Desain Solver untuk Puzzle 'Cryptogram' Menggunakan Pencocokan String dan Regular Expression

Muhamamd Zakkiy - 10122074

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: muhammadzakkiy@gmail.com , 10122074@mahasiswa.itb.ac.id

***abstrak*— Makalah ini menyajikan desain dan implementasi *solver* otomatis untuk puzzle *cryptogram*. Penelitian ini bertujuan mengembangkan sistem komputasi untuk mendekripsi pesan terenkripsi secara efisien. Metodologi yang digunakan mengintegrasikan pencocokan *string* dan ekspresi reguler untuk identifikasi pola serta penyaringan kandidat kata dari kamus yang didukung oleh algoritma *backtracking* sebagai inti pencarian rekursif. Pengujian pada berbagai *cryptogram* menunjukkan *solver* efektif dalam menemukan solusi dengan akurasi tinggi dan waktu eksekusi yang bervariasi sesuai kompleksitas. Kombinasi teknik ini terbukti menjadi pendekatan yang tangguh dan efisien untuk pemecahan *kriptogram* otomatis**

Kata Kunci*— *cryptogram, solver, Backtracking, Regular Expression, Pencocokan String

I. PENDAHULUAN

A. Latar Belakang

Cryptogram adalah teka-teki kata yang menampilkan teks terenkripsi yang didekripsi oleh pengguna untuk mengungkap suatu pesan [1]. Awalnya *cryptogram* digunakan untuk keamanan pesan. Saat ini *cryptogram* biasanya hanya digunakan untuk tujuan hiburan di surat kabar dan majalah.

Meskipun secara konseptual sederhana, pemecahan *cryptogram* secara manual seringkali merupakan tugas yang memakan waktu dan membutuhkan keterampilan analitis yang tinggi. Pembaca harus melakukan analisis frekuensi huruf, mengidentifikasi pola kata seperti huruf ganda atau trigram yang khas, serta mencoba berbagai hipotesis penggantian huruf, yang semuanya harus tetap konsisten di seluruh puzzle. Proses *trial and error* yang berulang ini, terutama pada *cryptogram* yang lebih panjang atau memiliki ambiguitas, dapat menjadi sangat kompleks dan melelahkan.

Dalam era komputasi modern, masalah-masalah yang sebelumnya hanya dapat diselesaikan secara manual kini dapat diotomatisasi dengan kekuatan algoritma. Desain *solver* otomatis untuk *cryptogram* menawarkan aplikasi praktis yang menarik dari prinsip-prinsip ilmu komputer, khususnya dalam algoritma pencarian. Dengan memanfaatkan teknik seperti pencocokan *string* untuk mengidentifikasi pola dalam kata-kata terenkripsi, regular expression untuk merepresentasikan pola-pola tersebut secara fleksibel, dan algoritma backtracking untuk secara sistematis menjelajahi ruang kemungkinan penggantian huruf, dimungkinkan untuk membangun sebuah sistem yang dapat memecahkan *cryptogram* secara efisien.

Penelitian ini bertujuan untuk mengeksplorasi dan mendesain *solver* Cryptogram otomatis yang tidak hanya berfungsi secara efektif, tetapi juga mendemonstrasikan integrasi konsep-konsep komputasi ini. Dengan demikian, laporan ini akan menjelaskan pendekatan yang diambil dalam merancang arsitektur *solver*, detail implementasi, serta evaluasi kemampuannya dalam mendekripsi berbagai contoh *cryptogram*, sekaligus menyoroti potensi dan batasan dari metode yang digunakan.

B. Rumusan Masalah

Berdasarkan latar belakang yang telah diuraikan, pemecahan puzzle *Cryptogram* secara otomatis menghadirkan serangkaian tantangan komputasi yang memerlukan pendekatan terstruktur dan efisien. Untuk itu, penelitian ini berfokus pada pertanyaan-pertanyaan kunci sebagai berikut:

1. Bagaimana arsitektur *solver* otomatis untuk puzzle *Cryptogram* dapat dirancang agar mampu memproses *input* terenkripsi, mengidentifikasi pola, dan mendekripsi pesan secara sistematis?
2. Bagaimana teknik pencocokan *string* dan regular expression dapat dimanfaatkan secara efektif untuk mengekstrak pola unik dari kata-kata terenkripsi dan memfilter kandidat kata yang relevan dari sebuah kamus?

3. Bagaimana algoritma backtracking dapat diimplementasikan dan dioptimalkan untuk menavigasi ruang solusi yang luas, mengelola konsistensi pemetaan huruf, dan secara efisien menemukan solusi yang valid untuk *cryptogram*?
4. Seberapa efektif dan efisien *solver* yang dirancang dalam memecahkan berbagai jenis *cryptogram*?

C. *Tujuan Penelitian*

Adapun tujuan-tujuan yang ingin dicapai melalui penelitian ini adalah sebagai berikut:

1. Mendesain arsitektur modular yang komprehensif untuk sistem *solver* otomatis *Cryptogram*, meliputi modul *input*, manajemen kamus, ekstraksi pola, mesin pencocokan, dan modul *output*.
2. Mengimplementasikan fungsi-fungsi inti untuk pemrosesan teks, termasuk ekstraksi pola kata terenkripsi dan konversi pola tersebut menjadi regular expression yang dapat digunakan untuk pencarian kata kandidat dalam kamus.
3. Menerapkan algoritma backtracking secara efisien untuk mengelola proses pencarian solusi, memvalidasi konsistensi pemetaan huruf, dan menangani skenario kegagalan serta percobaan hipotesis alternatif.
4. Melakukan evaluasi kinerja *solver* yang telah dibangun pada berbagai contoh *Cryptogram* untuk mengukur akurasi dalam menemukan solusi yang benar dan menganalisis waktu eksekusi yang dibutuhkan.

II. LANDASAN TEORI

A. *Puzzle Cryptogram*

Puzzle Cryptogram adalah salah satu bentuk teka-teki kriptografi yang populer di mana suatu pesan dienkripsi dengan mengganti setiap huruf aslinya dengan huruf lain dalam alfabet tersandi [1]. Bentuk enkripsi ini secara historis dikenal sebagai cipher substitusi monoalfabetik. Terminologi "monoalfabetik" mengacu pada karakteristik kunci dari cipher ini: setiap huruf dalam alfabet asli akan selalu dipetakan secara konsisten ke satu dan hanya satu huruf lain dalam alfabet tersandi di seluruh pesan. Sebagai contoh, jika huruf 'E' dalam *plaintext* dienkripsi menjadi 'X', maka setiap kemunculan 'E' di seluruh pesan asli akan selalu diwakili oleh 'X' dalam *ciphertext*. Sebaliknya, 'X' dalam *ciphertext* akan selalu didekripsi kembali menjadi 'E'.

Struktur dasar sebuah *cryptogram* adalah serangkaian kata-kata terenkripsi yang membentuk kalimat atau kutipan bermakna. Tujuan dari puzzle ini adalah untuk mengungkap pesan asli dengan menemukan kunci substitusi yang benar. Meskipun aturan substitusi ini relatif sederhana, pemecahan *cryptogram* secara manual bisa menjadi tantangan intelektual yang signifikan.

B. *Pencocokan String dan Regular Expression*

Efisiensi dalam memecahkan *cryptogram* secara otomatis sangat bergantung pada kemampuan sistem untuk secara cepat mengidentifikasi dan memvalidasi pola-pola dalam teks. Dalam konteks ini, pencocokan *string* dan *regular expression* memegang peranan krusial.

Pencocokan *String* adalah konsep dasar dalam ilmu komputer yang melibatkan proses pencarian satu atau lebih pola di dalam sebuah teks atau *string* yang lebih besar [2]. Algoritma pencocokan *string* dapat digunakan untuk menemukan semua kemunculan pola tertentu atau mengonfirmasi apakah suatu pola ada dalam teks. Kemampuan ini esensial dalam membandingkan pola kata terenkripsi dengan pola kata-kata yang ada dalam kamus.

Sementara pencocokan *string* dapat bersifat sederhana, kompleksitas pola yang ditemukan dalam *cryptogram* membutuhkan alat yang lebih canggih, yaitu *regular expression* (Regex). Regex adalah urutan karakter yang membentuk pola pencarian [3]. Mereka menyediakan cara yang kuat dan fleksibel untuk mendeskripsikan dan mencocokkan serangkaian *string* yang memenuhi kriteria tertentu. Kekuatan utama regex dalam konteks *cryptogram* terletak pada kemampuannya untuk merepresentasikan pola struktural kata, bukan hanya urutan karakter literal. Beberapa komponen regex yang fundamental untuk penyelesaian *cryptogram* ini meliputi:

- Titik (.): Ini adalah karakter *wildcard* yang cocok dengan satu karakter tunggal apa pun (kecuali karakter *newline*). Dalam konteks *cryptogram*, “.” memungkinkan kita untuk mengatakan "karakter ini bisa berupa huruf apa saja."
- Kurung Kurawal (() - *Grouping and Capturing*): Kurung kurawal memiliki fungsi ganda. Pertama, mereka mengelompokkan bagian dari pola regex, memungkinkan operator lain untuk diterapkan pada seluruh kelompok. Kedua, dan yang terpenting untuk *cryptogram*, mereka berfungsi sebagai "grup tangkapan" (*capturing group*). Setiap sub-pola yang diapit oleh “()” akan menangkap *string* yang cocok dengannya yang dapat diakses atau dirujuk kembali. Grup tangkapan dinomori secara berurutan dari kiri ke kanan, dimulai dari 1.
- Referensi Balik (\N - *Backreference*): Ini adalah fitur kunci untuk menangani pola huruf berulang dalam *cryptogram*. \N (di mana N adalah angka) akan cocok dengan *teks yang persis sama* yang sebelumnya ditangkap oleh grup tangkapan N. Misalnya, \1 merujuk ke konten Grup 1, \2 ke Grup 2, dan seterusnya.

Kombinasi komponen-komponen regex ini memungkinkan *solver* untuk mengonversi pola kata abstrak dari *cryptogram* menjadi ekspresi yang dapat digunakan untuk menyaring kamus kata secara efisien.

Dengan memanfaatkan kekuatan regex, *solver* dapat dengan cepat mengidentifikasi semua kata dalam kamus yang memiliki pola struktural yang identik dengan kata

terenkripsi. Ini secara drastis mengurangi ruang pencarian kandidat potensial untuk setiap kata.

C. Algoritma *Backtracking*

Algoritma *Backtracking* adalah algoritma pencarian rekursif yang secara bertahap membangun solusi. Pada setiap langkah, algoritma mencoba memperluas solusi parsial yang sedang dibangun dengan membuat sebuah "pilihan" atau "tebakan". Setelah membuat pilihan, algoritma memeriksa apakah pilihan tersebut valid dan konsisten dengan batasan-batasan masalah yang ada [4].

Berdasaran referensi [5], prinsip kerja *backtracking* dapat dijelaskan sebagai berikut:

1. Pembangunan Solusi Inkremental: Solusi dibangun satu per satu dengan menambahkan komponen baru.
2. Pilihan dan Konsistensi: Pada setiap titik di mana ada beberapa pilihan yang mungkin, algoritma memilih salah satu. Pilihan ini kemudian divalidasi. Jika pilihan tersebut konsisten dengan batasan masalah yang sudah ada, algoritma melanjutkan ke langkah berikutnya.
3. Mundur (*Backtrack*) saat Gagal: Jika pilihan yang diambil ternyata mengarah pada situasi yang tidak konsisten atau buntu, algoritma membatalkan pilihan terakhir yang dibuat dan kembali ke kondisi sebelumnya.
4. Eksplorasi Alternatif: Setelah mundur, algoritma mencoba pilihan lain yang belum dieksplorasi dari titik tersebut. Proses ini berlanjut hingga solusi ditemukan atau semua kemungkinan telah dicoba dan terbukti tidak mengarah pada solusi.

D. *Basis Data Kata (Kamus)*

Keberhasilan dan efisiensi sebuah *Cryptogram Solver* sangat bergantung pada kualitas dan organisasi basis data kata atau kamus yang digunakannya. Kamus berfungsi sebagai referensi utama bagi *solver* untuk memvalidasi dan mengidentifikasi kata-kata potensial dalam *plaintext* yang cocok dengan pola kata terenkripsi. Tanpa kamus yang memadai, *solver* tidak akan memiliki kumpulan kata yang cukup untuk melakukan pencocokan dan deduksi.

Beberapa karakteristik penting dari kamus yang efektif untuk *solver cryptogram* meliputi:

1. Kamus harus cukup besar dan mencakup sebagian besar kata-kata umum dalam bahasa target. Kamus yang terlalu kecil dapat menyebabkan *solver* gagal menemukan solusi meskipun *cryptogram* tersebut memiliki jawaban yang valid.
2. Kamus hanya berisi kata-kata yang relevan dengan jenis *cryptogram* yang ingin dipecahkan. Hal ini untuk menghindari kata-kata yang sangat jarang yang tidak akan muncul dalam puzzle umum. Kamus yang terlalu besar dan tidak relevan dapat memperlambat proses pencarian secara signifikan.
3. Kata-kata dalam kamus harus bebas dari tanda baca, angka, atau karakter non-alfabet lainnya. Konsistensi dalam format sangat penting untuk pencocokan yang akurat dengan *regular expression*.

Dalam implementasi *solver* ini, kamus diproses dan diorganisir sedemikian rupa sehingga kata-kata dikelompokkan berdasarkan panjangnya. Struktur data seperti *dictionary* digunakan di mana kunci-kuncinya adalah panjang kata dan nilainya adalah daftar semua kata dengan panjang tersebut. Pendekatan ini secara drastis mengurangi ruang pencarian karena ketika *solver* mencoba mencocokkan kata terenkripsi dengan panjang tertentu, *solver* hanya perlu melihat subset kata-kata dari kamus yang memiliki panjang yang sama.

III. METODOLOGI PENELITIAN

A. *Arsitektur Solver*

Desain arsitektur *Cryptogram Solver* ini mengadopsi pendekatan modular, di mana setiap fungsionalitas inti dienkapsulasi dalam komponen yang terpisah, namun saling berinteraksi secara sistematis. Struktur ini memastikan kode mudah dipahami, dikelola, dan dikembangkan di masa mendatang. Secara keseluruhan, *solver* ini beroperasi di bawah koordinasi kelas CryptogramSolver yang mengelola alur kerja utama dari penerimaan *input* hingga penyajian *output* solusi.

Berikut adalah komponen-komponen utama arsitektur *solver* dan interaksinya:

1. Modul Manajemen Kamus (_load_and_process_dictionary)

   Modul ini bertanggung jawab untuk memuat basis data kata bahasa Inggris dari file eksternal. Modul ini melakukan *pre-processing* dasar seperti mengonversi kata menjadi huruf kapital dan memfilter kata-kata non-alfabetik. Kata-kata dari kamus kemudian disimpan dalam sebuah *dictionary* di mana kunci adalah panjang kata dan nilai adalah daftar semua kata dengan panjang tersebut. Hal ini memungkinkan pencarian kata kandidat berdasarkan panjangnya secara sangat efisien.

2. Modul Pembersihan Input

   Modul ini menerima *cryptogram* dalam format mentah. Modul ini membersihkan *string* dengan mengonversi semua huruf menjadi kapital dan

menghapus karakter non-alfabetik atau spasi yang tidak relevan dengan proses substitusi huruf. Kemudian, teks dipecah menjadi daftar kata-kata terenkripsi yang unik untuk diproses. Modul ini merupakan bagian awal dari metode solve() yang menerima *cryptogram* dari pengguna.

3. Manajemen Pemetaan Kunci (cipher_map, reverse_cipher_map)

   Atribut self.cipher_map menyimpan pemetaan huruf terenkripsi ke huruf asli, sedangkan self.reverse_cipher_map menyimpan pemetaan sebaliknya. Kedua *map* ini penting untuk memastikan konsistensi satu-ke-satu dari *cipher* substitusi. Kedua *map* ini diinisialisasi kosong pada setiap pemanggilan solve().

4. Modul Pembentuk Regex Dinamis (_pattern_to_regex)

   Modul ini mengambil sebuah kata terenkripsi dan current_cipher_map sebagai *input*. Modul ini tidak hanya mengidentifikasi pola struktural dasar kata, tetapi juga secara dinamis menyuntikkan huruf asli yang sudah diketahui ke dalam *regular expression* yang dihasilkan. Misalnya, jika 'K' sudah diketahui sebagai 'H', dan kata terenkripsi adalah KVSSF, regex akan dimulai dengan ^H daripada ^(.). Ini secara drastis mempersempit ruang pencarian di kamus.

5. Mesin Pencocokan dan Deduksi Inti (_solve_recursive)

   Modul ini adalah inti dari *solver*, mengimplementasikan algoritma backtracking. Modul ini mengelola alur pencarian solusi secara rekursif. Modul ini melakukan proses berikut.

   - Pemilihan Kata: Memilih kata terenkripsi berikutnya untuk diproses berdasarkan heuristik.
   - Pencarian Kandidat: Menggunakan *regex* yang dihasilkan oleh _pattern_to_regex untuk menyaring kata-kata yang cocok dari kamus.
   - Pemeriksaan Konsistensi (_is_consistent): Memvalidasi apakah kandidat kata dari kamus akan konsisten dengan semua pemetaan huruf yang sudah ada.
   - Rekursi & Backtracking: Untuk setiap kandidat yang valid, modul ini memperbarui *map* secara sementara dengan membuat salinan baru dari *map* yang ada dan memanggil dirinya sendiri secara rekursif dengan sisa kata-kata terenkripsi. Jika panggilan rekursif gagal menemukan solusi, ia secara otomatis "mundur" dan mencoba kandidat berikutnya.

6. Modul Pemeriksa Konsistensi (_is_consistent)

   Modul ini merupakan fungsi utilitas yang memeriksa validitas sebuah pemetaan hipotesis antara kata terenkripsi dan kata kandidat dari kamus relatif terhadap cipher_map dan reverse_cipher_map yang sudah ada. Ini mencegah konflik seperti satu huruf terenkripsi memetakan ke dua huruf asli atau dua huruf terenkripsi memetakan ke satu huruf asli.

7. Modul Tampilan Output (_display_solution)

   Modul ini enyajikan hasil dekripsi kepada pengguna dalam format yang mudah dibaca. Ini termasuk menampilkan *cryptogram* asli dan daftar lengkap pemetaan huruf yang ditemukan.

B. Algoritma Pemecahan

Proses pemecahan *cryptogram* diimplementasikan menggunakan algoritma *backtracking* yang dikelola oleh fungsi rekursif _solve_recursive. Alur kerja *solver* secara keseluruhan dapat dibagi menjadi beberapa tahapan utama:

1. Fase Inisialisasi dan Pra-pemrosesan (solve() method):

   - Penerimaan Input: Metode solve(cryptogram_text) menerima *string cryptogram* mentah dari pengguna.
   - Pembersihan Teks: *String cryptogram* dibersihkan dari tanda baca, angka, dan dikonversi menjadi huruf kapital penuh (re.sub(r'[^\w]+', '', cryptogram_text.upper())).
   - Ekstraksi Kata Unik: Teks yang sudah bersih dipecah menjadi daftar kata-kata

terenkripsi yang unik. Kata-kata ini akan menjadi unit dasar yang akan diproses oleh *solver*.

- Pengurutan Heuristik: Daftar kata-kata terenkripsi unik ini diurutkan. Dalam implementasi ini, heuristik yang digunakan adalah mengurutkan dari kata terpanjang ke terpendek. Kata-kata yang lebih panjang cenderung memberikan lebih banyak batasan huruf, yang dapat membantu mempersempit ruang pencarian lebih awal.

- Inisialisasi Map: *Cipher map* dan *reverse cipher map* diinisialisasi sebagai *dictionary* kosong untuk menyimpan pemetaan huruf.

- Panggilan Rekursif Awal: Proses pemecahan dimulai dengan memanggil _solve_recursive() pertama kali, meneruskan daftar kata-kata terenkripsi yang sudah diurutkan dan *map* yang masih kosong.

2. Fungsi Rekursif Utama (_solve_recursive):

   - Kondisi Henti (Basis Kasus):

     - Jika self.solution_found bernilai True (solusi sudah ditemukan oleh cabang rekursif lain), fungsi segera berhenti dan mengembalikan True.

     - Jika daftar encrypted_words yang diteruskan ke pemanggilan rekursif saat ini kosong (artinya, semua kata telah berhasil dipetakan secara konsisten), maka solusi telah ditemukan. self.solution_found disetel ke True, dan *map* final disimpan ke atribut kelas self.cipher_map dan self.reverse_cipher_map, lalu fungsi mengembalikan True.

   - Pemilihan Kata Berikutnya: Kata terenkripsi pertama dari daftar encrypted_words (current_encrypted_word = encrypted_words[0]) dipilih untuk dipecahkan pada level rekursi saat ini.

   - Pembentukan *Regular Expression* Dinamis:

     - Metode _pattern_to_regex(current_encrypted_word, current_cipher_map) dipanggil. Fungsi ini tidak hanya mengidentifikasi pola struktural kata, tetapi juga menggabungkan semua pemetaan huruf yang sudah diketahui dari current_cipher_map ke dalam *regex* tersebut.

     - Hasilnya adalah *regex* yang lebih spesifik, seperti ^H(.)(.)\3(.)$ jika 'H' sudah diketahui memetakan ke huruf pertama. Ini secara signifikan mengurangi jumlah kandidat kata yang harus diuji dari kamus.

3. Pencarian dan Penyaringan Kandidat Kata (_solve_recursive berlanjut):

   - Penyaringan Panjang: *Solver* mengambil daftar kata dari self.dictionary yang memiliki panjang yang sama dengan current_encrypted_word.

   - Pencocokan Regex: Setiap kata dari daftar kamus disaring menggunakan *regex* yang telah dikompilasi.

   - Pemeriksaan Konsistensi Awal (_is_consistent): Hanya kandidat kata yang cocok dengan pola *regex* dan yang konsisten dengan pemetaan huruf yang sudah ada yang ditambahkan ke possible_matches. Pemeriksaan ini memastikan bahwa tidak ada konflik yang sudah terlihat pada *map* yang ada.

4. Eksplorasi Hipotesis dan Rekursi (_solve_recursive loop):

   - Untuk setiap candidate_word yang ada di possible_matches:

     - Pembuatan Map Baru: Salinan baru dari current_cipher_map dan current_reverse_cipher_map dibuat. Ini adalah kunci dari pendekatan *backtracking* "passing copies". Setiap cabang rekursif akan bekerja pada salinan *map*-nya sendiri, tanpa memengaruhi *state* level rekursi sebelumnya.

- Penerapan Pemetaan Hipotesis: Huruf-huruf dari current_encrypted_word dipetakan ke huruf-huruf candidate_word dan ditambahkan ke new_cipher_map dan new_reverse_cipher_map.

- Pemeriksaan Konsistensi Lanjutan (Inline Check): Sebuah pemeriksaan tambahan dilakukan selama penerapan pemetaan. Jika karakter terenkripsi yang *baru* akan dipetakan ternyata mengarah ke karakter *plaintext* yang sudah dipetakan oleh karakter terenkripsi yang *berbeda*, maka ini adalah bad_translation. Jika nilai bad_translation bernilai True, maka kandidat ini dilewati.

- Panggilan Rekursif: _solve_recursive() dipanggil kembali dengan sisa kata-kata terenkripsi, new_cipher_map, dan new_reverse_cipher_map yang baru.

- Kondisi Berhasil Cabang: Jika panggilan rekursif mengembalikan True (solusi ditemukan di cabang ini), maka fungsi segera mengembalikan True ke level atasnya.

5. Backtracking:

   - Jika _solve_recursive yang dipanggil secara rekursif mengembalikan False, maka loop for candidate_word akan berlanjut ke candidate_word berikutnya. Nilai False menandakan bahwa tidak ada solusi yang dapat ditemukan dari jalur itu dengan candidate_word yang dipilih

   - Karena setiap panggilan rekursif bekerja pada salinan *map*-nya sendiri, "backtracking" dilakukan secara otomatis: ketika panggilan rekursif kembali ke pemanggilnya, *map* dari pemanggil tidak terpengaruh oleh perubahan yang tidak berhasil di level bawah, siap untuk mencoba kandidat berikutnya dengan *map* aslinya.

6. Gagal Menemukan Solusi:

   - Jika loop for candidate_word selesai, dan tidak ada satu pun kandidat yang menghasilkan solusi valid, maka fungsi _solve_recursive akan mengembalikan False. Ini menandakan bahwa tidak ada solusi yang dapat ditemukan dari current_encrypted_word dan current_cipher_map yang diberikan.

7. Fase Pelaporan:

   - Setelah _solve_recursive selesai, waktu eksekusi total dihitung dan ditampilkan.

   - Jika solusi ditemukan, metode _display_solution() dipanggil untuk menampilkan *cryptogram* asli, pesan yang didekripsi, dan pemetaan huruf yang ditemukan.

Alur algoritma ini secara sistematis menjelajahi ruang solusi, memanfaatkan kekuatan *regular expression* untuk pencarian pola yang efisien dan prinsip *backtracking* untuk mengelola kompleksitas dan memastikan konsistensi pemetaan huruf di seluruh *puzzle*.

C. Sumber Data dan Lingkungan Implementasi

Bagian ini menguraikan sumber data yang digunakan sebagai basis pengetahuan *solver* dan lingkungan teknis tempat *solver* dikembangkan dan diuji.

1) Sumber Data (Kamus Kata)

Sumber data utama bagi *Cryptogram Solver* ini adalah sebuah basis data kata bahasa Inggris. Kamus ini esensial untuk memvalidasi setiap hipotesis pemetaan huruf dan mencari kata-kata kandidat yang sesuai dengan pola terenkripsi. Kamus yang digunakan berasal dari proyek *open-source* https://norvig.com/ngrams/ [6]. Kamus yang digunakan adalah count_1w100k.txt. Dataset ini dipilih karena cakupannya yang luas, menyediakan lebih dari 100.000 kata-kata bahasa Inggris umum.

2) Lingkungan Implementasi

Cryptogram Solver diimplementasikan sepenuhnya menggunakan Python 3.x. Python dipilih karena sintaksisnya yang intuitif, dukungan komunitas yang kuat, dan ketersediaan pustaka yang kaya untuk pemrosesan

string dan ekspresi reguler. Pustaka yang digunakan adalah sebagai berikut.

- **re**

  re merupakan modul bawaan Python untuk bekerja dengan *regular expression*. Ini krusial untuk pembuatan pola dinamis dan pencocokan kata di kamus.

- **os**

  os merupakan modul bawaan Python yang menyediakan cara untuk berinteraksi dengan sistem operasi, digunakan untuk mengelola jalur file kamus secara *platform-agnostic*.

- **time**:

  time merupakan modul bawaan Python untuk mengukur kinerja dan waktu eksekusi program, yang diintegrasikan untuk mengevaluasi efisiensi *solver*.

Lingkungan implementasi ini menyediakan fondasi yang stabil dan alat yang diperlukan untuk mengembangkan dan mengevaluasi algoritma pemecahan *cryptogram* secara efektif.

IV. Implementasi dan Hasil Percobaan

A. Detail Implementasi

Implementasi *Cryptogram Solver* ini dibangun di atas bahasa pemrograman Python, memanfaatkan struktur berorientasi objek untuk modularitas dan efisiensi. Komponen-komponen utama yang dijelaskan dalam arsitektur solver diterjemahkan menjadi metode-metode dalam kelas CryptogramSolver, yang saling berinteraksi untuk mencapai tujuan dekripsi.

1) Struktur Kelas dan Modul

Seluruh fungsionalitas *solver* diringkas dalam kelas tunggal CryptogramSolver. Kelas ini mengelola atribut-atribut *state* seperti self.cipher_map, self.reverse_cipher_map, dan self.dictionary, serta menyediakan metode-metode untuk setiap langkah dalam proses pemecahan. Modul standar Python seperti re, os, dan time diimpor dan digunakan di seluruh implementasi.

2) Implementasi Modul Kunci

- Manajemen Kamus (_load_and_process_dictionary)

  Kamus dimuat dari file teks ke dalam memori. Setiap kata dibersihkan dengan dikonversi ke huruf kapital dan divalidasi sebagai alfabetis murni sebelum disimpan. Kata-kata diorganisir dalam sebuah *dictionary* Python, di mana kunci adalah panjang kata dan nilai adalah sebuah *list* berisi semua kata kamus dengan panjang tersebut. Hal ini memungkinkan pencarian O(1) untuk subset kata-kata berdasarkan panjangnya, yang sangat efisien. *Path* file kamus ditangani secara dinamis menggunakan os.path.join dan os.path.dirname(__file__) untuk memastikan kompatibilitas lintas *platform* dan kemudahan pengaturan dalam struktur repositori GitHub.

- Pembentuk *Regular Expression* Dinamis (_pattern_to_regex)

  Metode ini menerima kata terenkripsi yang sedang diproses dan pemetaan huruf yang sudah diketahui sejauh ini (current_cipher_map). Fungsi ini membangun *regex* dengan menyuntikkan huruf asli yang sudah diketahui dari current_cipher_map. Misalnya, jika 'K' terenkripsi sudah dipetakan ke 'H' asli, dan KVSSF sedang diproses, *regex* akan dimulai dengan ^H diikuti oleh pola untuk huruf yang belum diketahui. Untuk huruf terenkripsi yang belum diketahui, ia menggunakan kombinasi (.) untuk menangkap karakter dan \ (backreference) seperti \1, \2, dst., untuk memastikan konsistensi pola di dalam kata tersebut. Penggunaan re.escape() memastikan karakter asli yang disuntikkan diperlakukan sebagai literal dan tidak mengganggu sintaksis *regex*. Kompilasi *regex* dilakukan sekali di _solve_recursive untuk meningkatkan kinerja saat mencocokkan terhadap banyak kandidat kata.

- Pemeriksa Konsistensi (_is_consistent)

  Fungsi ini adalah penjaga gerbang penting untuk menjaga konsistensi pemetaan. Ia menerima sebuah pasangan (encrypted_word, candidate_word) beserta current_cipher_map dan current_reverse_cipher_map. Metode ini melakukan dua jenis pemeriksaan kunci: pertama, memastikan tidak ada huruf terenkripsi tunggal yang memetakan ke dua huruf asli yang berbeda; dan kedua, memastikan tidak ada huruf asli tunggal yang dipetakan oleh dua huruf terenkripsi yang berbeda. Jika salah satu konflik terdeteksi, fungsi segera mengembalikan False, menginstruksikan algoritma *backtracking* untuk tidak melanjutkan jalur tersebut.

- Algoritma *Backtracking* (_solve_recursive)

Metode ini adalah metode rekursif inti dari *solver*. Pendekatan yang digunakan adalah "melewatkan salinan *map*", di mana setiap panggilan rekursif menerima current_cipher_map dan current_reverse_cipher_map sebagai argumennya. Ketika sebuah hipotesis (pemetaan current_encrypted_word ke candidate_word) dibuat, salinan *map* yang baru (new_cipher_map, new_reverse_cipher_map) dibuat dan diperbarui dengan pemetaan hipotesis tersebut. Panggilan rekursif berikutnya kemudian menggunakan new_cipher_map ini. Jika panggilan rekursif tersebut mengembalikan False, *map* dari level rekursi sebelumnya tetap tidak terpengaruh, secara otomatis memfasilitasi *backtracking* tanpa perlu operasi "hapus pemetaan" eksplisit pada *map* global. Heuristik pengurutan kata terpanjang terlebih dahulu yang diterapkan di solve() membantu _solve_recursive untuk memproses kata-kata yang paling informatif di awal, berpotensi mengurangi kedalaman pencarian dan jumlah *backtracking* yang dibutuhkan.

- Fungsi Utama solve() dan Pengukuran Kinerja

  Metode solve() berfungsi sebagai titik masuk utama bagi pengguna. Ia mengelola pra-pemrosesan *input cryptogram*, inisialisasi *solver*, memulai proses rekursif dengan memanggil _solve_recursive, dan pada akhirnya menampilkan hasil akhir. Pengukuran waktu eksekusi diimplementasikan dengan merekam time.time() di awal dan akhir metode solve(), kemudian menghitung selisihnya untuk menampilkan durasi total pemecahan. Ini memungkinkan evaluasi efisiensi *solver* pada berbagai *cryptogram* dan membantu dalam identifikasi area untuk optimasi lebih lanjut.

B. Skenario Pengujian

Pengujian *Cryptogram Solver* ini dirancang untuk memvalidasi akurasi, efisiensi, dan robustas algoritma dalam memecahkan berbagai tingkat kesulitan puzzle. Proses pengujian dilakukan dengan menjalankan skrip *solver* Python (src/solver.py) pada sejumlah *cryptogram* yang dipilih secara spesifik, yang mewakili variasi dalam panjang dan kompleksitas.

1) Metrik Pengujian

Selama pengujian, dua metrik utama diamati dan dicatat untuk setiap *cryptogram*:

1. Akurasi Solusi: Menunjukkan apakah *solver* berhasil menemukan *plaintext* yang benar dan konsisten untuk *cryptogram* yang diberikan.
2. Waktu Eksekusi: Durasi waktu yang dibutuhkan program (dalam detik) untuk menyelesaikan proses dekripsi atau menyatakan tidak ada solusi. Metrik ini diukur menggunakan modul time bawaan Python.

2) Tipe Skenario Pengujian

Pengujian dilakukan pada berbagai kategori *cryptogram* sebagai berikut:

1. Cryptogram Sederhana:

   Pengujian *cryptogram* sederhana ini untuk memverifikasi fungsionalitas dasar *solver* dan memastikan pemetaan huruf yang paling umum dapat ditemukan dengan cepat. Kata-kata yang digunakan relatif pendek dan sering muncul dalam bahasa Inggris. Contohnya adalah SWJRKM RPKBFG GNSWOC JMG JXXWCNCW KDWE. *Cryptogram* ini mempunyai solusi REASON SHOULD DIRECT AND APPETITE OBEY.

2. Cryptogram Kompleks:

   Pengujian *cryptogram* kompleks ini bertujuan untuk mengevaluasi robustas dan efisiensi *solver* pada *cryptogram* berukuran substansial yang menyerupai kalimat penuh. Ini menguji seberapa baik algoritma *backtracking* menangani ruang pencarian yang lebih besar dan jumlah kemungkinan konflik yang lebih tinggi. Contohnya adalah FJP IWWA NMK QL FJP NMK YJW KW NMFFPG JWY NWGMBBE HKYWGFJE JP JML TPPK, QL NWXQKI FW TPZWNP TPFFPG. *Cryptogram* ini mempunyai solusi THE GOOD MAN IS THE MAN WHO NO MATTER HOW MORALLY UNWORTHY HE HAS BEEN, IS MOVING TO BECOME BETTER.

3. Cryptogram Tanpa Solusi:

   Pengujian *cryptogram* tanpa solusi ini bertujuan untuk memverifikasi bahwa *solver* dapat dengan benar mengidentifikasi ketika suatu *cryptogram* tidak memiliki solusi yang valid dengan kamus yang tersedia atau dalam batas-batas pencarian yang wajar, daripada terjebak dalam *loop* tak terbatas. Contohnya adalah Ciphertext: QHG PGF HGLY CGBG UHG HGS YGGX HG YRBBGB HG R YGLXHG

Selama pengujian, *output* konsol dari *solver* yang menampilkan teks terenkripsi, kata-kata unik yang dipecahkan, hasil dekripsi, pemetaan huruf, dan waktu eksekusi dicatat dan dianalisis untuk setiap skenario.

Ini memberikan data empiris untuk evaluasi kinerja yang komprehensif.

C. *Analisis hasil*

Bagian ini menyajikan dan menganalisis hasil yang diperoleh dari serangkaian pengujian yang dilakukan terhadap *Cryptogram Solver*. Analisis berfokus pada evaluasi efektivitas *solver* dalam menemukan solusi yang benar (akurasi) dan efisiensinya dalam hal waktu eksekusi pada berbagai tingkat kompleksitas *cryptogram*.

1) *Ringkasan Hasil Pengujian*

Tabel 1 meringkas hasil pengujian yang dilakukan pada empat skenario *cryptogram* yang berbeda:

Tabel 1: Ringkasan Hasil Pengujian Cryptogram Solver

No	Cryptogram	Jumlah Kata Unik	Akurasi	Waktu Eksekusi (detik)
1	SWJRKM RPKBFG GNSWOC JMG JXXWCNCW KDWE	6	100%	1.2301
2	FJP IWWA NMK QL FJP NMK YJW KW NMFFPG JWY NWGMBBE HKYWGFJE JP JML TPPK, QL NWXQKI FW TPZWNP TPFFPG.	17	100%	276.4959
3	QHG PGF HGLY CGBG UHG HGS YGGX HG YRBBGB HG R YGLXHG	11	Solusi tidak ditemukan	19.1585

(Catatan: Waktu eksekusi dapat bervariasi tergantung spesifikasi perangkat keras dan ukuran kamus.) |

2) *Diskusi dan Interpretasi Hasil*

Untuk *cryptogram* sederhana, *solver* menunjukkan kinerja yang sangat baik. Solusi ditemukan dengan cepat dan akurat. Ini menunjukkan bahwa algoritma dasar pencocokan *string* dan *regex* bekerja secara efektif untuk kasus-kasus dengan ruang pencarian yang kecil dan pola yang jelas. Kemampuan *solver* untuk langsung mengidentifikasi kata-kata umum dengan pola yang cocok berkontribusi pada efisiensi ini.

Cryptogram yang lebih kompleks menunjukkan peningkatan waktu eksekusi yang signifikan. Namun, *solver* masih berhasil menemukan solusi yang benar. Peningkatan waktu ini adalah cerminan dari ruang pencarian yang jauh lebih besar dan jumlah langkah *backtracking* yang lebih banyak yang diperlukan untuk menjelajahi semua kombinasi yang mungkin. Heuristik pengurutan kata terpanjang di awal terbukti bermanfaat di sini, karena kata-kata panjang dapat segera memberikan banyak pemetaan huruf yang membatasi pilihan untuk kata-kata berikutnya. Penggunaan *regex* dinamis yang menyuntikkan pemetaan huruf yang sudah diketahui juga sangat krusial dalam menekan jumlah kandidat kamus pada setiap langkah rekursi, sehingga mencegah ledakan kombinatorial yang tidak terkendali.

Pada *cryptogram* tidak memiliki solusi yang valid dengan kamus yang tersedia, *solver* berhasil mengidentifikasi bahwa solusi tidak dapat ditemukan dalam jangka waktu yang wajar. Hal ini menunjukkan bahwa algoritma *backtracking* mampu menjelajahi semua kemungkinan yang layak dan mengembalikan False ketika tidak ada jalur solusi yang ditemukan.

Secara keseluruhan, *Cryptogram Solver* yang dirancang menunjukkan kemampuan yang menjanjikan dalam memecahkan puzzle dengan berbagai tingkat kesulitan. Kinerjanya optimal untuk *cryptogram* pendek hingga sedang, dan mampu menangani kasus yang lebih kompleks, meskipun dengan waktu eksekusi yang meningkat secara proporsional dengan kompleksitas dan ambiguitas *puzzle*.

V. KESIMPULAN DAN SARAN

A. *Kesimpulan*

1) Arsitektur solver otomatis telah berhasil dirancang dengan pendekatan modular yang sistematis. Arsitektur ini mengintegrasikan modul untuk pembersihan input, manajemen kamus, ekstraksi pola, pembentukan regular expression dinamis, mesin pencocokan inti berbasis backtracking, serta modul untuk pemeriksaan konsistensi dan penyajian output. Setiap komponen dirancang untuk bekerja sama secara efisien dalam mengalirkan dan memproses data terenkripsi hingga pesan asli ditemukan.

2) Teknik pencocokan string dan regular expression telah dimanfaatkan secara efektif. Melalui metode _pattern_to_regex, pola struktural unik dari kata-kata terenkripsi berhasil diubah menjadi regex dinamis yang kuat. Regex ini secara cerdas menyertakan pemetaan huruf yang sudah diketahui, sehingga memungkinkannya untuk memfilter secara presisi kandidat kata yang relevan dari kamus dengan sangat efisien, secara signifikan mengurangi ruang pencarian.

3) Algoritma backtracking telah diimplementasikan dan dioptimalkan secara efektif melalui fungsi rekursif _solve_recursive. Algoritma ini secara sistematis menavigasi ruang solusi yang luas dengan mencoba berbagai hipotesis pemetaan huruf. Optimalisasi dicapai melalui heuristik pengurutan kata dan penggunaan regex dinamis yang secara efektif "memangkas" cabang-cabang pencarian yang tidak valid.

4) Solver yang dirancang terbukti efektif dan efisien dalam memecahkan berbagai jenis cryptogram. Pada cryptogram sederhana dan sedang, solver menunjukkan akurasi tinggi dengan waktu eksekusi yang sangat cepat. Meskipun waktu eksekusi meningkat pada cryptogram yang lebih kompleks, solver tetap berhasil menemukan solusi yang benar dalam waktu yang dapat diterima berkat optimasi yang diterapkan. Pada kasus cryptogram tanpa solusi, solver mampu mengidentifikasi ketiadaan solusi.

B. Saran

1) Mengimplementasikan heuristik pemilihan kata yang lebih canggih, seperti memprioritaskan kata yang paling membatasi atau yang menghasilkan kandidat paling sedikit dari kamus, untuk lebih mengoptimalkan proses *backtracking*.

2) Mengembangkan *solver* agar mampu menangani *cryptogram* yang lebih kompleks.

LAMPIRAN

Link Repository Github: https://github.com/UburUburLembur/Makalah_CryptogramSolver

UCAPAN TERIMA KASIH

Penulis ingin menyampaikan rasa terima kasih yang tulus kepada semua pihak yang telah memberikan dukungan dan kontribusi sehingga makalah tugas akhir mata kuliah Strategi Algoritma ini dapat terselesaikan dengan baik.

Ucapan terima kasih yang sebesar-besarnya ditujukan kepada Bapak Dr. Ir. Rinaldi, M.T. selaku dosen pengampu mata kuliah Stategi Algoritma, atas bimbingan, arahan, dan ilmu yang telah diberikan selama perkuliahan. Wawasan dan saran-saran beliau sangat berarti dalam penyusunan makalah serta pengembangan *Cryptogram Solver* ini.

Akhirnya, terima kasih yang mendalam disampaikan kepada keluarga dan teman-teman atas doa, dukungan moral, dan pengertiannya selama periode perkuliahan dan penyelesaian tugas akhir ini.

REFERENCES

[1] Ivy Wigmore, "What is cryptogram? | Definition from TechTarget," *TechTarget*, [Online]. Dapat diakses pada: https://www.techtarget.com/whatis/definition/cryptogram. [Diakses: Jun. 21, 2025].

[2] GeeksforGeeks, " Applications of String Matching Algorithms," *GeeksforGeeks*, [Online]. Dapat diakses pada: https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/. [Diakses: Jun. 21, 2025]

[3] GeeksforGeeks, "Regular Expressions in C++," *GeeksforGeeks*, [Online]. Dapat diakses pada: https://www.geeksforgeeks.org/dsa/write-regular-expressions/. [Diakses: Jun. 21, 2025]

[4] Rinaldi Munir, "IF2211 Strategi Algoritma - Semester II Tahun 2024/2025", *Strategi Algoritma*, [Online]. Dapat diakses pada: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm. [Diakses: Jun. 21, 2025]

[5] GeeksforGeeks, "Backtraking Algorithm," *GeeksforGeeks*, [Online]. Dapat diakses pada: https://www.geeksforgeeks.org/dsa/backtracking-algorithms/. [Diakses: Jun. 21, 2025]

[6] P. Norvig, "English Word Frequency," *norvig.com*, [Online]. Available: https://norvig.com/ngrams/. [Diakses: Jun. 21, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 23 Juni 2025

Muhammad Zakkiy (10122074)