

PENCARIAN RUTE PEMBELAJARAN HURUF KANJI DALAM KNOWLEDGE GRAPH

LAPORAN TUGAS AKHIR

Disusun sebagai syarat kelulusan tingkat sarjana

oleh :

Ferdian Ifkarsyah

NIM: 13517024



PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2021

**PENCARIAN RUTE PEMBELAJARAN HURUF KANJI
DALAM KNOWLEDGE GRAPH**

Laporan Tugas Akhir

Oleh

FERDIAN IFKARSYAH

NIM: 13517024

Program Studi: Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir
di Bandung, pada tanggal 23 September 2021.

Pembimbing

Dr.techn. Saiful Akbar, ST., MT.

NIP. 19740509 199803 1 002

LEMBAR PERNYATAAN

Dengan ini saya menyatakan bahwa:

1. Pengerjaan dan penulisan Laporan Tugas Akhir ini dilakukan tanpa menggunakan bantuan yang tidak dibenarkan.
2. Segala bentuk kutipan dan acuan terhadap tulisan orang lain yang digunakan di dalam penyusunan laporan tugas akhir ini telah dituliskan dengan baik dan benar.
3. Laporan Tugas Akhir ini belum pernah diajukan pada program pendidikan di perguruan tinggi mana pun.

Jika terbukti melanggar hal-hal di atas, saya bersedia dikenakan sanksi sesuai dengan Peraturan Akademik dan Kemahasiswaan Institut Teknologi Bandung bagian Penegakan Norma Akademik dan Kemahasiswaan khususnya Pasal 2.1 dan Pasal 2.2.

Bandung, 23 September 2021



Ferdian Ifkarsyah

NIM. 13517024

ABSTRAK

Huruf kanji adalah sistem huruf yang digunakan dalam bahasa Jepang. Salah satu pembeda huruf kanji dibandingkan huruf alfabet adalah huruf kanji dapat tersusun atas radikal, atau bagian kanji yang lebih kecil. Misalnya, kanji 休 (rest; retire) disusun oleh radikal 亻 (person) dan 木 (tree). Susunan ini juga membentuk suatu cerita bahwa salah satu bentuk istirahat adalah orang yang tertidur di pohon.

Keterkaitan antara satu kanji, radikalnya, dan kanji yang lainnya ini dapat direpresentasikan dengan *knowledge graph*. Untuk memanfaatkan keterkaitan kanji dan radikal ini, algoritma pencarian rute diperlukan untuk mencari rute pembelajaran kanji. Kami menguji 3 algoritma pencarian rute, yaitu *Dijkstra*, *A**, dan *Steiner Tree*. Algoritma *Steiner Tree* memiliki waktu tempuh yang lebih cepat di antara ketiganya karena memiliki *time complexity* yang lebih baik. Algoritma *Steiner Tree* memiliki *time complexity* $O((|MOrig| + |MDest|) * |Vertices_G|^2)$, sementara algoritma *Dijkstra* dan *A** memiliki *time complexity* $O((|MOrig| * |MDest|) * (|Vertices_G| + |Edges_G|) \log |Vertices_G|)$. Ditemukan juga bahwa optimasi *Steiner Tree* dengan teknik mengekstraksi langkah pembangkitan *metricClosure* dan dilakukan memoisasi mempercepat kinerja dari Algoritma *Steiner Tree*. Algoritma *Steiner Tree* juga memiliki jumlah simpul hasil yang lebih sedikit dibandingkan *Dijkstra* maupun *A**. Hal ini karena graf hasil Algoritma *Steiner Tree* dipastikan berbentuk *tree* atau graf tanpa *cycle*, sedangkan algoritma *Dijkstra* dan *A** mungkin menghasilkan graf dengan *cycle*.

Kata kunci: *knowledge graph*, kanji, *Dijkstra*, *A** *Steiner Tree*

ABSTRACT

Kanji is the letter system used in Japanese. One of the differences between kanji and alphabet letters is that kanji can be made up of radicals, or smaller kanji parts. For example, the kanji 休(rest; retire) is composed of the radicals 亻(person) and 木(tree). This arrangement also forms a story that one form of rest is a person sleeping in a tree.

The relationship between one kanji, its radical, and the other kanji can be represented by knowledge graph. To take advantage of this linkage of kanji and radicals, a route search algorithm is needed to search for kanji learning routes. We tested 3 route search algorithms, namely Dijkstra, A*, and Steiner Tree. The Steiner Tree Algorithm has a faster travel time among the three because it has a better time complexity. The Steiner Tree Algorithm has time complexity $O((|MOrig| + |MDest|) * |Vertices_G|^2)$, while the Dijkstra and A* Algorithms have time complexity $O((|MOrig| * |MDest|) * (|Vertices_G| + |Edges_G|) \log |Vertices_G|)$. We found that the optimization of Steiner Tree by extracting the generation step of metricClosure and memoization accelerated the performance of the Steiner Tree Algorithm. The Steiner Tree Algorithm also has a smaller number of result nodes than Dijkstra and A*. This is because the resulting graph from the Steiner tree algorithm is certain to be in the form of tree or a graph without cycle, while the Dijkstra and A* Algorithms may produce a graph with cycle.

Key words: knowledge graph, kanji, Dijkstra, A* Steiner Tree

KATA PENGANTAR

Puji syukur bagi Tuhan Yang Mahaesa, atas berkat dan rahmatnya, sehingga penulis dapat menyelesaikan tugas akhir yang berjudul **Pencarian Rute Pembelajaran Huruf Kanji dalam Knowledge Graph** sebagai salah satu syarat kelulusan Program Studi S1 Teknik Informatika Institut Teknologi Bandung. Penulis juga ingin berterimakasih kepada berbagai pihak yang membantu dalam penyelesaian tugas akhir ini:

1. Bapak Dr.techn. Saiful Akbar, ST., MT. selaku dosen pembimbing yang telah memberikan bimbingan, arahan, dan dorongan selama pengerjaan Tugas Akhir.
2. Ibu Dr. Fazat Nur Azizah, S.T, M.Sc. dan Tricya Esterina Widagdo, S.T, M.Sc. selaku dosen penguji yang banyak memberikan saran untuk perbaikan Tugas Akhir.
3. Ibu Yanti Rusmawati, ST., M.Kom., M.Sc., Ph.D. selaku dosen wali yang telah membantu proses pendidikan penulis di Program Studi Teknik Informatika.
4. Segenap dosen pengampu mata kuliah yang telah memberikan keilmuannya selama penulis belajar di Institut Teknologi Bandung.
5. Ayah dan Ibu penulis yang sepanjang waktu memberikan dukungan baik moril dan materiil selama penulis belajar.
6. Teman-teman mahasiswa Teknik Informatika yang telah bersama-sama berbagi cerita selama penulis belajar di dunia perkuliahan.

Penulis menyadari bahwa masih banyak kekurangan dalam Tugas Akhir ini. Pada akhirnya, penulis berharap baik dan buruknya Tugas Akhir ini dapat menjadi pembelajaran bagi pembaca.

Bandung, 23 September 2021

Penulis

DAFTAR ISI

Lembar Pernyataan	ii
Abstrak	iii
Abstract	iv
Kata Pengantar	v
Daftar Isi	vi
Daftar Gambar	ix
Daftar Tabel	x
Daftar Algoritma	xi
Daftar Singkatan	xii
BAB I PENDAHULUAN	1
I.1 Latar Belakang	1
I.1.1 Bahasa Jepang dan Kanji	1
I.1.2 Mempelajari Kanji	2
I.2 Rumusan Masalah	4
I.3 Tujuan	4
I.4 Batasan Masalah	5
I.5 Metodologi	5
I.6 Sistematika Pembahasan	6
BAB II STUDI LITERATUR	7
II.1 <i>Knowledge Graph</i>	7
II.2 Permasalahan MOMD	8
II.3 Algoritma Pencarian Rute pada <i>Knowledge Graph</i>	10
II.3.1 Algoritma Dijkstra	10

II.3.2	Algoritma A*	11
II.3.3	Algoritma <i>Steiner Tree</i>	13
II.4	Kakas Pendukung Pengembangan	14
II.4.1	NetworkX	14
II.4.2	Matplotlib	15
BAB III	ANALISIS MASALAH DAN RANCANGAN SOLUSI	16
III.1	Analisis Permasalahan	16
III.1.1	Definisi Fungsi	16
III.1.2	Pilihan Algoritma	17
III.1.2.1	Algoritma Dijkstra	17
III.1.2.2	Algoritma A*	18
III.1.2.3	Algoritma <i>Steiner Tree</i>	20
III.1.3	Metriks Evaluasi	25
III.2	Rancangan Solusi	26
III.2.1	Gambaran Umum Solusi	26
III.2.2	Persiapan Dataset <i>Knowledge Graph</i>	26
III.2.2.1	Dataset <i>Joyo Kanji</i>	27
III.2.2.2	Dataset <i>Radicals.csv</i>	28
III.2.2.3	Dataset <i>Kanjiapi.dev</i>	29
III.2.2.4	Mengagregasi Data dari Seluruh Dataset	31
III.2.3	Rancangan Solusi Pencarian Rute <i>Knowledge Graph</i> Relasi Kanji	32
BAB IV	EKSPERIMEN DAN ANALISIS	34
IV.1	Kasus Uji	34
IV.2	Analisis <i>Time Complexity</i>	34
IV.2.1	Algoritma Brute Force Dijkstra	34
IV.2.2	Algoritma A*	35
IV.2.3	Algoritma <i>Steiner Tree</i>	36
IV.2.4	Optimasi Algoritma <i>Steiner Tree</i>	37
IV.3	Hasil dan Analisis Uji Waktu Empiris	39

IV.4 Hasil dan Analisis Uji Jumlah Simpul Antara	41
BAB V KESIMPULAN DAN SARAN	43
V.1 Kesimpulan	43
V.2 Saran	43
Daftar Referensi	xiii

DAFTAR GAMBAR

Gambar I.1	Tiga contoh kanji yang dibentuk dari radikal dan membentuk radikal.	2
Gambar I.2	Aplikasi <i>The Kanji Map</i>	3
Gambar II.1	Hasil pencarian Google untuk <i>query</i> "Abraham Lincoln" .	8
Gambar II.2	Knowledge Graph.	9
Gambar II.3	Perbandingan Algoritma Dijkstra dan A*	12
Gambar II.4	Perbandingan permasalahan <i>Spanning Tree</i> dan <i>Steiner Tree</i>	13
Gambar III.1	Penjelasan Fungsi <i>findRoute</i>	17
Gambar III.2	Perbandingan hasil Dijkstra dan A* versus Steiner Tree .	25
Gambar III.3	Langkah pengerjaan solusi secara umum	26
Gambar III.4	Rancangan persiapan dataset <i>knowledge graph</i> kanji . . .	26
Gambar III.5	Dataset <i>Joyo Kanji</i> dalam bentuk Website	27
Gambar III.6	Dataset Hasil Transformasi <i>Joyo Kanji</i>	28
Gambar III.7	Dataset <i>Radicals.CSV</i>	28
Gambar III.8	Dataset Hasil Transformasi Radikal	29
Gambar III.9	Contoh respons kanjiapi.dev	30
Gambar III.10	Dataset Hasil Transformasi kanjiapi.dev	30
Gambar III.11	Representasi simpul pembelajaran <i>Knowledge Graph</i> relasi kanji	31
Gambar III.12	Representasi sisi <i>Knowledge Graph</i> relasi kanji	32
Gambar III.13	Langkah pengerjaan algoritma pencarian rute pembelajaran <i>Knowledge Graph</i> relasi kanji	32
Gambar IV.1	Hasil uji waktu empiris dalam bentuk grafik	39
Gambar IV.2	Hasil uji jumlah simpul terpakai dalam bentuk grafik . . .	42

DAFTAR TABEL

Tabel III.1	Daftar Transformasi Dataset	31
Tabel IV.1	Daftar kasus uji	35
Tabel IV.2	Hasil uji waktu empiris dalam bentuk tabel(dalam detik) . .	39
Tabel IV.3	Hasil uji optimasi Steiner Tree dengan memoisasi(dalam detik)	40
Tabel IV.4	Hasil uji jumlah simpul terpakai dalam bentuk tabel	41

DAFTAR ALGORITMA

Algoritma IV.1	Algoritma Brute Force Dijkstra	34
Algoritma IV.2	Algoritma A*	35
Algoritma IV.3	Fungsi <i>Metric Closure</i>	36
Algoritma IV.4	Algoritma <i>My Steiner Tree</i>	37
Algoritma IV.5	Algoritma <i>My Steiner Tree</i> dengan pendekatan <i>Dynamic Programming</i> dan <i>Lazy Loading</i>	38

DAFTAR SINGKATAN

Singkatan	Kepanjangan
KG	<i>Knowledge Graph</i>
NLP	<i>Natural Language Processing</i>
BF	<i>Brute Force</i>
CN	<i>Common Neighbor</i>
JS	<i>Jaccard Similarity</i>

BAB I

PENDAHULUAN

I.1 Latar Belakang

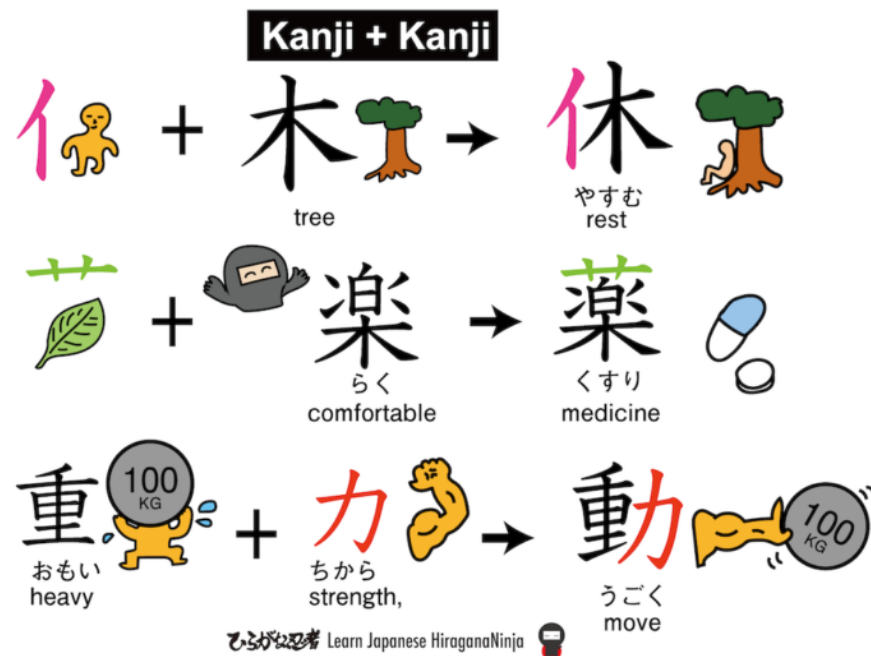
I.1.1 Bahasa Jepang dan Kanji

Bahasa Jepang adalah bahasa nasional yang digunakan oleh negara Jepang dan dituturkan oleh sekitar 128 juta orang penduduknya. Di luar masyarakat Jepang itu sendiri, bahasa Jepang juga populer seiring dengan meningkatnya antusiasme terhadap budaya visual modern Jepang dan wisata ke negeri matahari tersebut. Kemajuan teknologi dan tingkat pendidikan Jepang pun menjadi magnet bagi para pelajar di seluruh penjuru dunia untuk belajar di Jepang, sehingga mereka harus belajar bahasa Jepang sebagai salah satu syarat untuk mencari ilmu di negeri matahari tersebut.

Sistem penulisan bahasa Jepang mengenal 3 jenis penulisan, yaitu: hiragana (ひらがな), katakana (カタカナ), dan kanji (漢字). Hiragana dan katakana berfungsi sebagai suku kata. Hiragana berfungsi sebagai dasar pengucapan suku kata untuk kosakata yang berasal asli dari Jepang sedangkan Katakana digunakan untuk kosakata serapan. Sistem penulisan bahasa Jepang juga banyak menggunakan karakter pinjaman dari Cina yang lebih dikenal dengan nama kanji. Kanji digunakan dalam komponen pembentuk kosakata bahasa Jepang.

Huruf kanji memiliki keunikan tersendiri yang menjadi pembeda dari jenis huruf lainnya. Keunikan dari huruf kanji adalah tersusun atas radikal, yaitu bagian kanji yang lebih kecil dan susunan radikal ini membentuk suatu cerita. Beberapa contohnya dapat dilihat pada Gambar I.1. Misalnya, kanji 休 (rest; retire) disusun oleh radikal 亻 (person) dan 木 (tree). Dalam kasus ini, kedua radikal mengisyaratkan seseorang yang bersandar pada pohon, dengan kata lain, beristirahat atau pensiun. Contoh berikutnya adalah kanji 薬 (medicine) yang disusun oleh radikal 艹 (leave) dan 樂 (comfortable) yang berarti daun atau bahan alami yang membuat nyaman, yaitu

obat. Contoh terakhir adalah kanji 動(move) yang tersusun oleh 重(heavy) dan 力(power) yang berarti bahwa kita perlu kekuatan untuk memindahkan sesuatu yang berat.



Gambar I.1 Tiga contoh kanji yang dibentuk dari radikal dan membentuk radikal.

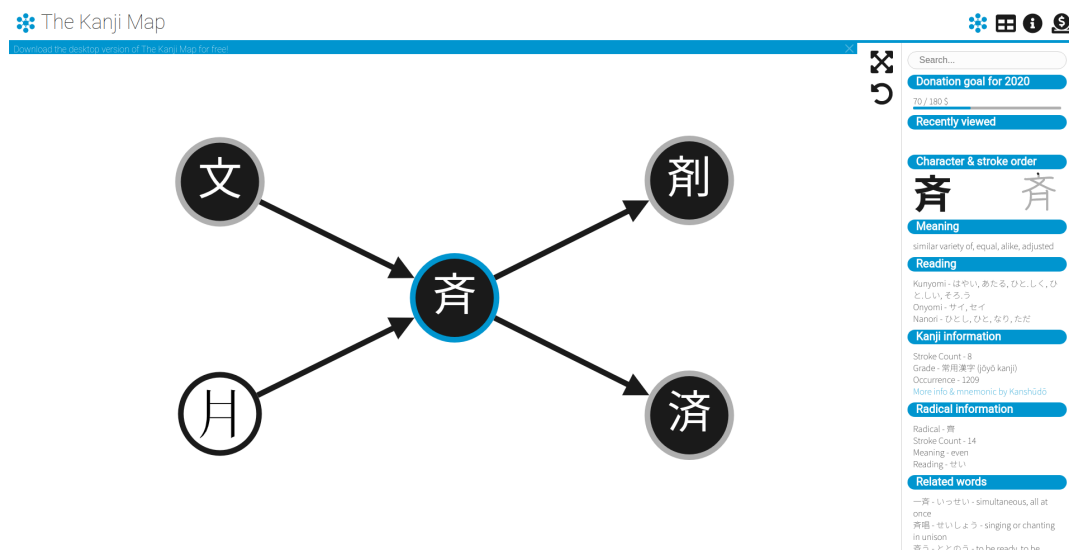
Sumber: <http://hiragana.world/wp2/kanjikanji/>

I.1.2 Mempelajari Kanji

Dalam mempelajari karakter kanji, pembelajar bahasa Jepang dapat dibagi menjadi dua golongan, yaitu pembelajar dengan target tertentu dan pembelajar dengan motivasi eksploratif. Golongan pertama, pelajar dan profesional memiliki tujuan administratif untuk masuk ke sebuah lembaga pendidikan atau perusahaan di Jepang. Jepang sebagai negara dengan teknologi dan pendidikan yang tinggi merupakan daya tarik sendiri bagi para pelajar dan profesional untuk belajar dan berkarir di Jepang. Lembaga pendidikan dan perusahaan Jepang di lain sisi, menerapkan syarat atau sertifikasi kemampuan berbahasa tertentu untuk para pelajar dan tenaga kerja asing sehingga mereka dapat beraktivitas secara lancar di tempat mereka nantinya. Golongan kedua, pembelajar mandiri dengan motivasi intrinsiknya untuk belajar

bahasa Jepang. Bahasa Jepang duduk di posisi 13 sebagai bahasa yang paling banyak digunakan dengan sekitar 128 juta penutur di dunia[1]. Keinginan untuk berwisata di Jepang ataupun sekadar untuk lebih menikmati budaya modernnya untuk mereka. Pembelajar model pertama ini juga tidak mempunyai tenggat khusus untuk harus bisa mempelajari daftar tertentu dan lebih senang belajar secara lebih eksploratif.

Pembelajar golongan pertama atau pembelajar dengan target akan mempelajari kanji bahasa Jepang dari *list* atau daftar kanji yang akan diuji dan keluar dalam JLPT(*Japan Language Proficiency Test* yang terdiri atas 5 level mulai dari N5 yang termudah sampai N1 yang tersulit. Cara ini cukup efektif dan efisien karena pembelajar hanya harus belajar kanji yang sesuai dengan target ujian yang diikutinya. Namun, cara ini kurang dapat dipelajari secara mudah oleh manusia karena hanya mengandalkan hafalan murni. Cara ini juga tidak memanfaatkan keunikan-keunikan kanji yang menghubungkan antarkanji.



Gambar I.2 Aplikasi *The Kanji Map*

Sedangkan pembelajar golongan kedua atau pembelajar eksploratif akan melihat keunikan-keunikan bahasa Jepang yang akan memudahkannya dalam belajar karakter kanji. Mereka akan mempelajari kanji secara visual dengan memanfaatkan representasi *knowledge graph*. Sayangnya, hal baik ini tidak didapatkan pada

pembelajar tertarget. Padahal, keterkaitan kanji dan relasi ini dapat dimanfaatkan untuk mempermudah pembelajar dengan target untuk lebih mudah mengingat kanji targetnya. Tugas akhir ini akan membantu pembelajar dengan target untuk bisa memanfaatkan keterkaitan kanji dan relasi yang direpresentasikan dengan *knowledge graf*.

I.2 Rumusan Masalah

Salah satu produk yang telah ada yang melakukan cara representasi dengan *knowledge graph* adalah The Kanji Map (thekanjimap.com) yang dapat dilihat pada Gambar I.2. *The Kanji Map* bersifat sebagai ensiklopedia memang cocok untuk pembelajar golongan pertama yang eksploratif, tapi ini kurang cocok dengan pembelajar dengan target. Padahal, hubungan antarkanji yang diberikan *The Kanji Map* dapat juga membantu pembelajar dengan target untuk bisa lebih belajar secara mudah dengan bantuan relasi radikalnya. Oleh karena itu, tugas akhir ini akan mencoba untuk menambahkan fitur pencarian urutan pembelajar kanji. Untuk mendukung *user experience* algoritma tersebut harus berjalan dalam waktu yang relatif cepat dan menghasilkan rekomendasi simpul yang sedikit.

Berdasarkan bahasan pada latar belakang, rumusan masalah yang akan dijawab dalam penelitian ini adalah "Apa algoritma dengan efisiensi waktu dan simpul tersedikit untuk mencari rute hubungan antara dua buah kumpulan simpul dalam *knowledge graph* relasi kanji?"

I.3 Tujuan

Aktivitas utama dari tugas akhir ini adalah untuk melakukan perbandingan dan perbaikan pada berbagai algoritma untuk mencari rute dalam *knowledge graph*. Lalu, aktivitas pendukungnya adalah untuk mengkonstruksi *knowledge graph* yang akan digunakan sebagai media eksperimen algoritma yang akan diujikan.

Adapun tujuan dari pembuatan tugas akhir ini berdasarkan rumusan masalah adalah menemukan algoritma pencarian rute hubungan antara dua buah kumpulan simpul

dalam *knowledge graph* relasi kanji dengan metrik waktu tercepat dan jumlah simpul hasil tersedikit.

I.4 Batasan Masalah

Dalam pengerjaan tugas akhir ini, akan diberikan batasan sebagai berikut:

1. Sumber data huruf kanji yang digunakan akan bersumber wikipedia List of Joyo Kanji, Jimmy Crequer Roth 2019 dan kanjiapi.dev.
2. Relasi yang diimplementasikan terbatas pada kesamaan radikal penyusunnya saja. Relasi kesamaan tema, kesamaan pengucapan(homofon), kesamaan arti(sinonim), tidak akan dilibatkan.
3. Aplikasi yang dibangun bukan aplikasi yang memiliki fitur lengkap sebagai media pembelajaran. Namun, hanya sebagai fungsi yang dapat didemonstrasikan untuk menerima masukan *user*.
4. Fungsi yang dihasilkan akan berfokus untuk pembelajar dengan target yang ingin secepat mungkin mencapai targetnya dengan sesedikit mungkin simpul yang dipelajari.

I.5 Metodologi

Berikut adalah metodologi yang diterapkan dalam penyelesaian tugas akhir ini:

1. Pengumpulan Data
Data huruf kanji yang digunakan akan bersumber dari wikipedia List of Joyo Kanji, Jimmy Crequer Roth 2019 dan kanjiapi.dev.
2. Perancangan Solusi
Tahap selanjutnya adalah perancangan solusi. Pada tahap ini, berbagai kaskas yang telah dieksplorasi akan dirancang sedemikian sehingga dapat bekerja sama satu sama lainnya memenuhi solusi.
3. Implementasi Solusi

Solusi yang telah dirancang akan diimplementasikan dengan menggunakan kaskas Python3, NetworkX, dan Matplotlib.

4. Analisis dan Evaluasi Solusi

Terakhir, implementasi solusi akan dianalisis dan dievaluasi untuk melihat kekurangan yang masih di dapat. Hasil analisis dan evaluasi ini akan menjadi dasar untuk mengembangkan aplikasi secara lebih lanjut.

I.6 Sistematika Pembahasan

Tugas akhir ini menggunakan sistematika pembahasan sebagai berikut:

Bab I PENDAHULUAN

Pendahuluan membahas latar belakang, rumusan masalah, tujuan, batasan masalah, metodologi, dan sistematika pembahasan tugas akhir yang akan dikerjakan.

Bab II STUDI LITERATUR

Studi Literatur berisi tentang konsep dasar dan teori yang menjadi fondasi pengerjaan tugas akhir. Konsep dasar dan teori yang digunakan adalah *knowledge graph*, permasalahan MOMD, Algoritma Dijkstra, Algoritma A* dan Algoritma Steiner Tree.

Bab III ANALISIS MASALAH DAN RANCANGAN SOLUSI

Permasalahan yang sudah dijelaskan secara singkat pada Bab 1 akan dianalisis secara detail pada Bab 3. Setelah dianalisis, akan dijabarkan rancangan solusi dalam pemecahan masalah ini.

Bab IV EKSPERIMEN DAN ANALISIS

Dalam bab ini akan diceritakan tentang implementasi solusi, analisis algoritma implementasi, serta evaluasi yang dapat dilakukan.

Bab V KESIMPULAN DAN SARAN

Berisi kesimpulan akhir yang akan menjawab rumusan masalah dan saran untuk penelitian yang akan dilakukan kedepannya.

BAB II

STUDI LITERATUR

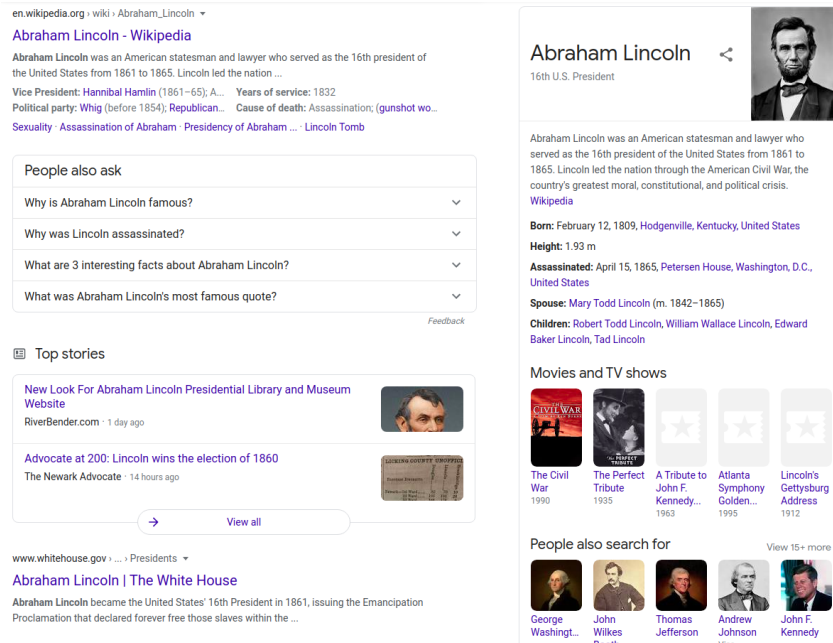
II.1 *Knowledge Graph*

Schneider mencetuskan istilah *Knowledge Graph* pada tahun 1972 dalam sebuah diskusi tentang pembangunan sistem instruksional modular[2]. Lalu, beberapa *knowledge graph* awal dibangun dengan memilih topik yang spesifik. Misalnya, Wordnet yang dibangun pada tahun 1985 oleh Miller, Beckwith, Fellbaum, dkk. yang mengumpulkan hubungan semantik antara kosakata dan semantik dari sebuah bahasa[3].

Knowledge Graph dengan tujuan umum dibangun pada tahun 2007 melalui DBPedia dan Freebase. DBpedia dikonstruksi dengan mengekstraksi data semiterstruktur dari Wikipedia dan mempublikasikannya dalam bentuk *linked data*. Hasil dari konstruksi ini memungkinkan penggunaanya untuk mencari hubungan antarartikel di Wikipedia[4]. Sementara itu, Freebase mengambil satu langkah lebih maju dengan mengumpulkan data dari dataset publik Bollacker, Evans, Paritosh, dkk.

Kemudian pada tahun 2012, istilah *Knowledge Graph* dipopulerkan oleh Google dalam Google Knowledge Graph (selanjutnya disebut Google KG). Google KG adalah upaya Google membuat mesin pencari lebih dari sebagai *search engine* yang mencari berdasarkan kemiripan teks saja. Google KG adalah *knowledge engine* yang mengerti arti semantik teks tersebut. Hasil dari Google KG ditampilkan dalam panel di sebelah kanan hasil pencarian. Misalnya, pada Gambar II.1, dapat dilihat bahwa Abraham Lincoln berkaitan erat dengan George Washington yang merupakan sesama presiden Amerika Serikat.

Dua konsep yang membantu tujuan ini adalah entitas dan relasi. Sebuah entitas dapat memiliki beberapa properti. Contoh KG ditunjukkan oleh Gambar II.2. Ada sebuah entitas bernama "Da Vinci". Entitas ini memiliki beberapa relasi. Salah satunya adalah relasi "is a" yang menghubungkannya dengan entitas lain bernama "Person". Relasi "is a Person" bukanlah satu-satunya relasi yang ada di graf ini.



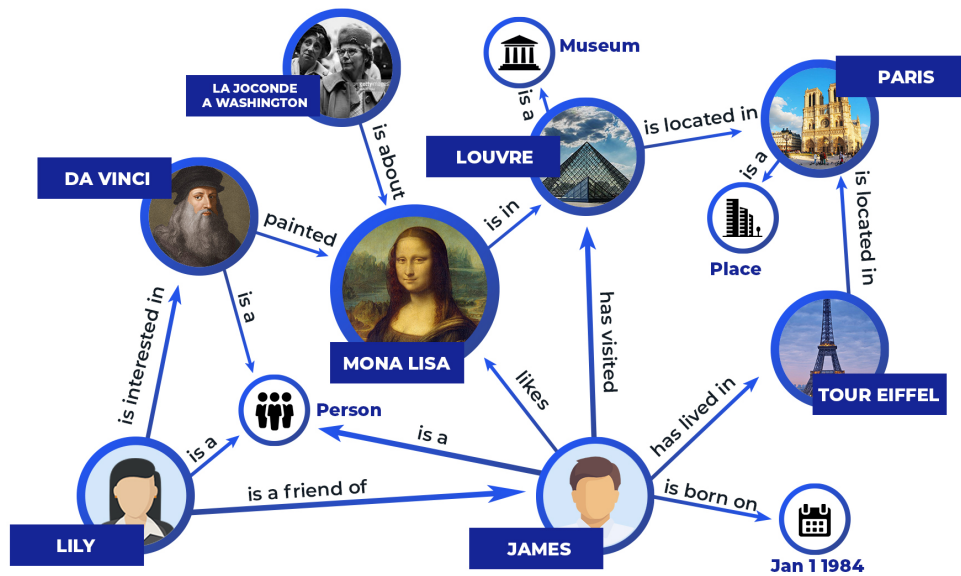
Gambar II.1 Hasil pencarian Google untuk *query* "Abraham Lincoln"

Entitas lain bernama "James" dan "Lily" juga memiliki hubungan ini. Dari sini, kita ketahui bahwa "Da Vinci", "James", dan "Lily" memiliki kesamaan berupa "is a Person".

Sejak saat itu, beberapa perusahaan multinasional besar seperti Facebook, LinkedIn, Airbnb, Microsoft mengikuti jejak Google dengan membagikan penggunaan *knowledge graph* mereka kepada publik.

II.2 Permasalahan MOMD

Permasalahan pencarian rute penghubung antara dua buah kumpulan simpul yang disebut dalam rumusan masalah dikenal dengan permasalahan **multiple-origin-multiple-destination** atau disingkat permasalahan **MOMD** yang disebutkan dalam penelitian Barták, Dovier, dan Zhou. Dalam penelitiannya, Barták, Dovier, dan Zhou menjelaskan bahwa permasalahan MOMD ini sendiri adalah versi sederhana dari masalah perencanaan logistik di mana paket harus diangkut dari asalnya ke tujuan dengan beberapa truk dengan total biaya minimum. Fungsi objektif dari permasalahan MOMD ini adalah untuk membentuk subgraf dengan memilih sisi yang dapat meminimalkan biaya untuk menghubungkan seluruh simpul terminal[6].



Gambar II.2 Knowledge Graph.

Sumber: people.virginia.edu/~jl6qk/sp20-graph-mining/lec2.pdf

Secara lebih formal, permasalahan MOMD dijelaskan sebagai berikut. Misalkan:

- $G = (V, E, w)$, dengan V adalah kumpulan simpul, E adalah kumpulan sisi graf G , dan w adalah bobot tiap $e \in E$.
- $MOrig \in V$, adalah daftar simpul origin.
- $MDest \in V$, adalah daftar simpul destinasi.
- P adalah daftar pasangan origin-destinasi, dengan $p \in P$, didefinisikan sebagai tuple $(orig(p), dest(p))$ dengan $orig(p) \in MOrig$ dan $dest(p) \in MDest$.

Lalu, tujuan dari permasalahan MOMD adalah untuk mencari subgraf $G' = (V', E', w')$, dengan V' adalah gabungan $MOrig$, $MDest$, dan simpul penghubungnya, dan E' adalah subset E dengan jumlah bobot w' minimum.

Permasalahan MOMD ini termasuk kedalam kategori NP-hard atau permasalahan yang tidak mempunyai solusi polinomial. Barták, Dovier, dan Zhou mengusulkan dua model berbasis SAT (*boolean satisfiability problem*) untuk menyelesaikan masalah secara optimal. Model berdasarkan *flow-preserving constraint* lebih efisien daripada model berdasarkan *reachability constraint*.

II.3 Algoritma Pencarian Rute pada *Knowledge Graph*

Algoritma pencarian rute bertujuan untuk menemukan rute optimal antara dua buah simpul dalam suatu graf. Pada graf berbobot, rute optimal didefinisikan sebagai jumlah bobot dari simpul awal sampai simpul akhir. Sedangkan pada graf tidak berbobot, rute optimal adalah banyaknya simpul yang dilewati dari simpul awal sampai simpul akhir. Berdasarkan perilaku ini, algoritma jenis ini dapat digunakan untuk beberapa hal berikut:

- Menemukan rute optimal antara dua buah kota dalam sebuah aplikasi map seperti Google Maps.
- Menemukan jarak pertemanan antara akun dalam sebuah jejaring sosial. Misalnya, pada LinkedIn, kita dapat melihat akun orang lain apakah dia koneksi pertama, koneksi kedua, dan koneksi ketiga atau lebih.

II.3.1 Algoritma Dijkstra

Algoritma Dijkstra dibangun pada tahun 1956 dan dinamakan sesuai penemunya Dijkstra, seorang ilmuwan komputer dan *software engineer* yang berasal dari Belanda. Algoritma ini adalah salah satu varian dari algoritma pencarian *shortest paths* yang banyak digunakan dalam permasalahan pencarian rute dalam aplikasi map, pemasangan jaringan telepon ataupun internet, dan berbagai masalah yang melibatkan pengalamatan seperti pada domain robotik dan *embedded system* [7].

Algoritma Dijkstra bekerja dengan prinsip *greedy* dengan cara memilih tetangga dengan bobot terkecil untuk tiap simpul yang sedang dihindarkannya. Untuk menjalankan Algoritma dijkstra, diperlukan tiga buah variabel, yaitu:

- *dist*, sebuah daftar yang menyimpan jarak dari simpul asal ke tiap simpul di dalam graf. Seluruh simpul akan diinisialisasi dengan $dist(v) = \infty$ kecuali simpul asal s yang akan diinisialisasi dengan $dist(s) = 0$. Q , sebuah *queue* yang menampung simpul yang akan diproses. Q akan kosong di akhir algoritma. S , sebuah *set* yang menampung simpul yang telah dikunjungi

oleh algoritma. S ini akan berisi seluruh simpul graf di akhir algoritma.

Dengan tiga variabel di atas, berikut akan dijelaskan langkah-langkah dari Algoritma Dijkstra:

1. Selama Q tidak kosong, keluarkan simpul v yang belum ada di S dari Q dengan $dist(v)$ terkecil). Pada iterasi pertama, simpul asal s akan dipilih karena $dist(s)$ diinisialisasi ke 0. Pada iterasi berikutnya, simpul dengan nilai $dist$ minimum dipilih.
2. Tambahkan simpul v ke S , untuk menunjukkan bahwa v telah dikunjungi
3. Perbarui nilai $dist$ dari simpul yang berdekatan dari simpul saat ini v sebagai berikut:
 - untuk setiap simpul baru yang bertetanggaan v , jika $dist(v) + weight(u, v) < dist(u)$, ada jarak minimal baru yang ditemukan untuk u , jadi perbarui $dist(u)$ ke jarak minimal baru tersebut.
 - jika tidak, jangan lakukan apa-apa.

Dalam iterasi terakhir, algoritma ini telah mengunjungi semua simpul dalam graf dan menemukan jarak minimal dari sumber asal v ke tiap simpul dalam graf yang ditampung dalam variable $dist$. Dalam varian permasalahan untuk mencari jarak terpendek antar dua buah simpul(bukan antara simpul asal dan seluruh simpul), Algoritma Dijkstra dapat diterminasi lebih cepat dengan melakukan terminasi pada iterasi pada langkah 1 jika simpul yang sedang diproses adalah simpul tujuan.

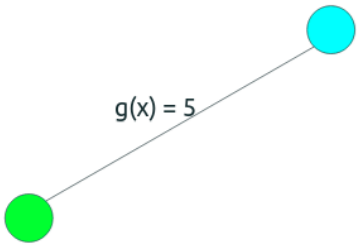
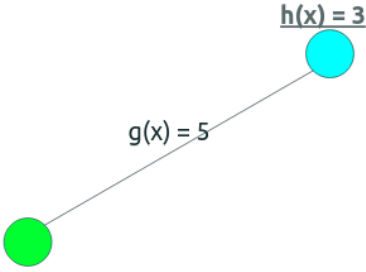
Dalam menjalankan Algoritma Dijkstra, satu syarat penting yang wajib diingat adalah algoritma ini hanya dapat bekerja dengan graf yang memiliki bobot positif. Bobot negatif akan membuat perhitungan pencarian nilai minimum pembaruan nilai $dist$ pada langkah ketiga akan menjadi kacau.

II.3.2 Algoritma A*

Tiga ilmuwan Stanford Research Institute(SRI International), Hart, Nilsson, dan Raphael mempublikasikan Algoritma A* pada tahun 1968. Algoritma A*

ini merupakan perluasan dari Algoritma Dijkstra yang telah ada sebelumnya. Algoritma A* adalah pendekatan yang lebih pintar dibandingkan Dijkstra. Dalam cabang bidang ilmu komputer yang bernama kecerdasan buatan atau *artificial intelligence*, algoritma Dijkstra digolongkan ke dalam *uninformed search algorithm* dan Algoritma A* digolongkan ke dalam *informed search algorithm*.

Langkah-langkah untuk menjalankan Algoritma A* hampir sama dengan Algoritma Dijkstra. Namun, terdapat perbedaan dalam perhitungan bobot untuk pembaruan nilai minimum variable *dist*. Jika Algoritma Dijkstra hanya bertumpu pada bobot sisi antara simpul bertetangga yang sedang dihitung, Algoritma A* juga mempertimbangkan hubungan heuristik antara keduanya seperti yang dapat dilihat pada Gambar II.3. Lebih formalnya, Algoritma A* memilih sisi yang meminimalkan fungsi $f(x) = g(x) + h(x)$ dengan x adalah simpul berikutnya pada jalur, $g(x)$ adalah jarak jalur dari simpul awal ke x , dan $h(x)$ adalah fungsi heuristik yang mengestimasi jarak jalur termurah dari x ke tujuan.

Dijkstra	A*
	
$f(x) = g(x)$	$f(x) = g(x) + \underline{h(x)}$

Gambar II.3 Perbandingan Algoritma Dijkstra dan A*

Komputasi hubungan heuristik ini dengan beberapa fungsi, namun fungsi heuristik ini harus memenuhi syarat *admissable*, yaitu fungsi terjamin untuk mengembalikan jalur dengan jarak terpendek dari awal ke tujuan. Pada medan grid misalnya digunakan *Manhattan Distance* dan *Euclidean Distance*.

$$MH(a, b) = |a_x - b_x| + |a_y - b_y| \quad (II.1)$$

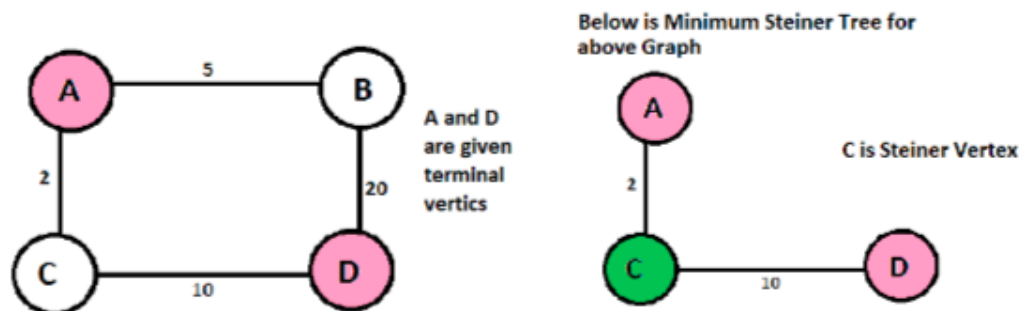
$$ED(a,b) = \sqrt{(a_x^2 - b_x^2) + (a_y^2 - b_y^2)} \quad (\text{II.2})$$

dengan:

- a dan b adalah simpul graf
- a_x dan a_y adalah titik kordinat (x,y) dari a

II.3.3 Algoritma *Steiner Tree*

Steiner Tree adalah permasalahan pencarian subgraf yang mencakup seluruh simpul dalam suatu list yang terdapat dalam graf. Lebih formalnya, L. Kou dan Berman menjelaskan bahwa persoalan Steiner Tree dijelaskan sebagai berikut: diberikan sebuah graf $G = (V, E)$, dan sebuah *list of nodes* L , temukan sebuah pohon(subset graf G) yang mencakup seluruh simpul dari L . Permasalahan Steiner Tree mirip dengan Spanning Tree. Dalam Gambar II.4, dapat dilihat bahwa perbedaannya adalah jika Minimum Spanning Tree mencakup seluruh simpul dalam graf, Steiner Tree hanya mencakup sebagian simpul.



Gambar II.4 Perbandingan permasalahan *Spanning Tree* dan *Steiner Tree*

Sumber: <https://www.geeksforgeeks.org/steiner-tree/>

Dalam penelitian tersebut, juga disebutkan bahwa persoalan Steiner Tree termasuk dalam kategori NP-Hard. Artinya, tidak ada solusi polinomial untuk persoalan ini. Oleh karena itu, beberapa pihak mencoba mencari solusi polinomial dengan pendekatan aproksimasi.

Salah satu usaha dilakukan oleh L. Kou dan Berman, yang membangun Algoritma Steiner Tree sebagai berikut:

1. Bangun *complete distance graph* $G_1 = (V_1, E_1, d_1)$, dengan $V_1 = S$, dan untuk tiap $(v_i, v_j) \in E_1$, $d_1(v_i, v_j)$ sama dengan jarak terpendek dari v_i ke v_j di graf G .
2. Temukan G_2 , dengan G_2 *minimum spanning tree* dari G_1
3. Bangun subgraf G_3 dari G dengan mengganti nilai tiap sisi di G_2 dengan nilai jarak terpendeknya di G .
4. Temukan G_4 , dengan G_4 *minimum spanning tree* dari G_3
5. Bangun Steiner Tree G_5 dari G_4 dengan menghapus sisi sehingga semua simpul di G_5 adalah titik steiner.

II.4 Kakas Pendukung Pengembangan

Dalam pengerjaannya tugas akhir ini, digunakan tiga buah kakas pendukung, yaitu **networkx**, **matplotlib**, dan **jupyter notebook**.

II.4.1 NetworkX

NetworkX adalah *library* Python yang sering digunakan untuk menganalisis *graph* atau *network*. NetworkX menyediakan struktur data *graf* dalam representasi yang mirip dengan *adjacency list* namun lebih baik, yaitu *dictionary of dictionary*. Representasi ini lebih baik dari *adjacency list* karena mampu untuk mengakses simpul manapun dengan cepat melalui *hash value*-nya.

Graf dalam *NetworkX* dapat menampung berbagai macam objek Python asalkan objek tersebut *hashable*. Syarat suatu objek *hashable* dalam Python adalah objek tersebut mempunyai *hash value* dan memiliki properti: jika dua buah contoh dari objek tersebut memiliki *hash value* yang sama, maka dua objek tersebut dianggap objek yang sama.

Selain menyediakan struktur data untuk graf, NetworkX juga menyediakan berbagai macam algoritma untuk mengakuisisi graf untuk berbagai macam kasus seperti: pencarian jalur terpendek, pencarian sentralitas, pengelompokan simpul, distribusi derajat, dan banyak kasus lainnya. Dalam hal kompatibilitas, NetworkX mampu membaca dan menulis graf dalam berbagai macam format seperti GEFX, GML, Pickle, GraphML, JSON, LEDA, Shapefile, dan lainnya dalam pengembangan.

II.4.2 Matplotlib

Matplotlib adalah sebuah pustaka python yang dapat digunakan untuk membuat berbagai macam visualisasi. Matplotlib menyediakan berbagai macam API untuk menyematkan visualisasi ke dalam aplikasi berbasis grafis seperti Tkinter, wxPython, Qt, ataupun GTK.

Pada dasarnya, NetworkX pada dasarnya bukan pustaka yang cukup untuk melakukan visualisasi. NetworkX hanya menyajikan beberapa visualisasi graf dasar. Sementara itu, Matplotlib dapat digunakan untuk melakukan kustomisasi lanjut. Kedua pustaka dapat bekerja sama untuk menghasilkan visualisasi yang baik dan cepat. Visualisasi dapat dibentuk dengan NetworkX terlebih dahulu, kemudian baru dilakukan kustomisasi dengan Matplotlib.

BAB III

ANALISIS MASALAH DAN RANCANGAN SOLUSI

III.1 Analisis Permasalahan

III.1.1 Definisi Fungsi

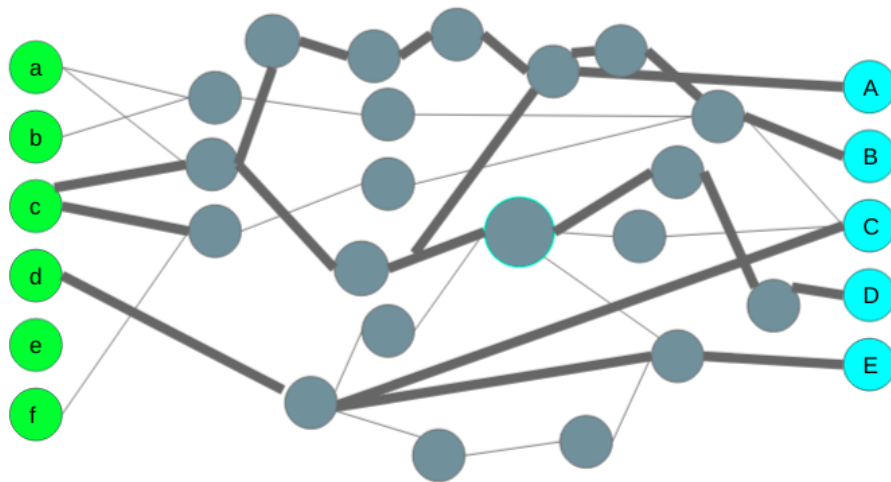
Permasalahan pencarian rute pembelajaran kanji dalam *knowledge graph* ini dapat diformalkan dengan fungsi sebagai berikut:

$$G' = findRoute(MOrig, MDest) \quad (III.1)$$

dengan:

- *MOrig* adalah *list* kanji yang sudah dipelajari. Dapat diasumsikan *list* ini berisi yang lebih populer di kalangan pelajar pemula. Misal, $MOrig = [\text{日、月、字}]$.
- *MDest* adalah *list* kanji yang ingin dipelajari atau kanji yang memiliki frekuensi lebih rendah dalam percakapan sehari-hari. Misal, $MDest = [\text{皿、屋、機}]$.
- G' adalah subgraf yang menggambarkan rute simpul yang harus dilewati.

Arti dan penggunaan fungsi *findRoute* di atas akan dijelaskan dengan Gambar III.1. Jadi, hal yang pertamakali dilakukan pengguna adalah memberikan dua buah masukan. Dalam Gambar III.1, masukannya adalah simpul a, b, c, d, e, dan f yang berwarna hijau dan keluarannya adalah simpul A,B,C,D,E,dan F yang berwarna biru. Kemudian, sistem akan menjalankan fungsi tersebut untuk menemukan subgraf antara yang menghubungkan kedua masukan tadi. Pada Gambar III.1, subgraf ini ditandai dengan sisi yang lebih tebal. Subgraf ini lah yang akan diberikan sebagai keluaran oleh sistem dan akan digunakan oleh pengguna untuk membantunya mempelajari kanji tujuannya.



Gambar III.1 Penjelasan Fungsi *findRoute*

Dalam permasalahan logistik pada penelitian Barták, Dovier, dan Zhou, studi kasus yang dipakai adalah permasalahan di dunia logistik untuk menghubungkan dua buah kota yang masing-masing punya beberapa terminal truk dengan biaya pengantaran seminimal mungkin[6]. Jika kita menari permasalahan ini ke permasalahan kanji, maka sebuah terminal truk di dua kota tersebut dapat disimbolkan simpul kanji. Lalu, kumpulan terminal dalam sebuah kota dapat diandaikan sebagai sebuah daftar kanji awal atau kanji target. Sedangkan, tujuan untuk menghubungkan dua buah kota ini dapat diandaikan sebagai tujuan untuk menghubungkan dua buah kumpulan kanji. Sehingga dapat dikatakan bahwa fungsi objektif yang ditetapkan oleh penelitian Barták, Dovier, dan Zhou dapat diterapkan juga dalam tugas akhir ini.

III.1.2 Pilihan Algoritma

Akan ada 3 algoritma yang akan diperbandingkan pada tugas akhir ini, yaitu Algoritma Dijkstra, Algoritma A*, dan Algoritma Steiner Tree.

III.1.2.1 Algoritma Dijkstra

Algoritma pertama yang akan diuji untuk menyelesaikan permasalahan MOMD, *Algoritma Dijkstra*, dilakukan dengan cara membangkitkan semua kemungkinan pasangan tiap elemen dari *list MOrig* dan *list MDest*, mencari *shortest path* antar

dua simpul tersebut dengan algoritma *Dijkstra*, dan terakhir mencari gabungan dari subgraf *path* yang terbentuk. Algoritma 1 memberikan langkah-langkah ini.

Algorithm 1: Algoritma Dijkstra

Input: $MOrig$, daftar kanji asal yang sudah dipelajari.

$MDest$, daftar kanji yang akan dipelajari

G_{world} , graf yang berisi seluruh kanji dan atributnya

Output: G_{result} , subgraf G_{world} yang menghubungkan $MOrig$ dan $MDest$

```

1 Function FindRouteBruteForce( $MOrig, MDest, G_{world}$ ):
2    $list_{paths} \leftarrow []$ 
3   foreach  $o \in MOrig$  do
4     foreach  $d \in MDest$  do
5        $path \leftarrow shortestPath(o, d)$ 
6        $list_{paths} \leftarrow list_{paths} + path$ 
7    $G_{result} \leftarrow union(list_{paths})$ 
8   return  $G_{result}$ 
9 End Function

```

Tentu, solusi ini mungkin dan cukup mudah untuk diimplementasikan. Namun, solusi ini memiliki kompleksitas algoritma yang cukup buruk dan akan memakan waktu yang cukup lama.

III.1.2.2 Algoritma A*

Algoritma kedua, A*, memiliki langkah yang cukup mirip dengan Algoritma Dijkstra seperti yang dapat dilihat pada Algoritma 2. Perbedaannya, pada komputasi *shortestPath* yang melibatkan fungsi heuristik. Algoritma A* diharapkan lebih dapat

memilih sisi total yang minimum dengan bantuan fungsi heuristik yang digunakan.

Algorithm 2: Algoritma A*

Input: $MOrig$, daftar kanji asal yang sudah dipelajari.

$MDest$, daftar kanji yang akan dipelajari

G_{world} , graf yang berisi seluruh kanji dan atributnya

$H_{heuristic}$, fungsi heuristik yang dipakai

Output: G_{result} , subgraf G_{world} yang menghubungkan S dan D

1 **Function** FindRouteHeuristic($MOrig$, $MDest$, G_{world}):

```

2    $list_{paths} \leftarrow []$ 
3   foreach  $o \in MOrig$  do
4       foreach  $d \in Mdest$  do
5            $path \leftarrow ShortestPath(o, d, h)$ 
6            $list_{paths} \leftarrow list_{paths} + path$ 
7    $G_{world} \leftarrow union(list_{paths})$ 
8   return  $G_{world}$ 

```

9 **End Function**

Akan ada dua fungsi heuristik yang akan digunakan dalam implementasi Algoritma A*, yaitu *Common Neighbor*(Persamaan III.2) dan *Jaccard Similarity*(Persamaan III.3). Perbedaan dari kedua fungsi heuristik tersebut adalah JS memiliki pembagi $|N(a) \cup N(b)|$ yang berguna untuk menghilangkan bias pada simpul yang memiliki derajat tinggi. Dalam *knowledge graph* kanji, simpul yang memiliki derajat tinggi adalah simpul yang memiliki radikal banyak seperti 𠂇 yang memiliki 5 buah radikal, yaitu 丨, 儿, 𠂇, 尸, 山, dan 穴.

$$CN(a, b) = |N(a) \cap N(b)| \quad (III.2)$$

$$JS(a, b) = \frac{|N(a) \cap N(b)|}{|N(a) \cup N(b)|} \quad (III.3)$$

dengan:

- a dan b adalah pasangan simpul yang ingin dihitung.
- $N(a)$ adalah himpunan tetangga(neighbor) dari simpul a
- $|N(a)|$ adalah banyak anggota himpunan $N(a)$

III.1.2.3 Algoritma Steiner Tree

Algoritma terakhir, Steiner Tree, mencoba menyelesaikan permasalahan ini dengan sudut pandang yang berbeda, yaitu dengan memperlakukan permasalahan ini sebagai sebuah permasalahan *tree* dibanding permasalahan *graf*.

Penyelesaian dengan pendekatan permasalahan *tree* dapat dilakukan karena permasalahan Steiner Tree dapat direduksi menjadi permasalahan MOMD. Dalam Steiner Tree, subgraf hasil yang diinginkan harus mencakup seluruh simpul terminal L . Sedangkan, dalam MOMD, subgraf hasil yang diinginkan adalah seluruh simpul yang mencakup daftar simpul asal $MOrig$, daftar simpul tujuan $MDest$, dan simpul-simpul antara penghubungnya V' .

Salah satu versi Algoritma Steiner Tree dibangun oleh L. Kou dan Berman. Algoritma ini lah yang akan digunakan sebagai basis implementasi pada Algoritma 3. Algoritma versinya memiliki langkah sebagai berikut:

1. Bangun *complete distance graph* $G_1 = (V_1, E_1, d_1)$, dengan $V_1 = S$, dan untuk tiap $(v_i, v_j) \in E_1$, $d_1(v_i, v_j)$ sama dengan jarak terpendek dari v_i ke v_j di graf G .
2. Temukan G_2 , dengan G_2 *minimum spanning tree* dari G_1
3. Bangun subgraf G_3 dari G dengan mengganti nilai tiap sisi di G_2 dengan nilai jarak terpendeknya di G .
4. Temukan G_4 , dengan G_4 *minimum spanning tree* dari G_3
5. Bangun Steiner Tree G_5 dari G_4 dengan menghapus sisi sehingga semua simpul di G_5 adalah titik steiner.

Algorithm 3: Algoritma Steiner Tree

Input: S , daftar kanji asal yang sudah dipelajari.

D , daftar kanji yang akan dipelajari

G_{world} , graf yang berisi seluruh kanji dan atributnya

Output: G_{result} , subgraf G_{world} yang menghubungkan S dan D

1 **Function** FindRouteSteinerTree(S, D, G_{world}):

```
2    $terminal_{nodes} \leftarrow D + S$ 
3    $M \leftarrow \text{empty graph}$ 
4    $Gnodes \leftarrow \text{emptyset}$ 
5   foreach  $u, (dist, path) \in allPathsIter(G)$  do
6        $Gnodes.remove(u)$ 
7       foreach  $v \in Gnodes$  do
8            $M.add_{edge}(u, v, dist, path)$ 
9    $H \leftarrow M.subGraph(terminal_{nodes})$ 
10   $mst_{edges} \leftarrow minimumSpanningTree(H)$ 
11   $T \leftarrow G.edgesubgraph(edges)$ 
12   $\rightarrow T$ 
```

13 **End Function**

Jika diperhatikan, komputasi dari baris ke-3 sampai ke-8 tidak melibatkan variabel $terminal_{nodes}$ dan hanya melibatkan G_{world} . Oleh karena itu, komputasi ini dapat dilakukan di luar fungsi utama tersebut. Akan didefinisikan algoritma *metricClosure*

seperti yang dapat dilihat pada Algoritma 4.

Algorithm 4: Algoritma *Metric Closure*

Input: G , daftar kanji asal yang sudah dipelajari.

Output: M , metric closure dari graf G

```
1 Function MetricClosure( $G_{world}$ ):  
2    $M \leftarrow \text{empty graph}$   
3    $Gnodes \leftarrow \text{emptyset}$   
4   foreach  $u, (dist, path) \in allPathsIter(G)$  do  
5      $Gnodes.remove(u)$   
6     foreach  $v \in Gnodes$  do  
7        $M.add_{edge}(u, v, dist, path)$   
8    $\rightarrow M$   
9 End Function
```

Setelah ekstraksi komputasi tersebut, fungsi *metricClosure* dapat dipanggil oleh Algoritma Steiner Tree sebagaimana diperlihatkan pada Algoritma 5.

Algorithm 5: Algoritma Steiner Tree setelah Ekstraksi Fungsi *Metric Closure*

Input: S , daftar kanji asal yang sudah dipelajari.

D , daftar kanji yang akan dipelajari

G_{world} , graf yang berisi seluruh kanji dan atributnya

Output: G_{result} , subgraf G_{world} yang menghubungkan S dan D

```
1  $m_{cg} \leftarrow metricClosure(G)$   
2 Function FindRouteSteinerTree( $S, D, G_{world}$ ):  
3    $M \leftarrow m_{cg}$   
4    $terminal_{nodes} \leftarrow D + S$   
5    $H \leftarrow M.subGraph(terminal_{nodes})$   
6    $mst_{edges} \leftarrow minimumSpanningTree(H)$   
7    $T \leftarrow G.edgesubgraph(edges)$   
8    $\rightarrow T$   
9 End Function
```

Kemudian, sebenarnya komputasi **metricClosure** pada tersebut bisa dilakukan

dengan lebih baik. Algoritma 6 memakai teknik *dynamic programming* dapat dimanfaatkan untuk mengoptimasi penggunaan *space* pada program.

Dalam Algoritma 6, komputasi $shortestPath(G, u, v)$ dilakukan dengan prinsip *lazy loading*. Untuk melakukannya diperlukan tabel tabulasi bernama *DP* seperti yang dapat dilihat pada baris 1 program. Kemudian, pada baris 8 sampai 13, setiap akan dilakukan komputasi $shortestPath(G, u, v)$, akan diperiksa terlebih dahulu apakah ini pasangan u, v yang pernah dikomputasi sebelumnya dan disimpan di dalam *DP*. Jika iya, ambil nilai yang ada dalam variabel *DP* tersebut. Jika tidak, lakukan

komputasi sebagaimana seharusnya.

Algorithm 6: Algoritma Steiner Tree + *Dynamic Programming*

Input: S , daftar kanji asal yang sudah dipelajari.

D , daftar kanji yang akan dipelajari

G_{world} , graf yang berisi seluruh kanji dan atributnya

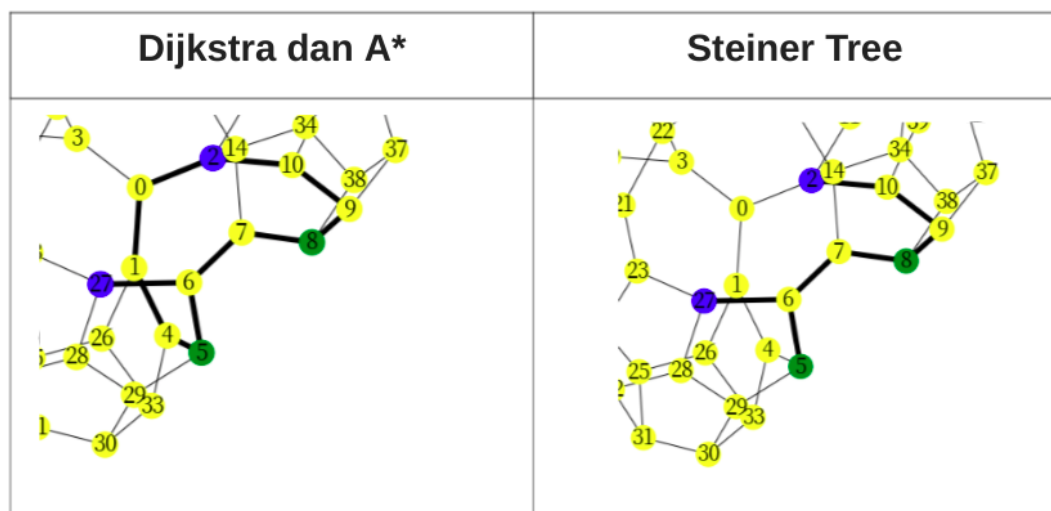
Output: G_{result} , subgraf G_{world} yang menghubungkan S dan D

```
1  $DP \leftarrow \text{empty table}$ 
2 Function FindRouteSteinerTree( $S, D, G_{world}$ ):
3    $terminal_{nodes} \leftarrow D + S$ 
4    $H \leftarrow \text{empty graph}$ 
5    $Gnodes \leftarrow \text{empty set}$ 
6   foreach  $u \in terminal_{nodes}$  do
7      $Gnodes.remove(u)$ 
8     foreach  $v \in Gnodes$  do
9       if  $(u, v)$  in  $DP$  then
10          $dist, path \leftarrow DP[(u, v)]$ 
11       else
12          $path \leftarrow shortestPath(G, u, v)$ 
13          $DP[u, v] \leftarrow (dist, len(dist))$ 
14        $H.add_{edge}(u, v, dist, path)$ 
15    $H \leftarrow M.subGraph(terminal_{nodes})$ 
16    $mst_{edges} \leftarrow minimumSpanningTree(H)$ 
17    $T \leftarrow G.edgesubgraph(edges)$ 
18    $\rightarrow T$ 
19 End Function
```

III.1.3 Metriks Evaluasi

Dalam tugas akhir ini, akan ada dua metriks yang akan digunakan:

1. **Kecepatan**, secara analisis dengan *time complexity* maupun empiris. Metriks ini digunakan dengan alasan algoritma yang lebih cepat akan memberikan pengalaman pengguna yang lebih baik. Analisis *Time complexity* digunakan untuk memeriksa kecepatan algoritma independen terhadap *noise* yang mungkin terjadi selama eksekusi, sedangkan kecepatan empiris dilakukan untuk memeriksa *overhead* yang terjadi dalam implementasi program.
2. **Jumlah *simpul* antara yang terpakai di graf hasil**. Metriks ini digunakan dengan alasan semakin minimal banyak *simpul* non-input yang ada pada hasil, maka semakin dekat, cepat, dan hemat tenaga juga pengguna dalam mempelajari kanji tujuannya. Misalnya, dalam Gambar III.2, untuk belajar kanji biru dari kumpulan kanji hijau butuh 6 simpul antar, sedangkan pada bagian kanan hanya butuh 4 simpul antar. Dengan begitu dapat kita katakan, bahwa algoritma sebelah kanan memiliki hasil yang lebih baik.



Gambar III.2 Perbandingan hasil Dijkstra dan A* versus Steiner Tree

III.2 Rancangan Solusi

III.2.1 Gambaran Umum Solusi

Untuk mencapai solusi tugas akhir ini, diperlukan dua langkah seperti yang diperlihatkan pada Gambar III.3.

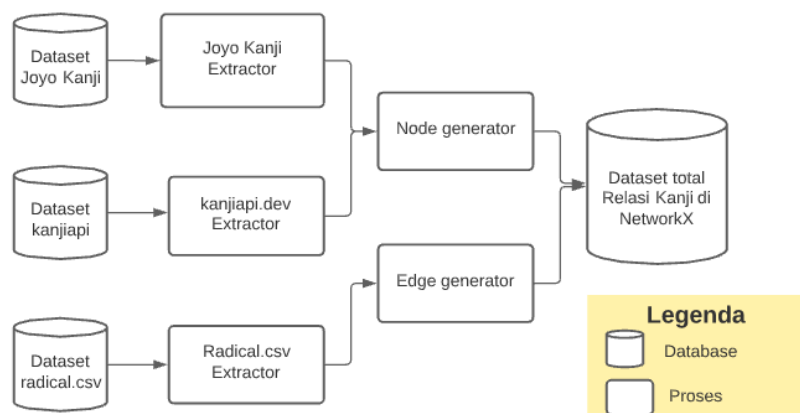


Gambar III.3 Langkah pengerjaan solusi secara umum

Pertama, data huruf kanji akan diekstraksi dari berbagai sumber yang ada. Data ini akan dijadikan sebagai *knowledge graph* dan disimpan di dalam *graph database*. Setelah *knowledge graph* terbentuk, akan dikembangkan algoritma yang memberikan data kanji yang harus dipelajari oleh user. Algoritma ini akan dievaluasi akurasi dan kecepatannya, dan akan divisualisasikan juga hasil subgrafnya.

III.2.2 Persiapan Dataset *Knowledge Graph*

Dataset *knowledge graph* relasi kanji disiapkan dengan melibatkan beberapa tahapan seperti yang diperlihatkan Gambar III.4.



Gambar III.4 Rancangan persiapan dataset *knowledge graph* kanji

Knowledge Graph relasi kanji yang dibangun akan berasal dari beberapa sumber data berbeda. Ada tiga sumber data berbeda yang dipakai dalam tugas akhir ini, yaitu Wikipedia *Joyo Kanji*, daftar radikal kanji dalam bentuk file CSV, dan *kanjiapi.dev*.

Setelah didapat data dari masing-masing sumber, data tersebut akan diintegrasikan menjadi satu dataset yang konsisten dan koheren. Terakhir, dari dataset yang sudah konsisten dan koheren ini, data akan dibentuk menjadi bentuk graf dan dimasukkan ke dalam struktur data *graph* di *NetworkX*.

III.2.2.1 Dataset *Joyo Kanji*

Dataset pertama ditunjukkan pada Gambar III.5. Dataset ini adalah dataset *Joyo Kanji* yang berasal dari Wikipedia. *Joyo Kanji* adalah daftar kanji resmi yang dikeluarkan pemerintah Jepang sebagai standar yang harus dicapai seseorang yang sudah melewati pendidikan wajib (SD-SMP-SMA). Dari dataset ini akan diambil 3 elemen: urutan, kanji, dan arti seperti yang terlihat pada Gambar III.6.

List of jōyō kanji

From Wikipedia, the free encyclopedia

The jōyō kanji system of representing written Japanese consists of 2,136 characters.

Contents [hide]

- List of characters
- See also
- Notes
- External links

#	New	Old	Radical	Strokes	Grade	Year added	English meaning	Readings
1	亜	亜	二	7	5		sub-	ア a
2	哀		口	9	5		pathetic	アイ, あわれ, あわ-れい ai, awa-re, awa-rei
3	挨		手	10	5	2010	push open	アイ ai
4	愛		心	13	4		love	アイ ai
5	曖		日	17	5	2010	not clear	アイ ai

Japanese writing

Components

- Kanji [show]
- Kana [show]
- Typographic symbols [show]

Uses

- Syllabograms [show]
- Romanization [show]
- Rōmaji [show]

V · T · E

Gambar III.5 Dataset *Joyo Kanji* dalam bentuk Website

Dengan statusnya sebagai daftar resmi tersebut, dataset ini juga akan digunakan sebagai landasan kelengkapan data yang akan digunakan sebagai *knowledge graph* dalam tugas ini nantinya. Jadi, *knowledge graph* yang dibangun dikatakan lengkap jika memiliki 2136 kanji.


```
1 num,kanji,meaning
2 1,亜,sub-
3 2,哀,pathetic
4 3,挨,push open
5 4,愛,love
6 5,曖,not clear
7 6,惡,bad
8 7,握,grip
9 8,压,pressure
10 9,扱,handle
11 10,宛,allocate
12 11,嵐,storm
13 12,安,cheap
14 13,案,plan
15 14,暗,dark
```

Gambar III.8 Dataset Hasil Transformasi Radikal

III.2.2.3 Dataset *Kanjiapi.dev*

Gambar III.9 menunjukkan Kanjiapi.dev yang merupakan JSON API buatan seorang pegiat opensource bernama *onlyskin*. Kanjiapi.dev menyediakan lebih dari 13000 kanji. Kanjiapi.dev menyediakan banyak endpoint yang dapat digunakan untuk menarik informasi tentang suatu kanji dari karakternya, cara baca, maupun kelasnya. Contoh respons dari kanjiapi.dev dapat dilihat pada Gambar III.9.



Gambar III.9 Contoh respons kanjiapi.dev

Di sini akan digunakan endpoint **https://kanjiapi.dev/v1/kanji/character** untuk menarik informasi kanji dari karakternya. Dari dataset ini akan diambil 2 elemen: kanji dan macam artinya III.10.

```

1 kanji,meanings
2 亜,Asia:rank next:come after:-ous
3 啞,mute:dumb
4 逢,meeting:tryst:date:rendezvous
5 悪,bad:vice:rascal:false:evil:wrong
6 以,by means of:because:in view of:compared with
7 伊,Italy:that one
8 井,well:well crib:town:community
9 稻,rice plant
10 印,stamp:seal:mark:imprint:symbol:emblem:trademark:evidence:souvenir:India
11 引,pull:tug:jerk:admit:install:quote:refer to
12 鵜,cormorant
13 丑,sign of the ox or cow:1-3AM:second sign of Chinese zodiac
14 渦,whirlpool:eddy:vortex

```

Gambar III.10 Dataset Hasil Transformasi kanjiapi.dev

III.2.2.4 Mengagregasi Data dari Seluruh Dataset

Daftar skema tiap dataset dan transformasi yang akan dilakukannya ditunjukkan pada tabel III.1.

Dataset	Skema Awal	Transformasi
Joyo Kanji	Kanji → Arti	Kanji → Arti
Radikal.csv	Radikal → Daftar Kanji	Kanji → Daftar Radikal
kanjiapi.dev	Kanji → (Kanji, ragam atribut)	Kanji → (Kanji, Daftar Arti)

Tabel III.1 Daftar Transformasi Dataset

Dataset Joyo Kanji dan kanjiapi.dev akan digunakan untuk mendapatkan informasi tentang simpul kanji. Sedangkan dataset radical.csv akan digunakan untuk membangkitkan relasi kanji dan radikalnya.

```
{'堯-kanji': {'color': 'red',
              'idx': 2070,
              'meaning': 'high',
              'symbol': '堯',
              'visual': '堯\nhigh'},
 '整-kanji': {'color': 'red',
              'idx': 125,
              'meaning': 'organize',
              'symbol': '整',
              'visual': '整\norganize'},
 '樟-kanji': {'color': 'red',
              'idx': 3279,
              'meaning': 'camphor',
              'symbol': '樟',
              'visual': '樟\ncamphor'},
 '湊-kanji': {'color': 'red',
              'idx': 1136,
              'meaning': 'rising waters',
              'symbol': '湊',
              'visual': '湊\nrising waters'},
 ...}
```

Gambar III.11 Representasi simpul pembelajaran *Knowledge Graph* relasi kanji

Pada Gambar III.11, kita dapat melihat dalam sebuah simpul kanji, terkandung informasi berupa warna representasinya dalam *knowledge graf* nanti, id, arti, simbol, dan karakter teks representasi. Kemudian, pada Gambar III.12, diperlihatkan bahwa tiap sisi pada daftar sisi akan berisi satu pasangan kanji-relasi. Jika sebuah kanji memilih lebih dari satu radikal, maka akan disebar ke dalam beberapa elemen.

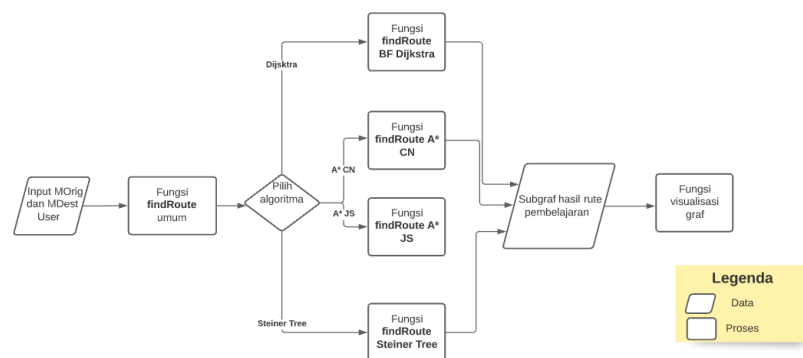
```
In [19]: full_edges = get_graph_edge(data_edges)
full_edges[:10]

Out[19]: [('垂-kanji', '口-radical'),
('啞-kanji', '丿-radical'),
('啞-kanji', '口-radical'),
('逢-kanji', '欠-radical'),
('逢-kanji', '辶-radical'),
('逢-kanji', '二-radical'),
('惡-kanji', '心-radical'),
('惡-kanji', '口-radical'),
('以-kanji', '人-radical'),
('伊-kanji', '厶-radical')]
```

Gambar III.12 Representasi sisi *Knowledge Graph* relasi kanji

III.2.3 Rancangan Solusi Pencarian Rute *Knowledge Graph* Relasi Kanji

Rancangan solusi pencarian rute dalam *Knowledge Graph* Relasi Kanji melibatkan beberapa tahapan seperti yang diperlihatkan Gambar III.13.



Gambar III.13 Langkah pengerjaan algoritma pencarian rute pembelajaran *Knowledge Graph* relasi kanji

Masukan **MOrig** dan **MDest** dari pengguna akan ditransformasi terlebih dahulu sesuai dengan format data dari *simpul* yang didefinisikan. Setelah sesuai, data masukan akan diproses oleh fungsi **findRoute** umum yang memberikan pilihan kepada pengguna untuk menggunakan salah satu algoritma dari tiga pilihan: Dijkstra, A* CN, A* JS dan Steiner Tree. Data masukan pengguna akan diproses

oleh masing-masing algoritma yang tiap definisinya terdapat di bagian Analisis Masalah. Tiap algoritma menghasilkan sebuah subgraf yang merepresentasikan rute pembelajaran kanji yang harus ditempuh oleh pengguna. Subgraf hasil akan ditampilkan melalui bantuan pustaka NetworkX dan Matplotlib. NetworkX akan menampilkan visualisasi dasar. Lalu, dengan visualisasi dasar tersebut dapat dilakukan kustomisasi dengan Matplotlib. Contoh kustomisasi yang dilakukan adalah dengan memberikan opsi representasi simpul melalui salah satu atributnya maupun kombinasi dari dua atau lebih atributnya.

BAB IV

EKSPERIMEN DAN ANALISIS

Bab 4 akan mendiskusikan hasil eksperimen dan analisis yang dilakukan.

IV.1 Kasus Uji

Kasus uji yang digunakan ditampilkan pada tabel IV.1. Kasus uji ini berisikan **MOrig** dan **MDest**. **MOrig** adalah daftar kanji yang sudah dipelajari dan **MDest** adalah daftar kanji yang akan dipelajari. Kasus uji ini akan bertingkat mulai **tc_ias_1** yang hanya memiliki total 6 simpul sampai kasus uji **tc_ias_6** yang melibatkan 1129 simpul.

IV.2 Analisis *Time Complexity*

Berikut adalah hasil analisis *Time Complexity* yang dihasilkan dari tiap algoritma yang dipakai.

IV.2.1 Algoritma Brute Force Dijkstra

Pada Algoritma IV.1, dapat dilihat bahwa sumber utama kompleksitas Algoritma Dijkstra adalah dua buah perulangan bersarang untuk **MOrig** dan **MDest**. Kompleksitas untuk dua buah perulangan ini adalah $O(|MOrig| * |MDest|)$. Lalu, untuk fungsi **nx.dijkstra_path**-nya sendiri memiliki kompleksitas $O((|Vertices_G| + |Edges_G|) \log |Vertices_G|)$. Sehingga, kompleksitas total dari Algoritma IV.1 adalah $O((|MOrig| * |MDest|) * (|Vertices_G| + |Edges_G|) \log |Vertices_G|)$.

Algoritma IV.1 Algoritma Brute Force Dijkstra

```
def find_route_bf(G: nx.Graph, MOrig: List, MDest: List) -> nx.Graph:
    result = []
    for kin in MOrig: #  $O(|MOrig|)$ 
        for kout in MDest: #  $O(|MDest|)$ 
```

Kasus Uji	Jumlah Simpul		
	MOrig	MDest	Jumlah
tc_ias_1	3	3	6
tc_ias_2	33	66	99
tc_ias_3	100	200	300
tc_ias_4	200	215	415
tc_ias_5	400	429	829
tc_ias_6	429	700	1129

Tabel IV.1 Daftar kasus uji

```

sp_raw = nx.dijkstra_path(G, source=kin, target=kout)
#  $O((|GV|+|GE|) \log |GV|)$ 
sp_graph = generate_graph(G, sp_raw)
result.append(sp_graph)

return nx.compose_all(result)

```

IV.2.2 Algoritma A*

Algoritma A* mirip dengan Algoritma Dijkstra, sehingga kompleksitas keduanya pun sama. Kompleksitas Algoritma A* berasal dua buah perulangan untuk **MOrig** dan **MDest**. Kompleksitas untuk dua buah perulangan ini adalah $O(|MOrig| * |MDest|)$. Lalu, untuk fungsi **nx.dijkstra_path**-nya sendiri memiliki kompleksitas $O((|Vertices_G| + |Edges_G|) \log |Vertices_G|)$.

Perbedaannya dapat dilihat pada komputasi jarak terdekat dua buah simpul di Algoritma IV.2 yang menggunakan fungsi heuristik. Namun, fungsi heuristik yang digunakan pun baik *common neighbor* ataupun *jaccard function* dapat dilakukan dengan $O(\min(|N_u|, |N_v|))$ karena graf disimpan dalam representasi **adjacency list**. Dari komponen-komponen tadi, didapatkan kompleksitas total dari fungsi ini adalah $O((|MOrig| * |MDest|) * (|Vertices_G| + |Edges_G|) \log |Vertices_G|)$.

Algoritma IV.2 Algoritma A*

```
def find_route_astar(G: nx.Graph, MOrig: List, MDest: List,  
    ↪ heuristic_func) -> nx.Graph:  
    result = []  
    for kin in MOrig: #  $O(|MOrig|)$   
        for kout in MDest: #  $(|MDest|)$   
            sp_raw = nx.astar_path(G, source=kin, target=kout,  
                ↪ heuristic=heuristic_func)  
            #  $O((|GV|+|GE|) \log |GV|)$   
            sp_graph = generate_graph(G, sp_raw)  
            result.append(sp_graph)  
    return nx.compose_all(result)
```

IV.2.3 Algoritma Steiner Tree

Dari Algoritma IV.3, dapat dilihat beban komputasi terbesar Algoritma Steiner Tree terdapat pada pemanggilan fungsi **nx.allPairsDijkstra** yang terdapat pada fungsi **metricClosure**. Fungsi untuk membangkitkan *shortest path* untuk semua pasangan simpul pada graf memiliki kompleksitas $O(|Vertex_G|^2)$.

Algoritma IV.3 Fungsi Metric Closure

```
def metric_closure(G, weight="weight"):  
    M = nx.Graph()  
    Gnodes = set(G)  
    all_paths_iter = nx.all_pairs_dijkstra(G, weight='weight') #  $O(|$   
        ↪  $GV|^2)$   
    u, (distance, path) = next(all_paths_iter)  
    for u, (distance, path) in all_paths_iter:  
        Gnodes.remove(u)  
        for v in Gnodes:  
            M.add_edge(u, v, distance=distance[v], path=path[v])  
  
    return M
```

Fungsi **metricClosure** dapat dilakukan diluar fungsi utama sebagai Algoritma IV.4 untuk digunakan dalam fungsi utama Algoritma Steiner Tree. Fungsi utama ini sendiri tidak memiliki langkah yang melebihi kompleksitas fungsi **metricClosure**. Kemudian, kompleksitas fungsi ini bergantung pada besarnya **terminalNodes** yang diterima. Komponen **terminalNodes** ini sendiri berasal dari jumlah **MOrig** dan **MDest**. Berdasarkan pembahasan di atas, didapatkan kompleksitas waktu total solusi dengan pendekatan Steiner Tree ini adalah sebesar $O((|MOrig| + |MDest|) * |Vertices_G|^2)$

Algoritma IV.4 Algoritma *My Steiner Tree*

```
def my_steiner_tree(G, terminal_nodes, weight="weight", is_mcg=True):
    global mcg
    # H is the subgraph induced by terminal_nodes in the metric
    #   ⇨ closure M of G.
    if is_mcg:
        M = mcg
    else:
        M = metric_closure(G, weight=weight) # O(|GV|^2)
    H = M.subgraph(terminal_nodes) # O(|GV|^2) * O(|MOrig| + |MDest|)
    mst_edges = nx.minimum_spanning_edges(H, weight="distance", data=
        #   ⇨ True) # O (|GE| log GV)
    edges = chain.from_iterable(pairwise(d["path"]) for u, v, d in
        #   ⇨ mst_edges)
    T = G.edge_subgraph(edges)
    return T
```

IV.2.4 Optimasi Algoritma *Steiner Tree*

Perbaikan selanjutnya dilakukan pada Algoritma Steiner Tree versi Algoritma IV.4 dilakukan dengan Algoritma IV.5. Algoritma IV.5 menggunakan *dynamic programming* dan *lazy loading*. Perubahan ini membawa dampak positif pada

pemakaian *space* pada program karena komputasi *shortest path* untuk pasangan simpul yang dilakukan secara seperlunya sesuai simpul yang diminta.

Algoritma IV.5 Algoritma *My Steiner Tree* dengan pendekatan *Dynamic Programming* dan *Lazy Loading*

```
def my_steiner_tree(G, terminal_nodes, weight="weight"):
    global dp

    # Step 1,2,3: Compute metric_closure complete undirected distance
    ↪ graph
    H = nx.Graph()
    Gnodes = set(G)
    for u in terminal_nodes:
        Gnodes.remove(u)
        for v in Gnodes:
            if (u,v) in dp:
                dist, path = dp[(u,v)]
            else:
                path = nx.dijkstra_path(G, u, v)
                dist = len(path)
                dp[(u,v)] = (dist, path) # memoization
            H.add_edge(u, v, distance=dist, path=path)

    # Step 4: Find MST of H
    mst_edges = nx.minimum_spanning_edges(H, weight="distance")

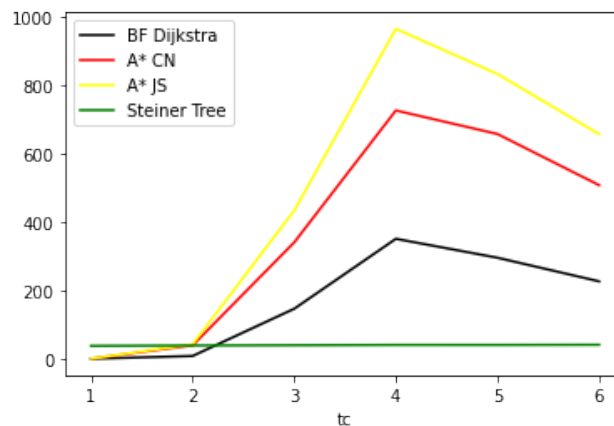
    # Step 5: Construct Steiner Tree from mst_edges
    edges = chain.from_iterable(pairwise(d["path"]) for u, v, d in
    ↪ mst_edges)
    T = G.edge_subgraph(edges)

    return T
```

Kasus Uji	Algoritma			
	BF Dijkstra	A*		Steiner Tree
		CN	JS	
tc_ias_1	0.04	0.16	0.17	37.7
tc_ias_2	8.11	39.21	42.16	38.85
tc_ias_3	146.37	340.68	434.08	39.6
tc_ias_4	351.03	726.61	965.62	40.46
tc_ias_5	295.6	657.67	833.11	40.43
tc_ias_6	226.46	507.98	657.18	40.95

Tabel IV.2 Hasil uji waktu empiris dalam bentuk tabel(dalam detik)

IV.3 Hasil dan Analisis Uji Waktu Empiris



Gambar IV.1 Hasil uji waktu empiris dalam bentuk grafik

Tabel IV.2 dan Gambar IV.1 memberikan hasil pengujian tabel empiris. Dari hasil tersebut dapat dilihat bahwa ternyata algoritma **A*** tidak memberikan hasil yang baik daripada **BF Dijkstra**. Meskipun, sebenarnya waktu yang dibutuhkannya sebanding lurus dengan **BF Dijkstra**. Ada dua buah kemungkinan yang dapat menyebabkan terjadinya proses ini. Pertama, proses komputasi fungsi heuristik menggunakan mekanisme *callback* akan memberikan *overhead* pada *running time* program karena dibutuhkannya proses alokasi *call stack*. Kemungkinan kedua adalah kesalahan

Kasus Uji	Steiner Tree + DP(ya/tidak)	
	Tidak	Ya
tc_ias_1	37.7	0.05
tc_ias_2	38.85	0.007
tc_ias_3	39.6	2.43
tc_ias_4	40.46	0.73
tc_ias_5	40.43	1.481
tc_ias_6	40.95	2.53

Tabel IV.3 Hasil uji optimasi Steiner Tree dengan memoisasi(dalam detik)

pemilihan fungsi heuristik itu sendiri. Mungkin saja bahwa kedua fungsi heuristik yang digunakan, baik *common neighbor* maupun *jaccard similarity* tidak memeneuhi syarat fungsi heuristik untuk algoritma A*, yaitu harus *admissable*.

Lalu, untuk Algoritma Steiner Tree, algoritma ini memberikan performa yang lebih baik dibandingkan algoritma A* maupun BF Dijkstra. Algoritma ini memiliki waktu tempuh algoritma tetap stabil meskipun jumlah simpul yang dipakai membesar. Namun, untuk kasus uji yang kecil, Algoritma Steiner Tree memiliki performa yang lebih buruk dibandingkan algoritma A* maupun BF Dijkstra. Penyebabnya adalah adanya proses komputasi untuk seluruh pair di graf pada fungsi **metricClosure**. Fungsi ini hanya bergantung pada besarnya graf **G** dan tidak bergantung pada **MOrig** atau **MDest**. Sehingga, untuk kasus uji yang kecil proses ini banyak membuang sumber daya komputasi yang sebenarnya tidak perlu.

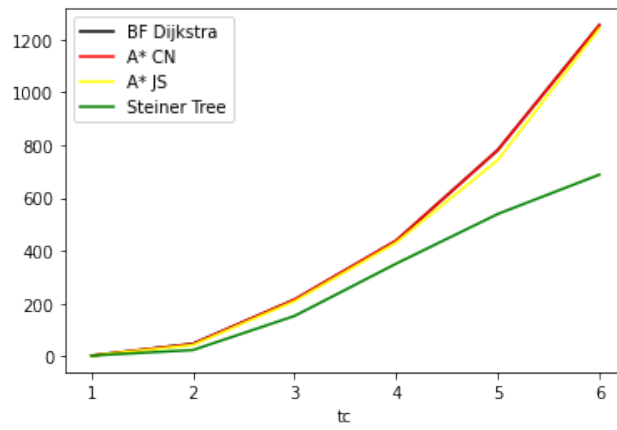
Adapun, solusi untuk menangani hal ini adalah dengan cara menyimpan hasil **metricClosure** terlebih dahulu di luar algoritma utama Steiner Tree. Teknik ini lazim disebut dengan memoisasi. Jika memoisasi pada **metricClosure** sudah dilakukan dan disimpan di variable tertentu, misalnya **mcg**. Fungsi **metricClosure** ini dapat dipanggil dalam $O(1)$ untuk setiap kasus uji. Pada tabel IV.3, dapat dilihat bahwa cara ini mempercepat eksekusi algoritma dari sekitar 40 detik menjadi di bawah 2 detik.

Test Case	Algoritma			
	BF Dijkstra	A*		Steiner Tree
		CN	JS	
tc_ias_1	3	3	3	3
tc_ias_2	48	47	43	24
tc_ias_3	215	215	209	153
tc_ias_4	438	438	431	351
tc_ias_5	781	781	743	539
tc_ias_6	1254	1254	1241	688

Tabel IV.4 Hasil uji jumlah simpul terpakai dalam bentuk tabel

IV.4 Hasil dan Analisis Uji Jumlah Simpul Antara

Steiner tree secara umum menghasilkan jumlah simpul antara yang lebih sedikit dibandingkan algoritme BF Dijkstra dan A*. Hal ini terjadi karena BF Dijkstra dan A* memiliki kemungkinan memberikan hasil graf, sedangkan hasil Algoritma Steiner Tree pasti sebuah *tree* (graf tanpa cycle). Hal tersebut dapat dilihat pada Gambar III.2. Untuk input **MOrig**(5,8) dan **MDest**(27,2), Dijkstra dan A* memberikan hasil sebuah graf, sedangkan Steiner Tree sebuah *tree*. Jika dikaitkan dengan konteks pembelajaran kanji, untuk belajar kanji biru dari kumpulan kanji hijau butuh 6 simpul antar, sedangkan pada bagian kanan hanya butuh 4 simpul antar. Dengan begitu dapat kita katakan, bahwa pada Gambar 5, graf sebelah kanan memiliki hasil yang lebih baik.



Gambar IV.2 Hasil uji jumlah simpul terpakai dalam bentuk grafik

BAB V

KESIMPULAN DAN SARAN

V.1 Kesimpulan

Dari tugas akhir yang diberikan, berikut adalah beberapa hal yang disimpulkan.

1. Dari tiga buah algoritma yang diperbandingkan dalam tugas akhir ini, algoritma Steiner Tree memiliki waktu tempuh yang jauh lebih cepat karena memiliki *time complexity* $O((|MOrig| + |MDest|) * |Vertices_G|^2)$, lebih baik dibanding algoritma Dijkstra dan A* yang memiliki *time complexity* $O((|MOrig| * |MDest|) * (|Vertices_G| + |Edges_G|) \log |Vertices_G|)$.
2. Algoritma Steiner Tree kurang optimal jika jumlah kanji masukan terlalu sedikit. Hal ini karena adanya langkah pembangkitan pasangan *shortest path* yang hanya bergantung pada besarnya graf, tapi tidak dengan simpul input-output. Hal ini dapat diatasi dengan teknik memoisasi pada langkah tersebut yang dapat memperbaiki *time complexity*-nya menjadi $O((|MOrig| + |MDest|) * \log |Edges_G|)$.
3. Walaupun kompleksitas waktu algoritma A* sama dengan algoritma Dijkstra, algoritma A* memiliki waktu tempuh yang lebih lama akibat komputasi fungsi *heuristik* yang digunakan.

V.2 Saran

Adapun, berikut adalah saran yang ditunjukkan untuk pengerjaan tugas akhir ini jika diberikan tambahan sumber data baik melalui waktu maupun keahlian.

1. Dilakukan konstruksi dataset *knowledge graph* kanji dengan relasi yang lebih lengkap dengan melibatkan relasi homofon, homonim, kesamaan konsep, dan kesamaan pembentuk kata.
2. Dilakukan pengembangan lebih lanjut pada algoritma A* dan Steiner Tree dengan melibatkan paralelisme. Algoritma yang dipakai saat ini hanya

dilakukan secara serial dan melewatkan beberapa optimasi yang dapat dilakukan lewat paralelisme.

3. Dilakukan analisis pada kasus uji yang lebih beragam, dengan mengambil subgraf *knowledge graph* relasi kanji yang membentuk graf lengkap, graf condong, dan graf tidak terhubung.

DAFTAR REFERENSI

- [1] Ethnologue, *What are the top 200 most spoken languages?* Online; diakses pada 4 November 2020, 2020. URL:
<https://www.ethnologue.com/guides/ethnologue200>.
- [2] E. W. Schneider, "Course modularization applied: The interface system and its implications for sequence control and data analysis.," 1973. URL:
<https://eric.ed.gov/?id=ED088424>.
- [3] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, dan K. Miller, "Introduction to wordnet: An on-line lexical database*," vol. 3, Jan. 1991. DOI:
[10.1093/ijl/3.4.235](https://doi.org/10.1093/ijl/3.4.235).
- [4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, dan Z. Ives, "Dbpedia: A nucleus for a web of open data," dalam *The Semantic Web*, K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, dan P. Cudré-Mauroux, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, halaman 722–735, ISBN: 978-3-540-76298-0.
- [5] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, dan J. Taylor, "Freebase: A collaboratively created graph database for structuring human knowledge," dalam *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, halaman 1247–1250, ISBN: 9781605581026. DOI: [10.1145/1376616.1376746](https://doi.org/10.1145/1376616.1376746). URL:
<https://doi.org/10.1145/1376616.1376746>.
- [6] R. Barták, A. Dovier, dan N.-F. Zhou, "Multiple-origin-multiple-destination path finding with minimal arc usage: Complexity and models," dalam *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2016, halaman 91–97. DOI: [10.1109/ICTAI.2016.0024](https://doi.org/10.1109/ICTAI.2016.0024).
- [7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, halaman 269–271, Dec. 1959, ISSN: 0945-3245. DOI: [10.1007/BF01386390](https://doi.org/10.1007/BF01386390). URL:
<https://doi.org/10.1007/BF01386390>.

- [8] P. E. Hart, N. J. Nilsson, dan B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, halaman 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.
- [9] G. M. L. Kou dan L. Berman, “A fast algorithm for steiner tree,” dalam *Acta Informatika 15*, 1981. DOI: 10.1007/BF00288961.