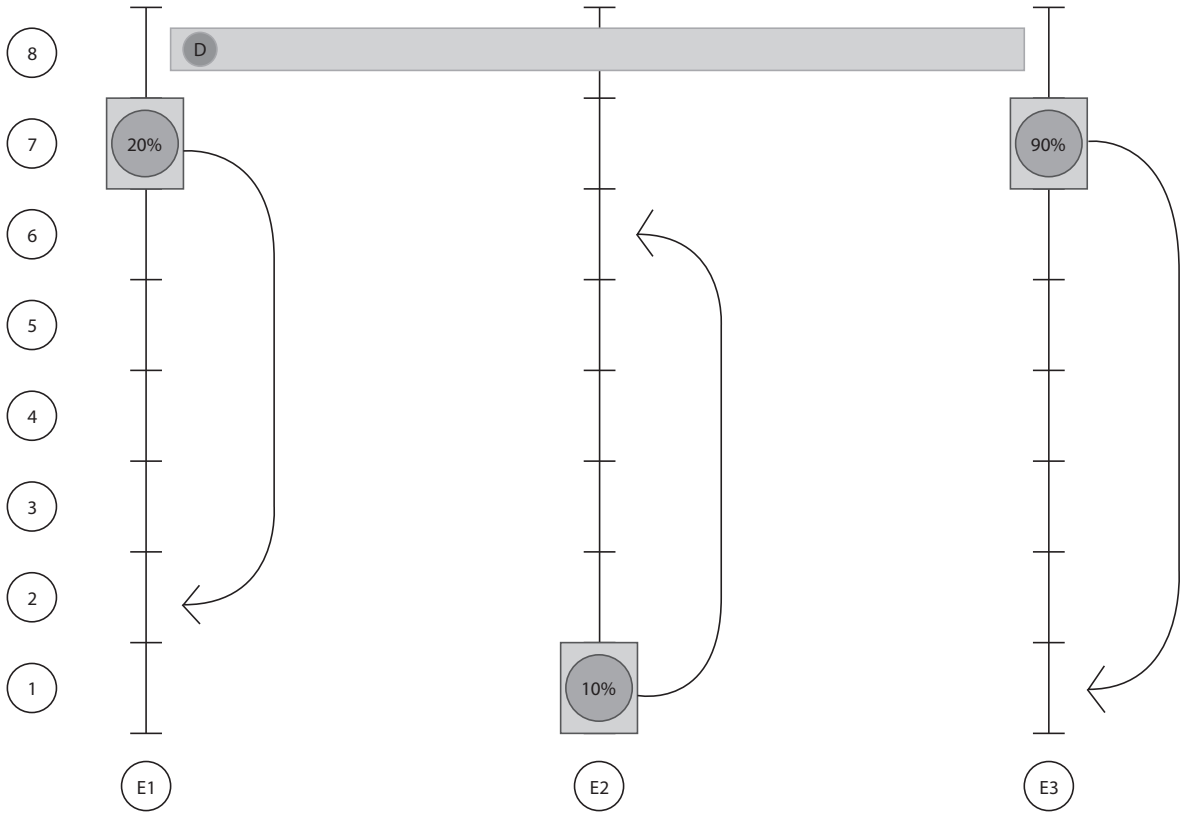

Figura 4.2 – Cenário exemplo #2 de prédio com 8 andares e 3 elevadores.

Podemos calcular o custo de atender a esta chamada para cada elevador do prédio. Para o Elevador $E1$:

$$J(7, 0.2, 8) = 2 \times 0.2 \times (|8 - 7|) = 0.4$$

Neste caso, λ é 2, pois o elevador deverá mudar de sentido (*i.e.*, ele deverá subir até o oitavo andar e então voltar para o sétimo andar, o que representa um deslocamento de dois andares).

Para o Elevador $E2$:

$$J(1, 0.1, 8) = 1 \times 0.1 \times (|8 - 1|) = 0.7$$

Neste caso $\boldsymbol{I}$ é 0.1, pois sua lotação é 10%[6], e $(p - e)$ é 7, pois o elevador deverá subir sete andares[7].

Para o Elevador $E3$:

[6]Como não há informações a respeito do futuro, não é possível considerar alterações na carga de um elevador.

[7]Pela regra de cálculo, foi considerada apenas a posição atual do elevador, e não a posição que ele estará ao fim de sua última atividade. Seria possível alterar esta regra, gerando uma função de custo levemente diferente, com resultados diferentes. É um teste válido para a implementação.

$$J(7, 0.9, 8) = 2 \times 0.9 \times (|8 - 7|) = 1.8$$

Neste caso $\boldsymbol{l}$ é 0.9, pois sua lotação é 90%, e $\boldsymbol{\lambda}$ é 2, pois o elevador $E3$ deve subir do sétimo para o oitavo andar e descer novamente até o sétimo.

Observa-se que, para esta função de custo, neste sistema, é vantajoso mudar o sentido de $E1$ para atender a chamada no oitavo andar, e depois continuar na direção original. Várias funções de custo podem ser experimentadas e comparadas: outras funções de custo levariam em consideração mudanças de direção de viagem[8], ou ainda tentar manter todos os custos o mais baixo possível, ao mesmo tempo que o mais próximos uns dos outros. Por exemplo, uma outra função de custo possível seria:

$$J(e, l, p) = l(\lambda(|p - e|))^2$$

Esta função penaliza mais o movimento, elevando-o ao quadrado. Para o exemplo da Figura 4.2, tem-se:

$$J(7, 0.2, 8) = 0.2 \times (2 \times (|8 - 7|))^2 = 0.8$$

$$J(1, 0.1, 8) = 0.1 \times (1 \times (|8 - 1|))^2 = 6.4$$

$$J(7, 0.9, 8) = 0.9 \times (2 \times (|8 - 7|))^2 = 3.6$$

Novamente, para esta função, o elevador $E1$ é eleito para atender a chamada.

4.3.1 Algoritmo

O Algoritmo 4.1 ilustra um algoritmo com o comportamento desta função. Os argumentos para este algoritmo são:

client Abstração de um cliente;

elevatorset Conjunto de todos os elevadores do prédio;

costFunction Ponteiro para a função de cálculo de custo.

Seu funcionamento pode ser descrito como: dada uma chamada, para cada elevador presente no conjunto calcula-se a função de custo e o algoritmo retorna o elevador com a menor função de custo.

[8] *E.g.*, pode ser vantajoso um elevador mudar de direção para atender uma chamada a um andar de distância, caso a alternativa seja fazer a chamada esperar um deslocamento de dezenas de andares de outro elevador.

```
ChooseElevator(client, elevatorSet, costFunction) {
  selectedElevador = elevatorSet.front();
  bestCost = infinite;
  for each (elevador in elevatorSet) {
    cost = costFunction(client, elevatorSet)
    if cost < bestCost {
      selectedElevator = elevator;
      bestCost = cost;
    }
  }
  return selectedElevator;
}
```

Algoritmo 4.1 – Minimização da *função de custo*.

4.3.2 Nearest neighbour como função de custo

É possível definir o algoritmo de *Nearest Neighbour* (Seção 4.1) como uma função de custo:

$$J(e, p) = |p - e|$$

Onde considera-se apenas $p - e$, que é a distância entre a posição atual do elevador (e) e o número do andar onde a chamada foi originada (p).

4.4 Planning

Neste algoritmo é realizada a expansão no espaço de estados em uma árvore de decisões, avaliando-se para cada passo o tempo que um elevador levará para chegar até o estado desejado, vários passos no futuro [9], como em um jogo de xadrez. Em cada nodo desta árvore de decisões (Figura 4.3) é feita a pergunta: *qual elevador deve atender qual chamada*?

A expansão da árvore pode ocorrer diversas vezes, fazendo com que a árvore de decisões possua múltiplos níveis. No fim desta expansão, que se dá quando o horizonte de expansão é atingido, é realizado o somatório, a partir da raiz até cada folha da árvore, dos custos calculados. No fim destes somatórios, o algoritmo toma a decisão pelo ramo com menor custo (a partir da raiz). Assim, avança-se um passo na simulação e executa-se o algoritmo novamente.

O parâmetro que limita a expansão da árvore é chamado de *horizonte de cálculo*. O valor deste horizonte é arbitrário e representa o número de níveis da expansão. Em uma

regra geral, quanto maior o horizonte de cálculo maior a chance do algoritmo tomar a melhor escolha. Porém, a complexidade de memória $O(N^N)$, onde N é o horizonte de cálculo e E é o número de elevadores, é um fator limitador do horizonte de cálculo.

Para cada nodo da árvore, instancia-se um simulador novo e executa-se esta simulação até o momento em que o elevador atenderia o chamado do cliente. Isto é importante para ter-se uma noção realista do comportamento dos outros elevadores. A única diferença deste simulador instanciado para um simulador real está na geração de novos eventos. Estes novos simuladores não geram eventos de chegada de cliente, dado que é impossível, no mundo real, prever quando um cliente novo chegará.[9]

Na Figura 4.3, vemos um exemplo hipotético de *planning* sendo executado com $N = 3$ e $E = 2$. Para cada nível da árvore a decisão a ser tomada é "qual elevador deve atender a próxima chamada da fila?", e, para isto, calcula-se o custo de tempo para cada elevador. O resultado do custo pode ser visto entre parênteses em cada nodo da árvore da Figura 4.3.

Ao atingir o horizonte (no caso da Figura 4.3, no nível 3 da árvore), soma-se os custos até cada folha (na Figura 4.3, os círculos abaixo do nível mais baixo representam a soma dos custos até aquela folha). O caminho de menor custo total é o que deve ser tomado. No exemplo da Figura 4.3, o elevador $E2$ será agendado para atender a primeira chamada da fila. O agendamento das demais chamadas é gerado somente na próxima execução do *scheduler*. Isto se dá pelo fato de que o estado do simulador pode ter sido alterado pela chegada de um cliente neste meio tempo. Mais detalhes podem ser vistos no Capítulo 6.

[9] O simulador tem esta capacidade, mas isto não seria justo, já que um sistema real não tem como ter este tipo de informação.

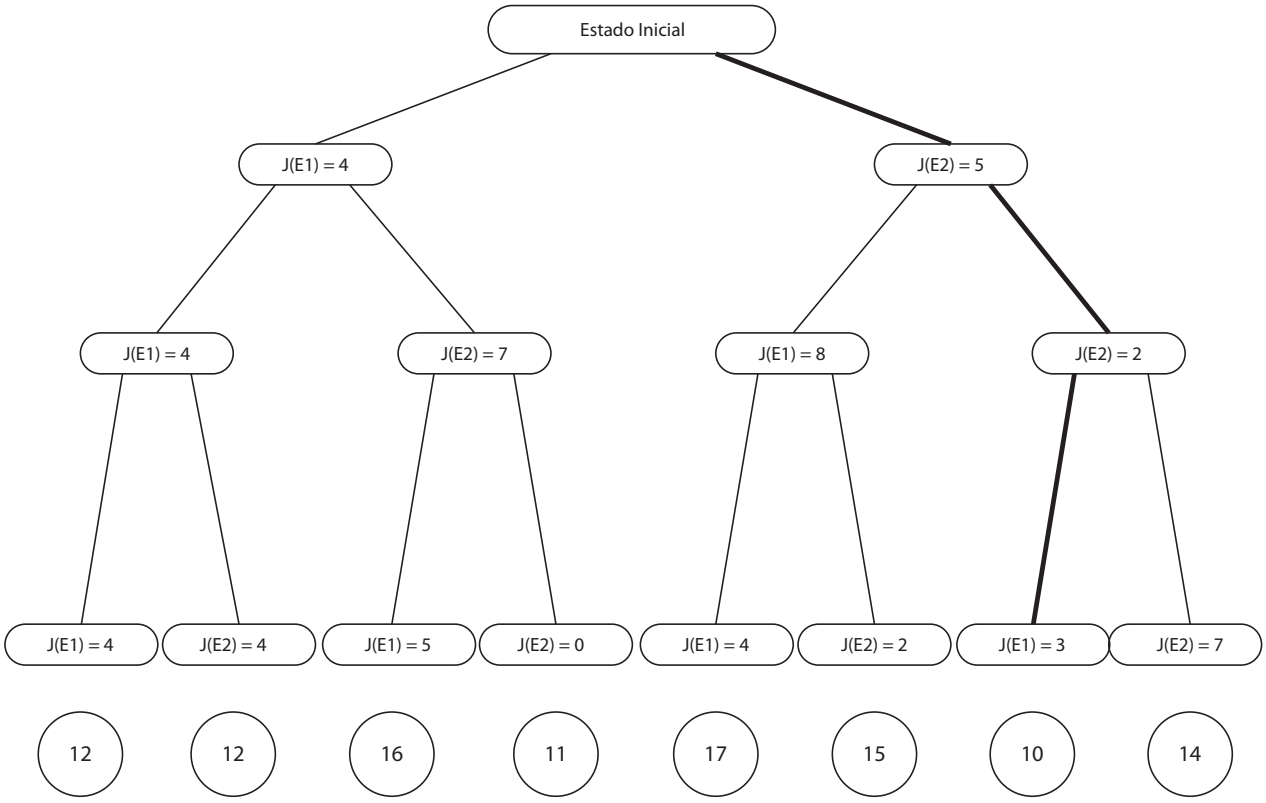

Figura 4.3 – Exemplo de *planning* com horizonte 3 e dois elevadores.

5 SIMULADOR

De acordo com Banks (2005, p. 3):

> Simulação é a imitação da operação de um sistema ou processo do mundo real ao longo do tempo. Ela envolve a geração de uma história artificial de um sistema ou processo e a observação desta história de modo a realizar inferências a respeito das características operacionais da realidade ali representada.

Entretanto, a simulação não é a única abordagem existente para se estudar e compreender um sistema e suas características. É senso comum que cada sistema, possuindo suas próprias características e idiossincrasias, deve ser analisado através da ferramenta correta. Logo, embora a simulação pareça ser uma boa alternativa à primeira vista em diversos casos, é possível que ela não seja a forma mais apropriada para o seu estudo. Assim, se faz importante a existência de métodos objetivos para que se possa verificar se a simulação é realmente a ferramenta apropriada para cada caso estudado.

5.1 Motivação

A fim de decidir de forma objetiva qual a melhor abordagem para um dado sistema, Law [12] propõe uma reflexão (Figura 5.1) através das seguintes perguntas:

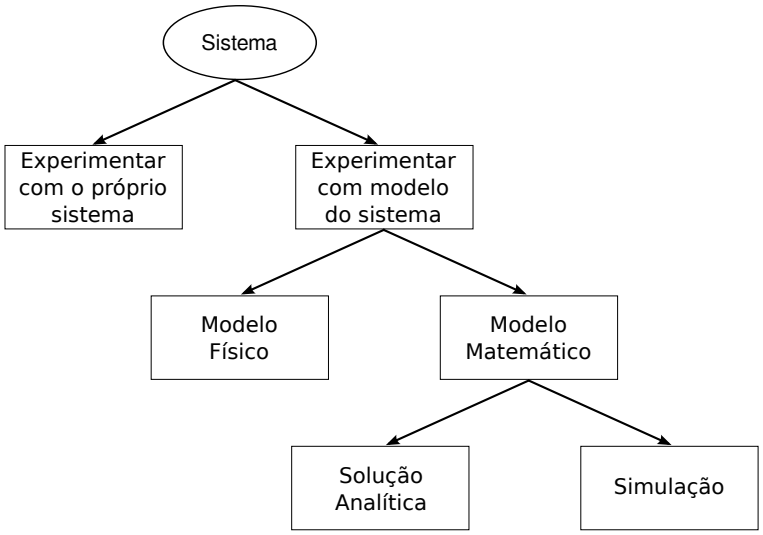

Figura 5.1 – Formas de estudar um sistema. Fonte: [12]

1. *Experimentar com o próprio sistema ou experimentar com um modelo do mesmo?*

   Um sistema de elevadores operacional e pronto para realizar as experimentações não encontra-se entre os recursos disponíveis para a elaboração da pesquisa. Ainda, mesmo que tal estrutura estivesse disponível, os cenários de testes possíveis seriam

limitados pelas restrições físicas do sistema, tornando o estudo das melhorias mais caro e menos flexível. Logo, optou-se pela utilização de um *modelo do sistema*.

2. *Experimentar utilizando um modelo físico do sistema ou utilizar um modelo matemático?*

   Um modelo físico de um sistema de elevadores poderia ser constituído por uma maquete de um prédio com mini-elevadores movidos a motores de passo, que por sua vez seriam controlados por microcontroladores programáveis conectados a uma rede de sensores. Este projeto por si só, entretanto, já seria grandioso demais - além de, obviamente, fugir do escopo da Ciência da Computação e ser mais adequado à um trabalho de conclusão de Engenharia Elétrica ou Engenharia de Controle e Automação. Além disso, da mesma forma que em um sistema real de elevadores, os cenários de testes possíveis seriam limitados pelas restrições físicas do modelo do sistema. Portanto, optou-se pela utilização de um *modelo matemático do sistema*, reproduzível em ambiente computacional e parametrizável para diferentes cenários.

3. *O problema pode ser resolvido de forma analítica?*

   Conforme afirmado no capítulo 1 deste estudo, o problema de atribuir elevadores para atender chamadas feitas pelos passageiros minimizando alguma métrica encontra-se no conjunto de problemas NP-difícil (ou NP-hard, ou NP-complexo) [15]. Assim, uma solução ótima, computável em tempo polinomial, ainda não é conhecida para este problema. Este fato vai ao encontro dos grandes esforços da indústria em procurar soluções para resolver o problema ao longo das décadas, não poupando esforços e investimentos em busca desta solução.

Portanto, se justifica a escolha da simulação de um modelo matemático do sistema como uma forma apropriada para o estudo e experimentação de um sistema de elevadores.

5.1.1 Classificação do modelo de simulação

A partir de um modelo matemático a ser estudado por meio de simulação (doravante chamado de *modelo de simulação*), o mesmo pode ser classificado em três dimensões [7, 12]:

1. *Estático ou Dinâmico*

   O modelo de simulação neste escopo é dinâmico. Deseja-se verificar o comportamento do sistema ao longo de um intervalo de tempo finito, à medida que passageiros chegam e elevadores os transportam através dos andares do prédio.

2. *Determinístico ou Estocástico*

Um modelo de simulação que não possua nenhum componente probabilístico (i. e, aleatoriedade) é chamado de *determinístico*. Neste tipo de modelo, a saída fornecida pelo modelo de simulação será sempre a mesma para uma mesma entrada. Porém a grande maioria dos sistemas do mundo real possuem, no mínimo, algum grau de aleatoriedade na sua entrada - por isso são chamados *estocásticos*. Estes, diferentemente de modelos *determinísticos*, fornecem uma saída igualmente aleatória - e, por esta razão, esta saída deve ser considerada como um conjunto de *estimativas* das características reais do sistema, e não as características propriamente ditas [7].

Em um **EGCS** real não é possível prever quantos passageiros utilizarão o sistema, quando eles chegarão e tampouco para qual andar irão. Portanto, um modelo de simulação válido neste escopo deve ser *estocástico* de modo a lidar com a aleatoriedade da entrada de passageiros no sistema.

3. *Contínuo ou Discreto*

A Figura 5.2 ilustra o comportamento de uma variável de estado em modelos de simulação *contínuos* e *discretos*. O modelo *contínuo* é aquele onde os valores das variáveis mudam continuamente ao longo do tempo. Já em um modelo *discreto* as variáveis de estado tem seu valor alterado em instantes separados no tempo [7]. Para este projeto, não há a necessidade de informações instantâneas a respeito da movimentação de passageiros e elevadores, e sim de observar o comportamento do sistema dada a ocorrência de eventos discretos. Portanto, o modelo de simulação utilizado neste estudo é *discreto*.

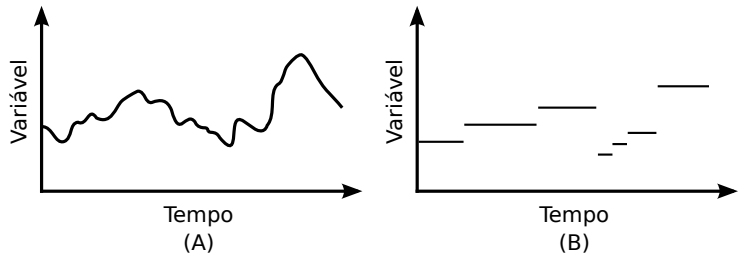

Figura 5.2 – Variável de estado em um modelo contínuo (A) e discreto (B). Fonte: [7]

5.1.2 Simulação baseada em eventos discretos

Segundo Law (2000, p. 6):

> A Simulação de Eventos Discretos (*Discrete-Event Simulation*) compreende a modelagem de um sistema à medida que ele **evolui ao longo do tempo** através de uma representação na qual as variáveis de estado são alteradas instantaneamente em **instantes separados no tempo**.

A abordagem sugerida por Law é o padrão de projeto de simuladores ao utilizar-se um modelo de simulação *dinâmico*, *estocástico* e *discreto* para representar o sistema do

mundo real. Em função da natureza dinâmica desta abordagem, é necessário acompanhar o tempo da simulação à medida que a simulação é executada, armazenando este valor em uma variável. Esta variável é chamada de *relógio da simulação*[1] [12].

Também se faz necessário um mecanismo de avanço de tempo que gerencie o valor desta variável. Neste simulador é utilizada a abordagem de *avanço de tempo para o próximo evento*, onde o *relógio da simulação* é inicializado em 0 e é determinado em que ponto tempo ocorrerão eventos futuros - em outras palavras, é feito o agendamento de eventos. Então, o *relógio da simulação* é avançado para o tempo da ocorrência do *primeiro* destes eventos futuros. Neste ponto do tempo, o estado do modelo é atualizado de acordo com o evento que ocorreu e novas ocorrências de eventos futuros são agendadas. Então, o *relógio da simulação* avança para o instante da ocorrência do *novo* primeiro dos eventos futuros e o estado do sistema é atualizado em função da ocorrência deste evento. Este processo repete-se até que uma condição de parada seja satisfeita.

Uma vez que toda alteração no estado do sistema ocorre na ocasião de um evento, os períodos de espera, que são o tempo entre a ocorrência de um evento e o próximo - não são relevantes para a simulação. Afinal, o estado do sistema não foi alterado neste ínterim - ou seja, nada de interessante ocorreu durante aquele tempo [2]. Deste modo, é possível reduzir o esforço computacional necessário para executar a simulação.

Um exemplo desta dinâmica é ilustrado pela Figura 5.3. O eixo do tempo, iniciado em 0, marca os tempos agendados para os eventos $\{e_0, e_1, e_2, e_3, e_4, e_5\}$. As setas indicam os valores assumidos pelo *relógio da simulação*, ou seja, $\{t_0, t_1, t_2, t_3, t_4, t_5\}$. Observa-se que t_0 = *tempo*(e_0), t_1 = *tempo*(e_1) e assim sucessivamente - ou seja, o *relógio da simulação* avança diretamente para o momento da ocorrência do próximo evento.

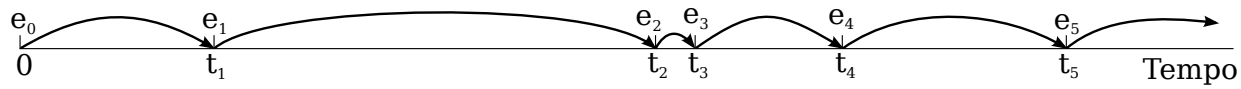

Figura 5.3 – Ilustração da evolução do *relógio da simulação* utilizando a abordagem de *avanço de tempo para o próximo evento*. Fonte: [12]

Sendo assim, o modelo deste estudo será uma simulação de eventos discretos com avanço de tempo para o próximo evento.

[1] Geralmente, não há relação entre o tempo de simulação e o tempo necessário para a simulação ser executada. Por exemplo, um experimento pode simular o funcionamento de um banco entre 9h e 17h (tempo de simulação), mas o tempo necessário para executar a simulação poderia ser de 4 minutos.

5.2 Fluxo de Execução

Independente do paradigma utilizado em sua implementação, um simulador de eventos discretos com avanço de tempo para o próximo evento possui um conjunto de componentes com responsabilidades bem definidas. São eles:

Estado do Sistema Coleção de variáveis necessárias para descrever o estado do sistema em um instante em particular;

Relógio de Simulação Variável contendo o valor atual do tempo de simulação;

Lista de Eventos Lista de eventos futuros, contendo, para cada evento, o seu tipo e o instante do tempo no qual este ocorrerá;

Contadores Estatísticos Conjunto de variáveis utilizados para armazenar informações estatísticas a respeito do funcionamento do sistema;

Rotina de Inicialização Subprograma para inicializar a simulação no instante zero do *relógio da simulação*;

Rotina de Temporização Subprograma responsável por determinar qual o próximo evento a ocorrer e avançar o relógio de simulação até o instante da sua ocorrência;

Rotina de Evento Subprograma que modifica o estado do sistema quando um determinado tipo de evento ocorre - para cada tipo de evento há uma rotina correspondente;

Rotinas Auxiliares Conjunto de subprogramas utilizados para gerar observações aleatórias sobre distribuições de probabilidade determinadas como parte do modelo de simulação;

Gerador de Relatórios Subprograma que computa estimativas das métricas desejadas e dão origem ao relatório quando a simulação termina;

Programa Principal Subprograma que invoca as rotinas de inicialização e temporização e delega o tratamento de cada evento à sua rotina correspondente, repetindo o processo até que a condição de parada da simulação seja verificada.

A Figura 5.4 ilustra a organização existente entre estes componentes, bem como o fluxo de execução da simulação. O início da simulação se dá pela execução do *Programa Principal*. Este, por sua vez, invoca a *Rotina de Inicialização*. Nesta rotina, o *Relógio da Simulação* é zerado, marcando o instante inicial. Além disso, são inicializados o *Estado Inicial* do sistema, os *Contadores Estatísticos* e são gerados os eventos iniciais, sendo adicionados à *Lista de Eventos*.

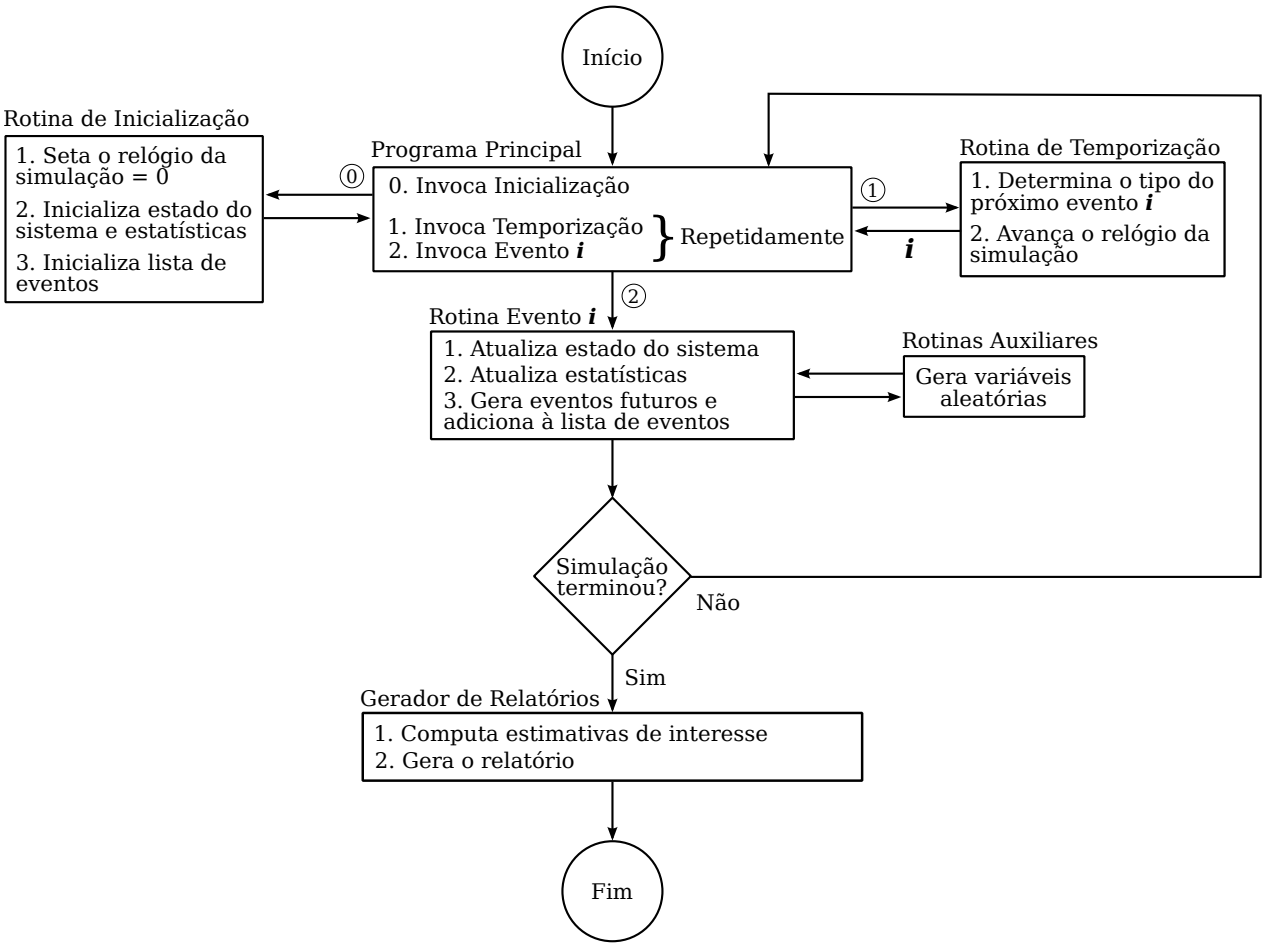

Figura 5.4 – Fluxo básico de um simulador com avanço de tempo para o próximo evento. Fonte: [12]

Após o controle retornar para o *Programa Principal*, o mesmo invoca a *Rotina de Temporização*. Esta rotina tem a responsabilidade de avaliar a *Lista de Eventos* e determinar qual é o primeiro evento a ser executado. Ao determinar qual é o evento, a rotina avança o *Relógio da Simulação* para o instante de sua ocorrência e devolve informações a respeito deste evento para o *Programa Principal*. De volta ao *Programa Principal*, o mesmo avalia o evento retornado pela *Rotina de Temporização* e invoca sua *Rotina de Evento* correspondente.

Para cada tipo de evento existe uma *Rotina de Evento* correspondente. Todas elas, obrigatoriamente, devem executar as seguintes tarefas: (1) atualizar o *Estado do Sistema* em função do evento ocorrido; (2) atualizar os *Contadores Estatísticos* em função do evento ocorrido; e (3) gerar novos eventos futuros e adicioná-los à *Lista de Eventos*. Para esta última tarefa, as *Rotinas Auxiliares* são utilizadas para diversas tarefas - dentre elas, a geração de variáveis aleatórias para dar suporte à geração de eventos estocásticos.

Após a execução da *Rotina de Evento*, é feita a verificação de uma condição de parada da simulação. Caso seja determinado que o fim da simulação deve ocorrer, o controle passa para o *Gerador de Relatórios*, cuja responsabilidade é computar as estimativas

de interesse, baseando-se nos *Contadores Estatísticos* e produzir um relatório com estes dados. Em caso contrário, o *Programa Principal* é invocado novamente e o ciclo se repete. É importante salientar que a *Rotina de Inicialização* somente é invocada na primeira execução do *Programa Principal*, sendo ignorada nas iterações posteriores.

5.3 Requisitos de Projeto

O projeto do simulador foi concebido de forma a garantir alguns requisitos considerados de importância em relação ao simulador e as simulações. São elas:

Configurável Deve ser possível definir cenários de forma simples e sem a necessidade de recompilar ou reconstruir os binários do simulador.

Para que isto fosse possível, os cenários para simulação são armazenados em um arquivo de configuração no formato *YAML*[2] e são carregados pelo simulador em tempo de execução com o auxílio da biblioteca `yaml-cpp`[3]. O arquivo de configuração é detalhado na Seção 6.6.1.

Determinístico Deve ser possível reproduzir simulações de maneira idêntica, independente do horário e da ordem em que for iniciada. Isto é especialmente importante ao comparar o mesmo cenário sob diferentes estratégias, quando é imperativo garantir que os passageiros cheguem nos mesmos instantes e locais em todas as simulações de um mesmo cenário.

Para que isto fosse possível, os cenários configurados no arquivo de configuração contém, dentre outras informações, uma semente utilizada para inicializar os geradores de números aleatórios e distribuições de probabilidade dentro do simulador.

Escalável Deve ser possível simular diversos cenários em uma mesma execução do simulador e em um tempo aceitável: alguns segundos para cenários simples e até 30 minutos para cenários complexos.

Para que isto fosse possível, optou-se por desenvolver o simulador na linguagem `C++11`. Tal linguagem é tida na indústria de software como capaz de oferecer desempenho superior às demais linguagens de propósito geral - devido, em grande parte, ao fato de ser compilada e permitir ao programador gerenciar diretamente o uso de memória do processo. Além disso, juntamente de sua biblioteca padrão, o `C++11` oferece um conjunto atraente de estruturas de dados, ferramentas de desenvolvimento e depuração.

[2] *YAML Ain't Markup Language*, http://yaml.org/

[3] *yaml-cpp is a YAML parser and emitter in C++*, https://github.com/jbeder /yaml-cpp

Rastreável Para cada simulação, o simulador deve gerar arquivos de *log*, detalhando os eventos e situações ocorridos durante a simulação no escopo daquele cenário.

Para que isto fosse possível, foi utilizada a biblioteca `glog`[4], desenvolvido pelo *Google* e com implementações para várias linguagens, dentre elas o `C++`.

Testável Deve ser possível desenvolver e executar testes unitários nos componentes do simulador.

Para que isto fosse possível, foi utilizada a biblioteca `Google Test`[5], desenvolvido pelo *Google* e com implementações para várias linguagens, dentre elas o `C++`.

[4] *C++ implementation of the Google logging module*, https://github.com/google/glog
[5] *Google's C++ test framework*, https://github.com/google/googletest

6 MODELO E IMPLEMENTAÇÃO

Conforme visto na Seção 5.2, um sistema de simulação possui uma série de componentes conceituais, cada um com suas responsabilidades bem definidas. Para projetar um simulador, diversas abordagens e paradigmas poderiam ser aplicadas. Neste estudo optou-se pelo paradigma de *Programação Orientada a Objetos*. Esta escolha se deu pelos seguintes motivos:

Capacidade de Abstração
: Conceitos da *Programação Orientada a Objetos*, como classes, interfaces, polimorfismo, herança e sobrecarga permitem a realização de uma modelagem conceitual em alto nível de abstração, permitindo uma explanação de fácil entendimento sem ser necessário abordar questões da implementação em si (linguagem de programação, arquitetura, etc).

Padrão de Mercado
: Desde meados dos anos 90, a *Programação Orientada a Objetos* tornou-se frequentemente utilizada no mercado de desenvolvimento de software e nos ambientes acadêmicos relacionados à computação. Assim, é possível atingir uma maior audiência.

Domínio dos Autores
: O paradigma é de domínio dos autores deste estudo.

Suporte nativo no `C++`
: O paradigma é suportado de forma no `C++11`, linguagem escolhida para a implementação.

Nas próximas seções serão apresentados os modelos[1] e detalhes de implementação dos componentes genéricos de um simulador. Após isto, serão apresentados os componentes específicos de um simulador de elevadores.

6.1 Eventos

A simulação de eventos discretos, como o próprio nome já diz, é orientada a eventos. Isso significa dizer que as alterações no estado do sistema ocorrerão somente na ocasião de algum evento e é preciso ser possível representar um evento no contexto do simulador. Um evento é uma estrutura que deve possuir as seguintes informações: (1) um número identificador; (2) o horário agendado para a ocorrência do evento; (3) o tipo do evento.

[1] Algumas simplificações foram realizadas nos modelos UML a fim de permitir uma melhor compreensão - por exemplo, omissão de métodos *getters* e *setters*.

Existem dois tipos de eventos que podem ocorrer durante uma simulação:

Chegada de cliente Um cliente chegou na fila de um andar.

Fim da simulação A simulação atingiu a duração especificada.[2]

Para obter tal funcionalidade foram criadas a classe abstrata `Event` e as classes concretas que a especializam, representando os eventos de **chegada de cliente** e **fim da simulação**, respectivamente: `ClientArrival` e `FinishSimulation` (Figura 6.1).

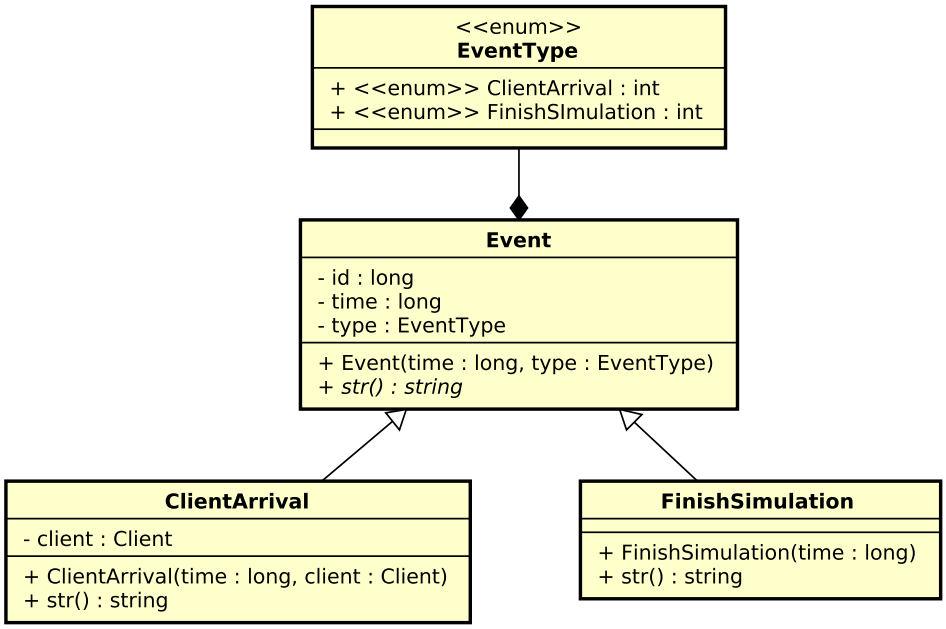

Figura 6.1 – Diagrama de classes dos *eventos e seus derivados*.

Event

Evento base composto pelo conjunto básico de informações para um evento.

`id` Número identificador do evento.

`time` Horário de ocorrência do evento.

`type` Tipo do evento.

ClientArrival

Evento especializado do tipo **chegada de cliente**. Contém, adicionalmente, as informações do cliente.

`client` Cliente que gerou o evento.[3]

[2] Este evento não necessariamente implica no fim imediato da simulação, apenas que a chegada de novos clientes não ocorrerá mais. O simulador ainda irá processar os clientes que estiverem nos andares ou dentro de elevadores.

[3] Tipo complexo armazenando diversas informações sobre o cliente (Seção 6.7.1).

FinishSimulation

Evento especializado do tipo **fim da simulação**. Não é necessária nenhuma informação adicional.

6.2 Gerenciamento de eventos

Na Seção 5.2 foram apresentados os componentes de um simulador. Alguns destes componentes devem reagir na ocorrência de um evento, alterando o seu estado interno de acordo com o tipo de evento ocorrido e as informações que ele carrega consigo. Porém resta saber qual evento será o próximo a ocorrer e como notificar os elementos reativos disso. Portanto, precisamos de mecanismos para criar e ordenar eventos e notificar os componentes da ocorrência de um evento.

6.2.1 Criação

Não é possível prever em qual andar e qual momento um cliente irá chegar ao prédio, tampouco qual o destino desejado por ele. Portanto, é correto afirmar que a chegada de clientes em um prédio é um processo estocástico.

Horário de chegada

Segundo [14], a taxa de chegada de clientes é um processo de Poisson, com razão λ. Ou seja, o tempo entre novos clientes são variáveis independentes, com valor esperado $\frac{1}{\lambda}$. Pode-se ver na Figura 6.2 um exemplo de diferentes valores de λ gerando diferentes distribuições.

Em um prédio no mundo real, esta distribuição varia de acordo com a hora do dia. Por exemplo, nos horários do início do turno da manhã e no início do turno da tarde, muito mais passageiros chegam ao térreo, com destino ao andar onde trabalham. Para fim de simplificação, esta variação ao longo do dia não foi considerada neste estudo. Conforme visto na Seção 6.6, cada andar do prédio possui um valor para λ, que permanece constante no decorrer de toda a simulação.

Andar de destino

Já a probabilidade de um cliente ir de um andar para outro, em um tráfego chamado de *interfloor*, é um processo de Markov, com distribuições normais (*i.e.* uma média e um desvio padrão) que variam de acordo com a hora do dia. Para fim de simplificação, esta variação ao longo do dia não foi considerada neste estudo. Além disso, considera-se que a probabilidade de um cliente ir para qualquer andar é a mesma -

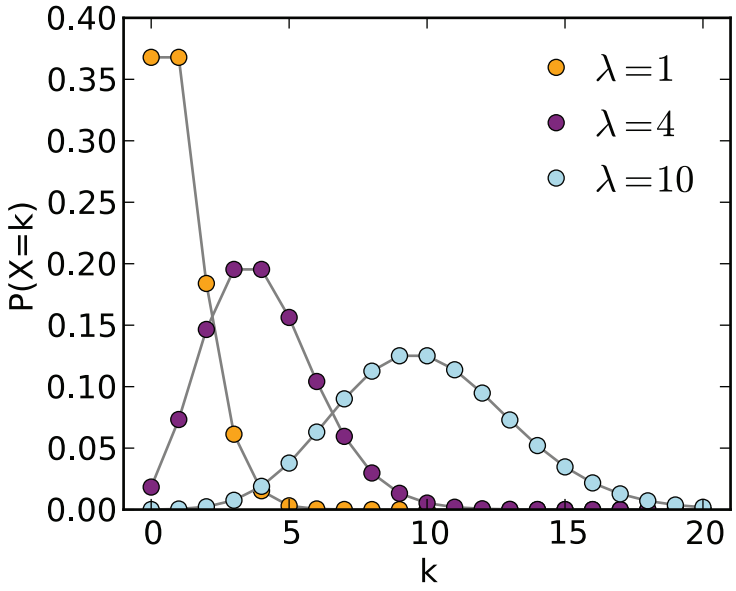

Figura 6.2 – Exemplos de distribuições de Poisson.

com exceção do próprio em que se encontra, que é nula. Abaixo uma matriz de probabilidades para um prédio com 3 andares, representada por uma cadeia de Markov na Figura 6.3.

$$\begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} = \begin{bmatrix} 0.0 & 0.5 & 0.5 \\ 0.5 & 0.0 & 0.5 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

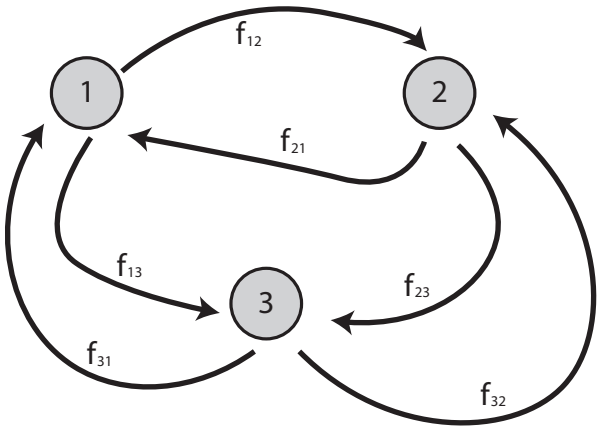

Figura 6.3 – Exemplo de cadeia de Markov para um prédio de 3 andares.

EventFactory

A classe `EventFactory`, ou *fábrica de eventos*, foi criada para ser uma unidade de coesão ao criar chegadas de clientes no prédio. Cada andar do prédio possui uma instância desta classe. Seus componentes são:

`clock` Referência para o relógio da simulação.

`floor`	Referência para o andar daquele `EventFactory`.
`random engine`	Gerador de números aleatórios.
`destination`	Distribuição Discreta, utilizada para sortear andares de destino.
`arrival`	Distribuição de Poisson, utilizada para sortear horários de eventos futuros.

O Algoritmo 6.1 ilustra a criação de um novo evento. Primeiro, `CreateFutureArrival` calcula o horário do novo evento. Isto é feito somando-se o horário atual do *relógio da simulação* com um tempo sorteado pela distribuição de Poisson específica daquele andar. Na sequência, é sorteado o andar de destino pela distribuição discreta também específica do andar. Feito isso, é criado um novo evento do tipo **chegada de cliente** com as informações sorteadas e tendo como origem o andar atual. No fim, o evento criado é adicionado à fila de eventos (Seção 6.2.2).

```
1  EventFactory::createFutureArrival(eventQueue) {
2    eventTime = clock.currentTime() + getNextTime();
3    destination = getNextDestination();
4    clientArrival = new ClientArrival(floor, destination, eventTime);
5    eventQueue.push(clientArrival);
6  }
7
8  EventFactory::getNextTime() {
9    return arrival(randomEngine);
10 }
11
12 EventFactory::getNextDestination() {
13   return destination(randomEngine);
14 }
```

Algoritmo 6.1 – Criação de um novo *evento de chegada de cliente*.

6.2.2 Priorização

Na Seção 5.1.2 foi apresentado o *mecanismo de avanço de tempo para o próximo evento*, onde deve-se verificar, em uma lista de eventos, qual é o próximo evento a ocorrer. Dado um conjunto de eventos agendados (ou seja, ainda não ocorridos), o primeiro evento a ocorrer é justamente o que possui o menor tempo de agendamento. Um tipo abstrato de dados que serve para este propósito é uma *fila prioritária*, ou *priority queue*, que funciona de forma similar a filas *FIFO*, com a diferença de que cada elemento armazenado possui uma prioridade associada. A *fila prioritária*, implementada na classe `EventQueue`, irá atender os elementos por ordem de prioridade, da maior para a menor. Ao considerar que a prioridade de um evento é inversamente proporcional ao instante em que irá ocorrer - ou seja, quanto

menor o tempo do evento maior é a sua prioridade -, temos uma fila na qual o próximo elemento a ser atendido sempre será o próximo evento a ocorrer.

EventQueue

Encapsula uma fila prioritária de eventos. Adicionalmente, utiliza a classe `EventComparator`, que implementa a relação de ordem entre os eventos.

`push`	Insere um evento na fila.
`top`	Recupera o próximo evento da fila.
`pop`	Recupera e remove o próximo evento da fila.
`hasNextEvent`	Verifica se a fila possui eventos.

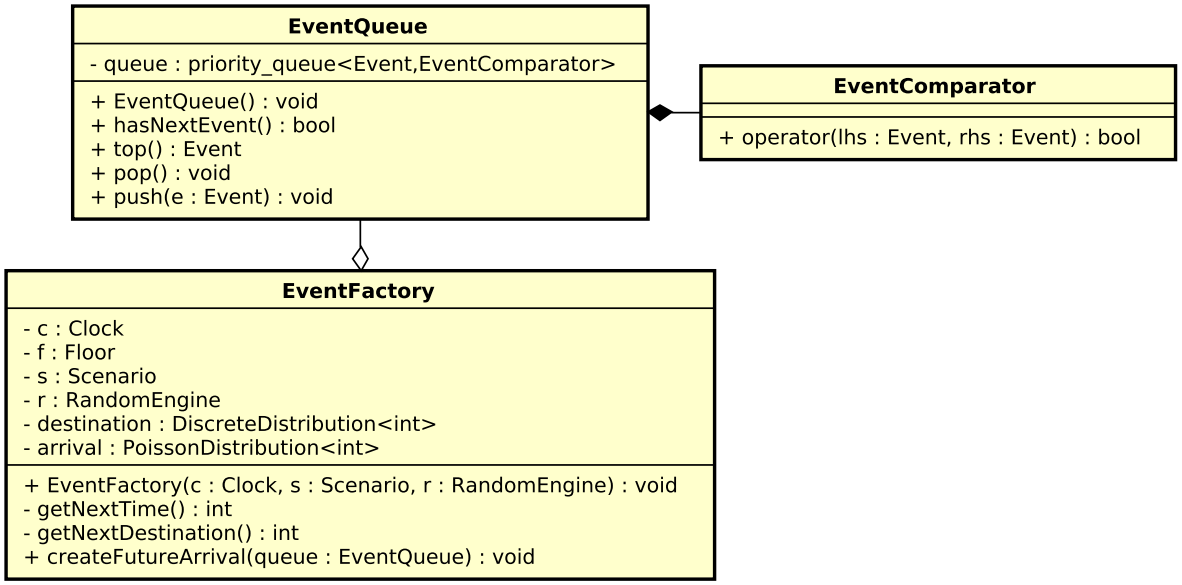

Figura 6.4 – Diagrama de classes da *criação e priorização de eventos*.

6.2.3 Notificação

Quando o próximo evento a ocorrer é conhecido, o problema passa a ser notificar os elementos reativos que este evento ocorreu para que os mesmos possam atualizar seus estados internos. De acordo com Gamma [5], o padrão *Observer* é um *design pattern* indicado para resolver este problema. Este *pattern* define uma dependência de um-para-muitos ($1 : N$) entre objetos de modo que, quando este um objeto (*subject*) tem seu estado alterado, todos os seus dependentes (*observers*) são notificados deste mudança. Por consequência, estes dependentes podem modificar seu estado interno baseando-se nas informações desta notificação.

Este padrão foi utilizado para implementar a funcionalidade de notificação de eventos, ilustrado no diagrama da Figura 6.5. Para isto, foram criados os seguintes componentes:

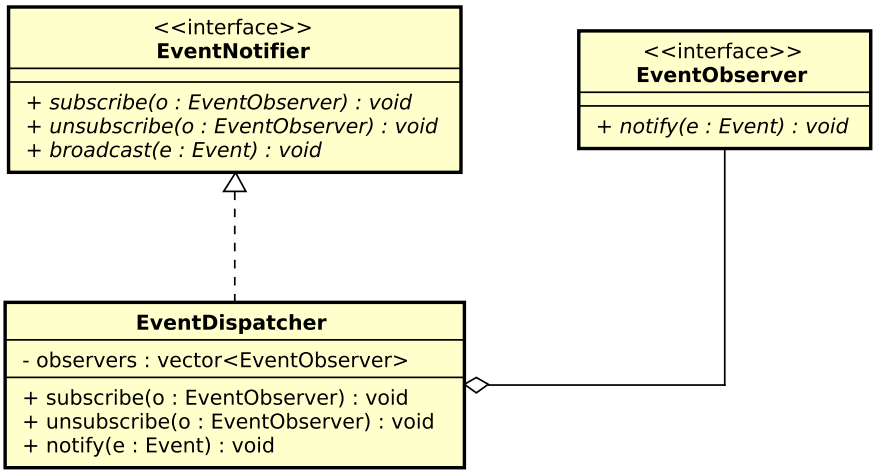

Figura 6.5 – Diagrama de classes da *notificação de eventos*.

EventObserver

Classe abstrata a ser realizada por qualquer outra classe que deseje receber notificações de eventos. Possui um único método abstrato.

`notify` Recebe a notificação da ocorrência de um evento.

EventNotifier

Classe abstrata a ser realizada por qualquer outra classe que deseje notificar a ocorrência de eventos. Define métodos para que objetos que implementem a classe abstrata *EventObserver* possam registrar-se para receber notificações de ocorrências de eventos. Seus métodos abstratos são:

`subscribe` Adiciona um *observer*.

`unsubscribe` Remove um *observer*.

`broadcast` Notifica todos os *observers* registrados da ocorrência de um evento.

EventDispatcher

Classe concreta que realiza a interface `EventNotifier`. Possui uma estrutura de dados para armazenar quais *observers* se registraram através dos métodos `subscribe` e `unsubscribe`.

Três importantes componentes do simulador podem se beneficiar desta construção: (1) o *relógio do sistema* (classe `Clock`); (2) os *contadores estatísticos* (classe `Statistics`); e (3) o *estado do sistema* (classe `Building`). Na ocorrência de um evento, estas três entidades devem ser notificadas e cada uma irá alterar seu estado interno da

forma adequada. Para isto, devem implementar a interface `EventObserver` e registrarem-se no `EventDispatcher`. Assim, o `EventDispatcher` e a `EventQueue` podem, juntos, notificar aos componentes reativos exatamente qual evento ocorreu em cada iteração da simulação, na ordem correta dos eventos.

6.3 Relógio da simulação

Durante a execução da simulação, à medida que eventos ocorrem, componentes do simulador devem alterar seu estado interno de acordo com o evento ocorrido, levando o estado do simulador a uma nova situação. Um destes componentes é o *relógio da simulação*. Portanto, deve realizar a classe concreta `EventObserver`.

O *relógio da simulação* é representado pela classe `Clock` (Figura 6.6). Esta classe encapsula o atributo privado `time` e provê métodos para sua consulta e atualização.

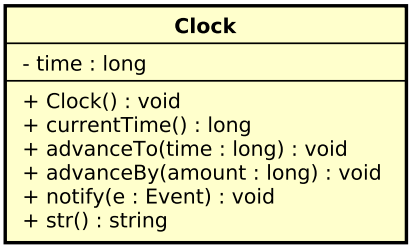

Figura 6.6 – Diagrama de classes do *relógio do sistema*.

`currentTime` Retorna o horário atual do *relógio da simulação*.

`advanceTo` Avança o *relógio da simulação* para um horário arbitrário.

`advanceBy` Avança o *relógio da simulação* em uma quantidade arbitrária de segundos.

`str` Representação textual do *relógio da simulação* (utilizado em *logs*).

`notify` Realização da classe abstrata `EventObserver`. Ao ser notificado de um evento, o *relógio da simulação* deve avançar o relógio interno para o instante da ocorrência do evento, independentemente do tipo de evento que ocorreu. O Algoritmo 6.2 ilustra este conceito.

```
1 Clock::notify(event) {
2   eventTime = event.getTime();
3   advanceTo(eventTime);
4 }
```

Algoritmo 6.2 – *Relógio do sistema* reagindo a um evento.

6.4 Contadores estatísticos

Os *contadores estatísticos* da simulação são responsáveis por coletar e sumarizar dados do sistema durante toda a execução da simulação. Sua existência permite a realização de análises qualitativas e quantitativas a respeito do sistema simulado. Assim como o *relógio da simulação*, os *contadores estatísticos* também devem ter o seu estado alterado na ocorrência de eventos. Portanto, devem realizar a classe concreta `EventObserver`.

Neste projeto, os *contadores estatísticos* são representados pelas estruturas `Arrival` e `Trip` e pela classe `Statistics` (Figura 6.7).

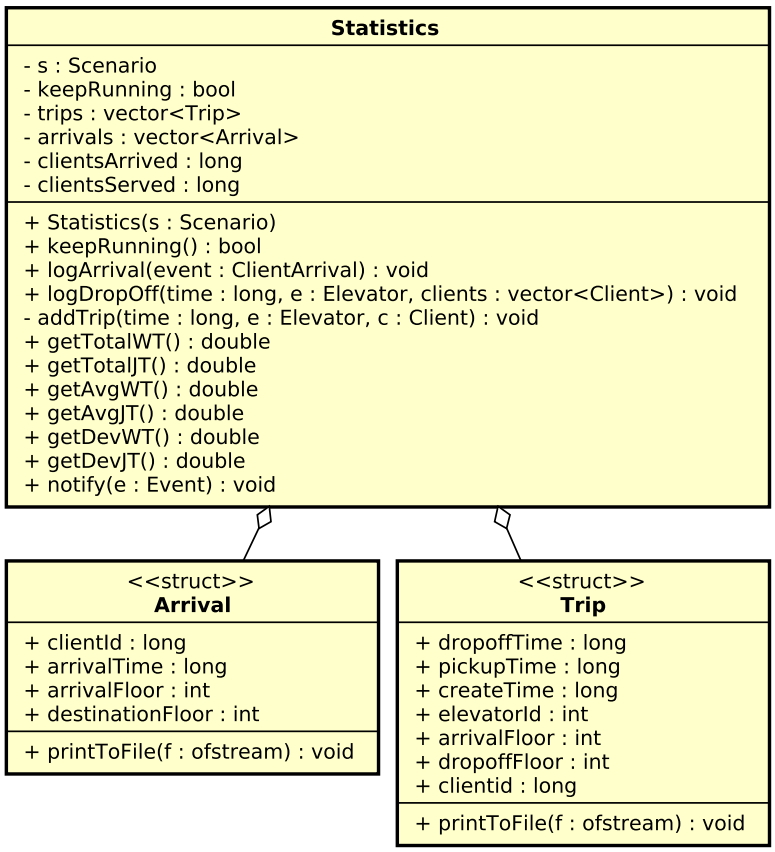

Figura 6.7 – Diagrama de classes dos *contadores estatísticos*.

Arrival

Uma instância desta estrutura armazena informações coletadas quando um cliente chega ao prédio. Tais informações são:

`clientId` Número identificador do cliente.

`arrivalTime` Horário em que o cliente chegou ao prédio.

`arrivalFloor` Número do andar no qual o cliente chegou.

`destinationFloor` Andar de destino ao qual o cliente deseja dirigir-se.

Trip

Uma instância desta estrutura armazena informações coletadas quando um cliente desembarca de um elevador. Tais informações são:

`clientId`	Número identificador do cliente.
`dropoffTime`	Horário em que o cliente desembarcou do elevador.
`createTime`	Horário em que o cliente chegou ao prédio.
`elevatorId`	Número do elevador do qual o cliente desembarcou.
`arrivalFloor`	Número do andar no qual o cliente chegou.
`dropoffFloor`	Número do andar no qual o cliente desembarcou.

Statistics

Armazena as estatísticas coletadas durante a simulação e fornece estatísticas sobre os dados coletados no fim da simulação. Seus métodos são:

`logDropOff`	Registra a ocorrência de um desembarque.
`logTrip`	Registra a ocorrência de uma chegada de um cliente.
`getAvgWT`	Calcula o tempo de espera médio.
`getDevWt`	Calculo o desvio padrão do tempo de espera.
`getTotalWT`	Calcula o tempo de espera total.
`getAvgWT`	Calcula o tempo de jornada médio.
`getDevWt`	Calcula o desvio padrão do tempo de jornada.
`getTotalWT`	Calcula o tempo de jornada total.
`keepRunning`	Avalia se a simulação deve terminar[4].
`notify`	Realização da classe abstrata `EventObserver`. Ao ser notificado de um evento: se for do tipo **chegada de cliente**, deve coletar e armazenar as informações desta nova chegada; se for do tipo **fim da simulação**, deve altera o estado do atributo privado `keepRunning` para `false`. Esta ação irá interromper a chegada de novos clientes. O Algoritmo 6.3 ilustra este conceito.

6.5 Simulador

O uso conjunto de todos estes componentes e classes dá forma ao simulador, conforme proposto por [12, 7]. Entretanto, todas as interações entre as instâncias devem ser organizadas conforme o fluxo proposto na Figura 5.4. O componente responsável por

[4]por exemplo, se o tempo de duração da simulação já se passou.

```
1 Statistics::notify(event) {
2   if (event.getType() == ClientArrival) {
3     logArrival(event);
4   }
5
6   if (event.getType() == FinishSimulation) {
7     keepRunning = false;
8   }
9 }
```

Algoritmo 6.3 – *Contadores estatísticos* reagindo a um evento.

orquestrar estas interações e garantir o fluxo de execução é representado pela classe *Simulator*. Esta classe não possui atributos ou métodos específicos; somente as instâncias das outras classes e um método que implementa o laço de execução do simulador.

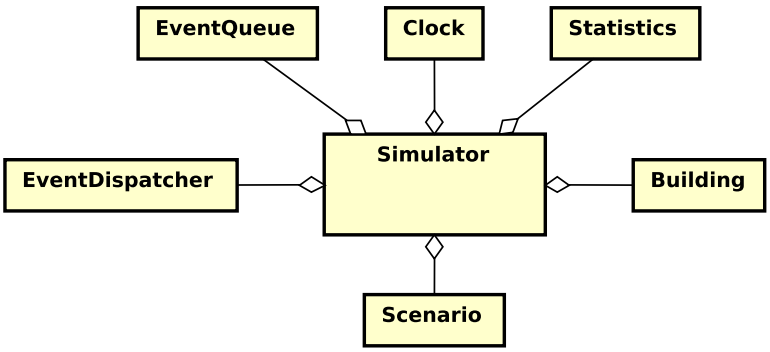

Figura 6.8 – Diagrama de classes do simulador.

O Algoritmo 6.4 mostra como funciona o laço de execução da simulação. Na linha 2, uma instância da classe `Building` (Seção 6.7.5) é criada baseada nos atributos definidos para o cenário 6.6. Nas linhas 4 a 6, os objetos *observers* se registram no *notifier* para receberem notificações de eventos. É importante frisar que a ordem na qual se registram é a mesma ordem na qual serão notificados da ocorrência de eventos. Nas linhas 8 a 10, é criado um evento para marcar o fim da simulação e o mesmo é adicionado à *fila de eventos*. Na linha 12, são inicializados os eventos de chegada de clientes em cada um dos andares (Seção 6.7.5). Por fim, o simulador entra em um laço indefinido nas linhas 14 a 17. A cada iteração do laço é retirado o próximo evento da fila e os *observers* são notificados. O laço permanece em execução enquanto houver eventos disponíveis na fila ou até atingir o tempo limite da simulação.

6.6 Cenário

Até este ponto foram apresentados componentes genéricos de um simulador. A partir daqui serão detalhados componentes específicos de um simulador de elevadores. O

```
1  Simulator::run() {
2    building = new Building(scenario);
3
4    dispatcher.subscribe(building);
5    dispatcher.subscribe(clock);
6    dispatcher.subscribe(statistics);
7
8    duration = scenario.getDuration();
9    finishSimulation = new FinishSimulation(duration);
10   eventQueue.push(finishSimulation);
11
12   building.initializeClientArrivals(eventQueue);
13
14   while (statistics.keepRunning() and eventQueue.hasNextEvent()) {
15     nextEvent = eventQueue.pop();
16     dispatcher.broadcast(nextEvent);
17   }
18 }
```

Algoritmo 6.4 – Laço de execução da simulação.

primeiro deles é o *cenário*. Como entrada o simulador recebe um conjunto de cenários e realiza a simulação de cada um deles. Um cenário é composto pelas seguintes informações:

Nome	Identificação textual do cenário.
Duração	Quantidade de unidades de tempo durante a qual serão geradas chegadas de clientes aos andares do prédio. Após esta duração, não chegarão mais clientes. Porém, clientes que já entraram no prédio e estão distribuídos nos andares e elevadores precisam terminar suas viagens antes que a simulação chegue ao fim. Devido a isto, é normal que o duração final seja maior que a duração estipulada no cenário.
Agendamento	Lista de algoritmos de agendamento.
Horizonte	Horizonte de expansão do agendamento com algoritmo de *planning*.
Função de Custo	Lista de funções de custo.
Semente	Semente textual para inicialização os geradores de números aleatórias.
Elevadores	Número de elevadores do prédio.
Capacidade	Capacidade dos elevadores (a mesma para cada elevador).
Andares	Lista com o intervalo médio[5] de chegada de passageiros, em segundos, para cada andar. A lista começa com a média do andar térreo e segue sucessivamente. O número de andares é igual ao tamanho da lista.

[5] O valor médio informado é utilizado como parâmetro λ de entrada para uma Distribuição de Poisson. Cada andar do prédio possui uma distribuição distinta.

O parâmetro *função de custo* será utilizado no agendamento *Simple*[6] e o parâmetro *horizonte* no agendamento *Planning*.

6.6.1 Arquivo de Configuração

A entrada de dados para o simulador se dá através de um arquivo de configuração chamado `config.yaml`. Este arquivo obedece o padrão *YAML*, um formato de serialização de dados legíveis por humanos.

```
scenarios:
  - name: Scenario 1
    duration: 86400
    scheduler: [ 0, 1 ] # simple, planning
    planningHorizon: 5
    cost_function: [ 0, 2, 3 ] # randon, bnn, weighted
    seed: 54TH7hboAG1iOsDIDhJp
    elevators: 2
    capacity: 6
    floors: [ 60, 520, 360, 240, 240, 90, 90, 90 ]

  - name: Scenario 2
    # ...

  - name: Scenario N
    # ...
```

Listagem 1: Exemplo de arquivo de configuração `config.yaml`.

No exemplo da Listagem 1 são definidos alguns cenários. No primeiro, chamado `Scenario 1`, clientes chegarão durante 12 horas distribuídos ao longo dos 8 andares do prédio e serão atendidos por 2 elevadores, cuja capacidade é de 8 passageiros cada. Pode ser definido um número ilimitado de cenários no arquivo de entrada.

Para representar as informações referentes a um cenário foi criada a classe `Scenario` (Figura 6.9). Além de armazenar as informações, disponibiliza o método `Load`, cujo objetivo é carregar os cenários a partir do arquivo de configuração utilizando a biblioteca `yaml-cpp`.

6.7 Estado do sistema

Entre os componentes fundamentais de um simulador destaca-se a representação do *estado do sistema*, uma coleção de objetos e variáveis necessárias para descrever o sistema em um instante em particular da simulação [12]. Neste projeto, a classe

[6] Será realizada uma simulação para cada função de custo.

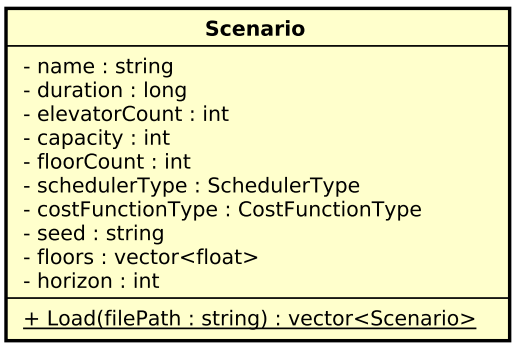

Figura 6.9 – Diagrama de classes dos *cenários*.

`Building` é responsável por encapsular o conjunto de informações que definem este estado (Figura 6.11). Esta classe é responsável por gerenciar múltiplas instâncias de elevadores (classe `Elevator`), andares (classe `Floor`) e clientes (classe `Client`) e relacionar estas instâncias entre si - reproduzindo, deste modo, as dinâmicas do sistema do mundo real que está sendo simulado.

6.7.1 Cliente

Uma pessoa que chegou a um andar e deseja se dirigir a outro andar é representada, dentro do simulador, por um objeto da classe `Client`. Possui os seguintes atributos:

`id` Número identificador do cliente.

`destination` Número do andar ao qual o cliente deseja dirigir-se.

`arrivalFloor` Número do andar no qual o cliente chegou.

`createTime` Horário em que o cliente chegou ao prédio.

`pickupTime` Horário em que o cliente embarcou em um elevador.

6.7.2 Andar

Parte componente de um prédio, representada pela classe `Floor`. Possui os seguintes atributos:

`number` Número do andar em que a parada deve ser realizada.

`lambda` Valor de entrada para a distribuição de Poisson do andar.

`upLine` Fila contendo os clientes que desejam subir[7] a partir do andar.

[7] No mundo real, apesar de aparentemente as pessoas formarem uma fila única, os membros da fila respeitam o sentido de viagem do elevador e implicitamente separam-se em duas filas: uma para subir e outra para descer.

`downLine` Fila contendo os clientes que desejam descer a partir do andar.

`eventFactory` Instância da classe `EventFactory` (Seção 6.2.1) do andar.

6.7.3 Elevador

Um elevador é representado pela classe `Elevator`. Possui os seguintes atributos:

`number` Número identificador do elevador.

`capacity` Capacidade máxima do elevador (em número de clientes).

`location` Número do andar no qual o elevador se encontra.

`destination` Parada[8] de destino do elevador, composto pelo número do andar e a direção da parada.

`status` Situação atual do elevador: *moving* (movendo-se) ou *idle* (ocioso).

`direction` Direção na qual o elevador está se movendo (no caso de não estar ocioso).

`passengers` Conjunto com os passageiros que embarcaram no elevador e ainda não desembarcaram.

6.7.4 Gerenciador de Paradas

Para tornar a tarefa de controlar paradas mais simples, foi criada uma classe auxiliar que implementa um gerenciador de paradas chamada `StopManager`. Expõe as seguintes funcionalidades:

`hasStop` Verifica se há algum elevador programado para parar em dado andar e direção.

`set` Adiciona uma parada em dado andar e direção à lista de paradas do elevador.

`clear` Remove um parada em dado andar e direção da lista de paradas do elevador.

`getStops` Retorna o conjunto de andares com paradas programadas para um elevador em dada direção.

[8] Por exemplo, parar no andar 5 subindo é diferente de parar no andar 5 descendo.

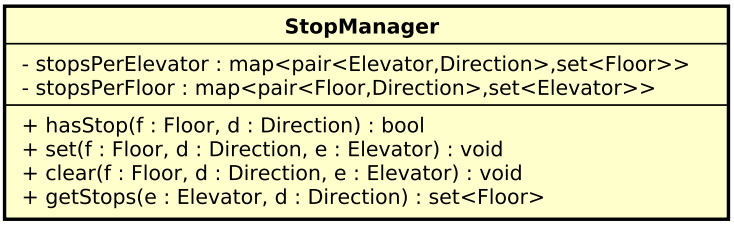

Figura 6.10 – Diagrama de classes do *gerenciador de paradas*.

6.7.5 Prédio

O *prédio* sendo simulado é representado pela classe `Building`. Esta classe agrupa os demais componentes do prédio, possuindo um conjunto de andares e elevadores. Assim como o *relógio da simulação* e os *contadores estatísticos*, o *prédio* também terá seu estado alterado na ocorrência de eventos. Portanto, a classe `Building` deve implementar a interface para *observers*. As responsabilidades desta classe estão expostas para o simulador da seguinte forma:

Inicializar a lista de chegadas de clientes

Conforme visto nas Seções 6.2.1 e 6.7.2, há um objeto da classe `EventFactory` para cada objeto `Floor` do prédio. O objeto `Building` é responsável por realizar a invocação do `EventFactory` para criar novos eventos de chegada de cliente e adicioná-los à fila de eventos. O Algoritmo 6.5 mostra um excerto do código da classe `Building`, responsável por realizar esta inicialização. O método `initializeArrivals` itera por todos os andares do prédio e invoca a criação de um novo evento, passando como parâmetro um ponteiro para a fila de eventos para que o `EventFactory` consiga adicionar o evento recém criado (Algoritmo 6.1).

```
1 Building::initializeArrivals() {
2   eventQueue = simulator.getEventQueue();
3   for each (floor in floors)
4     floor.createFutureArrival(eventQueue);
5 }
```

Algoritmo 6.5 – Inicialização dos eventos de chegada de cliente.

Reagir à ocorrência de eventos

Como o estado interno deve ser alterado na ocorrência de eventos, o `Building` implementa a interface `EventObserver` conforme o Algoritmo 6.6. Nas linhas 2 a 4 do método é feita a atualização do estado interno do prédio para refletir o que ocorreu no ínterim entre o momento do último evento processado e o que está sendo notificado - como o *prédio* recebe notificações antes do *relógio do sistema*, o segundo ainda está no estado referente ao evento ocorrido anteriormente. Com o estado do prédio atua-

lizado, o evento recebido pode ser processado de acordo com o seu tipo (linhas 7 e 11).

```
1  Building::notify(event) {
2    for (time = clock.currentTime(); time < event.getTime(); ++time) {
3      step();
4    }
5
6    if (event.getType() == ClientArrival) {
7      doClientArrival(event);
8    }
9
10   if (event.getType() == FinishSimulation) {
11     doFinishSimulation(event);
12   }
13 }
```

Algoritmo 6.6 – *Prédio* reagindo a um evento.

Reagir à chegada de um novo cliente

O Algoritmo 6.7 ilustra como o `Building` trata o evento de *chegada de cliente*. Nas linhas 2 a 4 são identificados o cliente, a direção em que irá se locomover e o andar em que ocorreu o evento de sua chegada. Na linha 5 o cliente é adicionado à fila do andar correspondente. Caso não exista uma parada programada de um elevador no andar e na direção (linha 7), será necessário escolher um elevador para atender este cliente. Neste momento, o *algoritmo de agendamento* é invocado (linha 8) e a decisão é informada ao *gerenciador de paradas* (linha 9). Por fim, nas linhas 12 a 13 é gerado um novo evento futuro de *chegada de cliente* para o andar onde o evento ocorreu, de modo a repopular as filas de cada andar.

```
1  Building::doClientArrival(event) {
2    client = event.getClient();
3    direction = client.getDirection();
4    location = client.getArrivalFloor();
5    location.addClient(client);
6
7    if (!stopManager.hasStop(location, direction)) {
8      elevator = scheduler.schedule(costFunction, building, client);
9      stopManager.set(location, direction, elevator);
10   }
11
12   eventQueue = simulator.getEventQueue();
13   location.createFutureArrival(eventQueue);
14 }
```

Algoritmo 6.7 – *Prédio* reagindo a uma *chegada de cliente*.

Reagir ao fim da simulação

Quando ocorre o evento de *fim da simulação*, não serão mais gerados eventos de

chegadas de clientes. Porém, ainda existirão clientes dentro do prédio, seja esperando por um elevador ou dentro de um deles. O prédio deve atualizar o seu estado interno e gerar estatísticas até que o último cliente seja atendido - ou seja, desembarque no seu andar de destino. O Algoritmo 6.8 mostra este comportamento, onde o prédio atualiza o seu estado de forma indefinida, até que todos os clientes sejam atendidos.

```
1 Building::doFinishSimulation(event) {
2   statistics = simulator.getStatistics();
3   while (statistics.getClientsArrived() > statistics.getClientsServed()) {
4     step();
5   }
6 }
```

Algoritmo 6.8 – *Prédio* reagindo ao *fim da simulação*.

Atualizar o seu estado interno

Atualizar o estado interno do prédio é uma tarefa essencial para o correto funcionamento de um simulador de elevadores. O Algoritmo 6.9 avança este estado em 1 unidade de tempo no futuro. Em `step` é realizado o incremento do valor do *relógio da simulação* e iterado sobre cada um dos elevadores do prédio, atualizando a situação de cada um deles ao invocar o método `updateElevator`.

```
1 Building::step() {
2   clock.advanceBy(1);
3   for each (elevator in elevators) {
4     updateElevator(elevator);
5   }
6 }
```

Algoritmo 6.9 – *Prédio* atualizando seu *estado interno* em 1 *unidade de tempo*.

Já o Algoritmo 6.10 detalha esta atualização no contexto de um elevador. Nas linhas 2 a 5 é feita a atribuição de um novo destino (se houver) para o elevador caso ele esteja ocioso (*idle*). Internamente, este método também já define a direção de movimento baseada na localização atual do elevador e na do destino. Após, o elevador se move[9] (linha 7).

Se, após mover-se, houver uma parada programada deste elevador no andar e na direção em que o mesmo se encontra, esta parada é realizada marcada como realizada no *gerenciador de paradas* (linha 12). Após é realizado o desembarque de passageiros se dirigindo ao andar atual (linha 14) e embarque de novos passageiros (linha 15).

[9] A menos que esteja ocioso. Neste caso, permanece no local onde se encontra.

Neste momento há uma peculiaridade. Se, após o embarque, ainda houver clientes na fila daquele andar naquela direção, significa que a ocupação do elevador lotou e estas pessoas ainda não conseguiram embarcar. Neste caso, é feita uma nova execução do *algoritmo de agendamento* - porém desta vez excluindo das opções o elevador que acabou de parar ali (linhas 17 a 22).

```
1  Building::updateElevator(elevator) {
2    if (elevator.getStatus() == Idle) {
3      nextDestination = getNextDestination(elevator);
4      elevator.setDestination(nextDestination);
5    }
6
7    elevator.move();
8    location = elevator.getLocation();
9    direction = elevator.getDirection();
10
11   if (mustStop(elevator, location, direction)) {
12     stopManager.clear(location, elevator, direction);
13
14     elevator.dropPassengers();
15     location.boardElevator(elevator, direction, stopManager);
16
17     if (!location.getLine(direction).empty()) {
18       if (!stopManager.hasStop(location, direction)) {
19         client = location.getLine().front();
20         newElevator = scheduler.schedule(costFunction, building, client, elevator);
21         stopManager.set(location, direction, newElevator);
22       }
23     }
24   }
25 }
```

Algoritmo 6.10 – Atualizando o *estado interno* de um *elevador*.

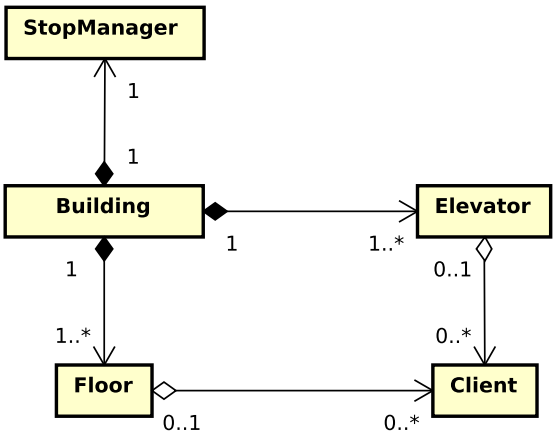

Figura 6.11 – Diagrama de classes do *prédio*.

6.8 Algoritmos de Agendamento

Neste trabalho, dois algoritmos de agendamento foram implementados, o *Simple Scheduler* (Seção 6.8.1) e o *Planning Scheduler* (Seção 6.8.2). Ambos são implementados em classes próprias, que herdam de uma classe `Scheduler`. Este relacionamento pode ser visto mais claramente na Figura 6.12.

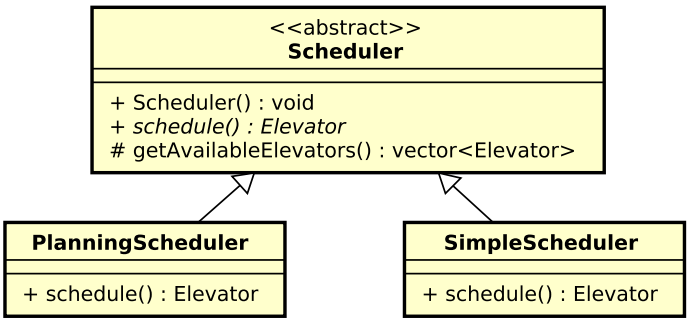

Figura 6.12 – Diagrama de classes das *estratégias de agendamento*.

Os *schedulers* têm apenas um método público, chamado `int schedule()`, que recebe como parâmetro a função de custo e um ponteiro para o prédio, além de, opcionalmente, um elevador para ser excluído.

Excluir um elevador é importante, como foi falado na Seção 6.10. Há um método protegido na classe `Scheduler`, chamado `getAvailableElevators()`, que retorna uma lista de elevadores, excluindo aqueles que não podem ser utilizados.

6.8.1 Simple

O *Simple Scheduler*, como o nome sugere, tem um comportamento bem simples: itera pela lista de elevadores disponíveis, calculando a função de custo para cada um, e retorna o de menor custo.

6.8.2 Planning

A ideia por trás do algoritmo de *planning* é simples. O objetivo é, a cada ramo da árvore de decisões, criar uma cópia do simulador e avançar o seu estado interno até que a parada tenha sido realizada. O tempo necessário para isto é medido e a escolha se dá sempre pelo elevador capaz de atender no menor tempo. Este conceito é ilustrados nos Algoritmos 6.12 e 6.13.

```
1  SimpleScheduler::schedule(costFunction, building, client, excluded) {
2    elevators = getAvailableElevators(building, excluded);
3    bestCost = infinite;
4    selectedElevator = elevators.front();
5
6    for each (elevator in elevators) {
7      cost = costFunction.calculate(building, elevator, client);
8      if (cost < best_cost) {
9        bestCost = cost;
10       selectedElevator = elevator;
11     }
12   }
13
14   return selectedElevator;
15 }
```

Algoritmo 6.11 – Agendamento *simple*.

No começo do Algoritmo 6.12 recupera-se o horizonte parametrizado no cenário e é montada uma fila de clientes do tamanho do horizonte. Esta fila de clientes é composta pelo cliente que acabou de chegar e seguido pelos *N* primeiros clientes que chegaram ao prédio e ainda não foram atendidos. São estes clientes que o *planning* irá utilizar para realizar a sua decisão através do método recursivo `calculate` (Algoritmo 6.13), consumindo 1 cliente por nível da árvore.

O método recursivo `calculate` começa com a condição de parada na linha 3, retornando no caso de não haver mais clientes. Caso ainda existam clientes na fila, as linhas 4 e 5 recuperam o primeiro e o retiram da fila. Este é o cliente referente à este nível da árvore de decisão. Nas linhas 7 a 11, a lista de elevadores é atualizada.

Após, é feita uma iteração entre estes elevadores. A cada iteração é realizada uma nova cópia do simulador e registra-se uma nova parada para o andar e direção do cliente e o elevador da iteração. Então avança-se esta nova cópia da simulação até o instante em que esta parada registrada é atendida. O custo base deste elevador é proporcional à diferença do *relógio da simulação* antes e depois deste avanço.

A este valor soma-se o seu filho de menor custo através da chamada recursiva do método *calculate* na linha 29. Desta forma, no retorno da recursão, tem- se sempre o menor custo para um nodo. Ao fim da recursão - *i.e.* quando todos os clientes da fila original forem atendidos - , tem-se o elevador de menor custo.

```
1 PlanningScheduler::schedule(costFunction, building, client, excluded) {
2   horizon = scenario.getPlanningHorizon();
3   clients = getClients(horizon, client, building);
4
5   result = calculate(simulator, clients, excluded);
6
7   return result.elevator;
8 }
```

Algoritmo 6.12 – Agendamento *planning*.

6.9 Algoritmos de Função de Custo

As funções de custo herdam de uma classe base, chamada `CostFunction`, que possui apenas um método público, `float calculate()`, que retorna o valor da função para aquela combinação entre o elevador e o cliente. Isto pode ser visto em mais detalhes na Figura 6.13.

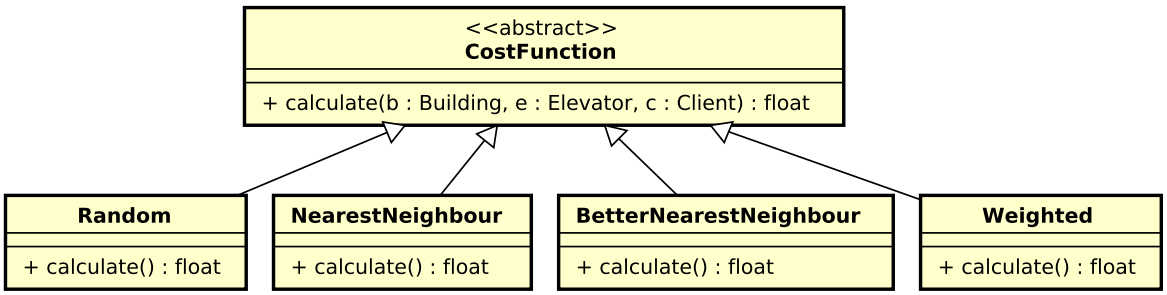

Figura 6.13 – Diagrama de classes das *funções de custo*.

6.9.1 Random

A Função de Custo *Random* serve como base de comparação para as demais. Ela retorna um valor aleatório entre zero e o número de andares do prédio cada vez que for chamada, e, portanto, espera-se como resultado um agendamento não ótimo.

Olhando-se a Figura 6.13, nota-se que ela é a única função de custo com um construtor. Isto se dá por que é necessário inicializar-se a geração de números aleatórios, de modo a gerar estes números de forma consistente.

```
PlanningScheduler::calculate(simulator, clients, excluded) {

  if (clients.empty()) return { null, 0.0 };
  client = clients.front();
  clients.pop();

  building = simulator.getBuilding();
  elevators = getAvailableElevators(building, elevatorToExclude);

  bestCost = infinite;
  selectedElevator = elevators.front();

  for each (elevator in elevators) {
    simulatorCopy = copy(simulator);

    arrivalFloor = client.getArrivalFloor();
    direction = client.getDirection();
    simulatorCopy.setStop(arrivalFloor, direction, elevator);

    clockBefore = simulatorCopy.getClock().currentTime();
    while (simulatorCopy.mustStop(arrivalFloor, direction, elevator)) {
      simulatorCopy.step();
    }
    clockNow = clockBefore = simulatorCopy.getClock().currentTime();

    baseCost = clockBefore - clockNow;

    clientsCopy = copy(clients);
    futureCost = calculate(simulatorCopy, clientsCopy, elevatorToExclude).cost;

    cost = baseCost + futureCost;
    if (cost < bestCost) {
      bestCost = cost;
      selectedElevator = elevator;
    }
  }

  return { selectedElevator, bestCost };
}
```

Algoritmo 6.13 – Recursão do agendamento *planning*.

6.9.2 Nearest Neighbour

Esta função de custo retorna, como custo, a distância do elevador para o cliente. Ela ignora a direção na qual o elevador está viajando.

```
float RandomCostFunction::calculate(
    Building building,
    Elevator elevator,
    Client client) {

  if (distribution == null) {
    number_of_floors = scenario.getFloorCount();
    distribution = (0.0, number_of_floors);
  }

  return distribution();
}
```

Algoritmo 6.14 – Função de custo *random*.

```
float NearestNeighbourCostFunction::calculate(
      Building building,
      Elevator elevator,
      Client client) {

    where_it_is = elevator.getLocation();
    where_to = client.getArrivalFloor();
    distance = where_it_is - where_to;

    return abs(distance);
}
```

Algoritmo 6.15 – Função de custo *nearest neighbour*.

6.9.3 Better Nearest Neighbour

A Função de Custo *Better Nearest Neighbour* bonifica elevadores que estão indo na direção do chamado, dividindo seu custo por $\sqrt{2}$ e bonifica mais ainda os elevadores parados, dividindo seu custo por 2. O custo antes deste bônus é calculado de maneira igual à *Nearest Neighbour*.

6.9.4 Weighted

A Função *Weighted* toma uma decisão diferente da *Better Nearest Neighbour* para melhorar a *Nearest Neighbour*. Em vez de bonificar elevadores parados ou elevadores que estão indo na direção do cliente, esta função bonifica elevadores mais vazios, dividindo o custo (que é a distância entre o cliente e o elevador) pela ocupação do elevador.[10]

[10]Como foi visto no Capítulo 4, podemos inferir a ocupação pela balança interna do elevador, em alguns cenários reais.

```
float BetterNearestNeighbourCostFunction::calculate(
      Building building,
      Elevator elevator,
      Client client) {

    where_it_is = elevator.getLocation();
    where_to = client.getArrivalFloor();
    distance = where_it_is - where_to;

    floors = building.getFloors();
    current_floor = floors.at(where_it_is);
    request_floor = floors.at(where_to);

    if (elevator.getStatus() == Idle)
      return abs(distance) / sqrt(4.0);

    if (elevator.getDirection() == current_floor.compareTo(request_floor))
      return abs(distance) / sqrt(2.0);

    return abs(distance);
}
```

Algoritmo 6.16 – Função de custo *better nearest neighbour*.

```
float WeightedCostFunction::calculate(
      Building building,
      Elevator elevator,
      Client client) {

    if (elevator.getOccupation() == 0.0)
      return abs(client.getArrivalFloor() - elevator.getLocation());

    return abs(client.getArrivalFloor() - elevator.getLocation())
               / sqrt(elevator.getOccupation());
}
```

Algoritmo 6.17 – Função de custo *weighted*.

6.10 Geração de Relatórios

A cada fim de simulação, os *contadores estatísticos* resultantes são armazenados em uma coleção dentro da classe `Reporter`. Esta classe é responsável por gerar relatórios da simulação. A listagem 2 mostra um exemplo de estrutura de diretórios e arquivos de saída.

```
output
└── High-rise ..... Nome do cenário.
    ├── 1 Simple Random ..... Diretório da estratégia.
    │   ├── run.log ..... Log de execução da estratégia.
    │   ├── report.log ..... Relatório das métricas da estratégia.
    │   └── trips.log ..... Arquivo contendo os desembarques de clientes.
    ├── ...
    ├── arrivals.log ..... Arquivo contendo as chegadas de clientes.
    └── report.log ..... Relatório unificado de todas as estratégias do cenário.
```

Listagem 2: Estrutura de diretórios e arquivos de saída.

Log de Execução O *log de execução* apresenta o histórico da simulação, mostrando sequencialmente o fluxo de processamento, ocorrência de eventos e registro de decisões realizadas dentro do simulador.

Métricas da Estratégia O *relatório de métricas da estratégia* apresenta as métricas calculadas para a simulação do cenário utilizando uma estratégia de agendamento específica.

Arquivo de Desembarques O *arquivo de desembarques* apresenta os dados de todos os desembarques realizados para a simulação do cenário utilizando uma estratégia de agendamento específica. Este arquivo é utilizado para a geração de gráficos.

Arquivo de Chegadas O *arquivo de chegadas* apresenta os dados de todas as chegadas de clientes para a simulação do cenário utilizando uma estratégia de agendamento específica. Este arquivo é utilizado para a geração de gráficos.

Relatório Unificado O *relatório unificado* apresenta as métricas geradas por todas as estratégias selecionadas para simulação de um cenário. Assim, fornece um meio de observar e analisar os resultados obtidos.

6.10.1 Geração de Gráficos

Os gráficos são gerados a partir do arquivo de desembarques. A Listagem 3 mostra um exemplo de estrutura de diretórios e arquivos de saída.

```
output
└── High-rise ..... Nome do cenário.
    └── 1 Simple Random ..... Diretório da estratégia.
        ├── arrivalsPerFloor.eps ..... Chegadas por andar.
        ├── averageTravelTime.eps ..... Tempo médio de jornada de andar para andar.
        ├── averageWaitTime.eps ..... Tempo médio de espera por andar.
        ├── clientsPerElevator.eps ..... Total de clientes em cada elevador.
        └── dropoffsPerFloor.eps ..... Total de clientes entregues em cada andar.
```

Listagem 3: Estrutura de diretórios e arquivos de saída.

Um script em Python é chamado para cada arquivo de desembarques, de modo a processá-lo e gerar estes gráficos. Este script lê o arquivo, que está em formato CSV, e chama uma função para cada tipo de gráfico da Listagem 3. Exemplos destes gráficos podem ser vistos no Apêndice A.

7 RESULTADOS

Para os cenários de teste definidos na Seção 3.1 foram realizadas simulações. Os resultados são apresentados a seguir.

7.1 Cenário *Low-rise*

A Listagem 4 exibe um excerto do arquivo de configuração que define o cenário *Low-rise*. A Tabela 7.1 mostra os resultados[1] obtidos após a simulação.

```
# ...
- name: Low-rise
  duration: 43200 # unidades de tempo
  scheduler: [ 0, 1 ] # simple, planning
  planningHorizon: 5
  cost_function: [ 0, 1, 2, 3 ] # randon, nn, bnn, weighted
  seed: 54TH7hboAG1iOsDIDhJp
  elevators: 2
  capacity: 2
  floors: [ 60, 520, 360, 360, 360, 240, 240, 240, 90, 90, 90 ]
# ...
```

Listagem 4: Configuração do cenário *Low-rise*.

Tabela 7.1 – Resultados para o cenário *Low-rise*.

	Tempo de Espera			**Tempo de Jornada**		
Estratégia	*Médio*	*Desvio*	*Total*	*Médio*	*Desvio*	*Total*
Simple / Random	5.4064	4.1792	16933	4.4949	2.8676	14078
Simple / NN	3.5144	3.5768	11007	4.4700	2.8565	14000
Simple / BNN	3.3547	3.2931	10507	4.4674	2.9163	13992
Simple / Weighted	3.7149	3.8036	11635	4.4719	2.7997	14006
Planning	3.1731	2.9171	9938	4.4770	3.0009	14022

O *tempo de espera médio* do *planning* foi apenas 5.41% melhor em relação ao segundo colocado (*better nearest neighbour*). Entretanto, somado ao fato do *desvio padrão* ter sido 12% melhor, é uma forte indicação de que *planning* é a melhor estratégia para este cenário.

[1] Destacados encontram-se os melhores resultados obtidos para cada métrica.

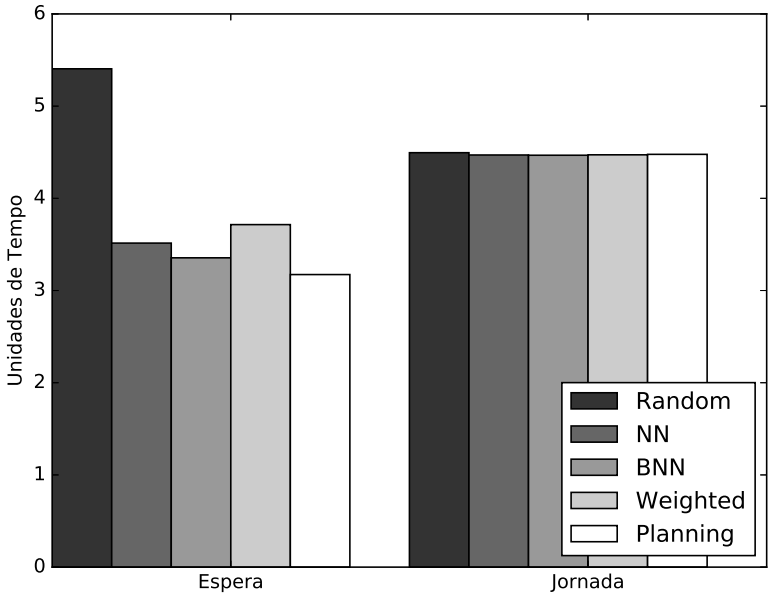

Figura 7.1 – Gráfico de resultados para o cenário *Low-rise*.

7.2 Cenário *High-rise*

A Listagem 5 exibe um excerto do arquivo de configuração que define o cenário *High-rise*. A Tabela 7.2 mostra os resultados[2] obtidos após a simulação.

```
1   # ...
2   - name: High-rise
3     duration: 43200 # unidades de tempo
4     scheduler: [ 0, 1 ]
5     planningHorizon: 2
6     cost_function: [ 0, 1, 2, 3 ]
7     seed: w9JwgykwejtoL2icSgHo
8     elevators: 8
9     capacity: 10
10    floors: [ 60, 520, 520, 520, 360, 360, 360, 360, 360, 360, 360, 360, 360,
11              360, 360, 360, 360, 360, 360, 360, 360, 360, 360, 360, 360, 360,
12              360, 360, 240, 240, 240, 240, 240, 240,  90,  90,  90,  90,  90 ]
13  # ...
```

Listagem 5: Configuração do cenário *High-rise*.

Em cenários maiores e com um maior número de clientes percorrendo o prédio, os ganhos em se utilizar *planning* ficam ainda mais evidentes. Neste cenário, seu *tempo de espera médio* foi 35.74% melhor em relação ao segundo colocado (*better nearest neighbour*). Já o *desvio padrão* foi 31.28% melhor.

[2]Destacados encontram-se os melhores resultados obtidos para cada métrica.

Tabela 7.2 – Resultados para o cenário *High-rise*.

	Tempo de Espera			Tempo de Jornada		
Estratégia	*Médio*	*Desvio*	*Total*	*Médio*	*Desvio*	*Total*
Simple / Random	25.9981	28.0464	190124	15.2595	14.9190	111593
Simple / NN	20.3395	37.1902	148743	15.0993	11.4479	110421
Simple / BNN	13.5243	23.5950	98903	15.0550	10.2476	110097
Simple / Weighted	31.0373	54.5909	226976	14.9915	18.9700	109633
Planning	8.6901	16.2139	63551	15.1627	12.0920	110885

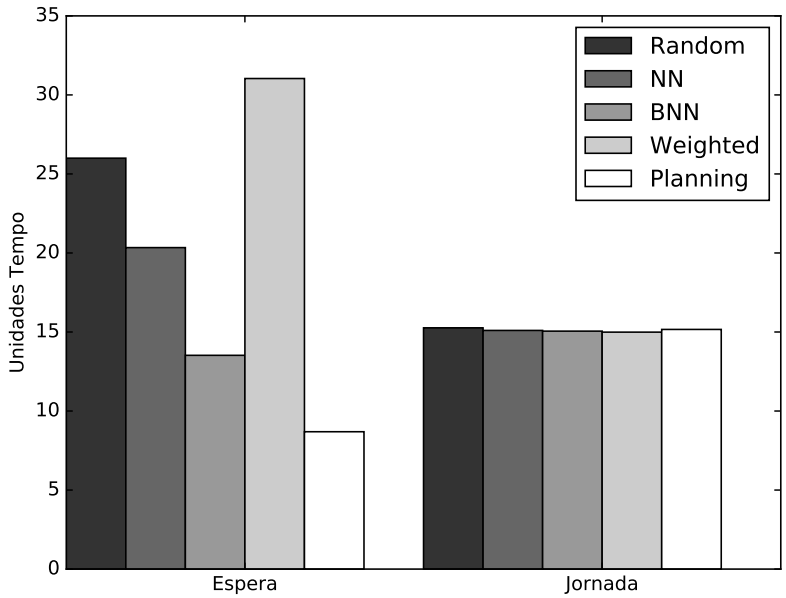

Figura 7.2 – Gráfico de resultados para o cenário *High-rise*.

7.3 Cenário *Skyscraper*

A Listagem 6 exibe um excerto do arquivo de configuração que define o cenário *Skyscraper*. A Tabela 7.3 mostra os resultados[3] obtidos após a simulação.

Tabela 7.3 – Resultados para o cenário *Skyscraper*.

	Tempo de Espera			Tempo de Jornada		
Estratégia	*Médio*	*Desvio*	*Total*	*Médio*	*Desvio*	*Total*
Simple / Random	442.2165	1775.3156	10538904	54.7816	389.3520	1305554
Simple / NN	343.5988	1426.3821	8188647	54.9789	291.2258	1310258
Simple / BNN	287.1803	1137.2342	6844080	55.0124	235.4007	1311056
Simple / Weighted	293.8791	1018.5118	7003726	54.6163	242.3278	1301616
Planning	78.3855	230.8554	1868084	60.1728	48.8070	1434038

Novamente, o crescimento do cenário - tanto em número de andares como em número de clientes - resultou em uma melhoria significativa dos resultados obtidos pelo

[3] Destacados encontram- se os melhores resultados obtidos para cada métrica.

```
  # ...
  - name: Skyscraper
    duration: 43200 # unidades de tempo
    scheduler: [ 0, 1 ]
    planningHorizon: 2
    cost_function: [ 0, 1, 2, 3 ]
    seed: NimatYvEnU9QeE3GkF4J
    elevators: 16
    capacity: 12
    floors: [ 150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450, 150,
              300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450, 150, 300,
              450, 600, 150, 300, 450, 600, 150, 300, 450, 450, 300, 450, 450,
              450, 150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450,
              150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450, 150,
              300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450, 300, 450,
              450, 450, 150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450,
              450, 150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450,
              150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450, 300,
              450, 450, 450, 150, 300, 450, 600, 150, 300, 450, 600, 150, 300,
              450, 450, 150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450,
              450, 150, 300, 450, 600, 150, 300, 450, 600, 150, 300, 450, 450,
              300, 450, 450, 450, 450, 450, 450 ]
  # ...
```

Listagem 6: Configuração do cenário *Skyscraper*.

planning. No maior de todos os cenários simulados, obteve-se *tempo de espera médio* e *desvio padrão* 72.71% e 79.70% superiores em relação ao segundo colocado, respectivamente.

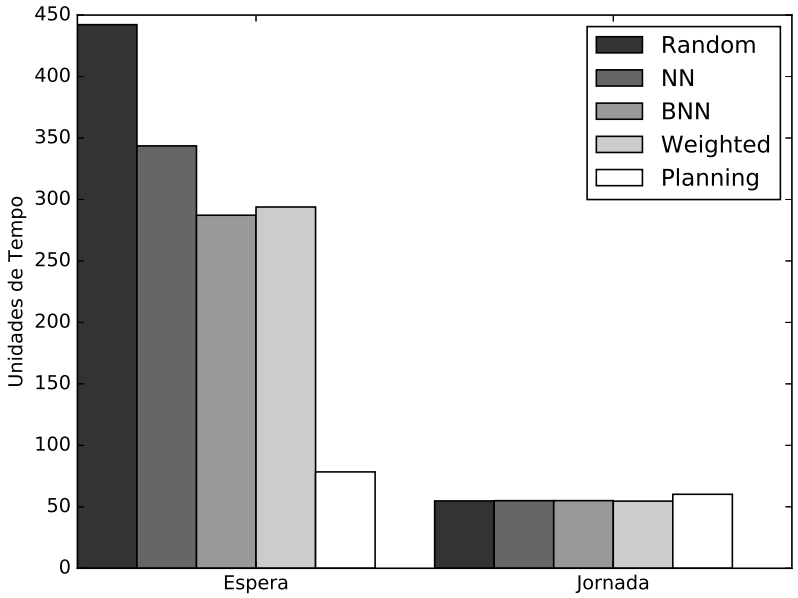

Figura 7.3 – Gráfico de resultados para o cenário *Skyscraper*.

7.4 Tempo de Decisão do *Planning*

A Tabela 7.4 contém estatísticas do tempo de execução do algoritmo, i.e. o tempo necessário para tomar a decisão de qual elevador deve atender um cliente. Esta estatística é importante pois fornece uma medida do impacto pela aplicação da solução em um sistema real: o agendamento seria executado a cada vez que um botão de chamada é pressionado em todo o prédio.

Tabela 7.4 – Tempo de decisão do *planning*.

	Low-rise	**High-rise**	**Skyscraper**
Mínimo	0.2220 ms	0.002097 s	0.03882 s
Mediana	0.2900 ms	0.031623 s	2.35068 s
Média	0.4137 ms	0.043263 s	2.12851 s
Máximo	3.6810 ms	0.358810 s	4.99215 s

Observa-se que o tempo é muito reduzido para os cenários *Low-rise* e *High-rise*, causando quase nenhum impacto. Já para o cenário *Skyscraper* os números parecem ruins a primeira vista. Porém, ao considerar os resultados para tempo de espera, um tempo médio de 2.13 segundos e um máximo de 4.99 segundos não causam prejuízo significativo. Por estas razões, o desempenho do *planning* foi considerado plenamente satisfatório.

Mais gráficos a respeito dos resultados podem ser encontrados no Apêndice B.

8 CONCLUSÃO

Após a realização deste estudo, foi possível enxergar claramente a simulação como uma excelente ferramenta para estudar e experimentar sistemas. No objeto deste trabalho, os resultados obtidos através das simulações evidenciam que é possível diminuir o tempo médio de espera de sistemas de elevadores substituindo o software do sistema de controle dos mesmos por alternativas que utilizem técnicas algorítmicas mais avançadas em relação às utilizadas pela indústria. Deste modo, é possível beneficiar não só usuários de futuras instalações, mas também usuários de sistemas já existentes.

Neste contexto, a escolha adequada de uma estratégia de agendamento possui alta relevância. Conforme as estatísticas apresentadas no Capítulo 7 mostram, a utilização de uma estratégia equivocada para determinado cenário pode causar um aumento significativo no tempo de espera médio. Pode-se verificar a validade desta afirmação ao notar que, nos cenários *low-rise*, *high-rise* e *skyscraper*, a melhor estratégia foi 41.31%, 66.57% e 82.27% superior à pior opção, respectivamente.

Dentre as estratégias construídas durante este trabalho, o *Planning* obteve destaque positivo nos resultados. Ao avaliar esta estratégia obteve-se o menor tempo de espera médio em todos os cenários. Os resultados foram 5.41%, 35.74% e 72.71% superiores em relação à segunda melhor estratégia nos cenários *low-rise*, *high-rise* e *skyscraper*, respectivamente. Embora estes números sejam positivos, é preciso observar que, em cenário com menos andares (*low-rise*), os benefícios de tal estratégia foram menos expressivos. Em face disto, é possível que o benefício da utilização desta solução nesta categoria de prédios não compense os custos da sua implantação - embora esta análise esteja fora do escopo deste trabalho. Após o *Planning*, a estratégia que obteve melhores resultados foi o *Better Nearest Neighbour* - uma opção de implementação trivial e baixo custo computacional. Isto denota que estratégias triviais possivelmente já sejam suficientes para prédios pequenos, enquanto estratégias mais complexas tornam-se mais atraentes à medida que os prédios tornam-se mais altos.

Não houve tempo hábil para explorar outras possibilidades algorítmicas aplicáveis a este problema devido à limitação de tempo para a realização deste estudo. De fato, a maior parcela do tempo disponível foi consumida pelo projeto, implementação e validação do simulador. Isto pois, para gerar dados confiáveis e dignos de comparação entre si, o simulador e o sua modelagem devem estar corretas. Do contrário, os dados gerados pela simulação estariam postos em xeque.

Sendo assim, fica em aberto a seguinte questão: podemos melhorar ainda mais? Este questionamento faz valer a pena o tempo e esforço dedicados a criação do simulador. Pois, ao mesmo tempo em que serve como ferramenta para validação das estratégias

propostas e implementadas, este também serve como uma plataforma flexível e expansível para construção, validação e *benchmarking* de novas estratégias no futuro.

Por fim, conclui-se que os objetivos listados no Capítulo 3 foram atingidos. Foi projetado e implementado um simulador de elevadores flexível, parametrizável e expansível; bem como estratégias de agendamento utilizando variadas técnicas algorimicas. Com base nisto, foi possível realizar a comparação destas estratégias nos cenários propostos e elencar quais estratégias possuem o menor tempo de espera médio.

8.1 Trabalhos Futuros

É a esperança dos autores deste trabalho que o conhecimento - e, de maneira mais prática, o código - aqui desenvolvidos sirvam de base para mais estudos nas áreas de simulação de elevadores, com a implementação de novas técnicas e a comparação das mesmas em diferentes cenários.

A literatura sugere que algumas técnicas não são vantajosas. Por exemplo, o uso de algoritmos genéticos [8] para a definição de zonas de atuação não é uma boa solução, bem como outras que forçam o usuário a descobrir qual carro atenderá seu chamado [8]. O mesmo artigo mostra que o sistema de controle de destino [8] obtém bons resultados, mas também sofre do problema de onerar o usuário.

Outros artigos vistos trouxeram algumas soluções mais promissoras: o primeiro, a utilização de lógica *fuzzy* para reconhecimento de padrões de tráfego e ajuste dinâmico do comportamento dos carros [16]; o segundo, propondo um modelo estatístico [13] que resultou em uma métrica para sucesso: o tempo de espera do usuário sendo reduzido de 5% a 55% em comparação com o algoritmo trivial [13].

Há também uma gama de políticas de ociosidade dos elevadores[1], que pode ser testada em combinação com as políticas de escolha de elevador.

O simulador desenvolvido neste trabalho pode ser extendido para realizar o teste e comparação destas políticas, além de como elas se comportam em conjunto com as estratégias de agendamento.

Simulações mais ricas podem ser obtidas variando as distribuições de probabilidade de chegada e saída nos andares. Por exemplo, variando os parâmetros com o tempo, para fazer mais clientes chegarem no lobby pela manhã ou simular um prédio onde há muito tráfego entre um par de andares durante algumas horas do dia.

[1] *i.e.*, o que fazer quando o elevador está parado - deixá-lo no mesmo lugar, ou levá-lo para algum outro? Como tomar esta decisão?

As estatísticas extraídas das simulações executadas neste trabalho foram, de certo modo, bastante simples. Trabalhos futuros podem extrair conclusões mais sofisticadas a partir dos dados brutos gerados pelo simulador.

Trabalhos mais simples, mas não por isso menos interessantes, podem avaliar o impacto da escolha do número de elevadores para um determinado prédio, ou ainda o impacto de políticas de exclusão de atendimento por alguns elevadores a alguns andares - *i.e.* um elevador que só atende certos andares.

Há, sem dúvida, muitas opções para trabalhos futuros que utilizem o conhecimento construído neste trabalho, e que poderão gerar valor real, tanto acadêmico quanto para a indústria.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] “Emporis standards | emporis”. Capturado em: http://www.emporis.com/building/standard/, Set 2015.

[2] Banks, J. “Handbook of Simulation”. John Wiley & Sons, Inc, 1998, 1 ed..

[3] Bureau of Labor Statistics. “Table b-3. average hourly and weekly earnings of all employees on private nonfarm payrolls by industry sector, seasonally adjusted”. Capturado em: http://www.bls.gov/news.release/empsit.t19.htm, Ago 2015.

[4] Friese, P.; Rambau, J. “Online-optimization of multi-elevator transport systems with reoptimization algorithms based on set-partitioning models”, *Discrete Applied Mathematics*, vol. 154–13, 2006, pp. 1908 – 1931, traces of the Latin American Conference on Combinatorics, Graphs and ApplicationsA selection of papers from {LACGA} 2004, Santiago, Chile.

[5] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Design Patterns: Elements of Reusable Object-oriented Software”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] IBM. “Smarter buildings survey”. Capturado em: http://www-03.ibm.com/press/attachments/IBM_Smarter_Buildings_Survey_White_Paper.pdf, Ago 2015.

[7] Jerry Banks, John S. Carson II, B. L. N.; Nicol, D. M. “Discrete-Event System Simulation”. Prentice Hall, 2005, 4 ed..

[8] Koehler, J.; Ottiger, D. “An AI-based approach to destination control in elevators”, *AI Magazine*, vol. 23–3, 2002, pp. 59–78.

[9] Koehler, J.; Schuster, K. “Elevator control as a planning problem”, 2000, pp. 331–338.

[10] Kuzunuki, S.; Hirasawa, K. “Elevator group control system”. US Patent 4,448,286, Capturado em: https://www.google.com/patents/US4448286, Maio 15 1984.

[11] Kylstra, C. “10 things your commute does to your body”. Capturado em: http://time.com/9912/10-things-your-commute-does-to-your-body/, Set 2015.

[12] Law, A. M.; Kelton, W. D. “Simulation Modeling and Analysis”. McGraw-Hill Higher Education, 2000, 3 ed..

[13] Nikovski, D.; Brand, M. “Marginalizing out future passengers in group elevator control”, *CoRR*, vol. abs/1212.2499, 2012.

[14] Ross, S. M. “Introduction to Probability Models, Ninth Edition”. Orlando, FL, USA: Academic Press, Inc., 2006.

[15] Seckinger, B.; Koehler, J., “Online synthesis of elevator controls as a planning problem”, 1999, em alemão.

[16] Siikonen, M.-L. “Elevator group control with artificial intelligence”, Relatório Técnico, Helsinki University of Technology, 1997.

[17] United Nations. “World’s population increasingly urban with more than half living in urban areas”. Capturado em: http://www.un.org/en/development/desa/news/population/world-urbanization-prospects-2014.html, Ago 2015.

APÊNDICE A – TIPOS DE GRÁFICOS

Diferentes tipos de gráficos podem ser gerados depois das simulações. Alguns deles são:

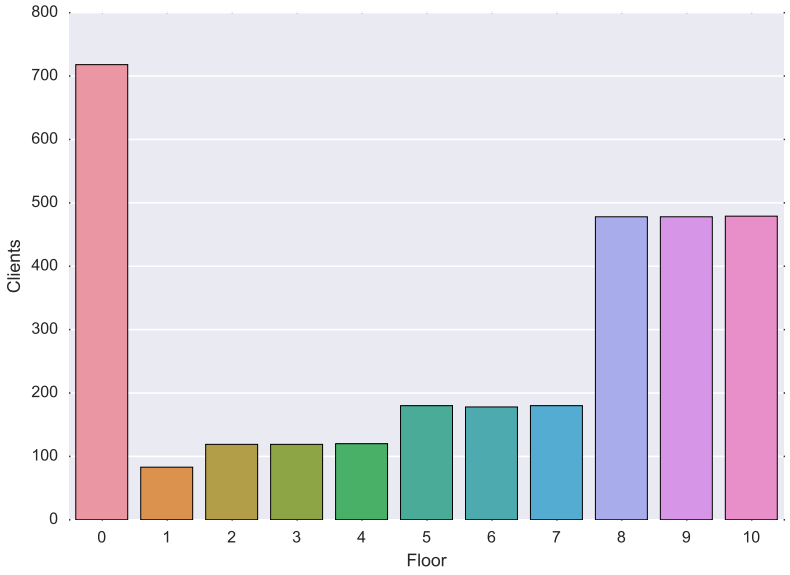

Figura A.1 – Chegadas por andar.

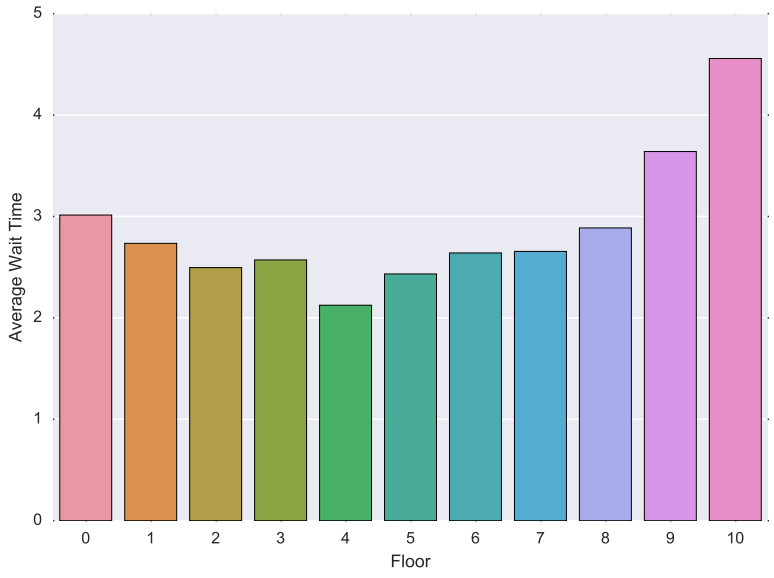

Figura A.2 – Tempo médio de espera por andar.

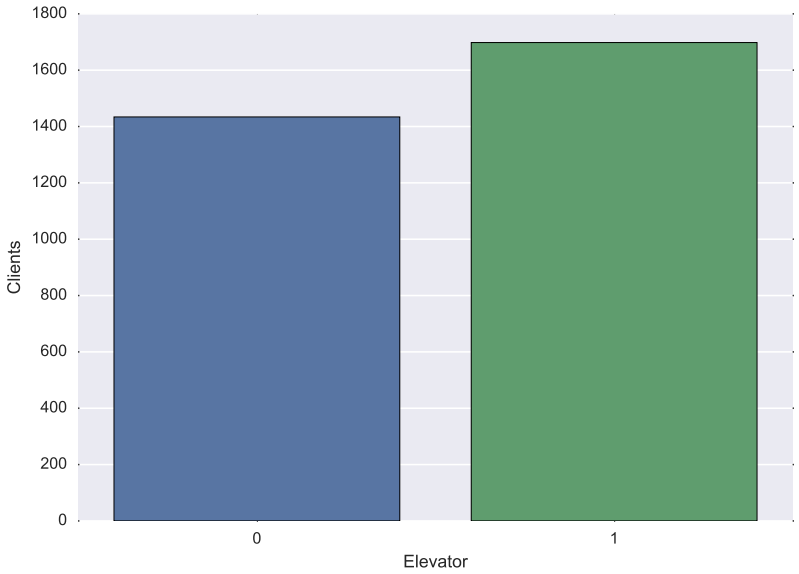

Figura A.3 – Clientes por elevador.

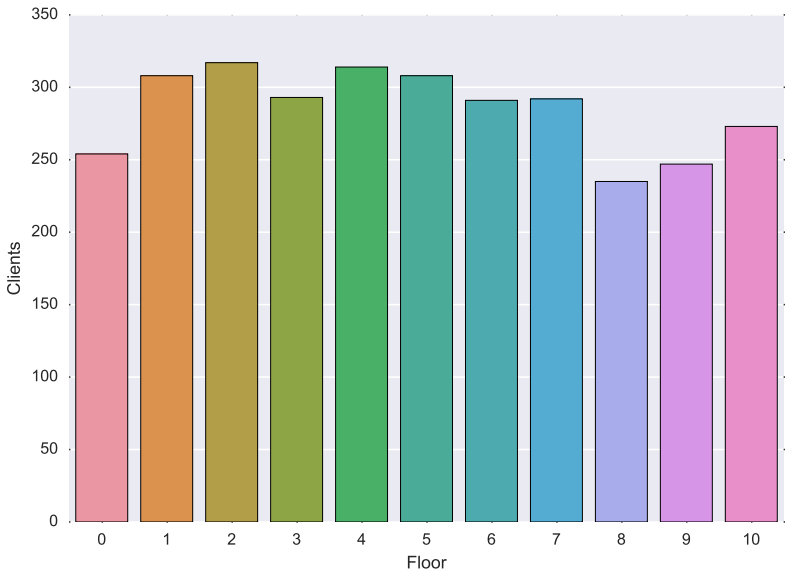

Figura A.4 – Total de clientes entregues por andar.

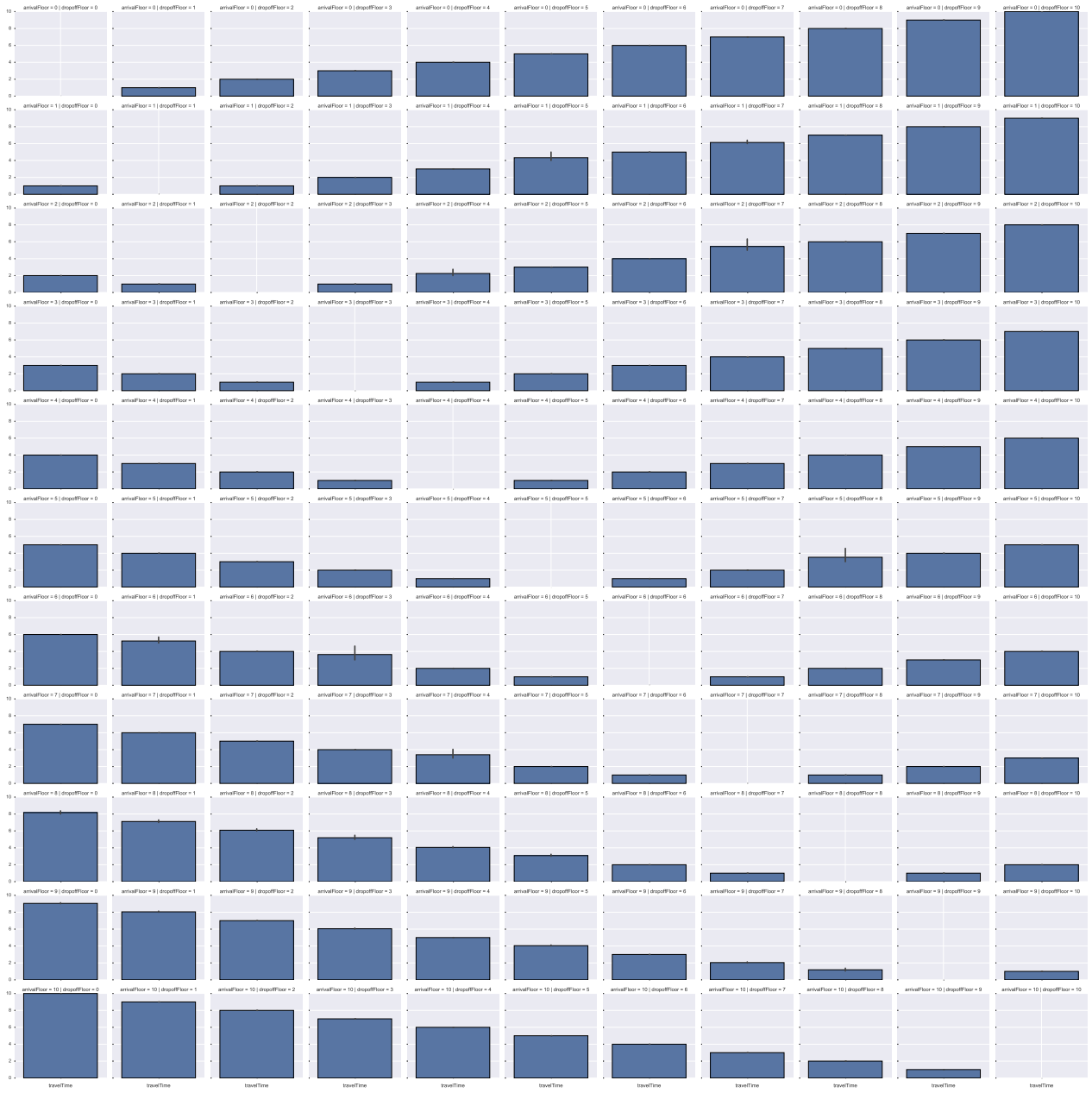

Figura A.5 – Tempo médio de jornada de andar para andar.

APÊNDICE B – GRÁFICOS DE RESULTADOS

Além das tabelas e gráficos exibidos no Capítulo 7, foram gerados gráficos para *tempo de espera médio por andar* para cada cenário e estratégia simulados.

B.1 Cenário *Low-rise*.

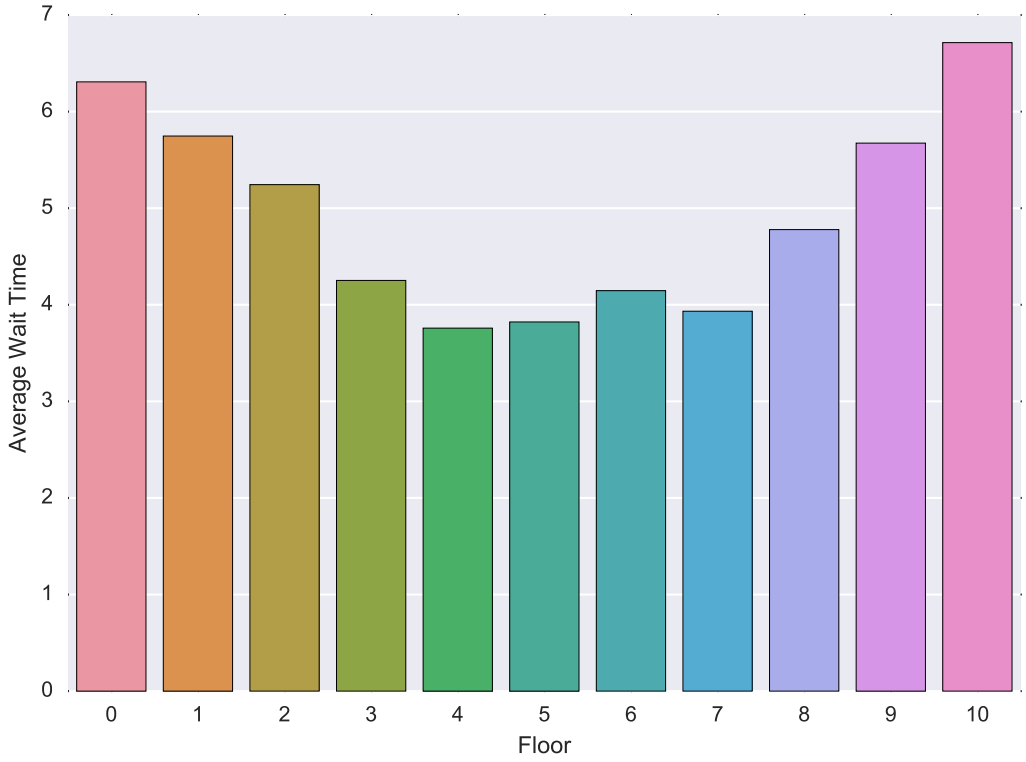

Figura B.1 – *Espera média por andar* para *random* e *Low-rise*.

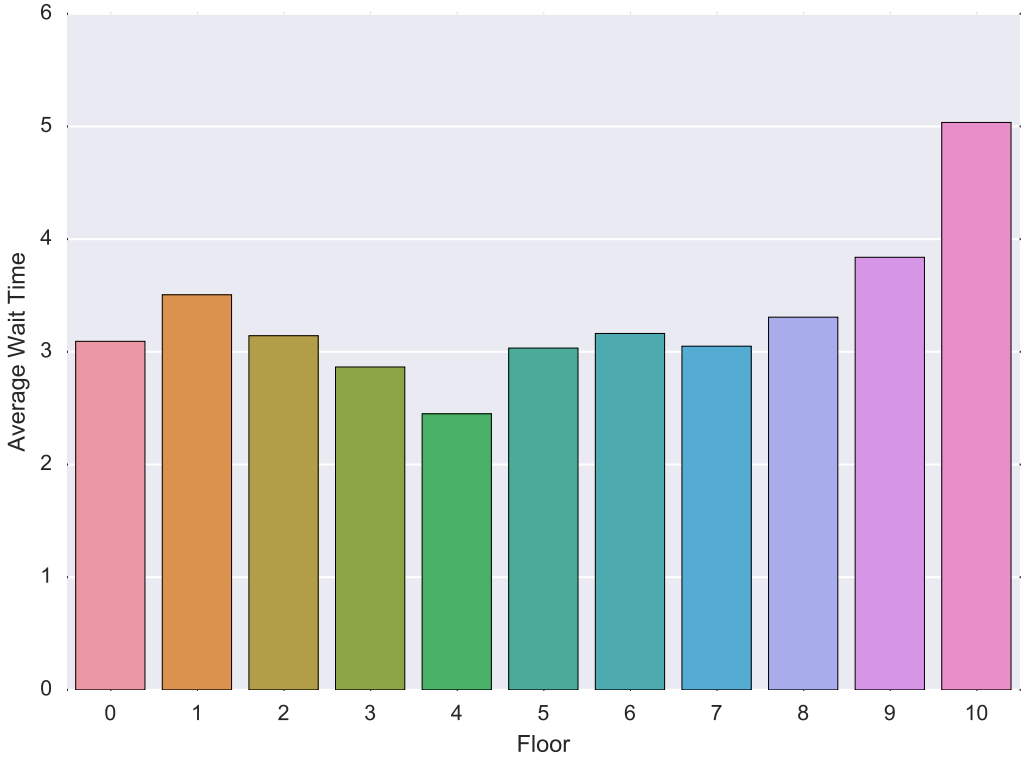

Figura B.2 – *Espera média por andar* para *nearest neighbour* e *Low-rise*.

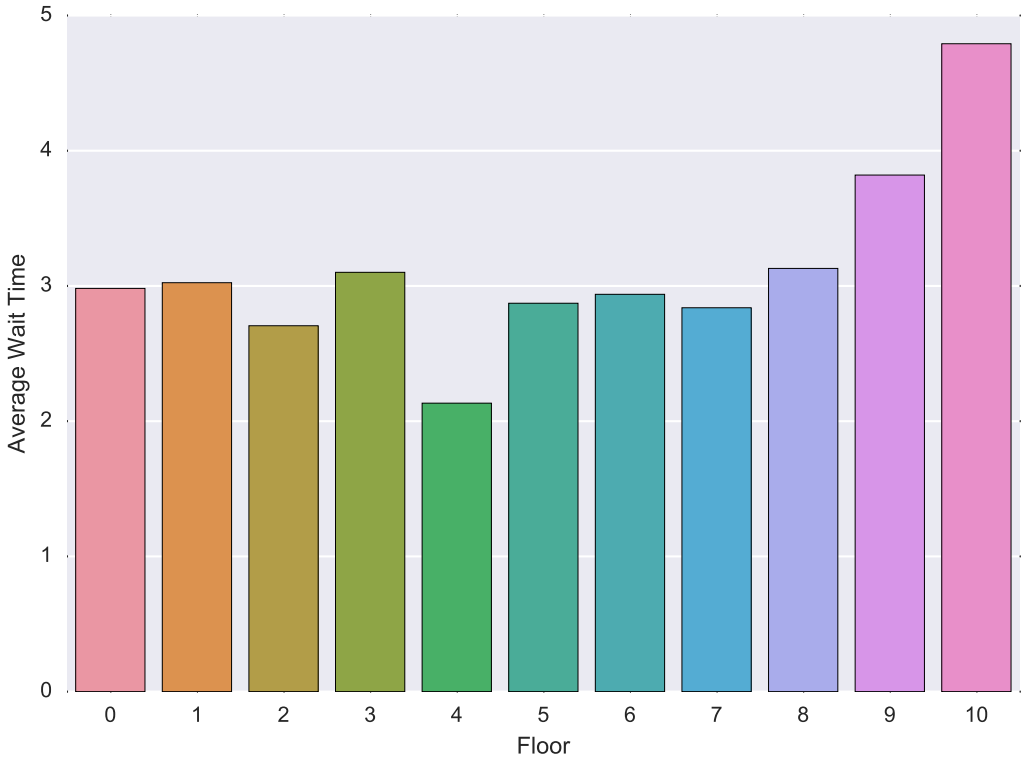

Figura B.3 – *Espera média por andar* para *better nearest neighbour* e *Low-rise*.

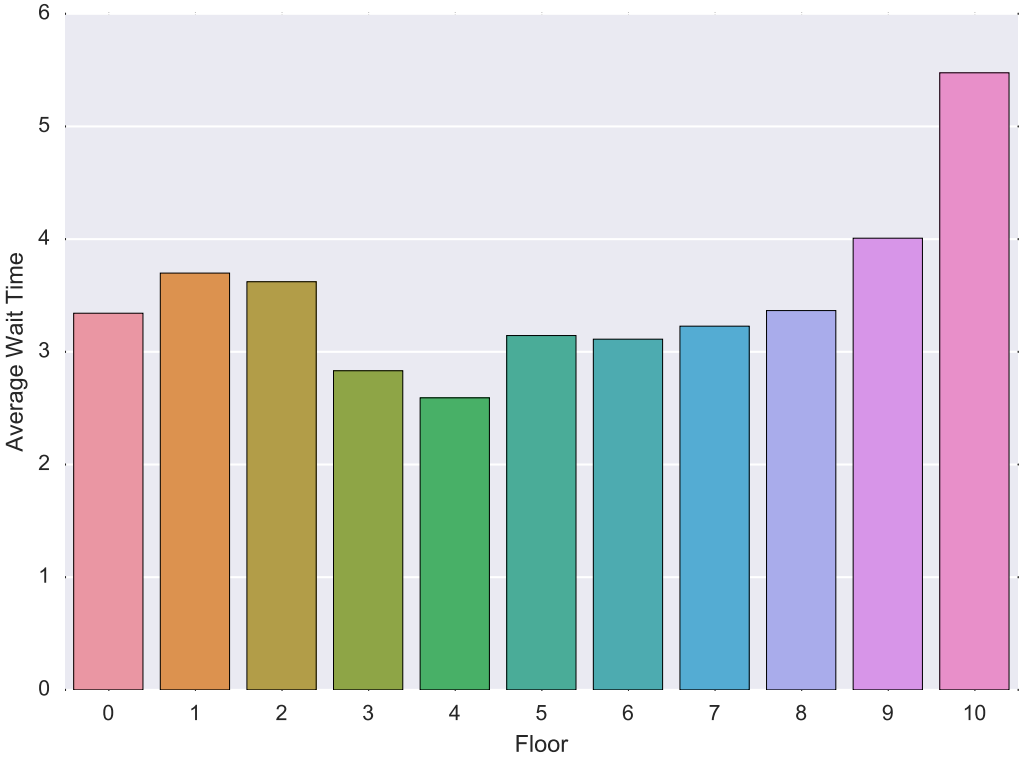

Figura B.4 – *Espera média por andar* para *weighted* e *Low-rise*.

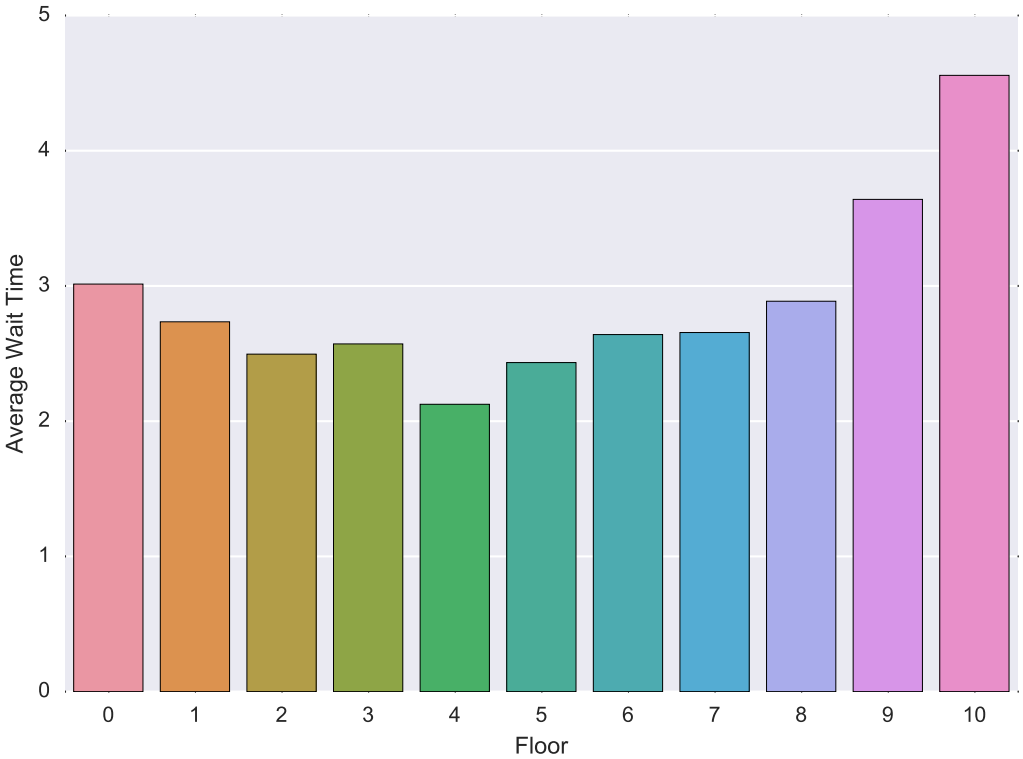

Figura B.5 – *Espera média por andar* para *planning* e *Low-rise*.

B.2 Cenário *High-rise*.

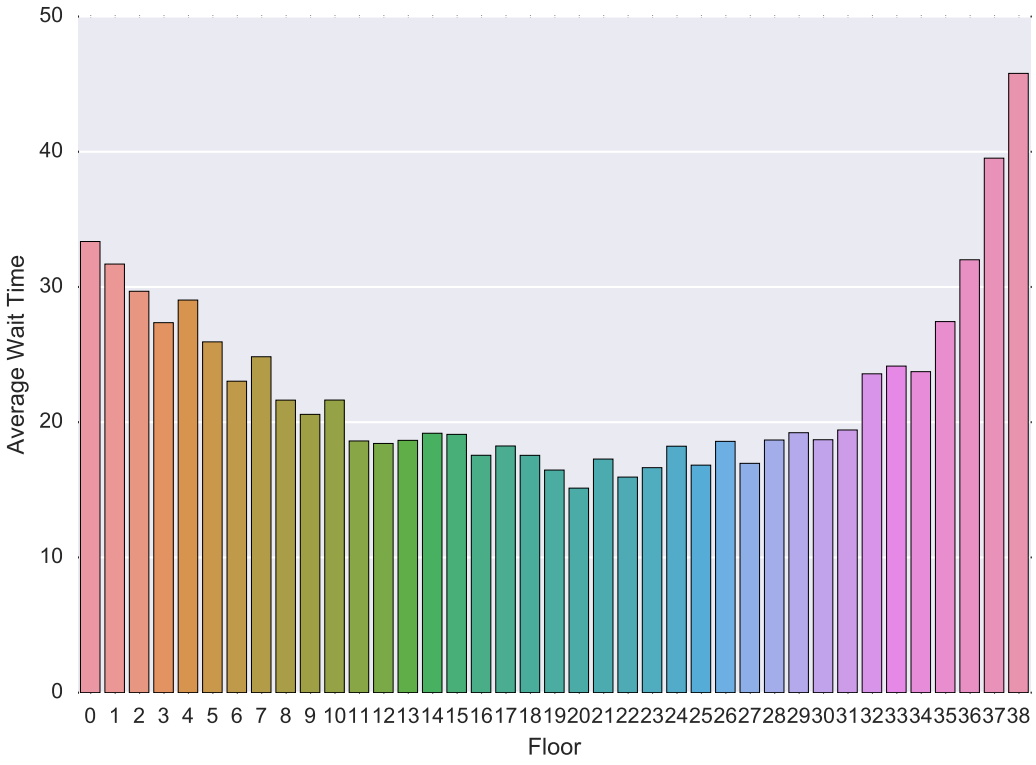

Figura B.6 – *Espera média por andar* para *random* e *High-rise*.

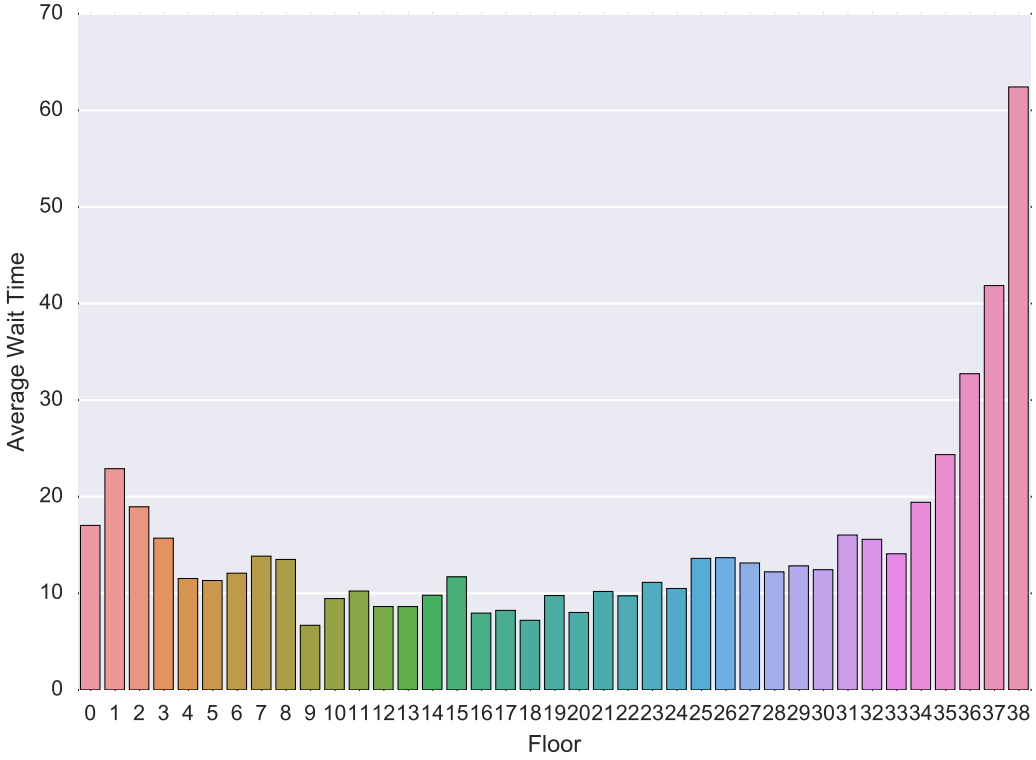

Figura B.7 – *Espera média por andar* para *nearest neighbour* e *High-rise*.

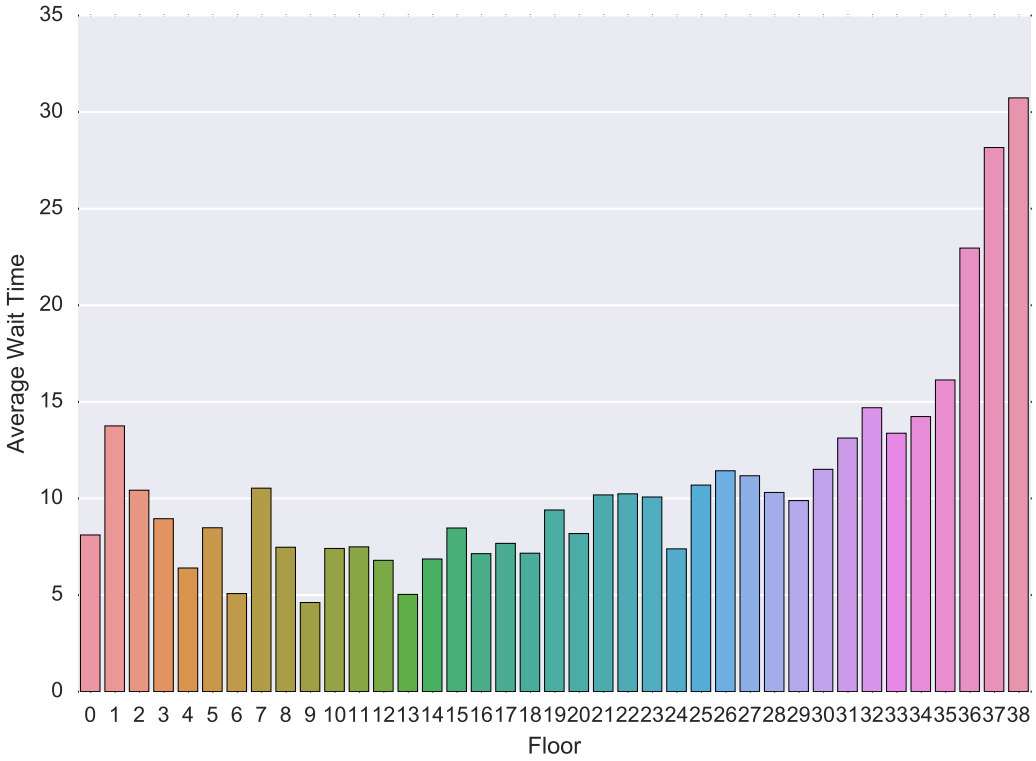

Figura B.8 – *Espera média por andar* para *better nearest neighbour* e *High-rise*.

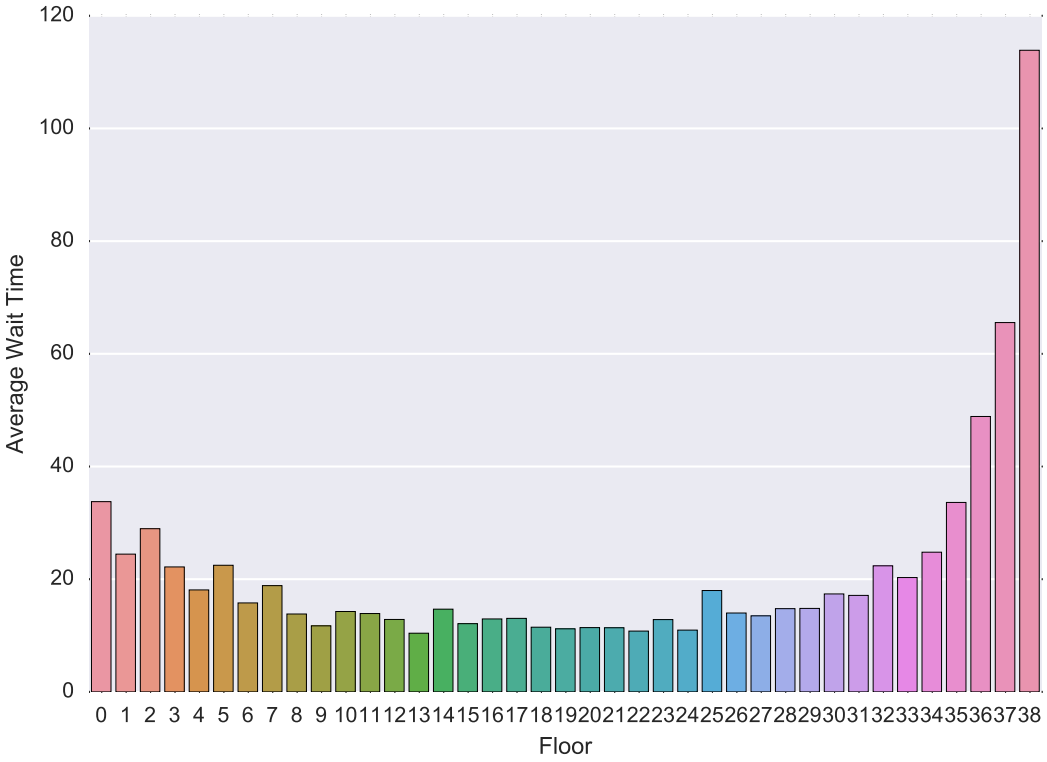

Figura B.9 – *Espera média por andar* para *weighted* e *High-rise*.

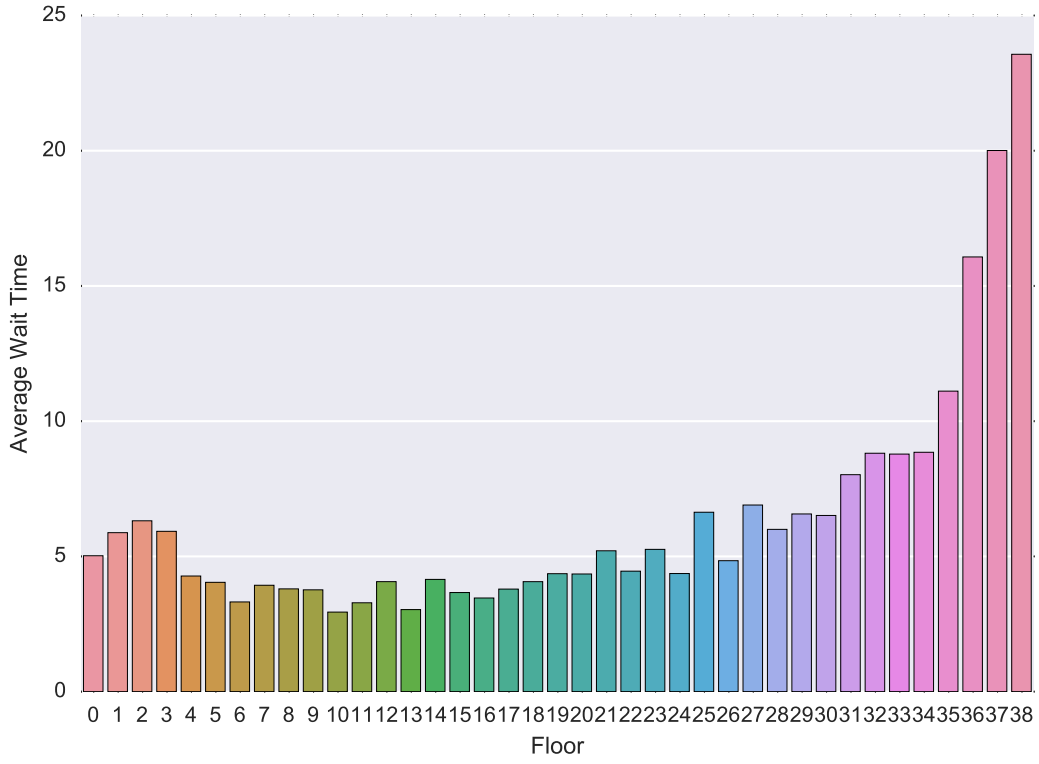

Figura B.10 – *Espera média por andar* para *planning* e *High-rise*.

B.3 Cenário *Skyscraper*.

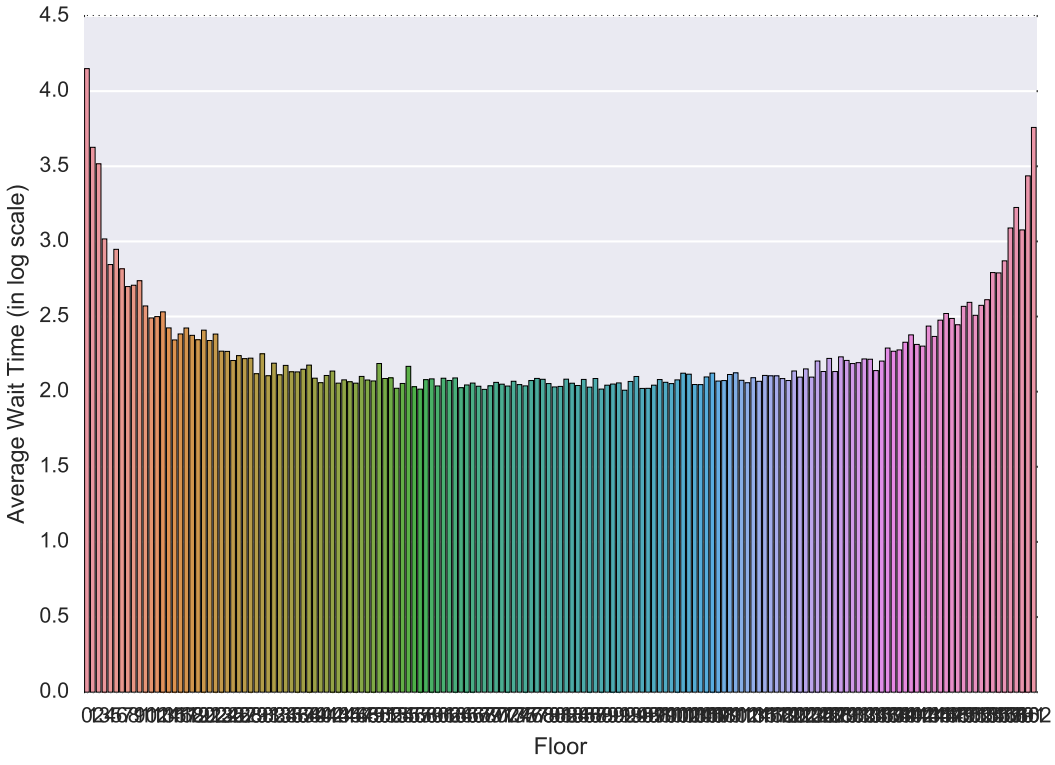

Figura B.11 – *Espera média por andar* para *random* e *Skyscraper*.

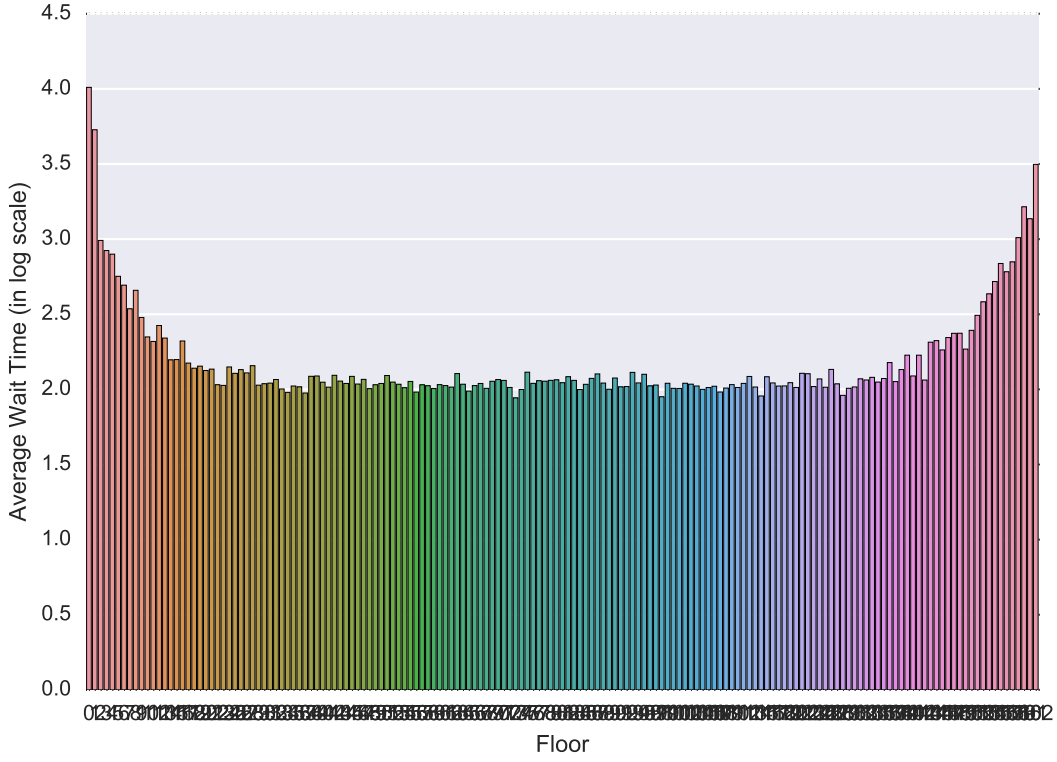

Figura B.12 – *Espera média por andar* para *nearest neighbour* e *Skyscraper*.

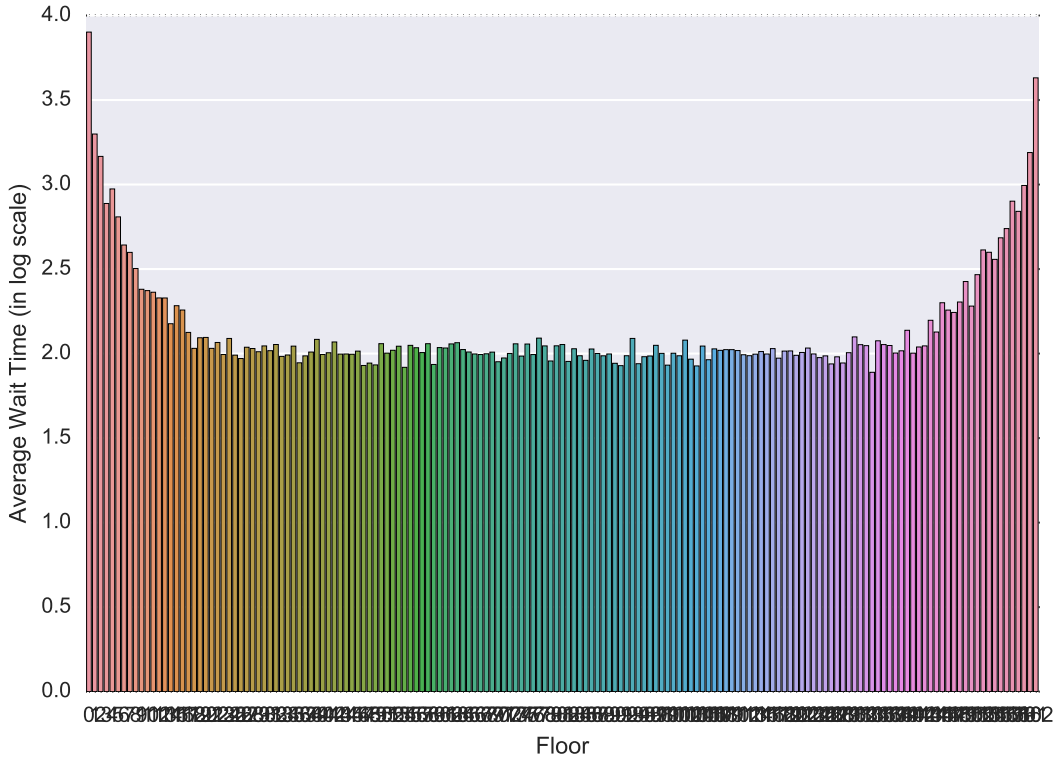

Figura B.13 – *Espera média por andar* para *better nearest neighbour* e *Skyscraper*.

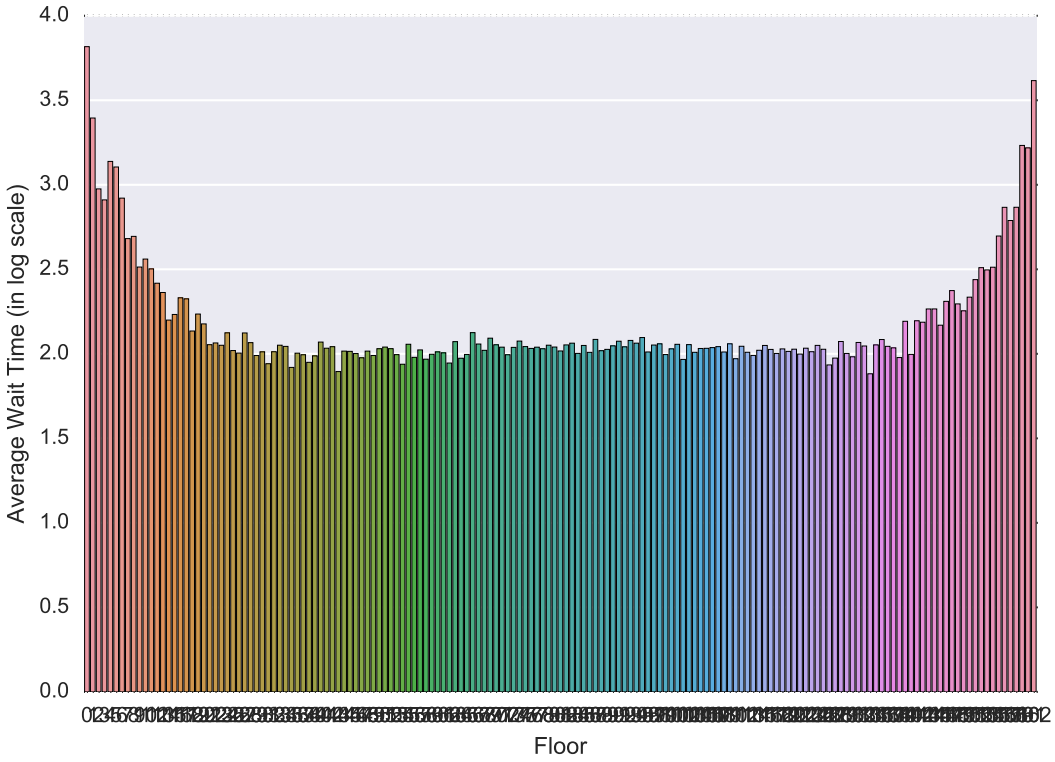

Figura B.14 – *Espera média por andar* para *weighted* e *Skyscraper*.

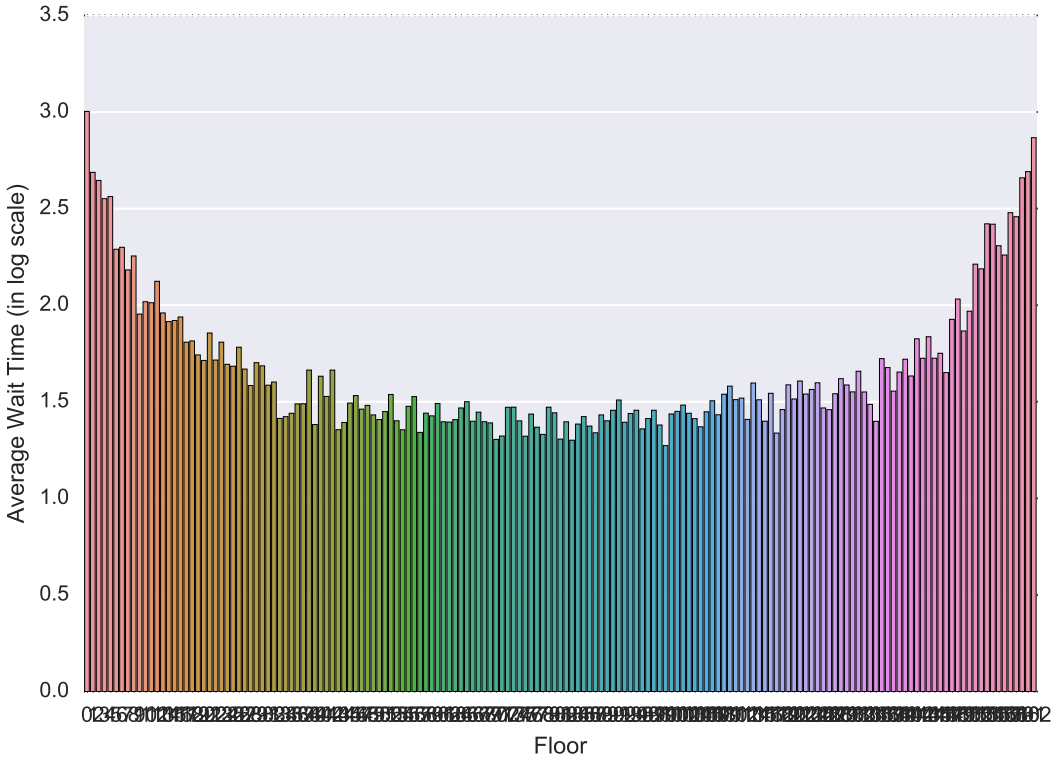

Figura B.15 – *Espera média por andar* para *planning* e *Skyscraper*.