Relatório Técnico — MC558
Comparação de Algoritmos de Caminho Mínimo

Andrey Torres de Lima - 231442

2025

1 Introdução

Neste relatório, apresento uma comparação prática entre diferentes algoritmos clássicos de caminho mínimo, incluindo abordagens bidirecionais e heurísticas de bucket (Dial). Além disso, uma modelagem via Programação Linear usando o solver Gurobi foi usada como baseline de precisão.

Os algoritmos analisados foram:

- Dijkstra (clássico)
- Dijkstra Bidirecional
- Dial (Dijkstra com buckets)
- Dial Bidirecional
- Programação Linear (modelo de fluxo) usando Gurobi

Executei todos os métodos sobre as 10 instâncias do conjunto de testes, comparando custos e tempos de execução.

2 Modelagem via Programação Linear

O modelo que utilizei foi o clássico modelo de fluxo unitário:

$$\text{minimizar} \quad \sum_{(i,j)\in E} w_{ij}\, x_{ij}$$

$$\text{sujeito a} \quad \sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = \begin{cases} 1 & \text{se } i = s, \\ -1 & \text{se } i = t, \\ 0 & \text{caso contrário,} \end{cases}$$

$$x_{ij} \in \{0, 1\}.$$

Também testei a formulação relaxada por potenciais (LP), principalmente para depurar resultados mais rapidamente, já que a versão inteira pode explodir em instâncias grandes. Confesso que apanhei várias vezes com o Gurobi até estabilizar o modelo e montar um parser que funcionasse para todas as entradas.

3 Resultados de Caminho Mínimo

Todos os algoritmos retornaram exatamente os mesmos custos para as 10 instâncias, o que me deu confiança de que as implementações estavam corretas:

Arquivo	Custo Mínimo
arq01	44.000
arq02	43.000
arq03	17.000
arq04	21.883
arq05	31.388
arq06	48.787
arq07	839.000
arq08	853.000
arq09	794.000
arq10	943.510

Essa concordância entre PL e métodos clássicos foi essencial para identificar erros na entrada e na leitura baseados em índices.

4 Medições de Tempo

Para os algoritmos em C++, cada execução foi repetida três vezes por instância. Usei `date +%s.%N` para medir o tempo com precisão de nanossegundos (dica que descobri enquanto apanhava para o Dial).

Para o Gurobi (`plShortestPath.py`), medi apenas uma execução por instância, porque rodar o solver três vezes nos casos grandes seria inviável (chegou a dar mais de minuto por execução nas últimas entradas). Aqui eu realmente apanhei do Gurobi: só para configurar os parâmetros e evitar a poluição das mensagens de licença já gastei mais tempo que o desejado.

4.1 Resumo (C++ – amostra)

Abaixo segue uma amostra da tabela final de tempos (segundos):

Algoritmo /Variação	**Arquivo**	**Tempo Médio (s)**	
		Dial	**Dijkstra**
Clássico	arq01	0.0059	0.0020
Bidirecional	arq01	0.0026	N/A
Clássico	arq02	0.0019	0.0016
Bidirecional	arq02	0.0019	0.0017
Clássico	arq03	0.0017	0.0019
Bidirecional	arq03	0.0019	0.0020
Clássico	arq04	0.0022	0.0021
Bidirecional	arq04	0.0024	0.0021
Clássico	arq05	0.0019	0.0017
Bidirecional	arq05	0.0021	0.0026
Clássico	arq06	0.0022	0.0020
Bidirecional	arq06	0.0021	0.0022
Clássico	arq07	0.2247	0.2345
Bidirecional	arq07	0.2662	0.2647
Clássico	arq08	0.3438	0.3388
Bidirecional	arq08	0.4157	0.3299
Clássico	arq09	0.2651	0.2428
Bidirecional	arq09	0.2820	0.2401
Clássico	arq10	0.3108	0.3167
Bidirecional	arq10	0.3772	0.2913

Table 1: Tempos médios (s) dos algoritmos Dial e Dijkstra, Clássico e Bidirecional, para os 10 arquivos.

4.2 Tempos medidos para o PL (Gurobi)

Como mencionei, não repeti três vezes porque o solver demorou muito nas últimas instâncias. Eis os tempos:

Arquivo	Tempo (s)
arq01	0.07
arq02	0.07
arq03	0.07
arq04	0.07
arq05	0.07
arq06	0.07
arq07	47.12
arq08	96.86
arq09	60.93
arq10	95.34

Table 2: Tempos de execução do script `plShortestPath.py` (Gurobi).

Observações pessoais

- As instâncias pequenas mal dão tempo do Gurobi "respirar", então todas caíram em torno de 0.07s.
- Já nas grandes, o solver sofre. Aqui eu realmente percebi a diferença entre usar algoritmos especializados e deixar o solver resolver tudo na força bruta.
- Considerando o esforço, definitivamente o PL vale apenas para depuração e checagem — jamais para produção.

5 Discussão e dificuldades encontradas

Algumas dificuldades apareceram no caminho:

- **Indexação 0-based vs 1-based:** tive que revisar a leitura porque os arquivos não são consistentes.
- **Mensagens do Gurobi:** precisei suprimir licença, username e logs para não poluir a saída.
- **Formulações diferentes:** a formulação MIP explode rápido; a LP é bem mais amigável e funcionou perfeitamente para validação.
- **Precisão das medições:** timestamps em nanossegundos foram essenciais para diferenciar execuções em instâncias pequenas.
- **Debug do Dial:** a implementação Dial foi a que mais me tomou tempo para validar; erros mínimos de bucket fazem o algoritmo ir para $O(V^2)$ na prática.

6 Conclusão

Os resultados confirmam que as implementações baseadas em Dijkstra (clássico e variações) são extremamente rápidas e adequadas para praticamente todos os cenários reais. O solver Gurobi provou ser útil como baseline e para garantir correção, mas é inviável para uso prático em instâncias grandes, especialmente quando o tempo importa.

Possíveis extensões: medir memória, analisar o comportamento do Dial em grafos com pesos concentrados, testar versões paralelas e avaliar heurísticas para instâncias ainda maiores.