Optimasi Seleksi File Konteks AI Coding Agent Berbasis Bug Report Menggunakan Program Dinamis pada Precedence Constrained Knapsack

Fayyaz Akmal Lauda - 13524076
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: fayyazakmallauda@gmail.com , 13524076@std.stei.itb.ac.id

***Abstract*—Ketika seorang pengembang menggunakan AI coding agent untuk memperbaiki sebuah bug, agent tersebut perlu diberi tahu file-file kode yang relevan dengan masalah itu. Pada repositori besar dengan ratusan file, memilih file secara sembarangan sering menghasilkan kebingungan karena ada referensi ke kelas atau fungsi yang tidak ia kenal. Masalah tersebut dapat dimodelkan sebagai persoalan Precedence-Constrained 0/1 Knapsack, dimana setiap file dipandang sebagai berat dan nilai (tingkat relevansi terhadap bug report), sementara relasi import antarfile menjadi syarat bahwa sebuah file baru boleh dipilih jika seluruh file yang ia butuhkan juka ikut dipilih. Program dinamis digunakan untuk mencari kombinasi file dengan total relevansi tertinggi tanpa melebihi batas token yang tersedia, kemudian dilengkapi mekanisme tambahan untuk memastikan tidak ada file yang kehilangan prasyaratnya. Pendekatan ini diuji pada sebuah repositori Java berisi 110 file menggunakan enam skenario bug yang berbeda. Hasilnya menunjukkan bahwa pendekatan yang diusulkan berhasil mengurangi jumlah file yang "kehilangan prasyarat" secara signifikan dibandingkan cara pemilihan file yang umum digunakan, sambil tetap menyertakan file yang sesungguhnya perlu diperbaki pada hampir semua skenario uji.**

Kata Kunci—Program Dinamis, Knapsack, AI Coding Agent, Dependency Graph, Bug Report

I. PENDAHULUAN

Dalam beberapa tahun terakhir, semakin banyak pengembang perangkat lunak yang menggunakan AI coding agent, asisten berbasis kecerdasan buatan seperti GitHub Copilot atau Claude, untuk membantu menulis dan memperbaiki kode. Salah satu cara paling umum menggunakan alat ini adalah dengan menempelkan deskripsi sebuah bug yang ditemukan, lalu meminta agen tersebut mencari penyebabnya dan menyarankan perbaikan. Tantangan menyediakan konteks kode yang tepat pada skala repositori, bukan hanya pada satu fungsi atau satu file, dikenal secara luas sebagai Repository-Level Code Generation, dan menjadi salah satu kendala utama yang masih diteliti secara aktif karena keterbatasan kapasitas konteks model serta kompleksitas dependensi antarfile [1].

Masalahnya, agen ini tidak otomatis mengetahui seluruh isi proyek. Ia hanya bisa "melihat" kode yang sengaja disertakan oleh pengguna ke dalam percakapan, dan jumlah kode yang bisa disertakan dibatasi oleh kapasitas yang disebut context window, diukur dalam satuan token. Pada proyek kecil dengan belasan file, menyertakan semuanya bukan masalah. Namun pada proyek nyata dengan puluhan hingga ratusan file, pengembang harus memilih sendiri file mana yang paling relevan untuk disertakan, dan di sinilah kesulitan dimulai. Tinjauan terhadap berbagai pendekatan retrieval-augmented untuk pembuatan kode tingkat repositori menyimpulkan bahwa pemilihan potongan kode yang tepat untuk dimasukkan ke dalam konteks merupakan salah satu faktor paling menentukan keberhasilan model dalam memahami dependensi antarfile, jauh lebih menentukan dibandingkan sekadar memperbesar kapasitas context window itu sendiri [1].

Kesulitan utamanya bukan hanya menebak file mana yang "kelihatannya" berhubungan dengan bug. Kode pada proyek berorientasi objek, khususnya yang ditulis dalam Java, hampir selalu memiliki keterkaitan antarfile melalui pernyataan import. Sebuah kelas yang menangani transaksi bank, misalnya, hampir pasti mengimpor kelas lain yang mendefinisikan jenis-jenis resource atau struktur data yang ia pakai. Jika pengembang hanya menyertakan kelas transaksi tersebut tanpa menyertakan kelas-kelas pendukungnya, AI coding agent akan kebingungan karena ia melihat pemanggilan fungsi atau penggunaan tipe data yang tidak pernah ia ketahui definisinya, sehingga analisis yang dihasilkan menjadi tidak akurat atau bahkan menyesatkan.

Kondisi ini adalah masalah optimasi, yakni dari sekian banyak file yang tersedia, pilih sekumpulan file yang totalnya tidak melebihi kapasitas token, tetapi total relevansinya terhadap bug report semaksimal mungkin, sekaligus memastikan bahwa setiap file yang dipilih tidak "kehilangan" prasyaratnya. Persoalan memilih kombinasi barang terbaik dalam batasan kapasitas adalah ciri khas dari persoalan klasik dalam strategi algoritma yang disebut 0/1 Knapsack, dan karena masalah ini memiliki tambahan syarat berupa keterkaitan antarfile, makalah ini menyebutnya sebagai Precedence-Constrained 0/1 Knapsack.

Makalah ini menguraikan bagaimana persoalan seleksi file konteks tersebut dipetakan ke dalam bentuk Knapsack dengan syarat tambahan, bagaimana Program Dinamis digunakan untuk menyelesaikannya, serta bagaimana keterkaitan antarfile, termasuk kasus dua file yang saling membutuhkan satu sama lain, ditangani secara sistematis. Sebagai bukti bahwa pendekatan ini benar-benar bekerja, sebuah program diimplementasikan dan diuji pada repositori kode Java sungguhan dengan beberapa skenario bug yang realistis, lalu hasilnya dibandingkan dengan dua cara pemilihan file yang lebih sederhana dan umum digunakan.

II. Landasan Teori

A. Program Dinamis

Program Dinamis adalah salah satu metode penyelesaian persoalan optimasi yang bekerja dengan memecah suatu persoalan besar menjadi tahapan-tahapan keputusan yang lebih kecil, lalu menyimpan hasil dari setiap tahap tersebut agar tidak perlu dihitung ulang [2]. Metode ini hanya dapat digunakan jika persoalannya memenuhi dua sifat berikut.

Optimal substructure berarti solusi terbaik dari keseluruhan persoalan dapat dibentuk dari solusi-solusi terbaik pada bagian-bagian yang lebih kecil. **Overlapping subproblems** berarti, dalam proses mencari solusi, sub-persoalan yang sama akan muncul berulang kali, sehingga menyimpan hasil sub-persoalan tersebut jauh lebih efisien dibandingkan menghitungnya ulang setiap kali dibutuhkan [3].

Perbedaan mendasar Program Dinamis dengan pendekatan greedy terletak pada cara keduanya mengambil keputusan. Greedy hanya mempertimbangkan satu pilihan yang terlihat paling menguntungkan pada setiap langkah, tanpa pernah mempertimbangkan kemungkinan lain. Program Dinamis, sebaliknya, secara sistematis mempertimbangkan seluruh kemungkinan keputusan pada setiap tahap sebelum menentukan kombinasi mana yang benar-benar menghasilkan solusi terbaik secara keseluruhan [2].

B. Persoalan 0/1 Knapsack

Persoalan 0/1 Knapsack adalah salah satu contoh klasik persoalan optimasi yang dapat diselesaikan dengan Program Dinamis. Bayangkan seseorang memiliki sebuah tas dengan kapasitas berat tertentu, dan harus memilih barang-barang mana yang akan dimasukkan agar total nilai barang yang dibawa sebesar mungkin, tanpa melebihi kapasitas tasnya. Setiap barang hanya bisa dipilih secara utuh (dimasukkan atau tidak), tidak bisa dipotong sebagian [2].

Secara matematis, diberikan n barang, masing-masing dengan berat wi dan nilai vi, serta tas berkapasitas W, persoalan ini dirumuskan sebagai mencari kombinasi barang yang dipilih (dinyatakan dengan xi = 1 jika dipilih, 0 jika tidak) sehingga:

$$\max \Sigma_i \, v_i \, x_i \qquad (1)$$

$$\text{s.t. } \Sigma_i \, w_i \, x_i \leq W, \; x_i \in \{0,1\} \qquad (2)$$

Solusi Program Dinamis untuk persoalan ini dibangun dengan menyusun tabel dp[i][w], yaitu nilai terbaik yang bisa dicapai apabila hanya boleh memilih dari i barang pertama dengan kapasitas tas sebesar w. Tabel ini diisi mulai dari kasus paling sederhana (tidak ada barang sama sekali, nilainya pasti nol) hingga ke kasus lengkap, menggunakan aturan berikut: untuk setiap barang baru yang dipertimbangkan, bandingkan hasil jika barang itu tidak diambil dengan hasil jika barang itu diambil, lalu simpan yang lebih besar.

$$dp[i][w] = dp[i-1][w], \text{ jika } w_i > w \qquad (3)$$

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i]+v_i), \text{ jika } w_i \leq w \qquad (4)$$

Nilai pada dp[0][w] selalu nol untuk berapapun w, karena tidak ada barang yang bisa dipilih (ini disebut kasus basis). Setelah seluruh tabel terisi, jawaban akhir persoalan berada pada dp[n][W], dan barang-barang mana saja yang sebenarnya terpilih dapat ditelusuri kembali dari tabel tersebut (proses ini disebut traceback). Pendekatan ini menyelesaikan persoalan dalam waktu sebanding dengan $n \times W$, jauh lebih cepat dibandingkan mencoba seluruh 2^n kemungkinan kombinasi barang secara satu per satu, sebuah pendekatan yang menjadi sangat lambat begitu jumlah barang bertambah, karena persoalan 0/1 Knapsack pada dasarnya termasuk golongan persoalan yang sulit diselesaikan secara efisien tanpa Program Dinamis (*NP-hard Problem*) [3].

C. Syarat Keterkaitan Antarbarang (Precedence Constraint)

Pada persoalan 0/1 Knapsack yang umum, setiap barang dianggap berdiri sendiri, sehingga keputusan mengambil satu barang tidak memengaruhi barang lainnya. Namun pada masalah seleksi file konteks yang dibahas makalah ini, hal itu tidak berlaku. Sebuah file boleh dipilih hanya jika seluruh file lain yang ia butuhkan (file yang diimpor) juga ikut dipilih. Inilah yang disebut sebagai syarat keterkaitan atau *precedence constraint* [4], yang secara matematis dapat dituliskan sebagai:

$$x_i \leq x_j, \; \forall j \in P(i) \qquad (5)$$

Artinya, jika file i dipilih (xi=1), maka setiap file j yang menjadi prasyaratnya juga harus dipilih (xj=1); jika tidak, maka pemilihan dianggap melanggar syarat ini.

D. Menangani File yang Saling Membutuhkan

Salah satu kerumitan yang muncul saat menerapkan syarat keterkaitan di atas pada kode asli adalah kemungkinan dua file saling mengimpor satu sama lain, dimana file A membutuhkan file B, sementara file B juga membutuhkan file A. Situasi ini disebut *circular dependency*, dan jika dibiarkan, akan menghasilkan syarat yang saling bertentangan (xA harus ≤ xB, tetapi sekaligus xB harus ≤ xA).

Solusi yang digunakan adalah dengan mendeteksi kelompok file yang saling bergantung tersebut menggunakan algoritma Kosaraju, lalu memperlakukan seluruh kelompok itu sebagai satu kesatuan. Apabila salah satu anggotanya dipilih, seluruh anggota lain dalam kelompok itu otomatis ikut dipilih, dengan berat dan nilai yang dijumlahkan menjadi satu. Konsep

komponen terhubung kuat (*strongly connected component*) pada graf berarah yang menjadi dasar algoritma ini dijelaskan dalam bahan kuliah graf [5]. Algoritma Kosaraju sendiri bekerja melalui dua tahap pencarian mendalam (DFS), tahap pertama mencatat urutan selesainya penelusuran pada graf keterkaitan yang asli, sedangkan tahap kedua menelusuri graf yang arahnya dibalik, mengikuti urutan dari tahap pertama, untuk menemukan kelompok-kelompok file yang saling terhubung tersebut.

E. Mengukur Tingkat Relevansi File

Selain memenuhi syarat keterkaitan, file yang dipilih juga harus relevan dengan bug report yang diberikan. Untuk mengukur seberapa relevan sebuah file, makalah ini menggunakan tiga indikator sederhana yang digabungkan dengan bobot yang sama. Indikator pertama adalah seberapa sering kata-kata penting dari bug report muncul pada nama dan lokasi file, diukur dengan metode *Term Frequency Inverse Document Frequency* (TF-IDF). Metode ini bekerja dengan dua prinsip, yakni kata yang sering muncul pada suatu dokumen menunjukkan kata tersebut penting bagi dokumen itu (*term frequency*), sementara kata yang justru sering muncul di hampir semua dokumen dalam koleksi dianggap kurang membedakan dan diberi bobot lebih rendah (inverse document frequency), sehingga dokumen yang benar-benar relevan terhadap suatu query dapat dibedakan dari dokumen yang hanya kebetulan memuat kata umum [6]. Indikator kedua adalah seberapa cocok nama filenya dengan kata kunci di bug report, dan indikator ketiga adalah seberapa cocok nama folder atau kelasnya. Ketiga indikator ini dirata-rata menjadi satu skor relevansi R untuk setiap file.

III. Pemodelan Masalah

Sebelum Program Dinamis dapat diterapkan, persoalan seleksi file konteks perlu diterjemahkan ke dalam bentuk yang sesuai dengan kerangka 0/1 Knapsack yang telah dijelaskan pada Bab II. Bagian ini menjelaskan secara rinci bagaimana setiap komponen persoalan nyata dipetakan ke dalam istilah-istilah Program Dinamis.

A. Barang (Item)

Setiap file Java dalam repositori dipandang sebagai satu barang yang dapat dipilih atau tidak. Apabila ditemukan kelompok file yang saling membutuhkan (lihat Bab II.D), seluruh kelompok tersebut digabung menjadi satu barang gabungan, sehingga jumlah barang yang sebenarnya diproses oleh Program Dinamis bisa lebih kecil dari jumlah file aslinya.

B. Berat (Weight)

Berat dari setiap barang adalah jumlah token yang dibutuhkan untuk menampilkan isi file tersebut secara penuh kepada AI coding agent. Untuk barang gabungan hasil penggabungan kelompok file yang saling membutuhkan, beratnya adalah jumlah token seluruh file anggota kelompok tersebut.

C. Nilai (Value)

Nilai dari setiap barang adalah skor relevansi R yang dijelaskan pada Bab II.E, yaitu seberapa besar kaitan file tersebut dengan deskripsi bug report yang diberikan. Sama seperti berat, nilai pada barang gabungan dihitung dengan menjumlahkan nilai seluruh file anggotanya.

D. Kapasitas (Capacity)

Kapasitas tas pada persoalan ini adalah batas token yang tersedia pada context window yang ingin disimulasikan, merepresentasikan keterbatasan nyata yang dialami pengguna AI coding agent saat kode yang ingin dibahas jauh lebih besar dari jumlah token yang ia mampu sertakan dalam satu percakapan.

E. Syarat Keterkaitan (Precedence)

Syarat keterkaitan antarbarang diperoleh dari relasi import antarfile yang ditemukan melalui pembacaan kode sumber: jika file A mengimpor file B, maka barang A memiliki barang B sebagai prasyaratnya, sesuai formula pada Bab II.C.

F. Fungsi Tujuan

Tujuan akhir dari seluruh pemodelan ini adalah memilih kombinasi barang (file) yang memaksimalkan total nilai (total relevansi), tanpa melebihi kapasitas (batas token), sekaligus sebisa mungkin memenuhi seluruh syarat keterkaitan (tidak ada file yang kehilangan prasyaratnya). Dengan pemodelan ini, persoalan seleksi file konteks sepenuhnya dapat diselesaikan menggunakan kerangka Program Dinamis yang telah dijelaskan sebelumnya, dengan tambahan satu langkah pemeriksaan ulang yang akan dijelaskan pada Bab III.G.

Tabel VII merangkum keseluruhan pemetaan istilah dari Bab III.A sampai III.E ke dalam satu pandangan ringkas, untuk memudahkan pembaca melihat hubungan antara konsep Knapsack dan konsep pada masalah seleksi file konteks secara sekaligus.

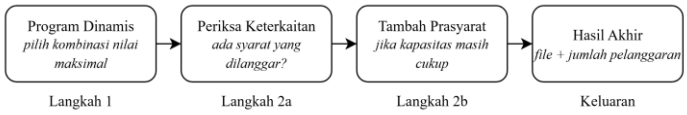
Tabel VII. Pemetaan Konsep Knapsack ke Masalah Seleksi File Konteks

Konsep pada 0/1 Knapsack	Konsep pada Seleksi File Konteks
Barang (item)	File Java (atau kelompok file yang saling membutuhkan)
Berat barang (wi)	Jumlah token untuk menampilkan isi file
Nilai barang (vi)	Skor relevansi file terhadap bug report
Kapasitas tas (W)	Batas token pada context window AI coding agent
Barang dipilih (xi=1)	File disertakan dalam konteks yang diberikan ke agen
Syarat tambahan: $x_i \leq x_j$	File i baru valid dipilih jika file j (yang ia impor) juga dipilih

G. Algoritma Penyelesaian

Program Dinamis standar, seperti dijelaskan pada Bab II.B, dapat menjamin solusi dengan total nilai maksimal dalam batas kapasitas, tetapi ia tidak dirancang untuk memeriksa syarat keterkaitan antarbarang. Oleh karena itu, makalah ini menggunakan dua langkah: Program Dinamis standar

dijalankan terlebih dahulu untuk mendapatkan kombinasi file dengan nilai terbaik, lalu hasilnya diperiksa ulang. Apabila ditemukan file terpilih yang prasyaratnya belum ikut terpilih, prasyarat tersebut ditambahkan selama kapasitas masih mencukupi. Langkah kedua ini diberi nama repair, dan keseluruhan algoritmanya dirangkum pada Algoritma 1 dan Algoritma 2 berikut.



Gambar 1. Alur dua langkah penyelesaian: Program Dinamis diikuti pemeriksaan dan perbaikan syarat keterkaitan.

Algoritma 1. Program Dinamis 0/1 Knapsack

```
FUNGSI KnapsackDP(barang, W):
  // Kasus basis: tidak ada barang, nilai nol
  dp[0..n][0..W] <- 0

  UNTUK i <- 1 SAMPAI n LAKUKAN
    UNTUK w <- 0 SAMPAI W LAKUKAN
      dp[i][w] <- dp[i-1][w]
      JIKA wi <= w MAKA
        nilai_baru <- dp[i-1][w-wi] + vi
        JIKA nilai_baru > dp[i][w] MAKA
          dp[i][w] <- nilai_baru

  // Telusuri kembali barang yang terpilih
  S <- himpunan kosong;  w <- W
  UNTUK i <- n MENURUN SAMPAI 1 LAKUKAN
    JIKA dp[i][w] != dp[i-1][w] MAKA
      S <- S gabung {barang i};  w <- w - wi
KEMBALIKAN S
```

Algoritma 2. Pemeriksaan dan Perbaikan Syarat Keterkaitan

```
FUNGSI Repair(S, W, prasyarat):
  S* <- KnapsackDP(barang, W)
  ADA_PERUBAHAN <- BENAR
  SELAMA ADA_PERUBAHAN LAKUKAN
    ADA_PERUBAHAN <- SALAH
    UNTUK SETIAP barang i di S* LAKUKAN
      UNTUK SETIAP prasyarat j dari i LAKUKAN
        JIKA j BUKAN anggota S* DAN
           (total_berat(S*) + wj <= W) MAKA
          S* <- S* gabung {j}
          ADA_PERUBAHAN <- BENAR
  pelanggaran <- hitung barang di S* yang
                 prasyaratnya belum masuk S*
KEMBALIKAN S*, pelanggaran
```

Pendekatan dua langkah ini, yaitu Program Dinamis lalu repair, dipilih karena lebih mudah dipertahankan kebenarannya dibandingkan memodifikasi langsung tabel dp agar mengenali syarat keterkaitan, yang akan membuat ukuran tabel membengkak drastis karena harus menyimpan informasi himpunan barang yang sudah terpilih, bukan sekadar nilai optimalnya saja.

Kekhawatiran berdasar pada persoalan optimasi berbasis graf lain yang juga diselesaikan dengan Program Dinamis, seperti *Travelling Salesperson Problem* (TSP), status pada tabel dp memang harus merepresentasikan himpunan bagian (subset) simpul yang telah dikunjungi, bukan sekadar satu nilai numerik seperti pada Knapsack biasa, sehingga jumlah status yang mungkin tumbuh secara eksponensial terhadap jumlah simpul [7]. Pola pertumbuhan status yang serupa berpotensi terjadi apabila syarat keterkaitan pada persoalan ini dimasukkan langsung ke dalam status tabel dp, karena status tersebut juga perlu mencatat himpunan bagian file yang sudah terpilih, bukan hanya sisa kapasitas token.

Relasi rekurens TSP yang menjadi sumber kekhawatiran ini dituliskan sebagai berikut [7]:

$$f(i, \emptyset) = c(i, 1) \qquad (6)$$

$$f(i, S) = \min\{c(i,j) + f(j, S-\{j\})\}, j \in S \qquad (7)$$

S pada relasi tersebut adalah himpunan bagian simpul yang belum dikunjungi, sehingga untuk setiap simpul i terdapat sebanyak 2 pangkat (n - 1) kemungkinan nilai S yang harus disimpan pada tabel dp, bukan hanya satu nilai tunggal seperti pada dp[i][w] Knapsack biasa. Inilah yang dimaksud dengan pertumbuhan status secara eksponensial pada paragraf sebelumnya.

IV. Implementasi dan Eksperimen

A. *Repositori yang Digunakan*

Untuk menguji apakah pemodelan pada Bab III benar-benar bekerja pada kode sungguhan, makalah ini menggunakan repositori open-source "Nimons: Banana Republic" (github.com/Labpro-22/tugas-besar-2-dj1), sebuah implementasi permainan papan bergaya Catan menggunakan JavaFX yang ditulis sebagai tugas kuliah Pemrograman Berorientasi Objek di ITB. Repositori ini dipilih karena memiliki struktur antarfile yang khas proyek Java berorientasi objek, lengkap dengan keterkaitan antarkelas yang realistis, serta jumlah file yang cukup banyak (110 file, 59.317 token) untuk diuji pada beberapa skala ukuran yang berbeda.

Untuk melihat bagaimana metode ini bekerja pada berbagai ukuran repositori, dataset dibagi menjadi tiga cakupan: cakupan kecil hanya mengambil tiga folder inti (board, resource, player) sebanyak 31 file; cakupan sedang menambahkan tiga folder lagi (card, core, audio) menjadi 57 file; dan cakupan besar mengambil seluruh repositori kecuali berkas pengujian, sebanyak 108 file.

B. *Skenario Bug yang Diuji*

Enam skenario bug dibuat dengan cara membaca isi kode repositori secara langsung dan mengidentifikasi kesalahan logika yang benar-benar ada pada baris kode tertentu, bukan kesalahan rekaan. Setiap skenario dilengkapi dengan daftar file yang sesungguhnya perlu diperbaiki (disebut ground truth), yang digunakan untuk mengukur apakah metode seleksi file berhasil menyertakan file-file penting tersebut.

Tabel I. Skenario Bug yang Digunakan dalam Eksperimen

ID	Cakupan	Deskripsi Singkat Bug	File yang Perlu Diperbaiki
BUG-S1	Kecil	Bank mengeluarkan resource melebihi stok yang tersedia	BankStorage.java
BUG-S2	Kecil	Inventory tidak memeriksa kecukupan resource sebelum dikurangi	Inventory.java, ResourceBundle.java
BUG-M1	Sedang	Pipa transportasi bisa dibangun tanpa terhubung ke jaringan pemain	PlacementManager.java, RoadBuildingCard.java
BUG-M2	Sedang	Kartu Jalan Terpanjang salah berpindah pemegang saat terjadi seri	ScoreManager.java
BUG-L1	Besar	Skor Jalan Terpanjang tidak diperbarui setelah pipa baru dibangun	GameEngine.java, ScoreManager.java
BUG-L2	Besar	Distribusi resource gagal sebagian saat stok bank menipis	BankStorage.java, ResourceManager.java

C. *Metode Pembanding*

Untuk mengetahui seberapa besar manfaat dari pendekatan yang diusulkan, hasilnya dibandingkan dengan tiga cara lain dalam memilih file. Cara pertama, disebut Greedy Relevansi, memilih file dengan skor relevansi tertinggi satu per satu sampai kapasitas penuh, tanpa memikirkan keterkaitan antarfile sama sekali. Cara ini merepresentasikan kebiasaan banyak pengembang yang asal menyertakan file yang "terlihat relevan". Cara kedua, Greedy Rasio, serupa tetapi mengutamakan file dengan rasio relevansi dibanding ukurannya, sehingga lebih menyukai file-file kecil. Cara ketiga, Knapsack Standar, adalah Program Dinamis biasa tanpa langkah repair tambahan. Cara ini menunjukkan apa yang terjadi bila keterkaitan antarfile sepenuhnya diabaikan meski menggunakan algoritma optimal. Cara terakhir adalah metode yang diusulkan makalah ini, yaitu Knapsack dengan Repair.

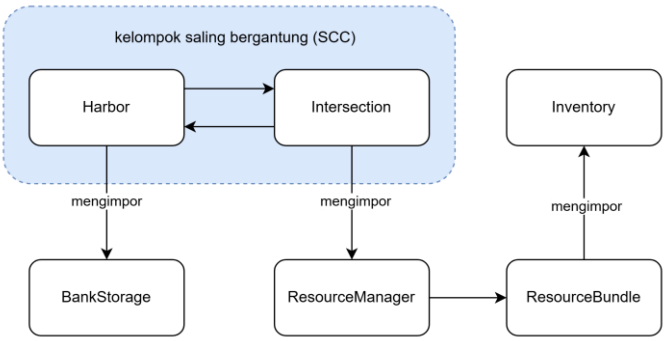
D. *Cara Mengukur Keberhasilan*

Tiga hal yang diukur dari setiap cara pemilihan file: pertama, berapa banyak file yang terpilih ternyata kehilangan prasyaratnya (disebut pelanggaran, semakin kecil semakin baik); kedua, berapa persen file ground truth yang berhasil ikut terpilih (disebut recall, semakin besar semakin baik, idealnya 100%); dan ketiga, berapa lama waktu komputasi yang dibutuhkan.

V. HASIL DAN PEMBAHASAN

A. *Kelompok File yang Saling Membutuhkan*

Sebelum melihat hasil seleksi file, perlu dicatat bahwa algoritma Kosaraju (Bab II.D) menemukan beberapa pasangan atau kelompok file yang saling mengimpor satu sama lain pada repositori ini: satu kelompok pada cakupan kecil, empat kelompok pada cakupan sedang, dan lima kelompok pada cakupan besar. Salah satu kelompok yang ditemukan, misalnya, terdiri atas empat file (PlacementManager, ExperimentCard, GameState, dan PlayerState) yang jika digabung, beratnya mencapai 1.016 token. Angka ini akan menjadi penting untuk menjelaskan satu hasil yang tidak terduga pada bagian C nanti.



Gambar 2. Potongan graf dependensi pada Subset S. Sisi terarah menunjukkan relasi import antarfile. Kelompok Harbor dan Intersection membentuk SCC yang digabung menjadi satu item.

Gambar 2 menunjukkan potongan graf dependensi pada cakupan kecil. Terlihat bahwa Harbor dan Intersection saling mengimpor satu sama lain, membentuk kelompok SCC yang ditandai dengan kotak putus-putus, sehingga keduanya digabung menjadi satu item tunggal dalam formulasi knapsack. File-file lain seperti BankStorage, ResourceManager, ResourceBundle, dan Inventory memiliki relasi import satu arah biasa tanpa membentuk siklus.

B. *Hasil pada Cakupan Kecil dan Sedang*

Tabel II menunjukkan hasil pengujian BUG-M1, kasus di mana pendekatan yang diusulkan memberikan hasil paling memuaskan: seluruh 33 pelanggaran yang dihasilkan oleh cara Greedy berhasil dikurangi menjadi nol oleh metode Knapsack

dengan Repair, sementara persentase file ground truth yang berhasil ditemukan (recall) tetap 100% pada keempat metode.

Tabel II. Hasil Pengujian BUG-M1 (cakupan sedang, kapasitas 4.000 token)

Metode	Pelanggaran	Recall	Waktu (ms)
Greedy Relevansi	33	100%	0,02
Greedy Rasio	33	100%	0,10
Knapsack Standar	16	100%	28,2
Knapsack + Repair	**0**	**100%**	**18,2**

Pada pengujian BUG-S2 (Tabel III), ditemukan hasil: Cara Greedy Rasio berhasil menemukan 100% file ground truth, sedangkan ketiga metode lain, termasuk metode yang diusulkan, hanya menemukan 50%. Hal ini terjadi karena kedua file ground truth pada kasus ini (Inventory.java dan ResourceBundle.java) berukuran kecil, sehingga rasio relevansi-per-ukurannya menjadi sangat tinggi dan diprioritaskan oleh metode Greedy Rasio, sementara metode lain yang berpatokan pada nilai relevansi mutlak justru lebih memilih file-file lain yang berukuran lebih besar.

Tabel III. Hasil Pengujian BUG-S2 (cakupan kecil, kapasitas 1.500 token)

Metode	Pelanggaran	Recall	Waktu (ms)
Greedy Relevansi	9	50%	0,01
Greedy Rasio	2	100%	0,01
Knapsack Standar	8	50%	4,54
Knapsack + Repair	5	50%	4,35

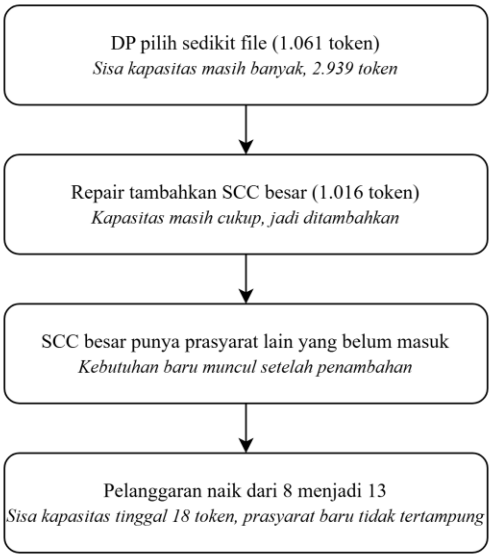
C. Satu Kasus Ketika Repair Justru Memperburuk Hasil

Hasil pada BUG-M2 (Tabel IV) memunculkan satu kondisi yang perlu dijelaskan secara hati-hati, karena pada kasus ini, metode yang diusulkan justru menghasilkan pelanggaran yang lebih banyak (13) dibandingkan Knapsack Standar (8). Hal ini sekilas terdengar aneh, mengingat tujuan repair seharusnya mengurangi pelanggaran, bukan menambahnya.

Tabel IV. Hasil Pengujian BUG-M2 (cakupan sedang, kapasitas 4.000 token)

Metode	Pelanggaran	Recall	Waktu (ms)
Greedy Relevansi	10	100%	0,01
Greedy Rasio	10	100%	0,11
Knapsack Standar	8	100%	17,0
Knapsack + Repair	13	100%	17,2

Penjelasannya berkaitan dengan kelompok empat file yang saling membutuhkan, yang telah disinggung pada bagian A. Knapsack Standar pada kasus ini memilih sedikit file saja, sehingga sisa kapasitasnya masih banyak. Saat proses repair mencoba melengkapi prasyarat yang kurang, kelompok empat file tersebut (seberat 1.016 token) berhasil dimasukkan karena kapasitas masih cukup. Namun kelompok itu sendiri ternyata membutuhkan beberapa file lain yang belum ikut terpilih, dan pada titik itu kapasitas yang tersisa sudah hampir habis sehingga prasyarat-prasyarat baru tersebut tidak lagi bisa ditambahkan. Akibatnya, usaha melengkapi satu kekurangan justru memunculkan kekurangan baru yang lebih banyak.



Gambar 3. Rangkaian sebab-akibat anomali pada BUG-M2.

Kejadian ini mengungkap satu keterbatasan dari pendekatan repair yang digunakan, yakni ia menambahkan prasyarat begitu ditemukan kekurangan, tanpa mempertimbangkan terlebih dahulu apakah prasyarat tambahan itu sendiri akan membutuhkan prasyarat lain yang lebih besar lagi. Cara ideal untuk mengatasinya adalah dengan mempertimbangkan seluruh rantai keterkaitan sejak awal proses pemilihan, bukan menambalnya setelah pemilihan selesai. Pendekatan inilah yang menjadi salah satu arah pengembangan lanjutan yang akan disampaikan pada bab kesimpulan.

D. Hasil pada Cakupan Besar

Pada cakupan besar dengan 108 file, manfaat dari metode yang diusulkan terlihat paling jelas. Pada BUG-L1, jumlah pelanggaran berkurang dari 70 (Greedy) menjadi hanya 9. Pada BUG-L2 (Tabel V), pelanggaran berhasil dikurangi hingga nol, sekaligus menyertakan seluruh file ground truth.

Tabel V. Hasil Pengujian BUG-L2 (cakupan besar, kapasitas 8.000 token)

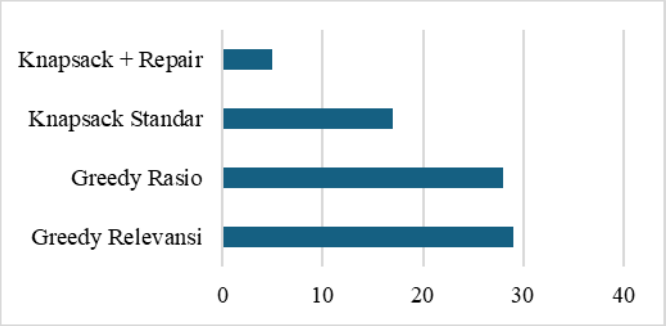
Metode	Pelanggaran	Recall	Waktu (ms)
Greedy Relevansi	42	100%	0,05
Greedy Rasio	42	100%	0,03
Knapsack Standar	16	100%	71,0
Knapsack + Repair	**0**	**100%**	**72,6**

Menariknya, Knapsack Standar pada kasus ini hanya memakai 2.430 dari 8.000 token kapasitas yang tersedia (kurang dari sepertiganya). Ia memilih file-file kecil bernilai tinggi tanpa mempedulikan keterkaitan. Metode yang diusulkan justru memakai 6.320 token (hampir 80% kapasitas), karena proses repair menambahkan file-file

prasyarat yang ukurannya lebih besar. *Trade-off* nya kapasitas yang sedikit lebih banyak terpakai, ditukar dengan konteks yang jauh lebih lengkap secara struktural.

E. Ringkasan Keseluruhan

Untuk melihat tren secara cepat, Gambar 4 menampilkan rata-rata jumlah pelanggaran dari keempat metode di seluruh enam skenario uji dalam bentuk grafik batang, sebelum melihat rincian angka pada Tabel VI.



Gambar 4. Rata-rata Jumlah Pelanggaran per Metode (Seluruh Skenario)

Tabel VI. Ringkasan Jumlah Pelanggaran pada Seluruh Skenario Uji

ID Bug	Greedy Rel.	Greedy Rasio	Knapsack Std	Knapsack+Repair
BUG-S1	9	9	8	5
BUG-S2	9	2	8	5
BUG-M1	**33**	**33**	**16**	**0**
BUG-M2	10	10	8	13
BUG-L1	70	70	45	9
BUG-L2	**42**	**42**	**16**	**0**

Dari enam skenario uji, metode Knapsack dengan Repair menghasilkan pelanggaran paling sedikit pada lima skenario, dengan satu pengecualian yang sudah dijelaskan secara mendalam pada bagian C. Pola yang konsisten muncul: semakin besar repositori yang diuji, semakin besar pula selisih jumlah pelanggaran antara metode yang diusulkan dan metode pembanding (dari selisih 4 pada BUG-S1 menjadi selisih 42 pada BUG-L2). Hal ini menunjukkan bahwa manfaat memperhatikan keterkaitan antarfile semakin terasa pada proyek-proyek besar.

VI. KESIMPULAN

Hasil percobaan dan analisis menunjukkan bahwa masalah memilih file kode mana yang perlu disertakan saat meminta bantuan AI coding agent memperbaiki sebuah bug dapat diselesaikan secara sistematis menggunakan Program Dinamis, dengan memandang setiap file sebagai barang dalam persoalan 0/1 Knapsack yang dilengkapi syarat tambahan berupa keterkaitan antarfile. Kontribusi utama makalah ini adalah menunjukkan secara konkret bagaimana persoalan praktis ini dapat dipetakan ke dalam pemodelan program dinamis, lengkap dengan cara mengukur relevansi file, cara menangani file yang saling membutuhkan, dan cara memastikan file yang terpilih tidak kehilangan prasyaratnya, serta membuktikan pada kode program sesungguhnya bahwa pendekatan ini benar-benar memberikan hasil yang lebih baik dibandingkan cara pemilihan file yang lebih sederhana.

Pada enam skenario bug yang diuji pada repositori berisi 110 file Java, metode yang diusulkan berhasil mengurangi jumlah fileyang kehilangan prasyaratnya secara signifikan dibandingkan cara pemilihan file berbasis greedy maupun program dinamis biasa, dengan keberhasilan menyertakan file yang sesungguhnya perlu diperbaiki tetap terjaga pada hampir seluruh skenario. Satu kasus pengecualian yang ditemukan justru memberi pemahaman baru bahwa proses perbaikan yang dilakukan satu per satu tanpa melihat ke depan dapat menimbulkan kebutuhan baru yang lebih besar, sehingga ke depannya pendekatan yang mempertimbangkan seluruh rantai keterkaitan sejak awal akan lebih kuat dibandingkan pendekatan tambal-sulam yang digunakan saat ini.

Beberapa arah pengembangan lanjutan yang dapat ditindaklanjuti antara lain mengintegrasikan langsung syarat keterkaitan ke dalam proses pencarian Program Dinamis sejak awal, menggunakan cara pengukuran relevansi yang lebih canggih berbasis pemahaman makna kalimat, serta mengembangkan pendekatan ini menjadi sebuah alat bantu praktis, misalnya ekstensi pada editor kode, yang dapat dipakai sehari-hari oleh pengembang perangkat lunak saat berinteraksi dengan AI coding agent.

LAMPIRAN

Kode sumber program pipeline seleksi file konteks (knapsack_pipeline.py) beserta dataset eksperimen tersedia pada repositori berikut: https://github.com/pepayaz/context-knapsack

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas kelancaran dalam penulisan makalah ini. Terima kasih kepada Pak Rila sebagai dosen pengampu mata kuliah IF2211 Strategi Algoritma atas bimbingan yang diberikan selama semester ini, juga kepada Pak Rinaldi atas materi yang disediakan di web nya, serta kepada rekan-rekan kelompok Tugas Besar IF2210 yang repositori proyeknya digunakan sebagai dataset eksperimen dalam penelitian ini.

REFERENSI

[1] Y. Tao, dkk., "Retrieval-Augmented Code Generation: A Survey with Focus on Repository-Level Approaches," arXiv:2510.04905 [cs.SE], Okt. 2025. Tersedia gratis: arxiv.org/abs/2510.04905

[2] R. Munir, "Program Dinamis (Dynamic Programming) Bagian 1," Bahan Kuliah IF2211 Strategi Algoritma, STEI-ITB, 2025. [Online].

Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, dan C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA: MIT Press, 2009, bab 15 dan 16. Tersedia untuk dipinjam gratis di Internet Archive: archive.org/details/introduction-to-algorithms-third-edition-2009

[4] H. Kellerer, U. Pferschy, dan D. Pisinger, Knapsack Problems. Berlin: Springer, 2004, bab 13.2, hal. 402 sampai 407. Daftar isi dan ringkasan tersedia gratis di laman penulis: hjemmesider.diku.dk/~pisinger/knapsack/toc.pdf

[5] R. Munir, "Graf Bagian 1," Bahan Kuliah IF2120 Matematika Diskrit, STEI-ITB, 2024. [Daring]. Tersedia: informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

[6] D. S. Harjanto, S. N. Endah, dan N. Bahtiar, "Sistem Temu Kembali Informasi pada Dokumen Teks Menggunakan Metode Term Frequency Inverse Document Frequency (TF-IDF)," Jurnal Sains dan Matematika, vol. 20, no. 3, hal. 64 sampai 70, 2012. Universitas Diponegoro. Tersedia gratis: ejournal.undip.ac.id/index.php/sm/article/view/8015

[7] R. Munir, "Program Dinamis (Dynamic Programming) Bagian 2," Bahan Kuliah IF2211 Strategi Algoritma, STEI-ITB, 2026. [Daring]. Tersedia: informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/26-Program-Dinamis-(2026)-Bagian2.pdf

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, 19 Juni 2026

Fayyaz Akmal Lauda 13524076