Token-Budgeted Context Packing for Retrieval-Augmented Generation with Dynamic Programming and Greedy Baselines

Nazwan Siddqi Muttaqin - 18223066
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: nazwan.siddqi@gmail.com , 18223066@std.stei.itb.ac.id

***Abstract*— Retrieval-Augmented Generation can expose a generator to external evidence, but retrieved passages may exceed the available context window. This paper formulates context packing as a 0/1 knapsack problem in which chunk length is the cost, relevance is the value, and the prompt allowance is the capacity. A lightweight Python prototype compares relevance top-k, relevance-density greedy, dynamic programming, and a redundancy-aware greedy heuristic. Experiments use 120 candidate passages converted from 12 validation queries in the Aryanp088/StratRAG dataset. Gold documents receive relevance 1.0, while distractors receive deterministic lexical-overlap scores. Across budgets from 500 to 3000 tokens, dynamic programming is optimal for the stated additive objective, but relevance-density greedy ties it at three budgets and differs by only 0.031 and 0.075 relevance at the other two. Top-k is fastest, whereas redundancy-aware greedy has the lowest mean lexical redundancy at substantially higher runtime. The evaluation isolates context selection on a flattened benchmark-derived sample; it does not evaluate retrieval or answer generation end to end.**

Keywords—Retrieval-Augmented Generation; Dynamic Programming; Greedy Algorithm; Knapsack; Token Budget; Context Selection

I. Introduction

Retrieval-Augmented Generation (RAG) is a common architecture for building language-model applications that need external knowledge. Instead of asking a generator to answer only from its parametric memory, a RAG pipeline first retrieves passages from a corpus and then places selected passages into the prompt. This design is attractive because the system can expose evidence to the generator, update the corpus without retraining the model, and reduce dependence on memorized facts [4]. In practice, however, the retrieval step usually returns more text than can fit into the final context window.

The context window is a finite computational resource. Each selected chunk uses some number of tokens, and every token assigned to one chunk cannot be used for another chunk, for the user's question, or for instructions to the generator. A simple top-k strategy that keeps the highest-ranked chunks may therefore be wasteful. A very relevant but long chunk can crowd out several shorter chunks whose combined evidence is more useful. The context selection problem is not merely a ranking problem; it is also a packing problem.

This paper studies token-budgeted context packing as an application of algorithmic strategy. The central question is: given a set of retrieved chunks, each with a relevance score and token length, which subset should be placed in the prompt so that total relevance is high and the token budget is not exceeded? This formulation maps naturally to the binary knapsack problem, where each item can be selected or discarded, and the objective is to maximize total value under a capacity constraint [1], [2].

The contribution of this paper is a reproducible prototype and experimental comparison of four strategies. First, relevance top-k represents a common retrieval baseline. Second, relevance-density greedy chooses chunks by relevance per token. Third, dynamic programming (DP) solves the 0/1 knapsack formulation exactly for integer token budgets. Fourth, redundancy-aware greedy adds a lightweight diversity penalty based on Jaccard similarity. The work focuses on the algorithmic part of RAG and intentionally does not call an external language model API.

II. Theoretical Foundation

A. Information Retrieval and RAG

Information retrieval (IR) studies how a system finds information objects that satisfy an information need. A user query is not usually expected to identify exactly one object; instead, many documents may partially match the query, and the system ranks them by estimated relevance [5]. RAG can be viewed as a modern use of IR inside a generation pipeline. The retriever proposes candidate passages, while the generator uses the selected passages as evidence when producing an answer.

In a classical retrieval setting, the primary output may simply be a ranked list of documents. In a RAG setting, the ranked list is only an intermediate artifact. The system must still decide how much of that retrieved material should be included in the generator input. This extra decision creates a bridge between IR and algorithm design: the ranking score estimates utility, but the prompt budget imposes a hard capacity constraint.

B. Chunks, Token Budgets, and Context Windows

A document is commonly split into chunks so retrieval can operate on paragraphs or passages rather than entire files. Chunking makes retrieval more precise, but it also creates a large pool of candidate items. Each candidate has a length measured in tokens. Although tokenization depends on the downstream model, the central constraint is model-independent: the final context has a maximum size.

A token budget is therefore an explicit engineering abstraction. It allows the context-building step to be evaluated independently from generation. When token counts are known, the selection algorithm can guarantee that the chosen subset fits within the prompt window. When exact tokenization is not available, a deterministic estimate such as whitespace token count can still be used for controlled experiments. The prototype in this project stores token counts directly in the dataset and estimates missing values only as a fallback.

C. Binary Knapsack

The binary knapsack problem consists of a set of items, where each item has a value and a weight, and a capacity constraint limits the total weight of the selected items. The objective is to select a subset of items that maximizes the total value without exceeding the capacity. The term binary indicates that each item has only two possible states: selected or not selected. This formulation is suitable for modeling chunk selection in Retrieval-Augmented Generation (RAG), where each retrieved chunk must be either included in or excluded from the final prompt.

In the RAG context-packing problem, let $C = \{c_1, c_2, ..., c_n\}$ denote the set of retrieved chunks. Each chunk c_i has a token cost $tokens(c_i)$ and a relevance score $relevance(c_i)$. The token budget B represents the maximum number of tokens allowed for the selected chunks. To represent the binary selection decision, a variable x_i is introduced as follows:

$$x_i = \begin{cases} 1, if\ chunk\ c_i\ is\ selected, \\ 0, otherwise. \end{cases} \quad (1)$$

The objective is to maximize the total relevance of the selected chunks:

$$max \sum_{i=1}^{n} x_i r_i \quad (2)$$

subject to the token budget constraint:

$$\sum_{i=1}^{n} x_i \times tokens(c_i) \leq B \quad (3)$$

where:

$$x_i \in \{0, 1\},\ i = 1, 2, ..., n \quad (4)$$

This formulation ignores generation quality directly, but it captures a clear and testable pre-generation objective. Therefore, the RAG context selection problem can be treated as a binary knapsack problem, where document chunks correspond to items, token counts correspond to weights or costs, relevance scores correspond to values or profits, and the context token budget corresponds to the knapsack capacity.

TABLE I. MAPPING RAG CONTEXT PACKING TO BINARY KNAPSACK

RAG Component	***Knapsack Component***
Document chunk	Item
Token count	Weight or cost
Relevance score	Value or profit
Context token budget	Capacity
Selected context	Selected item subset

D. Dynamic Programming

Dynamic programming solves problems by decomposing them into overlapping subproblems and storing subproblem results [3]. For 0/1 knapsack, define DP[*i*][*b*] as the best total relevance obtainable using only the first i chunks and capacity b. For chunk i with cost w_i and value v_i, the recurrence compares two possibilities: skip the chunk and keep DP[*i-1*][*b*], or take the chunk and add v_i to DP[*i-1*][*b*-w_i] if w_i is not greater than b. The larger value becomes DP[*i*][*b*].

The recurrence has two important consequences. First, the method is exact for the stated objective because every feasible include/exclude combination is represented implicitly through subproblems. Second, the cost is pseudo-polynomial: $O(nB)$ time and, in the straightforward table form, $O(nB)$ memory. This is acceptable for a controlled prototype and moderate budgets, but it may become expensive when the number of chunks or the token budget is very large.

E. Greedy Algorithms

Greedy algorithms make a sequence of local decisions according to a priority rule. They are often attractive because they are simple, fast, and easy to explain. For context packing, relevance top-k is one greedy rule: always consider the highest-relevance chunk first. A more budget-aware rule is relevance density, defined as relevance divided by tokens. This rule favors chunks that provide more relevance per token.

$$\rho_i = \frac{v_i}{w_i} \quad (5)$$

Greedy density resembles the optimal strategy for fractional knapsack, where items may be divided. However, RAG chunks are indivisible in this prototype: a chunk is either included or excluded. Therefore, density sorting does not guarantee optimality for 0/1 knapsack [1]. It can choose a locally efficient chunk and still miss a combination of medium-density chunks that fits better as a whole. This makes greedy methods useful baselines rather than exact solvers.

F. Redundancy and Diversity

Relevance alone is not always sufficient. A retriever may return several passages that discuss the same concept with slightly different wording. Selecting all of them can inflate relevance scores while adding little new evidence. In prompt construction, redundant chunks consume tokens that could have been used for complementary evidence.

Maximal Marginal Relevance (MMR) is a classic idea for balancing relevance and novelty in retrieval and summarization [6]. The redundancy-aware method in this project follows that spirit but remains intentionally lightweight. For each candidate chunk, it computes the maximum similarity to already selected chunks, subtracts a penalty from the relevance score, and divides the adjusted score by token count. A parameter lambda controls the balance between relevance and redundancy penalty.

G. Jaccard Similarity

Jaccard similarity measures overlap between two sets. For two normalized word sets A and B, it is the size of the intersection divided by the size of the union. A value of 0 means no shared terms, while a value of 1 means identical term sets. This metric is simple and lexical, so it does not understand semantic paraphrases. Nevertheless, it is useful for a deterministic prototype because near-duplicate chunks often share a large fraction of words.

$$\mathrm{J}(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{6}$$

H. Complexity Trade-Offs

The four methods represent different positions on the quality-speed spectrum. Top-k and greedy-density are dominated by sorting, so their complexity is approximately $O(n \log n)$. Dynamic programming is $O(nB)$, which can be slower but provides an exact reference point. Redundancy-aware greedy repeatedly compares candidates to selected chunks, so it may be slower than the other greedy methods even though it remains a heuristic rather than an exact solver.

$$T_{\text{topk}}(n) = T_{\text{density}}(n) = O(n \log n)$$
$$T_{\text{DP}}(n,B) = M_{\text{DP}}(n,B) = O(nB)$$

A RAG system can also be interpreted as a pipeline with three decision points: what to retrieve, what to keep, and how to order the kept evidence. This paper focuses on the second decision. The retrieval score is treated as an input, not as something learned by the prototype. This separation is useful for algorithm analysis because it isolates the packing problem from the many modeling choices involved in building an actual retriever.

Chunk length matters because a prompt is not an abstract list of documents. It is a serialized sequence of tokens. A chunk that is twice as long does not only take more storage; it directly reduces the remaining space for other evidence. This is the key reason that the context construction problem differs from simple document ranking. A ranking may say which chunks are individually relevant, but it does not say which combination should be selected under a shared capacity.

The 0/1 restriction is also important. In this prototype, a chunk is selected as a whole unit. The selector does not cut a chunk in half, summarize it, or rewrite it. This assumption keeps the problem close to binary knapsack and makes the algorithmic comparison clean. In a production system, chunk compression could be added as another stage, but then the weight and value of an item would no longer be fixed.

The DP table can be understood as a systematic record of all decisions that could have been made earlier. When item i is processed, the algorithm does not need to remember every subset explicitly. It only needs the best value for each capacity using previous items. This compression of history is the essence of the dynamic programming design. It is why the method can avoid enumerating all 2^n subsets while still reaching the optimum.

The token_usage table in the implementation is not necessary for the basic knapsack recurrence, but it improves tie handling. If two subsets have the same relevance, the subset with fewer tokens leaves more unused capacity. Although the current objective does not reward leftover capacity directly, choosing the lower-token solution is a reasonable deterministic convention and prevents arbitrary ties from changing across runs.

Greedy algorithms should not be dismissed simply because they are not always optimal. In many engineering systems, a fast and understandable heuristic is preferable to an exact method that is too expensive at scale. The relevant question is therefore not only which method wins on objective value, but also how much quality is lost in exchange for speed. This paper uses DP as the reference point for answering that question.

The redundancy-aware method illustrates a multi-objective tension. If relevance is the only objective, repeated evidence can still look valuable because every chunk has its own relevance score. If diversity is also valuable, then a chunk should be penalized when it overlaps strongly with selected evidence. The MMR-inspired score operationalizes this idea while still producing a single scalar score for greedy selection.

Jaccard similarity is intentionally simple. Its weakness is that it misses semantic equivalence between different words. Its strength is that it is transparent, deterministic, and cheap to compute. For an IF2211 technical report, this choice keeps the implementation focused on algorithmic strategy instead of introducing extra machine learning dependencies that would obscure the contribution.

I. Multi-Hop Retrieval Benchmarks

A multi-hop question may require evidence from more than one document. Candidate pools for such questions are useful for context packing because a selector must preserve complementary evidence while rejecting distractors. StratRAG provides a query, a document pool, and gold document indices for each example [7]. These fields are sufficient to construct a controlled packing instance without implementing a new retriever.

Gold membership is a categorical signal, whereas knapsack requires a numeric value for every item. The adaptation therefore assigns value 1.0 to gold documents and a smaller deterministic value to non-gold documents. For a distractor d and query q, the implemented score is 0.05 + 0.45 times the fraction of normalized query terms that also occur in d. This is a transparent proxy, not a calibrated probability or a learned relevance estimate.

The distinction between retrieval relevance and packing value is important. A gold indicator identifies evidence expected to support a question, but the additive objective assumes values from multiple chunks can be summed. That assumption makes the problem compatible with knapsack, yet it does not model evidence interactions, missing reasoning hops, or diminishing returns beyond the separate lexical redundancy metric.

III. Proposed Approach

The proposed approach is a context-packing layer placed between retrieval and generation. The layer receives a query, a list of candidate chunks, their token counts, their relevance scores, and a token budget. It returns a subset of chunks that can be inserted into a prompt. The layer does not perform retrieval itself and does not generate final answers. This separation keeps the contribution focused on algorithm selection.

The first method, top-k relevance, sorts chunks by decreasing relevance and adds a chunk whenever it fits. This method intentionally ignores token efficiency. It is included because it mirrors a common baseline: trust the retriever ranking and take the best passages until capacity is full.

The second method, greedy-density, sorts by relevance divided by token count. If two chunks have similar relevance, the shorter one receives higher priority. This method operationalizes the intuition that prompt space is scarce. A chunk should not only be relevant; it should also justify its cost.

The third method, dynamic programming, uses the binary knapsack recurrence. The implementation stores a value table, a token-usage table, and a keep table for reconstruction. Ties are resolved by choosing lower token usage, then deterministic chunk order. This tie-breaking rule makes repeated runs stable and avoids wasting capacity when two subsets have equal relevance.

The fourth method, redundancy-aware greedy, modifies the greedy score. At each step, it computes the maximum Jaccard similarity between a feasible candidate and the chunks already selected. The adjusted score is lambda times relevance minus one minus lambda times that maximum similarity. The result is divided by token count to obtain a density-like score. The default lambda is 0.75, so relevance remains dominant while redundancy still matters.

$$\mathrm{a}(c) = \lambda v(c) - (1-\lambda)\max_{s \in S} J(c,s) \tag{7}$$

$$\mathrm{score}(c) = \frac{a(c)}{w(c)}, \lambda = 0.75 \tag{8}$$

TABLE II. Summary of Selection Methods

Method	*Main Criterion*	*Expected Role*
topk	Relevance	Simple retrieval baseline
greedy-density	Relevance per token	Fast token-aware heuristic
dp	Optimal knapsack value	Exact reference solution
redundancy-aware	Density with overlap penalty	Diversity-oriented heuristic

The input abstraction can be written as a collection $C = \{c_1, c_2, ..., c_n\}$. Each c_i contains a text field, an integer token cost, and a floating-point relevance score. The selector receives a budget B and returns a subset S of C. A valid output must satisfy total_tokens(S) <= B. Among valid outputs, the main knapsack objective prefers the subset with maximum total_relevance(S).

The top-k baseline uses the same validity constraint but not the same global optimization process. It visits chunks in sorted relevance order and accepts a chunk only if it fits. If a chunk does not fit, the method skips it and continues to the next chunk. This detail is important: the method is not simply taking the first k chunks; it is filling a budget with a relevance-ordered scan.

The greedy-density method changes only the sorting key. It ranks chunks by relevance/tokens, then by higher relevance, then by lower token count, and then by chunk_id for determinism. This tie-breaking order reflects the idea that a chunk with the same density but higher absolute relevance is more useful, while a chunk with equal density and relevance but lower token cost is safer.

The DP selector uses a two-dimensional table because the paper prioritizes clarity over memory optimization. A one-dimensional DP array could reduce memory usage, but reconstructing the exact chosen chunks would require additional bookkeeping. The two-dimensional implementation makes the recurrence and backtracking process easier to inspect and explain in a teaching-oriented paper.

For redundancy-aware greedy, the selected set changes the score of every remaining candidate. This means the initial order is not fixed. After each selection, the method recomputes candidate scores using the maximum similarity to the current selected set. The method is therefore adaptive: a chunk that was attractive before may become less attractive after a similar chunk is selected.

The four methods also differ in the assumptions they make about the retrieval score. Top-k assumes the score is sufficient by itself. Density greedy assumes the score should be normalized by token cost. DP assumes the score is additive and optimizes the additive objective exactly. Redundancy-aware greedy assumes that marginal value decreases when selected evidence already covers similar terms.

For the StratRAG conversion, each document in a query's doc_pool becomes one item. The chunk identifier combines the query identifier and original document index, the document identifier is preserved, and the source title is prepended to the body when needed. Placeholder entries with source __pad__ or text __no_content__ are discarded before writing the local JSONL file.

Token cost is estimated as whitespace-separated word count. Gold documents are assigned relevance 1.0. Non-gold documents receive the lexical-overlap score defined above, rounded to three decimals. The local records also retain query_id, query, source, is_gold, and source_dataset for traceability, although the runtime Chunk dataclass consumes only the five fields required by the selectors.

$$\mathrm{w}(c) = \max\left(1, |\mathrm{split}_{ws}\left(\mathrm{text}(c)\right)|\right) \tag{9}$$

$$r(d,q) = 1.0, d \in D_{\text{gold}} \tag{10}$$

$$r(d,q) = 0.05 + 0.45\frac{|T(q) \cap T(d)|}{|T(q)|}, d \notin D_{\text{gold}} \tag{11}$$

The experiment runner currently loads the converted JSONL as one candidate list. Thus the 12 query-specific pools form one aggregate packing instance. This design allows the four algorithms to be compared on realistic passage lengths and benchmark-derived labels, but it does not preserve a separate context window for each query. This scope is reported explicitly in the validity analysis.

IV. Implementation

The prototype is implemented in Python 3.11 with a src package layout. The models module defines an immutable Chunk with chunk_id, doc_id, text, tokens, and relevance. SelectionResult stores the chosen chunks, total token use, summed relevance, method name, and measured runtime. Keeping the algorithm input small makes the selectors independent of Hugging Face and of any language-model framework.

Dataset acquisition is isolated in scripts/import_stratrag.py. The script requests rows from the Hugging Face Dataset Viewer API for Aryanp088/StratRAG [7], converts each doc_pool, and writes data/stratrag_chunks.jsonl. Its default command imports 12 validation examples beginning at offset zero. Download logic is not part of runtime loading, so the CLI and experiments remain reproducible and offline after the JSONL file has been created.

The checked-in sample contains 120 chunks from 12 queries, with exactly 10 retained candidates per query and 24 gold chunks in total. Passage lengths range from 22 to 412 whitespace tokens, with a mean of 95.38. The complete aggregate pool contains 11446 tokens. These figures are computed from the local JSONL used by the experiment rather than copied from a dataset description.

The runtime data module now has only three responsibilities: estimate missing token counts, coerce JSON objects into Chunk instances, and load local JSONL. It deliberately does not fetch or transform StratRAG. This separation makes network access an explicit preprocessing action and keeps normal program execution deterministic.

The scoring module lowercases text, extracts alphanumeric word tokens, computes Jaccard similarity, and averages similarity over selected pairs. The same normalizer supports distractor scoring in the importer. This reuse keeps lexical processing consistent, but Jaccard remains insensitive to paraphrases that share few surface terms.

The selectors module implements the four methods described in Section III. Top-k and greedy-density sort once. Dynamic programming constructs value, token-usage, and keep tables for all items and capacities, then backtracks to recover a deterministic optimal subset. Redundancy-aware greedy recomputes the maximum similarity of every feasible candidate after each selection, which explains its larger practical cost.

The command-line interface now defaults to data/stratrag_chunks.jsonl. The experiment runner likewise defaults to the stratrag dataset label, but it also accepts synthetic or custom JSONL input. It evaluates budgets 500, 1000, 1500, 2000, and 3000, then writes a CSV, a Markdown summary, and three plots. No automatic download occurs during experiments; a missing dataset produces an actionable error.

The evaluation row records method, budget, total tokens, budget-use ratio, summed relevance, selected count, average pairwise redundancy, and runtime in milliseconds. The plots are generated from the same rows saved to results_stratrag, and the journal builder reads that CSV directly. This data flow reduces the risk that prose, tables, and plotted values describe different experiment runs.

Eight automated tests currently pass. They cover the known optimum of a small knapsack instance, budget compliance for every selector, deterministic greedy behavior, redundancy bounds, evaluation schema, and StratRAG conversion. The mapping test verifies that a gold document receives 1.0, a distractor receives a value between zero and one, and chunk identifiers preserve query and document position.

V. Experiments and Results

The experiment evaluates all four methods on the converted StratRAG sample at token budgets of 500, 1000, 1500, 2000, and 3000. Every method receives the same 120-item aggregate pool for a given budget. Metrics expose objective value, capacity use, selection breadth, lexical redundancy, and computational cost. The reported runtime is a single in-process measurement and should be interpreted as comparative evidence on the current machine, not as a hardware-independent benchmark.

The principal metric is total relevance because it is exactly the objective optimized by dynamic programming. Budget compliance is a hard validity condition. Selected count helps explain whether a method favors many short passages or fewer long ones. Average pairwise Jaccard similarity measures lexical repetition, while runtime reveals the cost of exact optimization and adaptive diversity scoring.

$$R(S) = \sum_{c_i \in S} v_i \tag{12}$$

$$U(S) = \frac{\sum_{c_i \in S} w_i}{B} \tag{13}$$

$$\text{Red}(S) = \frac{2}{|S|(|S|-1)} \sum_{i<j} J(c_i, c_j) \tag{14}$$

Dynamic programming achieves the best or tied-best total relevance at every tested budget, as required by the 0/1 knapsack recurrence. At 500 tokens it obtains 10.325, only 0.031 above greedy-density. The two methods tie at budgets 1000, 1500, and 2000. At 3000 tokens, DP obtains 29.676, compared with 29.601 for greedy-density, a gap of 0.075.

TABLE III. RESULT AT 1500 TOKEN BUDGET

Method	*Tokens*	*Relevance*	*Redundancy*	*Runtime*
topk	1497	19.187	0.077	0.110 ms
greedy-density	1496	21.121	0.115	0.089 ms
dp	1496	21.121	0.115	59.208 ms
redundancy-aware	1491	20.462	0.077	1078.828 ms

TABLE IV. MEAN RELEVANCE BY METHOD

Method	*Mean Total Relevance*
dp	20.525
greedy-density	20.504
redundancy-aware	19.876
topk	17.720

At the representative budget of 1500, DP and greedy-density both select 29 chunks, use 1496 tokens, and reach relevance 21.121. Top-k selects 20 chunks and reaches 19.187. Redundancy-aware greedy selects 22 chunks and reaches 20.462. All four methods use more than 99 percent of the available capacity at this budget.

Averaged over all five budgets, DP has mean relevance 20.525, greedy-density 20.504, redundancy-aware greedy 19.876, and top-k 17.720. The small DP-greedy difference shows that density happens to align well with this converted instance. It does not establish a general optimality guarantee for greedy selection.

Top-k is the fastest method on average and greedy-density remains in the same sub-millisecond range. At 1500 tokens their measured runtimes are 0.110 ms and 0.089 ms. DP takes 59.208 ms because it fills an n by B table. Redundancy-aware greedy takes 1078.828 ms because candidate-to-selected similarities are repeatedly recomputed.

Redundancy-aware greedy has the lowest mean pairwise redundancy across the five budgets at 0.078. At 1500 tokens its redundancy is 0.077, compared with 0.077 for top-k and 0.115 for DP. However, it is not the least redundant method at every individual budget. The result supports a mean trade-off, not a universal per-budget dominance claim.

The plots reinforce three observations. First, relevance increases with capacity for all methods. Second, DP's runtime grows with the token budget while the two sorting baselines remain nearly flat at this scale. Third, redundancy-aware runtime grows most sharply. Figures 1-3 are generated from results_stratrag/experiment_results.csv, the same source used for Tables III and IV.

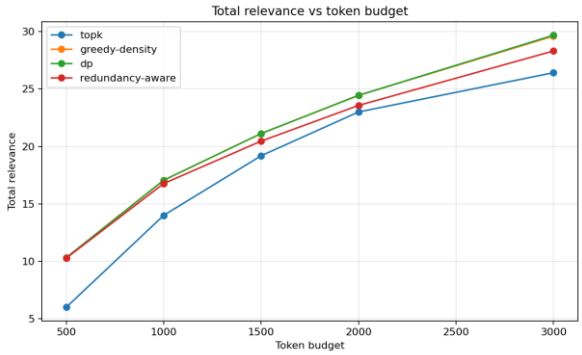

Fig. 1. Total relevance across token budgets

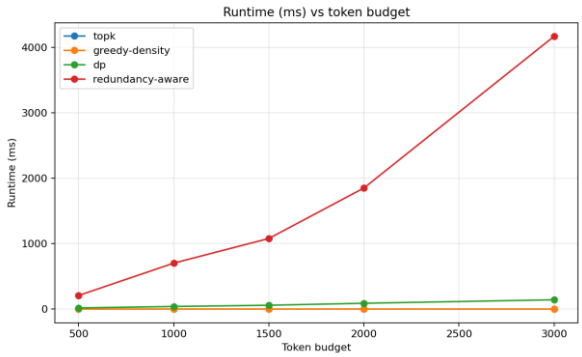

Fig. 2. Runtime across token budgets

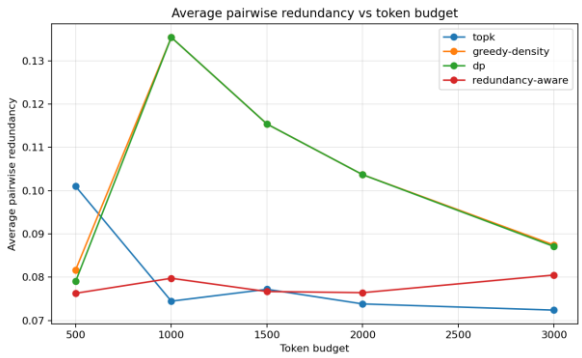

Fig. 3. Average pairwise redundancy across token budgets

The numerical results are deterministic for token use, relevance, selected count, and redundancy because the input and tie-breaking rules are fixed. Runtime is expected to vary between executions. Reproducibility therefore means that qualitative runtime ordering and all non-timing metrics can be regenerated, not that every millisecond value will be identical.

VI. Discussion

The StratRAG integration strengthens ecological relevance compared with synthetic technical sentences. Passage lengths, entities, and distractors now originate from a retrieval benchmark [7]. The experiment therefore exercises packing algorithms on less regular text and on candidate groups built for multi-hop questions.

Dynamic programming remains the correct exact method for the implemented objective. Its ties with greedy-density are not contradictions; an exact method may share an optimum with a heuristic. The important guarantee is that DP cannot return a lower summed relevance than another feasible subset under the same costs and values. The small observed gaps explain why density is attractive when latency matters.

Top-k performs worse because all gold documents have the same maximum value 1.0, so its deterministic tie order does not account for length. It can spend capacity on long gold chunks before shorter high-value evidence. Density and DP explicitly incorporate cost and can pack more value into the same allowance.

The converted relevance scale also shapes the result. Gold chunks have a large value advantage over distractors, while distractor scores are bounded by a simple overlap formula. This is useful for controlled comparison, but it means the experiment measures optimization over an engineered value function. It does not validate the lexical score as a production reranker.

The most important internal-validity limitation is aggregation across queries. The imported JSONL retains query identifiers, but load_chunks maps records into a core Chunk that does not keep query_id. The experiment consequently selects from all 120 chunks at once. A deployed RAG system would normally create one candidate pool and one token budget per user query.

Flattening is acceptable as an aggregate knapsack stress test, but it prevents claims about per-query gold coverage, multi-hop completion, or answer correctness. It also allows the selected subset to contain evidence for unrelated questions. A stronger evaluation should group records by query_id, run every selector independently for each group, and report macro-averaged relevance, gold recall, complete-hop coverage, redundancy, and runtime.

The whitespace token estimator is another approximation. Actual language-model tokenizers may split names, punctuation, and rare words differently. The current costs are deterministic and adequate for comparing algorithms, but absolute budgets should not be translated directly into a specific model's context limit.

Additive relevance is deliberately simple. Two gold documents may be jointly necessary for a multi-hop answer, making their combined value non-additive. Conversely, two passages may repeat the same fact. The Jaccard-based method addresses lexical repetition partially, but the exact DP objective does not encode hop coverage or semantic complementarity.

The redundancy-aware heuristic illustrates a cost-quality tension but is inefficient in its current form. It repeatedly tokenizes and compares text through nested loops. Caching normalized word sets and pairwise similarities would preserve the selection rule while reducing constant factors. Semantic embeddings could improve paraphrase detection, although they would introduce model dependencies outside this paper's lightweight scope.

DP's O(nB) time and memory are manageable for 120 items and capacities up to 3000. They become less attractive for much larger budgets or candidate pools. A one-dimensional value array can reduce memory, while approximation schemes or greedy selection may be preferable for interactive systems. DP nevertheless remains useful as an oracle for quantifying heuristic loss on moderate instances.

Future work should first correct the evaluation granularity rather than immediately adding a generator. Per-query experiments would make the StratRAG gold annotations meaningful and permit confidence intervals across examples. After that, an end-to-end study could measure exact match, answer faithfulness, citation coverage, and latency with a fixed retriever and language model.

Within its stated scope, the project still satisfies the IF2211 algorithmic objective. It formulates a modern engineering decision as 0/1 knapsack, implements exact dynamic programming and several greedy strategies, validates core behavior with tests, and reports reproducible trade-offs. StratRAG improves the input evidence, while the explicit limitations keep the conclusion proportional to what was actually measured.

VII. Conclusion

This paper models token-budgeted RAG context packing as a 0/1 knapsack problem and compares relevance top-k, relevance-density greedy, dynamic programming, and redundancy-aware greedy selection. The revised experiment uses 120 passages converted from 12 StratRAG validation queries rather than relying only on synthetic text. Gold membership and deterministic lexical overlap provide the values optimized under five token budgets.

Dynamic programming is optimal for the implemented additive objective, but density greedy reaches the same value at three budgets and nearly the same value at the other two with much lower runtime. Top-k is fastest but weaker in relevance, while the redundancy-aware heuristic lowers mean lexical overlap at a substantial runtime cost. These results demonstrate the practical trade-off between exactness, efficiency, and diversity in context construction.

The findings are limited to aggregate packing over a flattened benchmark-derived sample and do not establish per-query retrieval or generation quality. Grouped per-query evaluation is the most important next step. Even with that limitation, the prototype provides a clear and testable application of dynamic programming and greedy algorithmic strategies to a relevant RAG engineering problem.

APPENDIX

The methods and experiments presented in this paper are implemented in the following GitHub repository: NazwanSM/Paper-IF2211-token-budgeted-rag-context-packing

ACKNOWLEDGMENT

The author expresses heartfelt gratitude to God Almighty for His blessings and grace throughout the writing process, allowing the smooth completion of this paper. The author would also like to sincerely thank Prof. Dr. Ir. Rinaldi, M.T., the IF2211 Algorithm Strategies K1 course lecturer, for his invaluable guidance and knowledge that significantly contributed to completing this work.

Additionally, the author is deeply grateful to their parents for their moral and spiritual support as the source of inspiration and motivation during the writing journey. Finally, the author wishes to thank the friends in the STIma, cohort for their mutual support during lectures and for learning together throughout the semester.

May the Almighty God reward all the kindness that has been bestowed. May this paper serve as a valuable contribution to the academic community and be well-received.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] H. Kellerer, U. Pferschy, and D. Pisinger, Knapsack Problems. Berlin, Germany: Springer, 2004.

[3] R. Bellman, Dynamic Programming. Princeton, NJ, USA: Princeton University Press, 1957.

[4] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Advances in Neural Information Processing Systems, 2020.

[5] C. D. Manning, P. Raghavan, and H. Schutze, Introduction to Information Retrieval. Cambridge, U.K.: Cambridge University Press, 2008.

[6] J. Carbonell and J. Goldstein, "The use of MMR, diversity-based reranking for reordering documents and producing summaries," in Proc. 21st Annu. Int. ACM SIGIR Conf. Research and Development in Information Retrieval, 1998, pp. 335-336.

[7] Aryanp088, "StratRAG," Hugging Face Datasets. [Online]. Available: https://huggingface.co/datasets/Aryanp088/StratRAG. [Accessed: Jun. 15, 2026].

STATEMENT

In this statement, I declare that this paper I have written is my own work, not a duplication or translation of someone else's paper, and is not plagiarized.

Bandung, 19 Juni 2026

Nazwan Siddqi Muttaqin
18223066