Supplementary Note 1: Detailed methods for vg implementation

GCSA2 index generation

We generate the GCSA2 index for a vg graph by transforming the graph into an effective De Bruijn graph with $k = 256$, with positional labels that indicate where in the source graph a particular kmer starts. GCSA2 can then generate a GCSA from this labeled De Bruijn graph. To build the De Bruijn graph efficiently, we initially generate kmers with $k = 16$ labelled by their start, end, preceding and following bases in the vg graph. GCSA2 then undertakes four rounds of prefix doubling, wherein it uses the positional information in the input De Bruijn graph to increase the order of the graph.

In certain circumstances where there are repeated patterns of dense variants in the graph, the path complexity explodes exponentially, preventing practical enumeration of the De Bruijn graph. To avoid this, we prune edges from the vg graph which induce more than a certain number (default 3) of combinatorial bifurcations within a certain linear distance (default 24 bases), and remove any extremely short subgraphs that result from this destructive masking operation. Finally, where possible, such as in the human 1000 Genomes Project graph, we replace the pruned regions with the reference genome sequence. This transformation preserves the coordinate system of the graph, allowing us to use it as the basis for seed generation against the unpruned graph.

Maximal exact match based read alignment

vg can efficiently align reads against large graphs supported by xg and GCSA2 indexes through an alignment process based on maximal exact match sequences between the query sequence and the reference graph. First, a set of super-maximal exact matches (SMEMs) of a read are generated by traversing the suffix tree encoded in the GCSA2 index until the count of matching strings drops to 0, then backing off one step to find all longest exact matches. A recursive series of "reseed" passes through the traversal can then identify next-longest matches, which are used both to improve sensitivity. Then, chains of SMEMs that are consistent with the query sequence are found using a Markov model in which the optimal alignment is likely to form a Viterbi path. For each candidate chain, we then locally align the read against the graph. Scoring results from the local alignment are used to rank the candidate alignments. We then return the best alignment, or multiple candidates if multiple mappings are required.

Colinear chaining of MEMs

If the MEMs do not cover the full read length then we attempt to link them together into chains by building a Markov model in which the best possible chains form high-scoring paths. In this model the nodes correspond to the reference graph positions where MEMs in the read occur and the transitions between nodes correspond to a weight that is proportional to the indel size implied by the difference in distance between the positions of the MEMs in the read and their approximate or embedded path-relative distances in the graph. To consider the coordinate spaces implied by all applicable embedded paths, we record one node per embedded path that each MEM touches. We also record a node for each MEM based on its approximate position in the graph, which is estimated as its position in the concatenated nodes of the partially sorted graph. If we are aligning a read pair, the weight between MEMs on different fragments is proportional to the probability of that distance under a learned model of the fragment distribution. One we establish this model, we take the Viterbi path through it as our first candidate alignment. By masking this path out and re-running the Viterbi algorithm on the model, we can extract a series of candidate alignments in descending order of goodness. This clustering model implements a kind of co-linear chaining such that optimal chains are made of series of MEMs which have similar spacing in the read and in the reference system into which we are mapping the read. Although the exact algorithm is different, in spirit our implementation is similar to that developed in (Kuosmanen 2017), which extends colinear chaining to DAGs by running a similar model over the paths in a minimum path cover which they efficiently compute on the graph.

Locally aligning reads to the graph

Given a candidate cluster of exact matches that we have extracted from the SMEM chaining model described in the previous section, we want to derive a complete description of the alignment of the read to the reference graph. Sensitive alignment of each candidate is essential for distinguishing the optimal alignment.

We link seeds using local alignment based on dynamic programming. To make this efficient we extended an implementation (Zhao et al., 2013) of Farrar's SIMD-accelerated striped Smith Waterman (SSW) algorithm (Farrar, 2007), which we term "graph striped Smith-Waterman" GSSW. (Single input multiple data (SIMD) instructions allow vectorized mathematical operations in a single machine instruction, and can be used to greatly speed up algorithms which can be implemented in terms of operations on vectors.) GSSW generalizes all aspects of SSW to operate over sequence directed acyclic graphs, including affine gap penalties, and retains its matrices for traceback.

To interface with GSSW we transform a local region of our graph so that it is acyclic and only represents a single strand of the DNA. The two operations we use in this transformation are *unfold*, which expands the graph to include its reverse complement where accessible via an inversion, and *dagify*, which unrolls strongly connected components of the graph far enough that we are guaranteed to be able to find any sequence of given length k in the source graph in the unrolled one. This allows us to align any sequence of up to length k against a completely general section of a variation graph. Through these steps we retain a mapping from old node ids to new ones, which we will use to project alignments to the transformed graph back into our base coordinate space.

Chunked alignment of long reads

For very long reads, where in the worst case the local dynamic programming can become prohibitively expensive, we break the reads into "bands" of a fixed width w (default 256 base pairs) with overlap between successive bands of $w/8$. We align these bands independently, trim the overlaps from the alignments, and build a Markov model from them similar to that built for MEM chaining, only that here we consider sub alignments as nodes in the model rather than MEMs. In this model we put weights on transitions between alignments that relate to the estimated distance between the alignments in the graph versus their distance in the read, with the objective of making long co-linear chains be the highest-scoring walks through the Markov model. We take the viterbi path through the model to be the first-best alignment. Then, we mask out this path, re-score, and take the viterbi path to get the 2nd, 3rd, and ultimately Nth best alignment, where we want to obtain multiple alignments. After they have been extracted from the model, alignments are "patched" using local alignment of unaligned regions anchored in the graph near the end of previous mapped regions, so that sub-alignments which may have been misaligned due to repeats may be locally aligned correctly. This model allows `vg` to map noisy reads of arbitrary length, and is used as a core component in the long read progressive assembler `vg msga`.

Unfolding

Every node has an implicit default orientation (see Graph Representation below) so that it is possible to determine edges that cause an inversion, i.e. those which connect between a forward and a reverse complement node orientation. In `VG::unfold` we use a breadth first search starting at every inverting edge in the graph to explore the reverse complemented portions of the graph that we can reach within length k from the inverting edge. We then copy this subgraph, take its reverse complement, and replace the inverting edges connecting it to the forward strand of the graph with non-inverting ones. If k is greater than any length in our graph, then we are guaranteed to duplicate the entire reverse complement of the graph on the forward strand, effectively doubling the size of the graph if we have any inversions in it, as shown in Supplementary Figure 3.

DAGification

Variation graphs may have cycles. These are useful as compact representations of copy number variable regions, and arise naturally in the process of genome assembly. However, our partial order alignment implementation does not handle these structures, and so when they occur we convert them into an approximately equivalent acyclic graph in order to align with GSSW. To do so, we unroll cyclic structures by copying their internal nodes an appropriate number of times to allow a given query length to align through the unrolled version of the component.

We first detect all strongly connected components by using a recursion-free implementation of Tarjan's strongly connected components algorithm (Tarjan, 1972). Then, we copy each strongly connected component and its internal edges into a new graph. We greedily break edges in this graph that introduce cycles. Next we k-DAGify the component progressively copying the base component and, for each edge between nodes in the component, connecting from the source node in the previous copy to the target node in the current copy.

We use dynamic programming to track the minimum distance back through the graph to a root node outside the component at each step. When this reaches our target k, we stop unrolling, and add the expanded component back into the graph by reconnecting it with its original neighborhood. For each copy of a node in the DAG-ified component we copy all its inbound and outbound edges where the other end of the edge lies outside the strongly connected component. The resulting graph is acyclic and supports queries up to length k on the original graph using the translation that we maintain between the new graph and the source one. Supplementary Figure 4 provides a visual description of the process.

Base quality adjusted alignment scores and mapping qualities

Base qualities are typically reported on the Phred scale so that the probability of error for a given quality Q is $\epsilon = 10^{-Q/10}$. Assuming no bias in which bases are mistaken for each other, this defines a posterior distribution over bases b for a base call x.

$$P(b|x, \epsilon) = \begin{cases} 1 - \epsilon & b = x \\ \frac{1}{3}\epsilon & b \neq x \end{cases} \tag{1}$$

We use this distribution to derive an adjusted score function. Normally, the match score for two bases is defined as the logarithm of the likelihood ratio between seeing two bases x and y

aligned and seeing them occur at random according to their background frequencies.

$$s_{x,y} = \log\left(\frac{p_{x,y}}{q_y q_x}\right) \tag{2}$$

Next we marginalize over bases from the posterior distribution to obtain a quality adjusted match score.

$$\tilde{s}_{x,y}(\epsilon) = \log\left(\frac{(1-\epsilon)p_{x,y} + \frac{\epsilon}{3}\sum_{b\neq x} p_{b,y}}{q_y\left((1-\epsilon)q_x + \frac{\epsilon}{3}\sum_{b\neq x} q_b\right)}\right) \tag{3}$$

vg works backwards from integer scoring functions to the probabilistic alignment parameters in this equation. After doing so, the match scores are given by

$$\tilde{s}_{x,y}(\epsilon) = \frac{1}{\lambda}\log\left(\frac{(1-\epsilon)q_x q_y \exp\lambda s_{x,y} + \frac{\epsilon}{3}\sum_{b\neq x} q_b q_y \exp\lambda s_{b,y}}{q_y\left((1-\epsilon)q_x + \frac{\epsilon}{3}\sum_{b\neq x} q_b\right)}\right). \tag{4}$$

Here, λ is a scale factor that can be computed from the scoring parameters, and the background frequencies q_x are estimated by their frequency in the reference graph. Since base quality scores are already discretized, the adjusted scores can be precomputed and cached for all reasonable values of ϵ.

Mapping qualities

The algorithm for mapping qualities in vg is also motivated by a probabilistic interpretation of alignment scores. The score of an alignment A of two sequences X and Y is the sum of scores given in (2). This makes it a logarithm of a joint likelihood ratio across bases, where the bases are assumed independent (a more complete justification including gap penalties involves a hidden Markov model, but it can be shown to approximate this formula). We denote this score $S(A|X,Y)$. Thus, assuming a uniform prior over alignments, we can use Bayes' Rule to motivate a formula for the Phred scaled quality of the optimal alignment, $\hat{A}$.

$$\begin{aligned}
Q(\hat{A}|X,Y) &= -10\log_{10}(1 - P(\hat{A}|X,Y)) \\
&= -10\log_{10}\left(1 - \frac{P(X,Y|\hat{A})}{\sum_A P(X,Y|A)}\right) \\
&= -10\log_{10}\left(1 - \frac{\exp \lambda S(\hat{A}|X,Y)}{\sum_A \exp \lambda S(A|X,Y)}\right)
\end{aligned} \tag{5}$$

Using the close approximation of the LogSumExp function by element-wise maximum, there is a fast approximation to this formula that does not involve transcendental functions.

$$Q(\hat{A}|X,Y) \approx \frac{10\lambda}{\log 10}\left(S(\hat{A}|X,Y) - \max_{A \neq \hat{A}} S(A|X,Y)\right) \tag{6}$$

In practice, we do not compare the optimal alignment to all possible alignments, but to the optimal alignments from other seeds. Thus, the mapping quality indicates the confidence that we have aligned the read to approximately the correct part of the graph rather then that the fine-grained alignment in that part of the graph is correct. It is also worth noting that since this formula is based on alignment scores, it can incorporate base quality information through the base quality adjusted alignment scores.

Succinct graph representation (xg)

When using a variation graph as a reference system, we are unlikely to need to modify it. As such we can compress it into a system that provides efficient access to important aspects of the graph. Specifically, we care about the node and edge structure of the graph and queries that allow us to extract and seek to positions in embedded paths. We would like to be able to query a part of the graph corresponding to a particular region of a chromosome in a reference path embedded in the graph. Similarly, if we find an exact match on the graph using GCSA2, we would like to load that region of the graph into memory for efficient local alignment.

We implement a succinct representation of variation graphs in the xg library, using data structures from SDSL-lite. Node labels and node ids are stored in a collection of succinct vectors, augmented by rank/select dictionaries that allow the lookup of node sequences and node ids. An internal node rank is given for each node, and we map from and to this internal coordinate system using a compressed integer vector of the same order as the node id range of the graph we have indexed. To allow efficient exploration of the graph, we store each node's edge context in a structured manner in an integer vector, into which we can jump via a rank/select dictionary keyed by node rank in the graph. Efficient traversal of the graph's topology via this structure is enabled by storing edges a relative offsets to "to" or "from" node, which obviates the need for secondary lookups and reduces the cost of step-wise traversal to member access on the containing vector and the cost of parsing each node context record that we encounter. Paths provided to XG are used to induce multifarious coordinate systems over the graph. We store them using a collection of integer vectors and rank/select dictionaries that allow for efficient queries of the paths at or near a given graph position, as well as queries that give us the graph context near a given path position.

Supplementary References

Farrar, M. Striped Smith-Waterman database searches six times over other SIMD implementations. *Bioinformatics,* 23(2):156–161, 2007.

Grossi, R., Gupta, A. and Scott Vitter, J. High-order entropy-compressed text indices. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms,* pages 841–850, Society for Industrial and Applied Mathematics, 2003.

Kuosmanen, Anna, et al. Speeding up Dynamic Programming on DAGs through a Fast Approximation of Path Cover. *arXiv preprint arXiv:1705.08754*. 2017.

Okanohara, D. and Sadakane, K. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Experiments,* pages 60–70. Society for Industrial and Applied Mathematics, 2007.

Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing.* 1(2):146–160, 1972.

Zhao, M., Lee, W-P., Garrison, E. and Marth, G. SSW library: An SIMD smith-waterman C/C++ library for use in genomic applications. *PloS One*, 8:e82138, 2013.

Supplementary Table 1

Reference set	*N vars*	*vg*		*index*		*search time*	
	(M)	*time*	*size*	*time*	*size*	*PE*	*SE*
GRCh37	0	1:09:54	1.76	23:30:41	25.11	33:34	28:33
1000GP AF0	84.8	3:42:01	3.92	51:05:07	63.28	45:10	39:46
1000GP AF0.001	30.2	2:00:08	2.58	31:45:12	38.10	39:33	32:53
1000GP AF0.01	14.3	1:35:02	2.17	27:18:53	30.94	33:13	27:09
1000GP AF0.1	6.8	1:23:04	1.97	26:06:38	27.79	32:35	28:43

Table 1: Numbers of variants, file sizes in gigabytes (GB) and build and search times in hours:minutes:seconds for various human vg graphs and associated indexes. Reference sets are the linear reference GRCh37, the full 1000 Genomes Project set 1000GP AF0, and subsets of 1000GP AF0 including only variants with allele frequency above thresholds 0.001 (0.1%), 0.01 (1%) and 0.1 (10%) respectively. The number of variants in millions for each of these data sets is shown. Search times are for 10 million 150+150bp read pairs simulated from NA24385.