Cuttlefish: Fast, parallel, and low-memory compaction of de Bruijn graphs from large-scale genome collections

Jamshed Khan and Rob Patro

Department of Computer Science and
Center for Bioinformatics and Computational Biology
University of Maryland, College Park, MD 20742, USA
{jamshed, rob}@cs.umd.edu

Abstract
Motivation: The construction of the compacted de Bruijn graph from a large collection of reference genomes is a task of increasing interest in genomic analyses. For example, compacted colored reference de Bruijn graphs are increasingly used as sequence indices for the purposes of alignment of short and long reads. Also, as we sequence and assemble a greater diversity of individual genomes, the compacted colored de Bruijn graph can be used as the basis for methods aiming to perform comparative genomic analyses on these genomes. While algorithms have been developed to construct the compacted colored de Bruijn graph from reference sequences, there is still room for improvement, especially in the memory and the runtime performance as the number and the scale of the genomes over which the de Bruijn graph is built grow.
Results: We introduce a new algorithm, implemented in the tool Cuttlefish, to construct the colored compacted de Bruijn graph from a collection of one or more genome references. Cuttlefish introduces a novel modeling scheme of the de Bruijn graph vertices as finite-state automata, and constrains the state-space for the automata to enable tracking of their transitioning states with very low memory usage. Cuttlefish is also fast and highly parallelizable. Experimental results demonstrate that the algorithm scales much better than existing approaches, especially as the number and scale of the input references grow. For example, on a typical shared-memory machine, Cuttlefish constructed the compacted graph for 100 human genomes in less than 7 hours, using ~29 GB of memory; no other tested tool successfully completed this task on the testing hardware. We also applied Cuttlefish on 11 diverse conifer plant genomes, and the compacted graph was constructed in under 11 hours, using ~84 GB of memory, while the only other tested tool able to complete this compaction on our hardware took more than 16 hours and ~289 GB of memory.
Availability: Cuttlefish is written in `C++14`, and is available under an open source license at `https://github.com/COMBINE-lab/cuttlefish`.
Contact: `rob@umd.edu`

Keywords: De Bruijn Graph · Compacted De Bruijn Graph · Colored De Bruijn Graph · Minimal Perfect Hash · Finite-State Automaton

1 Introduction

With the increasing affordability and throughput of sequencing, the set of whole genome references available for comparative analyses and indexing has been growing tremendously. Modern sequencing technologies can generate billions of short read sequences per-sample, and with state-of-the-art *de novo* and reference-guided assembly algorithms, we now have thousands of mammalian-sized genomes available. Moreover, we have now successfully assembled genomes that are an order of magnitude larger than typical mammalian genomes, with the largest among these being the sugar pine (~31 Gb) [48] and the Mexican walking fish (~32 Gb) [38]. With modern long-read sequencing technologies and assembly techniques, the diversity and completeness of reference-quality genomes is expected to continue increasing rapidly. Representation of these reference collections in compact forms that facilitates common genomic analyses are thus of acute interest and importance, as are efficient algorithms for constructing these representations.

To this end, the de Bruijn graph has become an object of central importance in many genomic analysis tasks. While it was initially used mostly in the context of genome (and transcriptome) assembly (EULER [42],

Velvet [51,52], ALLPATHS [9,30], EULER-SR [10], ABySS [46], SOAPdenovo [25,29], Trans-AByss [43], SPAdes [5], Minia [13]), it has seen increasing use in comparative genomics (Cortex [19], DISCOSNP [50], Scalpel [15], BubbZ [34]) and has also been used increasingly in the context of indexing genomic data, either from raw sequencing reads (Mantis [40,1], Vari [37], VariMerge [36], MetaGraph [20]), or from assembled reference sequences (deBGA [27], Pufferfish [2], deSALT [28]), or from both (BLight [32], Bifrost [17]). These latter applications most frequently make use of the (colored) compacted de Bruijn graph, a variant of the de Bruijn graph in which maximal non-branching paths (unitigs) are condensed into single vertices in the underlying graph structure. This retains all the information of the original graph, while typically requiring much less space to store, index, and process. The set of maximal unitigs for a de Bruijn graph is unique and forms a node decomposition of the graph [12].

The colored variant is built over multiple references. The (colored) compacted de Bruijn graph has become an increasingly useful data structure in computational genomics, for its use in detecting and representing variation within a population of genomes, comparing whole genome sequences, as well as, more recently, its accelerated use as a foundational structure to assist with indexing one or more related biological sequences. Applying de Bruijn graphs for these whole genome analysis tasks highlights a collection of algorithmic challenges, such as efficient construction of the graphs (out-of-core [23], khmer [41], BOSS [7], Minia [13], kFM-index [44], DBGFM [11], BFT [18]), memory-efficient representations for the graph once constructed (SplitMEM [33], `BWT`-based algorithm [4], TwoPaCo [35], deGSM [16], Bifrost [17]), building space- and time-efficient indices on these representations (BFT [18], Pufferfish [2], BLight [32]), and performing read-alignment on the graph using the indices (deBGA [27], deSALT [28], PuffAligner [3]), to mention a few.

In this work, we tackle the initial step of the pipelines associated with whole genome and pan-genome analysis tasks based on de Bruijn graphs: the time and memory-efficient construction of colored compacted de Bruijn graphs. We present a novel algorithm, implemented in the tool Cuttlefish, for this purpose, that has excellent scaling behavior and low memory requirements. It exhibits better performance than state-of-the-art tools for constructing compacted colored de Bruijn graphs from reference collections. The algorithm models each distinct k-mer (i.e. vertex of the de Bruijn graph) of the input references as a finite-state automaton, and designs a compact hash table structure to store succinct encodings of the states of the automata. It characterizes the k-mers that flank maximal unitigs through an implicit traversal over the original graph — without building it explicitly — and dynamically updates the states of the automata with the local information obtained along the way. The maximal unitigs are then extracted through another implicit traversal, using the obtained characterization of the k-mers.

We assess our algorithm on both individual genomes, and collections of genomes with diverse structural characteristics: 7 closely related humans; 7 related but different apes; 11 related, different, huge conifer plants, and 100 humans. We compare our tool to two existing state-of-the-art algorithms, TwoPaCo and Bifrost. Cuttlefish is competitive with or faster than these tools under all settings of k-mer length and thread count, and uses less memory (often multiple *times* less) on all but the smallest datasets.

2 Related work

The amount of short read sequences produced from samples, like whole genome DNA, RNA, or metagenomic samples, can be in the order of billions. Construction of long-enough contigs [47] (contiguous sequences) from the sets of reads is known as the fragment assembly problem, and is a central and long-standing problem in computational biology. Many short-read fragment assembly algorithms (BCALM [11], BCALM2 [12], Bruno [39], deGSM [16]) use the de Bruijn graph to represent the input set of reads, and assemble fragments through graph compaction. More generally, fragment assembly algorithms are typically present as parts of larger and more complex genome assembly algorithms. In this work, we study the closely related problem of constructing and compacting de Bruijn graphs in the whole genome setting. While the computational requirements for constructing the compacted de Bruijn graph from genome references are typically substantial compared to the latter phases of sequence analysis (whole- and pan-genome) tasks, only a few tools have focused on the specific problem.

SplitMEM [33] exploits topological relationships between suffix trees and compacted de Bruijn graphs. It introduces a construct called *suffix skips*, which is a generalization of suffix links and is similar to pointer jumping techniques, to fast navigate suffix trees for identifying MEMs (maximal exact matches), and then transforms those into vertices and edges of the output graph. SplitMEM is improved upon using two algorithms [4], based on a compressed suffix tree, and the Burrows-Wheeler transform.

TwoPaCo [35] takes the approach of enumerating the edges of the original de Bruijn graph, which aids in identifying the *junction* positions in the genomes, i.e. positions that correspond to vertices in TwoPaCo's compacted graph format. Starting out having the entire set of vertices as junction candidates, it shrinks the candidates set using a Bloom filter [6] with a pass over the graph. Since the Bloom filter may contain false positive junctions, TwoPaCo makes another pass over the graph, this time keeping a hash table for the reduced set of candidates and thus filtering out the false positives.

deGSM [16] builds a Burrows-Wheeler transform [8] of the maximal unitigs without explicitly constructing the unitigs themselves. It partitions the $(k+2)$-mers of the input into a collection of buckets, and applies parallel external sorting on the buckets. Then it characterizes the k-mers at the ends of the maximal unitigs using the sorted information, and merges the sorted tables in a way to produce the unitigs-BWT.

Bifrost [17] initially builds an approximate de Bruijn graph using blocked Bloom filters. Then for each k-mer in the input, it extracts the maximal unitig that contains that k-mer, by extending the k-mer forward (and backward), constructing the suffix (and prefix) of the unitig. This produces an approximate compacted de Bruijn graph. Then using a k-mer counting and minimizer-based policy, it refines the extracted unitigs by removing the false positive k-mers present in the compacted graph.

3 Preliminaries

We consider all strings to be over the alphabet $\Sigma = \{A, C, G, T\}$. For some string s, $|s|$ denotes its length. $s[i..j]$ denotes the substring of s from the i'th to the j'th character, inclusive (with 1-based indexing). $\mathtt{suf}_\ell(s)$ and $\mathtt{pre}_\ell(s)$ denote the ℓ-length suffix and prefix of s, respectively. For two strings x and y such that $\mathtt{suf}_\ell(x) = \mathtt{pre}_\ell(y)$, the *glue* operation $\odot$ is defined as $x \odot^\ell y = x \cdot y[\ell+1..|y|]$, where $(\cdot)$ is the append operation.

A k-*mer* is a string of length k. For some string x, its *reverse complement* is the string obtained by reversing the order of the characters in x and complementing each character according to the nucleotide bases' complementarity, and is denoted with $\overline{x}$. The *canonical form* $\widehat{x}$ of a string x is defined as the string $\widehat{x} = \min(x, \overline{x})$, according to the lexicographic ordering.

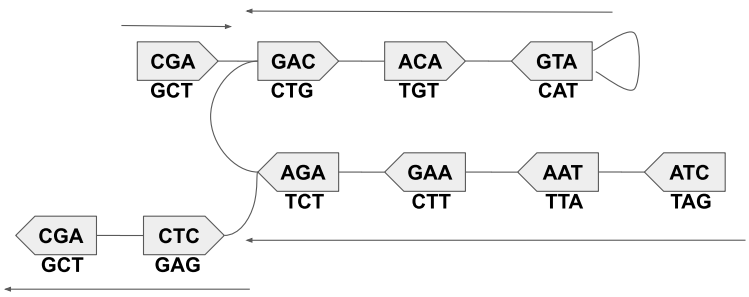
For a set S of strings and an integer $k > 0$, the corresponding *de Bruijn graph* $G(S, k)$ is defined as a directed graph with – (i) its set of vertices being exactly the set of k-mers of S; and (ii) two vertices u and v being connected with a directed edge $u \to v$ iff there exists some $(k+1)$-mer e in S such that $\mathtt{pre}_k(e) = u$ and $\mathtt{suf}_k(e) = v$. Note that, we adopt an *edge-centric* definition of the de Bruijn graph where an edge exists iff there is some corresponding $(k+1)$-mer present in S, as opposed to the *node-centric* definition where the edges are implicit given the vertices, i.e. an edge $u \to v$ exists iff $\mathtt{suf}_{k-1}(u) = \mathtt{pre}_{k-1}(v)$. A *walk* in the de Bruijn graph is a sequence of vertices $(v_0, \ldots, v_m)$ such that for any two successive vertices v_i and v_{i+1} in the sequence, there exists an edge $v_i \to v_{i+1}$ in the graph. A *path* is a walk with no repeated vertex. A path $p = (v_0, \ldots, v_m)$ is said to be a *unitig* if $|p| = 1$, or for all $0 < i < m$, the in- and out-degrees of v_i are 1, and the out-degree of v_0 and the in-degree of v_m are 1.

For our purposes, we use the bidirected variant of de Bruijn graphs. For the bidirected de Bruijn graph $G(S, k)$, the set of vertices is exactly the set of canonical k-mers obtained from S. A vertex v is said to have exactly two *sides*, referred to as the *front* side s_{vf} and the *back* side s_{vb}. An edge exists between two vertices u and v iff there is some $(k+1)$-mer e in S such that u and v are the canonical forms of the k-mers $\mathtt{pre}_k(e)$ and $\mathtt{suf}_k(e)$, respectively; i.e. $\widehat{\mathtt{pre}_k(e)} = u$ and $\widehat{\mathtt{suf}_k(e)} = v$. e is incident to exactly one side of u and one side of v. These incidence sides are determined using the following rule:

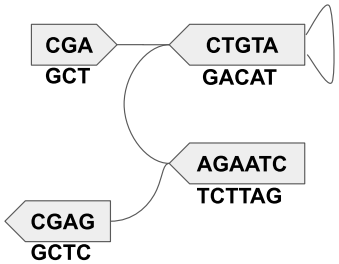
1. If $\mathtt{pre}_k(e) = u$, i.e. $\mathtt{pre}_k(e)$ is in its canonical form, then e is incident to the back side s_{ub} of u; otherwise it is incident to u's front side s_{uf}.
2. If $\mathtt{suf}_k(e) = v$, i.e. $\mathtt{suf}_k(e)$ is in its canonical form, then e is incident to the front side s_{vf} of v; otherwise it is incident to v's back side s_{vb}.

If $u = v$, then e is called a *loop*. So, an edge e can be defined as a 4-tuple (u, s_u, v, s_v), with s_u and s_v being the sides of the vertices u and v respectively to which e is incident to. The canonical k-mer $\widehat{x}$ corresponding to a vertex v is referred to as its *label*, i.e. $\mathtt{label}(v) = \widehat{x}$. An example illustration of a bidirected de Bruijn graph is given in Fig. 1a.

A walk $w = (v_0, e_1, v_1, \ldots, v_{m-1}, e_m, v_m)$ is defined as an alternating sequence of vertices and edges such that any two successive vertices v_{i-1} and v_i in the sequence are connected with the edge e_i (through any



(a) A bidirected de Bruijn graph for $S = \{CGACATGTCTTAG, GCTCTTAG\}$, with $k = 3$. The vertices are the *canonical* k-*mers* from S, and each edge corresponds to some 4-mer(s) in S. Each pentagon is a vertex, with the vertically flat and pointy sides denoting its *front* and *back*, respectively. For each vertex v, the string inside it is $\mathtt{label}(v)$, to be read in the visual direction from the front to the back. The string below it is $\overline{\mathtt{label}(v)}$, to be read in the opposite direction. E.g., the 4-mers CGAC and GCTC correspond to the edges $\{CGA, GAC\}$ and $\{AGC, CTC\}$, respectively. The edge corresponding to the 4-mer CATG is a *loop* $\{ATG, ATG\}$.



(b) The corresponding compacted de Bruijn graph, with each maximal unitig in its canonical form.

Fig. 1: For $G(S, k)$ in (1a), the sequence of vertices $w_1 = (GAC, ACA, ATG, ATG)$ forms a *walk* (the edges are not listed). The walk *spells* the string GACATG. It is not a *path* as the vertex ATG is repeated here. Whereas the walk $w_2 = (GAC, ACA, ATG)$ is a path, spelling GACAT. Besides, it is a *unitig*, and also a maximal one as it cannot be extended on either side while retaining to be a unitig. There are four *maximal unitigs* in the graph (the paths referred with the arrows), with the canonical spellings: CGA, ATGTC, CTAAGA, and GAGC.

side). v_0 and v_m are called the *endpoints*, and the v_i for $0 < i < m$ are called the *internal vertices* of the walk. w is said to *enter* v_i using e_i, and *exit* v_i using e_{i+1}. For the endpoints v_0 and v_m, w does not have an entering and an exiting edge, respectively. For $0 < i \leq m$, w is said to enter v_i through its front side $s_{v_i f}$ if e_i is incident to $s_{v_i f}$. Otherwise, it is said to enter v_i through its back side $s_{v_i b}$. For $0 \leq i < m$, the terminology for the exiting side is similarly defined using the incidence side of e_{i+1} to v_i.

w is said to be *input-consistent* if, for each of its internal vertices v_i, the sides of v_i to which e_i and e_{i+1} are incident are different. Intuitively, an input-consistent walk enters and exits a vertex through different sides. We prove in lemma 1 that the reconstruction (spelling) of the input strings $s \in S$ are possible only from input-consistent walks over $G(S, k)$. Therefore, we are only interested in such walks, and unless otherwise specified, we refer to input-consistent walks whenever using the term walk from now on.

For a vertex v_i in w, let its label be l_i, i.e. $l_i = \mathtt{label}(v_i)$. We say that w sees the *spelling* s_i for the vertex v_i (for $0 < i \leq m$), such that s_i is l_i if w enters v_i from the front, and is $\overline{l_i}$ otherwise. s_0 is defined analogously but using the exiting edge e_1 for v_0. The *spelling* of w is defined as the glued together spellings for all the vertices in the walk traversal, i.e. $s_0 \odot^{k-1} s_1 \odot^{k-1} \ldots s_m$. Fig. 1 illustrates examples of walks, paths, and their spellings in bidirected de Bruijn graphs.

A path $p = (v_0, e_1, v_1, \ldots, e_m, v_m)$ is said to be a *unitig* if $|p| = 1$, or in $G(S, k)$ –

1. each internal vertex has exactly two incident edges, e_i and e_{i+1};
2. and v_0 has exactly one edge e_1 and v_m has exactly one edge e_m incident to the sides respectively through which p exits v_0 and enters v_m.

A unitig is said to be maximal if it cannot be extended by a vertex on either side. Fig. 1 illustrates examples of maximal unitigs.

The *compacted de Bruijn graph* $G_c(S, k)$ of the graph $G(S, k)$ is obtained by collapsing each of its maximal unitigs into a single vertex. Fig. 1b shows the compacted form $G_c(S, k)$ of the graph $G(S, k)$ from Fig. 1a. The problem of compacting a de Bruijn graph is to compute the set of its maximal unitigs.

We use a slightly modified definition for the de Bruijn graph $G(S, k)$ in solving this problem. We refer a vertex v as a *sentinel* if the first or the last k-mer x of some input string $s \in S$ corresponds to it. For these vertices, we do not allow them to have any edge incident to the side that is empty at the sentinel-inducing occurrence of x. This is to ensure correct reconstruction-ability of the input strings in S from $G_c(S, k)$. For example, including a new string ACGA to S at Fig. 1a introduces the edge $\{ACG, CGA\}$. But we do not allow CGA to have an edge incident to its front side, since this side is empty at the sentinel-inducing occurrence of CGA as the first k-mer in the first input string.

4 Algorithm

4.1 Motivation

Given a set S of whole-genome references and an odd integer $k > 0$, a simple naive algorithm to construct the corresponding compacted de Bruijn graph $G_c(S, k)$ is to – first construct the ordinary de Bruijn graph $G(S, k)$, and then enumerate all the maximal non-branching paths from $G(S, k)$. The graph construction can be performed by a linear scan over the references in S and storing information along the way in some graph representation format, like an adjacency list. Enumeration of the non-branching paths can be obtained using a linear-time graph traversal algorithm [14]. However, storing the ordinary graph $G(S, k)$ usually takes an enormous space for large genomes, and also there is no simple way to traverse it without having the entire graph present in memory. So, the compacted graph $G_c(S, k)$ needs to be built directly from S without having to construct $G(S, k)$.

For some input reference s, it is important to note that by the definition of edge-centric de Bruijn graphs, each $(k + 1)$-mer in s is an edge of $G(s, k)$, and thus we can obtain a complete walk traversal over $G(s, k)$ through a linear scan over s, without having to build $G(s, k)$ explicitly. Also, each maximal unitig of the graph is contained as a subpath in this walk, as proven in lemma 2. For a set of references S, the maximal unitigs are similarly contained in a collection of walks $W(S)$.

Thus, the objective is to extract off the maximal unitigs from these walks $W(S)$, without building $G(S, k)$.

4.2 Flanking vertices of the maximal unitigs

Similar to the TwoPaCo [35] algorithm, our algorithm is based on the observation that there exists a bijection between the maximal unitigs of $G(S, k)$ and the substrings of the references in S whose terminal k-mers correspond to the endpoints of the maximal unitigs. We refer to these vertices as *flanking* vertices. The observation reduces the graph compaction problem to the problem of determining the set of the flanking vertices. Once each vertex in $G(S, k)$ can be characterized as either flanking or internal with respect to maximal unitigs, the unitigs themselves can then be enumerated using a walk over $G(S, k)$, by identifying subpaths having flanking k-mers at both ends.

Consider a maximal unitig p and one of its endpoints v. Say that v is connected to p at its side s_{vi}, and its other side is s_{vo}. v is a flanking vertex as it is not possible to extend p through s_{vo} while retaining itself a unitig, due to one of the following:

i. There are either zero or multiple edges incident to s_{vo};
ii. There is exactly one edge (v, s_{vo}, u, s_u) incident to s_{vo}, but s_u has multiple incident edges.

For non-trivial unitigs, the first scenario violates the first property of unitigs while the other one violates the second property.

From this, we can observe that the adjacency information of the sides of the vertices can be used to determine the flanking vertices. As per lemma 4, a side of a vertex can have at most four distinct edges. So one approach to keep the information for each side of each vertex is to have four binary markers for it, and another one to denote whether it is blocked off due to the vertex being a sentinel. So, each vertex needs ten binary markers in total. In one complete walk over the graph, we can fill up these markers accordingly. In another walk, the maximal unitigs can then be extracted off as per the flanking vertices, determined using the adjacency information.

4.3 Dealing with redundancy: a finite-state automaton model for vertices

As described, ten binary markers are sufficient to store the adjacency information for each vertex. These markers, as a whole, characterize a vertex as either flanking or internal with respect to maximal unitigs. Each vertex starts out with all the markers being empty. For each encountered edge (u, s_u, v, s_v), during the walks over the input references, the markers for the vertices u and v are updated accordingly. The sentinel markers for the two endpoints of each walk are also updated. Then using another set of walks over the graph, the maximal unitigs themselves can be extracted.

A source of redundancy in the information kept is the sides that have multiple incident edges, or are sentinels (so, blocked off). By definition, these sides belong to the flanking vertices of the maximal unitigs. So, at the first walk over the graph, once the side of a vertex has been determined to have more than one distinct edge, or to be a sentinel side, the adjacency information for the side becomes redundant: for our

purposes, we do not need to be able to enumerate the incident edges for this side, or differentiate between them. Only the information that this side makes the vertex flanking is necessary and sufficient. Whereas for each vertex side that is internal to the walk, there is exactly one incident edge; hence keeping track of that single edge is sufficient instead of having a set of markers.

Thus we observe that, the only required information to keep per side of a vertex is –

i. whether it has exactly one distinct edge, and if so, which edge it is among the four possibilities (as per lemma 4);
ii. or if there are multiple (or zero) distinct incident edges.

Hence, a side can have one of five different configurations: one for each possible singleton edge, and one for when it does not have exactly one distinct edge. This implies that in a de Bruijn graph $G(S, k)$, each vertex can be in one of $(5 \times 5) = 25$ different configurations. We refer to a configuration as a *state* for a vertex. Figuring out the states of the vertices provides us with enough information to extract the maximal unitigs off $G(S, k)$.

To compute the actual state of each vertex v in $G(S, k)$, we treat v as a deterministic finite-state automaton (DFA). Prior to traversing the graph, no information is available for v; we designate a special state for such, the *unvisited* state. Then, in response to each incident edge or sentinel k-mer visited for v, an appropriate state-transition is made, as necessary.

Formally, a vertex v is treated as a finite-state automaton $(Q, \Sigma', \delta, q_0, Q')$, where –

States: $Q = Q' \cup \{q_0\}$ is the set of all possible 26 states. The 25 actual states (configurations) of the vertices in $G(S, k)$ can be partitioned into four disjoint classes based on their properties, as in Fig. 2a.

Input symbols: $\Sigma' = \{ (c_1, c_2) \mid c_1, c_2 \in \Sigma \cup \{\epsilon\} \}$ [1] is the set of input symbols for the automaton. In the algorithm, we actually make state-transitions not per edge, but per vertex. That is, for some walk $w = (v_0, \ldots, e_i, v_i, e_{i+1}, \ldots, v_m)$, we process the vertices v_i. Processing v_i means checking the edges e_i and e_{i+1} simultaneously and only for v_i (excluding the edges' other endpoints); making state transitions for v_i's automaton as required. [2] Thus, the two incident edges $\{e_i, e_{i+1}\}$ are being used as input for the automaton. The edges can be represented succinctly with a pair of characters (c_1, c_2) for v_i. c_1 and c_2 encode the edges incident to v_i's front and back respectively, in the subwalk (e_i, v_i, e_{i+1}). [3]

Transition function: $\delta : Q \times \Sigma' \to Q$ is the function governing the state-transitions. Fig. 2b presents a high-level view of the possible types of transitions between states of the four classes; and Fig. 3 illustrates a detailed overview of the transitions defined for the states as per various input symbols.

Initial state: q_0 is the initial state for the automaton, which is the unvisited state.

Accept states: Q' is the set of the possible 25 different configurations (i.e. states) for the vertices. [4]

4.4 Hash table for the vertices

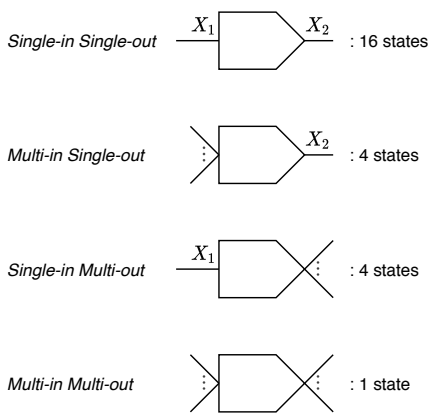
To maintain the (transitioning) states for the vertices in $G(S, k)$ throughout the algorithm, we need some associative data structure for the vertices (i.e. canonical k-mers) and their states. We design a hash table for this purpose, with a *(k-mer, state)* key-value pairing. An efficient representation of hash tables for k-mers is a significant data structure problem in its own right, and some attempts even relate back to the compacted de Bruijn graph [32]. In solving the subproblem efficiently for our case, we exploit the fact that the set K of the keys (i.e., canonical k-mers) are static here, and it can be built prior to the states classification task. We

[1] ϵ denotes the empty character, used for sentinel-sides.

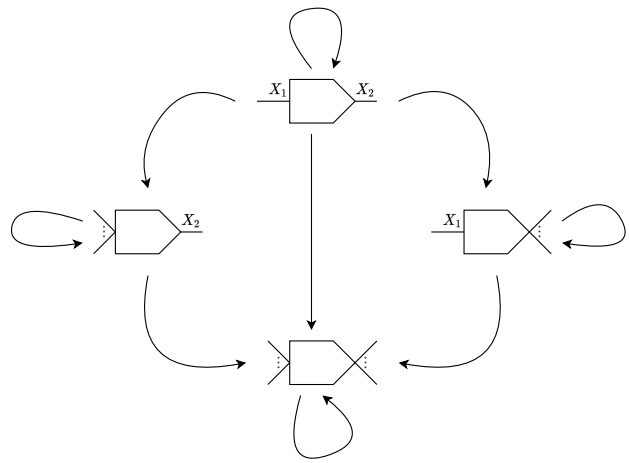
[2] This shrinks the state-space for the automata. For the subwalk (e_i, v_i, e_{i+1}), if the edges e_i and e_{i+1} are to be processed independent of each other, then during each processing, only one side of v_i can be seen. This requires each side to have an unvisited state independently, making the state-space size $(6 \times 6) = 36$. From lemma 1, the edges e_i and e_{i+1} are incident to different sides of v_i. Processing them simultaneously for v_i thus ensures that both the sides are seen together, making one state sufficient to denote the unvisited status' of both the sides together. This reduces the state-space size to $(5 \times 5 + 1) = 26$.

[3] c_1 and c_2 does not necessarily correspond to e_i and e_{i+1}, in this order; the order can be the opposite depending on the side of entrance of w to v_i.

[4] Formally, this parameter denotes the set of states reachable from the initial state under certain patterns of the input symbols. As our purpose is not the acceptance of any specific input patterns, rather just to compute the final state of each automaton, we define the accept states as the entire set of possible final states.



(a) Four disjoint classes of states of the vertices, based on the properties of their sides. The pictorial shape of the classes correspond to the actual incidence properties of the vertices. For example, the first class is for vertices that have exactly one edge incident per side: the edges incident to the front and the back being encoded with the characters X_1 and X_2, respectively. There can be $(4 \times 4) = 16$ different configurations of this shape, and this class contains those 16 states. Whereas the second class is for vertices that have $\neq 1$ distinct edges incident to the front, and one unique edge incident to the back. Due to 4 different configurations with this property, this class contains 4 states.
Note that, pictorially, a singular incident edge denotes a unique edge, whereas multiple incident edges mean either > 1 or 0 edges being incident.



(b) Possible transition types between the various classes of the states. For example, consider a state of the class *single-in single-out*, with the unique edges incident to its front and back being encoded with the characters X_1 and X_2, respectively. Now, if the state is provided with the input (Y_1, Y_2), then based on the four different joint outcomes of the conditionals $(X_1 = Y_1)$ and $(X_2 = Y_2)$, the following transitions can happen:

1. $(Y_1 = X_1) \wedge (Y_2 = X_2)$: self-transition;
2. $(Y_1 \neq X_1) \wedge (Y_2 = X_2)$: transition to a state of the class *multi-in single-out* that has X_2 at the back;
3. $(Y_1 = X_1) \wedge (Y_2 \neq X_2)$: transition to a state of the class *single-in multi-out* that has X_1 at the front;
4. $(Y_1 \neq X_1) \wedge (Y_2 \neq X_2)$: transition to the only state of the class *multi-in multi-out*.

Fig. 2: Classes of states of the vertices, and the transition relationships among those.

build the set K efficiently from the provided set S of references using the k-mer counting algorithm `KMC3` [22]. Afterwards we construct a minimal perfect hash function (MPHF) h over the set K, employing the `BBHash` algorithm [26].

A perfect hash function h_p for a set K of keys is an injective function from K to the the set of integers, i.e. for $x_1, x_2 \in K$, if $x_1 \neq x_2$, then $h_p(x_1) \neq h_p(x_2)$. Perfect hash functions guarantee that no collisions are made while hashing the keys in K. A perfect hash function is minimal if it maps K to the set $[0, |K|)$. Since we do not need to support lookup of k-mers nonexistent in the references in S, i.e. no alien keys will be queried, an MPHF works correctly in this case. Also, as an MPHF produces no collisions, we do not need to store the keys along with the hash table structure for collision resolution – thus reducing the space requirement of the structure. Although we need to store the function itself with the hash table instead of the keys, the function takes much less space than the set of keys. In our setting, the function constructed from the `BBHash` [26] algorithm takes ≈ 3.7 bits/k-mer, irrespective of the size k. Whereas storing the keys would take $2k$ bits/k-mer.

As the hash value $h(x)$ for some key $x \in K$ is at most $(|K|-1)$, we use a linear array of size $|K|$, indexed by $h(x)$, to store the state of the vertices (k-mers) x as the hash table values. We call this array the buckets table B. So, our hash table structure consists of two components: an MPHF h, and an array B. For a canonical k-mer $\widehat{x}$, its associated bucket is at the $h(\widehat{x})$'th index of B. This bucket $B_{h(\widehat{x})}$ stores the state of $\widehat{x}$ throughout the algorithm.

4.5 Computing the states of the vertices

Given a walk $w(s)$ over $G(S, k)$ spelling a string $s \in S$, a k-mer x in s, an MPHF h over the canonical k-mers of S, and a buckets table B, the PROCESS-k-MER(x, s, h, B) algorithm checks the two incident edges of x in the walk, as well as the state of $\widehat{x}$ from B using h; making appropriate state-transitions for $\widehat{x}$ if required. For brevity, handling of the sentinel sides are not shown in the algorithm, but the extension is straightforward.

For a set of of strings S with n distinct canonical k-mers and an MPHF h over those, the CLASSIFY-VERTICES(S, h, n) algorithm computes the final state of each vertex of $G(S, k)$, performing all the required

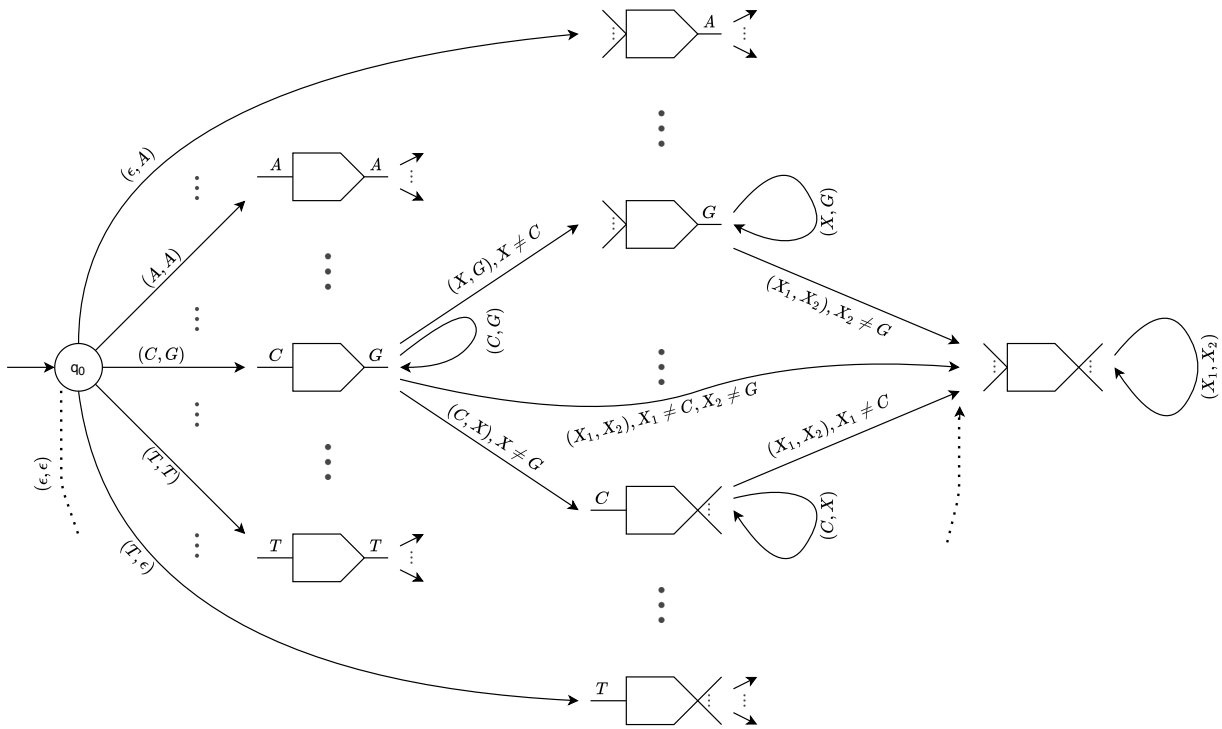


Fig. 3: Partial state-transition diagram for the automaton of a vertex. q_0 is the initial state, denoting the *unvisited* state. The rest 25 states are represented with the pictorial shapes as described in Fig. 2a, and we present some of the states as representatives. The ϵ characters in the input symbols denote sentinel occurrences in the corresponding sides.
For example, in a complete walk over $G(S, k)$, say an unvisited vertex v_i is encountered for the first time in the subwalk $(.., e_i, v_i, e_{i+1}, ..)$, and the edges incident to the front and to the back of v_i among e_i and e_{i+1} are encoded with the characters C and G, respectively. Then v_i transitions from the q_0 state to a state of the class *single-in single-out* with the (C, G) configuration. Then when v_i is encountered next, its state is transitioned as per the rules described in Fig. 2b. Say this next occurrence of v_i has edges encoded by A and G at its front and back, respectively. Then v_i transitions to a state of the class *multi-in single-out* with the configuration (X, G). The X here means that there are $\neq 1$ distinct edges incident to the front, and as such we do not care about what the actual edges are – thus, we are throwing away the information of the existence of the C-encoded edge at the front of v_i. Then as long as the next occurrences of v_i in the walk has G-encoded edges at its back, irrespective of the front-incident edges, v_i remains in this state. Whenever a back-incident edge is encountered with any of the other three encodings (or ϵ, for a sentinel), v_i transitions to the only state of the class *multi-in multi-out*, which is a dead-end for states.

state-transitions. Before traversing over $G(S, k)$, it initializes the buckets table B with the state *unvisited* for each vertex. Then for each $s \in S$, it initiates a walk $w(s)$ on $G(S, k)$, spelling s. For each k-mer x encountered through $w(s)$, the PROCESS-k-MER(x, s, h, B) algorithm is executed.

4.6 Extracting the maximal unitigs

Having computed the states of the vertices in $G(S, k)$, extraction of the maximal unitigs can be done through another complete traversal over the graph. A walk is initiated from the starting k-mer of each input reference, i.e., from the corresponding sentinel vertex. In the walk, a subpath continues to be extracted as long as the exiting side for the walk and the next side, both have exactly one incident edge. The subpath is ended when the condition fails, producing a maximal unitig. Another extraction is started from the next vertex, and the process is continued throughout the walk.

For a walk $w = (v_0, e_1, v_1, \ldots, e_i, v_i, e_{i+1}, \ldots, e_m, v_m)$ over $G(S, k)$ spelling some $s \in S$, say w enters v_i through its side s_i, and the class of the state of v_i is c_i. Then,

```
PROCESS-k-MER(x, s, h, B)
1  x̂ ← CANONICAL(x)
2  st ← B_{h(x̂)}
3  if st == STATE(multi-in-multi-out)
4      return

5  entrance ← FRONT if x == x̂; BACK otherwise
6  e_front ← PREV(x, s) if entrance is FRONT,
           NEXT(x, s) otherwise
7  e_back ← NEXT(x, s) if entrance is FRONT,
           PREV(x, s) otherwise
   // PREV(x, s) and NEXT(x, s) is the previous
      and the next character of x in s

8  if st == unvisited
9      st ← STATE(single-in-single-out, e_front, e_back)
10 else
11     if st == STATE(single-in-single-out, c_f, c_b)
12         transition st jointly based on
           (e_front == c_f) and (e_back == c_b)
13     elseif st == STATE(multi-in-single-out, c_b)
14         transition st based on e_back == c_b
15     else // st is STATE(single-in-multi-out, c_f)
16         transition st based on e_front == c_f

17 B_{h(x̂)} ← st
```

```
CLASSIFY-VERTICES(S, h, n)
1  B ← buckets table with n entries,
       each having the unvisited state

2  for each s ∈ S
3      for each k-mer x in s
4          PROCESS-k-MER(x, s, h, B)
```

```
EXTRACT-MAXIMAL-UNITIGS(s, h, B)
1  start ← ϕ, end ← ϕ
2  Let w be the walk spelling s

3  for each k-mer x in s
4      if IS-UNIPATH-START(x, w)
5          start ← x
6      if IS-UNIPATH-END(x, w)
7          end ← x
8          EXTRACT-UNITIG(s, start, end)
```

v_i initiates a maximal unitig traversal in w if –

1. c_i = *multi-in multi-out*; or
2. s_i is the front of v_i and c_i = *multi-in single-out*; or
3. s_i is the back of v_i and c_i = *single-in multi-out*; or
4. v_{i-1} terminates a maximal unitig traversal in w.

v_i terminates a maximal unitig traversal in w if –

1. c_i = *multi-in multi-out*; or
2. s_i is the front of v_i and c_i = *single-in multi-out*; or
3. s_i is the back of v_i and c_i = *multi-in single-out*; or
4. v_{i+1} initiates a maximal unitig traversal in w.

For a walk $w(s)$ over $G(S, k)$ spelling the string $s \in S$, an MPHF h over the canonical k-mers of S, and a buckets table b containing the actual states of each vertex in $G(S, k)$, the algorithm EXTRACT-MAXIMAL-UNITIGS(s, h, B) computes all the maximal unitigs present in $w(s)$. Procedures IS-UNIPATH-START(x, w) and IS-UNIPATH-END(x, w) checks whether the vertex for the k-mer x in the walk w initiates or terminates a maximal unitig in it, respectively; as per the conditions delineated here. Procedure EXTRACT-UNITIG(s, x, y) extracts the unitig (substring) flanked by the k-mers x and y off the string s, formatted as per the end user requirement.

4.7 Complete algorithm

For a set of input references S, the complete algorithm, CUTTLEFISH(S) is presented as follows.

```
CUTTLEFISH(S)
1  K ← EXTRACT-UNIQUE-k-MERS(S)
2  h ← CONSTRUCT-MINIMAL-PERFECT-HASH-FUNCTION(K)
3  B ← CLASSIFY-VERTICES(S, h, |K|)
4  for each s ∈ S
5      EXTRACT-MAXIMAL-UNITIGS(s, h, B)
```

4.8 Parallelization

The Cuttlefish algorithm is designed to be efficiently parallelizable on a shared memory multi-core machine. The first step, i.e. the generation of the set K of k-mers is parallelized in the `KMC3` [22] algorithm itself. The two major steps of it: splitting the collection of k-mers from the input references into different bins based on super k-mer signatures, and then sorting and compacting the bins, are highly parallelized.

The next step of constructing the MPHF with the `BBHash` [26] algorithm is also parallelized by partitioning its input key set K to multiple threads; distributing subsets of keys with some threshold size to the threads as soon as it is read off disk.

The next step of classifying the states of the vertices are parallelized as follows. For each input reference $s \in S$, s is partitioned into a number of uniform sized chunks. Each chunk is provided to an available thread, and each thread is responsible to process the k-mers having their starting indices in its chunk. Although the threads process disjoint sets of k-mer instances, they query and update the same entry in the hash table structure for identical canonical k-mers. Accessing the MPHF h concurrently from multiple threads is safe, as the accesses are read-only. Whereas accessing a hash table entry itself, i.e. an entry into the buckets table B is not, since all the threads are accessing (read / write) the same table B. We ensure that only a single thread can probe and / or update some specific hash table entry at any given time through maintaining a sparse collection of locks, which are to be acquired to access ranges of indices of B. As we discuss at section 5.3, we ensure low memory usage by using a bit-packed table for B. Even with locked accesses to index ranges of B, multiple threads can access the same underlying memory-word concurrently while accessing nearby indices, producing data races. We use a thread-safe bit-packed vector data structure [31] for B that ensures that such accesses are handled safely using compare-and-swap atomic operations.

The last step of extracting the maximal unitigs is parallelized similarly, by distributing disjoint chunks of an input reference s to different threads, where each thread is responsible to extract each maximal unitig that has its starting k-mer (in the traversal of s) in its chunk. Each maximal unitig is assigned a unique k-mer as its signature, and unique extraction of a maximal unitig is ensured by transitioning its signature k-mer to some special "output"-tagged state of the same state-class when the unitig is encountered first.

5 Proofs and Asymptotics

5.1 Proofs

Lemma 1. *For a string* s *and an odd integer* $k > 0$*,* s *can only be spelled by a walk* w *in* $G(s,k)$ *that enters and exits each vertex* v_i *through different sides of it.*

Proof. For $G(s,k)$, consider a walk $w = (v_0, e_1, v_1, \ldots, e_i, v_i, e_{i+1}, \ldots, e_m, v_m)$ that spells s. Assume that w is not input-consistent and e_i and e_{i+1} are incident to the same side of v_i. Without loss of generality, say that w enters v_i through its front side $s_{v_i f}$. Then it exits v_i using the same side. There exists some $(k+1)$-mers x and y in s that correspond to the edges e_i and e_{i+1}. Since w spells s completely, w would spell $pre_k(x)$ from v_{i-1}, $suf_k(x)$ (or $pre_k(y)$) from v_i, and $suf_k(y)$ from v_{i+1}. So, x and y overlap at v_i with their suffix and prefix k-mers respectively, i.e. $suf_k(x) = pre_k(y)$ holds.

Since w enters v_i through the front with e_i (i.e. x), from the definition of incidence sides of edges, $suf_k(x) = label(v_i)$. And as w exits v_i through the front with e_{i+1} (i.e. y), $\overline{pre_k(y)} = label(v_i)$. So we get, $suf_k(x) = \overline{pre_k(y)}$. Combining with $suf_k(x) = pre_k(y)$, we get $pre_k(y) = \overline{pre_k(y)}$. This implies that $y_1 = \overline{y_k}, \ldots, y_i = \overline{y_{k-i+1}}, \ldots, y_k = \overline{y_1}$. As k is odd, the ${}^{(k+1)}/_2$'th character exists in y, and for $i = {}^{(k+1)}/_2$, $y_i = \overline{y_i}$ holds. But a character cannot be its own nucleotide complement, resulting into a contradiction.

Therefore, the initial assumption of w not being input-consistent is false. So, e_i and e_{i+1} must be incident to different sides of v_i for w to spell s.

Lemma 2. *For a string* s *and an integer* $k > 0$*, a complete walk traversal* $w(s)$ *over* $G(s,k)$ *contains each maximal unitig of* $G(s,k)$ *as some subpath.*

Proof. Consider a maximal unitig $p = (v_0, e_1, v_1, \ldots, e_m, v_m)$ in $G(s,k)$. Without loss of generality, we will prove that $w(s)$ contains p in the orientation from v_0 through v_m. A complete walk traversal $w(s)$ over $G(s,k)$ is obtained through a scan over s; and thus from the definition of edge-centric de Bruijn graphs, $w(s)$ traverses all the vertices and edges of the graph. Therefore, it must traverse v_0 and then e_1 at some point. Since p is a unitig, each of its internal vertices $v_i (0 < i < m)$ has degree 1 on each side – thus p must

be traversed entirely by $w(s)$ after it reaches e_1, as it is not possible for $w(s)$ to branch off from some v_i without exhausting p. Thus, each maximal unitig p is contained in $w(s)$.

Lemma 3. *For some string* s *and an integer* $k > 0$*, the* $(k+1)$*-mers* e *and* $\overline{e}$ *correspond to the same edge in* $G(s,k)$*.*

Proof. Consider a $(k+1)$-mer e in $G(s,k)$. e can be expressed as $e = x \cdot n_x = n_y \cdot y$, where $x = pre_k(e)$, $y = suf_k(e)$, $n_x = e[k+1]$, and $n_y = e[1]$.

As $e = x \odot y$, e induces an edge between the vertices corresponding to $\widehat{x}$ and $\widehat{y}$. It is incident to the back of $\widehat{x}$ if $x = \widehat{x}$ holds, and is incident to the front of $\widehat{x}$ if $\overline{x} = \widehat{x}$ holds otherwise.

e's reverse complement is $\overline{e} = \overline{n_x} \cdot \overline{x} = \overline{y} \cdot \overline{n_y}$. As $\overline{e} = \overline{y} \odot \overline{x}$, it induces an edge between the same pair of vertices $\widehat{y}$ and $\widehat{x}$. Note the order of the k-mers in $\overline{e}$. The edge is incident to the front of $\widehat{x}$ if $\overline{x} = \widehat{x}$ holds, and to the back of $\widehat{x}$ if $x = \widehat{x}$ holds otherwise.

So, e and $\overline{e}$ induce edges between the same pair of vertices $\widehat{x}$ and $\widehat{y}$, and both the edges are incident to the same side of $\widehat{x}$. Similarly, it can be shown that both the edges are incident to the same side of $\widehat{y}$. Thus, e and $\overline{e}$ induce the same edge in $G(s,k)$.

Lemma 4. *For some string* s *and an integer* $k > 0$*, a side of a vertex in* $G(s,k)$ *can have at most four distinct incident edges.*

Proof. Without loss of generality, consider the back side s_{vb} of a vertex v in $G(s,k)$. From the definition of incidence sides of edges, we get that s_{vb} will have an incident edge for a $(k+1)$-mer e iff – i. $pre_k(e) = label(v)$; or ii. $\overline{suf_k(e)} = label(v)$, which is the same as $suf_k(e) = \overline{label(v)}$.

Say that $label(v) = l_v$. Recall that all our strings are over the alphabet $\Sigma = \{A, C, G, T\}$. For the first case, there can be at most $|\Sigma| = 4$ distinct edges, the corresponding set of $(k+1)$-mers being $E_1 = \{l_v \cdot c \mid c \in \Sigma\}$. Similarly, there can be at most four distinct edges for the second case, with the set of $(k+1)$-mers being $E_2 = \{c \cdot \overline{l_v} \mid c \in \Sigma\}$.

From lemma 3, the $(k+1)$-mers $l_v \cdot c$ and $\overline{c} \cdot \overline{l_v}$ induce the same edge for some $c \in \Sigma$. This implies that the set of edges induced from E_1 is the same as the set induced from E_2. Thus, a vertex can have at most four distinct edges incident to each side.

5.2 Running time

Let m be the total length of the input strings, and n be the number of distinct k-mers in the input. While the exact asymptotic complexity for the initial phase of distinct k-mers enumeration using `KMC3` is somewhat difficult to pin down precisely, we provide a rough upper-bound for it. It consists of two major steps: bucketing the k-mers based on super k-mer signatures, and then radix sorting and compacting the buckets. The internal representation of k-mers make it possible to compare them using 64-bit machine words, i.e. 32 nucleotide-bases at a time. Splitting the k-mers collection takes time $O\big(\lceil k/32 \rceil m\big)$ (using a constant signature length), and radix sorting the buckets take total time $O\big(\sum_{i=1}^{b} \lceil k/32 \rceil B_i\big) = O\big(\lceil k/32 \rceil m\big)$, where b is the number of buckets, and B_i is the size of the i'th bucket. So a loose upper bound is $O\big(\lceil k/32 \rceil m\big)$.

CUTTLEFISH stores a k-mer using $\lceil k/32 \rceil$ machine words (of 64-bit). To use the MPHF, we convert a k-mer to a 64-bit signature, taking $O\big(\lceil k/32 \rceil\big)$ time, and use this as the key to hash. Assuming that the `BBHash` algorithm takes $O(h)$ time (an expected constant) to hash each word, the time to hash a k-mer is then $H(k) = O\big(\lceil k/32 \rceil + h\big)$. `BBHash` construction algorithm requires an expected number of hashing operations linear in the number of keys n. So an expected bound for the construction time is $O\big(n\, H(k)\big)$.

Querying and filling the buckets table at the vertices classification task take time $O\big(m\, H(k)\big)$. For the next phase of the unitigs extraction, the time complexity is asymptotically the same. Therefore, the (expected) total running time of CUTTLEFISH is $O\big(\lceil k/32 \rceil m + n\, H(k) + m\, H(k)\big) = O\big((m+n)\, H(k)\big)$.

The dependence of the running time on both the input size (m) and the variation in the input (expressed through the number of distinct k-mers n) is exhibited for an apes dataset, detailed in subsection 6.5 and illustrated in Figs. 5a and 5c. The dependence on k is discussed at subsection 6.4, with benchmarking in Table 3.

5.3 Memory usage

The k-mer set construction by KMC3 [22] is done by splitting the input k-mers collection into bins, and then sorting and compacting the bins. This binning-based procedure lends effectively to operate under strict memory bounds – the bounds can be set as deemed feasible for the working platforms.

The MPHF construction is a multi-step algorithm, assigning final hash values to a subset of keys at each step and building a bit-array H progressively as the output data structure. The entire k-mer set K need not be present in memory, rather the algorithm processes K chunk-by-chunk in each step. In the last step, the total array H is present in memory; thus the algorithm requires $\Omega(|\mathsf{H}|)$ memory. The algorithm has a parameter γ that trades-off the construction and the query time with $|\mathsf{H}|$, and we set it as $\gamma = 2$. This makes $|\mathsf{H}|$ use ~3.7 bits/k-mer. Each step d also requires an additional bit-array $\mathsf{C_d}$ for the keys yet to be assigned their final hash values. If $\mathsf{R_d}$ is the set of these keys, then $|\mathsf{C_d}| = \gamma|\mathsf{R_d}|$. $\mathsf{R_0} = \mathsf{K}$, and $\mathsf{R_d}$ shrinks whereas H expands in size with each step. Thus, the memory usage is loosely bounded by $\mathsf{O}(|\mathsf{H}| + \gamma|\mathsf{K}|)$. Specifically in our setting, the memory usage of the construction can be at most $(3.7 + 2) = 5.7$ bits/k-mer.

Having constructed the hash function, the buckets table B is allocated as a bit-packed array. Each vertex (canonical k-mers) can be in one of 26 different states (including *unvisited*). At least $\lceil \log_2 26 \rceil = 5$ bits are necessary to represent such a state. Thus, the buckets consume $5|\mathsf{K}|$ bits in total. Therefore, the total memory usage of the algorithm is $(8.7 \times |\mathsf{K}|) = \mathsf{O}(|\mathsf{K}|)$ bits; translating to roughly a byte per distinct k-mer. This linear relationship between the memory usage and the distinct k-mers count is illustrated for a humans and an apes dataset at Figs. 5b and 5d.

6 Results

We evaluated the performance of Cuttlefish compared to other state-of-the-art tools for compacting de Bruijn graphs from whole genome references. We also assessed its scaling properties, and effects of the input structure on performance. Experiments are performed on a server with an Intel Xeon CPU (E5-2699 v4) with 44 cores and clocked at 2.20 GHz, 512 GB of memory, and a 3.6 TB Toshiba MG03ACA4 HDD.

6.1 Dataset characteristics

We use a varied collection of datasets to benchmark Cuttlefish in evaluating its performance on diverse input characteristics. First, we assess its performance on single input genomes. We use individual references of (i) a human (*Homo sapiens*, ~3.2 Gb), (ii) a western gorilla (*Gorilla gorilla*, ~3 Gb), and (iii) a sugar pine (*Pinus lambertiana*, ~27.6 Gb). We also evaluate its performance in building colored compacted de Bruijn graphs, i.e. for collections of references as inputs. We use a number of genome collections exhibiting diverse characteristics: (i) 62 E. Coli (*Escherichia coli*) references (~310 Mb), a dataset with small bacterial genomes; (ii) 7 Human (*Homo sapiens*) references (~21 Gb), very closely related moderate-sized mammalian genomes from the same species; (iii) 7 Ape (*Hominoid*) references (~18 Gb), related moderate-sized mammalian genomes from the same order (*Primate*) and superfamily, but different species; and (iv) 11 Conifer (*Pinophyta*) references (~204 Gb), related huge plant genomes from the same order (*Pinales*), but different species.

The E. Coli dataset with 62 strains (available at NCBI) was used in benchmarking SplitMEM [33]. The human dataset was used in benchmarking a BWT-based algorithm [4] for compacted de Bruijn graph construction, and includes 5 different assemblies of the human genome (from the UCSC Genome Browser), as well as the maternal and paternal haplotype of an individual NA12878 (Utah female), from the 1000 Genomes Project. The ape dataset includes 7 available references from the orangutan and the chimp genera, a western gorilla, a human, and a bonobo. The conifer dataset consists of 9 references belonging to the pine (*Pinaceae*) family: from the pine, the spruce, and the larch genera, and a douglas fir; and 2 references from the redwoods (*Sequoioideae*) subfamily. We assembled the ape and the conifer datasets from the GenBank [45] database of NCBI.

We also assessed Cuttlefish's performance on compacting a huge number of closely related genomes. For such, we used the 93 human references, generated using the FIGG genome simulator [21], that was used to benchmark TwoPaCo [35]. Coupled with the previous 7 human genomes, this gives us a dataset of 100 human references (~322 Gb).

6.2 Benchmarking comparisons

We benchmarked CUTTLEFISH against two other implementations of de Bruijn graph compaction algorithms: TWOPACO [35] and BIFROST [17]. While TWOPACO, like CUTTLEFISH, is specialized for construction of the compacted de Bruijn graph from reference sequences, BIFROST is also capable of constructing the compacted graph from short read sequences as well. Here we compare against BIFROST using the appropriate settings for reference-based construction. Finally, we note that BIFROST builds an index over the compacted de Bruijn graph during (and as part of) the construction process. TWOPACO, on the other hand, builds the compacted de Bruijn graph without indexing the underlying graph in any way. CUTTLEFISH does not intend, itself, to provide an index over the compacted de Bruijn graph, however it does construct a minimum perfect hash function over all k-mers of the graph, and therefore could be well-suited to subsequent indexing of the graph.

Both TWOPACO and BIFROST are multi-threaded, and are run without any memory restrictions. All the stats reported for each tool are produced with a warm cache. The runtimes and peak resident memory usage numbers are obtained using the GNU `time` command. Tables 1 and 2 contain a summary of the comparison results. We do not compare against compaction algorithms applicable only for short read sequences. These tools usually filter k-mers based on certain conditions and compacts node-centric de Bruijn graphs. Thus such comparisons are not like-for-like, per se.

Table 1: Time- and memory-performance benchmarking for compacting single input reference de Bruijn graphs.

Dataset	Thread-count	k	BIFROST		TWOPACO		CUTTLEFISH	
			Build	Output	Build	Output	Build	Output
Human	1	31	04:54:50 (27.23)	15:18	01:02:40 (4.32)	43:00 (4.50)	37:19 (2.79)	19:23 (2.84)
		61	05:16:51 (50.19)	01:49	01:35:58 (5.31)	14:16 (4.36)	37:17 (3.06)	15:37 (3.08)
	8	31	01:33:54 (27.23)	03:59	09:59 (4.38)	–	07:30 (2.79)	05:13 (2.92)
		61	01:20:28 (50.18)	00:40	09:42 (5.45)	–	10:03 (3.06)	03:20 (3.18)
	16	31	01:24:40 (27.24)	03:30	06:35 (4.80)	–	05:30 (2.79)	02:57 (2.93)
		61	01:12:33 (50.18)	00:52	06:14 (5.50)	–	07:56 (3.06)	01:54 (3.19)
Gorilla	1	31	07:26:24 (57.91)	20:46	01:29:57 (4.04)	01:00:55 (4.49)	31:26 (2.74)	17:07 (2.77)
		61	05:31:06 (50.13)	02:05	01:02:27 (5.26)	00:17:31 (4.30)	38:31 (3.02)	15:59 (3.03)
	8	31	03:04:15 (57.91)	03:31	11:28 (4.36)	–	06:33 (2.74)	04:37 (2.87)
		61	01:24:21 (50.13)	00:54	10:10 (6.04)	–	09:12 (3.02)	02:54 (3.12)
	16	31	02:14:13 (57.91)	02:38	07:13 (4.84)	–	04:52 (2.74)	03:25 (2.87)
		61	01:10:06 (50.13)	04:04	06:47 (5.51)	–	07:41 (3.02)	02:06 (3.14)
Sugar pine	16	31	X (293.23)	X	08:41:51 (43.63)	–	01:00:01 (14.24)	01:56:52 (14.28)
		61	X (364.25)	X	06:55:14 (64.86)	–	03:46:05 (20.88)	01:26:26 (20.90)

Each cell contains the running time in wall clock format, and the maximum memory usage in gigabytes, in parentheses. The output steps used for the algorithms are employed to report the compacted graph in the GFA2 format.
BIFROST builds the compacted graph and outputs it using the same command; we could split the timing of the steps but were unable to tease apart the maximum memory for the output step. TWOPACO takes a logarithmic filter-size parameter f as input, and f is critical to the performance. It uses $(2^f/8)$ bytes of memory for a bloom filter in the first-pass, which significantly affects the memory usage in the second-pass. For both human and gorilla, the statistics reported are for $f = 35$ in both $k = 31$ and $k = 61$; for sugar pine, $f = 38$ for $k = 31$ and $f = 39$ for $k = 61$. We have set f such that the maximum memory usage is minimized, by first approximating its optimal value, and then trying a few of the nearby values. The best statistics found are reported. Also, the output step of TWOPACO is single-threaded, and the dashes in their output column indicate this inapplicability of multi-threading. The cells with X in the BIFROST columns indicate that BIFROST execution terminated with a segmentation fault (for $k = 31$) and `std::bad_alloc` (for $k = 61$), and the memory usage in parentheses is the peak resident memory usage until the point of program termination.

We verified the correctness of the extracted unitigs by designing a validator, that confirms the following – (i) the unique existence of each k-mer in the output unitigs collection (the set of maximal unitigs for a de Bruijn graph is unique and forms a node decomposition of the graph [12]); and (ii) the complete spellingability of the set of input references by the unitigs (i.e. each input reference is correctly constructible from the compacted graph).

Table 2: Time- and memory-performance benchmarking for compacting colored de Bruijn graphs (i.e. multiple input references) for k = 31, using 16 threads.

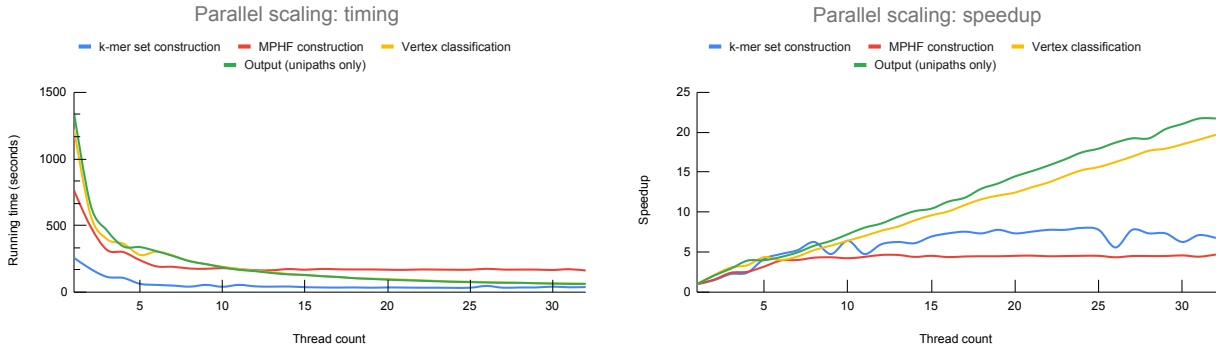
Dataset	Total genome-length (bp)	Distinct k-mers count	Bifrost	TwoPaCo	Cuttlefish
62 E. Coli	310M	24M	1 (0.66)	1 (0.76)	1 (0.96)
7 Humans	21G	2.6B	125 (58.69)	37 (4.77)	23 (2.88)
7 Apes	18G	7.1B	404 (195.03)	47 (21.44)	34 (7.42)
11 Conifers	204G	82B	–	981 (288.99)	646 (84.12)
100 Humans	322G	28B	–	X (64.88)	473 (28.75)

Each cell contains the running time in minutes, and the maximum memory usage in gigabytes, in parentheses. The output step is excluded from executions. The filter-sizes for the TwoPaCo executions are set as described in Table 1. Empty cells in the Bifrost column indicate that the experiments were not performed, as it is anticipated that insufficient memory would be available given that the program terminated while exhausting memory for smaller datasets. The X cell for TwoPaCo indicates that it terminated with a segmentation fault.

A straightforward correspondence between the outputs of the different tools is not easy. Bifrost compacts the node-centric de Bruijn graph of the references, whereas both TwoPaCo and Cuttlefish use the edge-centric variant. Also, there are two notable non-trivial difference between the graphs compacted by TwoPaCo and Cuttlefish. For TwoPaCo, each *junction* k-mer is present in all the maximal unitigs that are incident to it, so the maximal unitigs do not form a node (k-mer) decomposition of the original graph. Another difference is in the handling of inverted repeats [49] with zero intervening distance, which forms palindromic sequences. This phenomenon is surprisingly abundant relative to the expectation, e.g. large fragments of the human X and Y chromosomes are palindromic [24]. An example of such is present in Fig. 1a, where the sequence GACATGTC is palindromic for the unitig GACAT. TwoPaCo canonicalizes (k+1)-mers, and fails to capture and break such palindromic sequences, reporting the whole sequence as one unitig. Cuttlefish however, can naturally capture these phenomena. We spot-checked that our output matches to TwoPaCo's output, when junction k-mers are properly compacted to an individual unitig (or split into their own) and when the palindromic sequences in TwoPaCo's output are broken and taken as a unitig without repeated k-mers.

6.3 Parallel scalability

To assess the scalability of Cuttlefish across varying number of processor-threads, we used the human genome and set k = 31, and executed Cuttlefish using 1–32 processor threads. Figs. 4a and 4b show the scaling performance charts.



(a) Time taken by each step.

(b) Speedup for each step.

Fig. 4: Scalability metrics of Cuttlefish for varying number of threads, using k = 31 for the human genome.

Generation of the set of keys (i.e. k-mers) is very fast for individual references, and the timing is quite low to begin with – the speedup is almost linear up-to around 8 threads, and then saturates afterwards. The

MPHF (minimal perfect hash function) construction does not scale past 6 threads; which we perceive as due to limitations in our implementation involving bottlenecks in disk-read throughput, and memory access bottlenecks associated to the increasing cache misses in the `BBHash` [26] algorithm. The next two steps: classifying the states of the vertices and extracting the maximal unitigs of the compacted graph, both scale roughly linearly all the way up-to 32 threads. For the extraction step, we chose to report each maximal unitig, and skipped outputting the edges (and paths) for the compacted graph in some GFA format.

6.4 Scaling with k

Next, we assess Cuttlefish's performance with a range of different k-values. We use the 7 human genomes dataset for the experiment, and use 16 threads during construction. Table 3 show the performance of each step with varying k-values.

Table 3: Time- and memory-performance benchmarking for the steps of Cuttlefish across different k.

k	k-mer count (B)	Build steps (s)			Build memory (GB)	Output step (s)		Output memory (GB)
		k-mer set construction	MPHF construction	Vertex classification		Unipaths only	GFA2	
23	2.39	154	160	762	2.67	744	1345	2.82
31	2.59	391	197	791	2.88	737	1203	3.01
61	2.96	439	351	797	3.25	798	831	3.37
91	3.12	1118	573	830	3.42	806	860	3.49
121	3.24	1483	863	841	3.55	850	820	3.62

The running times are in seconds, and the maximum memory usages are in gigabytes.

We observe that the k-mer set construction step slows down with the increasing k, which may be attributable to overhead associated with the `KMC3` [22] algorithm in processing larger k-mer sizes. Although the count of distinct k-mers grows slowly with increasing k, the MPHF construction slows down. Since the MPHF is constructed by iterating over the k-mer set on disk, we believe that the decreased speed in this step that we observe in our implementation with increasing k may be related to disk-read throughput limiting performance. Specifically, reading the k-mers from the set constructed from `KMC3` [22] and feeding those to the `BBHash` [26] algorithm becomes less efficient (time-wise) with increasing k. The steps of classifying the vertices' states and extracting the maximal unitigs are affected very little in their runtime by altering the value of k. Outputting the compacted graph in the GFA2 (and in GFA1) format takes much longer than outputting only the unitigs for lower values of k. At lower k-values, the original graph contains a much larger number of maximal unitigs than it contains at higher k-values; e.g. this dataset contains ~70M maximal unitigs at $k = 23$, but only ~11M at $k = 121$. Therefore, the compacted graph contains a much larger number of vertices at smaller k's, resulting in a much higher number of edges compared to larger k's. For example, the compacted graph for this dataset contains ~141M edges at $k = 121$, and ~3.5B edges at $k = 23$. This also increases the length (in terms of vertex count) of the paths covering the input references. So, the larger amount of disk-write operations required to output the larger GFA makes this step relatively slower at smaller values of k, and it becomes relatively faster as k grows larger.

The memory usages are directly proportional to the k-mer-counts, and thus the maximum memory usage grows with k.

6.5 Effects of the input structure

Next, we evaluate the effects of some of the structural characteristics of the input genomes on the performance of Cuttlefish. Specifically, we assess the impact of the genome sizes (total reference lengths) and their structural variations (through distinct k-mers counts) on the time and memory consumptions of Cuttlefish. The input size determines the total number of k-mers to be processed at the k-mer set construction, vertices classification, and the unitigs extraction steps. Whereas the variation in the input determines the performance of the MPHF construction, and indirectly affects the classification and the extraction steps through their use of the hash table structure. We use the 7 humans and the 7 apes datasets with $k = 31$ for such, using 16

processor threads. References are incrementally added to the input set, and the performance is measured for each input instance. The benchmarks include both the steps of building the compacted graph and extracting the maximal unitigs, and are illustrated in Fig. 5.

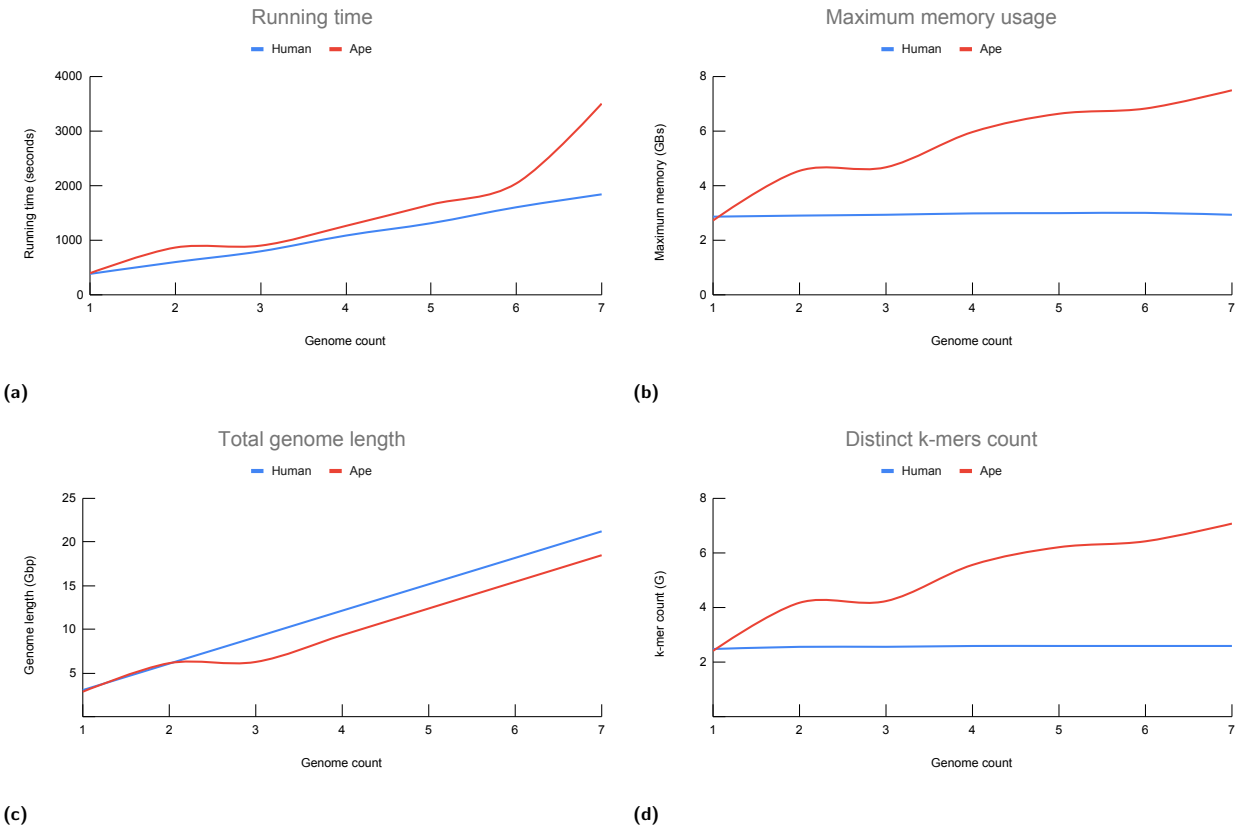


Fig. 5: Input structure effects on the performance of Cuttlefish. For genome counts varying from 1 to 7, the corresponding running time (**a**) in seconds and the maximum memory usage (**b**) in gigabytes are reported. The corresponding total length of the genomes (**c**) and the number distinct k-mers (**d**) for each input collection are presented too.

From Fig. 5, we observe that the running time varies both with the input size and with the structural variation. For the humans dataset, the distinct k-mers count does not increase much from one reference to seven: the increase is just ~5%. This is because the dataset contains references from the same species, hence the genomes are very closely related. Thus the effect of the distinct k-mers count remains roughly similar on all the instances of the dataset. The increase in running time is almost completely dominated by the total workload, i.e. the total size of the genomes. For the apes dataset however, the genomes are not from the same species; and the variations in the genomes contribute to ~200% increase in the distinct k-mers count, from one reference to seven. Thus the running time on this dataset increases with the total genome size, with additional non-trivial effects from the varying k-mers count.

On the other hand, the memory usage by the algorithm is completely determined by the distinct k-mers count, with no effect from the input size. As described at subsection 5.3, the memory usage of the algorithm is constant for a given dataset and k, taking ~8.7 bits/k-mer. Figs. 5b and 5d conforms to the theory: the shapes of the memory consumption and the distinct k-mers count plots are identical.

7 Conclusion

We present a highly scalable and very low-memory algorithm, Cuttlefish, to construct the compacted colored de Bruijn graph for collections of whole genome references. It models the vertices of the original

graph as finite-state automata, and without building the original graph, it correctly determines the state of each automaton. These states characterize the vertices of the graph that flank the maximal unitigs (non-branching paths), allowing efficient extraction of those unitigs. The algorithm is very scalable in terms of time, and it uses a small and constant number of bits per distinct k-mer in the dataset.

Besides being efficient for medium and large-scale genomes, e.g. common mammalian genomes, the algorithm is highly applicable for huge collections of very large genomes. For example, Cuttlefish constructed the compacted de Bruijn graph for 100 closely related human genomes of total length ~322 Gbp in less than 8 hours, taking just 29 GB of memory. For 11 conifer plants that are from the same taxonomic order, each with very large individual genomes and having a total genome length of ~204 Gbp, Cuttlefish constructed the compacted graph in less than 11 hours, using 84 GB of memory. The next best method required more than 16 hours, and used 289 GB of memory.

The improvement in performance over the state-of-the-art tools stems from the novel modeling of the graph vertices as finite-state automata. The core data structure is a hash table, designed using a minimal perfect hash function to assign k-mers to indicies, and a bit-packed buckets table storing succinct encodings of the states of the automata. This compact data structure obtains a memory usage of 8.7 bits/k-mer, leading the algorithm greatly outperform existing tools at scale in terms of memory consumption, while being equally fast if not faster. For scenarios with further memory savings requirements, the memory usage can be reduced to 8 bits/k-mer, trading off the speed of the hash function.

The algorithm is currently only applicable for whole genome references. The assumption of the absence of sequencing errors in these references makes complete walk traversals over the de Bruijn graph possible without having the graph at hand. This is not the case for short read sequences, and the sequencing errors make such implicit traversals difficult. A significant line of future research is to extend Cuttlefish to be applicable on raw sequencing data. Also, a bottleneck in the algorithm is hash function construction, which does not scale as well as the other steps. The bottleneck originates from some implementation-specifics that we aspire to rectify.

On repositories with large databases containing many genome references, Cuttlefish can be applied very fast in an on-demand manner, as follows. First, one can build and store a set of hash functions, each one over some class of related genome references. This consists of the first two steps of the algorithm and is a one-time procedure, containing the bottleneck part of the algorithm. Then, whenever some set of references is to be compacted, the hash function of the appropriate class can be loaded into memory, and the algorithm then executes only the last two steps, which are quite fast. This works correctly because the sets of keys that these hash functions are built from are supersets of the sets of keys being used later for the compactions. Cuttlefish provides the option to save and load hash functions, making the scheme feasible.

Also, one may imagine having the compacted graph output in more than one format, e.g. only a list of the maximal unitigs, the entire graph in GFA1 / GFA2 format etc. The algorithm can output a data structure, taking 5.7 bits/k-mer: the 3.7 bits/k-mer hash function, and a 2 bits/k-mer compressed buckets table. For each type of output requirement, this compressed representation is to be loaded into memory and the algorithm resumes at the last step. The memory usage for the outputting task is 6.7 bits/k-mer in this way.

As the number of sequenced and assembled reference genomes increases, the compacted colored de Bruijn graph will likely continue to be an important foundational representation for comparative analysis and indexing. To this end, Cuttlefish implements a new algorithm for constructing these graphs that is fast, highly-parallelizable, and very memory frugal. We believe it will be an important method in large-scale analysis and indexing pipelines.

Acknowledgments

Funding This work is supported by NIH R01 HG009937, and by NSF CCF-1750472, and CNS-1763680.

Disclosure RP is a co-founder of Ocean Genomics Inc.

Command line configurations and datasets

We used Bifrost, TwoPaCo, and Cuttlefish with the following commands:

- `Bifrost build -v -r <reference_file> -o <output_file> -k <k-value> -t <threads_count>`

- `twopaco -o <output_file> --tmpdir <working_directory> -f <filter_size> -k <k-value> -t <threads_count> <reference_file>`
- `cuttlefish build -r <reference_file> -k <k-value> -s <KMC_database_prefix> -t <threads_count> -w <working_directory> -f <output_type> -o <output_file>`

The KMC3 database to be used by Cuttlefish is produced with the following command.

```
kmc -k<k-value> -m<maximum_memory> -sm -fm -ci0 -t<threads_count> <reference_file>
<output_database> <working_directory>
```

For building compacted de Bruijn graphs with multiple input references, the following commands have been used.

- `Bifrost build -v -r <reference_list> -o <output_file> -k <k-value> -t <threads_count>`
- `twopaco -o <output_file> --tmpdir <working_directory> -f <filter_size> -k <k-value> -t <threads_count> <ref_1> <ref_2> ... <ref_N>`
- `cuttlefish build -l <reference_list> -k <k-value> -s <KMC_database_prefix> -t <threads_count> -w <working_directory> -f <output_type> -o <output_file>`

The KMC3 database is produced with the following.

```
kmc -k<k-value> -m<maximum_memory> -sm -fm -ci0 -t<threads_count> @<reference_list>
<output_database> <working_directory>
```

A description of the datasets used in the benchmarks in the paper, and the URLs from which they were obtained can be found at https://doi.org/10.5281/zenodo.4116608.

References

1. Almodaresi, F., Pandey, P., Ferdman, M., Johnson, R., Patro, R.: An efficient, scalable and exact representation of high-dimensional color information enabled via de bruijn graph search. In: Cowen, L.J. (ed.) Research in Computational Molecular Biology. pp. 1–18. Springer International Publishing, Cham (2019)
2. Almodaresi, F., Sarkar, H., Srivastava, A., Patro, R.: A space and time-efficient index for the compacted colored de bruijn graph. Bioinformatics **34**(13), i169–i177 (Jun 2018). https://doi.org/10.1093/bioinformatics/bty292
3. Almodaresi, F., Zakeri, M., Patro, R.: Puffaligner: An efficient and accurate aligner based on the pufferfish index. BioRxiv (2020). https://doi.org/10.1101/2020.08.11.246892
4. Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the BurrowsWheeler transform. Bioinformatics **32**(4), 497–504 (Oct 2015). https://doi.org/10.1093/bioinformatics/btv603
5. 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., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, N., Tesler, G., Alekseyev, M.A., Pevzner, P.A.: SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. Journal of Computational Biology : a Journal of Computational Molecular Cell Biology **19**(5), 455–477 (May 2012). https://doi.org/10.1089/cmb.2012.0021, 22506599[pmid]
6. Bloom, B.H.: Space/Time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422426 (Jul 1970). https://doi.org/10.1145/362686.362692
7. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de bruijn graphs. In: Raphael, B., Tang, J. (eds.) Algorithms in Bioinformatics. pp. 225–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
8. Burrows, M., Wheeler, D.: A Block-sorting Lossless Data Compression Algorithm. Digital SRC, Digital, Systems Research Center (1994), https://books.google.com/books?id=wPIkAQAAIAAJ
9. Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E.S., Nusbaum, C., Jaffe, D.B.: ALLPATHS: de novo assembly of whole-genome shotgun microreads. Genome Research **18**(5), 810–820 (May 2008). https://doi.org/10.1101/gr.7337908, 18340039[pmid]
10. Chaisson, M.J., Pevzner, P.A.: Short read fragment assembly of bacterial genomes. Genome Research **18**(2), 324–330 (Feb 2008). https://doi.org/10.1101/gr.7088808, 18083777[pmid]
11. Chikhi, R., Limasset, A., Jackman, S., Simpson, J.T., Medvedev, P.: On the representation of de bruijn graphs. In: Sharan, R. (ed.) Research in Computational Molecular Biology. pp. 35–55. Springer International Publishing, Cham (2014)
12. Chikhi, R., Limasset, A., Medvedev, P.: Compacting de bruijn graphs from sequencing data quickly and in low memory. Bioinformatics **32**(12), i201–i208 (Jun 2016). https://doi.org/10.1093/bioinformatics/btw279
13. Chikhi, R., Rizk, G.: Space-efficient and exact de bruijn graph representation based on a bloom filter. Algorithms for Molecular Biology : AMB **8**(1), 22–22 (Sep 2013). https://doi.org/10.1186/1748-7188-8-22, 24040893[pmid]

14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
15. Fang, H., Bergmann, E.A., Arora, K., Vacic, V., Zody, M.C., Iossifov, I., O'Rawe, J.A., Wu, Y., Jimenez Barron, L.T., Rosenbaum, J., Ronemus, M., Lee, Y.H., Wang, Z., Dikoglu, E., Jobanputra, V., Lyon, G.J., Wigler, M., Schatz, M.C., Narzisi, G.: Indel variant analysis of short-read sequencing data with scalpel. Nature Protocols **11**(12), 2529–2548 (Dec 2016). https://doi.org/10.1038/nprot.2016.150, 27854363[pmid]
16. Guo, H., Fu, Y., Gao, Y., Li, J., Wang, Y., Liu, B.: deGSM: memory scalable construction of large scale de bruijn graph. IEEE/ACM Transactions on Computational Biology and Bioinformatics pp. 1–1 (2019)
17. Holley, G., Melsted, P.: Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. Genome Biology **21** (2020). https://doi.org/10.1186/s13059-020-02135-8
18. Holley, G., Wittler, R., Stoye, J.: Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms for Molecular Biology **11**(1), 3 (Apr 2016). https://doi.org/10.1186/s13015-016-0066-8
19. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de bruijn graphs. Nature Genetics **44**(2), 226–232 (Feb 2012). https://doi.org/10.1038/ng.1028
20. Karasikov, M., Mustafa, H., Danciu, D., Barber, C., Zimmermann, M., Rätsch, G., Kahles, A.: MetaGraph: indexing and analysing nucleotide archives at petabase-scale. BioRxiv (2020). https://doi.org/10.1101/2020.10.01.322164
21. Killcoyne, S., Sol, A.: FIGG: simulating populations of whole genome sequences for heterogeneous data analyses. BMC Bioinformatics **15**, 149 (May 2014). https://doi.org/10.1186/1471-2105-15-149
22. Kokot, M., Dugosz, M., Deorowicz, S.: KMC 3: counting and manipulating k-mer statistics. Bioinformatics **33**(17), 2759–2761 (May 2017). https://doi.org/10.1093/bioinformatics/btx304
23. Kundeti, V.K., Rajasekaran, S., Dinh, H., Vaughn, M., Thapar, V.: Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. BMC Bioinformatics **11**(1), 560 (Nov 2010). https://doi.org/10.1186/1471-2105-11-560
24. Larionov, S., Loskutov, A., Ryadchenko, E.V.: Chromosome evolution with naked eye: palindromic context of the life origin. Chaos **18 1**, 13105 (2008)
25. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., Wang, J.: De novo assembly of human genomes with massively parallel short read sequencing. Genome Research **20**(2), 265–272 (Feb 2010). https://doi.org/10.1101/gr.097261.109, 20019144[pmid]
26. Limasset, A., Rizk, G., Chikhi, R., Peterlongo, P.: Fast and scalable minimal perfect hashing for massive key sets. In: 16th International Symposium on Experimental Algorithms (SEA 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 75, pp. 25:1–25:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.25
27. Liu, B., Guo, H., Brudno, M., Wang, Y.: deBGA: read alignment with de bruijn graph-based seed and extension. Bioinformatics **32 21**, 3224–3232 (2016)
28. Liu, B., Liu, Y., Li, J., Guo, H., Zang, T., Wang, Y.: deSALT: fast and accurate long transcriptomic read alignment with de bruijn graph-based index. Genome Biology **20**(1), 274 (Dec 2019). https://doi.org/10.1186/s13059-019-1895-9
29. Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., Liu, Y., Tang, J., Wu, G., Zhang, H., Shi, Y., Liu, Y., Yu, C., Wang, B., Lu, Y., Han, C., Cheung, D.W., Yiu, S.M., Peng, S., Xiaoqian, Z., Liu, G., Liao, X., Li, Y., Yang, H., Wang, J., Lam, T.W., Wang, J.: SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. GigaScience **4**(1) (Jul 2015). https://doi.org/10.1186/s13742-015-0069-2, s13742-015-0069-2
30. MacCallum, I., Przybylski, D., Gnerre, S., Burton, J., Shlyakhter, I., Gnirke, A., Malek, J., McKernan, K., Ranade, S., Shea, T.P., Williams, L., Young, S., Nusbaum, C., Jaffe, D.B.: ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. Genome Biology **10**(10), R103 (Oct 2009). https://doi.org/10.1186/gb-2009-10-10-r103
31. Marçais, G.: Compact vector: Bit packed vector of integral values. `https://github.com/gmarcais/compact_vector` (2020)
32. Marchet, C., Kerbiriou, M., Limasset, A.: Indexing de bruijn graphs with minimizers. BioRxiv (2019). https://doi.org/10.1101/546309
33. Marcus, S., Lee, H., Schatz, M.C.: SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics **30**(24), 3476–3483 (Nov 2014). https://doi.org/10.1093/bioinformatics/btu756
34. Minkin, I., Medvedev, P.: Scalable pairwise whole-genome homology mapping of long genomes with BubbZ. IScience **23**(6), 101224–101224 (Jun 2020). https://doi.org/10.1016/j.isci.2020.101224, 32563153[pmid]
35. Minkin, I., Pham, S., Medvedev, P.: TwoPaCo: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. Bioinformatics (2016). https://doi.org/10.1093/bioinformatics/btw609
36. Muggli, M.D., Alipanahi, B., Boucher, C.: Building large updatable colored de bruijn graphs via merging. Bioinformatics **35**(14), i51–i60 (Jul 2019). https://doi.org/10.1093/bioinformatics/btz350

37. Muggli, M.D., Bowe, A., Noyes, N.R., Morley, P.S., Belk, K.E., Raymond, R., Gagie, T., Puglisi, S.J., Boucher, C.: Succinct colored de bruijn graphs. Bioinformatics (Oxford, England) **33**(20), 3181–3187 (Oct 2017). https://doi.org/10.1093/bioinformatics/btx067, 28200001[pmid]
38. Nowoshilow, S., Schloissnig, S., Fei, J.F., Dahl, A., Pang, A., Pippel, M., Winkler, S., Hastie, A., Young, G., Roscito, J., Falcon, F., Knapp, D., Powell, S., Cruz-Ramrez, A., Cao, H., Habermann, B., Hiller, M., Tanaka, E., Myers, E.: The axolotl genome and the evolution of key tissue formation regulators. Nature **554** (Feb 2018). https://doi.org/10.1038/nature25458
39. Pan, T., Nihalani, R., Aluru, S.: Fast de bruijn graph compaction in distributed memory environments. IEEE/ACM Transactions on Computational Biology and Bioinformatics (2018)
40. Pandey, P., Almodaresi, F., Bender, M.A., Ferdman, M., Johnson, R., Patro, R.: Mantis: A fast, small, and exact large-scale sequence-search index. BioRxiv (2017). https://doi.org/10.1101/217372
41. Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J.M., Brown, C.T.: Scaling metagenome sequence assembly with probabilistic de bruijn graphs. Proceedings of the National Academy of Sciences of the United States of America **109**(33), 13272–13277 (Aug 2012). https://doi.org/10.1073/pnas.1121464109, 22847406[pmid]
42. Pevzner, P.A., Tang, H., Waterman, M.S.: An eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences **98**(17), 9748–9753 (2001). https://doi.org/10.1073/pnas.171285098
43. Robertson, G., Schein, J., Chiu, R., Corbett, R., Field, M., Jackman, S.D., Mungall, K., Lee, S., Okada, H.M., Qian, J.Q., Griffith, M., Raymond, A., Thiessen, N., Cezard, T., Butterfield, Y.S., Newsome, R., Chan, S.K., She, R., Varhol, R., Kamoh, B., Prabhu, A.L., Tam, A., Zhao, Y., Moore, R.A., Hirst, M., Marra, M.A., Jones, S.J.M., Hoodless, P.A., Birol, I.: De novo assembly and analysis of RNA-seq data. Nature Methods **7**(11), 909–912 (Nov 2010). https://doi.org/10.1038/nmeth.1517
44. Rødland, E.A.: Compact representation of k-mer de bruijn graphs for genome read assembly. BMC Bioinformatics **14**(1), 313 (Oct 2013). https://doi.org/10.1186/1471-2105-14-313
45. Sayers, E.W., Cavanaugh, M., Clark, K., Ostell, J., Pruitt, K.D., Karsch-Mizrachi, I.: GenBank. Nucleic Acids Research **47**(D1), D94–D99 (Oct 2018). https://doi.org/10.1093/nar/gky989
46. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J.M., Birol, I.: ABySS: a parallel assembler for short read sequence data. Genome Research **19**(6), 1117–1123 (Jun 2009). https://doi.org/10.1101/gr.089532.108, 19251739[pmid]
47. Staden, R.: A new computer method for the storage and manipulation of DNA gel reading data. Nucleic Acids Research **8**(16), 3673–3694 (Aug 1980). https://doi.org/10.1093/nar/8.16.3673, 7433103[pmid]
48. Stevens, K., Wegrzyn, J., Zimin, A., Puiu, D., Crepeau, M., Cardeno, C., Paul, R., Gonzalez-Ibeas, D., Koriabine, M., Holtz-Morris, A., Martínez-García, P., Sezen, U., Marçais, G., Jermstad, K., McGuire, P., Loopstra, C., Davis, J., Eckert, A., de Jong, P., Yorke, J., Salzberg, S., Neale, D., Langley, C.: Sequence of the sugar pine megagenome. Genetics **204**(4), 1613–1626 (Dec 2016). https://doi.org/10.1534/genetics.116.193227
49. Sutherland, G.R., Richards, R.I.: Simple tandem DNA repeats and human genetic disease. Proceedings of the National Academy of Sciences of the United States of America **92**(9), 3636–3641 (Apr 1995). https://doi.org/10.1073/pnas.92.9.3636, 7731957[pmid]
50. Uricaru, R., Rizk, G., Lacroix, V., Quillery, E., Plantard, O., Chikhi, R., Lemaitre, C., Peterlongo, P.: Reference-free detection of isolated SNPs. Nucleic Acids Research **43**(2), e11–e11 (Nov 2014). https://doi.org/10.1093/nar/gku1187
51. Zerbino, D., Birney, E.: Velvet: Algorithms for de novo short read assembly using de bruijn graphs. Genome Research **18**, 821–829 (Jun 2008). https://doi.org/10.1101/gr.074492.107
52. Zerbino, D.R., McEwen, G.K., Margulies, E.H., Birney, E.: Pebble and rock band: Heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler. PLOS ONE **4**(12), 1–9 (Dec 2009). https://doi.org/10.1371/journal.pone.0008407