BAB 5
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Berdasarkan hasil implementasi dan pengujian prototipe aplikasi pencarian rute Transjakarta menggunakan algoritma Dijkstra berbasis *framework* Laravel yang telah diuraikan pada Bab IV, dapat ditarik kesimpulan sebagai berikut:

1. Rancang bangun prototipe aplikasi berbasis GTFS berhasil dilakukan dengan mengintegrasikan data Transjakarta secara lengkap dan tervalidasi. Sistem berhasil mengimpor dan mengintegrasikan data GTFS Statis Transjakarta yang terdiri dari 243 rute (koridor), 8.651 halte, 650 perjalanan (*trip*), 218.834 titik *shape*, dan 24.568 *stop times*. Selain data GTFS, sistem juga mengintegrasikan 32.000+ data *Point of Interest* (POI) dari OpenStreetMap yang tersebar di wilayah Jabodetabek. Proses validasi data GTFS menggunakan Canonical GTFS Schedule Validator berhasil menurunkan total notifikasi dari 1.039 menjadi 52, dengan seluruh 47 *error* berhasil dieliminasi dan jumlah *warning* berkurang 96% (dari 964 menjadi 38). Data yang telah dibersihkan menggunakan Python dengan *library* Pandas melalui 7 tahapan pembersihan (perbaikan masa berlaku *service_id*, pembuatan *feed_info.txt*, konversi nama halte ke *Mixed Case*, normalisasi tipe data integer, penanganan *shape_dist_traveled* duplikat, sinkronisasi jarak *trip-shape*, dan penghapusan 227 halte tidak terpakai) selanjutnya diimpor ke *database* MySQL dan disimpan dalam *Cache* selama 24 jam melalui GtfsCacheService untuk mempercepat *loading* halaman peta.

2. Penerapan algoritma Dijkstra berhasil menghasilkan rute optimal dengan dua preferensi pencarian (jarak terpendek dan minim transfer) serta terbukti efisien secara komputasi. Algoritma Dijkstra diimplementasikan pada dua lapisan yaitu lapisan *frontend* JavaScript sebagai implementasi utama yang berjalan di *browser* pengguna, dan lapisan *backend* PHP melalui NavigasiService untuk keperluan *logging* dan pengujian API. Sistem menyediakan dua preferensi pencarian dengan parameter bobot yang berbeda yaitu Cari 1 (Prioritas Jarak Terpendek) dengan penalti transfer 2.500 m dan bobot naik bus 800 m, serta Cari 2 (Prioritas Minim Transfer) dengan penalti transfer 50.000 m dan bobot naik bus 600 m. Berdasarkan pengujian pada rute Blok M – Tosari, Cari 1 menghasilkan jarak 5,82 km dengan waktu eksekusi 1.551 ms, sedangkan Cari 2 menghasilkan jarak 6,98 km dengan waktu eksekusi 1.535 ms. Rata-rata dari 6 skenario pengujian (3 halte ke halte

dan 3 POI ke POI), Cari 2 berhasil mengurangi jumlah koridor hingga 43% (dari 5,3 menjadi 3,0 koridor) dengan tambahan jarak hanya 0,2% (22,11 km vs 22,16 km). Algoritma Dijkstra dengan kompleksitas $O((V + E)\ log\ V)$ mampu memproses graf dengan 8.651 simpul dan 55.000+ sisi dalam waktu rata-rata kurang dari 2 detik, membuktikan efisiensinya untuk aplikasi pencarian rute transportasi publik. Analisis regresi terhadap 30 data log acak menunjukkan bahwa *bottleneck* utama sistem terletak pada *overhead* tetap *rendering* peta di *frontend* (*intercept* 1839,37 ms), bukan pada kompleksitas algoritma (slope hanya 2,45 ms per *node*), dengan koefisien korelasi r=0,151 dan R^2=0,023 yang mengonfirmasi bahwa variasi waktu eksekusi hampir seluruhnya (97,7%) disebabkan oleh faktor non-algoritmik.

3. Berdasarkan verifikasi independen menggunakan *script* Python, akurasi algoritma Dijkstra terbukti 100% untuk jumlah transfer dan sangat mendekati untuk jarak tempuh, dengan selisih maksimal 0,75 km (2,2%) pada skenario kompleks Tanjung Priok → St. MRT Lebak Bulus. Dari sisi efektivitas terlihat berdasarkan 6 skenario pengujian berupa 3 halte ke halte dan 3 POI ke POI, mode Cari 2 berhasil mengurangi jumlah koridor rata-rata hingga 43% (dari 5,3 menjadi 3,0 koridor) dengan tambahan jarak hanya 0,05 km atau sekitar 0,2% dibanding Cari 1 (22,11 km berbanding 22,16 km), membuktikan bahwa preferensi minimum transfer dapat tercapai tanpa mengorbankan perpindahan bus yang terlalu banyak dan jarak tempuh secara signifikan.

4. Performa komputasi algoritma Dijkstra sesuai dengan kompleksitas teoritis $O((V + E)\ log\ V)$ dan masih dalam batas responsif untuk aplikasi web. Dengan V = 8.651 *node* dalam *database* (atau 8.424 *node* aktif dalam graf setelah penyaringan halte tanpa *stop_times*) dan E = 55.000+ *edge*, estimasi operasi maksimum adalah sekitar 815.230 operasi. Waktu eksekusi aktual dari log pencarian menunjukkan rata-rata Cari 1 sebesar 1.450 ms dan Cari 2 sebesar 1.200 ms, dengan seluruh pencarian selesai di bawah 3 detik. Total ukuran *database* sekitar 44,16 MB, dengan komponen terbesar *tb_shapes* (25,06 MB), sementara tabel *Cache* berukuran 7,53 MB. Dengan demikian, sistem dinyatakan efisien secara komputasi dan layak digunakan untuk kebutuhan pencarian rute Transjakarta di wilayah Jabodetabek. Pengujian beban infrastruktur mengungkap kontribusi arsitektural penting berupa penempatan pembangunan graf dan eksekusi Dijkstra di sisi *frontend* (JavaScript) berhasil mendistribusikan beban komputasi ke *browser* pengguna, sehingga *server* tetap stabil dengan 0% *error*

hingga 20 *virtual users* pada skenario *ramp-up*. Namun, *trade-off* yang teridentifikasi adalah peningkatan *response time* seiring bertambahnya jumlah pengguna dari rata-rata 1.210 ms (1 user, *Cache hit*) menjadi 12.018 ms (10 user) dan 16.320 ms (20 user) pada lingkungan lokal, serta 35.093 ms pada lingkungan *production* Railway. Peningkatan ini disebabkan oleh volume transfer data graf (11,44 MB per *request*) melalui REST API yang semakin membebani *bandwidth* dan koneksi *database* seiring bertambahnya user. Temuan ini mengindikasikan bahwa *bottleneck* sistem bergeser dari komputasi algoritma (yang tetap $O((V + E)\ log\ V)$) ke transfer data dan koneksi konkuren, yang menjadi dasar saran untuk pengembangan selanjutnya.

5. Pengujian fungsional dengan metode *Black Box Testing* berhasil memvalidasi seluruh fitur aplikasi dan menunjukkan bahwa prototipe berjalan sesuai spesifikasi. Aplikasi web berbasis Laravel berhasil menampilkan peta interaktif menggunakan Leaflet.js dengan *layer* koridor Transjakarta (243 koridor), *marker* halte (8.651 halte), dan *marker* POI (32.000+ titik) yang dilengkapi dengan fitur *toggle* ON/OFF. Fitur *autocomplete* pada pencarian halte dan POI diimplementasikan dengan debounce 300 ms, batasan hasil 8 halte dan 5 POI, serta network *timeout* 3 detik pada *request* AJAX POI untuk mencegah antarmuka menggantung akibat gangguan koneksi jaringan, dilengkapi dengan *asynchronous yield setTimeout*(…, 100) atau secara bahasa *set Timeout* sebesar 100ms pada eksekusi Dijkstra agar *thread* utama *browser* tetap dapat merender UI selama proses komputasi berlangsung. Fitur pencarian POI dilengkapi dengan seleksi halte terdekat secara bertahap (radius 2 km, 5 km, dan 10 km) serta notifikasi peringatan jika jarak POI ke halte melebihi 1 km. Pengujian fungsional dengan metode *Black Box Testing* terhadap 13 fungsi sistem dan 10 skenario *input-output* menunjukkan tingkat keberhasilan 100%, pengujian API terhadap 16 *endpoint* menunjukkan seluruh *endpoint* mengembalikan respons sesuai spesifikasi. Selain itu, pengujian akseptabilitas pengguna melalui *Usability Testing* yang melibatkan 21 responden menghasilkan indeks kepuasan sebesar 87,11% dari skor maksimal 1.365, melampaui ambang batas kelayakan minimum 80%. Hasil ini mengkategorikan prototipe sebagai “Sangat Layak” dan diterima oleh pengguna akhir, dengan skor tertinggi pada parameter responsivitas Leaflet.js (97/105) dan efisiensi *autocomplete* halte (95/105). Dengan demikian, kelima rumusan masalah telah terjawab dan seluruh tujuan penelitian tercapai.

5.2. Saran

Bab ini menyajikan saran hasil implementasi sebagai berikut:

1. Integrasi Data *Real-time* (GTFS-*Real-time*)

   Sistem saat ini hanya menggunakan data GTFS statis (jadwal tetap). Pengembangan ke depan dapat menambahkan integrasi dengan GTFS-*Real-time* yang menyediakan tiga fitur utama yaitu posisi kendaraan (*vehicle position*) untuk melacak lokasi bus secara *live*, perkiraan kedatangan (*trip update*) untuk memberikan estimasi waktu kedatangan yang akurat, dan peringatan gangguan (*service alert*) untuk menginformasikan penutupan halte atau perubahan rute mendadak. Hal ini akan membuat sistem lebih responsif terhadap kondisi aktual di lapangan.

2. Pemanfaatan Data *shape_dist_traveled*

   Data *shape_dist_traveled* telah berhasil diimpor ke *database* dengan total 24.567 record (100% lengkap tanpa data *null*). Penelitian selanjutnya dapat memanfaatkan data ini untuk menghitung jarak tempuh berdasarkan lintasan jalan aktual (bukan garis lurus Haversine). Namun, perlu dilakukan pembersihan data (data *Cleaning*) terlebih dahulu karena data *shape_dist_traveled* bersifat kumulatif per *trip-id* tertentu, sehingga ketika diekstrak secara global per koridor dapat terjadi anomali nilai bobot. Setelah data bersih, akurasi perhitungan jarak pada mode Cari 1 dapat ditingkatkan secara signifikan.

3. Fitur Pencarian Berbasis Waktu (*Time-based Routing*)

   Sistem dapat dikembangkan untuk mendukung pencarian rute dengan parameter waktu, misalnya "berangkat setelah jam 08.00", "tiba sebelum jam 10.00", atau "pencarian berdasarkan jam keberangkatan tertentu". Fitur ini dapat diimplementasikan dengan memanfaatkan data *arrival_time* dan *departure_time* pada tabel *tb_stop_times*, serta data *calendar.txt* untuk mengetahui hari operasional. Dengan fitur ini, pengguna dapat merencanakan perjalanan sesuai jadwal keberangkatan yang diinginkan.

4. Integrasi dengan Moda Transportasi Lain

   Penelitian selanjutnya dapat memperluas cakupan sistem dengan mengintegrasikan operator transportasi lain seperti KRL Commuter Line, MRT Jakarta, dan LRT Jakarta. Hal ini memerlukan penerapan adapter pattern untuk menyatukan data GTFS dari berbagai operator yang mungkin memiliki format berbeda, serta sinkronisasi jadwal

antarmoda untuk menghasilkan rekomendasi rute Transjakarta yang bisa multi-moda sebenarnya (misal Transjakarta dengan MRT atau KRL atau LRT).

5. Performance *Testing* Formal

   Pengujian performa saat ini masih terbatas pada data dari log pencarian. Pengembangan ke depan dapat melakukan performance *testing* formal menggunakan tools seperti Apache JMeter atau K6 untuk mengukur *response time* API di bawah beban normal dan puncak, *throughput* maksimal (*request* per detik), skalabilitas sistem dengan jumlah pengguna bersamaan, serta ketahanan sistem terhadap lonjakan trafik.

6. Perbaikan Data *Shape* Koridor

   Sekitar 5% dari total koridor (12 dari 243 koridor) mengalami deviasi *shape* hingga 100 meter dari jalur jalan sebenarnya. Pengembangan selanjutnya dapat memperbaiki data *shape* menggunakan sumber data alternatif yang lebih presisi seperti OpenStreetMap, atau menerapkan algoritma simplifikasi *shape* untuk mengurangi titik-titik yang melenceng.

7. Optimasi Arsitektur *Client-side* dan Mekanisme Transfer Data

   Selain itu, berdasarkan hasil pengujian beban yang mengungkap peningkatan *response time* seiring bertambahnya jumlah pengguna serta temuan bahwa *bottleneck* utama bergeser dari komputasi algoritma ke transfer data graf (11,44 MB per *request*) melalui REST API, disarankan untuk melakukan optimasi arsitektur dengan memindahkan pembangunan graf ke sisi *backend* (*server-side pre-computation*) agar *frontend* hanya menerima hasil rute jadi, menerapkan *lazy loading* dan *chunk*ing data spasial berdasarkan area peta yang terlihat, menggunakan kompresi data seperti GZIP atau Brotli untuk mengurangi ukuran *payload* hingga 70-80%, serta memanfaatkan *Service Worker* untuk menyimpan data graf di *Cache browser* secara permanen sehingga pengguna tidak perlu mengunduh ulang data besar setiap kali membuka halaman.