UNIVERSIDADE FEDERAL DE SERGIPE PRÓ-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA COORDENAÇÃO DE PESQUISA

PROGRAMA INSTITUCIONAL DE BOLSAS DE INICIAÇÃO CIENTÍFICA – PIBIC

Modelo Inteligente para Navegação Offline em Ambientes *Indoor* baseado no Enviesamento de Informações em Algoritmos de Buscas

Aprendizado por Reforço aplicado à Navegação Bias *Offline* em Ambientes *Indoor*

Área do conhecimento: Ciências Exatas e da Terra
Subárea do conhecimento: Ciências da Computação
Especialidade do conhecimento: Visão Computacional

Relatório Final
Período: de Setembro de 2022 a Agosto de 2023

Este projeto foi desenvolvido com bolsa de iniciação científica
PIBIC/COPES

Orientador: Prof. Dr. Alcides Xavier Benicasa
Autor: Éricles dos Santos Cunha
Co-Autores: Kevenny de Jesus Santos e Moisés Junio Fagundes dos Santos

Sumário

Resumo

A navegação em ambientes internos apresenta desafios únicos em comparação com a navegação em ambientes externos, onde o GPS é amplamente utilizado. Este projeto teve como objetivo criar um modelo inteligente para aprimorar a navegação *offline* em ambientes *indoor*, onde o sinal de GPS é limitado. Com base em um ambiente previamente mapeado com coordenadas em um plano cartesiano, foram empregadas técnicas de Inteligência Artificial (IA), incluindo aprendizado por reforço, buscas cegas e heurísticas, para controlar a trajetória no ambiente. A pesquisa analisou e comparou os resultados obtidos por meio desses métodos, além de considerar estratégias de busca para gerar trajetórias entre pontos de origem e destino, levando em consideração também fatores de interesse e gestão do espaço. Os resultados foram avaliados e comparados entre os modelos por meio de experimentos de percurso, nos quais comparamos os mesmos pontos de origem e destino para os algoritmos, demonstrando as trajetórias no mapa do ambiente. Também realizamos um experimento em lote, gerando origens e destinos aleatórios e usando cada combinação para determinar qual algoritmo percorreu a menor distância. Em conclusão, este projeto demonstrou a eficácia do modelo proposto para melhorar a navegação em ambientes *indoor* com limitações de sinal de GPS. Os resultados dos experimentos indicaram que a abordagem de Inteligência Artificial mostrou-se promissora na resolução dos desafios associados à navegação interna.

Apoio Financeiro:
Este projeto é desenvolvido com bolsa de iniciação científica PIBIC/COPES.

Palavras-Chaves: Inteligência artificial; navegação *indoor*; aprendizado por reforço; enviesamento de informações; algoritmos de busca.

1 Introdução

A navegação em locais abertos *Outdoor*, baseada em tecnologias consolidadas, como o sistema de posicionamento global, mais conhecida como *GPS* (do inglês *Global Positioning System*), é uma realidade comum no dia-a-dia das pessoas ao redor do mundo. Por outro lado, existe a dificuldade de sua aplicação em ambientes internos (*Indoor*), uma vez que os sinais de rádio de satélites sofrem o bloqueio de estruturas, como paredes de tijolos, metal, pedra ou madeira, por exemplo.

Encontrar determinados destinos em locais fechados, como *Shoppings* e aeroportos, por exemplo, pode ser uma tarefa complexa, caso não haja familiaridade com o ambiente. Além disto, embora possam existir diversos serviços de assistência e/ou autoatendimento, existe a oportunidade de propostas de pesquisas e inovações tecnológicas nesta área.

De uma maneira geral, os sistemas de posicionamento possuem como função determinar a localização de um objeto ou indivíduo em um ambiente e, a partir disto, utilizá-la para a definição de trajetórias entre pontos de origem e destino.

Por outro lado, considerando a ausência de sinas de *GPS* em ambientes *Indoor*, e de acordo com Chelly and Samama (2009), novas técnicas de posicionamento *Indoor*, ou sistemas de posicionamento interno (IPS - *Indoor Positioning System*), têm sidos propostos baseados na identificação da localização como, por exemplo, a partir da análise da potência do sinal de redes sem fio (*WiFi*) ao qual o indivíduo está conectado, tendo assim uma estimativa de sua localização a partir da localização do roteador (Mazuelas and et al., 2009), previamente cadastrado.

Diversos outros dispositivos tecnológicos também podem ser encontrados na literatura com a finalidade de auxiliar na identificação de posicionamentos *Indoor* como, por exemplo, sensores de comunicação infravermelho, sistemas ultrassônicos, identificação de frequências de rádio (*RFID* – do inglês *Radio Frequency Identification*), dentre outros (Liu et al., 2010).

Embora o acompanhamento em tempo real seja uma característica importante para a navegação em ambientes abertos e fechados, o que torna possível a realização de novos cálculos de rotas, e que também justifica a necessidade de dispositivos *Online*, é importante notar que, para qualquer uma das técnicas de posicionamento apresentadas, seja por meio de tecnologias *GPS* ou *IPS*, faz-se necessária a existência de conectividade, ou seja, dispositivos específicos e online atendendo requisições referente ao sistema de posicionamento.

Outro assunto importante ao tema deste projeto de pesquisa está relacionado a estratégia de busca para a geração da trajetória entre os pontos de origem e destino. De um modo algorítmico e objetivo, várias alternativas são exploradas, até se encontrar uma resolução para o problema em questão, verificando ao final se a sequência encontrada é uma solução ao eficiente (Stuart Russell, 2003).

Mas o que é uma solução eficiente? Uma resposta intuitiva seria, possivelmente, o caminho mais curto. Entretanto, em sistemas de navegação *Outdoor*, por exemplo, soluções já contam com opções para evitar trechos congestionados, ou com altos índices de criminalidade, dentre diversas outras possibilidades de configurações. De maneira similar, em um contexto *Indoor*, além da análise do caminho mais curto, a possibilidade do direcionamento da rota por locais de maior interesse pode também ser um interessante ponto de investigação, tanto para o usuário, tendo uma trajetória mais agradável e de acordo com seus interesses, quanto para a administração do ambiente, possibilitando uma melhor gestão dos espaços.

O presente trabalho encontra-se dividido como a seguinte estrutura: seção 2 apresentará os objetivos do plano em questão, na seção 3 será abordada a metodologia utilizada

para o desenvolvimento deste projeto, sendo composta pelo referencial teórico e a arquitetura do modelo proposto. A seção 4 abordará os resultados e discussões alcançadas por intermédio da validação dos testes realizados. As seções 5 e 6 trarão, respectivamente, as conclusões deste plano de trabalho e as perspectivas para futuros trabalhos em torno do estudo. A seção 7 trará as referências bibliográficas utilizadas. Por fim, a seção 8 abordará outras atividades realizadas ao longo da pesquisa, constituída por atividades complementares e resultados alcançados pelo projeto a partir dos planos de trabalho envolvidos neste projeto PIBIC.

2 Objetivos

Dado que o projeto de pesquisa desenvolvido é divido em dois planos, destaca-se de forma separada os objetivos gerais do projeto e do plano em questão.

2.1 Objetivos Gerais

Este projeto de pesquisa tem como objetivo desenvolver um sistema inteligente que possa auxiliar a navegação em ambientes fechados sem a necessidade de dispositivos conectados para definir a localização. Para alcançar esse propósito, será utilizado um ambiente previamente mapeado, onde a localização de todos os pontos de origem e destino será intrinsecamente conhecida por meio de um sistema de coordenadas bidimensionais.

A trajetória no ambiente será controlada com base em princípios de Inteligência Artificial (IA), com foco na resolução de problemas usando métodos de busca cega, heurísticas e aprendizado por reforço. Além disso, será proposto um modelo que seja sensível ao ajuste dos pesos e dos critérios locais e heurísticos.

Ao final do estudo, os resultados obtidos por esses métodos serão analisados e comparados.

2.2 Objetivos Específicos

Neste plano de trabalho, os objetivos específicos compreendem as seguintes etapas:

- Realizar um estudo aprofundado da literatura pertinente para embasar teoricamente o desenvolvimento do projeto.
- Analisar, definir e desenvolver o mapeamento do ambiente *Indoor*, juntamente com as informações locais e heurísticas relevantes.
- Abordar o estudo, análise e desenvolvimento de mecanismos de aprendizado de trajetórias com base em algoritmos de aprendizado por reforço.
- Realizar treinamentos das trajetórias utilizando os mecanismos desenvolvidos.
- Explorar e analisar os resultados obtidos durante os treinamentos, considerando os métodos de busca utilizados.

Cada um desses objetivos será perseguido ao longo do plano de trabalho para alcançar o êxito do projeto.

De maneira elucidativa, o primeiro passo é realizar uma pesquisa aprofundada em livros, artigos científicos e outras fontes de informações relevantes para adquirir o conhecimento teórico necessário para fundamentar o desenvolvimento do projeto. Isso ajudará a compreender as bases teóricas e os conceitos-chave relacionados ao tema do projeto.

Em seguida, pretendemos avaliar minuciosamente o ambiente *Indoor* que será utilizado no projeto. Para isto, será necessário definir como será realizado o mapeamento desse espaço, ou seja, como será representada a sua estrutura no contexto do projeto. Isso inclui identificar todos os pontos importantes, como paredes, obstáculos, portas e outras características relevantes do ambiente.

A fase seguinte se refere à criação de um conjunto de informações locais e heurísticas, ou seja, dados e estratégias que permitirão ao sistema inteligente tomar decisões, que servirão como orientações para o sistema durante a navegação pelo ambiente.

Uma fase bastante importante para o desenvolvimento deste projeto é o processo de aprendizado de máquina. Pretendemos realizar o estudo e criação de mecanismos que possam permitir que o sistema aprenda a traçar trajetórias de navegação de forma autônoma. Isso será realizado utilizando algoritmos de aprendizado por reforço, uma abordagem da Inteligência Artificial, que baseia a tomada de decisões a partir de recompensas, positivas ou negativas, recebidas ao longo do processo de aprendizado.

Por fim, após o desenvolvimento dos mecanismos de aprendizado, o sistema será submetido a treinamentos para aprender a navegar no ambiente *Indoor*. Durante esses treinamentos, serão coletados dados e resultados que, posteriormente, serão analisados para avaliar o desempenho do sistema. Esses resultados serão então comparados com os obtidos em diferentes abordagens de treinamento e geração de rotas, para demonstrar a eficácia do modelo proposto.

3 Metodologia

Nos tópicos a seguir, serão apresentados os conceitos fundamentais para a compreensão desta proposta. Também serão abordadas as tecnologias e ferramentas adotadas ao longo do projeto, assim como sua utilização.

3.1 Revisão Bibliográfica

Na seção, serão apresentadas a revisão bibliográfica e a arquitetura do modelo proposto para este plano de trabalho.

3.1.1 Métodos de Navegação *Indoor* e *Outdoor*

Com o intuito de obter um posicionamento preciso e uma rota exata, os métodos de navegação evoluíram, adequando-se tanto para ambientes internos (navegação *Indoor*) quanto para espaços externos (navegação *outdoor*).

A navegação *Indoor*, de acordo com Taddeo et al. (2018), é o processo de determinar a localização de um dispositivo ou pessoa em um ambiente interior, como por exemplo edifícios, *Shoppings*, supermercados, fóruns, entre outros ambientes grandes e isolados.

Neste sentido, diversos trabalhos têm sido desenvolvidos considerando tais ambientes, como em Yao et al. (2010), Fagundes (2008) e Bisdikian (2001), que evidencia a possibilidade da navegação *Indoor* ser utilizada para melhorar a experiência do usuário, fornecendo direções precisas dentro de ambientes fechados. Além disso, pode ser utilizada em aplicativos de localização para fornecer informações relevantes ao usuário, como lojas, banheiros e saídas de emergência em um ambiente *Indoor*, que é o foco desta pesquisa.

Por outro lado, e de acordo com Hasegawa et al. (1999), a navegação *outdoor* é processo de determinar a localização de um dispositivo ou pessoa em um ambiente externo, como por exemplo, ruas. Geralmente é realizada utilizando tecnologias como o *GPS (Global positioning system)*, e sistemas de navegação baseados em satélite.

Conforme estudos e discussões realizadas, a tecnologia GPS é o mais utilizada para navegação *outdoor*, pois fornece informações precisas sobre a localização. Nos dias atuais, a navegação *Outdoor* é bastante utilizada em aplicativos de navegação, como por exemplo, o *Google maps* (Mehta et al., 2019) e o *Waze* (Jeske, 2013), para ajudar os usuários a chegarem em seu destino de forma segura e eficiente.

3.1.2 Grafos

Os grafos são utilizados para modelar diversas situações, nas quais, objetos podem possuir conexões entre si, como, por exemplo, no trabalho de Marciano et al. (2018), voltado para redes de computadores, ou trabalhos direcionados para redes sociais, como em Brandes et al. (2013), além de diversas outras aplicações. Em nosso trabalho, a teoria de grafo será aplicada para a modelagem de localizações e caminhos em mapas.

Segundo Goodrich et al. (2014), um grafo é simplesmente um conjunto de vértices e e um coleção de pares de arestas, sendo assim uma forma de representar conexões ou relação entre pares de objetos.

Outro conceito abordado nesse trabalho, de acordo com Goodrich et al. (2014) um grafo pode ser direcionado ou não direcionado, os grafos direcionados possuem uma direção única, indicando um movimento de um vértice para o outro. Grafos não direcionados, como mostrado na Figura 1 abaixo, existe um caminho do vértice 1 para o vértice 2 e do vértice 2 para o vértice 1.

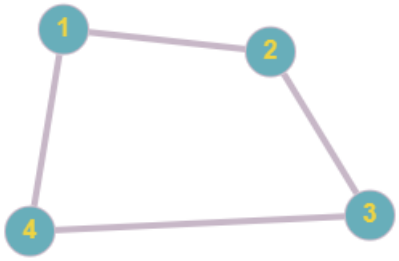


Figura 1: Grafo não direcionado, figura gerada pelo site: GraphOnline (2015).

Segundo Cormen et al. (2009), podemos representar um grafo de duas formas: como uma coleção de listas de adjacência ou como uma matriz de adjacência (uma matriz bidimensional que indica se há uma aresta entre dois vértices).

O tamanho da matriz vai depender da quantidade de vértices do grafo. A Figura 2 apresenta um exemplo de um grafo (a) e sua respectiva matriz de adjacência (b), onde as células marcadas com uma letra representam a existência de arestas entre os vértices do grafo.

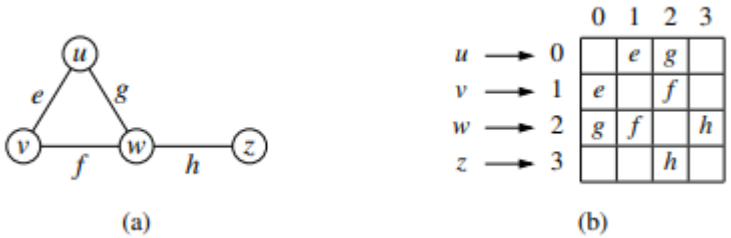


Figura 2: Matriz de adjacência (Goodrich et al., 2014).

3.1.3 Estratégias de Busca e Trajetórias

Conforme previamente mencionado, os objetivos deste projeto de pesquisa estão relacionados às estratégias de busca para a geração da trajetória entre pontos de origem e destino. Nesta seção destacaremos as principais características dos métodos utilizados nesta pesquisa.

3.1.4 Algoritmos de Busca Cega e Heurística

De acordo com Stuart Russell (2003), um agente é definido como qualquer entidade capaz de perceber seu ambiente através de sensores e de atuar sobre esse ambiente por meio de atuadores. Ainda segundo o autor, um algoritmo de busca pode ser considerado um exemplo de agente, pois explora múltiplas opções para resolver um problema específico. Simultaneamente, ele avalia se a sequência de ações encontrada representa uma solução otimizada. Neste contexto, exploramos algumas abordagens de busca, apresentadas à seguir.

Busca em Largura

Iniciamos pela Busca em Largura (do inglês, *Breadth-first search*- Bfs) que, de acordo com Permana et al. (2018), é um algoritmo de busca não-informada que expande os nós de um grafo a partir de um nó inicial, mantendo uma fila dos nós a serem visitados. A partir do

estudo de seu algoritmo, de um modo geral, este método explora todos os nós vizinhos do nó atual antes de avançar para o próximo nó na fila. A BFS assegura a descoberta da solução mais curta (em termos do número de arestas percorridas) entre o nó inicial e qualquer outro nó presente no grafo. Abaixo um algoritmo do BFS:

Algorithm 1 :Busca em largura

1. Escolher o vértice inicial e marcar como visitado;
2. Inicializar uma fila e enfileirar o vértice inicial;
3. Enquanto a fila não estiver vazia:
   (a) Desenfileirar o primeiro vértice da fila e definir como vértice Atual;
   (b) Para cada vértice vizinho adjacente do vértice Atual:
      i. Se o vértice adjacente não foi visitado:
         A. Marcar o vértice adjacente como visitado;
         B. Enfileirar o vértice adjacente;
   (c) Se o vértice Atual for igual ao vértice procurado:
      i. Retornar o vértice Atual;
4. Retornar "Vértice não encontrado".

Segundo Stuart Russell (2003) a BFS assegura achar a solução mais curta em termos de número de arestas percorridas mas não garante encontrar a solução mais eficiente em termos de custo. Pode encontrar soluções rasas antes de soluções mais profundas, mesmo que estas sejam melhores. Ou seja, a busca em largura vai achar soluções ótimas somente quando os custos de caminho são iguais.

Busca em Profundidade

O segundo método de busca estudado foi o Busca em Profundidade (do inglês, *Depth-first search* - Dfs). Conforme explicado por Holzmann et al. (1996), este é um algoritmo de busca também cego, que explora os nós de um grafo a partir de um nó inicial, utilizando uma pilha de nós a serem visitados. Ele expande o nó atual e insere seus nós filhos na pilha antes de avançar para o próximo nó na pilha. A DFS percorre o grafo em profundidade, visitando todos os nós filhos antes de visitar seus nós pais.

Algorithm 2 :Busca em Profundidade

1. Marcar vértice Inicial como visitado;
2. Marcar primeiro Vizinho como visitado;
3. Inserir primeiro Vizinho na pilha;
4. Definir vértice Atual como primeiro Vizinho;
5. Repetir até que vértice Procurado seja encontrado ou a pilha esteja vazia:
   (a) Definir vértice Encontrado como Falso;
   (b) Para cada vértice vizinho adjacente adj de vértice Atual:
      i. Se adj não foi visitado:
         A. Marcar adj como visitado;
         B. Inserir adj na pilha;
         C. Definir vértice Atual como adj;
         D. Definir vértice Encontrado como Verdadeiro;
         E. Parar o *loop*;
   (c) Se vértice Encontrado for Falso:
      i. Desempilhar o topo da pilha;
   (d) Se vértice Atual for igual a vértice Procurado:
      i. Retornar vértice Atual;
6. Retornar "Vértice não encontrado".

Segundo Stuart Russell (2003), a busca em profundidade envolve a exploração exaustiva de um ramo específico da árvore de busca até atingir o nível mais profundo antes de retroceder e explorar outras opções. Isso pode levar a problemas como ciclos infinitos ou escolha de caminhos ineficientes. Embora não garanta a descoberta do caminho mais curto entre o nó inicial e outros nós no grafo, é valiosa para situações que demandam a visitação completa de todos os nós de um grafo.

Busca em A*

Por fim, o último algoritmo de busca investigado neste projeto foi o A* (lê-se A-estrela). De acordo com Foead et al. (2021), o A* é um algoritmo de busca heurística utilizado para encontrar a rota mais curta entre dois pontos em um grafo.

De acordo com os estudos realizados, a busca $A*$ utiliza-se de uma função heurística, $h(n)$, responsável por estimar a distância a partir do nó inicial, até o objetivo. Esta função heurística é combinada com uma função de custo local, $g(n)$, que simboliza o custo do caminho do vértice inicial até o vértice atual. Esse algoritmo é totalmente inspirado na heurística, quanto melhor for a heurística, melhores serão os resultados.

A função (Equação 1), apresentada a seguir, é utilizada para determinar a prioridade dos nós que devem ser expandidos durante a busca.

$$f(n) = g(n) + h(n) \tag{1}$$

Em uma definição formal do algoritmo, sendo Q o conjunto de nós a serem pesquisados e S o estado inicial da busca, o algoritmo pode ser descrito como segue:

Algorithm 3 :Busca A*

1. Inicialize Q com o nó de busca S como única entrada;
2. Enquanto Q não está vazio:
   (a) Escolha o melhor elemento de Q;
   (b) Se o estado n é um objetivo:
      i. Retorne n;
   (c) Caso contrário:
      i. Remova n de Q;
      ii. Para cada descendente d de n que não está em visitados:
         A. Crie todas as extensões de n para d;
         B. Adicione os caminhos estendidos a Q;

Conforme mencionado por Russell Stuart Russell (2003), a busca $A*$ é notável por ser completa, ótima e altamente eficiente entre os algoritmos desse tipo. Apesar dessas qualidades, é importante reconhecer que a busca $A*$ não é uma solução universal para todas as situações de busca. A complexidade surge, em grande parte, da constatação de que a quantidade de estados diferentes a serem explorados para alcançar o objetivo tende a aumentar exponencialmente à medida que a solução se torna mais complexa.

Stuart Russell (2003) destaca que, devido à sua complexidade, nem sempre é viável buscar a solução perfeita usando o $A*$. Em vez disso, podem ser usadas variações que priorizam soluções rápidas, embora sub-ótimas. Usar uma heurística eficaz no $A*$ é crucial. Isso porque, em geral, conseguimos resultados melhores em performance e eficiência ao optar por uma abordagem informada em vez de uma busca não informada.

3.1.5 Aprendizado por Reforço

De acordo com Kaelbling et al. (1996), o Aprendizado por Reforço (do inglês, *Reinforcement learning* - RL) é uma das técnicas de aprendizado de máquina, no qual um sistema agente tenta aprimorar suas decisões por meio de uma recompensa, comparando o resultado obtido pelo resultado esperado.

Apresentaremos a seguir duas abordagens fundamentais no campo do Aprendizado por Reforço: o Q-*Learning* e o *Sarsa*. Esses algoritmos desempenham um papel crucial ao permitir que um agente aprenda a tomar decisões em ambientes desconhecidos, buscando maximizar as recompensas ao longo do tempo.

Q-Learning

De acordo com Winder (2020) e Watkins and Dayan (1992), o aprendizado por reforço tem diversos métodos, dentre eles, estudamos inicialmente o Q-*Learning*, que é um dos métodos mais simples e amplamente utilizados de aprendizado por reforço, baseado em uma tabela de valores para armazenar as estimativas de valor para cada estado-ação.

A fórmula de atualização do valor Q para um estado e ação no Q-*Learning* é dada na Equação 2, apresentada à seguir:

$$Q(s,a) = Q(s,a) + \alpha \cdot \Big(r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)\Big) \tag{2}$$

Onde:

- $Q(s,a)$ é o valor Q do estado s e a ação a,
- α é a taxa de aprendizado (*Learning Rate*),
- r é a recompensa recebida por tomar a ação a no estado s,
- γ é o fator de desconto (*Discount Factor*) que controla a importância das recompensas futuras,
- s' é o próximo estado após tomar a ação a no estado s,
- $\max_{a'} Q(s',a')$ é o valor máximo da função Q para todas as possíveis ações a' no próximo estado s'.

Sobre a parametrização deste método, consideramos alguns exemplos que seguem. Por exemplo, se utilizarmos $\alpha = 0.1$ (Equação 3), ou seja, uma taxa de aprendizado de 10%, isso significa que apenas 10% da nova informação substituirá o valor Q antigo. Isto pode resultar em aprendizado mais lento, permitindo que o agente mantenha mais da experiência anterior.

$$\alpha = 0.1 \tag{3}$$

Já o parâmetro r, caso o agente receber uma recompensa imediata, por exemplo, $r = 10$ (Equação 4), isso terá um impacto significativo no valor de Q atualizado. Assim, o agente associará mais valor a essa ação específica no estado atual, o que poderá influenciar suas escolhas futuras.

$$r = 10 \tag{4}$$

O fator de desconto γ é um valor que varia entre 0 e 1 e é utilizado para determinar a importância relativa das recompensas futuras em comparação com as recompensas imediatas. Quando $\gamma = 0.7$ (Equação 5), por exemplo, ele influencia como o agente avalia as ações a longo prazo e afeta a decisão de escolher ações que maximizem o retorno total esperado ao longo do tempo. Isso significa que o agente considera as recompensas futuras com menos importância do que consideraria se fosse 1, mas ainda assim leva em conta o impacto das ações a longo prazo em suas decisões presentes.

$$\gamma = 0.7 \tag{5}$$

Ao termo $\max_{a'} Q(s',a')$, considerando $\max_{a'} Q(s',a') = 6$ (Equação 6), por exemplo, isto indica que, entre todas as ações possíveis (a'), em relação ao próximo estado (s'), a de valor 6 será a selecionada.

$$\max_{a'} Q(s',a') = 6 \tag{6}$$

Um outro fator importante para o processo de aprendizado por reforço, e também de acordo com Dann et al. (2022), é que a seleção do valor $\max_{a'} Q(s',a')$ pode ser influenciada pela política denominada de *Greedy*, que envolve escolher a ação com o maior valor Q com base na nossa função Q atual. No entanto, para garantir um equilíbrio entre explorar novas opções e aproveitar as ações de maior valor, incorporamos também a política denominada *Epsilon-Greedy*, que incorpora elementos de exploração e exploitação.

De maneira elucidativa, por exemplo, dentro desta abordagem poderia ser definida uma probabilidade de 0.7 (ou 70%) para a exploração, o que significa que em certos momentos, mesmo que uma ação não tenha o maior valor Q, ainda há uma chance razoável de escolhê-la para descobrir novas possibilidades. Por outro lado, 0.3 (ou 30%) seria a probabilidade reservada para utilizar o conhecimento prévio, neste caso, a exploitação, onde priorizamos a escolha da ação de maior valor Q, maximizando o retorno.

Podemos concluir que a combinação de exploração e exploitação visa encontrar um equilíbrio entre a busca por novas informações e o aprimoramento das ações que conhecemos serem mais lucrativas com base em nosso conhecimento atual.

Sarsa

Agora que exploramos o conceito do Q-*Learning*, avançaremos para outra estratégia de aprendizado chamada *Sarsa*.

O *Sarsa (State Action Reward State Action)* é um método derivado do Q-*Learning* que utiliza a informação da próxima ação selecionada em vez da ação ótima para o estado. Ele é um algoritmo simples e eficaz para aprender políticas, especialmente em ambientes com poucos estados e ações (Alfakih et al., 2020).

Assim como no Q-*Learning*, no *Sarsa* também é empregada a política *Epsilon-Greedy* para guiar a seleção da próxima ação com base nos valores Q aprendidos até o momento.

O *Sarsa* aprende uma função $Q(s, a)$ assim como o Q-*Learning*, porém, ele adota uma abordagem diferente para a atualização. A fórmula de atualização do valor Q para um estado e ação no *Sarsa* é dada na Equação 7, apresentada à seguir:

$$Q(s, a) = Q(s, a) + \alpha \cdot (r + \gamma \cdot Q(s', a') - Q(s, a)) \tag{7}$$

Onde:

- $Q(s, a)$ é o valor Q para o estado s e a ação a,
- α é a taxa de aprendizado (*Learning Rate*),
- r é a recompensa recebida ao executar a ação a no estado s,
- γ é o fator de desconto (*Discount Factor*) que controla a importância das recompensas futuras,
- s' é o próximo estado após executar a ação a no estado s,
- a' é a próxima ação a ser tomada no próximo estado s'.

Utilizando um valor de $\alpha = 0.9$ (Equação 8), ou seja, uma taxa de aprendizado de 90%, é importante ressaltar que 90% da nova informação substituirá o valor Q anterior. Isso pode resultar em um processo de aprendizado excessivamente rápido, possivelmente levando o agente a concluir os ciclos de aprendizagem sem ter efetivamente internalizado as melhores trajetórias. Esse cenário pode gerar uma lacuna no aprendizado, onde o agente não consegue adquirir conhecimento suficiente para otimizar suas decisões de maneira apropriada.

$$\alpha = 0.9 \tag{8}$$

Se o agente for recompensado imediatamente com $r = 10$ (Equação 9), isso terá um impacto substancial na atualização do valor Q, e fará com que o agente atribua maior

importância a essa ação específica no estado atual, o que tem o potencial de influenciar suas decisões em momentos subsequentes.

$$r = 10 \tag{9}$$

Quando γ é ajustado para 0.3 (Equação 10), por exemplo, isto molda a maneira como o agente avalia as ações a longo prazo, afetando a escolha de ações que buscam maximizar o retorno total esperado ao longo do tempo. Nesse contexto, as recompensas futuras serão ponderadas com um fator de 0.3. Isto indica que o agente dá menos importância às recompensas futuras em comparação com um valor de 1, mas ainda assim considera a influência das ações de longo prazo em suas decisões no momento presente.

$$\gamma = 0.3 \tag{10}$$

Quando observamos que $Q(s', a') = 4$ (Equação 11), isso significa que, dentre várias alternativas de ações possíveis a', quando estamos no próximo estado s', a política *Epsilon-Greedy* guiará a escolha da ação, e essa ação específica possuirá um valor $Q = 4$.

$$Q(s', a') = 4 \tag{11}$$

Em consonância com a filosofia do aprendizado por reforço, o *Sarsa* também se beneficia da busca de um equilíbrio entre explorar novas opções e aproveitar as ações de maior valor. De acordo com Alfakih et al. (2020), a seleção da próxima ação no *Sarsa* é guiada pela política *Epsilon-Greedy*.

Portanto, assim como no Q-*Learning*, a abordagem *Epsilon-greedy* no *Sarsa* estabelece um equilíbrio entre a exploração de novas informações e o refinamento das ações que se revelaram mais lucrativas até o momento. Esse balanceamento visa aprimorar o processo de aprendizado do agente, permitindo-o fazer escolhas informadas e eficazes em ambientes de reforço.

3.1.6 Ferramentas Utilizadas

Para o desenvolvimento deste projeto de pesquisa, além do estudo bibliográfico, empregamos também uma variedade de ferramentas para garantir a conclusão dos objetivos propostos. As principais serão apresentadas nesta seção, à seguir.

Linguagem Python

De acordo com Van Rossum and Drake Jr (1995) *Python* é uma linguagem de programação de alto nível, versátil e poderosa. É conhecida por sua sintaxe clara e legibilidade, o que a torna uma escolha popular entre desenvolvedores de todos os níveis de experiência.

A escolha de utilizar a linguagem de programação *Python* para este projeto de pesquisa foi baseado nos fatos de que o orientando e orientador já possuem um amplo conhecimento nessa linguagem de programação. Esse conhecimento prévio permite aproveitar as habilidades existentes, aumentando assim a curva de aprendizado e garantindo uma maior eficiência no desenvolvimento do projeto.

Além disso, *Python* é conhecido por sua legibilidade e clareza, características que desempenham um papel crucial na tomada de decisões de desenvolvimento de software. A sintaxe simples de *Python* torna o código mais compreensível, facilitando sua manutenção ao longo do tempo. Com uma linguagem que é fácil de ler e entender, a equipe pode se concentrar mais na lógica do problema a ser resolvido, em vez de se perder em complexidades técnicas.

Outro ponto importante é a ampla gama de bibliotecas, *Frameworks* e módulos disponíveis para *Python*. Nesse projeto, foram utilizados o *FastAPI*, *Matplotlib* e *Graph-Tool*, que são algumas das bibliotecas e *Frameworks* que serão detalhadas mais adiante neste relatório.

Framework FastAPI

De acordo com Biehl (2015), uma Interface de Programação de Aplicativos ou API (do inglês, *Application programming interface*), é um conjunto de definições e protocolos que permitem que dois aplicativos se comuniquem entre si. As *Apis* são usadas para compartilhar dados, funcionalidades e recursos entre diferentes aplicações.

Existem muitos tipos diferentes de *APIs*, mas todas elas têm um objetivo comum: permitir que aplicativos diferentes se comuniquem uns com os outros. As *Apis* podem ser usadas para uma variedade de propósitos, como por exemplo: compartilhar dados entre aplicativos, chamar funcionalidades de outros aplicativos e acessar recursos de outros aplicativos.

As *Apis* são uma ferramenta poderosa que podem ser usada para melhorar a interoperabilidade entre diferentes aplicações. Na qual vamos utilizar para compartilhar dados entre o servidor que gera os caminhos por meio do aprendizado por reforço ou pelos algoritmos de busca e o aplicativo móvel com a interface de usuário simplificada.

Neste trabalho utilizamos como API a ferramenta *FastAPI*. De acordo com Ramírez (2023), o *FastAPI* é um *Framework web* moderno e de alto desempenho para criação de *API (Application programming interface)* em *Python*. Uma das principais características do *FastAPI* é sua velocidade e eficiência, se destacando também sua facilidade de uso e simplicidade, utilizando de anotações de tipo *(Type hints)* do *Python* 3.7+ para inferir os tipos de dados das requisições e respostas da *API*, gerando automaticamente uma documentação interativa e amigável com o *Swagger*[1].

Com o *FastAPI* foi possível criar *Apis RESTful* de forma rápida e eficiente, aproveitando os recursos modernos do *Python* e os benefícios da programação assíncrona.

Linguagem *Dart* e *Framwork flutter*

De acordo com Dart Team (2023) a linguagem *Dart* é uma linguagem de programação desenvolvida pelo *Google*, com o principal objetivo de construir aplicativos móveis, *Web* e *Desktop* de forma eficiente. Ela foi criada como uma alternativa moderna ao *Framework React native* [2], com o objetivo de fornecer uma sintaxe mais limpa, melhor desempenho e ferramentas integradas para desenvolvedores.

De acordo com a equipe do Google (2023), o *Flutter* é um *Framework* de código aberto desenvolvido pelo *Google* para criar aplicativos nativos de alta qualidade para várias plataformas, como *IOS*, *Android*, *Web* e *Desktop*, a partir de um único código-base. Ele utiliza a linguagem de programação *Dart* e possui uma arquitetura baseada em *Widgets*, onde tudo é um *Widget*, desde os elementos básicos da interface do usuário até os *Layouts* complexos.

[1] O *Swagger* é uma ferramenta para documentação de *Apis* que permite descrever, visualizar e testar *Apis* de forma fácil e interativa. O *Swagger* é baseado em um formato de especificação chamado *OpenAPI Specification (OAS)*, que define como a *API* deve ser estruturada e quais *Endpoints* estão disponíveis, entre outras informações.

[2] De acordo com React Native Community (2023) o *React native* é um *Framework* de desenvolvimento de aplicativos móveis que permite criar aplicativos nativos para *iOS* e *Android* usando *JavaScript*. Ele usa a biblioteca *React*, também desenvolvida pelo *Facebook*, para criar interfaces de usuário declarativas e componentes reutilizáveis.

O *Flutter* se destaca por oferecer um conjunto abrangente de *Widgets* personalizáveis e alta performance, já que ele renderiza os componentes nativos do sistema destino. Isso resulta em interfaces de usuário suaves e responsivas, semelhantes às encontradas em aplicativos nativos. Além disso, o *Flutter* possui muitas bibliotecas e ferramentas que facilitam o desenvolvimento.

Por todos esses motivos, decidimos utilizar o *Flutter* como ferramenta para gerar as interfaces gráficas desse projeto.

Módulo *Graph-Tool*

De acordo com Peixoto (2014), o *Graph-Tool* é um módulo *Python* que fornece uma ampla gama de funções para criar, manipular e analisar grafos de forma eficiente. Ele é baseado em uma biblioteca *C++* de alto desempenho e apresenta uma interface de programação fácil de usar.

O *Graph-Tool* permite criar grafos direcionados e não direcionados, atribuir propriedades a vértices e arestas, calcular métricas de centralidade e caminho mínimo, gerar subgrafos, entre outras funcionalidades. Além disso, ele também oferece ferramentas para visualizar grafos de forma interativa e exportá-los em vários formatos.

O uso do *Graph-Tool* pode melhorar significativamente a eficiência e a facilidade de implementação de grafos em um sistema. Com suas funções avançadas e sua interface de programação amigável, além disso, o *Graph-Tool* é uma escolha popular para pesquisadores e desenvolvedores que trabalham com grafos, por isso foi escolhido para implementação dos algoritmos *A**, largura e profundidade, além da aceleração do desenvolvimento, tem a parte de visualização dos grafos.

Nesse trabalho foi utilizado esse módulo do *Python*, alguns conceitos e métodos que utilizamos serão apresentados a seguir.

Inicialmente, para a utilização da ferramenta, é necessário a importação de todos os módulos e submódulos, conforme segue:

```
1 from graph_tool .all import *
```

Em seguida, um grafo pode ser criado instanciando a classe *Graph*. O grafo pode ser direcionado ou não direcionado. Por padrão, um grafo sempre é direcionado e, para gerar um grafo não direcionado, deve ser chamado o método `set_directed(False)`, se for *False* será não direcionado, se for *True* será direcionado, como código abaixo:

```
1 g = Graph()
2 g = Graph(directed = True) #direcionado
3 g.set_directed(False)       #nao direcionado
```

Uma vez que um grafo é criado, ele pode ser preenchido com vértices e arestas. Um vértice pode ser adicionado com o método `add_vertex()`, que retorna uma instância de uma classe *Vertex*, também chamada de descritor de vértice. Esses vértices podem ser adicionados arestas utilizando `add_edge()`, esse método retorna uma instância *Edge*. Veja no código a seguir:

```
1 v1 = g.add_vertex() #adicionando vertice
2 v2 = g.add_vertex() #adicionando vertice
3 e = g.add_edge(v1, v2) #adicionando aresta
```

Por fim, através da função `graph_draw()` é possível visualizar o grafo que foi construído (Figura 3), como segue:

```
graph_draw(g, vertex_text=g.vertex_index, output="two-nodes.pdf")
```

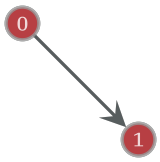


Figura 3: Dois vértices ligados por uma aresta (Peixoto, 2023b).

É possível utilizar um arquivo no formato `XML` (*Extensible markup language*), `CSV` (*Comma-separated values*) ou `TXT` (*Text*) e, a partir das informações contidas nesses arquivos, realizar a construção de um grafo. Segue o código abaixo:

```
g = gt.load_graph("search_example.xml")
name = g.vp["name"]
weight = g.ep["weight"]
pos = g.vp["pos"]
gt.graph_draw(g, pos, vertex_text=name, vertex_font_size=12, vertex_shape="double_circle",
              vertex_fill_color="#729fcf", vertex_pen_width=3,
              edge_pen_width=weight, output="search_example.svg")
```

Por fim, a partir deste código apresentado é gerado um grafo, conforme apresentado na Figura 4 abaixo:

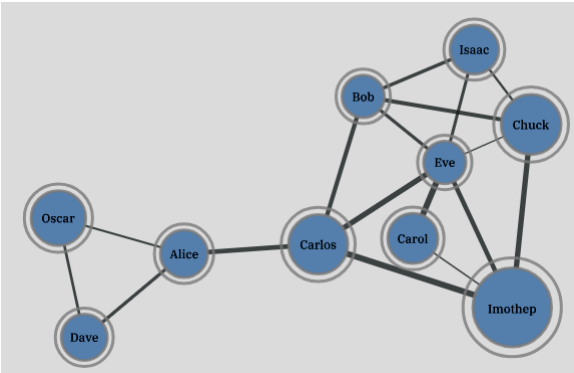


Figura 4: Grafo não direcionado Peixoto (2023a)

Busca em Largura com o *Graph-Tool*

Nesta seção serão mostrado alguns métodos que foram utilizados para o desenvolvimento da busca em largura usufruindo do módulo *Graph-Tool*.

O método utilizado para a busca em largura no *Graph-Tool* é o `bfs_search(g, source=None, visitor= graph_tool.search.BFSVisitor object>)`. São passados os parâmetros *g*, que é o grafo em questão, *source*, que é o vértice de origem, e a classe *visitor*, que é utilizada para personalizar o comportamento durante a execução da busca em largura. É importante notar que, a classe *visitor* não foi utilizada somente na busca em largura, mas também no algoritmo A* e na busca em profundidade.

```
class VisitorExample(gt.BFSVisitor):
    def __init__(self, name, pred, dist):
        self.name = name
        self.pred = pred
        self.dist = dist

    def discover_vertex(self, u):
        print("-->", self.name[u], "has been discovered!")

    def examine_vertex(self, u):
        print(self.name[u], "has been examined...")

    def tree_edge(self, e):
        self.pred[e.target()] = int(e.source())
        self.dist[e.target()] = self.dist[e.source()] + 1
```

O método `discover_vertex(self,u)` é chamado quando um vértice é encontrado durante a busca em largura. O método `examine_vertex(self,u)` é chamado quando um vértice é removido da fila, mostrando uma mensagem com o nome do nó. O método `tree_edge()` é chamado quando uma aresta é descoberta durante a busca em largura e se torna uma aresta de árvore no processo. Isto ocorre para cada aresta de árvore encontrada, atualizando as informações do predecessor e da distância para o vértice de destino.

Esses métodos podem ser usados para realizar ações personalizadas durante a busca em largura, como rastrear predecessores, calcular distâncias, coletar informações sobre os vértices e arestas visitados, entre outras tarefas específicas.

A chamada da função `gt.bfs_iterator(g, g.vertex(0))` recebe como parâmetros o grafo e o vértice de origem. Se nenhum vértice de origem for especificado, todos os vértices serão percorridos em ordem crescente. No código abaixo está a representação da expansão do grafo da Figura 4:

```
g = gt.load_graph("search_example.xml")
name = g.vp["name"]
for e in gt.bfs_iterator(g, g.vertex(0)):
    print(name[e.source()], "->", name[e.target()])
#Bob -> Eve
#Bob -> Chuck
#Bob -> Carlos
#Bob -> Isaac
#Eve -> Imothep
#Eve -> Carol
#Carlos -> Alice
#Alice -> Oscar
#Alice -> Dave
```

Busca em Profundidade com o *Graph-Tool*

No *Graph-tool* a função utilizada para realizar essa busca é `gt.dfs_search(g, g.vertex(0), VisitorExample(name, pred, dist))`, onde g é o grafo no qual se deseja realizar a busca, e o segundo parâmetro, `g.vertex(0)`, indica o vértice a partir do qual a busca será iniciada. Neste caso, a busca começa no primeiro vértice. A classe `VisitorExample` foi abordada na seção anterior.

A busca em profundidade também possui uma função *Iterator*. No entanto, essa função difere da função utilizada na busca em largura devido ao comportamento distinto do algoritmo. A função *iterator* utilizada é `gt.dfs_iterator(g,g.vertex(0)`, em que os parâmetros passados são o grafo g e o vértice a partir do qual o *Iterator* começará a expandir. Se não for especificado, todos os vértices serão percorridos, iterando sobre os vértices iniciais de acordo com seu índice em ordem crescente. O grafo utilizado para o *Iterator* pode ser visto na Figura 4. Segue o código abaixo:

```
g = gt.load_graph("search_example.xml")
name = g.vp["name"]
for e in gt.dfs_iterator(g, g.vertex(0)):
    print(name[e.source()], "->", name[e.target()])
#Bob -> Eve
#Eve -> Isaac
#Isaac -> Chuck
#Chuck -> Imothep
#Imothep -> Carol
#Imothep -> Carlos
#Carlos -> Alice
#Alice -> Oscar
#Oscar -> Dave
```

Busca A* com o *Graph-Tool*

No graph-tool, a função utilizada para a busca A* é a `graph_tool.search.astar_search (g, source, weight, visitor = <graph_tool.search.AStarVisitor object>, heuristic = <function <lambda», dist_map=None, pred_map=None, cost_map=None, combine=<function<lambda», compare=<function<lambda»,zero=0,infinity=inf, implicit=False)`. Existem vários parâmetros a serem passados, dentre eles, a classe *visitor*, explicada anteriormente, o grafo g, onde a busca será realizada, o peso das arestas *weight* e a função heurística para calcular $h(n)$, exemplificada no código a seguinte utilizando a distância euclidiana:

```
def h(v, target, pos):
    return sqrt(sum((pos[v].a - pos[target].a) ** 2))
```

É importante ressaltar que outros parâmetros também podem ser ajustados de acordo com as necessidades do problema, tais como `dist_map`, `pred_map`, `cost_map`, `combine`, `compare`, `zero`, `infinity` e `implicit`.

A função *Iterator* apresentará uma travessia pelos vértices, de acordo com o ponto de origem definido. Neste exemplo, a origem é o vértice 0. Segue abaixo a função e o resultado da travessia do grafo da Figura 4:

```
g = gt.load_graph("search_example.xml")
name = g.vp["name"]
weight = g.ep["weight"]
for e in gt.astar_iterator(g, g.vertex(0), weight):
    print(name[e.source()], "->", name[e.target()])
#Bob -> Eve
#Bob -> Chuck
#Bob -> Carlos
#Bob -> Isaac
#Eve -> Imothep
#Eve -> Carol
```

```
#Carlos -> Alice
#Alice -> Oscar
#Alice -> Dave
```

3.1.7 Trabalhos Relacionados

De acordo com os objetivos deste trabalho, nesta seção serão apresentados alguns trabalhos baseados em navegação *Indoor* e suas principais características relacionadas à localização *Indoor*.

Inicialmente, no trabalho proposto por Mazuelas and et al. (2009), foi introduzido um método de posicionamento que se baseia em medições de intensidade do sinal recebido para localizar estações móveis em ambientes *Indoor* a partir de redes *Wifi*. O método proposto estima, dinamicamente, o posicionamento *Indoor*, mesmo considerando a natureza dinâmica e complexa das condições de propagação em redes sem fio internas.

Em seguida, no trabalho de Leppakoski et al. (2013), que trata da navegação pedestre baseada em sensores inerciais, mapas internos e também sinais de redes *Wifi*, foi abordado o desafio de se obter informações precisas de posicionamento em ambientes *Indoor*, onde os sinais de satélite, como o GPS, são fracos ou indisponíveis. Para resolver esse problema, foram propostos métodos que combinam dados de sensores inerciais (como giroscópios e acelerômetros), mapas internos e sinais de redes sem fio para a navegação de pedestres em ambientes internos.

Tendo, também, como base a utilização da tecnologia *Wifi*, em Mengual et al. (2013), foi apresentada uma arquitetura multiagente flexível, em conjunto com uma metodologia de localização *Indoor*, para sistemas móveis como *Laptops*, *Smartphones*, *Tablets* ou sistemas robóticos em ambientes internos usando a tecnologia de redes sem fio, complementando o GPS.

Já no trabalho de Chelly and Samama (2009) foram apresentadas propostas técnicas para posicionamento *Indoor*, combinando métodos determinísticos e de estimativa. Os autores enfatizaram o constante desenvolvimento de técnicas de localização que buscam atingir novos objetivos, com foco especial na parte desafiadora dos sistemas de localização que lida com ambientes fechados ou complexos, uma vez que muitas técnicas de localização projetadas para ambientes externos enfrentam dificuldades quando aplicadas em ambientes internos ou em cenários complicados.

Diversos outros dispositivos tecnológicos podem ser encontrados na literatura com a finalidade de auxiliar na identificação de posicionamentos *Indoor* como, por exemplo, sensores de comunicação infravermelho, sistemas ultrassônicos, identificação de frequências de rádio (*RFID* – do inglês *Radio Frequency Identification*) (Liu et al., 2010).

Recentemente, no trabalho de El-Sheimy and Li (2021), foi realizada uma análise do estado-da-arte e as tendências futuras sobre posicionamento, localização e navegação em ambientes fechados envolvendo, onde, além dos dispositivos já citados neste trabalho, outros mecanismos foram analisados como, por exemplo, percepção ambiental a partir de mapas de alta definição (mapa HD), câmeras, tecnologias de comunicação de redes móveis (5G), fusão de informações multi-plataforma, multi-dispositivos e multi-sensores, sistemas de auto-aprendizagem e tecnologias integradas com inteligência artificial.

Por fim, pretendemos em nossa pesquisa propor um modelo inteligente para navegação *offline* em um ambiente *Indoor*, porém sem a utilização de dispositivos específicos para localização, de forma que, a partir do mapeamento prévio do ambiente, a navegação ocorrerá baseada no enviesamento de informações e algoritmos inteligentes de busca. A arquitetura do modelo e experimentos seguem apresentados nas seções seguintes.

3.2 Arquiteturas do modelo

Ao considerarmos a necessidade de um ambiente *Indoor* para aplicar os conceitos mencionados anteriormente, decidimos utilizar mapas de centros comerciais, em planta baixa[3], como referência para a construção do ambiente. Além de facilitar a criação do grafo em questão, assim como seus vértices e arestas.

A planta baixa, apresentada na Figura 5, representa um ambiente fictício inspirado em um ambiente *Indoor*, neste caso, um *Shopping* de pequeno porte.

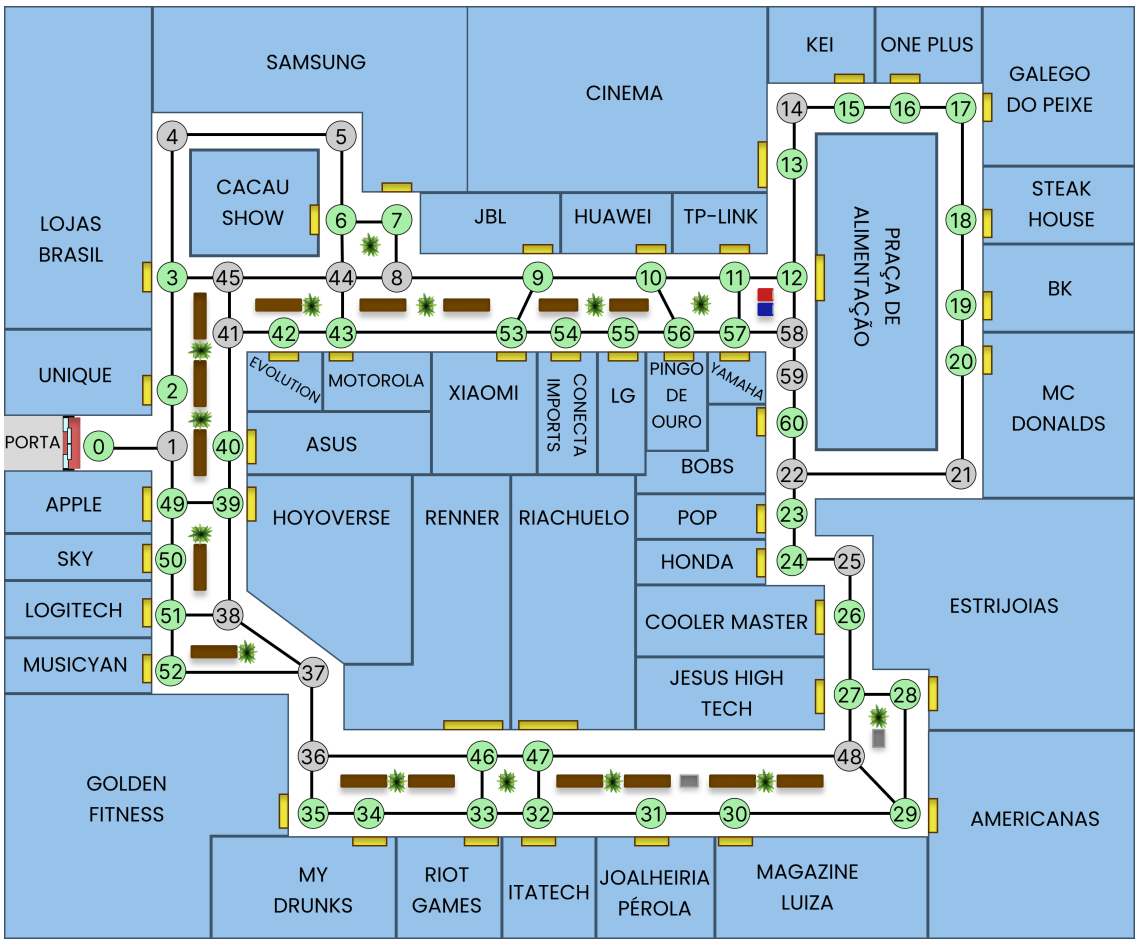


Figura 5: Planta baixa fictícia do modelo

Utilizando esta planta baixa como referência, foi criado um grafo representativo para o ambiente (Figura 5). Como pode ser observado no grafo, cada nó representa uma localização específica, como um estabelecimento ou um cruzamento. No total, existem 43 estabelecimentos. Os círculos *verdes* representam as lojas, as linhas em *preto* representam as arestas, e os círculos em *cinza* representam pontos de cruzamento ou locais que não possuem lojas. Os números nos círculos servem para identificar os vértices na tabela que foi criada para o ambiente, que será abordada com mais detalhes à seguir.

É importante notar que, a proposta de utilização deste ambiente foi de tornar possível a avaliação da eficácia de cada algoritmo na navegação estudado neste plano de trabalho. O que permitirá a comparação dos resultados obtidos por cada algoritmo nesse contexto específico.

[3] A planta baixa é um desenho em escala que apresenta a disposição de quartos, espaços e características físicas quando observado de cima. Também conhecida como planta de casa, planta arquitetônica ou desenho arquitetônico, ela proporciona uma visualização da circulação das pessoas no ambiente (Lucidchart, 2023)

3.2.1 Modelo de dados

O modelo de dados deste projeto foi idealizado com base no ambiente fictício previamente mencionado, de forma que, por meio de um grafo, foi possível representar estabelecimentos e cruzamentos como vértices, e conexões entre estes como arestas, ou seja, as ligações entre pontos adjacentes.

id	name	category	pos_x	pos_y
0	Porta	#	12	80
1	#	#	25	80
2	Unique	Fashion	25	70
3	Lojas Brasil	Market	25	50
4	#	#	25	25
5	#	#	55	25
6	Cacau Show	Food	55	40
7	Samsung	Tech	60	40
8	#	#	60	50
9	JBL	Tech	90	50
10	Huawei	Tech	110	50
11	TPLINK	Tech	125	50
12	Praça de Alimentação	Food	135	50
13	Cinema	Entertainment	135	30
14	#	#	135	20
15	Kei	Market	145	20
16	OnePlus	Tech	155	20
17	Galego Do Peixe	Food	165	20
18	Steak House	Food	165	40
19	BK	Food	165	55
20	Mc Donalds	Food	165	65
21	#	#	165	85
22	#	#	135	85
23	POP	Food	135	90
24	Honda	Automotive	135	100
25	#	#	145	100
26	Cooler Master	Tech	145	105
27	Jesus High Tech	Tech	145	120
28	Estrijoias	Fashion	155	120
29	Americanas	Market	155	145
30	Magazine Luiza	Market	125	145
31	Joalheira Pérola	Fashion	110	145
32	ItatechJr	Tech	90	145
33	Riot Games	Games	80	145
34	My Drunks	Drink	60	145
35	Golden Fitness	Fit	50	145
36	#	#	50	135
37	#	#	50	120
38	#	#	35	110
39	Hoyoverse	Games	35	90
40	Asus	Tech	35	80
41	#	#	35	60
42	Evolution	Fit	40	60
43	Motorola	Tech	55	60
44	#	#	55	50
45	#	#	35	50
46	Renner	Fashion	80	135
47	Riachuelo	Fashion	90	135
48	#	#	145	135
49	Apple	Tech	25	90
50	Sky	Tech	25	105
51	Logitech	Tech	25	110
52	Musicyan	Music	25	120
53	Xiaomi	Tech	85	60
54	Conecta Imports	Tech	95	60
55	LG	Tech	105	60
56	Pingo de Ouro	Fashion	115	60
57	Yamaha	Automotive	125	60
58	#	#	135	60
59	#	#	135	65
60	Bobs	Food	135	75

Tabela 1: Linhas e colunas do arquivo CSV de vértices.

Para representar os vértices do ambiente em questão, registramos as seguintes informações: um identificador único para cada estabelecimento, denotado por um número inteiro (*Id*); o nome do estabelecimento (no caso de ser um cruzamento, o símbolo utilizado é "#") (*Name*); a categoria correspondente ao estabelecimento (quando aplicável; caso contrário, usa-se "#") (*Category*); e as coordenadas x (Pos_x) e y (Pos_y), que indicam a localização do estabelecimento no plano cartesiano da planta baixa do centro comercial. Estes dados foram armazenados no formato `CSV`, conforme exemplificado na Tabela 1.

from	to	distance
0	1	13.0
1	2	10.0
1	49	10.0
2	3	20.0
3	4	25.0
3	45	10.0
4	5	30.0
5	6	15.0
6	7	5.0
6	44	10.0
7	8	10.0
8	9	30.0
8	44	5.0
9	10	20.0
9	53	11.2
10	11	15.0
10	56	11.2
11	12	10.0
11	57	10.0
12	13	20.0
12	58	10.0
13	14	10.0
14	15	10.0
15	16	10.0
16	17	10.0
17	18	20.0
18	19	15.0
19	20	10.0
20	21	20.0
21	22	30.0
22	23	5.0
22	60	10.0
23	24	10.0
24	25	10.0
25	26	5.0
26	27	15.0
27	28	10.0
27	48	15.0
28	29	25.0
29	30	30.0
29	48	14.1
30	31	15.0
31	32	20.0
32	33	10.0
32	47	10.0
33	34	20.0
33	46	10.0
34	35	10.0
35	36	10.0
36	37	15.0
36	46	30.0
37	38	18.0
37	52	25.0
38	39	20.0
38	51	10.0
39	40	10.0
39	49	10.0
40	41	20.0
41	42	5.0
41	45	10.0
42	43	15.0
43	44	10.0
43	53	30.0
44	45	20.0
46	47	10.0
47	48	55.0
49	50	15.0
50	51	5.0
51	52	10.0
53	54	10.0
54	55	10.0
55	56	10.0
56	57	10.0
57	58	10.0
58	59	5.0
59	60	10.0

Tabela 2: Linhas e colunas do arquivo CSV de arestas.

Para representar as arestas do ambiente, registramos as seguintes informações: origem (*From*); destino (*To*); e a distância (*Distance*) entre esses dois pontos. Tanto a origem quanto o destino correspondem aos identificadores únicos dos vértices envolvidos. O arquivo de arestas também está sendo representado no formato `CSV`, e podemos ter sua visualização parcial como representado na Tabela 2.

Podemos obter uma representação visual da relação dos vértices e das arestas por meio da Figura 6. Na tabela de arestas, temos as colunas *From* e *To*. Essas colunas referem-se aos *Ids* dos vértices na tabela de vértices. No entanto, os valores em *From* e *To* não

podem ser iguais, pois isso indicaria que um vértice está apontando para si mesmo. Isso nos proporciona uma visão mais clara e compreensível das informações em questão. A Figura 6 ajuda a ilustrar os conceitos discutidos anteriormente e a visualizar as relações entre os elementos envolvidos.

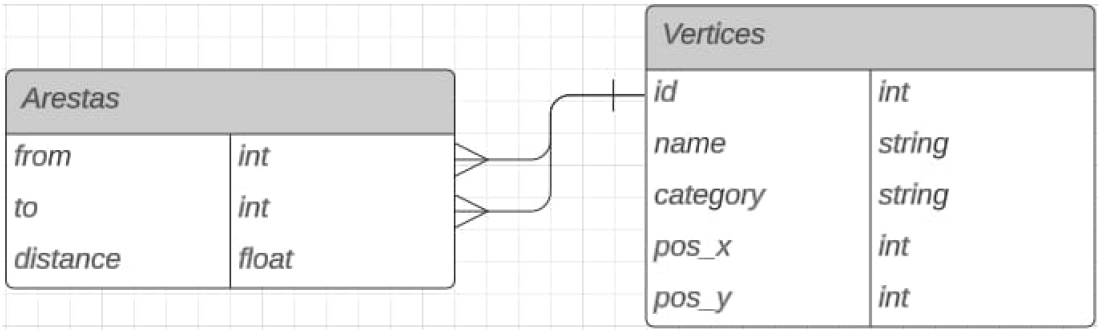


Figura 6: Atributos das tabelas de vértices e arestas.

Para calcular a distância entre dois pontos (vértices) no ambiente, foram experimentadas as abordagens de utilização das métricas de distância Euclidiana e Distância de Manhattan. Essas são duas métricas de distância comumente empregadas para medir a proximidade entre pontos em um espaço.

Como visto no trabalho de Carvalho Júnior et al. (2009), a distância euclidiana é uma medida usada para calcular a distância entre dois pontos em um espaço comum (como um espaço bidimensional ou tridimensional). Esse método foi utilizado na pesquisa para calcular a distância entre os vértices do ambiente, que foi salvo no arquivo `CSV` de arestas.

$$\text{Formula da Distância Euclidiana} = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{12}$$

De acordo com o trabalho de Strauss and von Maltitz (2017), a distância de Manhattan é uma medida usada para calcular a distância entre dois pontos em um espaço comum, como um espaço bidimensional (plano cartesiano). Ao contrário da Distância Euclidiana, que mede a distância em linha reta (como uma distância entre dois pontos em um mapa), a Distância de Manhattan considera apenas os deslocamentos ao longo dos eixos coordenados.

$$\text{Formula da Distância de Manhattan} = |p_1 - q_1| + |p_2 - q_2| \tag{13}$$

Como mostrado na Figura 7, a distância Euclidiana está sendo representada pela linha verde, ela ignora os pontos inteiros de um plano cartesiano, enquanto nas outras linhas podemos ver diversos exemplos de como a distância Manhattan pode se comportar.

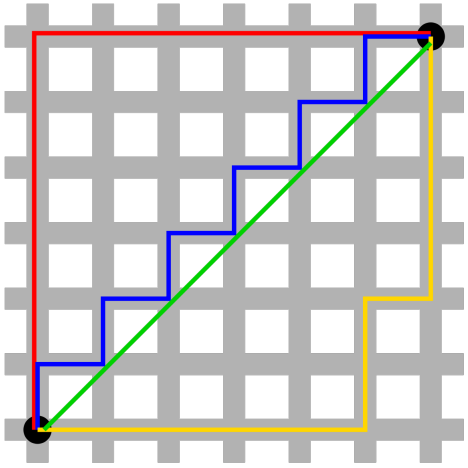


Figura 7: Representação da distância Euclidiana (verde) e distância Manhattan (vermelho, azul e amarelo). Fonte: Commons (2023)

.

No contexto da análise realizada, optamos por utilizar a métrica de distância Euclidiana como medida de avaliação, em virtude das características específicas do ambiente em questão, uma vez que em comparativos entre a distância Euclidiana e a distância de Manhattan, constatamos que não foram obtidas diferenças significativas.

3.2.2 Geração das Tabelas de Roteamento

As tabelas de roteamento, denominadas aqui por tabelas Q_{rot}, têm a função de armazenar informações sobre o próximo vértice a ser percorrido, dependendo da localização atual, até se atingir um destino específico.

A estrutura das tabelas Q_{rot} nos algoritmos de busca difere das utilizadas nos algoritmos de aprendizado por reforço. Nas tabelas dos algoritmos de busca, é registrada a trajetória completa até o destino, enquanto nas tabelas dos algoritmos de aprendizado por reforço, é gerado somente o próximo passo necessário para alcançar o destino. As Tabelas 3 e 4 apresentam as rotas, com destino igual ao vértice 0, para as buscas baseadas nos métodos de $A*$ e *Q-Learning*, respectivamente.

A arquitetura da geração de tabelas Q_{rot} pode ser vista na Figura 8. As tabelas Q_{rot} dos algoritmos de busca foram geradas por meio de um sistema contido em um contêiner *Docker*[4], que executa a imagem do *Graph-Tool* (Peixoto, 2014). Já as tabelas Q_{rot} referentes aos algoritmos de aprendizado por reforço foram criadas por um *Script* em *Python*, o qual implementa os algoritmos *Q-Learning* e *Sarsa*, sendo responsável por salvar as tabelas de aprendizado, onde apenas a melhor ação de um determinado vértice será mantida.

[4] *Docker* é um software que utiliza virtualização a nível de sistema operacional para entregar *Softwares* em pacotes chamados contêineres. Os contêineres são isolados uns dos outros e agrupam seus próprios sistemas, bibliotecas e arquivos de configuração (Anderson, 2015).

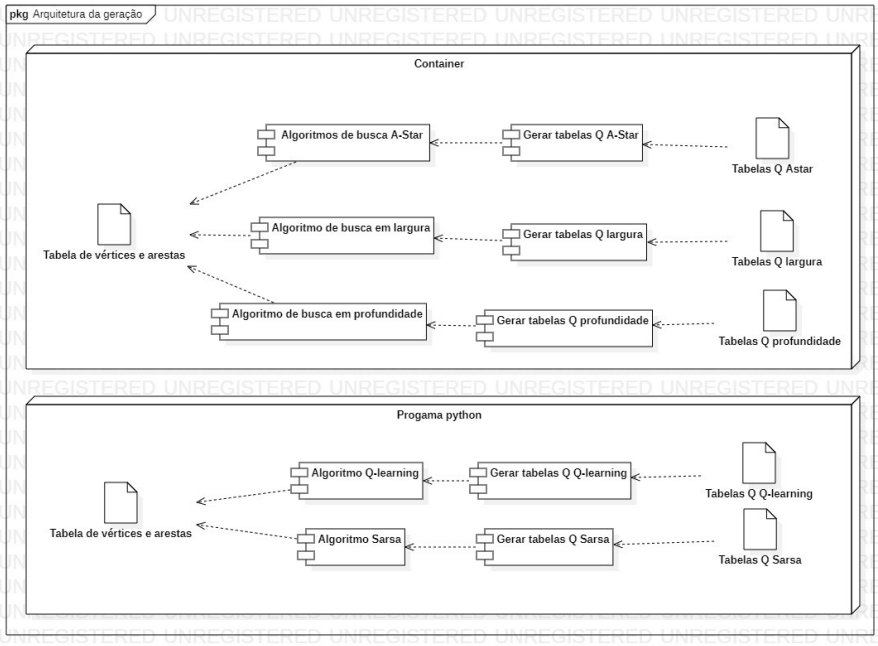


Figura 8: Arquitetura de geração das tabelas Q_{rot}.

Como podemos ver na Tabela 3, cada linha desta tabela representa um caminho até o vértice de destino 0 (*zero*), a primeira coluna de cada linha representa a origem, e a última representa o destino. Para alcançarmos o destino, é preciso percorrer as colunas de uma linha de forma sequencial.

1	0																
2	1	0															
3	2	1	0														
4	3	2	1	0													
5	6	44	45	3	2	1	0										
6	44	45	3	2	1	0											
7	6	44	45	3	2	1	0										
8	44	45	3	2	1	0											
9	8	44	45	3	2	1	0										
10	9	8	44	45	3	2	1	0									
11	10	9	8	44	45	3	2	1	0								
12	11	10	9	8	44	45	3	2	1	0							
13	12	11	10	9	8	44	45	3	2	1	0						
14	13	12	11	10	9	8	44	45	3	2	1	0					
15	14	13	12	11	10	9	8	44	45	3	2	1	0				
16	15	14	13	12	11	10	9	8	44	45	3	2	1	0			
17	16	15	14	13	12	11	10	9	8	44	45	3	2	1	0		
18	17	16	15	14	13	12	11	10	9	8	44	45	3	2	1	0	
19	20	21	22	60	59	58	57	56	10	9	8	44	45	3	2	1	0
20	21	22	60	59	58	57	56	10	9	8	44	45	3	2	1	0	
21	22	60	59	58	57	56	10	9	8	44	45	3	2	1	0		
22	60	59	58	57	56	10	9	8	44	45	3	2	1	0			
23	22	60	59	58	57	56	10	9	8	44	45	3	2	1	0		
24	23	22	60	59	58	57	56	10	9	8	44	45	3	2	1	0	
25	24	23	22	60	59	58	57	56	10	9	8	44	45	3	2	1	0
26	27	48	47	46	36	37	38	51	50	49	1	0					
27	48	47	46	36	37	38	51	50	49	1	0						
28	27	48	47	46	36	37	38	51	50	49	1	0					
29	48	47	46	36	37	38	51	50	49	1	0						
30	31	32	33	46	36	37	38	51	50	49	1	0					
31	32	33	46	36	37	38	51	50	49	1	0						
32	33	46	36	37	38	51	50	49	1	0							
33	46	36	37	38	51	50	49	1	0								
34	35	36	37	38	51	50	49	1	0								
35	36	37	38	51	50	49	1	0									
36	37	38	51	50	49	1	0										
37	38	51	50	49	1	0											
38	51	50	49	1	0												
39	49	1	0														
40	39	49	1	0													
41	45	3	2	1	0												
42	41	45	3	2	1	0											
43	42	41	45	3	2	1	0										
44	45	3	2	1	0												
45	3	2	1	0													
46	36	37	38	51	50	49	1	0									
47	46	36	37	38	51	50	49	1	0								
48	47	46	36	37	38	51	50	49	1	0							
49	1	0															
50	49	1	0														
51	50	49	1	0													
52	51	50	49	1	0												
53	43	42	41	45	3	2	1	0									
54	53	43	42	41	45	3	2	1	0								
55	54	53	43	42	41	45	3	2	1	0							
56	10	9	8	44	45	3	2	1	0								
57	56	10	9	8	44	45	3	2	1	0							
58	57	56	10	9	8	44	45	3	2	1	0						
59	58	57	56	10	9	8	44	45	3	2	1	0					
60	59	58	57	56	10	9	8	44	45	3	2	1	0				

Tabela 3: Tabela $Q_{rot_{A^*}}$ com destino ao vértice 0 *(zero)*.

Conforme observado na Tabela 4, cada linha da tabela representa um passo a ser realizado para alcançar o objetivo. Cada linha apresenta um vértice que serve como origem (*From*) e um vértice que serve como destino (*To*). Para atingir o objetivo, é necessário percorrer várias linhas dessa tabela até que o vértice de destino corresponda ao vértice objetivo.

from	to
1	0
2	1
3	2
4	3
5	4
6	5
7	8
8	44
9	8
10	9
11	10
12	11
13	12
14	13
15	14
16	15
17	16
18	17
19	18
20	21
21	22
22	60
23	24
24	25
25	26
26	27
27	48
28	29
29	48
30	29
31	32
32	33
33	46
34	35
35	36
36	37
37	38
38	39
39	49
40	39
41	45
42	41
43	44
44	45
45	3
46	36
47	46
48	47
49	1
50	49
51	50
52	51
53	43
54	53
55	54
56	10
57	11
58	12
59	58
60	59

Tabela 4: Tabela $Q_{rot_{QLearning}}$ com destino ao vértice 0 *(zero)*.

Geração de $Q_{rot_{A*}}$

O código abaixo é responsável por gerar as tabelas Q do algoritmo $A*$. Conforme mencionadas nas seções anteriores, o propósito fundamental destas tabelas é armazenar os caminhos para todos os vértices presentes no ambiente. Posteriormente, estas tabelas serão utilizadas pela *API* descrita na Seção 3.1.6.

```
1 for index_Destiny in range(0,61,1):
2     for index_Source in range(0,61,1):
```

Esse *Loop* irá percorrer todos os destinos e origens. Fazendo assim um destino para todos os outros vértices de origem.

```
    if index_Source == index_Destiny:
            continue
    def DistanciaEuclidiana(v, target):
            valor = (float(vposX[v]) - float(vposX[target]))**2 + (float(vposY[v]) - float(vposY[
    target]))**2
            return math.sqrt(valor)

    def PrenncherHeuristica(target):
            heuristic = []
            for v in v_name:
                index_atual = list(v_name).index(v)
                heuristic.append(DistanciaEuclidiana(index_atual, target))
            return heuristic

    heuristic = PrenncherHeuristica(index_Destiny)

    def he(v):
            return heuristic[v]
```

O trecho acima, se refere a como a heurística do $A*$ é preenchida, a função `Preencher Heuristica` calcula os valores de heurística para todos os vértices, com base em suas distâncias de um vértice alvo e inserindo esses valores em uma lista. A função `he` é definida para retornar os valores de heurística calculados anteriormente. Esses valores de heurística são utilizados posteriormente no algoritmo $A*$.

```
g_astar = gt.Graph()
v_name_dfs = g_astar.new_vertex_property("string")
e_ord = g_astar.new_edge_property("int")
e_action_bfs = g_astar.new_edge_property("string")

f_network = open("vertices.csv", 'r', encoding='utf-8')
reader_network = csv.reader(f_network, delimiter=",")
for vertice in reader_network:
    v = g_astar.add_vertex()
    v_name_dfs[v] = str(vertice[1])

f_network.close()


index_raiz = index_Source
ord = 1


for edge in gt.astar_iterator(g, g.vertex(index_raiz), e_distance, heuristic=lambda v: he(int(v)
)):
    e = g_astar.add_edge(int(edge.source()), int(edge.target()))
    e_ord[e] = ord
    e_action_bfs[e] = '(' + str(ord) + ') '
    ord += 1
```

O código lê informações de vértices de um arquivo `CSV` (`vertices.csv`) e cria vértices correspondentes no grafo `g_astar`. O algoritmo $A*$ é então aplicado usando a função `astar_iterator` do *Graph_ Tool*. A iteração ocorrerá através das arestas usando a heurística, construindo, por fim o, caminho no grafo `g_astar`.

```
    g_astar = gt.Graph()
    bfsv_name = g_astar.new_vertex_property("string")
    bfsv_time = g_astar.new_vertex_property("int")
    bfsv_name_time = g_astar.new_vertex_property("string")
    bfsv_color = g_astar.new_vertex_property("string")
    bfsv_dist = g_astar.new_vertex_property("int")
    bfsv_pred = g_astar.new_vertex_property("int64_t")
    bfse_color = g_astar.new_edge_property("string")
    bfse_action = g_astar.new_edge_property("string")
    bfse_ord = g_astar.new_edge_property("string")
    bfse_weight_bfs = g_astar.new_edge_property("float")
    v_pos  = g.new_vertex_property("vector<double>")

    index_raiz = index_Source
    index_alvo = index_Destiny

    gt.astar_search(g, g.vertex(index_Source), e_distance, VisitorExample(bfsv_name, bfsv_time,
     bfsv_name_time, bfsv_color, bfsv_dist, bfsv_pred, bfse_color, bfse_action, bfse_ord, g.vertex(
     index_Destiny)), heuristic=lambda v: he(int(v)))

    id_caminho = []
    id_caminho.insert(0, index_alvo)

    while index_alvo != index_raiz:
        e = g_astar.edge(bfsv_pred[index_alvo], index_alvo)
        if e is None:
            break;
        index_alvo = bfsv_pred[index_alvo]
        id_caminho.insert(0, index_alvo)
```

A busca $A*$ é realizada usando a função `astar_search`. As propriedades dos vértices e das arestas são atualizadas para manter informações relevantes durante a busca. O caminho é rastreado a partir do vértice de destino (`index_alvo`) até o vértice de origem (`index_raiz`). Na busca é utilizada a função (`he`) que contém as heurísticas de todos os vértices calculado para aquele destino. O caminho é colocado dentro da lista (`id_caminho`), que basicamente contém os *Ids* do vértices, é essa lista que é escrita no arquivo. Cada vértice vai ter um caminho para o destino, na iteração do segundo laço de repetição é realizado com que cada vértice tenha um caminho para o destino.

```
    path = [str(id) for id in id_caminho]
    path = ",".join(path)
     _file = open(f"table_q_AStar/table_q_{index_Destiny}.csv", "a", encoding="utf-8")
     _file.write(f"{path}\n")
     _file.close()
```

Por fim, `id_caminho` é convertido para *String* e armazenado em um arquivo `csv` correspondente ao destino.

Geração de $Q_{rot_{Largura}}$

A geração de caminhos será semelhante a do algoritmo $A*$, com a principal diferença sendo a abordagem de busca adotada. Por exemplo, na busca em largura, não há necessidade de incluir o código para calcular e preencher heurísticas, uma vez que a busca em

largura é um algoritmo de busca cega, como previamente mencionado. Portanto, tanto no método `gt.bfs_iterator()` quanto no `gt.bfs_search()`, não se faz uso de heurísticas. A principal diferença em relação ao $A*$ na geração dos caminho é justamente nesse aspecto. Á seguir, o código de geração dos caminhos do busca em largura abaixo:

```
index_Destiny = 0
index_Source = 0
for index_Destiny in range(0, 61, 1):
    for index_Source in range(0, 61, 1):
        if index_Source == index_Destiny:
            continue
        g_bfs = gt.Graph()
        v_name_bfs = g_bfs.new_vertex_property("string")
        e_ord = g_bfs.new_edge_property("int")

        f_network = open("vertices.csv", 'r', encoding='utf-8')
        reader_network = csv.reader(f_network, delimiter=",")
        for vertice in reader_network:
            v = g_bfs.add_vertex()
            v_name_bfs[v] = str(vertice[1])
        f_network.close()

        ord = 1
        for edge in gt.bfs_iterator(g, g.vertex(index_Source)):
            e = g_bfs.add_edge(int(edge.source()), int(edge.target()))
            e_ord[e] = ord
            ord += 1

        g_bfs = gt.Graph()
        v_name_bfs = g_bfs.new_vertex_property("string")
        v_name_time = g_bfs.new_vertex_property("string")
        dist = g_bfs.new_vertex_property("int")
        pred = g_bfs.new_vertex_property("int64_t")
        time = g_bfs.new_vertex_property("int")
        color = g_bfs.new_edge_property("string")
        v_color = g_bfs.new_vertex_property("string")

        index_raiz = index_Source
        gt.bfs_search(g, g.vertex(index_raiz), VisitorExample(v_name_bfs, pred, dist, time,
    v_name_time, color, v_color))

        index = index_Destiny
        path = []
        id_caminho = []

        path.insert(0, v_name_bfs[index])
        v_color[index] = "green"
        id_caminho.insert(0, index)
        while index != index_raiz:
            e = g_bfs.edge(pred[index], index)
            index = pred[index]
            path.insert(0, v_name_bfs[index])
            id_caminho.insert(0, index)

        path = [str(id) for id in id_caminho]
        path = ",".join(path)
        _file = open(f"table_q_Largura/table_q_{index_Destiny}.csv", "a", encoding="utf-8")
        _file.write(f"{path}\n")
        _file.close()
```

Geração de $Q_{rot_{Profundidade}}$

A mesma lógica aplicada na geração de caminhos para a busca em largura é aplicada na busca em profundidade, seguindo uma abordagem semelhante. No entanto, há modificações em dois métodos específicos: `gt.dfs_iterator()` e `gt.dfs_search()`, ambas as chamadas dos métodos acabam mudando, pois se tratar de uma busca em profundidade. Segue o código abaixo do gerador de caminhos para busca em profundidade:

```
index_Destiny = 0
index_Source = 0
for index_Destiny in range(0,61,1):
    for index_Source in range(0,61,1):
        if index_Source == index_Destiny:
            continue
        g_dfs = gt.Graph()
        v_name_dfs = g_dfs.new_vertex_property("string")
        e_ord = g_dfs.new_edge_property("int")

        f_network = open("vertices.csv", 'r', encoding='utf-8')
        reader_network = csv.reader(f_network, delimiter=",")
        for vertice in reader_network:
            v = g_dfs.add_vertex()
            v_name_dfs[v] = str(vertice[1])
        f_network.close()

        ord = 1
        for edge in gt.dfs_iterator(g, g.vertex(index_Source)):
            e = g_dfs.add_edge(int(edge.source()), int(edge.target()))
            e_ord[e] = ord
            ord += 1

        g_dfs = gt.Graph()
        dfsv_name      = g_dfs.new_vertex_property("string")
        dfsv_time      = g_dfs.new_vertex_property("int")
        dfsv_name_time = g_dfs.new_vertex_property("string")
        dfsv_color     = g_dfs.new_vertex_property("string")
        dfsv_dist      = g_dfs.new_vertex_property("int")
        dfsv_pred      = g_dfs.new_vertex_property("int64_t")
        dfse_color     = g_dfs.new_edge_property("string")
        dfse_action    = g_dfs.new_edge_property("string")
        dfse_ord       = g_dfs.new_edge_property("string")
        index_raiz = index_Source

        gt.dfs_search(g, g.vertex(index_raiz), VisitorExample(dfsv_name, dfsv_time,
    dfsv_name_time, fsv_color, dfsv_dist, dfsv_pred, dfse_color, dfse_action, dfse_ord
    ))

        index = index_Destiny
        path = []
        id_caminho = []

        path.insert(0,dfsv_name[index])
        dfsv_color[index] = "green"
        id_caminho.insert(0, index)
        while index != index_raiz:
            e = g_dfs.edge(dfsv_pred[index], index)
```

```
            index = dfsv_pred[index]
            path.insert(0,dfsv_name[index])
            id_caminho.insert(0, index)


        path = [str(id) for id in id_caminho]
        path = ",".join(path)
        _file = open(f"table_q_profundidade1/table_q_{index_Destiny}.csv", "a", encoding="utf-8")
        _file.write(f"{path}\n")
        _file.close()
```

Geração de $Q_{rot_{QLearning}}$

Para criar as tabelas de aprendizado nos algoritmos de aprendizado por reforço, foi necessário fazer uma preparação completa para a execução do ambiente. Isso incluiu a definição de variáveis constantes do projeto, a criação de classes para representar o ambiente proposto e os agentes que irão interagir com esse ambiente, bem como o desenvolvimento de métodos para manipulação de dados.

Primeiramente, definimos as variáveis do projeto, que são o `ALPHA`, `GAMMA`, `EPSILON` e o `DEFAULT_Q`.

```
ALPHA = 0.3  # Taxa de aprendizado
GAMMA = 0.7  # Fator de desconto
EPSILON = 0.9  # Taxa de exploracao
DEFAULT_Q = 0.0
```

Logo após, estabelecemos as classes, seus atributos e métodos destinados a representar e manipular o ambiente e o agente, conforme ilustrado a seguir:

```
class Edge:
    def __init__(self, start, end, distance: int, q=DEFAULT_Q) -> None:
        self.start = start
        self.end = end
        self.distance = distance
        self.q = q
```

A classe `Edge` representa uma aresta do nosso grafo. Como tal, ela possui os atributos `Start` e `End`, que indicam os vértices conectados por essa aresta (representa uma ação). Além disso, a classe armazena a distância entre esses vértices e o valor q associado a essa ação.

```
class Vertex:
    def __init__(self, id: int, name: str, category: str, r=0.0) -> None:
        self.id = id
        self.name = name
        self.edges = []
```

```
        self.category = category
        self.r = r

    def add_edge(self, edge: Edge) -> None:
        for e in self.edges:
            if e.end == edge.end:
                return
        self.edges.append(edge)

    def get_bigger_q_action(self) -> float:
        bigger_q = DEFAULT_Q
        for edge in self.edges:
            if edge.q > bigger_q:
                bigger_q = edge.q
        return bigger_q

    def get_best_action_index(self) -> int:
        edge_destiny = self.edges[0]
        for edge in self.edges:
            if edge.q > edge_destiny.q:
                edge_destiny = edge
        return self.edges.index(edge_destiny)
```

A classe `Vertex` corresponde a um vértice do grafo do ambiente, e ela guarda informações por meio dos atributos `id`, `name`, `category`, `r` e `edges`. Onde o `id` representa um identificador único, `name` corresponde ao nome do ponto ou estabelecimento, `category` denota a categoria do estabelecimento, `r` é o valor de recompensa imediata associado (atribuído ao vértice com o objetivo de ensinar o agente a alcançá-lo), e `edges` armazena as arestas que se conectam a esse vértice.

Adicionalmente a esses atributos, a classe `Vertex` possui alguns métodos para executar operações no próprio objeto. Estes incluem `add_edge`, que recebe uma aresta como parâmetro e a adiciona à lista atribuída a `edges`. Também encontramos a função `get_bigger_q_action`, a qual retorna o maior valor de Q armazenado dentro das arestas presentes na lista `edges`. Além disso, o método `get_best_action_index` é responsável por retornar o índice da lista de ações (`edges`) que contém a ação com o maior valor Q.

```
class Graph:
    def __init__(self) -> None:
        self.vertices = []
        self.start = None
        self.goal = None
        self.min_distance = 9999999
        self.max_distance = 0

    def read_csv(self) -> None:
        _file = open("vertices.csv", "r", encoding="utf-8-sig")
        for line in _file:
            line = line.split(",")
            vertex_id = int(line[0])
            name = str(line[1])
            category = str(line[2])
            self.add_vertex(Vertex(vertex_id, name, category))
        _file.close()
        _file = open("arestas.csv", "r", encoding="utf-8-sig")
        for line in _file:
```

```
                line = line.split(",")
                start = int(line[0])
                end = int(line[1])
                distance = int(line[2])
                if distance < self.min_distance:
                    self.min_distance = distance
                if distance > self.max_distance:
                    self.max_distance = distance
                self.add_edge(start, end, distance)
            _file.close()

    def get_all_vertices(self) -> list():
        return self.vertices

    def add_vertex(self, vertex: Vertex) -> None:
        self.vertices.append(vertex)

    def get_vertex_by_id(self, id: int) -> Vertex:
        for vertex in self.vertices:
            if str(vertex.id) == str(id):
                return vertex
        raise Exception(f"Vertex with id {id} not found")

    def set_goal(self, goal_vertex: Vertex) -> None:
        self.goal = goal_vertex

    def set_start(self, start_vertex: Vertex) -> None:
        self.start = start_vertex

    def add_edge(self, start: int, end: int, distance: int) -> None:
        start = self.get_vertex_by_id(start)
        end = self.get_vertex_by_id(end)
        start.add_edge(Edge(start, end, distance))
        end.add_edge(Edge(end, start, distance))

    def define_reward(self, reward: float, vertex: Vertex) -> None:
        vertex.r = reward
```

A classe `Graph` tem a responsabilidade de manter os atributos `vertices`, `start`, `goal`, `min_distance` e `max_distance`. O atributo `vertices` armazena a lista de todos os vértices do ambiente, `start` representa o vértice no qual o agente vai iniciar durante o treinamento. O atributo `goal` indica o ponto de destino do agente, `min_distance` e `max_distance` representam a menor e a maior distância entre dois vértices no ambiente. Estes valores são necessários para a aplicação posterior da fórmula de normalização $min - max$, que será explicada posteriormente.

Além destes atributos, também criamos os métodos `read_csv`, `get_all_vertices`, `add_vertex`, `get_vertex_by_id`, `set_goal`, `set_start`, `add_edge` e `define_reward`.

O método `read_csv` é responsável por ler um arquivo `CSV` para preencher as informações dos vértices no grafo, `get_all_vertices` retorna a lista de todos os vértices presentes no ambiente, `add_vertex` permite adicionar um novo vértice à lista de vértices do ambiente, `add_edge` adiciona uma nova aresta ao grafo, `get_vertex_by_id` recupera um vértice específico com base no seu `id`, `set_goal` define o vértice objetivo para o agente, `set_start` atribui o vértice inicial para o agente e, por fim, `define_reward` estabelece o valor de recompensa associado a um vértice específico.

Esses métodos são cruciais para a manipulação, configuração e utilização eficiente do grafo, contribuindo para a funcionalidade geral dos algoritmos de aprendizado por reforço.

```python
class Agent:
    def __init__(self, graph: Graph) -> None:
        self.graph = graph
        self.current = graph.start
        self.path = []
        self.epoch = 0
        self.path.append(self.current)
        self.delta_q_total = 0.0
        self.list_delta_q = [0]
        self.converged = False

    def get_random_action(self) -> int:
        return int(random.random() * len(self.current.edges))

    def random_policy(self) -> int:
        return int(self.get_random_action())

    def greedy_policy(self) -> int:
        # Has a chance of EPSILON to exploit
        if random.random() > EPSILON:
            if self.current.get_bigger_q_action() != DEFAULT_Q:
                return self.current.get_best_action_index()
        # Otherwise, explore
        return self.get_random_action()

    def greater_policy(self) -> int:
        return self.current.get_best_action_index()

    def reset_agent(self) -> None:
        random_vertex = self.graph.get_vertex_by_id(
            str(int(random.random() * len(self.graph.vertices))))

        self.current = random_vertex
        self.path.clear()
        self.path.append(self.current)

    def verify_convergence(self) -> bool:
        if is_converged(self.list_delta_q):
            self.converged = True
            return True
        return False

    def train(self) -> None:
        while not self.converged:
            while self.current != self.graph.goal:
                self.move()

            self.reset_agent()
```

A classe `Agent` contém os seguintes atributos: `graph`, `current`, `path`, `epoch`, `delta_q_total`, `list_delta_q` e `converged`.

O atributo `graph` armazena as informações do grafo que o agente percorrerá. O atributo `current` representa o vértice onde o agente se encontra no momento. Esse vértice é inicializado com o valor do atributo `start`, que é armazenado no objeto `graph`. O atributo `path` registra o percurso efetuado pelo agente durante uma iteração. O atributo `epoch` indica o número de iterações completas realizadas pelo agente, `delta_q_total` é a soma total das alterações nos valores Q ao longo de uma iteração, `list_delta_q` é uma

lista que guarda os valores individuais das mudanças nos valores Q. Essa lista é usada para verificar se o agente convergiu, ou seja, se ele atingiu ou está próximo do limite de aprendizado. Mais adiante, explicaremos o processo de verificação de convergência. A *Flag* `converged` indica se o agente conseguiu convergir a uma solução.

Esses atributos desempenham um papel fundamental no controle das ações do agente e na atualização dos valores Q durante o processo de aprendizado por reforço.

Além desses atributos, estão disponíveis os seguintes métodos: `get_random_action`, `random_policy`, `greedy_policy`, `greater_policy`, `reset_agent`, `verify_convergence` e `train`.

O método `get_random_action` gera uma decisão de ação aleatória para o agente, promovendo a exploração do ambiente, `random_policy` implementa uma política aleatória para o agente, enquanto `greedy_policy` e `greater_policy` introduzem políticas baseadas na maximização dos valores Q, `reset_agent` redefine o agente para um estado inicial, preparando-o para uma nova iteração e `verify_convergence` verifica se o agente convergiu, isto é, se suas ações estão estabilizando dentro de limites de aprendizado.

Por fim, o método `train` é central para o processo de aprendizado. Ele combina as políticas e os métodos mencionados acima para permitir que o agente explore, aprenda e atualize os valores Q conforme interage com o ambiente.

```
class QLearningAgent(Agent):
    def __init__(self, graph: Graph) -> None:
        super().__init__(graph)

    def update_q_value(self, next_vertex) -> None:
        has_change = False
        for edge in self.current.edges:
            if edge.end == next_vertex:
                normalized_distance = get_normalized_distance(edge.distance, self.graph.
    min_distance, self.graph.max_distance)
                delta_q = get_delta_q(edge.q, next_vertex.r,
                                      next_vertex.get_bigger_q_action(),
                                      normalized_distance)
                self.delta_q_total += delta_q
                edge.q = edge.q + delta_q
                if delta_q != 0:
                    has_change = True

        if has_change:
            self.list_delta_q.append(self.delta_q_total)

        self.epoch += 1
        if self.epoch % 5000 == 0:
            if is_converged(self.list_delta_q):
                self.converged = True

    def move(self) -> None:
        action_generated = self.greedy_policy()
        chosen_vertex = self.current.edges[action_generated].end
        self.update_q_value(next_vertex=chosen_vertex)
        self.current = chosen_vertex
        self.path.append(self.current)
```

A classe `QLearningAgent` herda as propriedades da classe `Agent`, mas além disso, ela possui 2 métodos que implementam a lógica do algoritmo Q-*Learning*: o método

`update_q_value` e o método `move`.

O método `update_q_value` recalcula e atualiza os valores Q com base nas recompensas obtidas e nas ações tomadas pelo agente. Ele emprega uma fórmula adaptada chamada `get_delta_q`, que leva em conta a distância ao atualizar os valores Q. Essa fórmula é apresentada abaixo:

```
def get_delta_q(actual_q: float, reward: float , max_q: float, normalized_distance: float) -> float:
    return (ALPHA + (1 - normalized_distance)) * (reward + GAMMA * max_q - actual_q)
```

Ao adicionar `(1 - normalized_distance)` à fórmula, estamos ajustando a quantidade de recompensa que a ação receberá, junto com a taxa de aprendizado. Dessa forma, a ação que envolve uma distância maior receberá um valor Q menor em comparação com a ação que possui uma distância menor. Isso equilibra a influência da distância nas atualizações dos valores Q durante o processo de aprendizado.

A seguir, temos a função que serve para mudar as distâncias entre os vértices para uma escala de 0 a 1. Isso ajuda a comparar as distâncias de forma justa, não importando o tamanho original delas. A fórmula da função é feita para manter os valores entre 0 e 1, mantendo as proporções de distância entre os números.

```
def get_normalized_distance(distance: float, max_distance: float, min_distance: float) -> float:
    return (distance - min_distance) / (max_distance - min_distance)
```

O método `move` orienta o agente do algoritmo Q-*Learning* a escolher a ação com o maior valor Q (seguindo a política $\epsilon - Greedy$) para o estado atual. Isso ajuda o agente a tomar decisões informadas. No entanto, também pode permitir a exploração ocasional para evitar ficar preso em decisões sub-ótimas.

Geração de $Q_{rot_{QSarsa}}$

Assim como a classe `QLearningAgent`, a classe `SarsaAgent` também herda as propriedades da classe `Agent` e os métodos que implementam a lógica do algoritmo *Sarsa*, sendo estes: o método `update_q_value` e o método `move`.

```
class SarsaAgent(Agent):

    def __init__(self, graph: Graph) -> None:
        super().__init__(graph)

    def update_q_value(self, next_vertex) -> None:
        has_change = False

        for edge in self.current.edges:
            if edge.end == next_vertex:
                old_current = self.current
                self.current = next_vertex
                next_action = self.greedy_policy()
                self.current = old_current
                next_edge = next_vertex.edges[next_action]
```

```
                    delta_q = get_delta_q_sarsa(edge.q, next_vertex.r, next_edge.q)
                    self.delta_q_total += delta_q
                    edge.q = edge.q + delta_q
                    if delta_q != 0:
                        has_change = True

            if has_change:
                self.list_delta_q.append(self.delta_q_total)

            self.epoch += 1

            if self.epoch == 500000:
                self.converged = True

    def move(self) -> None:
        next_action = self.greedy_policy()
        next_vertex = self.current.edges[next_action].end
        self.update_q_value(next_vertex)
        self.current = next_vertex
        self.path.append(self.current)
```

De maneira semelhante ao algoritmo Q-*Learning*, o método `update_q_value` recalcula e atualiza os valores Q com base nas recompensas obtidas e nas ações tomadas pelo agente. Entretanto, a fórmula de atualização foi denominada por `get_delta_q_sarsa`. Ao contrário da `get_delta_q`, essa fórmula não requer a utilização da distância, uma vez que a lógica de atualização de Q, utilizando a política atual do agente, pode levar a um aprendizado mais sensível às ações do agente em tempo real.

```
def get_delta_q_sarsa(actual_q: float, reward: float, next_q: float, normalized_distance: float) ->
    float:
    return ALPHA * (reward + GAMMA * next_q - actual_q))
```

De mesma maneira, o método `move` orientará o agente do algoritmo Q-*Sarsa* a escolher a ação com o maior valor Q (seguindo a política $\epsilon - Greedy$) para o estado atual.

Na sequência, apresentamos as funções responsáveis pela manipulação dos dados, os quais serão detalhados a seguir.

```
def full_run(algorithm: Agent):
    g = Graph()
    g.read_csv()
    all_vertex = g.get_all_vertices()
    for vertex in all_vertex:
        print(f"{int(all_vertex.index(vertex) / len(all_vertex) * 100)}% - Running for vertex {
    vertex.name} - {vertex.id + 1} of {len(all_vertex)} ")
        g = Graph()
        g.read_csv()
        start_vertex = g.get_vertex_by_id(1)
        g.set_start(start_vertex)
        goal_vertex = g.get_vertex_by_id(vertex.id)
        g.set_goal(goal_vertex=goal_vertex)
        g.define_reward(10, g.goal)
        a = algorithm(g)
        a.train()
        save_table_q(g, file_name=f"table_q_{vertex.id}.csv")
```

O método `full_run` recebe como parâmetro o tipo de agente que realizará o treinamento, podendo ser `QLearningAgent` ou `SarsaAgent`. A funcionalidade desse método é gerar as tabelas Q para todos os vértices, em relação a todos os outros vértices.

```
def save_table_q(g: Graph, file_name: str = "table_q.csv") -> None:
    if not os.path.exists("table_q"):
        os.mkdir("table_q")

    if file_name in os.listdir("table_q"):
        os.remove(f"table_q/{file_name}")
    _file = open(f"table_q/{file_name}", "w", encoding="utf-8")

    # filter just the biggest q for each edge
    for vertex in g.get_all_vertices():
        bigger_q = DEFAULT_Q
        start = vertex
        end = vertex
        for edge in vertex.edges:
            if edge.q > bigger_q:
                bigger_q = edge.q
                start = edge.start
                end = edge.end

        _file.write(f"{start.id},{end.id}\n")
    _file.close()

```

Por fim, o método `save_table_q` recebe um grafo como parâmetro, extrai os dados de aprendizado que foram salvos nele, filtra os dados deixando apenas a ação com maior valor Q de cada vértice e os armazena em arquivos no formato `CSV`. Os resultados desses arquivos podem ser visualizados na Tabela 3, a qual possui como destino o vértice 0 *(Zero)*.

3.2.3 Enviesamento de Rotas e Áreas de Interesse

Para introduzir o enviesamento nas rotas traçados pelo modelo, optamos por usar pontos de interesse. Neste caso, o usuário selecionará quais e quantos pontos de interesse deseja visitar.

Para tal finalidade, foi incluído um componente na arquitetura do modelo proposto que consistirá de um sistema de comunicação entre o cliente (dispositivo móvel) e o servidor responsável pela geração dos trajetos, conforme representado na Figura 9.

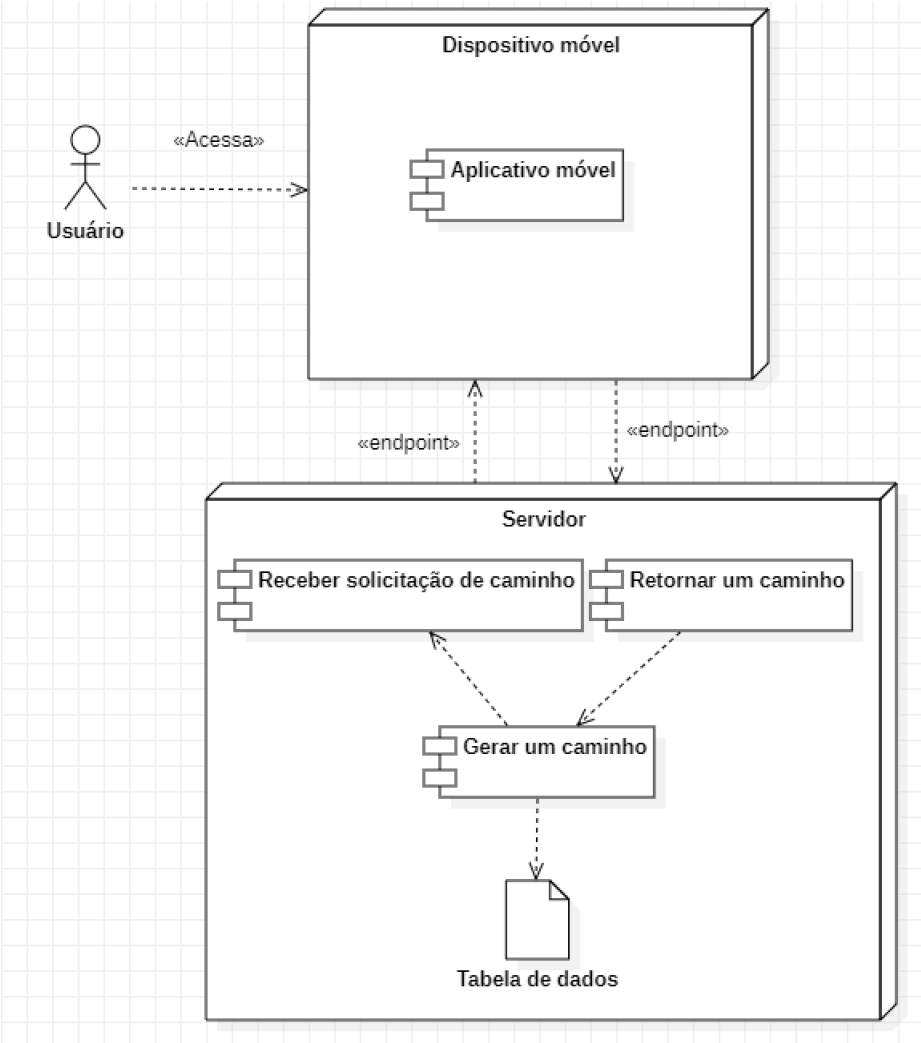


Figura 9: Arquitetura de comunicação entre o usuário e o servidor.

O modelo é composto por um aplicativo cliente integrado à aplicação móvel desenvolvida em *Flutter*. Nesse aplicativo, são executados trechos de código escritos em *Dart*, os quais realizam requisições ao servidor *Web*. O servidor abriga uma *Api* construída utilizando o *Framework FastAPI* em *Python*.

As informações transmitidas para o servidor nas requisições incluem a origem e o destino desejados, o algoritmo responsável pela tabela Q_{rot} a ser utilizada na determinação do trajeto, as categorias de estabelecimentos de interesse do usuário e o número de pontos de interesse pelos quais o usuário deseja passar.

O servidor, por sua vez, retorna um JSON[5] para o usuário que contém as seguintes informações: a rota, que é representada como uma lista de *IDs* dos vértices que o cliente deve percorrer, a distância total percorrida e a quantidade de passos necessários para atingir o objetivo. A seguir, apresentamos um exemplo de retorno fornecido pela *API* ao enviar a seguinte entrada de dados: $origem = 52$, $destino = 0$ e algoritmo responsável pela tabela Q_{rot} foi o Q-*Learning*.

[5]JSON (*JavaScript Object Notation*) é um formato leve de troca de dados amplamente utilizado para representar informações estruturadas de maneira legível por humanos e de fácil interpretação por máquinas (Crockford, 2006).

```
{
  "path": [52, 37, 38, 39, 40, 41, 45, 44, 8, 9],
  "total_distance": 158,
  "steps": 9
}
```

```
list_of_interests_available = [
    "Tech", "Food", "Entertainment", "Fashion", "Market", "Automotive",
    "Drink", "Fit", "Games"
]

app = FastAPI(swagger_ui_parameters={"defaultModelsExpandDepth": -1})

@app.get("/path/{id_origin}/{id_target}/{algorithm}", tags=["Path"])
async def get_path_request(
    id_origin: int,
    id_target: int,
    algorithm: Literal["QLearning", "Sarsa", "Astar", "Largura", "Profundidade"] = "QLearning",
    max_interests: int = 0,
    interests: str | None = Query(
        None,
        description=
        "Uma lista de palavras separadas por virgula. Interesses disponiveis: "
        + ", ".join(list_of_interests_available))):

    path = []
    algorithm = algorithm.lower()

    # sanitize the inputs of interests
    if interests is not None:
        interests = interests.split(",")
        interests = [i.lower().strip() for i in interests]

    # check if the algorithm is a search algorithm
    if algorithm in ["largura", "profundidade", "astar"]:
        # verify if the user wants to get the path without interests
        if interests is None:
            path, total_distance = get_path_search(start_id=id_origin,
                                                   goal_id=id_target,
                                                   algorithm=algorithm)
        # otherwise, get the path with interests
        else:
            path, total_distance = get_path_search_interest(
                start_id=id_origin,
                goal_id=id_target,
                max_interests=max_interests,
                interests=interests,
                algorithm=algorithm)

    # check if the algorithm is a reinforcement learning algorithm
    if algorithm in ["qlearning", "sarsa"]:
        # verify if the user wants to get the path without interests
        if interests is None:
            path, total_distance = get_path_rl(start_id=id_origin,
                                               goal_id=id_target,
                                               algorithm=algorithm)
        # otherwise, get the path with interests
        else:
```

```
            path, total_distance = get_path_rl_interest(
                start_id=id_origin,
                goal_id=id_target,
                max_interests=max_interests,
                interests=interests,
                algorithm=algorithm)

    # return a json with the path and the total distance
    return {
        "path": path,
        "total_distance": total_distance,
        "steps": len(path) - 1
    }


@app.get("/", include_in_schema=False)
async def root():
    return RedirectResponse(url="/docs")


def config_openapi():
    if app.openapi_schema:
        return app.openapi_schema
    openapi_schema = get_openapi(
        title="Api for Bias Pathfinding",
        version="1.0.0",
        description=
        "Essa e uma api para encontrar caminhos com vies em ambientes indoor.",
        routes=app.routes,
    )

    app.openapi_schema = openapi_schema

    return app.openapi_schema


app.openapi = config_openapi

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="localhost", port=8000
```

O código acima tem a responsabilidade de configurar o servidor, receber as requisições, realizar a sanitização dos dados recebidos e encaminhá-los para as funções correspondentes, dependendo do tipo de requisição feita. Isso inclui a diferenciação entre solicitações para métodos de busca ou de aprendizado por reforço, bem como a determinação se será aplicado enviesamento ou não. Os detalhes destes métodos serão apresentados logo a seguir:

```
edges_dict = {}
# save all edges in a dict
_file = open("arestas.csv", "r", encoding="utf-8-sig")
for line in _file.readlines():
    line_split = line.split(",")
    edges_dict[line_split[0] + "-" + line_split[1]] = float(line_split[2])
    edges_dict[line_split[1] + "-" + line_split[0]] = float(line_split[2])

```

```
vertices_dict = {}
# save all vertexes in a dict
_file = open("vertices.csv", "r", encoding="utf-8-sig")
for line in _file.readlines():
    line_split = line.split(",")
    vertices_dict[line_split[0]] = {
        "name": line_split[1],
        "category": line_split[2],
        "pos_x": float(line_split[3]),
        "pos_y": float(line_split[4])
    }
```

O código acima executa a operação de extrair os dados dos arquivos de arestas e vértices, e os armazena em um dicionário, uma estrutura de dados da linguagem *Python*. Ao utilizar esses dicionários para armazenar os dados, obtemos facilidade no acesso aos mesmos para realizar operações dentro das funções que serão discutidas a seguir.

```
def get_distance(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

A função `get_distance` tem a responsabilidade de receber 4 parâmetros que representam as coordenadas X e Y entre dois pontos, e retorna a distância euclidiana desse cálculo.

```
# get the path from search algorithms (BFS, DFS, A*) without interests
def get_path_search(start_id: int, goal_id: int, algorithm: str):
    # get path from start to goal
    directory = f"table_q/{algorithm}/table_q_{goal_id}.csv"
    _file = open(directory, "r", encoding="utf-8-sig")
    path = []
    for line in _file.readlines():
        line_split = line.split(",")
        if line_split[0] == str(start_id):
            path = [int(i) for i in line_split]
            break

    # calculate the total distance
    total_distance = 0
    for i in range(len(path) - 1):
        total_distance += float(edges_dict[str(path[i]) + "-" +
                                           str(path[i + 1])])

    return path, total_distance
```

A função `get_path_search` recebe os parâmetros `start_id` (vértice de origem), `goal_id` (vértice de destino) e `algorithm` (algoritmo de busca utilizado). Essa função acessa o arquivo gerado da tabela Q do vértice de destino, criada pelo algoritmo de busca que foi passado como parâmetro e retorna o caminho gerado sem a aplicação de enviesamento.

```
# get the path from reinforcement learning algorithms (Q-Learning, SARSA) without interests
def get_path_rl(start_id: int, goal_id: int, algorithm: str):
    # get table from start to goal
```

```
    directory = f"table_q/{algorithm}/table_q_{goal_id}.csv"
    _file = open(directory, "r", encoding="utf-8-sig")
    table = {}
    for line in _file.readlines():
        line_split = line.split(",")
        table[line_split[0]] = {
            "next_vertex": int(line_split[1]),
        }

    # save the path and calculate the total distance
    current = start_id
    path = []
    path.append(current)
    total_distance = 0
    while current != goal_id:
        distance = edges_dict[str(current) + "-" + str(table[str(current)]["next_vertex"])]
        current = int(table[str(current)]["next_vertex"])
        path.append(current)
        total_distance += distance

    return path, total_distance
```

A função `get_path_rl`, de maneira semelhante a função descrita anteriormente, recebe os parâmetros `start_id` (vértice de origem), `goal_id` (vértice de destino) e `algorithm` (algoritmo de aprendizado por reforço). Essa função acessa o arquivo gerado da tabela Q do vértice de destino, criada pelo algoritmo de aprendizado por reforço que foi passado como parâmetro e retorna o caminho gerado sem a aplicação de enviesamento.

```
def get_path_search_interest(start_id: int, goal_id: int, max_interests: int, interests: list,
algorithm: str):
    # save a dict of vertexes with the same category
    interest_vertexes_dict = {}
    for vertex_id in vertices_dict:
        if int(vertex_id) == start_id or int(vertex_id) == goal_id:
            continue
        if vertices_dict[vertex_id]["category"].lower() in [i.lower() for i in interests]:
            interest_vertexes_dict[vertex_id] = vertices_dict[vertex_id]

    # set the number of stabilishments of interest to visit
    n_iterations = max_interests if max_interests < len(
        interest_vertexes_dict) else len(interest_vertexes_dict)

    # go through the interest vertexes
    total_distance = 0
    total_path = []
    current = start_id
    while n_iterations > 0 and len(interest_vertexes_dict) > 0:
        smaller_distance = 99999999
        shortest_vertex = None
        # find the next shortest interest vertex
        for vertex_id in interest_vertexes_dict:
            distance = get_distance(vertices_dict[str(current)]["pos_x"],
                                    vertices_dict[str(current)]["pos_y"],
                                    vertices_dict[vertex_id]["pos_x"],
                                    vertices_dict[vertex_id]["pos_y"])
            if distance < smaller_distance:
                smaller_distance = distance
                shortest_vertex = vertex_id
```

```
30
31          # go through the next shortest interest vertex
32          path = []
33          directory = f"table_q/{algorithm}/table_q_{shortest_vertex}.csv"
34          _file = open(directory, "r", encoding="utf-8-sig")
35          for line in _file.readlines():
36              line_split = line.strip().split(",")
37              if line_split[0] == str(current):
38                  path = [int(i) for i in line_split]
39                  break
40
41          # calculate the total distance
42          for i in range(len(path) - 1):
43              total_distance += float(edges_dict[str(path[i]) + "-" +
44                                                 str(path[i + 1])])
45
46          # save path
47          total_path += path
48          total_path.pop()  # remove the last vertex of the path because it is the same as the first
    vertex of the next path
49
50          # update the current vertex and the number of iterations
51          current = shortest_vertex
52          n_iterations -= 1
53
54          # remove the vertex from the interest vertexes dict
55          interest_vertexes_dict.pop(shortest_vertex)
56
57      # get the path from the last interest vertex to the goal
58      path, distance = get_path_search(current, goal_id, algorithm)
59
60      # save the path and the total distance
61      total_path += path
62      total_distance += distance
63
64      return total_path, total_distance
65
```

A função `get_path_search_interest` aceita os seguintes parâmetros: `start_id` (vértice de origem), `goal_id` (vértice de destino), `max_interests` (número máximo de vértices que a rota deve obrigatoriamente gerar), `interests` (lista de categorias dos estabelecimentos que interessam ao usuário) e `algorithm` (algoritmo de aprendizado por reforço).

Essa função é bastante semelhante à função `get_path_search`, contudo a principal diferença está na criação de rotas com viés. O processo tem início com um *loop* através de todos os vértices, onde são armazenados aqueles que partilham os mesmos interesses especificados nos parâmetros. Depois, é determinado qual é o próximo ponto de interesse mais próximo em relação ao estado atual, desde que o número máximo de interesses (`max_interests`) não seja excedido.

Após identificar o ponto mais próximo, a função acessa a tabela Q relacionada a esse vértice de interesse e concatena a rota existente nessa tabela, visto que as tabelas Q dos algoritmos de busca guardam toda a rota. Após finalizar a iteração pelos estabelecimentos de interesse, é essencial concatenar essa rota com o trajeto até o destino.

Nesse ponto, o agente acessa a tabela Q vinculada ao vértice de destino e a incorpora à rota acumulada anteriormente.

```python
def get_path_rl_interest(start_id: int, goal_id: int, max_interests: int,
                         interests: list, algorithm: str):
    # save a dict of vertexes with the same category
    interest_vertexes_dict = {}
    for vertex_id in vertices_dict:
        if int(vertex_id) == start_id or int(vertex_id) == goal_id:
            continue
        if vertices_dict[vertex_id]["category"].lower() in [i.lower() for i in interests]:
            interest_vertexes_dict[vertex_id] = vertices_dict[vertex_id]

    # set the number of stabilishments of interest to visit
    n_iterations = max_interests if max_interests < len(
        interest_vertexes_dict) else len(interest_vertexes_dict)

    # go through the interest vertexes
    total_distance = 0
    total_path = []
    current = start_id
    while n_iterations > 0 and len(interest_vertexes_dict) > 0:
        # get the path from the current vertex to the next shortest interest vertex
        smaller_distance = 99999999
        shortest_vertex = None

        # find the next shortest interest vertex
        for vertex_id in interest_vertexes_dict:
            distance = get_distance(vertices_dict[str(current)]["pos_x"],
                                    vertices_dict[str(current)]["pos_y"],
                                    vertices_dict[vertex_id]["pos_x"],
                                    vertices_dict[vertex_id]["pos_y"])
            if distance < smaller_distance:
                smaller_distance = distance
                shortest_vertex = vertex_id
        # get the path from the current vertex to the next shortest interest vertex
        path, distance = get_path_rl(current, int(shortest_vertex), algorithm)

        # save the path and the total distance
        total_path += path
        total_distance += distance
        total_path.pop()  # remove the last vertex of the path because it is the same as the first
    vertex of the next path

        # update the current vertex and the number of iterations
        current = path[-1]
        n_iterations -= 1

        # remove the vertex from the interest vertexes dict
        interest_vertexes_dict.pop(str(current))

    # get the path from the last interest vertex to the goal
    path, distance = get_path_rl(current, goal_id, algorithm)

    # save the path and the total distance
    total_path += path
    total_distance += distance

    return total_path, total_distance
```

A função `get_path_rl_interest` recebe os seguintes parâmetros: `start_id` (vértice de origem), `goal_id` (vértice de destino), `max_interests` (número máximo de vértices que a rota deve gerar obrigatoriamente), `interests` (lista de categorias dos estabelecimentos que o usuário tem interesse) e `algorithm` (algoritmo de aprendizado por reforço).

Esta função é bastante semelhante à função `get_path_search_interest`, exceto pela maneira como as tabelas Q geradas pelos algoritmos de aprendizado por reforço são percorridas. O processo começa com uma iteração por todos os vértices, onde são armazenados aqueles que compartilham os mesmos interesses passados como parâmetros. Em seguida, é procurado o próximo ponto de interesse mais próximo em relação ao estado atual, desde que esse número não exceda o limite definido pela variável `max_interests`.

Ao identificar o ponto mais próximo, a função chama a função `get_path_rl` e concatena seu retorno. Após finalizar essa operação com os estabelecimentos de interesse, é necessário concatenar essa rota com o trajeto até o objetivo. Nesse ponto, o agente percorre os vértices até alcançar o vértice de destino.

3.2.4 Aplicação *Mobile*

Por fim, o último componente da arquitetura que compõe este projeto de pesquisa é a aplicação *mobile* desenvolvida. Para uma apresentação visual abrangente do fluxo de interação do usuário com o sistema, destacamos as três telas móveis centrais que compõem essa experiência.

De uma maneira descritiva, ao iniciar o aplicativo, o usuário será recebido pela tela de carregamento (ver Figura 10), que estabelece uma primeira impressão e prepara o ambiente para a interação subsequente.

Figura 10: Tela de Carregamento (Splash).

A segunda tela, apresentada na Figura 11, é a tela de formulário, onde o usuário será

solicitado a preencher informações essenciais. Nesse ponto, deverá inserir detalhes como origem e destino desejados, escolher o algoritmo responsável pela tabela Q a ser usada na determinação do trajeto, selecionar uma ou mais categorias e indicar o número de locais de interesse que deseja visitar.

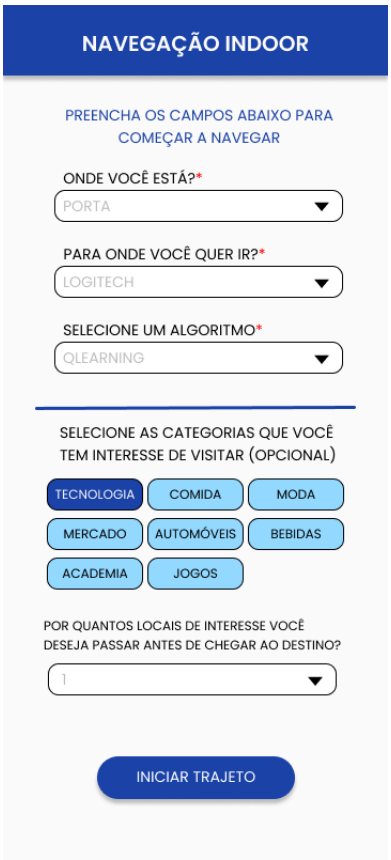


Figura 11: Tela de Formulário.

A terceira tela (Figura 12), apresenta três estados distintos: o primeiro ocorre quando a navegação é iniciada, o segundo durante a trajetória e o terceiro quando o destino é alcançado.

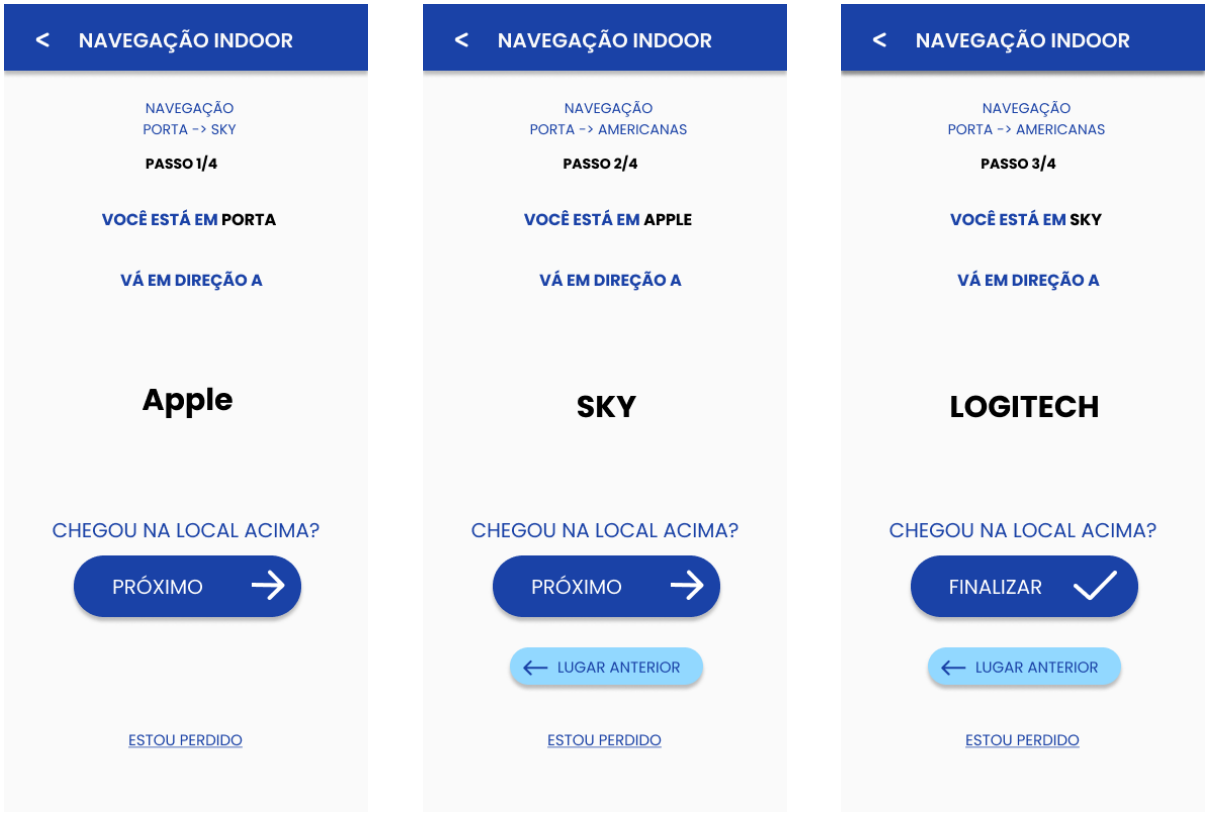
Sobre o funcionamento da aplicação, de uma maneira descritiva, ao iniciá-la, a tela de carregamento é exibida (Figura 10), e logo em seguida, será aberta a tela do formulário (Figura 11). Ao selecionar os valores *Porta* no campo *Onde você está?*, *Logitech* no campo *Para onde você quer ir?*, e *A** no campo *Selecione um Algoritmo*, e ao pressionar o botão *Iniciar um Trajeto*, o aplicativo apresentará a primeira tela do trajeto gerado, como mostrado na Figura 12(a). Essa tela fornece informações sobre a localização atual *Porta* e a direção ao qual o usuário deverá seguir, neste caso, em direção ao estabelecimento *Apple*, à fim de alcançar o objetivo final desejado.

Na parte inferior da Figura 12 (a), (b) e (c), estão os seguintes botões de orientação: *Próximo*, que o levará ao próximo estabelecimento da rota, *Lugar Anterior*, que retornará ao estabelecimento anterior (disponível após alguma movimentação), e *Estou Perdido*, que redirecionará o usuário de volta à tela inicial (Figura 11) para a inserção de novas informações e geração de uma nova rota.

Ao pressionarmos o botão *Próximo* até alcançar o penúltimo estabelecimento, esse botão será substituído pelo botão *Finalizar*, uma vez que o próximo passo será o objetivo, conforme representado na Figura 12 (c). Por fim, ao pressionar o botão *Finalizar*, o usuário será direcionado para a tela de formulário inicial.

Além do roteamento direto para o objetivo desejado, existe também a opção da seleção de uma ou mais áreas de interesse, pelas quais o usuário gostaria de passar à caminho de seu objetivo final. Neste caso, o roteamento será enviesado de acordo com as opções do usuário.

De qualquer forma, à nível de aplicação, caso seja selecionada alguma opção de interesse presente no formulário, que tratamos como enviesamento neste trabalho, a aplicação funcionará de maneira semelhante, tendo como única diferença que a rota passará pelos locais desejados, entretanto, de maneira transparente para o o usuário.



(a) (b) (c)

Figura 12: Tela de navegação ao longo do percurso, demonstrando os diferentes estados que pode assumir.

4 Resultados e discussões

Nesta seção, serão apresentados alguns resultados gerados pelas estratégias de buscas e aprendizado por reforço em um cenário de busca. Foram realizados experimentos e organizados em dois formatos.

No primeiro, será considerado um determinado percurso, composto por uma origem e um destino, onde analisaremos os resultados alcançados por cada um dos métodos de busca estudados.

No segundo, com o processamento organizado em *Batch*[6], uma sequência aleatória de origens e destinos serão analisadas, onde o objetivo é analisar o comportamento dos resultados alcançados a longo prazo.

4.1 Experimentos por percurso

O experimento à seguir realiza uma simulação de um percurso no ambiente, com os algoritmos apresentados no trabalho, com e sem enviesamento.

Percursos direto

Foram definidos como origem e destino os estabelecimentos *Musicyan* e *JBL*, respectivamente. Nas figuras à seguir é possível visualizar o caminho tracejado (vermelho) e os vértices visitados (azul) até alcançar o destino (verde).

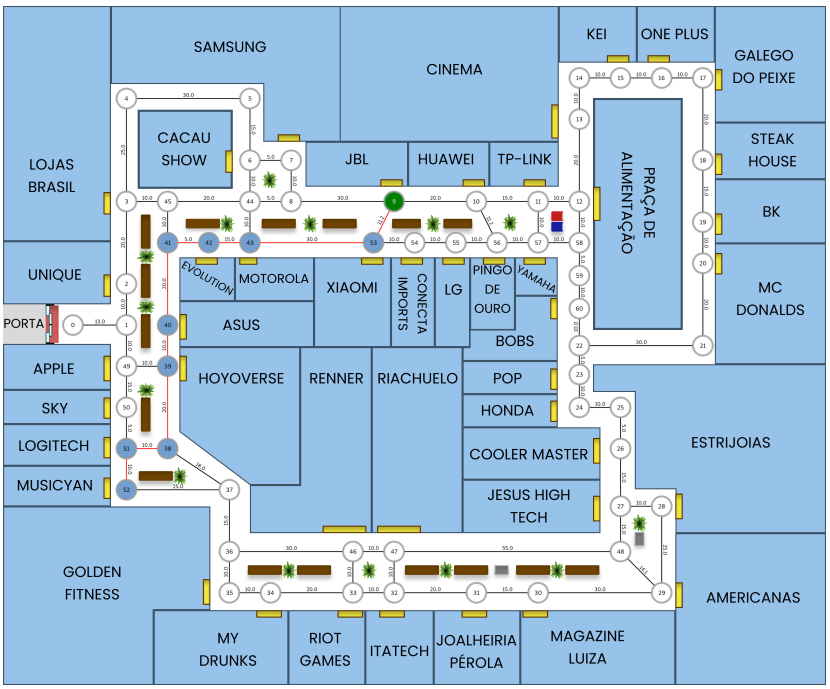


Figura 13: Percurso em A^* - Orig.: *Musicyan* / Dest.: *JBL*.

O primeiro experimento por percurso direto realizado utilizou a tabela $Q_{rot_{A^*}}$, no qual obteve uma distância de 131, 2 metros até o destino, passando por 9 pontos no ambiente. Podemos visualizar a representação gráfica na Figura 13.

[6]Execução de uma série de tarefas, sendo adequada para ambientes que não precisem de interatividade, e que possua tarefas de longa execução.

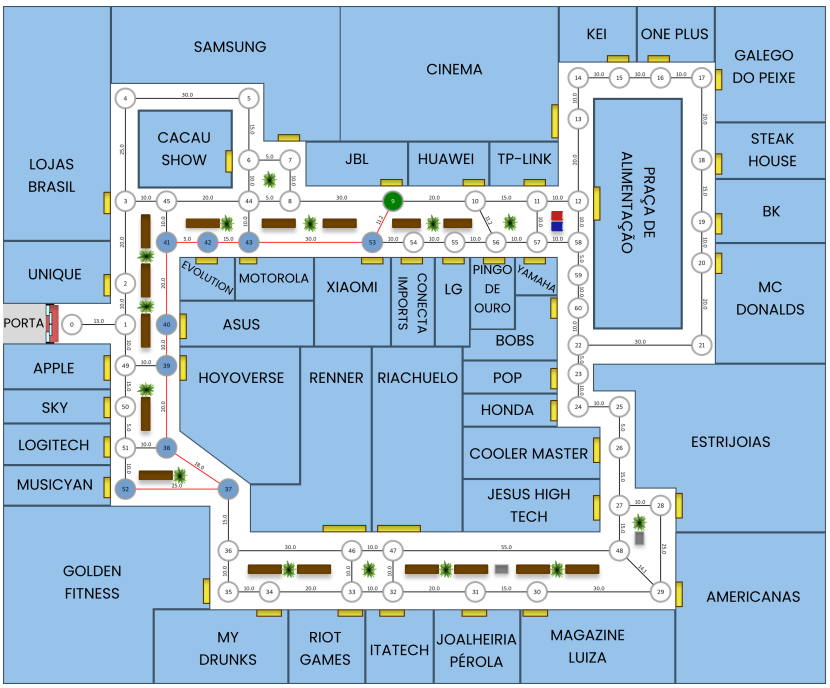


Figura 14: Percurso em *Largura* - Orig.: *Musicyan* / Dest.: *JBL*.

O segundo experimento por percurso direto foi realizado utilizando a tabela $Q_{rot_{Largura}}$, o qual resultou em uma distância total de 154, 2 metros, percorrendo 9 pontos no ambiente.

Neste experimento foi possível observar que, apesar de ter obtido o mesmo número de passos do experimento anterior utilizando a tabela $Q_{rot_{A^*}}$, ele obteve uma distância maior. Isto se deve ao fato de que o algoritmo de busca em largura não utiliza pesos locais e heurísticas, seu foco é exclusivamente expandir o menor número possível de nós. Sendo assim, durante a expansão dos nós, o primeiro caminho mais curto foi encontrado considerando, especificamente, quantidade de vértices, porém isto não garante uma distância menor. Podemos visualizar a representação gráfica na Figura 14.

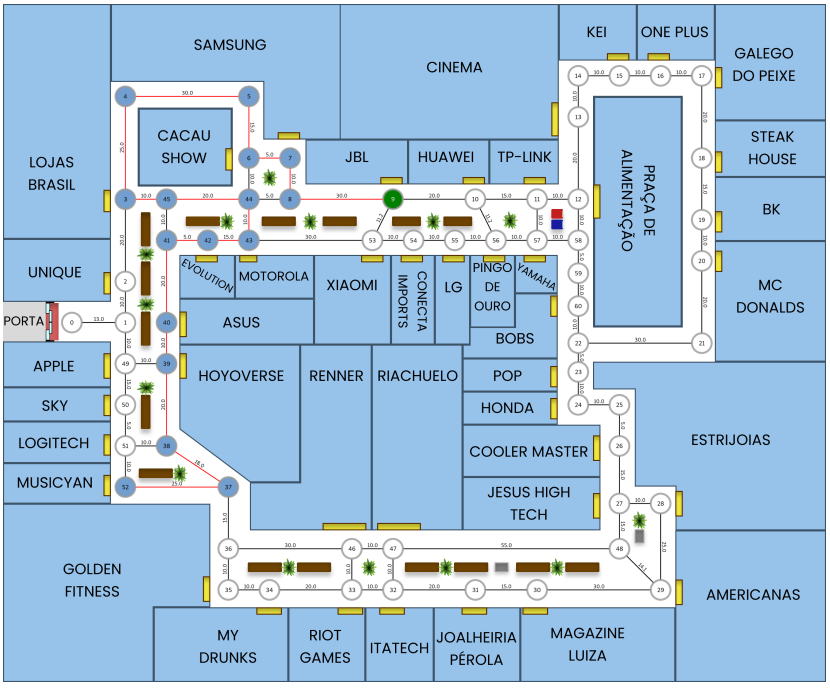


Figura 15: Percurso em *Profundidade* - Orig.: *Musicyan* / Dest.: *JBL*.

No terceiro experimento por percurso direto foi utilizada a tabela $Q_{rot_{Profundidade}}$. Observamos que, dentre todas as tabelas Q_{rot} dos algoritmos de busca, esta foi a que obteve o pior desempenho em relação à distância percorrida.

Neste experimento foi percorrida a distância total de 268 metros, passando por 16 pontos no ambiente. Tal distância se deve ao fato do algoritmo de busca em profundidade explorar um único ramo, e a ordem de expansão desse ramo gerou um desvio antes de ir diretamente ao objetivo, como podemos visualizar na Figura 15.

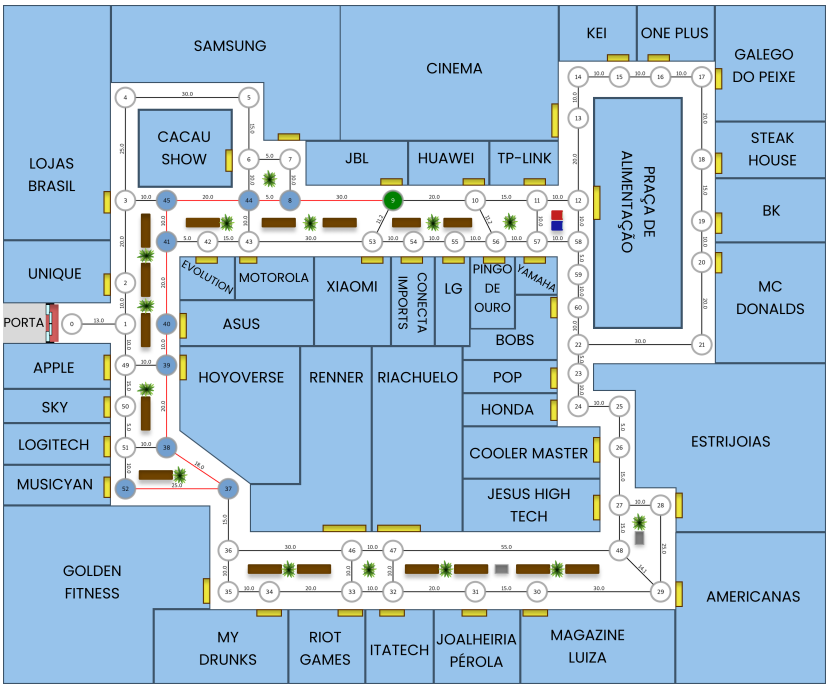


Figura 16: Percurso em *Q-Learning* - Orig.: *Musicyan* / Dest.: *JBL*.

No quarto experimento por percurso direto utilizamos a tabela $Q_{rot_{QLearning}}$, a qual resultou em uma distância percorrida de 158 metros, passando por 9 pontos no ambiente. Podemos visualizar a representação gráfica na Figura 16

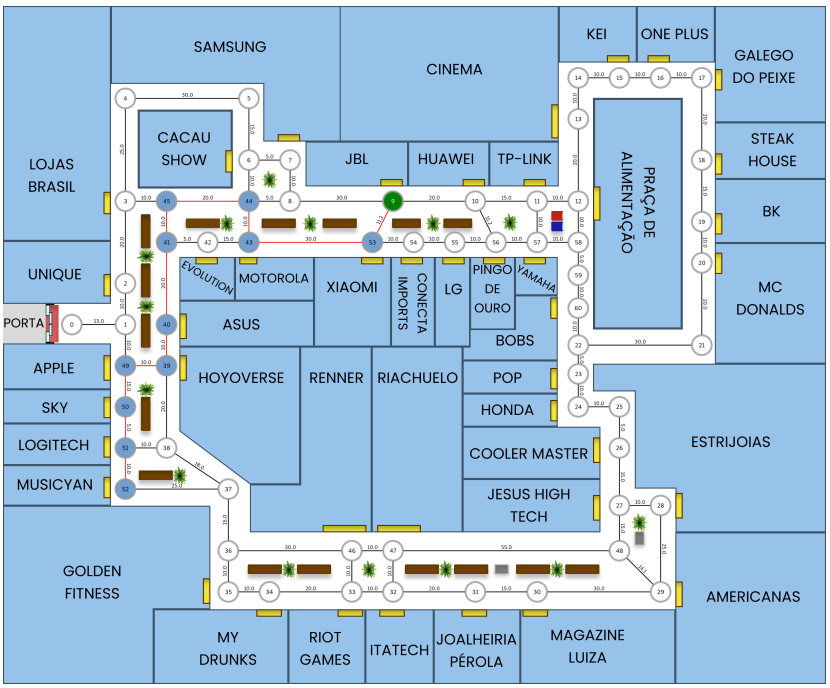


Figura 17: Percurso em *Sarsa* - Orig.: *Musicyan* / Dest.: *JBL*.

No quinto experimento de percurso direto, utilizamos a tabela $Q_{rot_{Sarsa}}$, a qual resultou em uma distância percorrida de 151,2 metros, atravessando 11 pontos no ambiente. Neste caso, a tabela $Q_{rot_{Sarsa}}$ obteve um desempenho superior ao da tabela $Q_{rot_{QLearning}}$, quando se considera a distância percorrida, embora tenha passado por mais pontos no ambiente a distância percorrida foi menor. A representação gráfica pode ser observada na Figura 17.

Como podemos observar nas Figuras 13, 14, 15, 16 e 17, cada algoritmo gerou rotas distintas entre si. Isso demonstra que cada algoritmo possui suas próprias vantagens e limitações, as quais têm um impacto direto na eficiência e qualidade das rotas geradas.

O algoritmo de busca $A*$ se destacou pela sua capacidade de encontrar o caminho mais curto, utilizando heurísticas e pesos locais para orientar suas decisões. Por outro lado, a busca em largura, embora priorize a expansão do menor número de nós, essa característica pode resultar em trajetos mais longos, contudo, nestes experimentos demonstrou resultados próximos ao $A*$.

Os algoritmos de aprendizado por reforço Q-*Learning* e *Sarsa* demonstraram-se eficientes para o ambiente, tendo obtido resultados próximos ao $A*$ em termos de distância percorrida.

Por outro lado, a busca em profundidade revelou limitações ao explorar um único ramo. Esse comportamento resultou em trajetos desviados e distâncias percorridas maiores, o que levou a resultados piores em relação aos demais algoritmos analisados.

Percursos com enviesamento

Consideramos neste experimento a mesma origem e destino do experimento anterior, ou seja, os estabelecimentos *Musicyan* e *JBL*. Para o enviesamento a ser analisado, a loja de categoria alimentícia *Cacau Show* foi selecionada como local de interesse.

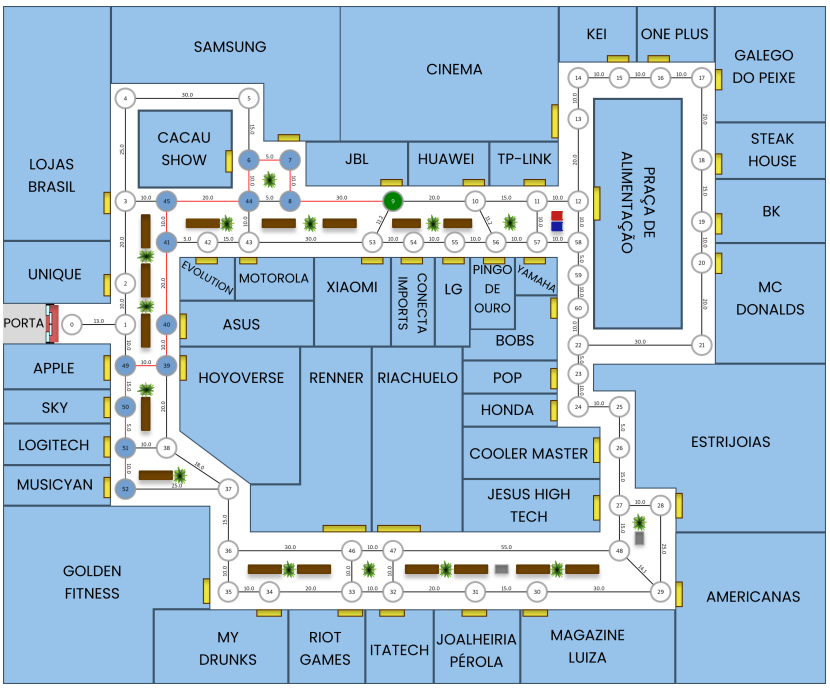


Figura 18: Percurso em $A*$ - Orig.: *Musicyan* / Viés: *Cacau Show* / Dest.: *JBL*.

No primeiro experimento de percurso com viés, utilizamos a tabela $Q_{rot_{A*}}$ (Figura 18), obtendo uma distância total percorrida de 155 metros ao passar por 12 pontos no ambiente. Por outro lado, na ausência de viés, o algoritmo $A*$ responsável pela tabela $Q_{rot_{A*}}$ também se destaca, percorrendo uma distância total de 131.2 metros em 9 passos.

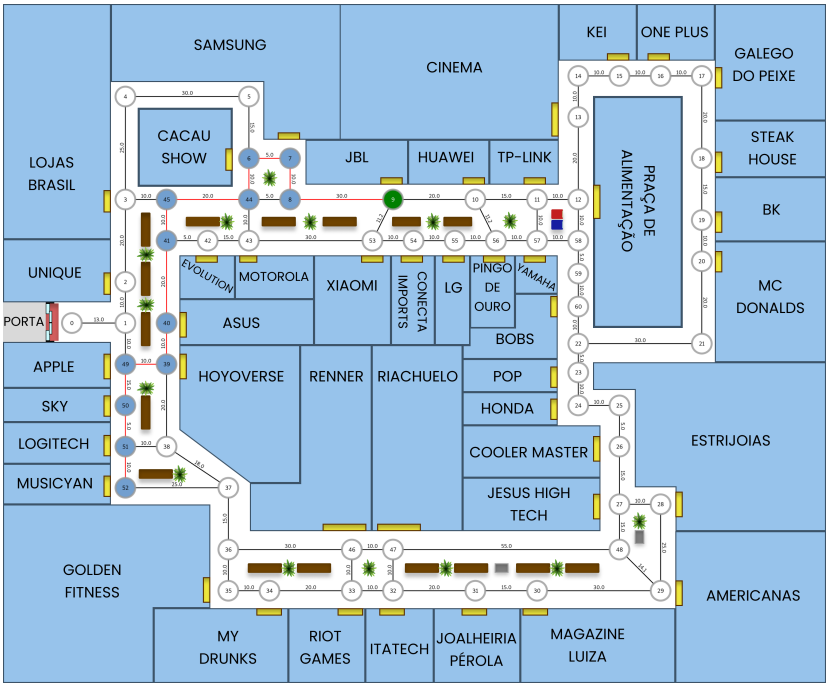


Figura 19: Percurso em Largura - Orig.: *Musicyan* / Viés: *Cacau Show* / Dest.: *JBL*.

No segundo experimento de percurso com viés, utilizando a tabela $Q_{rot_{Largura}}$ com viés, conforme ilustrado na Figura 19, o trajeto foi gerado percorrendo uma distância total de 178 metros. Com a introdução do viés, é possível observar que a rota é modificada para primeiro alcançar o ponto de interesse e, em seguida, prosseguir em direção ao destino.

Isso difere do caminho gerado na abordagem sem viés, conforme demonstrado na figura 14.

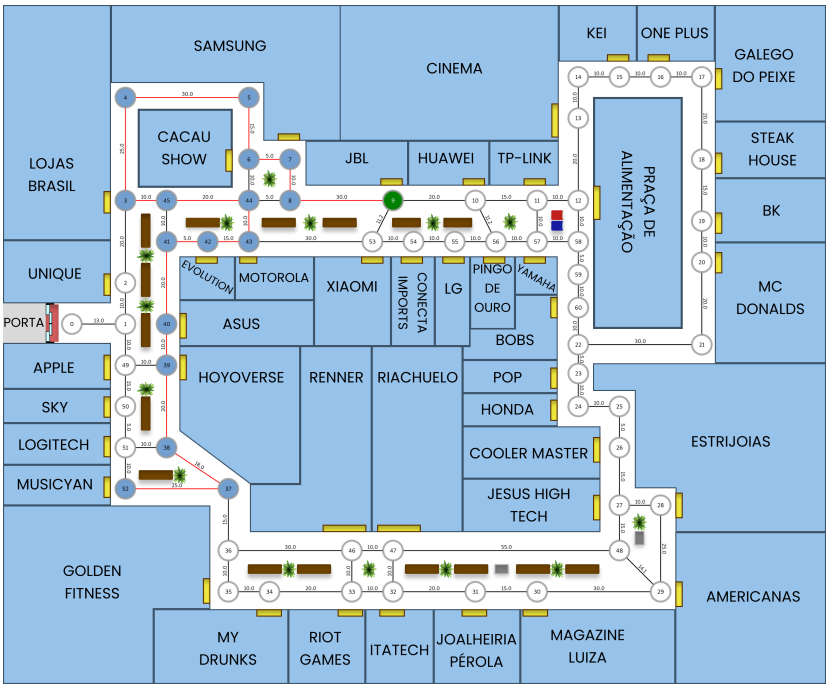


Figura 20: Percurso em Profundidade - Orig.: *Musicyan* / Viés: *Cacau Show* / Dest.: *JBL*.

No terceiro experimento de percurso com viés, ao utilizar a tabela $Q_{rot_{Profundidade}}$, conforme ilustrado na Figura 20, a rota permaneceu inalterada em ambas as circunstâncias: com enviesamento e sem enviesamento. A distância total percorrida foi de 268 metros, abrangendo 16 pontos no ambiente. Este resultado se deve ao fato de que, ao traçar a rota sem enviesamento, o algoritmo já passava por um estabelecimento alimentício.

É relevante ressaltar que a distância percorrida não variou entre as situações com e sem enviesamento, o que indica que o algoritmo de busca em profundidade não foi significativamente influenciado pelas informações de viés neste cenário.

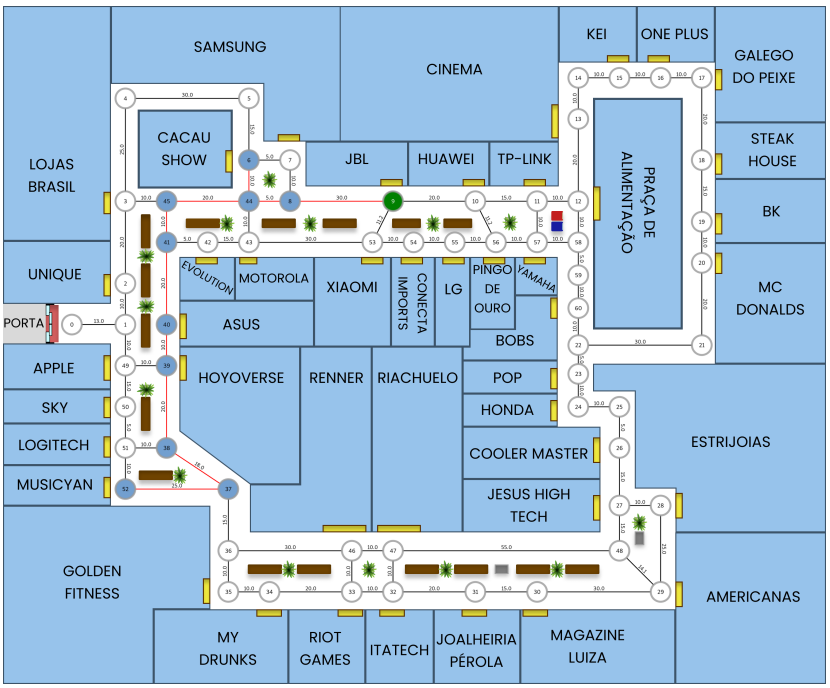


Figura 21: Percurso em *Q-Learning* - Orig.: *Musicyan* / Viés: *Cacau Show* / Dest.: *JBL*.

No quarto experimento de percurso com viés, foi utilizada a tabela $Q_{rot_{QLearning}}$. O trajeto gerado com viés apresentou uma distância total percorrida de 178 metros, passando por 11 pontos. Ao compararmos com o experimento de percurso enviesado anteriormente, utilizando a tabela $Q_{rot_{Largura}}$, obtivemos resultados de distância percorrida e número de passos iguais. No entanto, vale notar que os pontos percorridos diferem no ambiente, como pode ser observado na Figura 21.

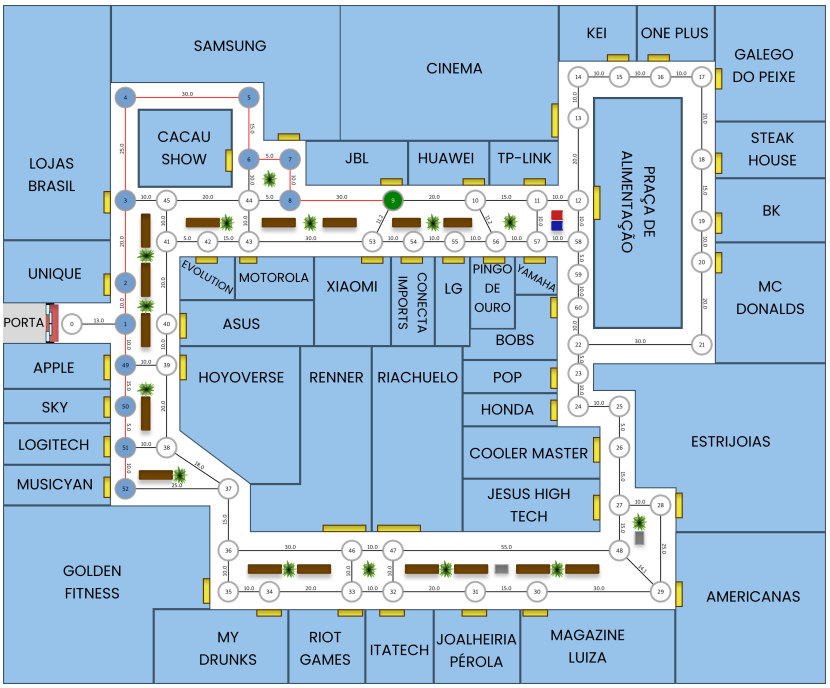


Figura 22: Percurso em *Sarsa* - Orig.: *Musicyan* / Viés: *Cacau Show* / Dest.: *JBL*.

No quinto experimento de percurso com viés (Figura 22), foi utilizada a tabela $Q_{rot_{Sarsa}}$, resultando em uma distância percorrida de 185 metros e a realização de 12 passos no ambiente. Essa distância e quantidade de passos superaram os valores obtidos no experimento

anterior, gerado através da tabela $Q_{rot_{QLearning}}$.

Os experimentos realizados com a introdução de viés em direção a uma loja de interesse apresentaram resultados, nos quais, as informações adicionais resultaram em trajetos distintos, quando comparados com os experimentos realizados sem viés.

O algoritmo A*, a partir do qual é gerada a tabela $Q_{rot_{A^*}}$, demonstrou a capacidade de encontrar um caminho ótimo ao combinar heurística e pesos locais, revelando-se suscetível ao viés introduzido.

No caso da busca em largura, que resulta na tabela $Q_{rot_{Largura}}$, o viés provocou uma alteração no trajeto quando comparado ao experimento sem o viés. Como discutido em tópicos anteriores, a abordagem do algoritmo de largura visa expandir o menor número de nós, o que nem sempre resulta em uma solução ótima.

Os experimentos realizados pelas tabelas $Q_{rot_{QLearning}}$ e $Q_{rot_{Sarsa}}$, novamente demonstraram conhecimento do ambiente e flexibilidade na adaptação às informações de viés. Ao compará-los, observamos que o resultado gerado pelo $Q_{rot_{QLearning}}$ com viés apresentou um desempenho melhor do que o $Q_{rot_{Sarsa}}$ em termos da distância percorrida, em comparação com a situação em que o viés não foi utilizado, onde ocorreu o contrário.

No experimento utilizando a tabela $Q_{rot_{Profundidade}}$, o viés não causou impactos na geração do caminho, uma vez que, ao percorrer até o viés, ela seguiria o mesmo caminho que utilizaria para chegar ao objetivo.

No contexto geral, fica claro que a adição de informações direcionais, como o viés, influencia a tomada de decisão dos algoritmos, ajustando o trajeto para alcançar estabelecimentos de interesse, como demonstrado neste experimento.

4.2 Experimentos em *batch*

Nos testes, foram gerados origem e destino aleatórios, no qual os algoritmos foram iterados: 10, 100, 1.000, 10.000, 100.000 e 1.000.000. Nesse contexto, o objetivo é determinar a eficácia de cada algoritmo na menor distancia percorrida, considerando variações no número de épocas ou iterações. Os resultados apresentados em diferente épocas fornecem uma visão de evolução do algoritmos á medida que as iterações aumentam, permitindo a comparação em diferentes cenários.

A seguir, apresenta-se um código de simulação responsável por gerar trajetórias aleatórias e realizar iterações múltiplas. O objetivo é obter uma compreensão mais clara de quais algoritmos acharam os melhores caminhos através da distancia percorrida. O código também gera um arquivo `CSV` contendo as distâncias totais percorridas por cada algoritmo nas trajetórias geradas aleatoriamente.

```
def get_distance(algorithm, origin, destiny):
    distance = 0
    if algorithm in ["QLearning", "Sarsa"]:
        _, distance = get_path_rl(origin, destiny, algorithm)
    elif algorithm in ["Astar", "Largura", "Profundidade"]:
        _, distance = get_path_search(origin, destiny, algorithm)
    return distance

def batch_run(epochs):
    algorithms = ["QLearning", "Sarsa", "Astar", "Largura", "Profundidade"]
    total_distances = {algorithm: 0 for algorithm in algorithms}

    for i in range(epochs):
        origin, destiny = generate_random_origin_destiny()
```

```
15
16          for algorithm in algorithms:
17              distance = get_distance(algorithm, origin, destiny)
18              total_distances[algorithm] += distance
19
20      save_results(total_distances, epochs)
21
22  def generate_random_origin_destiny():
23      vertices = list(vertices_dict.keys())
24      origin = random.choice(vertices)
25      destiny = random.choice(vertices)
26
27      while origin == destiny:
28          destiny = random.choice(vertices)
29
30      return int(origin), int(destiny)
31
32  def save_results(total_distances, epochs):
33      with open(f"batch/total_results_{epochs}.csv", "w", encoding="utf-8-sig") as _file:
34          for algorithm, distance in total_distances.items():
35              _file.write(f"{algorithm}, {distance}\n")
36
37  if __name__ == "__main__":
38      epochs = 1000000
39      batch_run(epochs)
```

Algoritmo	Distância percorrida
Q-Learning	1128.2
A*	1128.2
Sarsa	1135.2
Largura	1135.2
Profundidade	5318.0

Tabela 5: Resultados dos algoritmos para 10 iterações.

Na Tabela 5, foram submetidos a 10 iterações. O algoritmo *Q-Learning* empatou com o A*, ambos gerando soluções ótimas com uma distância percorrida de $1128,2$ metros. O algoritmo *Sarsa*, que possui semelhanças com o *Q-Learning*, obteve um resultado um pouco inferior, com $1135,2$ metros percorridos. O método de Busca em Largura, que visa percorrer a menor quantidade de vértices possível, também alcançou um resultado idêntico ao *Sarsa*, com $1135,2$ metros percorridos. Por outro lado, o algoritmo de Profundidade sugere que pode ter explorado níveis excessivamente profundos entre os nós das buscas, resultando em um desempenho desfavorável em comparação aos outros algoritmos, com uma distância percorrida de 5318 metros.

Algoritmo	Distância percorrida
A*	11660.9
Largura	12065.4
Sarsa	12168.5
Q-Learning	12360.5
Profundidade	39250.0

Tabela 6: Resultados dos algoritmos para 100 iterações.

Na Tabela 6, com 100 iterações, o algoritmo A* mais uma vez apresentou um resultado excelente, indicando sua contínua eficácia em encontrar o menor caminho, mesmo com o aumento das iterações, tendo percorrido uma distância de 11.660, 9 metros.

Neste experimento, os resultados também revelaram que o método de Busca em Largura foi capaz de identificar trajetórias mais curtas entre a origem e o destino, resultando em uma distância percorrida de 12.065,4 metros. Por sua vez, o algoritmo *Sarsa* demonstrou que o agente aprendeu bem o ambiente, mas houve uma ligeira piora em relação ao primeiro teste, com uma distância percorrida de 12.168,5 metros. O *Q-Learning* também mostrou a habilidade do agente em encontrar caminhos eficazes, resultando em uma distância de 12.369,5 metros. Entretanto, o algoritmo de Profundidade teve o pior desempenho dentre os algoritmos, explorando caminhos mais longos, o que resultou em uma distância percorrida de 39.250,0 metros. Isso reforça a tendência desse método a buscar soluções mais extensas em relação aos demais algoritmos.

Algoritmo	Distância percorrida
A*	112922.2
Largura	117961.2
Sarsa	118572.2
Q-Learning	120064.0
Profundidade	382057.8

Tabela 7: Resultados dos algoritmos para 1.000 iterações.

Na Tabela 7, o algoritmo *A** percorreu uma distância de 112.922, 2 metros ao longo de 1000 épocas. Esse resultado evidencia a contínua habilidade do algoritmo A* em encontrar caminhos otimizados ou de menor custo. O método de Busca em Largura percorreu uma distância de 117.691, 2 metros, reafirmando sua eficácia ao identificar as trajetórias mais curtas entre o ponto de origem e destino. O algoritmo *Sarsa* percorreu uma distância de 118.572, 2 unidades em 1.000 épocas, demonstrando que o agente continuou a aprender e aprimorar sua trajetória ao longo do tempo. De maneira semelhante, o algoritmo *Q-Learning* também apresentou esse comportamento, indicando que o agente adquiriu conhecimento sobre o ambiente e persistiu em encontrar distâncias mais curtas. Em contraste, o método de Profundidade percorreu uma distância de 382.067,8 metros, revelando uma tendência a explorar soluções mais longas em relação aos demais algoritmos.

Algoritmo	Distância percorrida
A*	1131415.2
Largura	1184169.6
Sarsa	1188926.3
Q-Learning	1206861.3
Profundidade	3825727.7

Tabela 8: Resultados dos algoritmos para 10.000 iterações.

Na Tabela 8, o algoritmo *A** percorreu uma distância de 1.131.415, 2 metros, reafirmando sua capacidade de encontrar soluções ótimas mesmo com um aumento no número de iterações. O método de busca em largura percorreu uma distância de 1.184.169, 6 metros, demonstrando sua habilidade de encontrar trajetórias curtas entre dois pontos mesmo com o aumento das iterações. O algoritmo *Sarsa* percorreu uma distância de 1.188.926, 3 metros, evidenciando sua eficácia contínua mesmo quando o número de iterações cresce. O *Q-Learning* percorreu uma distância de 1.206.861, 3 metros em 10.000

iterações, indicando que o processo de aprendizado do agente continua eficaz. Por outro lado, o algoritmo de Profundidade percorreu uma distância de 3.825.727, 7 metros em 10.000 iterações, mantendo uma discrepância significativa em relação aos outros algoritmos e reforçando sua tendência a explorar soluções mais longas.

Algoritmo	Distância percorrida
A*	11249987.2
Largura	11754256.9
Sarsa	11806972.1
Q-Learning	11973435.0
Profundidade	38390326.3

Tabela 9: Resultados dos algoritmos para 100.000 iterações.

Na Tabela 9, o algoritmo $A*$ percorreu uma distância de 11.249.987, 2 metros em 100.000 épocas. Isso sugere que, mesmo com um número significativamente maior de iterações, ele ainda é capaz de encontrar soluções ótimas. A Busca em Largura percorreu 11.754.256, 9 metros em 100.000 iterações, mantendo a capacidade de encontrar soluções de baixo custo mesmo com um alto número de iterações. O algoritmo *Sarsa* percorreu uma distância de 11.806.972, 1 unidades em 100.000 épocas, indicando que o processo de aprendizado do agente continua eficaz. O método *Q-Learning* percorreu 11.973.435 metros em 100.000 iterações, demonstrando que o agente continua eficiente em encontrar soluções com distâncias reduzidas no ambiente. Por outro lado, o algoritmo de Profundidade percorreu 38.390.326, 3 metros em 100.000 iterações, destacando-se ao explorar soluções mais extensas e, por essa razão, apresentando resultados consideravelmente diferentes dos demais.

Algoritmo	Distância percorrida
A*	112507637.1
Largura	117480611.1
Sarsa	118025939.4
Q-Learning	119697808.0
Profundidade	383693821.4

Tabela 10: Resultados dos algoritmos para 1.000.000 iterações.

Na Tabela 10, o algoritmo $A*$ percorreu uma distância de 112.507.637, 1 unidades em 1.000.000 de iterações. Isso demonstra que o algoritmo continua eficaz na busca de caminhos mais curtos no ambiente. A Busca em Largura percorreu 117.480.611, 1 metros em 1.000.000 de iterações, sendo capaz de identificar trajetórias diversas de maneira eficiente. O algoritmo *Sarsa* percorreu 118.025.939, 4 metros em 1.000.000 de iterações, evidenciando sua capacidade contínua de otimizar trajetórias no ambiente. O algoritmo *Q-Learning* percorreu uma distância de 119.697.808 unidades em 1.000.000 de iterações, sugerindo que o agente continua a aprender e a aprimorar suas trajetórias. Por fim, o método de Profundidade percorreu 383.693.821, 4 metros em 1.000.000 de iterações. A distância percorrida por este algoritmo continua sendo a maior, o que indica sua tendência a encontrar caminhos mais extensos.

De acordo com os experimentos realizados, os algoritmos de buscas e aprendizados por reforços continuam se adaptando e otimizando as trajetórias a medida que as iterações aumentam. Os algoritmos $A*$, Sarsa, Largura e Q-*Learning* demonstraram a capacidade de encontrar caminhos mais otimizados, enquanto o de profundidade continuou explorando trajetórias mais longas.

5 Conclusões

Com base no plano de trabalho desenvolvido, o projeto de pesquisa atingiu seus objetivos de maneira satisfatória. Foi proposto um modelo inteligente que auxilia na navegação offline em ambientes internos, considerando um ambiente previamente mapeado e a disponibilidade das localizações intrínsecas de pontos de partida e destino.

O projeto incorporou conceitos de Inteligência Artificial (IA) para lidar com os desafios da navegação. Foram empregados métodos de busca cega, heurísticas e aprendizado por reforço para controlar as trajetórias dentro do ambiente. Além disso, o modelo foi desenvolvido com a capacidade de ajuste de pesos e critérios locais e heurísticos.

Os resultados obtidos foram avaliados de forma positiva. As rotas geradas para a navegação *offline* em ambientes internos mostraram-se eficazes e atenderam às expectativas estabelecidas. Ao comparar os resultados dos algoritmos de geração de rotas, ficou evidente que o modelo proposto alcançou sucesso na realização da navegação interna.

Em síntese, o projeto foi concluído com sucesso, cumprindo os objetivos propostos e demonstrando a viabilidade do modelo inteligente para a navegação *offline* em ambientes internos.

6 Perspectivas

Conforme abordado nas seções anteriores, é evidente que os alvos estabelecidos neste plano de ação foram atingidos. No entanto, investigações futuras relacionadas a este projeto podem explorar alternativas adicionais para refinamentos ligados à criação de trajetos em ambientes internos.

Adicionalmente, é possível realizar uma avaliação referente à utilização de abordagens diversas no contexto do aprendizado por reforço enviesado, comparando-as aos resultados previamente alcançados. Como exemplificado no estudo mencionado por Oliveira (2021), o qual aborda algoritmos de aprendizado por reforço com múltiplos objetivos. Incorporar as abordagens propostas nesse trabalho seria altamente benéfico para aprimorar os algoritmos que percorrem os pontos de interesse destacados no âmbito deste projeto.

Outro ponto importante a mencionar é a possibilidade de transferir a implementação do servidor diretamente para o ambiente móvel. Dessa forma, seria evitada a necessidade de uma *API* para gerar trajetos, e o próprio dispositivo móvel poderia criar os trajetos usando as informações aprendidas nas tabelas.

7 Referências Bibliográficas

Referências

Alfakih, T., Hassan, M. M., Gumaei, A., Savaglio, C., and Fortino, G. (2020). Task offloading and resource allocation for mobile edge computing by deep reinforcement learning based on sarsa. IEEE Access, 8:54074–54084.

Anderson, C. (2015). Docker [software engineering]. IEEE Software, 32(3):102–c3.

Biehl, M. (2015). API Architecture. API-University Series. CreateSpace Independent Publishing Platform.

Bisdikian, C. (2001). An overview of the bluetooth wireless technology. IEEE Communications magazine, 39(12):86–94.

Brandes, U., Freeman, L. C., and Wagner, D. (2013). Social networks.

Carvalho Júnior, O. A. d., Couto Júnior, A. F., Silva, N. C. d., Martins, É. d. S., Carvalho, A. P. F. d., and Gomes, R. A. T. (2009). Avaliação dos classificadores espectrais de mínima distância euclidiana e spectral correlation mapper em séries temporais ndvi-modis no campo de instrução militar de formosa (go).

Chelly, M. and Samama, N. (2009). New techniques for indoor positioning, combining deterministic and estimation methods. In ENC-GNSS 2009 : European Navigation Conference - Global Navigation Satellite Systems, pages 1–12, Naples, Italy.

Commons, W. (2023). File:manhattan distance.svg — wikimedia commons, the free media repository. [Online; accessed 18-agosto-2023].

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Algoritmos: Teoria e Prática. Elsevier, Rio de Janeiro, 3 edition.

Crockford, D. (2006). The application/json media type for javascript object notation (json).

Dann, C., Mansour, Y., Mohri, M., Sekhari, A., and Sridharan, K. (2022). Guarantees for epsilon-greedy reinforcement learning with function approximation. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S., editors, Proceedings of the 39th International Conference on Machine Learning, volume 162 of Proceedings of Machine Learning Research, pages 4666–4689. PMLR.

Dart Team (Último acesso em 2023). Dart documentation. `https://dart.dev/guides`. Acesso em 13 de julho de 2023.

El-Sheimy, N. and Li, Y. (2021). Indoor navigation: state of the art and future trends. Satellite Navigation, 2:7.

Fagundes, L. P. (2008). Técnicas de localização de dispositivos móveis em redes wifi-tdoa.

Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., and Gunawan, E. (2021). A systematic literature review of a* pathfinding. Procedia Computer Science, 179:507–514.

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). Data Structures and Algorithms in Java, 6th Edition. Wiley, United States.

Google (Último acesso em 2023). Flutter documentation. `https://docs.flutter.dev`. Acesso em 13 de julho de 2023.

GraphOnline (2015). Graph online.

Hasegawa, J. K., Galo, M., Monico, J. F. G., and Imai, N. N. (1999). Sistema de localização e navegação apoiado por gps. In Congresso Brasileiro de Cartografia, volume 19.

Holzmann, G. J., Peled, D. A., and Yannakakis, M. (1996). On nested depth first search. The Spin Verification System, 32:23–32.

Jeske, T. (2013). Floating car data from smartphones: What google and waze know about you and how hackers can control traffic. Proc. of the BlackHat Europe, pages 1–12.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. Journal of artificial intelligence research, 4:237–285.

Leppakoski, H., Collin, J., and Takala, J. (2013). Pedestrian navigation based on inertial sensors, indoor map, and wlan signals. Journal of Signal Processing Systems, 71(3):287–296.

Liu, X., Makino, H., and Mase, K. (2010). Improved indoor location estimation using fluorescent light communication system with a nine-channel receiver. IEICE Transactions on Communications, E93-B(11):2936–2944.

Lucidchart (2023). O que e planta baixa. https://www.lucidchart.com/pages/pt/o-que-e-planta-baixa.

Marciano, C., Souza, F., and Souza, R. (2018). Analise da rede gnutella utilizando redes par-a-par e teoria dos grafos. https://www.gta.ufrj.br/ensino/eel878/redes1-2018-1/trabalhos-vf/p2p/grafos.html [Acesso em: 18/02/2023].

Mazuelas, S. and et al. (2009). Robust indoor positioning provided by real-time rssi values in unmodified wlan networks. IEEE Journal of Selected Topics in Signal Processing, 3(5):821–831.

Mehta, H., Kanani, P., and Lande, P. (2019). Google maps. International Journal of Computer Applications, 178(8):41–46.

Mengual, L. et al. (2013). Multi-agent location system in wireless networks. Expert systems with applications, 40(6):2244–2262.

Oliveira, T. H. F. d. (2021). Algoritmos de aprendizagem por reforço para problemas de otimização multiobjetivo. PhD thesis, Universidade Federal do Rio Grande do Norte.

Peixoto, T. (2023a). Example network. https://graph-tool.skewed.de/static/doc/search_module.html. Acessado em 12 de julho de 2023.

Peixoto, T. (2023b). A simple directed graph with two vertices and one edge. https://graphtool.skewed.de/static/doc/quickstart.html. Acessado em 8 de julho de 2023.

Peixoto, T. P. (2014). The graph-tool python library. figshare.

Permana, S. H., Bintoro, K. Y., Arifitama, B., and Syahputra, A. (2018). Comparative analysis of pathfinding algorithms a*, dijkstra, and bfs on maze runner game. IJISTECH (International J. Inf. Syst. Technol., vol. 1, no. 2, p. 1.

Ramírez, S. (2023). Fastapi documentation. https://fastapi.tiangolo.com/. Acessado em 11 de julho de 2023.

React Native Community (Último acesso em 2023). React native documentation. https://reactnative.dev. Acesso em 13 de julho de 2023.

Strauss, T. and von Maltitz, M. J. (2017). Generalising ward's method for use with manhattan distances. PloS one, 12(1):e0168288.

Stuart Russell, P. N. (2003). Artificial Intelligence: A Modern Approach. Prentice Hall, 2nd edition.

Taddeo, L., Alves, P., and Sobrinho, F. (2018). Navegação indoor–um estudo de caso. In Anais da VI Escola Regional de Informática de Goiás, pages 159–172, Porto Alegre, RS, Brasil. SBC.

Van Rossum, G. and Drake Jr, F. L. (1995). Python reference manual. Centrum voor Wiskunde en Informatica Amsterdam.

Watkins, C. J. and Dayan, P. (1992). Q-learning. Machine learning, 8:279–292.

Winder, P. (2020). Reinforcement learning. O'Reilly Media.

Yao, W., Chu, C.-H., and Li, Z. (2010). The use of rfid in healthcare: Benefits and barriers. In 2010 IEEE International Conference on RFID-Technology and Applications, pages 128–134. IEEE.

8 Outras atividades

A próxima seção trata-se de atividades desempenhadas ao decorrer do projeto com intuito de aprimorar ou capacitar habilidades e técnicas essências para a realização de ações do presente trabalho.

8.1 Atividades Complementares

Com o objetivo de adquirir conhecimento necessário para executar as atividades que foram destinadas ao atual plano de trabalho, foi necessário realizar o estudo de conteúdos mais específicos que estavam ligados ao tema, como: algoritmos de aprendizado por reforço, programação em *Python* e *Dart*, comunicação de sistemas por meio de *APIs*, criação e utilização de contêineres com o *Docker*, Grafos como estrutura de dados, além de normas e conceitos utilizados em trabalhos científicos.