Efficiency Comparison of Hashing and String Comparison Methods for de Bruijn Graph Construction from k-mer Sequences

Indronil Bhattacharjee, Jaspreet Thind

Abstract

Genome assembly is a crucial aspect of examining sequencing data received from high-throughput methods. Although several tools are accessible to biologists, de Bruijn graph-based assemblers are still prevalent. However, selecting the appropriate k value, which determines the size of k-mers obtained by breaking down reads, is challenging. Longer repeats, which are larger than k nucleotides, may cause the graph to tangle and disrupt contigs, whereas smaller k values decrease the accuracy of k-mers. In this study, we examined the effectiveness of hashing and string techniques for developing de Bruijn graphs from k-mer sequences. We discovered that the hashing approach is quicker than the string comparison method, particularly for larger k-mers (>2000). We also encountered a problem while working on the extra credit question on (k,d)-mer assembler, which we are still working on. In general, this study provided a valuable practical experience for comprehending genome assembly algorithms and their biological importance.

Introduction

Genome assembly continues to be an essential component in the analysis of sequencing data obtained through high-throughput methods. Over the years, there have been several enhancements to the initial methods, resulting in a plethora of tools available to biologists that offer various alternatives (Bankevich et al., 2012) (Luo et al., 2012) (Ribeiro et al., 2012) (Simpson & Durbin, 2012). A significant number of these tools use the de Bruijn graph approach, which involves breaking down reads into k-mers (substrings of length k). The de Bruijn graph strategy involves creating nodes using (k-1)-mers and edges with k-mers from the reads. Essentially, an assembler builds the graph, streamlines it through various simplification steps, and generates contigs, which are uninterrupted genomic regions that the assembler predicts.

In recent times, various meta-analyses on assemblers have highlighted inherent limitations in present techniques (Bradnam et al., 2013) (Salzberg et al., 2012) (Earl et al., 2011). The primary parameter in de Bruijn-based assemblers is k, which determines the size of k-mers obtained by splitting the reads. Longer repeats exceeding k nucleotides can tangle the graph and interrupt contigs, so a larger k value is preferable. However, increasing k also raises the likelihood of k-mers having errors, leading to a decrease in the number of accurate k-mers in the data. Additionally, when two reads overlap by less than k characters, they do not share a vertex in the graph, creating a coverage gap that can split a contig. Consequently, selecting k involves weighing several factors against each other. In this project, we evaluated the efficiency of hashing and string methods to compare their robustness to create a de Bruijn graph from k-mer sequences.

Methods

Task 1: DNAHasher

The DNAHasher is a struct that defines a custom hash function for DNA sequences. It overloads the operator() method, which takes a const string& as input and returns a size_t value as the hash code. The hash function uses a lookup table that maps each nucleotide ('A', 'C', 'G', 'T') to a unique integer code (0, 1, 2, 3). It iterates over the input sequence, converts each nucleotide to uppercase, looks up its code in the table, and appends it to the hash value by bit-shifting and bitwise OR-ing. The hash function only considers up to the first max_width characters of the sequence to limit the computation time and prevent potential collisions due to long sequences. This hash function can be used for efficient storage and retrieval of DNA sequences in hash tables, as it maps similar sequences to the same hash code with high probability, while minimizing the risk of hash collisions.

Task 2: create_deBruijn_graph_by_hashing()

The function create_deBruijn_graph_by_hashing creates a De Bruijn graph from a collection of k-mers by using a hash table to store and retrieve the node IDs of the nodes in the graph. The resulting graph is stored in a DiGraph object g. The function begins by creating a CSeqHash object node_ids to store the mapping between k-1 prefixes and node IDs in the De Bruijn graph. The create_hash_table function is called to create the hash table from the collection of k-mers. Next, an empty node vector is created for the DiGraph object g. For each k-mer in the input collection, the function retrieves the node ID for the k-1 prefix from the hash table, creates a new node for the prefix if necessary, retrieves the node ID for the k-1 suffix from the hash table, and creates a new node for the suffix if necessary. The function then creates a new edge in the graph from the prefix node to the suffix node by adding the suffix node's ID to the adjacency list of the prefix node and incrementing the suffix node's count of incoming edges. Once all k-mers have been processed, the resulting DiGraph object g contains the De Bruijn graph represented by the collection of k-mers.

Task 3: find_Eulerian_cycle()

The find_Eulerian_cycle() function takes a directed graph g as input and returns a list of node ids representing an Eulerian cycle on the graph if it exists. An Eulerian cycle is a cycle that visits every edge of the graph exactly once, and starts and ends at the same node. The function first initializes an empty list cycle to hold the main cycle. It then uses a stack and a boolean vector used to implement a depth-first search of the graph, starting from an arbitrary node with outgoing edges (found using a loop over all nodes). The search visits each node and its outgoing edges exactly once, and pushes nodes onto the stack as it traverses the graph. When a node with no outgoing edges is reached, it is popped off the stack and added to the front of the cycle list. The search continues until the stack is empty. Finally, the function checks whether all edges of the graph have been used in the search, throwing an exception if any unused edges are found.

Bonus Task: (K,d)-mer sequence assembler

(K,d)-mer assembler follows almost the same process, only the m_label of the node is replaced with a pair <string, string> in place of a string to transform it from a read to read-pair.

Compile all the .cpp files to executable main.exe with g++ compiler:

```
g++ main.cpp test.cpp sequence.cpp k-assembler.cpp kd-assembler.cpp EulerPath.cpp deBruijnByHash.cpp deBruijnByStringComp.cpp -o main
```

Run the executable:

```
.\main
```

Output:

```
-----------
Testing k-assembler by k-mer pairwise comparison

Example 0:
Elapsed time for building de Bruijn graph: 0
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 0 (assembled original sequence). Congratulations!

Example 1:
Elapsed time for building de Bruijn graph: 0
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 1 (assembled original sequence). Congratulations!

Example 2:
Elapsed time for building de Bruijn graph: 0
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 2 (assembled a sequence of the same composition with the original sequence). Congratulations!

Example 3:
Elapsed time for building de Bruijn graph: 8.901
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 3 (assembled original sequence). Congratulations!

Passed Test 2 (assembled original sequence). Congratulations!

-----------
Testing k-assembler by k-mer hashing

Example 0:
Elapsed time for building de Bruijn graph: 0
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 0 (assembled original sequence). Congratulations!

Example 1:
Elapsed time for building de Bruijn graph: 0.001
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 1 (assembled original sequence). Congratulations!

Example 2:
Elapsed time for building de Bruijn graph: 0
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 2 (assembled original sequence). Congratulations!

Example 3:
Elapsed time for building de Bruijn graph: 0.276
Passed test for existence of Eulerian path. Congratulations!
Passed Test 1 Example 3 (assembled original sequence). Congratulations!

Passed Test 2 (assembled original sequence). Congratulations!

-----------
Sequence: gcatactccggcctg
kmers:
ccgg
cctg
actc
ggcc
gcat
atac
cata
tccg
ctcc
gcct
tact
cggc
Passed Test 3 (assembled original sequence). Congratulations!
-----------
```

```
Sequence: ttctgcgccccgcgc
kdmers:
ctg-ccc
gcg-ccg
ttc-cgc
gcc-gcg
tgc-ccc
tct-gcc
cgc-cgc
ccc-cgc
ttctgcgccccgcgc
Passed Test 4 (assembled original sequence). Congratulations!
```

Fig 1: Output of the code

Table 1: Time required for String Comparison and Hashing method for different values of K

K	Time for String Comparison method	Time for Hashing method
1000	0.01	0.002
2000	0.016	0.003
5000	1.03	0.024
10000	3.38	0.163
20000	8.9	0.276

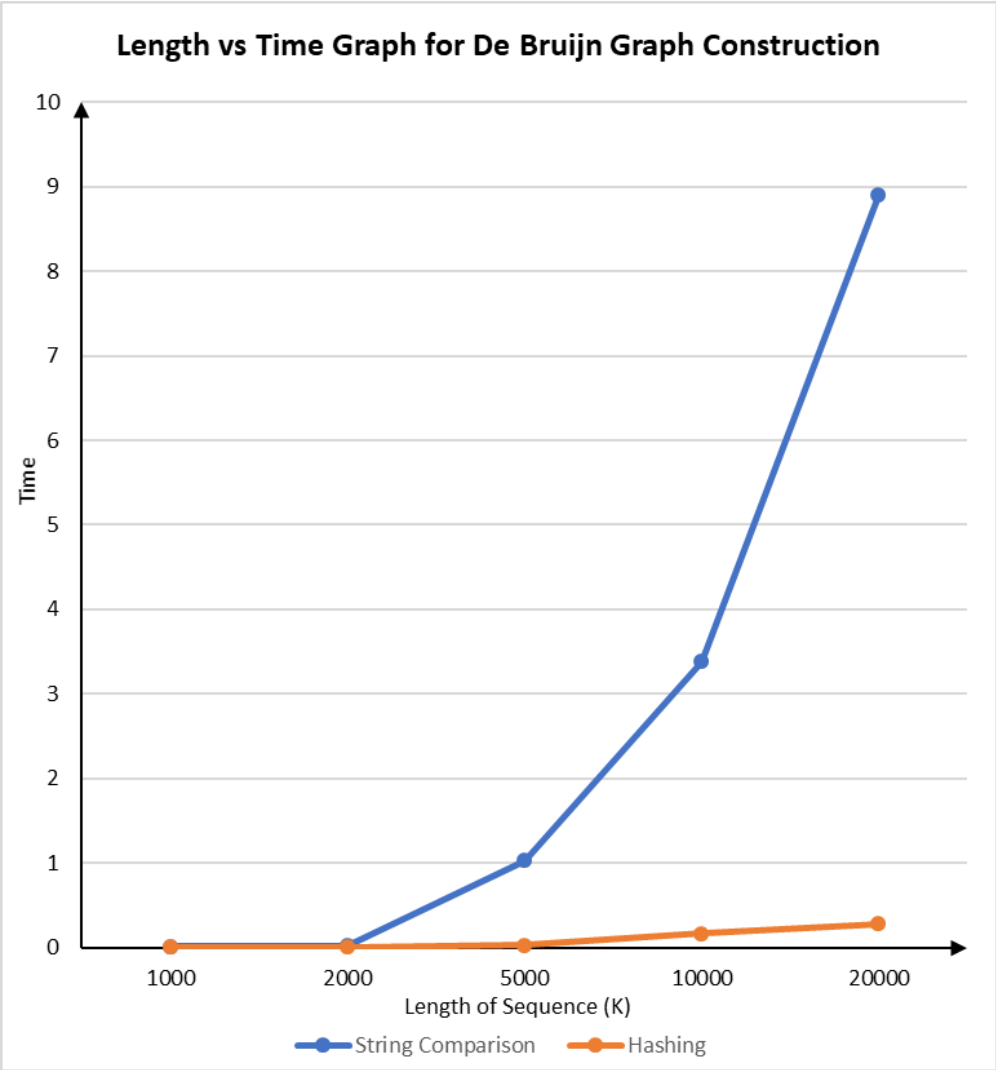

Fig 2: Length vs Time graph for De Bruijn Graph construction

Discussion

The implementation of all three tasks using C++ was not challenging. However, while attempting the extra credit question on (k,d)-mer assembler, we encountered an issue where the function sometimes produced an error, and we are still addressing it. The findings indicated that the hashing technique is quicker than the string comparison method. Interestingly, for larger k-mers (>2000), the time disparity between the two methods is significant. Further examination of these methods on actual assembler reads may yield more critical and meaningful outcomes. This project provided a valuable hands-on opportunity to understand assembly algorithms for sequencing the genome and their biological significance.

Distribution of work

Jaspreet: After Indronil explained the code of all three tasks, I was able to write the code for the first two tasks without much difficulty but struggled with task 3. Kudos to him!! and wrote the report.

Indronil: Coded for all three tasks and extra credit questions and wrote the report.

References

(1) Bankevich, A., Nurk, S., Antipov, D., Gurevich, A. A., Dvorkin, M., Kulikov, A. S., Lesin, V. M., Nikolenko, S. I., Pham, S., & Prjibelski, A. D. (2012). SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. Journal of Computational Biology, 19(5), 455–477.

(2) Bradnam, K. R., Fass, J. N., Alexandrov, A., Baranay, P., Bechner, M., Birol, I., Boisvert, S., Chapman, J. A., Chapuis, G., & Chikhi, R. (2013). Assemblathon 2: Evaluating de novo methods of genome assembly in three vertebrate species. *Gigascience*, *2*(1), 2047-217X.

(3) Earl, D., Bradnam, K., John, J. S., Darling, A., Lin, D., Fass, J., Yu, H. O. K., Buffalo, V., Zerbino, D. R., & Diekhans, M. (2011). Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, *21*(12), 2224–2241.

(4) Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., & Liu, Y. (2012). *SOAPdenovo2: An empirically improved memory-efficient short-read de novo assembler. GigaScience 1: 18.*

(5) Ribeiro, F. J., Przybylski, D., Yin, S., Sharpe, T., Gnerre, S., Abouelleil, A., Berlin, A. M., Montmayeur, A., Shea, T. P., & Walker, B. J. (2012). Finished bacterial genomes from shotgun sequence data. *Genome Research*, *22*(11), 2270–2277.

(6) Salzberg, S. L., Phillippy, A. M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T. J., Schatz, M. C., Delcher, A. L., & Roberts, M. (2012). GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, *22*(3), 557–567.

(7) Simpson, J. T., & Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, *22*(3), 549–556.