
Universidade Federal
do Rio de Janeiro
Escola Politécnica

# ESTUDO DE ALGORITMOS PARA REDES DE TRANSPORTE

Rodrigo Rodrigues Paim

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Daniel Ratton Figueiredo

Rio de Janeiro
Janeiro de 2015

ESTUDO DE ALGORITMOS PARA REDES DE TRANSPORTE

Rodrigo Rodrigues Paim

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO.

Examinado por:

Prof. Daniel Ratton Figueiredo, Ph.D.

Prof. Márcia Rosana Cerioli, D.Sc.

Prof. Celina Miraglia Herrera de Figueiredo, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
JANEIRO DE 2015

Rodrigues Paim, Rodrigo

Estudo de Algoritmos para Redes de Transporte/Rodrigo Rodrigues Paim. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2015.

IX, 53 p. 29, 7cm.

Orientador: Daniel Ratton Figueiredo

Projeto de Graduação – UFRJ/ Escola Politécnica/ Curso de Engenharia de Computação e Informação, 2015.

Referências Bibliográficas: p. 51 – 53.

I. Ratton Figueiredo, Daniel. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Computação e Informação. III. Título.

*Aos meus pais*

# Agradecimentos

Agradeço à minha mãe e ao meu pai pelo amor incondicional que têm por mim, à minha irmã e meus avós pelo suporte que sempre me deram em todas as etapas da minha vida. Agradeço à minha família por sempre estar presente e por me ajudar a construir o que sou hoje.

Não citarei nomes para não correr o risco de esquecer de alguém e não estabelecer uma ordem, o que seria impossível. Agradeço aos meus amigos do CEFET por estarem do meu lado há 9 anos. Aos meus amigos da UFRJ por esses anos de trabalho e companheirismo. Aos amigos que fiz na ENSTA por estarem ao meu lado enquanto vivia sozinho em outro país.

Agradeço a todos os meus professores pelo tempo e dedicação a esta profissão. Em especial, ao meu orientador por esses anos de trabalho que foram muito importantes para meu desenvolvimento como pessoa e profissional, além de aceitar me acompanhar neste projeto.

Agradeço à Amadeus e a todos com quem trabalhei; este trabalho de fim de curso é motivado por tudo o que aprendi lá enquanto era estagiário. Agradeço finalmente a todos com quem trabalhei na EPFL, com quem estudei no CNAM e a todos que não se enquadram em nenhum critério de agradecimento acima, mas fazem ou fizeram parte da minha vida.

*"No man is a failure who has friends"*
It's a Wonderful Life (1946)

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação.

ESTUDO DE ALGORITMOS PARA REDES DE TRANSPORTE

Rodrigo Rodrigues Paim

Janeiro/2015

Orientador: Daniel Ratton Figueiredo

Curso: Engenharia de Computação e Informação

O problema do caminho mínimo tem sido alvo de estudo em Teoria dos Grafos há muito tempo. Com o passar dos anos, as necessidades foram mudando e o problema original readaptado. Atualmente, com a popularização de softwares para se orientar em grandes cidades e a proliferação de aparelhos com GPS, servidores precisam calcular milhares ou mesmo milhões de rotas simultaneamente e em pouco tempo. Isto torna algoritmos clássicos como Dijkstra inviáveis, pois eles devem explorar uma grande parte da rede que em nada contribuirá para o resultado. Alternativas vêm surgindo há aproximadamente duas décadas sob a forma de melhorias a algoritmos clássicos ou com a criação de novos métodos de resolução.

Este trabalho estuda técnicas que aumentam o desempenho do cálculo de rotas para redes rodoviárias, além de abordar o problema do caminho mais rápido aplicado a redes de transporte público. Em vista da forte dependência temporal deste último, Dijkstra não pode ser usado de forma eficiente, o que abre espaço para a definição de novos algoritmos, como Transfer Patterns, RAPTOR e CSA. Estes dois serão estudados a fundo, enquanto uma breve descrição do primeiro será feita. Em seguida, um protótipo usando dois dois algoritmos distintos para a resolução de consultas será apresentado e uma interface web amigável proposta. Finalmente, resultados experimentais serão exibidos junto com uma análise de desempenho e uma visão geral do futuro da área.

*Palavras-chave:* Redes de Transporte, Transporte Público, Caminho Mínimo.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Engineer.

ON ALGORITHMS FOR TRANSPORTATION NETWORKS

Rodrigo Rodrigues Paim

January/2015

Advisor: Daniel Ratton Figueiredo

Course: Computer Engineering

The shortest-path problem has been studied for a long time in Graph Theory. It changed over the past years due to new formulations and is currently famous for finding paths in big cities helped by devices with GPS. However, this increased enormously servers' workload, since thousands or even millions of people now request to solve this problem simultaneously. This makes classic algorithms, such as Dijkstra, impractical because they have to explore a great part of the network (much of which useless for the result) before providing an answer. Some alternatives have appeared in the last two decades, improving previous methods or defining brand new ways of solving the problem.

In this work, we study tools to improve performance of path calculation in road networks and also visit the analogous problem for public transportation networks. Due to the strong time dependency of the latter, Dijkstra can't be applied in an efficient way. New methods such as Transfer Patterns, RAPTOR and CSA were developed to tackle this weakness. The latter two will be explored in details, while the former will be only briefly discussed. Thus, a prototype capable of answering queries with two different algorithms is presented and a friendly web interface proposed. Finally, we show experimental results along with performance analysis and discuss the state of the art on this subject.

*Keywords:* Transport Networks, Transit Networks, Shortest-Path Algorithms.

# Sumário

# Capítulo 1

# Introdução

Buscar caminhos mais curtos entre dois pontos de uma rede é um problema fundamental para diversas aplicações, como roteamento na Internet e caminhos entre usuários em redes sociais. Talvez a mais conhecida seja encontrar rotas em redes de transporte, usando seu veículo particular ou transporte público: uma pessoa procura se deslocar rapidamente entre dois pontos usando a estrutura viária (respectivamente linhas de transporte).

Algoritmos clássicos como o de Dijkstra são pouco úteis para redes de transporte, pois estas apresentam número de vértices e arestas da ordem de dezenas ou mesmo centenas de milhões, o que eleva consideravelmente o tempo de consulta para alguns poucos segundos. Em ambientes com milhares ou milhões de usuários simultâneos, isto se traduz em espera e congestionamento dos servidores. Diversas alternativas surgiram nos últimos anos, sendo capazes de reduzir o cálculo de caminhos mínimos a alguns microssegundos para redes de rodovias e de poucos milissegundos para redes de transporte público [9]. Estes algoritmos são implementados em softwares como Google Maps, Bing Maps, além de usado numa infinidade de aplicações para dispositivos móveis.

Outro ponto importante das redes de transporte é a dependência temporal inerente a elas: ruas e avenidas possuem diferentes níveis de congestionamento ao longo do dia. Este mesmo problema se traduz no transporte público no fato de linhas de ônibus terem instantes exatos de passagem pelos pontos de parada, além

de às vezes não funcionarem durante a noite ou finais de semana. Adicionar uma variável temporal torna estes problemas muito mais difíceis e, consequentemente, mais interessantes de ser tratados: é necessário definir novos modelos de grafo para representar as redes, além de utilizar outros tipos de algoritmo.

Este trabalho visa a estudar alguns métodos que podem ser usados para resolver o problema do caminho mínimo e derivados em redes de transporte. Modelos e algoritmos serão apresentados em conjunto com um protótipo para verificar o funcionamento de alguns deles. Além disso, ele objetiva criar uma maior familiaridade com o tema e apresentar referências bibliográficas relevantes para leitores mais interessados no assunto.

## Trabalhos Relacionados

Redes de transporte rodoviário são estudadas há algum tempo, mas têm recebido grande contribuição nos últimos anos. Algoritmos como Arc Flags [27, 28], Contraction Hierarchies [21, 24, 25], além de variantes de algoritmos clássicos, como ALT [29] surgiram há menos de uma década e trouxeram grandes melhorias no cálculo de rotas. Atualmente, são usados em conjunto com os melhores algoritmos disponíveis: Hub-Label Compression [1, 18] e Transit Nodes Routing [10].

Diferentemente do modelo anterior, Redes de transporte público surgiram há pouco tempo e têm despertado grande atenção [3–5, 16, 22, 31]. Tentativas de aplicar algoritmos para redes rodoviárias foram feitas [11, 14, 15, 23] com pouco sucesso. Assim, novos algoritmos foram definidos: Transfer Patterns [6, 8], RAPTOR [19] e CSA [20, 33]. Como será visto, existem diversos critérios que podem ser otimizados em consultas neste tipo de rede, criando novas possibilidades de pesquisa [2, 7, 12, 13, 17, 26, 32].

## Organização do Texto

No Capítulo 2 serão apresentadas duas técnicas que podem ser aplicadas para rodovias e grafos mais gerais, *Arc Flags* e *Contraction Hierarchies*. Mesmo podendo

acelerar consultas para redes de transporte público, elas não são em geral empregadas. Algoritmos para este último tipo de rede serão apresentados no Capítulo 3. Bastante recentes, *Transfer Patterns* (2010), *RAPTOR* (2012) e *CSA* (2012) são o estado da arte no que diz respeito a transporte público. O Capítulo 4 discorre sobre o processo de desenvolvimento de um software para o cálculo de rotas neste último tipo de rede usando Dijkstra e um dos algoritmos abordados. Uma interface web será utilizada, de modo que o usuário possa selecionar pontos no mapa e verificar o caminho empregado, dado um determinado tempo de partida da origem. Além disso, são exibidos resultados comparando o desempenho das duas abordagens para a solução do problema. Por fim, no Capítulo 5 é feito um apanhado geral e discussão sobre melhorias que podem ser introduzidas no padrão de dados usado para redes de transporte público.

# Capítulo 2

# Redes de Transporte

Pensar em Redes de Transporte é equivalente a pensar em grafos. Porém, existem diversas formas diferentes de modelá-las. Neste capítulo, serão consideradas apenas as Redes de Transporte Privadas (malha viária) sem dependência temporal, ou seja, tais que o custo de atravessar ruas e rodovias se mantém constante ao longo do tempo. É evidente que este modelo não representa perfeitamente a realidade, mas foge do escopo deste estudo introdutório apresentar alternativas que levem em conta trânsito e outros fatores ligados ao transporte de veículos privados.

O modelo mais simples de uma Rede de Transporte é um grafo $G = (V, E)$ tal que os vértices (ou nós) representam cruzamentos de vias e arestas mapeiam o fluxo entre dois cruzamentos: existe uma aresta $(u, v)$ caso seja possível ir de $u$ para $v$. Este grafo é direcionado, levando em conta o sentido em que o fluxo de veículos ocorre. A Figura 2.1 mostra um exemplo de modelagem de um pedaço da rede viária, com os vértices sendo representados pelos círculos vermelhos e as arestas por setas azuis. Em Teoria dos Grafos, existe uma diferença entre arestas e arcos: as primeiras são usadas em grafos sem direção, enquanto os outros aparecem na variante direcionada. Contudo, por abuso de notação, neste trabalho o termo aresta será usado para os dois casos; caso haja confusão, haverá uma identificação sobre o direcionamento ou não.

Esta modelagem é, num primeiro momento, contra-intuitiva, pois arestas em geral fazem o papel de ligações, assim como ruas. Esta representação pode ser

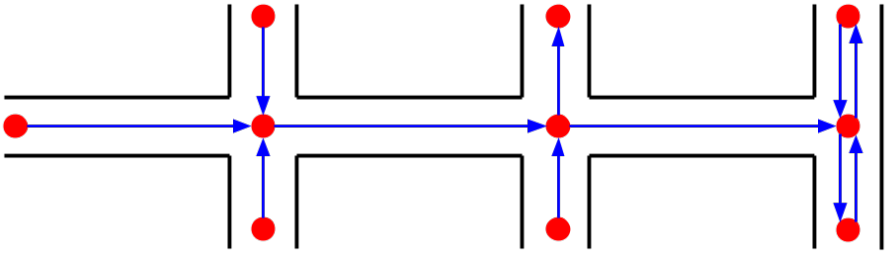


Figura 2.1: Exemplo da modelagem de ruas usando grafo

pensada como um tipo de forma dual (ou grafo linha). Modelos que usam o "grafo primal" fogem ao conhecimento do autor, mas podem existir. De todo modo, este modelo permite que algoritmos clássicos sejam usados de forma natural, além de abrir espaço para o surgimento de alternativas que explorem sua estrutura. Os grafos criados por rede de transporte são praticamente planares (a menos de algumas anomalias, como viadutos que passam por cima de rodovias), além de apresentarem baixos graus médios (cruzamentos tendem a ligar quatro pedaços de rua). Estas peculiaridades são exploradas em alguns tipos de algoritmos de busca que não serão abordados [9].

Para completar o modelo, é necessário definir uma função-peso $w : E \to \mathbb{R}^+$, que corresponde ao tempo médio de travessia entre os dois extremos da aresta. O objetivo de trabalhar com este tipo de rede é realizar buscas entre dois pontos no espaço, que podem ser aproximados pelos nós do grafo. Este problema é resolvido normalmente com a ajuda de algoritmos como Dijkstra e Bellman-Ford. Contudo, é possível aumentar, e muito, o desempenho dos mesmos através de uma etapa de pré-processamento.

A próxima seção revisita o algoritmo de Dijkstra, enquanto as duas subsequentes exploram técnicas que podem ser aplicadas a redes de transporte (e a qualquer outro tipo de grafo direcionado ponderado) para acelerar a busca. Outros problemas e algoritmos relacionados a redes de transporte podem ser encontrados em [9].

## 2.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra calcula o caminho mínimo entre uma fonte e todos os outros vértices do grafo com arestas de peso positivo. Ele classifica os vértices do grafo de acordo com três estados: *não descoberto*, *descoberto* (ou encontrado) e *explorado* (ou visitado).

Os vértices $v$ descobertos são mantidos em uma fila de prioridade de mínimo de acordo com sua distância $d(v)$ (soma dos pesos do caminho mais curto) em relação à fonte. Inicialmente, todos possuem distância infinita, com exceção da fonte, cuja distância é zero. O algoritmo trabalha de maneira gulosa e, a cada etapa, o topo desta fila é visitado (e removido da mesma). Ao visitar o vértice $u$, todas as arestas $(u, v)$ são relaxadas e a fila de prioridade atualizada com novas distâncias e novos vértices descobertos: $d(v) = \min\{d(v), d(u) + w(u, v)\}$. O algoritmo termina quando a fila de prioridade estiver vazia ou, caso exista, até a destinação ser alcançada.

A implementação original do algoritmo possui complexidade $\mathcal{O}(n^2)$, onde $n = |V|$, por causa do procedimento de encontrar o vértice descoberto mais próximo da fonte. Com a introdução de filas de prioridade, a complexidade torna-se $\mathcal{O}((m + n)\,\ell og\, n)$, onde $m = |E|$, se a implementação for feita usando heaps binários, ou $\mathcal{O}(m + n\,\ell og\, n)$ para heaps de Fibonacci.

### 2.1.1 Variante Bidirecional

É possível realizar a busca do caminho mais curto entre $s$ e $t$ por duas frentes: uma partindo de $s$ e outra de $t$ em sentido contrário. O número de vértices visitados por esta variante bidirecional é em média a metade do caso unidirecional [9], mas ainda assim continua com a mesma complexidade de pior caso.

O algoritmo mantém duas filas de prioridade e os vértices podem estar simultaneamente em ambas. A cada etapa, o vértice $u$ com a menor distância dentre os dois topos é removido, visitado e suas arestas relaxadas: caso seja a busca na ordem direta, arestas $(u, v)$ são exploradas, enquanto na ordem reversa, arestas $(v, u)$. O

algoritmo termina assim que houver um vértice $r$ visitado pelas duas: o caminho mínimo será, então, a união dos caminhos $s - r$ e $r - t$.

## 2.2 Arc Flags

Um ponto negativo importante do algoritmo de Dijkstra se encontra no fato de ele ter que explorar muitos vértices que em nada contribuem com o caminho mais curto procurado. Assim, é interessante tentar restringir o número de vértices descobertos e, mais, importante, a quantidade de visitados. Esta é exatamente a idéia por trás dos algoritmos que serão mostrados a seguir. Arc Flags [27] tenta orientar a busca de Dijkstra a seguir apenas arestas relevantes. Desta forma, o número de atualizações de distância é reduzido e consequentemente menos nós são explorados.

### 2.2.1 Funcionamento

Dado um grafo direcionado $G = (V, E)$ e uma função-peso $w : E \to \mathbb{R}^+$, sabe-se que uma aresta $e = (u, v)$ poderá fazer parte de alguns (ou nenhum) caminhos mínimos que partem de $u$. Em todo caso, é razoável pensar que nem todos os caminhos mínimos iniciados em $u$ usam $e$. Por exemplo, suponha que a rota de peso mínimo entre $u$ e $w$ não passe por $e$. Neste caso, explorar $v$ a partir desta aresta é desnecessário, além de poder induzir um custo de atualização da distância do vértice.

Uma primeira intuição para tratar este problema seria definir para cada aresta $e = (u, v)$ um conjunto de vértices $S_e$, tais que $z \in S_e$ se, e somente se, existe um caminho mínimo entre $u$ e $z$ passando por $e$. Neste caso, é natural modificar o algoritmo de Dijkstra a fim de explorar apenas arestas $a$ tais que $t \in S_a$, para uma consulta do tipo $s - t$. Contudo, esta idéia é extremamente custosa em memória, pois cada conjunto $S_e$ tem no pior caso $\mathcal{O}(n)$ elementos, totalizando um custo de armazenamento da ordem de $\mathcal{O}(mn)$. Redes de transporte público em geral apre-

sentam $m \in \mathcal{O}(n)$, mas de qualquer forma o gasto em memória continua inviável. A alternativa proposta consiste em utilizar a técnica de *arc flags*.

Arc Flags (AF) soluciona o problema de armazenamento dos conjuntos $S_e$ agrupando diversos vértices em partições (ou clusters). Seja $R = \{1, \ldots, K\}$ a identificação das partições e uma função $r : V \to R$ que associa vértices às diferentes partições. Desta forma, $S_e$ pode ser redefinido como sendo o conjunto de partições que são alcançadas por um caminho mínimo passando pela aresta $e$ com origem na cauda de $e$. Ora, a mesma consulta $s - t$ pode agora ser realizada usando apenas arestas $a$ tais que $r(t) \in S_a$. Claramente mais arestas deverão ser descobertas e, consequentemente, mais vértices explorados, mas é o preço a pagar pela redução do consumo de memória. Além disso, é fácil ver que os conjuntos $S_e$ podem ser armazenados em vetores de bits de tamanho $K$:

$$B_e = < 1, 0, 1, \ldots, 0, 1 >, \text{ onde } B_e(k) = \begin{cases} 1, & \text{se } k \in S_e \\ 0, & \text{se } k \notin S_e \end{cases} \tag{2.1}$$

Os vetores de bit $B_e$ reduzem o consumo de memória para $\mathcal{O}(mK)$ bits. Para efeito de comparação, para $m = 1.000.000$ e $K = 50$, apenas 12MB adicionais de memória devem ser usados. Além disso, a redução de espaço de armazenamento para cada aresta permite uma melhor utilização da memória cache da CPU, agilizando a parte de descoberta das arestas do algoritmo de Dijkstra.

O algoritmo por trás de AF trabalha em duas fases de pré-processamento distintas. Na primeira, o particionamento dos vértices é escolhido. Em seguida, para cada vértice $u$, os vetores $B_e$ de arestas $e = (u, v)$ incidentes a $u$ são criados a partir da árvore gerada pelo algoritmo de Dijkstra centrado em $u$. Esta operação é bastante custosa, pois é necessário rodar uma instância de Dijkstra para cada vértice, com complexidade total $\mathcal{O}(mn + n^2 \ell og\, n)$ - ou rodar algoritmos para encontrar todos os caminhos mínimos de um grafo, como Floyd-Warshall e Johnson. Algumas alternativas são discutidas em [28] e elas visam a podar a árvore de busca de Dijkstra para determinada partição $k$ a partir do momento em que uma aresta na fronteira

da mesma seja encontrada. Finalmente, como descrito anteriormente, a consulta percorre apenas arestas que participem de caminhos mínimos para a partição de destino.

### 2.2.2 Particionamento

O desempenho do AF depende fortemente da escolha das partições do conjunto de vértices. Normalmente, coordenadas geográficas são fornecidas para cada nó da rede, de forma que exista uma relação espacial que pode ser usada pelos algoritmos de particionamento. Sejam $\ell$, $r$ as coordenadas horizontais dos nós mais a leste e a oeste respectivamente e sejam $b$, $t$ as coordenadas verticais dos vértices mais ao sul e mais ao norte respectivamente. Todos os pontos se encontram no retângulo com diagonal indo de $(\ell, b)$ a $(r, t)$. Quatro estratégias são discutidas no artigo original [28], além de algumas variantes:

**Uniforme** A largura e altura do retângulo são definidas por $\overline{w} = r - \ell$ e $\overline{h} = t - b$. A maneira mais natural de dividi-lo em $K$ partições é fazê-lo de forma uniforme: cada partição será um retângulo de largura $\frac{\overline{w}}{\sqrt{K}}$ e altura $\frac{\overline{h}}{\sqrt{K}}$. Repare que não há nenhuma garantia sobre a quantidade de pontos em cada partição, o que acaba gerando valores bastante desproporcionais: a quantidade de rodovias no Amazonas certamente é ordens de grandeza menor que em São Paulo, por exemplo.

**Quad-tree** Este tipo de estrutura é bastante usada em geometria computacional para dividir um espaço 2D. O retângulo inicial é dividido em quatro regiões de mesmo tamanho. Em seguida, para cada uma das regiões formadas, uma nova divisão em quatro sub-regiões é realizada. O algoritmo continua recursivamente até todas as regiões terem no máximo um ponto ou o número de regiões tornar-se maior que $K$. Esta alternativa parece bem mais razoável que uma repartição uniforme, mas ainda cria uma desproporção no número de pontos por região.

**KD-tree** Outro tipo de estrutura de dados muito comum em geometria computacional. Ela tenta solucionar a desproporção no número de pontos por região causada pela Quad-tree. A cada iteração, todas as regiões que contêm dois ou mais pontos são divididas em duas sub-regiões, cada uma contendo o mesmo número de pontos. Esta divisão na mediana se dá através de uma linha horizontal ou vertical: em iterações pares, a divisão é horizontal, enquanto nas ímpares, vertical (ou o contrário, dependendo da implementação). No trabalho onde AF foi apresentado, é dito que isto obriga o número de regiões a ser uma potência de dois no final, o que faria $K = 2^q$, onde $q$ é o número de iterações do algoritmo. Contudo, esta restrição não é necessária, de forma que é possível construir KD-trees com valores $K$ arbitrários.

**Separadores de Arco** Todos os métodos de particionamento acima precisam de informações adicionais (latitude e longitude) dos nós para trabalhar. Em geral, elas estão facilmente disponíveis para grafos com interpretação geográfica, contudo não para redes genéricas. Existem algoritmos que não precisam de nenhum outro dado para trabalhar, um deles é o METIS [1].

Na prática, apenas os dois últimos são usados, enquanto o resultado do primeiro serve como parâmetro de comparação. O custo computacional de todos é insignificante frente à construção dos vetores de bit e, logo, qualquer um deles pode ser escolhido sem afetar o tempo de pré-processamento do AF.

### 2.2.3 Otimizações

Com apenas 32 bits para cada aresta, arc flags com KD-tree para particionamento já consegue um *speed-up* de mais de 20 vezes comparado ao algoritmo de Dijkstra original. Contudo, pequenas otimizações podem fazer o algoritmo atingir *speed-ups* de mais de 1400 vezes [27].

[1] http://glaros.dtc.umn.edu/gkhome/

**Particionamento Hierárquico**

Os criadores do AF perceberam que mais de 50% das arestas possuíam menos de 10% dos bits ativos, o que representava um gasto de memória desnecessário. Além disso, conforme o número de partições aumentava, o pré-processamento atingia valores da ordem de dezenas de horas.

A solução para o primeiro problema é tentar comprimir os bits ativos, algo que pode ser alcançado através de um mapeamento simples. Esta idéia é nova para o AF e não é proposta nos artigos pesquisados. Como existem apenas $m$ arestas, existem $\mathcal{O}(m)$ valores distintos para o vetor de bits de tamanho $K$ grande (maior que 50 bits). Ora, como em redes reais $m \ll 2^{50}$, é possível criar uma tabela para identificar cada vetor de bits, onde cada identificação caberá em 32 bits (ou mesmo 16) e cada arc flag caberá em um inteiro comum, reduzindo, assim o consumo de memória. Um exemplo de mapeamento para 4 bits é mostrado na Tabela 2.1.

| Identificação | Vetor de Bits |
|---|---|
| 0 | $< 0, 1, 1, 0 >$ |
| 1 | $< 1, 1, 1, 0 >$ |
| 2 | $< 0, 1, 0, 0 >$ |
| 3 | $< 0, 0, 0, 0 >$ |

Tabela 2.1: Exemplo de Mapeamento para 4 partições

Esta idéia provavelmente não é explorada porque não resolve o segundo problema, pois o número de partições continua grande e o pré-processamento demorado. Para solucionar este (e o primeiro) problema, o particionamento é feito em vários níveis. A intuição é dividir a região em poucas partições; em seguida, dividir cada sub-região em novas partições e assim sucessivamente. Apenas o particionamento em dois níveis é mostrado; o funcionamento para mais níveis é análogo.

Suponha que o número de partições necessárias seja 225. Uma solução é usar esta quantidade de bits e calcular arc flags para cada uma delas. Outra é dividir o domínio inicialmente em 25 regiões (grade 5x5) e calcular arc-flags para elas. Para cada região, 9 partições são criadas e arc flags internos calculados, úteis para se

orientar dentro da região apenas. Desta forma, cada aresta carregará 34 em vez de 225 bits. Para facilitar o entendimento, suponha que a divisão em primeiro nível seja mostrada na Tabela 2.2a e que a divisão da região $C$ seja a mostrada na Tabela 2.2b. Os 25 primeiros bits de cada aresta serão usados para dizer se a mesma pertence a um caminho mínimo terminando em uma das regiões $A$ a $Y$. Todas as arestas que terminam em $C$ terão os 9 últimos bits para dizer quais sub-regiões são alcançadas por caminhos mínimos.

| | | | | |
|---|---|---|---|---|
| A | B | C | D | E |
| F | G | H | I | J |
| K | L | M | N | O |
| P | Q | R | S | T |
| U | V | W | X | Y |

(a) Primeiro Nível

| | | |
|---|---|---|
| C1 | C2 | C3 |
| C4 | C5 | C6 |
| C7 | C8 | C9 |

(b) Segundo Nível

Tabela 2.2: Exemplo de particionamento em 2 níveis

A introdução de níveis de hierarquia altera pouco o funcionamento do algoritmo de Dijkstra para arc flags. Num primeiro momento, apenas os bits referentes ao primeiro nível são usados: a busca segue arestas que levem para a região do destino. Uma vez atingida, ela deve se restringir a olhar os bits referentes à sub-região, até que o alvo seja alcançado. Para mais regiões, o funcionamento é análogo.

Repare que existe um problema (*que não é abordado no artigo original*): talvez seja necessário sair de uma região após atingi-la. Suponha que na Figura 2.2 $A$ pertença a uma região e $B$, $C$ a outra. Se a busca começar em $B$ e quiser ir para $C$, nunca será possível passar por $A$, pois uma vez atingida a região em questão, a busca permanece confinada na mesma e somente se move entre sub-regiões. Este tipo de problema acontece quando a atribuição de pesos às arestas não segue a desigualdade triangular, o que pode efetivamente ocorrer em redes de transporte reais, pois pode valer mais a pena no quesito tempo pegar duas rodovias a seguir uma estrada com velocidade máxima baixa.

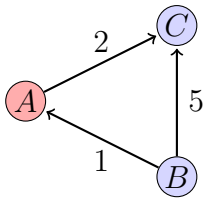



Figura 2.2: Problema com Arc Flags Hierárquico

**Busca Bidirecional**

Uma outra otimização que dá resultados muito bons é a aplicação do algoritmo de Dijkstra bidirecional para a consulta. Desta forma, para ir de $s$ a $t$, uma busca se inicia em $s$ e tenta chegar em $r(s)$, enquanto outra parte de $t$, usa as arestas no sentido oposto e tenta atingir $r(t)$. Elas páram assim que um vértice for explorado pelas duas. Caso esteja sendo usada a variante hierárquica do particionamento, se uma das buscas atingir a partição que contém a outra, os bits que representam a sub-região passam a ser usados.

## 2.3 Contraction Hierarchies

Contraction Hierarchies (CH) é uma evolução de dois métodos mais antigos de roteamento em redes de transporte, Highway Hierarchies e Highway-Node Routing [25]. Atualmente ele se encontra entre os algoritmos mais eficientes para cálculo de rotas de veículos [9], apresentando tempos de consulta da ordem de centenas de microssegundos aliado a uma fase de pré-processamento leve, tanto em consumo de memória, quanto em tempo.

A idéia básica da fase de pré-processamento do CH é tentar criar atalhos para evitar a exploração de vértices que não contribuirão com as consultas. A cada iteração, o vértice menos "importante" é removido junto com todas as suas conexões (arestas a ele incidentes). Para manter parte da estrutura anterior, algumas novas arestas são adicionadas entre antigos vizinhos deste vértice e farão o papel de atalhos. No final, quando restar apenas um nó, o grafo original é reconstruído e todas as novas ligações são adicionadas. Para realizar consultas, tenta-se usar ao máximo

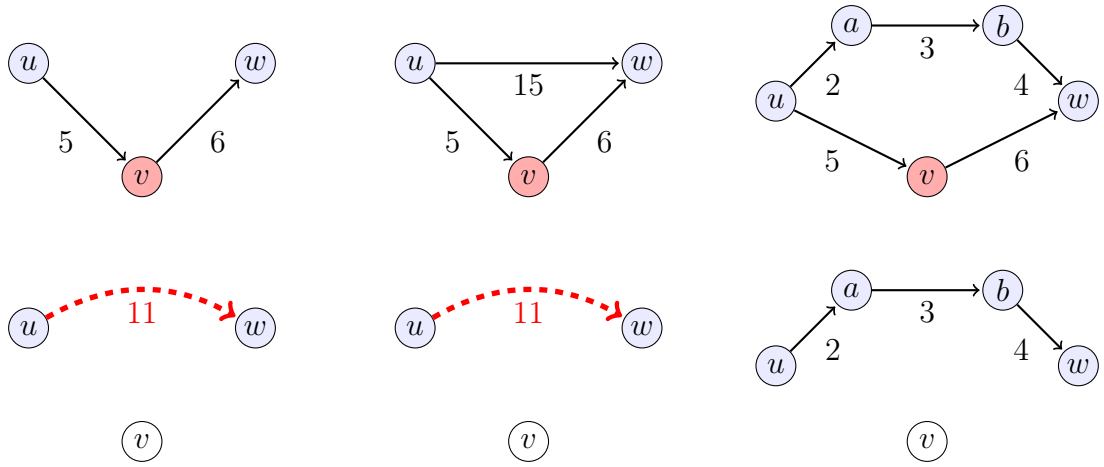
estes atalhos a fim de alcançar o destino mais rapidamente.

A intuição por trás do algoritmo é bastante interessante. Se um bom método de classificação dos vértices for usado, nós mais importantes serão aqueles que representam rodovias. Assim sendo, o CH buscará criar atalhos de pontos em pequenas ruas (baixa importância) para avenidas, estradas, entre outros. Logo, consultas entre lugares geograficamente distantes terão o tempo reduzido ao alcançar rapidamente vias que efetivamente serão usadas, em vez de explorar ruas vizinhas que pouco contribuirão para o resultado. Contudo, é importante notar que o número máximo de atalhos criados após a remoção de um vértice é quadrático na quantidade de vizinhos do mesmo. Assim, é preciso ter atenção para não criar muitas novas arestas e ameaçar o tempo de consulta.

### 2.3.1 Funcionamento

O algoritmo CH cria uma métrica de importância dos vértices de um grafo, que cumprirá o papel de uma relação de ordem total. Em seguida, ele remove sucessivamente o vértice de menor importância da rede e adiciona atalhos entre seus antigos vizinhos - por construção, mais importantes que ele - quando necessário. Os atalhos podem ser tanto novas arestas, como na Figura 2.3a, ou representar a mudança de peso de uma conexão já existente, mostrado na Figura 2.3b. Contudo, nem sempre atalhos se mostram necessários, pois podem existir *caminhos-testemunhas*, ou seja, ligações menos custosas entre os vizinhos. Este fenômeno é mostrado na Figura 2.3c, onde a remoção de $v$ não adiciona uma aresta entre $u$ e $w$, pois existe um caminho entre estes dois últimos com menor peso.

A busca por caminhos-testemunhas é a parte mais custosa do algoritmo, pois é preciso explorar a vizinhança de vértices com o uso de algoritmos de caminho mais curto, como Dijkstra. Ela é fundamental ao impedir que muitos atalhos sem utilidade sejam adicionados à rede. Além dela, a atribuição de importância para os vértices também possui um alto custo. Contudo, ambas são realizadas apenas uma vez para cada grafo e todas as consultas são feitas no grafo resultante. As




(a) Adição de Aresta (b) Atualização de Peso (c) Caminho-Testemunha

Figura 2.3: Grafo antes e depois da remoção de $v$ para três cenários distintos

duas principais etapas da fase de pré-processamento são descritas em detalhe nas próximas seções.

### 2.3.2 Pré-Processamento: Importância

Escolher um valor para a importância de um vértice não é algo simples, pois estratégias erradas de remoção dos vértices tendem a comprometer o desempenho do algoritmo, pois muitos vértices são visitados na hora da consulta. Além disso, este número tende a mudar conforme a rede é contraída a cada etapa do algoritmo de pré-processamento. Uma fila de prioridade (implementada com a ajuda de um heap) é usada; a cada momento, o vértice no topo é aquele de menor importância que ainda não foi contraído (isto é, removido da rede).

O cálculo da importância é feito a partir da combinação linear de vários fatores que serão explorados abaixo. A explicação completa dos mesmos se encontra em [25]. Alguns termos da soma possuem uma complexidade de cálculo elevada e, portanto, torna-se inviável a atualização de todos os valores após a contração de um vértice. A solução proposta é uma política de *lazy update*, onde apenas o topo da fila de prioridade é atualizado e eventualmente trocado (o mesmo procedimento é aplicado ao novo topo até que este processo convirja). Após muitas iterações, os valores da fila podem estar muito diferentes da realidade. A solução é recalcular todas

as importâncias dos vértices ainda presentes periodicamente. Além disso, outras atualizações são necessárias a cada remoção de nó, mas estas em geral comportam apenas os vizinhos imediatos do mesmo. Como os vértices de redes rodoviárias possuem em geral poucos vizinhos, esta política é aceitável.

### Diferença de Arestas

Quanto mais atalhos forem criados após a contração de um vértice, mais denso é o grafo resultante e mais custosa se torna a fase de pré-processamento. Além disso, a existência de muitos atalhos não é desejada, pois eles podem confundir o algoritmo de Dijkstra na hora da busca e resultar na visita de um número grande de vértices. Assim, uma boa estratégia de remoção de nós é tal que a quantidade de arestas diminua a cada iteração.


Figura 2.4: Exemplo da métrica de Diferença de Arestas para o vértice $v$. Antes de sua remoção, havia 5 arestas ao seu entorno; depois, apenas 3 atalhos foram criados. Seu valor de Diferença de Arestas é, portanto, -2

Contar o número de arestas a cada etapa é relativamente custoso. No mais, é fácil ver que contrair $v$ apenas acarretará mudanças no número de arestas em sua vizinhança: arestas $(u, v)$ e $(v, w)$ serão removidas; atalhos $(u, w)$ serão eventualmente criados. A Figura 2.4 mostra como este cálculo é feito. Quando $v$ é removido do grafo, os valores de diferença de aresta são calculados para todos os vértices que eram seus vizinhos e estes são atualizados na fila de prioridade.

### Custo da Contração

Quando um vértice $v$ é contraído, é preciso encontrar testemunhas entre todos os pares $(u, w)$, para arestas $(u, v)$ e $(v, w)$. O algoritmo de Dijkstra apenas termina

quando todos os vizinhos tiverem sido visitados. Se o número de vértices intermediários visitados for alto, o pré-processamento torna-se muito lento e provavelmente muitas arestas serão usadas na hora da consulta, pois a quantidade de atalhos tende a ser grande. Assim, este valor entra também na conta para a importância dos nós.

### Uniformidade da Remoção

Remover vértices de uma mesma região do mapa em sequência pode ter um efeito bastante negativo. Na Figura 2.5, se o primeiro vértice a ser removido for $v_1$, de acordo com as métricas acima, $v_2$ será o seguinte e assim sucessivamente. Contudo, neste caso, para chegar de $v_1$ até $w$, $k+1$ nós deverão ser visitados. Como a visita de um nó é a parte mais lenta do algoritmo de Dijkstra, isto ameaça o desempenho das consultas.

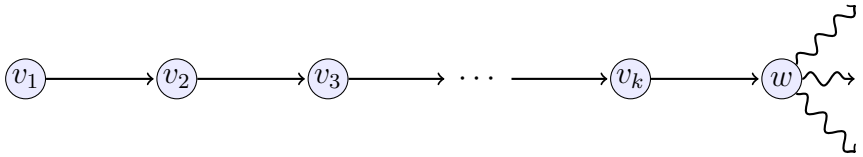



Figura 2.5: Exemplo onde o algoritmo de pré-processamento gera um grafo no qual a execução de Dijkstra numa consulta visita um número linear de vértices

Para evitar este tipo de problema, o número de vizinhos removidos é considerado no cálculo da prioridade. Desta forma, no exemplo anterior, após o vértice $v_1$, $v_3$ seria o próximo removido, depois $v_5$ e assim sucessivamente. Em seguida, a contração seria feita de maneira análoga com os nós que restam, até que sobre apenas um único. É fácil ver que o número de etapas para contrair todos os vértices entre $v_1$ e $v_k$ será da ordem $\ell og\, k$, ou melhor dizendo, $w$ se conectará a $\ell og\, k$ diferentes nós até que a todos os $v_i$ sejam contraídos. Além disso, atalhos serão seguramente criados, diferentemente do cenário anterior. Assim, será possível sair deste beco sem saída num número logarítmico de passos (nós visitados por Dijkstra).

Finalmente, outras estratégias são discutidas no trabalho original [25]. Uma delas, usando regiões de Voronoi, apresenta excelente desempenho e apresenta estratégias de atualização relativamente eficientes. Ela também garante que a ordem

de escolha dos vértices é feita de forma "espalhada" e, portanto, atende ao critério de uniformidade.

### Profundidade da Busca

Se durante a consulta, a árvore gerada pelo algoritmo de Dijkstra até atingir o destino for profunda, muitos vértices acabarão sendo visitados e o desempenho cairá muito. Durante a consulta, a busca só explora vértices com maior importância do que o atual. Assim, é possível realizar o cálculo da profundidade máxima de forma bastante simples. Seja $h(u)$ a altura máxima da árvore de busca que termina no vértice $u$. Ao remover $u$, cada um de seus vizinhos que ainda não foram contraídos (removidos) $v$ poderá ter altura $h(u) + 1$, pois talvez $v$ seja visitado logo após $u$.

Ao iniciar a fase de pré-processamento, todos os $h(v)$ valem 0. Na fase $i$, seja $v_i$ o vértice a ser removido e $V_i$ o conjunto de vértices com maior prioridade restantes. Os valores de $h$ são atualizados de acordo com a Equação 2.2.

$$h(v_j) = \max\{h(v_j), h(v_i) + 1\}, \quad v_j \in V_i \tag{2.2}$$

Após a contração de um vértice, apenas a prioridade de seus vizinhos deve ser atualizada. Intuitivamente, como os vértices de baixa "altura" terão menor prioridade, estes serão os primeiros a ser contraídos e, portanto, a árvore gerada pelo algoritmo de Dijkstra na hora da consulta não será muito profunda.

### Medidas Globais

Vértices que não contribuem para muitos caminhos mínimos podem ser removidos sem a adição de um número grande de atalhos. Logo, eles deveriam ser contraídos o mais cedo possível. O número de caminhos mínimos que passam por um nó é uma métrica de centralidade e se chama *betweenness* [30].

O grande problema com o betweenness é o fato de seu valor poder variar entre 0 e $n^2$ e, logo, a contribuição deste é muito mais alta no cálculo da importância do que as métricas anteriores. Uma solução muito simples para este problema é definir uma

função tal que $c(v)$ seja o percentual de vizinhos de $v$ que apresentam betweenness menor que a de $v$. A imagem desta função é o intervalo $[0, 1]$, amortizando o peso desta medida. O grande problema agora é que este valor possui um significado apenas local, e não mais global.

### 2.3.3 Pré-Processamento: Contração

A busca por caminhos-testemunhas é necessária na hora de calcular a importância do vértice e durante o processo de contração. Seja $v$ o vértice para o qual se quer encontrar caminhos-testemunhas e $N_-(v)$, $N_+(v)$ os conjuntos de predecessores e sucessores de $v$ respectivamente. Claramente é preciso procurar o caminho mais curto entre cada par $u \in N_-(v)$ e $w \in N_+(v)$. Durante a contração de $v$, estes conjuntos são menores, pois supõe-se que parte de seus vizinhos já foi contraída; mesmo assim, o custo computacional continua elevado, pois é necessário rodar Dijkstra para múltiplas fontes. Duas heurísticas para tornar este processo mais rápido são [25]:

- *Limitar o número de nós explorados*: certamente não é uma opção muito atrativa para ser feita durante a contração, pois poucos caminhos-testemunhas são ignorados e o grafo tende a se tornar cada vez mais denso. Na implementação original do CH, esta solução é proposta apenas para a fase de cálculo da importância dos nós, pois o conjunto de vizinhos é maior e se caminhos-testemunhas forem ignorados, apenas a prioridade do vértice será ligeiramente alterada;

- *Limitar o número de saltos*: o problema neste caso é achar uma estratégia para definir a quantidade de saltos que deve ser considerada como limite. Uma estratégia adaptativa (trade-off entre densidade do grafo resultante e custo computacional) é proposta e algoritmos para a acelerar a busca são descritos no artigo original.

### 2.3.4 Consulta

Seja $G = (V, E)$ o grafo resultante após o processo de contração (arestas originais mais atalhos) e $\omega : V \to \{1, \cdots, |V|\}$ uma bijeção representando a importância dos vértices do grafo. Para encontrar o caminho mais curto entre dois vértices $s$, $t \in V$, é preciso primeiramente definir dois subgrafos de $G$:

- *Grafo crescente* $G_{\uparrow}$: subgrafo contendo apenas as arestas "crescentes". Em termos matemáticos, $G_{\uparrow} = (\mathrm{V}, E_{\uparrow})$, onde $E_{\uparrow}$ é o conjunto das arestas $(x, y) = e \in E$ tais que $\omega(x) < \omega(y)$;

- *Grafo decrescente* $G_{\downarrow}$: subgrafo contendo apenas as arestas "decrescentes". Em termos matemáticos, $G_{\downarrow} = (\mathrm{V}, E_{\downarrow})$, onde $E_{\downarrow}$ é o conjunto das arestas $(x, y) = e \in E$ tais que $\omega(x) > \omega(y)$.

É razoável pensar que qualquer caminho entre $s$ e $t$ pode ser de apenas três formas: estritamente contido em $E_{\uparrow}$, estritamente contido em $E_{\downarrow}$ e contido na união dos dois. No último caso, é possível restringir ainda mais a forma, pois como é mostrado em [25], todo caminho mínimo $s-t$ pode ser representado por um caminho crescente $s-p$ e outro decrescente $p-t$. Assim, é natural definir um método usando Dijkstra bidirecional a fim de conectar $s$ e $t$, como mostrado no Algoritmo 1.

**Algoritmo 1** Dijkstra Bidirecional para a consulta no CH

```
Definir duas filas de prioridades P↑ e P↓
Inserir s, t em P↑ e P↓ respectivamente
repetir
  se topo de P↑ < topo de P↓ então
    Remover o topo de P↑ e explorá-lo
    Atualizar P↑
  senão
    Remover o topo de P↓ e explorá-lo
    Atualizar P↓
  fim se
  Definir p como sendo o vértice explorado
até p ter sido explorado pelas duas buscas
imprimir Concatenação dos caminhos {s − p} e {p − t}
```

Após realizar a consulta, é necessário imprimir o caminho. Contudo, como as arestas percorridas pelo algoritmo de Dijkstra podem ser atalhos, é necessário realizar um procedimento de "descompactação" do grafo, onde atalhos utilizados são traduzidos no conjunto de conexões (sequência de arestas) que representam.

# Capítulo 3

# Redes com Dependência Temporal

Como discutido no capítulo anterior, o peso das arestas de redes de transporte podem variar conforme o tempo por causa de condições de tráfego adversas. A dependência temporal também ocorre quando são consideradas redes de transporte público. Até o momento, apenas consultas usando transporte privado (rodovias) foram consideradas: como chegar de um ponto a outro de carro. Este capítulo aborda um problema muito mais difícil, que responde a consultas em redes em que apenas o transporte público está disponível (conhecidas em inglês como *transit networks*). A dificuldade se encontra na não disponibilidade de ônibus e outros veículos durante parte do dia, além de intervalos irregulares de serviços dos mesmo.

Este capítulo estudará modelos e algoritmos para *transit networks*. A próxima seção definirá conceitos importantes para o resto do capítulo. Em seguida, modelos de redes de transporte público serão mostrados, além dos diferentes tipos de consulta que podem ser feitos. Finalmente, dois algoritmos para este tipo de rede serão explorados em detalhes e um terceiro será explicado superficialmente.

## 3.1 Fundamentos

Redes de transporte público possuem uma nomenclatura particular que é usada na literatura. Como não foram encontradas publicações em português a respeito do assunto, todos os nomes foram traduzidos de maneira literal. As definições a seguir

falam de conceitos que serão usados ao longo do capítulo.

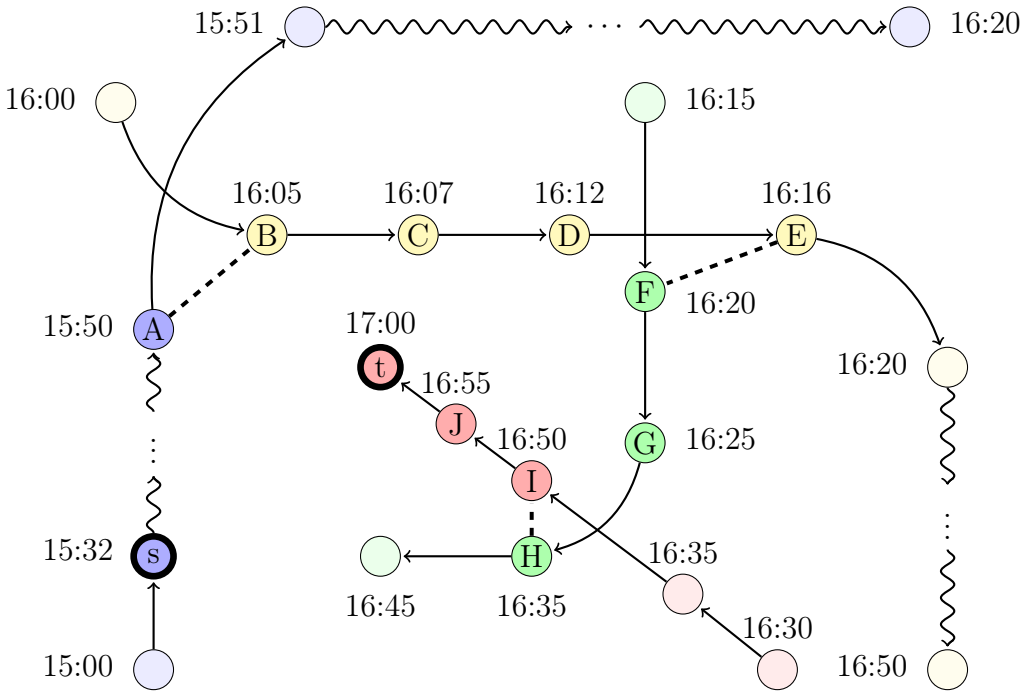



Figura 3.1: Exemplo de uma consulta EAT entre $s$ e $t$ partindo antes de 15:32

**Definição 1** (Parada)**.** *Lugar onde é possível subir ou descer de um veículo de transporte público.*

**Definição 2** (Rota ou Linha)**.** *Lista de paradas servidas em ordem por um veículo de transporte público. Exemplo: Rota de ônibus 485 General Osório - Penha.*

**Definição 3** (Viagem)**.** *Instância de uma rota, com atribuição de tempos de partida e chegada a cada parada. Exemplo: Linha 485 partindo de General Osório às 8:00 de terça-feira.*

**Definição 4** (Transferência)**.** *Mudança de uma rota para outra realizada em paradas compartilhadas pelas duas.*

**Definição 5** (Caminhos de Pedestre)**.** *Transferências entre rotas realizadas a pé, quando não existe uma parada compartilhada.*

A Figura 3.1 mostra um exemplo dos conceitos acima. São mostradas 4 rotas (em azul, amarelo, verde e vermelho), com uma atribuição de viagens para cada

uma delas. Para chegar de $s$ a $t$ partindo antes de 15:32, é preciso pegar a viagem que parte às 15:00 do terminal, realizar uma transferência para a linha amarela entre as estações $A$ e $B$, usar a viagem partindo às 16:00 da estação terminal, transferir de $E$ para $F$ para a linha verde. Em seguida, usa-se a viagem que parte da primeira parada às 16:15 até $H$, onde ocorre uma transferência para a parada $I$ da rota vermelha. Finalmente, toma-se a viagem partindo às 16:50 e chega-se em $t$ às 17:00. Todas as transferências mostradas por linhas pontilhadas podem se referir a transferências simples ou caminhos de pedestre. Esta mesma figura pode ser pensada numa consulta $EAT$ do tipo $(s \,@\, 15{:}32, t)$.

**Definição 6** (Consulta EAT)**.** *Uma consulta do tipo Earliest Arrival Time (EAT) procura encontrar uma sucessão de viagens e pontos de transferência entre rotas de forma a chegar o mais rápido possível ao destino. Ela é representada sob a forma de uma tupla* $(A \,@\, t, B)$*, para consultas que partem de* $A$ *no tempo* $t$ *e desejam chegar o mais cedo possível em* $B$*.*

Como será visto nas próximas seções, existem outros tipos de consulta possíveis, onde mais critérios são considerados. Consultas EAT tendem a retornar resultados com 3 ou mais transferências, mesmo para distâncias pequenas, algo que não é aceitável na vida real.

Em alguns algoritmos, o conceito de conexão é fundamental. Trata-se de um evento atômico que caracteriza quase completamente redes de transporte.

**Definição 7** (Conexão)**.** *Par de eventos de chegada e saída de um veículo de uma mesma viagem a uma parada. Cada conexão é representada por uma tupla* $(T_{id}, S_{id}, t_{arr}, t_{dep})$*, onde* $T_{id}$ *e* $S_{id}$ *são respectivamente a identificação da viagem e da parada e* $t_{arr}$*,* $t_{dep}$ *os instantes de chegada e partida de* $S_{id}$*.*

É possível pensar numa viagem como sendo uma lista de conexões. Para tanto, uma ordem de precedência pode ser definida: a conexão $C_j$ é sucessora de $C_i$ se $T^i_{id} = T^j_{id}$ e $t^j_{arr} \geq t^i_{dep}$. A partir de agora, quando aparecer a expressão "próxima conexão", será em referência à conexão imediatamente sucessora da atual. Finalmente, existe

uma forma de organizar viagens de uma mesma rota ao listar todas as respectivas conexões numa tabela, como na Figura 3.1.

**Definição 8** (Timetable)**.** *Organização das viagens de uma rota em forma de tabela, onde as entradas são conexões.*

| **Viagem** | **Parada 1** | **Parada 2** | **Parada 3** | $\cdots$ | **Parada 19** | **Parada 20** |
|---|---|---|---|---|---|---|
| 158796 | 08:00 | 08:05 | 08:10 | $\cdots$ | 09:38 | 09:45 |
| 184855 | 09:00 | 09:05 | 09:10 | $\cdots$ | 10:38 | 10:45 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 548974 | 18:00 | 18:05 | 18:10 | $\cdots$ | 19:38 | 19:45 |
| 564454 | 19:00 | 19:05 | 19:10 | $\cdots$ | 20:38 | 20:45 |

Tabela 3.1: Exemplo de timetable para uma rota

## 3.2 Modelos Clássicos

Historicamente, existem duas formas distintas de modelar redes de transporte público ou, melhor dizendo, os eventos que acontecem nela. Cada uma representa eventos de chegada e partida de formas distintas e necessita de algoritmos diferentes para realizar consultas. A grande dificuldade se encontra no fato de nem todas as arestas poderem ser usadas em qualquer instante, pois existem tempos pré-determinados para as conexões.

A fim de facilitar a visualização de alguns exemplos, três tipos de eventos serão usados para uma parada $S$: partida (dep@S), chegada (arr@S) e transferência (tra@S).

### 3.2.1 Rede de Dependência Temporal

Na rede de dependência temporal (ou *time-dependent graph*), cada parada de uma rota é representada por um vértice e arestas direcionadas modelam o sentido em que as viagens ocorrem. As arestas da rota podem ser usadas apenas em determinados instantes de tempo, em outras palavras, quando existir uma conexão entre os dois extremos da aresta. Esta dependência temporal para uma aresta $(A, B)$ é

mostrada na Tabela 3.2. Sempre que uma pessoa chegar em $A$, ela poderá pegar apenas a conexão subsequente para ir para $B$. Por exemplo, se a chegada em $A$ for às 08:50, o tempo de translado para $B$ corresponde a 20 minutos: 10 minutos de espera, somados a 10 minutos da conexão. Assim sendo, o peso das arestas é variável com o tempo e a cada vez que o algoritmo de Dijkstra chega em uma parada de uma rota, ele precisa procurar a próxima conexão que pode ser usada e, com isto, computar quanto tempo levará para chegar no próximo nó. O *label* usado por Dijkstra para marcar os nós corresponderá ao tempo de chegada no vértice.

| **Evento** | **Horário** | | | | | |
|---|---|---|---|---|---|---|
| dep@A | 08:00 | 08:30 | 09:00 | $\cdots$ | 21:00 | 21:30 |
| arr@B | 08:10 | 08:40 | 09:10 | $\cdots$ | 21:10 | 21:40 |

Tabela 3.2: Exemplo de conexões que servem uma aresta $(A, B)$

Diversas rotas podem servir uma mesma parada e o tempo de transferência entre elas pode não ser desprezível. Para solucionar este problema, um vértice adicional é criado para cada parada, representando o vértice de transferência entre rotas. Se uma rota passa pela parada $S$ e precisa de $t$ segundos para embarque (respectivamente desembarque), um arco entre o vértice que representa $S$ na rota em questão e o nó de transferência de $S$ (resp. o inverso) é criado com peso $t$. Além disso, se duas estações $S_1$ e $S_2$ podem ser alcançadas a pé, uma aresta - no caso, dois arcos, pois o grafo é direcionado - é adicionada entre as duas com custo equivalente ao tempo de translado entre elas. Estes dois novos tipos de aresta possuem peso fixo, diferentemente das que representam conexões de uma mesma rota. Assim, as consultas EAT podem ser respondidas a partir do vértice de transferência da parada inicial, chegando no nó que representa a transferência da estação de destino.

O modelo que acaba de ser mostrado considera que existem tempos de transferência entre diversos lugares da mesma parada. Este, contudo, nem sempre é o caso. Em algumas situações, o valor citado é desprezível e apenas a transferência entre estações diferentes deve ser considerada. Neste caso, é possível condensar todas as paradas iguais num mesmo nó. Desta forma, a noção de mudança de rota seria perdida e o número de vértices do grafo representaria exatamente a quantidade de

paradas. Arestas existiriam entre vértices que são ligados por alguma rota ou que podem ser alcançados por transferência a pé. O custo das primeiras seria calculado de maneira semelhante à anterior, mas considerando conexões geradas por todas as rotas que servem simultaneamente as duas paradas. Esta modelagem é pouco realista e não usada em aplicações reais. Contudo, trata-se de uma bom método de comparação do desempenho do algoritmo de Dijkstra com outros métodos especializados para redes de transporte público.

### 3.2.2 Rede de Tempo Expandido

Um grande problema das redes de dependência temporal encontra-se na necessidade de utilizar uma variante de Dijkstra, o que dificulta a aplicação de outros métodos para grafos, como visto no capítulo anterior. Para resolver este problema, existem as redes de tempo expandido (*time-expandend graph*), onde cada conexão de uma rota é representada por três vértices do grafo, um para cada tipo de evento possível, e cada um deles possui um tempo associado, que diz em que momento o evento ocorre. Arestas conectam vértices de chegada e partida subsequentes de uma mesma viagem. Além disso, cada vértice de chegada liga-se a um vértice de transferência e estes se conectam aos vértices de partida de uma mesma estação que partem o mais cedo possível (para um tempo associado de partida maior do que o tempo associado de transferência). Todos os vértices de transferência de uma mesma parada são ordenados e arestas conectam nós subsequentes. É possível também modelar caminhos a pé entre diferentes estações. Finalmente, o peso de cada uma das arestas descritas corresponde à diferença entre o tempo associado aos dois eventos ligados por ela. A Figura 3.2 representa um subconjunto de eventos de uma rede de tempo expandido, onde os vértices de cor cinza pertencem à mesma rota, os de cor preta são usados para transferências e todos os outros mapeiam eventos de diferentes rotas.

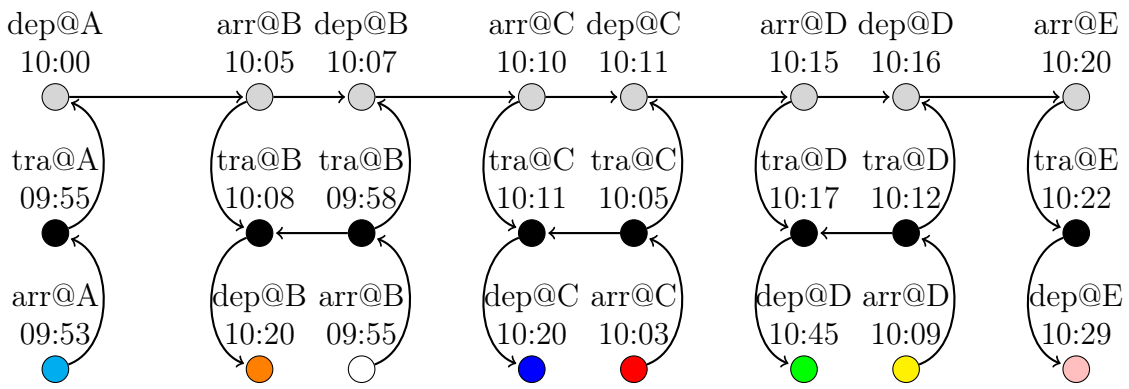



Figura 3.2: Exemplo de parte de uma rede de tempo expandido

O maior problema deste tipo de modelagem é o tamanho do grafo criado: $3n$ vértices (onde $n$ é o número de conexões), com grau médio de saída em torno de 2. O número de conexões pode atingir facilmente dezenas ou mesmo centenas de milhões para redes de grandes cidades. Isto torna o algoritmo de Dijkstra impraticável para aplicações reais. Contudo, diferentemente da rede de dependência temporal, é possível aplicar outros tipos de algoritmos vistos no capítulo anterior - além de abrir portas para a definição de um novo método que será descrito em breve. De qualquer forma, aplicar algoritmos de redes de transporte privado (Arc Flags, Contraction Hierarchies) se provou ineficaz em vários caso, pois estes precisam muitas vezes de uma interpretação geométrica dos nós para poder funcionar da melhor maneira possível, algo que é difícil de fazer para o caso de conexões, pois seus vértices possuem uma componente temporal atrelada à geométrica.

## 3.3 Otimização Multi-Critério

Geralmente usuários de softwares como *Google Maps* não estão interessados apenas em chegar o mais cedo possível na destinação com transporte público. Eles querem, sim, chegar relativamente cedo, mas tendo que fazer ao mesmo tempo poucas transferências e gastando pouco dinheiro com bilhetes. Desta forma, os algoritmos por trás destas ferramentas precisam otimizar diversos critérios ao mesmo tempo.

Contudo, não é claro se um resultado com 2 transferências e de translado de 30 minutos é melhor que um outro 1 transferência e 1 hora de duração. Isto se deve ao fato de nenhuma solução *dominar* a outra. Os dois resultados anteriores podem ser representados sob a forma de tuplas, onde cada posição indica o valor de um dos critérios de otimização: $(2, 30)$ e $(1, 60)$ respectivamente. Todos os problemas são de minimização (tempo de viagem, número de transferências, preço, etc).

**Definição 9** (Pareto-dominado)**.** *Uma solução* $(x_1, x_2, \cdots, x_n)$ *é dita (Pareto-) dominada por uma outra* $(y_1, y_2, \cdots, y_n)$ *se existe um* $i$ *tal que* $x_i > y_i$ *e, para todo* $j \neq i$, $x_j \geq y_j$.

O conjunto de soluções que não são Pareto-dominadas por nenhuma outra é chamado de *conjunto Pareto-ótimo* e cada um de seus elementos é referenciado como sendo um *ótimo de Pareto*. A cardinalidade deste conjunto é exponencial no número de critérios considerados (tamanho das tuplas). Uma vez que mais critérios são usados, a otimização se torna extremamente custosa ou mesmo inviável.

Neste trabalho, todos os algoritmos considerados tratarão apenas otimizações do tempo de chegada (consultas EAT) ou do tempo de chegada e número de transferências simultaneamente (consultas EAT+TR). Isto se deve a dois motivos: o primeiro é o custo adicional da adição de mais critérios; o segundo é não disponibilidade de informações a respeito de preço das viagens no formato dos dados que será descrito no próximo capítulo.

### 3.3.1 Consultas de Perfil

Outro tipo comum de problema é achar a resposta de consulta para diversos tempos de partida. Mais especificamente, dado um intervalo $[t_1, t_2]$, procurar todas as soluções que são ótimos de Pareto e cujo tempo de partida é $t_1 \geq t \geq t_2$. Este tipo de problema é chamado de *consulta de perfil* (ou *de intervalo*). Ele é importante para casos onde o tempo de chegada do usuário no ponto de partida é incerto ou quando se quer obter todas as soluções para um intervalo de tempo para futuramente

escolher a melhor. Muitas aplicações usam intervalos de 24 horas, como viagens de trem de longa distância.

Contudo, dependendo da granularidade do tempo, a quantidade de soluções pode crescer rapidamente, principalmente caso vários critérios sejam otimizados simultaneamente. Uma idéia natural é considerar o tempo de partida como um critério adicional e, desta forma, reduzir o número de respostas possíveis, tendo em vista que muitas delas serão dominadas. Na prática, porém, o número de respostas não é muito grande, pois raramente alguma parada possui mais do que algumas poucas viagens servindo em um intervalo pequeno de tempo.

## 3.4 Connection Scan Algorithm

*Connection Scan Algorithm* (CSA) foi proposto em 2013 em um artigo com título provocativo [20]. De fato o algoritmo é muito facilmente explicável e os autores delegaram pouco espaço para a tarefa. O algoritmo em sua formulação original é capaz de resolver consultas do tipo EAT para redes de transporte com dependência temporal. Além disso, ele pode ser estendido a fim de tratar múltiplos critérios ao mesmo tempo. Finalmente, os autores também propõem um método robusto de cálculo de rotas derivado do CSA, onde procura-se minimizar o tempo médio esperado de chegada. Neste trabalho, apenas serão tratadas consultas do tipo EAT e a extensão multi-critério onde o número de transferências também é considerado.

A maior vantagem do CSA é não precisar se apoiar em grafos para trabalhar, o que reduz o custo de armazenamento e, ao mesmo tempo, a complexidade de alguns algoritmos. Grande parte dos métodos vistos até agora se baseia em Dijkstra ou variantes, mas como visto, ele não é adequado para tratar consultas em redes grandes que necessitam de uma resposta em tempo-real. Este novo método mantém todos os dados necessários num vetor (e, portanto, com armazenamento contíguo em memória) e faz uma varredura sequencial no mesmo, reduzindo ao máximo o número de acessos aleatórios à memória principal. Mesmo tendo que ler mais informações, ele possui tempos de consulta competitivos com os melhores métodos disponíveis

pelo fato de ter um alto percentual de *cache hits*. Com memórias cache L1, L2 (e mesmo L3) cada vez maiores, CSA é uma boa opção por explorar ao máximo essas componentes. Finalmente, o algoritmo não apresenta um problema comum a vários outros métodos disponíveis na literatura: praticamente todos os cálculos por trás dele são feitos *on the fly* e, logo, ele é perfeito para situações onde a rede experimenta um grande dinamismo.

### 3.4.1 Funcionamento

Um aspecto fundamental a ser observado é que a Rede de Tempo Expandido é um DAG (grafo direcionado acíclico), onde cada nó representa uma conexão. Desta forma, existem meios melhores de fazer uma busca sem precisar usar algoritmos como Dijkstra ou Bellman-Ford. Uma delas é através de uma ordenação topológica do grafo. Trata-se de criar uma ordem de exploração dos vértices, tal que se existe uma aresta $(u, v)$, o nó $u$ é visitado antes de $v$. Todo DAG admite uma ordenação topológica, que não é única. Esta pode ser calculada com algoritmos de busca simples, como a DFS, em tempo linear no número de arestas e vértices da rede. Como a Rede de Tempo Expandido pode ser obtida diretamente através de sua timetable com ligeiras alterações, esta última será considerada de agora em diante.

O algoritmo possui uma pequena fase de pré-processamento, onde todas as conexões presentes na timetable são inseridas num vetor e ordenadas de acordo com o tempo de partida. Empates são desfeitos considerando também o tempo de chegada da conexão; caso ainda haja empate entre $C_i$ e $C_j$ - isto é, $t^i_{arr} = t^j_{arr}$ e $t^i_{dep} = t^j_{dep}$ -, qualquer ordem entre duas é válida. Não é difícil perceber que esta ordenação define de fato uma ordenação topológica (a direção das arestas é a mesma do sentido de crescimento do tempo de partida no grafo original). Contudo, ordenar *todas* as conexões de uma timetable quebra as relações de precedência dentro de uma mesma viagem. Para contornar este problema, um novo atributo é adicionado: um ponteiro para a próxima conexão dentro da mesma viagem (caso seja a última, não aponta para nada). O custo do pré-processamento equivale ao custo de uma ordenação de

todas as conexões de uma rede de transporte público de uma cidade, algo que pode ser feito em poucos segundos.

#### Caminhos de Pedestre

Com a remoção dos vértices de transferência da Rede de Tempo Expandido, não é mais possível realizar transferências entre duas paradas próximas geograficamente. Para contornar este problema, para cada parada $S$ é definido o conjunto $N(S)$ de paradas que podem ser alcançadas a pé partindo de $S$. Antes de uma conexão em $S$ ser analisada, todos os caminhos de pedestre são relaxados: seja $S' \in N(S)$ e o tempo para percorrer o caminho igual a $\theta$, então $\tau_S = \min\{\tau_S, \tau_{S'} + \theta\}$.

### 3.4.2 Consulta

Antes de mais nada, para toda parada $S$, são armazenados o tempo mínimo de chegada $\tau$ na mesma e a conexão $c$ que foi usada para chegar em $S$ no tempo $\tau$ (mesmo que tenha sido a pé). Isto permitirá a reconstrução do caminho entre a origem e o destino. Estes valores são atualizados ao longo da execução do CSA, sempre que uma conexão "mais interessante" for encontrada.

Para realizar uma consulta EAT do tipo $(A \,@\, t, B)$, o primeiro passo consiste em procurar a primeira conexão partindo de $A$ em $t$ ou mais tarde. É evidente que nenhuma conexão anterior a esta pode ser usada, pois $t$ é o tempo de chegada mínimo em $A$. Esta procura pode ser realizada com o auxílio de uma busca binária - em tempo logarítmico -, tendo em vista que todas as conexões estão ordenadas de acordo com o tempo de partida. O tempo $\tau_A$ de chegada em $A$ é atualizado para $t$ e o de todas as outras paradas é definido como sendo $\infty$.

Seja $c$ uma conexão partindo da parada $S$ e seja $c'$ a conexão sucessora direta de $c$ na mesma viagem. Ora, se $\tau_S \leq t^c_{dep}$, então a conexão $c$ pode ser tomada em $S$ (ela foi alcançada antes que o veículo saísse). Caso $c'$ exista (isto é, $S$ é o ponto final da viagem), é possível que $t^{c'}_{arr}$ seja menor que $\tau_{S'}$, ou seja, a conexão $c'$ chega na sua respectiva parada $S'$ antes do que o tempo mínimo de chegada. Neste caso, $\tau_{S'}$

é atualizado para o valor $t^{c'}_{arr}$ e $c'$ é armazenada como sendo a conexão que permitiu chegar em $S'$ no tempo $\tau_{S'}$.

O procedimento descrito no último parágrafo é repetido desde a primeira conexão possível partindo de $A$ até uma conexão $c^{\#}$ tal que $t^{c^{\#}}_{dep} > \tau_B$. Neste caso, é certo que as conexões que vêm a seguir não contribuirão para o cálculo do EAT entre $A$ e $B$. O Algoritmo 2 mostra uma versão sucinta dos passos descritos acima.

**Algoritmo 2** CSA aplicado a uma consulta $(A\,@\,t, B)$

**Condição Necessária** Conexões ordenadas de acordo com o tempo de partida
  Definir $\tau_P = \infty$ para toda parada $P \neq A$; escolher $\tau_A = t$
  Buscar primeira conexão $c$ partindo de $A$ após $t$
  **enquanto** $t^c_{dep} \leq \tau_B$ **faça**
    Relaxar todos os caminhos de pedestre que chegam em $S$
    **se** $t^c_{dep} \geq \tau_S$ **e** $c$ não é a última conexão da sua viagem **então**
      Recuperar a próxima conexão $c'$ da mesma viagem que $c$
      **se** $t^c_{arr} < \tau_{S'}$ **então**
        $\tau_{S'} = t^{c'}_{arr}$
        Definir a conexão que chega na parada $S'$ como sendo $c'$
      **fim se**
    **fim se**
    Atualizar $c$ para a próxima conexão do vetor de conexões
  **fim enquanto**
  **imprimir** Tempo Mínimo de Chegada em B $\tau_B$

Antes de prosseguir, é interessante ressaltar alguns pontos. Primeiramente, é possível notar que toda consulta necessita apenas do vetor de tempos de chegada mínimo para cada parada. Como o número de paradas é, em geral, duas ou mais ordens de grandeza menor que o número de conexões, é possível realizar diversas consultas ao mesmo tempo sem aumentar absurdamente a quantidade de memória usada. Em segundo lugar, fica claro que este algoritmo não vê a diferença entre ir de uma conexão para outra dentro do veículo e realizar uma transferência. Assim, é possível que no final o número de transferências para o resultado ótimo seja irreal, como 5 ou 6. Finalmente, é possível reconstruir o caminho a ser tomado de trás para frente, começando em $B$. Contudo, o algoritmo não é mostrado para não alongar ainda mais a descrição do CSA.

**Prova da Corretude**

Para esta prova, considera-se por simplicidade que caminhos a pé não são permitidos e que a consulta é da forma $(A \,@\, t, B)$. Seja $\tau_C(T)$ o tempo de chegada mínimo na parada $C$ considerando apenas conexões cujo tempo de partida se encontra no intervalo $[t, T)$. Para continuar, é preciso supor que não existam três paradas $X$, $Y$, $Z$ tais que exista uma conexão partindo de $X$ em $T'$ e chegando em $Y$ no mesmo tempo $T'$, além de uma segunda conexão partindo de $Y$ em $T'$ e chegando em $Z$ ainda em $T'$. Na prática, após a ordenação da timetable, a ordem das duas conexões poderia ser qualquer uma. Desta forma, uma possibilidade seria ter a conexão indo de $Y$ para $Z$ aparecendo antes da de $X$ para $Y$. Suponha, por simplicidade, que os tempos mínimos de chegada em $X$, $Y$ e $Z$ sejam respectivamente 0, $\infty$ e $\infty$. Claramente o algoritmo ignorará a primeira das duas conexões, pois $\tau_Y = \infty$, ao passo que a mesma deveria ser considerada. Felizmente, este tipo de situação é extremamente rara e seus efeitos colaterais podem ser negligenciados.

**Proposição 1.** *Dada a consulta EAT* $(A \,@\, t, B)$ *e um* $T \geq t$*, é impossível chegar em* $C$ *antes de* $\tau_C(T)$*.*

*Demonstração.* A prova é feita por indução no tempo $T$:

- Para $T = t$, apenas $A$ é alcançável num primeiro momento e $\tau_A = t$, que é claramente mínimo. Como nenhuma conexão válida tem um tempo de partida anterior a $T$, todos os outros $\tau_C(T) = \infty$ também são mínimos;

- Para $T + 1 > t$, está garantido que $\tau_C(T)$ é mínimo para cada parada. Todas as conexões partindo exatamente em $T$ podem ser expressas sob a forma $(X, T, X', T')$: saindo de $X$ no tempo $T$ e chegando na parada $X'$ em $T'$. Suponha que $T'_*$ seja o tempo de chegada mínimo para todas as conexões da forma $(\cdot, T, X', \cdot)$, saindo de alguma parada em $T$ e chegando em $X'$ no futuro. O loop no Algoritmo 2 procura repetidamente todas estas conexões e realiza operações de atualização que podem ser resumidas sob forma $\tau_{X'}(T + 1) = \min\{\tau_{X'}(T), T'_*\}$. Além do mais, é impossível ter duas conexões

partindo de $T$ e uma interferindo com a outra, senão a hipótese preliminar não seria observada. Assim, para $X'' \neq X'$, não é preciso usar $\tau_{X''}(T+1)$ para atualizar $\tau_{X'}(T+1)$ (ou, no máximo um valor que foi escolhido antes da conexão atual é usado) e, portanto, $\tau_{X'}(T+1)$ é mínimo.

Como $T$ é um inteiro positivo, isto é válido em todos os instantes. □

**Corolário 1.** *O Algoritmo 2 retorna o tempo de chegada mínimo em $B$ para uma consulta do tipo $(A\,@\,t, B)$.*

*Demonstração.* Aplicar a Proposição 1 e retornar $\tau_B(\infty)$. □

### 3.4.3 Critérios Adicionais

O CSA pode ser usado para resolver consultas de perfil ou com mais critério. Contudo, é preciso introduzir diversas modificações que acabam deteriorando o desempenho do algoritmo. Para minimizar simultaneamente o tempo de chegada e o número de transferências, é necessário alterar como o tempo de chegada em cada parada é armazenado.

Seja $K$ o número máximo de transferências que podem ser feitas para ir de $A$ até $B$. É possível chegar em cada parada com no mínimo 0 e no máximo $K$ mudanças de rota. Assim, $K+1$ valores de tempo de chegada são armazenados para cada parada $S$; o $k$-ésimo valor representa o tempo mínimo para atingir $S$ usando $k$ transferências. É necessário mudar o comportamento do Algoritmo 2 para que ele possa limitar o número de transferência e seja apto a usar os $K+1$ valores armazenados, além de mudar ligeiramente o critério de parada para evitar soluções Pareto-dominadas. No final da execução, em $B$ estarão disponíveis o tempo mínimo de chegada com diferentes números de transferências. Estas mudanças tornam esta variante do CSA mais lenta do que o próximo algoritmo na média, mas ainda assim bastante competitivo.

## 3.5 RAPTOR

Outro algoritmo que responde a consultas EAT é o RAPTOR (abreviação de *Round-bAsed Public Transit Routing*), desenvolvido pela Microsoft Research [19]. Ele é capaz de resolver este problema e simultaneamente retornar resultados com diferentes números de transferências, sendo até o instante o único método que otimiza dois critérios simultaneamente sem custo adicional. Ele também não se baseia em grafos para trabalhar e não precisa usar Dijkstra nem filas de prioridade. Contudo, seu funcionamento pode ser pensado como sendo uma BFS modificada, onde vértices são rotas e arestas transferências entre rotas.

Como visto anteriormente, uma rota pode ser definida como uma sequência de paradas. Desta forma, ao atingir qualquer ponto dentro da rota, é possível atingir todas as paradas subsequentes usando uma viagem. RAPTOR se resume a propagar o tempo de chegada das viagens a todas as paradas da mesma rota, até que o destino seja alcançado. Ele trabalha com a idéia de iterações (ou *rounds*) e a cada uma delas representa a realização de uma transferência.

### 3.5.1 Funcionamento

De maneira similar ao CSA, RAPTOR armazena para cada parada o tempo mínimo de chegada e como ela é realizada. Contudo, não há necessidade de conhecer a conexão, mas apenas a identificação da viagem que foi usada para alcançá-la (transferências são sinalizadas de forma diferente). Além disso, normalmente existem diversas formas de alcançar uma parada: umas com mais, outras com menos transferências. Desta forma, ele usa outra idéia presente no CSA e armazena um vetor de tempos de chegada $(\tau_0(p), \tau_1(p), \ldots)$, onde a $\tau_k(p)$ indica o tempo mínimo para alcançar a parada $p$ usando $k$ transferências.

Suponha que a consulta EAT seja $(A \,@\, t, B)$. Neste caso, é natural definir $\tau_0(A) = t$, e $\tau_0(p) = \infty$ para $p \neq A$. A versão mais simples do algoritmo trabalha em três etapas:

- *Relaxamento do Tempo*: se a iteração atual $k$ não é a primeira, $\tau_k(p) = \tau_{k-1}(p)$ para toda parada $p$;

- *Propagação de Viagens*: uma rota é servida por várias viagens e, logo, é preciso saber qual delas usar. Seja $p_1$ a primeira parada da rota $r$. A primeira viagem que pode ser usada na rota é aquela cujo tempo de partida de $p_1$ seja maior que $\tau_k(p_1)$. Esta rota é fixada e seus tempos de chegada são propagados para as paradas subsequentes, isto é, para cada parada $p_i$ em $r$, o tempo $\tau_k(p_i)$ é escolhido como sendo o mínimo entre o próprio valor e o tempo de chegada da viagem em $p_i$. Contudo, pode existir um $p_j$ tal que exista uma viagem que pode ser tomada antes da que foi usada em $p_1$, pois $\tau_k(p_j)$ pode ser suficientemente pequeno. Neste caso, a viagem que propagará o tempo é mudada a partir de $p_j$ para a que parte mais cedo e o mesmo procedimento é repetido até que a última parada da rota seja atingida;

- *Transferências*: todos os caminhos a pé são explorados. Suponha que exista um caminho entre $p_u$ e $p_v$ que demore $\ell_{uv}$ para ser percorrido. Neste caso, os tempos de chegada nos dois são atualizados: $\tau_k(p_u) = \min\{\tau_k(p_u), \tau_k(p_v) + \ell_{uv}\}$ e o análogo é feito para $\tau_k(p_v)$.

O custo de cada iteração é linear no tamanho das rotas, das viagens e dos caminhos a pé [19]. Entretanto, existe ainda espaço para simples otimizações deste método. Certamente nem todas as rotas necessitam ser exploradas; apenas as que tiveram alguma parada com tempo de chegada melhorado na iteração anterior. Caso isto seja desconsiderado, o mesmo cálculo de iterações passadas será repetido em vão. Isto é implementado marcando paradas que sofreram melhoria de tempo em cada iteração e criando uma lista com as rotas que incidem nas mesmas.

Outra otimização possível consiste em evitar caminhos Pareto-dominados. Seja $\tau^*(p)$ o tempo mínimo de chegada em $p$. Na iteração $k$, não faz sentido atualizar $\tau_k(p)$ para valores maiores que $\tau^*(p)$. Mais do que isto, atualizar tempos de chegada para valores maiores que $\tau^*(B)$ é totalmente desnecessário (onde $B$ é a parada de

destino). Estas duas constatações reduzem o custo da propagação de viagens.

### 3.5.2 Paralelismo

A propagação de viagens é feita quase independentemente para cada rota. Assim, é natural pensar em paralelizar este processo para atingir tempos de consulta menores. Contudo, a atualização do tempo de chegada numa mesma parada pode ser feita por diferentes rotas simultaneamente. Desta forma, condições de corrida podem ocorrer e até mesmo deadlocks. Uma alternativa é criar um grafo de conflito de rotas: cada vértice representa uma rota e dois nós possuem uma aresta caso as rotas que representam compartilhem uma parada. Dividir o conjunto de rotas em $m$ processadores é equivalente a colorir este grafo de conflito e tratar simultaneamente rotas de mesma cor. Os autores realizam a coloração de forma gulosa - posto que encontrar o número cromático do grafo é NP-completo - e explicam uma estratégia para poder tratar com segurança rotas de mesma cor simultaneamente.

## 3.6 Transfer Patterns

Transfer Patterns é o algoritmo usado pela Google para consultas em redes de transporte público [6] e implementado no *Google Maps*. A intuição por trás do algoritmo se encontra no fato que vários caminhos entre paradas $A$ e $B$ compartilham rotas e transferências entre rotas. Um *padrão de transferência* (do inglês *transfer pattern*) entre $A$ e $B$ é definido como uma sequência de rotas e transferências (entre duas paradas) que pode ser usada para atingir o destino. Por exemplo, um padrão de transferência entre o Méier e a Ilha do Fundão seria a construído usando duas rotas e uma transferência em Del Castilho, enquanto o padrão de transferência de Botafogo para a mesma destinação usa apenas uma rota de ônibus. O dois padrões possuem comprimento 2 e 1, respectivamente.

O algoritmo possui uma fase de pré-processamento bastante pesada onde padrões de transferência entre todos os pares de paradas são calculados. Na prática, isto é

computacionalmente inviável, pois usaria milhões de horas de cálculo e teriam um alto custo de armazenamento. A solução encontrada foi procurar nós importantes, os *hubs*, e calcular apenas os padrões de transferência de toda parada até os hubs e dos hubs para todas as paradas. Esta idéia é comum a um algoritmo de busca para redes de transporte comuns, conhecido como *Transit Node Routing* [9]. Na vida real, um usuário de transporte público não deseja viagens muito longas, com mais de 3 ou 4 transferências. Assim, uma heurística de caminhos de comprimento 3 (*3-leg heuristic*) é aplicada: nenhum padrão de transferência entre paradas comuns e hubs pode ter mais do que 2 transferências (ou seja, 3 viagens). Isto garante que os padrões de transferência tenham no máximo 6 viagens, pois eles serão a combinação de um padrão de transferência de $A$ até um hub e de um hub até $B$.

Para realizar a consulta, um DAG é criado a partir dos padrões de transferência considerados: paradas de transferência são vértices e se conectam caso exista uma viagem entre elas. Uma busca reversa por viagens é, então, realizada a partir das paradas de transferências que se ligam diretamente ao destino através de uma rota. Mais detalhes podem ser obtidos em [6].

Atualmente, Transfer Patterns é o método mais rápido para tratar problemas em redes de transporte público, respondendo a consultas EAT e mais complicadas em pouquíssimos milissegundos, mesmo para redes enormes, como a de Londres. Contudo, ele não é aplicável para redes continentais, tendo em vista que o tempo de pré-processamento para grandes metrópoles, como Londres, já chega a milhares de horas (distribuído nas máquinas do cluster da Google).

# Capítulo 4

# Software para Redes de Transporte com Dependência Temporal

Este capítulo apresenta uma breve descrição do software desenvolvido para encontrar caminhos em redes de transporte público com dependência temporal (*transit networks*). Para fins comparativos, duas formas de resolução são usadas: uma com **Dijkstra** sobre a *Rede de Dependência Temporal* e outra com o **CSA**, trabalhando sobre uma forma simplificada da *Rede de Tempo Expandido*.

No primeiro caso, o algoritmo de Dijkstra encontra o caminho mínimo entre duas paradas sem considerar atributos adicionais (consulta EAT). Para adicionar mais critérios, seria necessário considerar a variante *multi-label* ou *layered* do algoritmo [19], onde cada vértice pode ser explorado diversas vezes. Estes métodos são em geral lentos e não serão abordados aqui.

Para o CSA, consultas do tipo EAT+TR (tempo mínimo otimizando o número de transferências) serão realizadas. Repare que neste caso, o resultado encontrado não é único, mas ainda assim pequeno. Além disso, como discutido no capítulo anterior, uma pequena modificação deve ser introduzida no algoritmo original, onde agora a idéia de tempo mínimo de chegada numa parada é substituída por um vetor de tempos mínimos de chegada (cada valor representando um determinado número de transferências). Mesmo que estas alterações aumentem o tempo de cálculo do

CSA, observou-se que ele ainda apresenta tempos de resposta competitivos com o algoritmo de Dijkstra puro (sem nenhum limitante no número de transferências).

As próximas seções abordam o formato de dados de entrada (e o dataset usado como referência), além de explicar brevemente a modelagem usada no desenvolvimento no software. Em seguida, são mostrados alguns problemas com os dados que explicam por vezes resultados ruins. Por fim, resultados de consulta são mostrados para a rede utilizada, com o uso de critérios quantitativos e qualitativos de comparação dos dois algoritmos, além da construção de uma interface web para exibição do resultados de consultas.

## 4.1 Formato GTFS

O *General Transit Feed Specification* (GTFS) é um padrão de armazenamento de dados para redes de transporte público desenvolvido e mantido pela Google [1]. Trata-se do formato usado por diversos softwares de cálculo de rota, inclusive pelo Google Maps, além de ser usado por órgãos governamentais de diversas cidades para publicar dados sobre o serviço de transporte público das mesmas. Um *feed* do transporte de uma cidade é composto de até uma dúzia de arquivos, dos quais os mais importantes são os seguintes:

- *stops*: coordenadas das paradas e seus respectivos nomes;
- *routes*: nome das rotas que servem a cidade;
- *trips*: nome das viagens e das rotas das quais são instância, além do serviço, isto é, quando as viagens operam;
- *calendar*: serviços e dias de funcionamento dos mesmos;
- *transfers*: transferências válidas entre paradas, com o tempo necessário para percorrer a distância;
- *stop times*: eventos de chegada e partida das viagens nos pontos de parada.

[1] https://developers.google.com/transit/gtfs/

Diversas fontes de arquivos GTFS são disponíveis online.[2,3] Para este trabalho, foi considerada a rede de região metropolitana de Paris[4], pois a mesma possui tamanho equivalente a redes usadas em outros estudos, como a da cidade de Londres - cujo formato de dados é diferente -, além de apresentar o maior número de caminhos de pedestre dentre todas as redes disponíveis.

## 4.2 Modelagem

O protótipo para realizar os testes dos algoritmos foi escrito em C++. Um diagrama de classes simplificado é apresentado na Figura 4.1. A classe *Parser* é responsável pela leitura de arquivos no formato GTFS e criação de estruturas auxiliares, representando paradas, rotas, viagens, conexões e caminhos. Uma classe abstrata *Solver* possui métodos para resolução de consultas, retorna objetos do tipo *Solution* e é construída a partir de um objeto *Parser*. Finalmente, a classe *TD_Dijkstra* implementa a resolução de consultas EAT em Redes de Dependência Temporal usando Dijkstra, enquanto a classe *CSA* resolve consultas EAT+TR usando o algoritmo de mesmo nome.

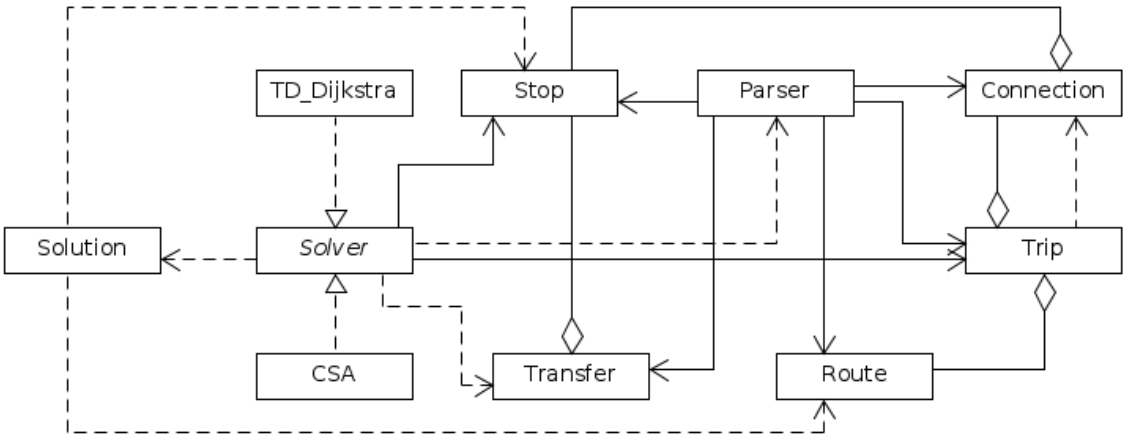



Figura 4.1: Diagrama de Classes Simplificado

O protótipo criado é multiplataforma e não depende de nenhuma biblioteca externa à STL em sua implementação original. Outras técnicas de resolução podem

[2]https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds
[3]http://www.gtfs-data-exchange.com/
[4]https://www.data.gouv.fr/en/datasets/offre-transport-de-la-ratp-format-gtfs-ratp/

ser adicionadas com a simples adição de classes que implementam *Solver*, sem a necessidade de mudar nenhuma outra parte do código.

### 4.2.1 Consultas Aleatórias

Quando consultas são feitas de forma aleatória, é provável que duas paradas pouco movimentadas - isto é, com poucas viagens servindo - sejam escolhidas. Neste caso, o número de transferências tende a ser grande - o que tende a piorar ainda mais devido ao número insuficiente de caminhos a pé - ou torna-se mesmo impossível de encontrar uma resposta válida, tendo em vista que o número máximo de transferências foi limitado a 5 para o CSA. Contudo, na vida real as pessoas realizam consultas onde pelo menos uma das paradas é bastante movimentada, o que torna a escolha aleatória irrealista. Uma idéia possível é tornar a probabilidade de escolher uma parada proporcional ao número de eventos (conexões) que passam por ela, facilitando assim, a escolha de *hubs*.

### 4.2.2 Interface Web

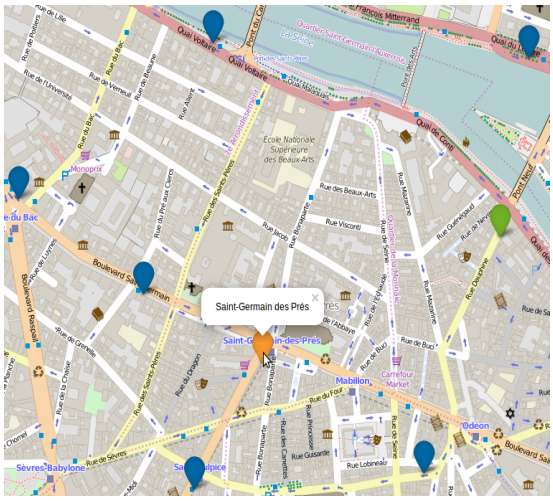
Uma interface Web foi desenvolvida para exibir os resultados para os usuários. O cliente foi criado usando *Leaflet*[5] com *OpenStreetMaps*[6] para a exibição de mapas. A primeira é uma biblioteca em JavaScript usada para manipulação fácil e leve (geralmente usada em aplicações para smartphones) de mapas, permitindo a inserção de símbolo e desenho de polígonos e caixas de texto, entre outros. Contudo, é preciso acoplar uma *engine* de mapas a ela para que a exibição ocorra corretamente. Existem muitas opções disponíveis, mas a maioria delas é paga; *OpenStreetMaps* é uma opção gratuita usada como alternativa ao Google Maps em muitas aplicações.

Para realizar a comunicação com o servidor descrito anteriormente, foi usado o *Mongoose*[7], um web server que pode ser acoplado a programas em C/C++ de maneira muito simples através de uma biblioteca (é necessária apenas a compilação

[5]http://leafletjs.com
[6]http://www.openstreetmap.org
[7]https://github.com/cesanta/mongoose

de um código-fonte em C). Mesmo não sendo muito estável, é uma ótima alternativa para protótipos ou projetos pequenos. Toda a comunicação é feita entre cliente e servidor usando o padrão *JSON* (JavaScript Object Notation), com o uso da biblioteca *rapidjson*[8] do lado do servidor, que é apresentada apenas sob a forma de headers e, portanto, não precisa nem de compilação!

Por fim, o código no cliente foi escrito em JavaScript, usando JQuery e Ajax. Para melhorar a formatação, *JQuery UI*[9] foi usado para os formulários e o plugin *Leaflet awesome-markers*[10] para a exibição de marcadores estilizados no mapa.




(a) Paradas aleatórias, dentre as quais duas selecionadas


(b) Resultado de uma consulta

Figura 4.2: Interface Web desenvolvida para exibir resultados das consultas

Ao usuário são exibidos diversos pontos de parada aleatórios, como mostra a Figura 4.2a. Ele deve selecionar dois e escolher a hora de partida do ponto de origem. Em seguida, ele envia uma requisição que é tratada no servidor, retornando o conjunto de paradas, rotas e transferências que respondem a uma consulta, desenhando o resultado de forma semelhante à Figura 4.2b. A todo momento o usuário pode pedir novos pontos, que substituirão os já existentes.

[8] https://github.com/miloyip/rapidjson
[9] http://jqueryui.com/
[10] https://github.com/lvoogdt/Leaflet.awesome-markers

### Comunicação

O formato dos dados para comunicação não é apresentado neste texto, sendo apenas descritos os tipos de comunicação que existem entre cliente e servidor:

1. Cliente pede as coordenadas (latitude e longitude) do ponto central do mapa que será exibido. Servidor responde com o ponto correspondendo à media de todas as latitudes e todas as longitudes das paradas da rede;

2. Cliente pede novos pontos de parada. Servidor recupera coordenadas do canto superior esquerdo e canto inferior direito; ele retorna pontos que se encontram no retângulo cuja diagonal é formada por estes dois;

3. Cliente realiza uma consulta EAT+TR entre duas paradas exibidas na tela. Servidor responde com a solução encontrada usando CSA que possui o número mínimo de transferências.

### Pontos de uma Região

A escolha dos pontos de uma região é em geral custosa, pois precisa percorrer todas as paradas para se descobrir quais pertencem à região em questão. Uma solução seria usar uma estrutura geométrica para armazenar os pares de latitude e longitude, como KD-trees. Contudo, no pior caso todos os pontos pertencerão à região e percorrer estruturas em árvore tornam-se mais custosas que varrer um vetor de paradas.

A solução encontrada foi ordenar todas as paradas por latitude e armazená-las em um vetor. Para saber quais pontos se encontram em uma faixa de latitude, usa-se uma busca binária para achar o limite inferior e o superior. Em seguida, todas as paradas deste intervalo têm a longitude verificada e as que pertencerem ao intervalo correto são direcionadas para a tiragem aleatória que definirá quais serão enviadas para o cliente e aparecerão no mapa.

## 4.3 Resultados

Para comparar os dois algoritmos de resolução, foram feitas 50 mil consultas entre vértices escolhidos totalmente ao acaso com horários de partida entre 8:00 e 18:00. O código foi compilado usando g++ versão 4.6.3 para C++11 usando flag de compilação -O2 para otimização. Os resultados apresentados foram obtidos num computador com as seguintes configurações: AMD FX 4100 Quad-Core, 4GB de RAM e sistema operacional Linux 64 bits, distribuição Ubuntu 12.04. Todas as consultas foram rodadas em um único core. O algoritmo CSA foi limitado a 5 transferências, enquanto Dijkstra não possuía nenhum valor limitante.

Como descrito anteriormente, a rede usada foi a dos transportes públicos da região metropolitana de Paris. Por comodidade, todos os resultados foram limitados a um único dia (13/01/2015). Os dados em geral são fixos para cada dia da semana e a maior parte dos *benchmarks* são feitos usando apenas um dia. A rede apresenta 26.906 paradas, ligadas por 80.443 caminhos de pedestre. Existem ainda 1.852.160 conexões, com 76.719 viagens de 1.072 rotas distintas.

Para medir a qualidade dos métodos e da base de dados, foi calculado o número de acertos, correspondendo à quantidade de vezes que o algoritmo encontrou o ponto de parada de destino. No mais, o tempo de cálculo é medido para todas as consultas que geraram uma resposta. Para o CSA, uma medida de interesse a mais é calculada: a quantidade de conexões percorridas quando a resposta é encontrada; a parada pode não ser alcançável em determinado horário. Para Dijkstra, são medidos o número médio de vértices encontrados e visitados. Por fim, para ajudar o CSA, o tempo máximo de viagem foi restrito a 2 horas e a busca sempre pára quando os tempos de partida das conexões são 30 minutos maiores que o melhor tempo de chegada no destino. Os valores médios das quantidades descritas são mostrados na Tabela 4.1.

Após analisar os valores dos experimentos, fica claro que limitar o número de transferências não altera muito a taxa de acerto do CSA em comparação ao Dijkstra. Num primeiro momento fica a impressão que rodar Dijkstra é muito melhor, pois

| | CSA (EAT+TR) | Dijkstra (EAT) |
|---|---|---|
| Respostas Válidas | 75,7% | 76,1% |
| Tempo (ms) | 79,1 | 6,7 |
| Conexões Visitadas | 191.881,1 | - |
| Vértices Encontrados | - | 12.473,8 |
| Vértices Visitados | - | 11.269,5 |

Tabela 4.1: Resultados de 50 mil execuções de Dijkstra e CSA

a diferença de tempo de execução é de quase 12 vezes. Contudo, é preciso lembrar que os tipos de consulta são diferentes: enquanto Dijkstra roda uma consulta EAT, só minimizando o tempo de chegada, CSA roda EAT+TR, onde o número de transferências também é minimizado. Para ter uma breve idéia da diferença de cenários, o processamento para cada conexão no CSA é 6 vezes maior que o processamento para cada vértice visitado no Dijkstra, desconsiderando a atualização da fila de prioridade. Isto porque CSA necessita atualizar os tempos de chegada da parada para diferentes números de transferências. Mais ainda, como visto, o número de conexões exploradas é quase 16 vezes maior que o número de vértices visitados. Finalmente, fica claro que Dijkstra tem que visitar em média uma grande parte do conjunto de vértices (quase a metade!) para chegar à resposta, enquanto CSA precisa explorar pouco mais de 10% das conexões.

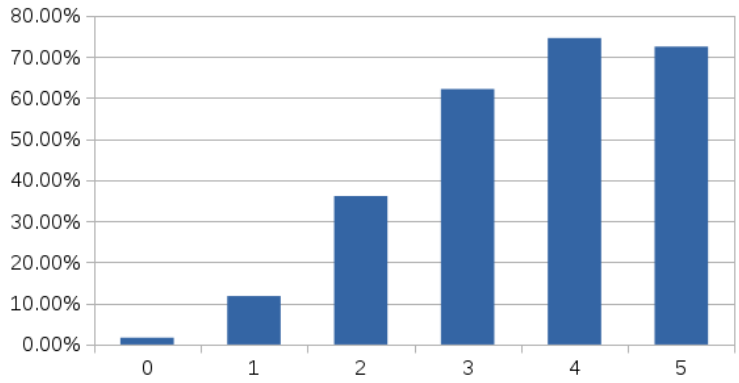



Figura 4.3: Porcentagem de soluções não dominadas encontradas pelo CSA

Para ter uma idéia da qualidade das soluções, foram calculados o número médio de transferências nos dois casos. Como isto não faz muito sentido para o CSA, foi armazenado o número de soluções não dominadas para diferentes números de

transferência. A Figura 4.3 mostra estes valores. Repare que pouquíssimas soluções são encontradas sem nenhuma transferência, ao passo que em uma grande quantidade de vezes, soluções Pareto-ótimas com 4 ou 5 transferências são achadas. Isto é causado em parte pela insuficiência dos caminhos a pé disponibilizados pela rede. Por exemplo, foi observado com a ajuda da interface web que por diversas vezes era necessário usar mais uma linha de ônibus para percorrer 300 metros ou menos, o que é impensável na vida real. Da mesma forma, Dijkstra encontra soluções com uma média de 5,3 transferências, o que já era esperado, tendo em vista que apenas a minimização de tempo ocorre. É importante notar que CSA encontra muitas vezes soluções com 2 ou 3 transferências, ou seja, na média as soluções com menos transferência encontradas são melhores que as de Dijkstra.

## 4.4 Problemas

Diversos problemas foram encontrados ao longo do desenvolvimento deste software. Talvez o mais importante seja o fato de viagens com sequências diferentes de paradas serem mapeadas na mesma rota. Isto impossibilitou a implementação do RAPTOR, pois este precisa que todas as viagens de uma mesma rota sejam exatamente iguais no que diz respeito à sequência de estações. Um jeito de contornar este problema seria definir diversas "formas" de representação para uma mesma rota, o que certamente comprometeria o desempenho do algoritmo citado.

Outro grande problema se deve ao fato de as transferências entre paradas presentes no arquivo do GTFS não serem suficientes. Assim, consultas entre duas paradas a 300 metros de distância podem retornar respostas com 2 ou mesmo 3 transferências! Uma alternativa seria criar caminhos a pé artificias: para cada par de paradas, se eles estiverem a uma distância menor que $d$, adicionar um caminho conectando os dois, cujo tempo de percurso seria dado de acordo com a velocidade média de caminhada (em geral, 4 km/h).

# Capítulo 5

# Conclusão

Analisando os resultados do capítulo anterior, é possível concluir que o CSA tem um "alcance" maior que Dijkstra para redes de transporte público. Se versões multi-critério deste algoritmo tivessem sido implementadas, certamente o tempo das consultas atingiria 1 segundo. Além do mais, o modelo usado é a rede de dependência temporal em sua variante mais simples, que é pouco adequada para problemas reais. Finalmente, o algoritmo de Dijkstra trabalhava indiscriminadamente, realizando muitas vezes transferências desnecessárias.

Decerto CSA não é a melhor opção para consultas EAT+TR, pois RAPTOR e Transfer Patterns possuem resultados melhores. Ainda assim ele se mostrou bastante interessante para aplicações reais, além de ser fácil de entender e não necessitar de filas de prioridade nem de outras estruturas de dados mais complexas.

## Problemas com o Padrão GTFS

Como mencionado no capítulo anterior, o padrão GTFS apresenta alguns problemas. O principal deles é a possibilidade de existir variações de rota dentro de uma cidade (paradas são puladas em determinadas viagens, por exemplo). Uma solução seria alterar o arquivo *routes* de forma que a lista de paradas fosse disponibilizada mais facilmente para o desenvolvedor; variantes de uma mesma rota seriam guardadas com o mesmo nome, mas com identificação diferente. Isto facilitaria a

implementação do algoritmo RAPTOR, além de outros métodos ainda a surgir.

Seria interessante também se este padrão apresentasse mais caminhos a pé entre pares de paradas, pois os dados atuais são insuficientes e causam aumentos desnecessários no número de transferências a serem empregadas. Criar caminhos artificiais não é uma solução muito correta, pois desconsidera aspectos geográficos que separam os pares de paradas (subidas, viadutos, pontes, etc).

## Estado da Arte

A fronteira para redes de transporte público se encontra na criação de algoritmos que possam tratar diferentes centros urbanos simultaneamente, por exemplo, Rio de Janeiro e São Paulo, dizendo como chegar de um endereço na primeira cidade a outro na última usando transporte público local e linhas de longa distância (ônibus interestaduais e mesmo avião). Outro problema importante e bastante complicado é como unir transporte público e privado [2, 17, 26], onde táxis e carros particulares podem ser usados em parte do caminho.

Este trabalho de conclusão de curso visou apenas ao estudo de questões mais simples em redes de transporte com ênfase em transporte público, mas abre portas para o autor explorar novos horizontes e, eventualmente, atacar estes problemas citados no futuro. Leitores mais interessados encontrarão nas Referências Bibliográficas artigos e trabalhos sobre assuntos do momento [7, 9, 24], além de métodos e algoritmos que não foram abordados aqui.

# Referências Bibliográficas


[1] ABRAHAM, I., DELLING, D., GOLDBERG, A. V., et al., 2011, "A hub-based labeling algorithm for shortest paths in road networks". In: *Experimental Algorithms*, Springer, pp. 230–241.

[2] AYED, H., GALVEZ-FERNANDEZ, C., HABBAS, Z., et al., 2011, "Solving time-dependent multimodal transport problems using a transfer graph model", *Computers & Industrial Engineering*, v. 61, n. 2, pp. 391–401.

[3] BAST, H. "Course "Efficient Route Planning" (Summer 2012) - University of Freiburg". `http://ad-wiki.informatik.uni-freiburg.de/teaching/EfficientRoutePlanningSS2012`. Accessed: 21 January 2015.

[4] BAST, H., STORANDT, S., 2013, "Frequency Data Compression for Public Transportation Network Algorithms". In: *Sixth Annual Symposium on Combinatorial Search*.

[5] BAST, H., STORANDT, S., 2014, "Flow-Based Guidebook Routing." In: *ALENEX*, pp. 155–165. SIAM.

[6] BAST, H., CARLSSON, E., EIGENWILLIG, A., et al., 2010, "Fast routing in very large public transportation networks using transfer patterns". In: *Algorithms–ESA (European Symposium on Algorithms) 2010*, Springer, pp. 290–301.

[7] BAST, H., BRODESSER, M., STORANDT, S., et al., 2013, "Result diversity for multi-modal route planning". In: *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems-2013*, v. 33, pp. 123–136, .

[8] BAST, H., STERNISKO, J., STORANDT, S., et al., 2013, "Delay-Robustness of Transfer Patterns in Public Transportation Route Planning". In: *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems-2013*, v. 33, pp. 42–54, .

[9] BAST, H., DELLING, D., GOLDBERG, A., et al., 2014, "Route Planning in Transportation Networks", .

[10] BAST, H., FUNKE, S., MATIJEVIC, D., 2009, "Ultrafast shortest-path queries via transit nodes", *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, v. 74, pp. 175–192.

[11] BATZ, G. V., DELLING, D., SANDERS, P., et al., 2009, "Time-Dependent Contraction Hierarchies." In: *ALENEX*, v. 9. SIAM.

[12] BRODESSER, M., 2013, "Multi-Modal Route Planning", *Dissertação de Mestrado, Albert-Ludwigs-Universität Freiburg*.

[13] COFFEY, C., NAIR, R., PINELLI, F., et al., 2012, "Missed connections: quantifying and optimizing multi-modal interconnectivity in cities". In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pp. 26–32. ACM.

[14] DELLING, D., 2011, "Time-dependent SHARC-routing", *Algorithmica*, v. 60, n. 1, pp. 60–94.

[15] DELLING, D., PAJOR, T., WAGNER, D., 2009, "Accelerating multi-modal route planning by access-nodes". In: *Algorithms-Esa 2009*, Springer, pp. 587–598.

[16] DELLING, D., KATZ, B., PAJOR, T., 2012, "Parallel computation of best connections in public transportation networks", *Journal of Experimental Algorithmics (JEA)*, v. 17, pp. 4.

[17] DELLING, D., DIBBELT, J., PAJOR, T., et al., 2013, "Computing multimodal journeys in practice". In: *Experimental Algorithms*, Springer, pp. 260–271, .

[18] DELLING, D., GOLDBERG, A. V., WERNECK, R. F., 2013, "Hub Label Compression." In: *SEA*, pp. 18–29. Springer, .

[19] DELLING, D., PAJOR, T., WERNECK, R. F., 2013, "Round-based public transit routing". In: *Sixth Annual Symposium on Combinatorial Search*, .

[20] DIBBELT, J., PAJOR, T., STRASSER, B., et al., 2013, "Intriguingly simple and fast transit routing". In: *Experimental Algorithms*, Springer, pp. 43–54.

[21] DIBBELT, J., STRASSER, B., WAGNER, D., 2014, "Customizable Contraction Hierarchies", *arXiv preprint arXiv:1402.0402*.

[22] GALVEZ-FERNANDEZ, C., KHADRAOUI, D., AYED, H., et al., 2009, "Distributed approach for solving time-dependent problems in multimodal transport networks", *Advances in Operations Research*, v. 2009.

[23] GEISBERGER, R., 2010, "Contraction of timetable networks with realistic transfers". In: *Experimental Algorithms*, Springer, pp. 71–82.

[24] GEISBERGER, R., 2011, *Advanced route planning in transportation networks*. Tese de Doutorado, Karlsruhe Institute of Technology, Alemanha.

[25] GEISBERGER, R., SANDERS, P., SCHULTES, D., et al., 2008, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks". In: *Experimental Algorithms*, Springer, pp. 319–333.

[26] KIRCHLER, D., 2013, *Efficient routing on multi-modal transportation networks*. Tese de Doutorado, Ecole Polytechnique, França.

[27] KÖHLER, E., MÖHRING, R. H., SCHILLING, H., 2006, "Fast point-to-point shortest path computations with arc-flags", *9th DIMACS Implementation Challenge*.

[28] MÖHRING, R. H., SCHILLING, H., SCHÜTZ, B., et al., 2005, "Partitioning graphs to speed up Dijkstra's algorithm". In: *Experimental and efficient algorithms*, Springer, pp. 189–202.

[29] NANNICINI, G., DELLING, D., LIBERTI, L., et al., 2008, "Bidirectional A$*$ search for time-dependent fast paths". In: *Experimental Algorithms*, Springer, pp. 334–346.

[30] NEWMAN, M., 2010, "Networks: an introduction". pp. 185–193, Oxford University Press.

[31] PYRGA, E., SCHULZ, F., WAGNER, D., et al., 2008, "Efficient models for timetable information in public transportation systems", *Journal of Experimental Algorithmics (JEA)*, v. 12, pp. 2–4.

[32] STRASSER, B., 2012, *Delay-Robust Stochastic Routing in Timetable Networks*. Tese de Doutorado, Karlsruhe Institute of Technology, Alemanha.

[33] STRASSER, B., WAGNER, D., 2014, "Connection Scan Accelerated." In: *ALENEX*, pp. 125–137. SIAM.