BMC
Research Notes

TECHNICAL NOTE Open Access

BetaSearch: a new method for querying β-residue motifs

Hui Kian Ho[1,2*], Graeme Gange[1], Michael J Kuiper[3] and Kotagiri Ramamohanarao[1]

Abstract

Background: Searching for structural motifs across known protein structures can be useful for identifying unrelated proteins with similar function and characterising secondary structures such as β-sheets. This is infeasible using conventional sequence alignment because linear protein sequences do not contain spatial information. β-residue motifs are β-sheet substructures that can be represented as graphs and queried using existing graph indexing methods, however, these approaches are designed for general graphs that do not incorporate the inherent structural constraints of β-sheets and require computationally-expensive filtering and verification procedures. 3D substructure search methods, on the other hand, allow β-residue motifs to be queried in a three-dimensional context but at significant computational costs.

Findings: We developed a new method for querying β-residue motifs, called BetaSearch, which leverages the natural planar constraints of β-sheets by indexing them as 2D matrices, thus avoiding much of the computational complexities involved with structural and graph querying. BetaSearch exhibits faster filtering, verification, and overall query time than existing graph indexing approaches whilst producing comparable index sizes. Compared to 3D substructure search methods, BetaSearch achieves 33 and 240 times speedups over index-based and pairwise alignment-based approaches, respectively. Furthermore, we have presented case-studies to demonstrate its capability of motif matching in sequentially dissimilar proteins and described a method for using BetaSearch to predict β-strand pairing.

Conclusions: We have demonstrated that BetaSearch is a fast method for querying substructure motifs. The improvements in speed over existing approaches make it useful for efficiently performing high-volume exploratory querying of possible protein substructural motifs or conformations. BetaSearch was used to identify a nearly identical β-residue motif between an entirely synthetic (Top7) and a naturally-occurring protein (Charcot-Leyden crystal protein), as well as identifying structural similarities between biotin-binding domains of avidin, streptavidin and the lipocalin gamma subunit of human C8.

Background

The β-sheet is a common secondary structure element that plays important functional and structural roles in proteins, for example, the ligand-binding pockets of biotin-binding proteins and the structure of the commonly-occurring TIM-barrel fold [1]. These processes are often mediated by interactions between adjacent pairs of residues across β-strands. These include the disulphide, ionic, and hydrogen bonds; and hydrophobic packing interactions frequently involved in maintaining the structural stability of a protein or in enzymatic active sites [1]. The influence of pairwise interactions within β-sheets and their tertiary structures have been studied experimentally [2] and statistically [3,4], the results of which have been used to predict β-sheet topology [5-7] and tertiary structure [8]. These studies have provided insights into the folding mechanisms of β-sheets although it remains an open problem [4]. Examining interresidue interactions at the single pairwise level however, provides only a limited view of a larger interaction network within a β-sheet. We refer to these clusters of interacting residues as *β-residue motifs*, which are contiguous subsets of β-sheet residues connected by peptide and/or hydrogen bonds (as shown in Figure 1D). Unlike sequence motifs,

*Correspondence: hohkhkh1@csse.unimelb.edu.au
[1] Department of Computing and Information Systems, The University of Melbourne, Victoria, Australia
[2] National ICT Australia (NICTA), The University of Melbourne, Victoria, Australia
Full list of author information is available at the end of the article

β-residue motifs encode information about both the peptide and bridge-partners of each residue. For the purposes of this study, we consider β-residue motifs to be provisional, since they may also exist via a general conservation between homologs rather than as independent functional units, as is the case for motifs in the traditional sense [9].

Many characteristic β-residue motifs are observed in the Protein Data Bank (PDB) [10]. For example, the β-sheets in leucine-rich repeat (LRR) domains contain consecutive adjacent interstrand pairs of buried leucines sterically-packed alongside their bridge-partners, contributing to structural stability [11]. Other β-residue motifs appear as a combination of inter- and intrastrand residue neighbours, as is the case for the TCT motif of certain antifreeze proteins [12] and glutamic acid/lysine motifs [13]. The conserved biotin-binding site in streptavidin [PDB:1STP] contains a β-residue motif of five inward-directing residues of a β-barrel: S88, T90, W92, W108, and L110. Identification of these β-residue motifs can be used to search for other proteins with similar structural elements or function with low sequence identities.

Searching for structural motifs, β-residue or otherwise, in the PDB using linear approaches such as sequence alignment is a difficult, if not impossible task because interresidue interactions can occur across secondary structures that are sporadically located throughout a protein sequence. Furthermore, pairwise residue interactions are not accounted for in conventional multiple-sequence alignment tools such as BLAST [14] and CLUSTALW [15], given the one-dimensional nature of sequences.

The conventional approach to motif querying, involves the use of protein substructure search methods that structurally align the 3D atomic coordinates of a query with known protein structures. These methods provide a structural context to each query hit and generally produce approximate matches in the form of a ranked list of hits but may take hours to perform few queries due to their reliance on structural alignment algorithms [16].

Alternatively, protein structures can be represented as graphs and queried for motifs using graph indexing approaches [17]. Unlike 3D substructure searches, these methods perform exact matching by querying only the discrete edge, node, and label features of graphs rather than by 3D similarity between continuous coordinates. The query matching algorithms used by existing graph indexing methods are based on solutions to the subgraph isomorphism problem, described briefly as follows:

> A *graph* $G = (V, E)$ is defined by a set of vertices $v \in V$ and a set of edges $e \in E$ where each edge represents a connection between a pair of vertices $\left(v_i, v_j\right) \in V$. A graph is *undirected* if its edges are unordered pairs and *directed* otherwise. The *degree* of a node is the number of edges it has to other nodes.
>
> If G_1 and G_2 are graphs defined as $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, then G_1 is a *subgraph* of G_2 if $V_1 \subseteq V_2 \;\wedge\; E_1 \subseteq E_2$. An *isomorphism* between G_1 and G_2 is a bijection $f : V_1 \rightarrow V_2$ such that $\forall\left(v_i, v_j\right) \in E_1 \iff \left(f(v_i), f(v_j)\right) \in E_2$.

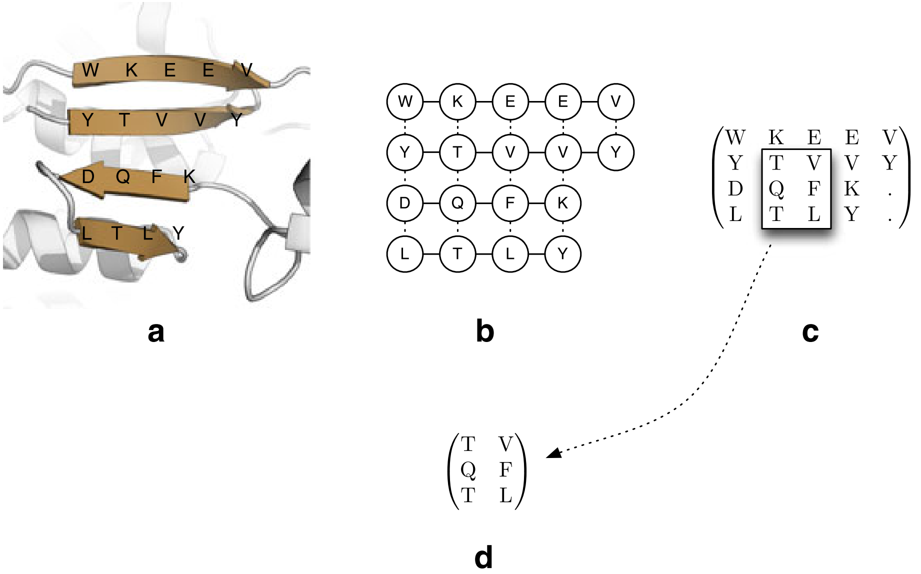

Figure 1 An example β-sheet with its corresponding β-graph, β-matrix, and β-residue motif. Here we show an example conversion of a β-sheet (**a**) to its graph representation (or "β-graph") (**b**) and the β-matrix projection (**c**). A β-residue motif (**d**) is a sub-matrix of its source β-matrix, or a connected subgraph of its source β-graph. The source code to perform these conversions are available for download (see Availability).

> A graph G_1 is *subgraph isomorphic* to G_2 if G_3 is a subgraph of G_2 and there exists an isomorphism between G_3 and G_1.

Graph representations of proteins [18] and β-sheets [17] have been previously described in which nodes represent residues and edges represent inter-residue interactions such as peptide or hydrogen bonds. For simplicity, we define a *β-graph* to be a graph representation of a β-sheet in which each node is labelled with a residue name, solid edges represent peptide bonds, and dotted edges represent a bridge-partner relationship between adjacent interstrand residue pairs (Figure 1B). β-residue motifs are considered to be connected subgraphs of β-graphs whose nodes are labelled with amino acids.

The planarity of β-graphs allows for a compact two-dimensional representation. We define a β-matrix to be a projection (or "flattening") of a β-graph onto a 2D matrix of amino acid characters. The residues in the same β-strand are located in the same row and residues connected by bridge edges lie in the same column (Figure 1C). β-residue motifs are then considered to be submatrices of β-matrices (Figure 1D).

Algorithms for detecting subgraph isomorphisms have been described for general graphs [19,20] that run in time factorial to the number of vertices in a query [21] and cannot be solved in polynomial time, as it is proven to be NP-complete for general graphs [22]. Naively performing a subgraph test on every graph in a database is therefore computationally-expensive [23]. Consequently, graph indexing methods were developed to simplify this problem. A potentially large number of non-matching graphs can be pruned from the search by using indexing techniques analogous to those in conventional search engines [24]. Graphs can be indexed by various features using disk- or memory-based indices. These approaches usually consist of three stages:

1. **Index construction**: The features of each graph in a database are obtained, each representing a graph characteristic. A data structure, usually an inverted index, is then constructed in which each feature is associated with the set of their originating graphs.
2. **Filtering**: The features of the query graph are obtained. An initial set of coarse-grained candidates containing these features are retrieved from the index (or indices). It is possible these candidates do not contain any query matches.
3. **Verification**: Each candidate is checked for a subgraph that exactly matches the query. This is performed using a subgraph test in most methods.

Graph indexing methods are loosely classified into three categories: path-based, subgraph-based, and tree-based.

Path-based methods (GraphGrep [25], GraphFind [26], GraphGrepSX [23], and SING [27]) index graphs using paths as features. These methods construct an inverted index I that maps each path p to a set of their originating graphs

$$I : p \mapsto \{G : p \in G\} \tag{1}$$

where p is a sequence of connected vertices in a graph G

$$p = \left(v_i, v_{i+1}, \cdots, v_{i+k-1}\right) \tag{2}$$

$$\forall k : 1 \leq k \leq l_p \tag{3}$$

where l_p is the maximum path length. The filtering process returns the set of candidate graphs C containing all the paths of a query graph Q

$$C = \bigcap_{p \in \text{paths}(Q)} I(p) \tag{4}$$

and verification of each candidate is performed using the VF2 algorithm [20].

Enumerating all paths up to and including length l_p produces large feature sets and consequently, large indices. GraphFind avoids these problems by pruning redundant features using data mining techniques similar to those of gIndex [28]. GraphGrepSX exploits feature redundancy by implementing its index as a suffix tree where each string is a path sequence. The suffix tree was shown to be more space-efficient than the hash tables used by other path-based methods [23]. Our empirical results corroborate these findings (Table 1). SING uses a second filtering stage that prunes candidates by using path locality information. For example, a path p in a candidate must be surrounded by the same paths as in the query. This improvement in filtering comes at the cost of maintaining an auxiliary hash table of locality information.

Subgraph-based methods (gIndex [28], FG-Index [29], and GDIndex [30]) use subgraphs as features and retain more topological information about graphs than paths due to their more complex structures. Index construction then requires time exponential to the number of nodes in each graph which also produces larger indices than those of path-based methods. These problems can be alleviated to a degree by indexing only the most frequent subgraphs [28].

Tree-based methods (TreePi [31], TreePi+δ [32], and CTree [33]) use subtrees as features and are purported to provide an ideal compromise between the small indices of path-based methods and the specificity of subgraph-based methods. Algorithmic operations on trees are generally more asymptotically efficient than those on graphs, in particular, subtree isomorphism can be tested in polynomial time [34]. However, previous results showed that certain path-based methods are still an order of magnitude faster in query time than existing tree-based methods [27].

Table 1 Indexing times and disk sizes

Method	**Dataset size**				
	1,000	**2,000**	**4,000**	**8,000**	**16,000**
GraphGrepSX,$l_p = 4$	27s	57s	1m 44s	3m 29s	7m 43s
	2Mb	*4Mb*	*9Mb*	*18Mb*	*37Mb*
GraphGrepSX,$l_p = 10$	49s	1m 44s	3m 31s	7m 09s	14m 50s
	75 Mb	*142 Mb*	*267 Mb*	*496 Mb*	*925 Mb*
SING,$l_p = 4$	34s	1m 11s	2m 17s	4m 29s	9m 12s
	10 Mb	*21 Mb*	*42 Mb*	*84 Mb*	*169 Mb*
SING,$l_p = 10$	4m 05s	8m 25s	16m 26s	32m 16s	1h 04m 39s
	329 Mb	*636 Mb*	*1187 Mb*	*2243 Mb*	*4279 Mb*
BetaSearch	**9s**	**19s**	**56s**	**2m 03s**	**4m 04s**
	10 Mb	***20 Mb***	***40 Mb***	***81 Mb***	***164 Mb***

The indexing times and disk sizes (italics) of each method. Indexing times were measured as the average over five repetitions. Disk sizes were measured using the POSIX "stat" command.

GCoding [35] generates numeric representations of graphs using encodings of their adjacency matrices and cannot be classified into any of the above groups. These representations allow efficient filtering without computationally expensive graph traversals or feature enumeration. A specialised subgraph isomorphism test is used for verification [35]. While these encodings provide a compact index, expensive eigenvalue calculations are required to compute them [27].

Each of these methods can be applied to a wide variety of problems because they were designed for general graphs (i.e. graphs with an unrestricted degree and/or node count) that do not make use of the inherent structural constraints of β-sheets. For example, each β-residue has at most four neighbours: the preceding and following peptide-bonded residues and one bridge-partner located on each of the two adjacent hydrogen-bonded β-strands.

The problem of protein 3D substructure searching involves searching a database of protein structures for structures that contain substructures similar to a query structure and remains a significant problem in structure biology. These substructures may be relevant to biological processes such as binding sites, enzymatic function, or may be representative of a particular fold family [16,36]. Current methods for substructure searching are based on the comparison of three-dimensional coordinates between structures and use computationally complex structural alignment algorithms. Methods such as Dali [37], DaliLite [38], and SHEBA [39] align protein structures at the residue level, that is, they find one-to-one residue alignments between pairs of proteins; methods such as QPTableauSearch [40] and SATableauSearch [16] align proteins at the level of secondary structure elements (SSEs) and therefore lack residue-level specificity but are generally faster [16]. A common drawback of these methods is that exhaustive pairwise comparisons are required between the query and the each protein structure in a database. This naive approach often leads to redundant comparisons between highly similar structures or structures with no obvious match, ultimately resulting in queries requiring hours or even days to complete [16].

Recently, LabelHash [36,41] was developed primarily for the 3D substructure matching of small motifs, commonly between 4 and 15 residues. This method is unique among structural search methods in general since it uses a precomputed index to vastly accelerate querying in a manner similar to those of graph indexing approaches. Indeed, the results in this paper show that LabelHash yields a considerable performance boost in compute time over a conventional pairwise structural alignment approach (see Results and discussion).

In this paper we describe *BetaSearch*, a method that allows fast querying of β-residue motifs in large datasets of protein structures. Our method leverages the natural planar constraints of β-sheets by indexing them as 2D matrices, known as β-matrices. This approach avoids the geometric, topological, and computational complexities usually involved in 3D substructure or graph querying. Furthermore, by using β-sheet representations independent of a 3D coordinate system, BetaSearch identifies matching β-residue motifs in structurally and sequentially dissimilar proteins.

Results and discussion

We have compared the performance of BetaSearch against state-of-the-art graph indexing and 3D substructure search methods separately. The results of three case studies are also presented, which provide biologically-relevant contexts in which BetaSearch could be used.

Comparisons with graph indexing methods

We compared BetaSearch against SING and GraphGrepSX. SING was shown to outperform existing methods in terms of query time on standard datasets of chemical compounds, protein transcription networks, protein-interaction networks, and synthetic graphs [27].

The elapsed indexing, total query, filtering, and verification times were averaged over five repetitions. SING and GraphGrepSX were run using $l_p = 4$ and $l_p = 10$, where l_p denotes the path length. These values were chosen by the authors of each method in their own comparisons [23,27]. We were unable to use larger l_p values or datasets of more than 16,000 β-sheets due to the memory consumption of the SING and GraphGrepSX implementations. We therefore only reported results for datasets up to and including N = 16,000. Accuracies of each method were not measured since each query matches at least one β-sheet and any non-matching β-sheet is excluded at the filtering and verification stages of each method.

Indexing

The elapsed times and disk space required for index construction are shown in Table 1.

BetaSearch recorded the fastest indexing times with a 1.9 times speed-up over the next fastest method (GraphGrepSX,$l_p = 4$) for the N = 16,000 dataset. The size of the BetaSearch indices were similar to those of SING,$l_p = 4$ since trimers have an effective path length of $l_p = 3$ and both use hash tables.

The $l_p = 10$ variants of SING and GraphGrepSX were slower than their $l_p = 4$ variants due to the increase in the number of features generated in the former case. This observation was consistent with those of general graphs [23].

GraphGrepSX,$l_p = 4$ generated the smallest indices by a considerable margin through the use of a suffix trees to store its indices. However, the results obtained in the following sections show that the reduction in index disk space came at a significant cost to the querying time.

Furthermore, the BetaSearch index is limited only by the size of the hard disk on which it is stored, whereas the implementations of SING and GraphGrepSX used in this study were memory-limited, requiring the entire index to be loaded into memory in order for queries to be performed.

Overall query times

The query time for a single query was calculated as the sum of its filtering and verification times. The time required to perform all the queries on a dataset was measured as the sum of its individual query times, shown in Table 2. These results show that BetaSearch consistently recorded the fastest querying times for all datasets by at least an order of magnitude over the next fastest method (SING,$l_p = 10$) and a 109 times speed-up over the baseline (GraphGrepSX,$l_p = 4$) for the N = 16,000 dataset.

The trade-off between the index disk size and querying times within the SING and GraphGrepSX variants can be seen in these results where the $l_p = 10$ variants required four to five times as much disk space but were at least twice as fast as the $l_p = 4$ variants.

The overall query time speedups for all query sizes were measured using the GraphGrepSX,$l_p = 4$ as the baseline, shown in Figure 2. Only the speedups for the N = 2,000 and 16,000 datasets were shown for the purposes of brevity. The speedups of each method generally tapers down after queries of approximately six to seven edges. This is an expected observation because larger β-sheet subgraph queries are more specific than smaller ones, resulting in fewer possible candidates and therefore a reduced filtering and verification load.

Table 2 Overall query times (graph indexing comparisons)

	Dataset size				
Method	**1,000**	**2,000**	**4,000**	**8,000**	**16,000**
	7205	***14,516***	***29,101***	***58,795***	***117,415***
GraphGrepSX,$l_p = 4$	2m 10s	9m 13s	44m 16s	3h 19m 37s	15h 03m 54s
GraphGrepSX,$l_p = 10$	1m 18s	6m 12s	24m 55s	1h 50m 52s	8h 02m 31s
SING,$l_p = 4$	50s	3m 27s	12m 16s	52m 41s	3h 53m 45s
SING,$l_p = 10$	20s	1m 19s	5m 16s	23m 06s	1h 37m 49s
BetaSearch	**2s**	**7s**	**33s**	**2m 14s**	**8m 19s**

The overall querying times of each method were measured as the average of five repetitions. The number of queries performed on each dataset are shown in italics. A description of the query generation process is provided in the "Benchmarking datasets" section of this paper.

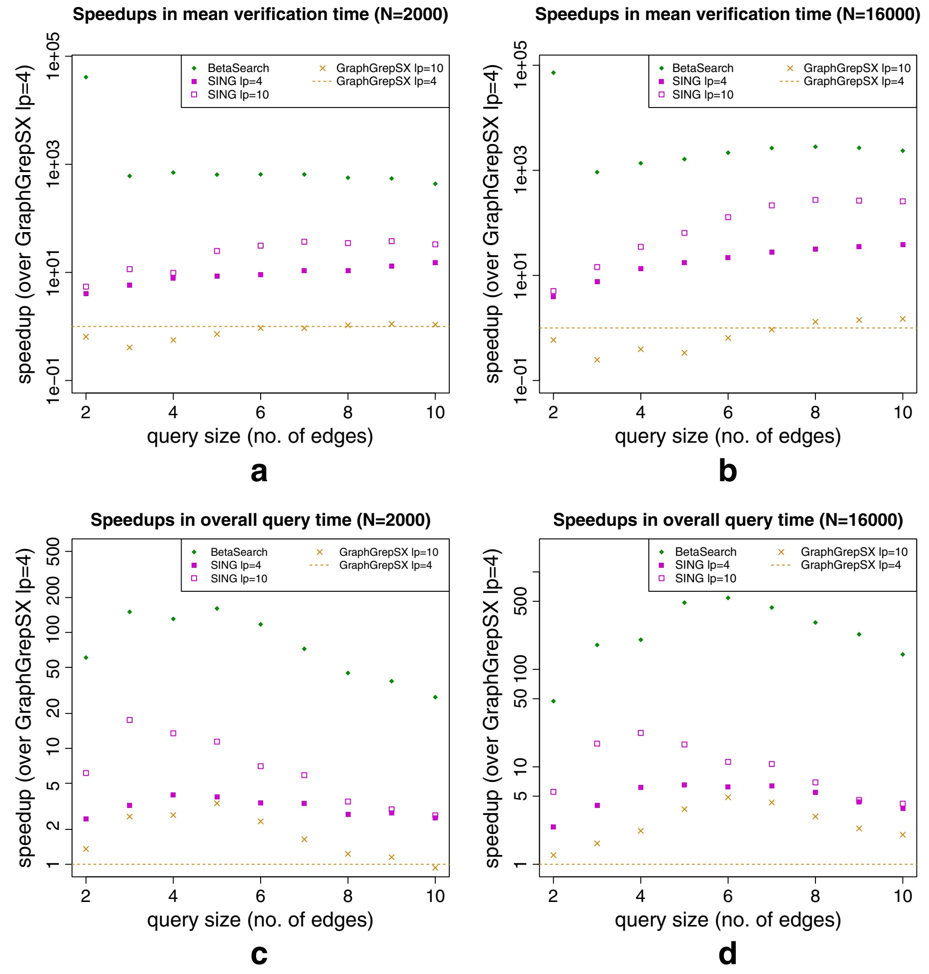

Figure 2 Speedups in mean verification and overall query times. The speedups in mean verification times and overall query times for the $n = 2{,}000$ and 16,000 datasets. The GraphGrepSX, $l_p = 4$ times were used as baselines for each plot. Mean verification times were measured as the total verification time divided by the total number of filtered candidates for each dataset.

Filtering

The filtering time was calculated as the time required to perform filtering for all the queries of a given dataset. The precision was calculated as the total number of actual query matches divided by the total number of filtered candidates for all the queries of a given dataset. The filtering results are shown in Table 3.

In contrast to their indexing performances, the $l_p = 10$ variants of SING and GraphGrepSX generally outperformed their $l_p = 4$ variants. A larger l_p value has more specificity and therefore results in fewer numbers of filtered candidates than a small l_p value, reducing the verification load. The BetaSearch and $l_p = 10$ precision values were consistently near 1.0 for all datasets and query sizes. The precision of the $l_p = 4$ variants were considerably lower than the $l_p = 10$ variants due to the aforementioned specificity limitations of smaller path lengths.

Verification

We measured the mean verification time as the total verification time for a dataset divided by the total number of filtered candidates for a dataset. The mean verification times were less than a second due to the relatively small query graphs involved in this study. The speedups of each method were measured using the GraphGrepSX,$l_p = 4$ times as the baseline and are shown in Figure 2.

BetaSearch consistently recorded the fastest verifications across all query sizes and datasets, this is because the BetaSearch verification algorithm runs in quadratic time whereas the VF2 algorithm employed by SING and GraphGrepSX was designed for general graphs and has a potential non-polynomial time complexity [21]. The largest speed-up by BetaSearch was achieved for queries with two edges, since these queries equated to individual trimers, there was no need for candidates to be verified.

Table 3 Filtering times and precisions (graph indexing comparisons)

Method	Dataset size				
	1,000	**2,000**	**4,000**	**8,000**	**16,000**
GraphGrepSX,$l_p = 4$	9s	40s	3m 04s	13m 30s	56m 27s
	0.38	*0.34*	*0.23*	*0.08*	*0.03*
GraphGrepSX,$l_p = 10$	10s	43s	3m 03s	13m 14s	55m 05s
	0.92	*0.97*	*0.97*	*0.34*	*0.20*
SING,$l_p = 4$	25s	1m 42s	6m 38s	28m 23s	2h 01m 40s
	0.39	*0.39*	*0.39*	*0.39*	*0.39*
SING,$l_p = 10$	13s	54s	3m 51s	16m 54s	1h 10m 32s
	0.98	*0.99*	*1.00*	*1.00*	*1.00*
BetaSearch	**2s**	**7s**	**32s**	**2m 13s**	**8m 16s**
	1.00	***1.00***	***1.00***	***1.00***	***1.00***

The filtering times and precisions (italics). Precision was calculated as the number of actual query matches divided by the number of filtered candidates for all queries of a given dataset.

Comparisons with 3D substructure search methods

We have compared BetaSearch with LabelHash and SHEBA since they each perform residue-level matching. SHEBA was shown to be amongst the most accurate substructure search methods in recent work [16], however, LabelHash has yet to be evaluated against other methods. LabelHash and SHEBA were run using default search parameters. Comparisons with DaliLite were unable to be performed due to the majority of our queries and β-sheets not meeting the minimum number of residues required by DaliLite. DaliLite was shown to have accuracies comparable to SHEBA but with considerably longer compute times [16].

Figure 3A shows the F_1 scores computed across all query sizes for each method. Exact matches for each method were considered to be those with $p' = 1$ for LabelHash and $m = 1$ for SHEBA. We also computed F_1 at $p' \geq 0.999$, however, the F_1 at $m \geq 0.999$ was identical to that of $m = 1$ so we instead computed F_1 at $m \geq 0.95$. BetaSearch, by virtue of inherent exact matching, produces unranked hits and consequently produces an F_1 score of 1.0 for the entire query set.

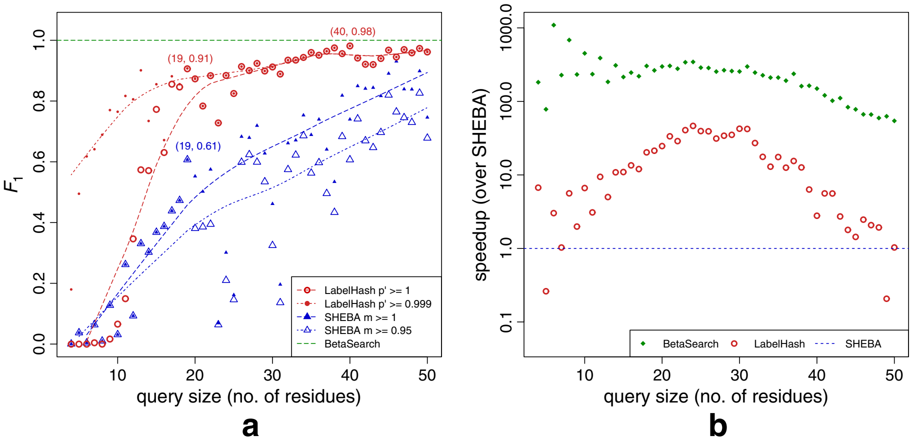

Figure 3 Comparisons with 3D substructure search methods. (**a**) The F_1 score versus query size, p' and m denote the hit scores at which an exact match is considered and all hits below this threshold are excluded. BetaSearch is an exact-matching method and does not produced ranked hits, it therefore yields an F_1 score of 1.0 for all queries. (**b**) The CPU time speedups of BetaSearch and LabelHash over SHEBA. Elapsed wallclock times were nearly identical to the CPU times and were therefore omitted.

LabelHash at $p' \geq 0.999$ clearly outperforms SHEBA on all query sizes, however, neither method performed particularly well on queries of 10 residues or less with the worst F_1 scores observed for queries of 4–5 residues, which have the largest number of hits amongst all query sizes (see Additional file 1: Figure S1A). Although, once the queries reach sizes of 25 residues, LabelHash maintained F_1 scores of at least 0.9 since the number of possible hits closely approaches the number queries (see Additional file 1: Figure S1B).

We measured the CPU times of each method and computed the speedups over SHEBA. The wallclock times were also measured but were omitted since they were analogous to the CPU times. The CPU times of each method for the ASTRAL95 query set were measured as follows:

- SHEBA – 239 h 25 m
- LabelHash – 33 h 17 m
- **BetaSearch – 0 h 59 m**

Figure 3B shows the speedups at each query size. BetaSearch achieved total speedups of 240 times over SHEBA and 33 times over LabelHash. The largest speedups of BetaSearch were obtained for queries of 4–15 residues, which are the sizes of commonly studied motifs [41]. The improved performance of BetaSearch and LabelHash over SHEBA can be attributed to their use of indices which removes the need to perform exhaustive pairwise comparisons for each query against the dataset. This naive approach to substructural searching can lead to query sets taking days to complete [16,40].

Case Studies

β-residue motifs can contribute both to the structural and functional features of a protein. For researchers who study protein structure, BetaSearch can be a useful tool for surveying particular β-sheet configurations across known protein structures. The frequency of a particular motif may give an indication of its relative stability as a β-sheet structural element. Researchers who study functional aspects of β-sheets can use BetaSearch to identify similar motifs in unrelated proteins, as we demonstrate with the biotin-binding pockets of avidin and streptavidin. BetaSearch is fast with a simple, intuitive search query context that allows the researcher to efficiently make comparisons against known β-sheets.

A typical BetaSearch workflow involves the researcher (i) inspecting a protein structure for a β-sheet of interest, (ii) identifying a specific β-residue motif, and (iii) manually entering the amino acids in the corresponding β-matrix into BetaSearch. Alternatively, this workflow can be automated, allowing BetaSearch to be used in a data mining or knowledge-discovery capacity which potentially allows interesting relationships between specific amino acid configurations and protein structures or functions, which would not be intuitively revealed by manual trial-and-error querying.

To demonstrate the capabilities and potential use-cases of BetaSearch we present the results of three case studies. These were drawn from real-world examples and illustrate the role β-residue motifs play in the structure and function of proteins. We also provide the matches from comparative queries using BLAST to demonstrate the difference in matches between a conventional sequence-based homology search and BetaSearch (see Additional files 2, 3, and 4).

Case Study 1 - Synthetic motifs in the Top7 protein

Top7 [PDB:1QYS] is the only engineered protein (non-hypothetical) not to be derived from the sequence or structure of any other protein [42,43]. Most notably, it adopts a unique fold that has yet to be observed in nature. Its structure consists of an amphipathic β-sheet and two α-helices. Inspection of the β-sheet revealed a repeating β-residue motif (Figure 4A). Using this as a query, we wanted to discover known protein structures that possessed this putative synthetic motif. BetaSearch was used to query the PDB2011 dataset, which revealed matches only in structures of the Charcot-Leyden crystal (CLC) protein [PDB:1G86,1HDK,1LCL,1QKQ].

A structural alignment of Top7 and CLC around the matching regions of the query motif (Figure 4B) shows remarkably, that the β-strand topology and sidechain directions are nearly identical. This does not suggest a homology between the two proteins because Top7 is entirely synthetic. However, our findings demonstrate that the RosettaDesign [44] approach used to engineer Top7 had inadvertently reproduced a known stable β-residue motif *ab initio*. The CLC protein was not found in a BLAST query of Top7 chain A (see Additional file 2).

Case Study 2 - Biotin-binding domains

Streptavidin [PDB:1STP] and avidin [PDB:1VYO] are structurally and functionally similar homologous proteins that bind strongly to biotin despite having a sequence identity of less than 35%. Both proteins consist of eight antiparallel β-strands that fold into a β-barrel, inside of which forms a highly conserved biotin-binding site. The β-residue motifs that line this highly specific site are shown in Figures 5A and 5B. When the residues on the non-binding face of the β-sheet are ignored, the two motifs are differentiated by only a single residue: $\mathrm{W}_{92}^{\mathrm{1STP}} \leftrightarrow \mathrm{F}_{79}^{\mathrm{1VYO}}$. The results from the corresponding BLAST query are shown in Additional file 3.

We have characterised these biotin-binding sites as a minimal, β-residue motif (Figure 5C). This putative

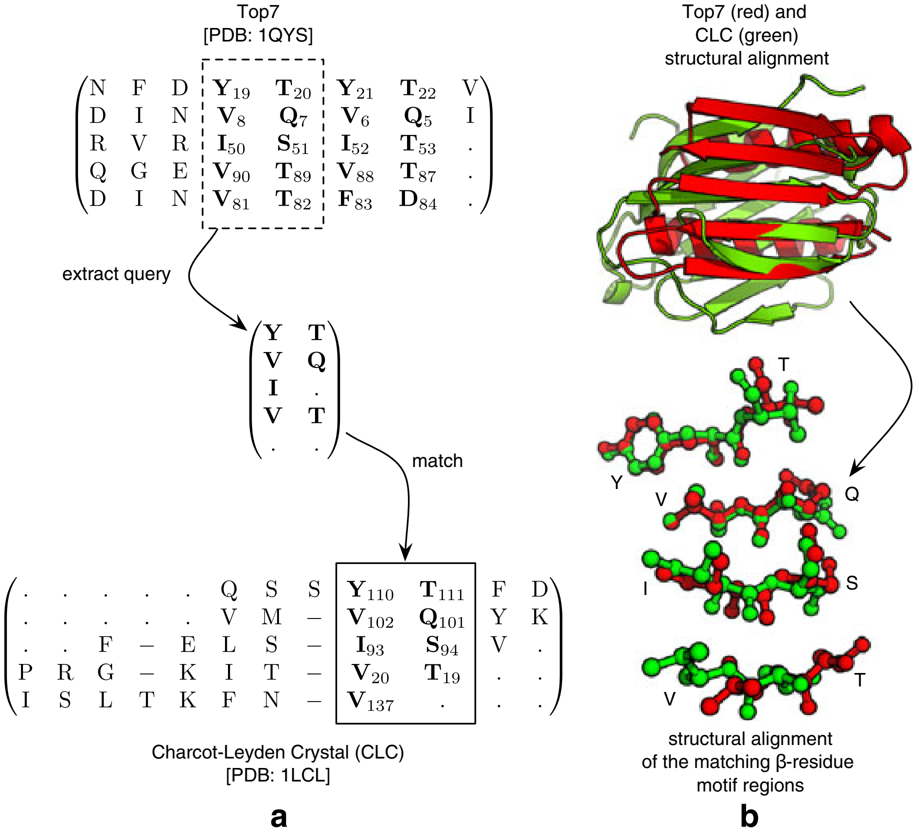

Figure 4 Matching the Top7 and CLC β-sheets (Case Study 1). The amphipathic query (**a**) is a characterisation of the repeating amphipathic region in the Top7 protein (bold). This query was matched only in the CLC protein [PDB: 1LCL]. A structural alignment (**b**) around the query-matching regions of both proteins where the strand and sidechain directions are conserved. The structural alignment was generated using the PyMOL "align" command: "`fetch 1QYS; fetch 1LCL; align 1QYS & (resi 8+19+50+81+90), 1LCL & (resi 20+93+102+110)`".

motif is evolutionary conserved between the avidins and has not yet been shown to be recurrent in evolutionarily distant proteins. Using this query, BetaSearch not only identifies the structures of avidin and streptavidin, but also xenavidin—a biotin-binding protein from *Xenopus tropicalis* (frog). A number of seemingly unrelated proteins were also matched including uncharacterised proteins from *Roseovarius nubinhibens* [PDB:3BVC] and *Oceanicola granulosus* [PDB:2RG4]; and the human complement protein C8 gamma [PDB:1IW2]. The complete set of matching β-sheets is listed in Additional file 4.

Inspection of the uncharacterised proteins reveal a similar arrangement of residues to the known-biotin binding proteins but with less room for the ligand to bind. More tantalising is the match with the gamma subunit of human C8 which is a crucial component of the cytolytic membrane attack complex (MAC) [45]. This subunit has a characteristic lipocalin fold with a distinctive binding pocket similar to the avidins, however, the ligand target of C8 gamma remains unknown [45]. Based on the spatial similarities of this binding pocket with the biotin-binding sites of avidin, one may suggest that these proteins could have an affinity for biotin or a biotin-like compounds.

These results demonstrate that a relatively small β-residue motif query can be matched in unrelated proteins. This capability can be particularly useful in characterising proteins of unknown function by similarities in β-residue motifs to those of known function.

Case Study 3 - β-strand pairing prediction

One of the unsolved problems of tertiary structure prediction is the ability to predict the pairs of β-strands which are hydrogen bonded, and therefore adjacent, in a β-sheet [46]. Information about adjacent β-strands can be used to determine the overall topology of a β-sheet. A number of β-sheet topology prediction algorithms exist that are based on well-known machine learning methods [46-49].

We used BetaSearch to predict the β-strand pairings of the five-stranded β-sheet found in chain A of c-src tyrosine kinase [PDB: 1A09]. This β-sheet contains five strands, is non-barreled, non-bifurcated, and therefore

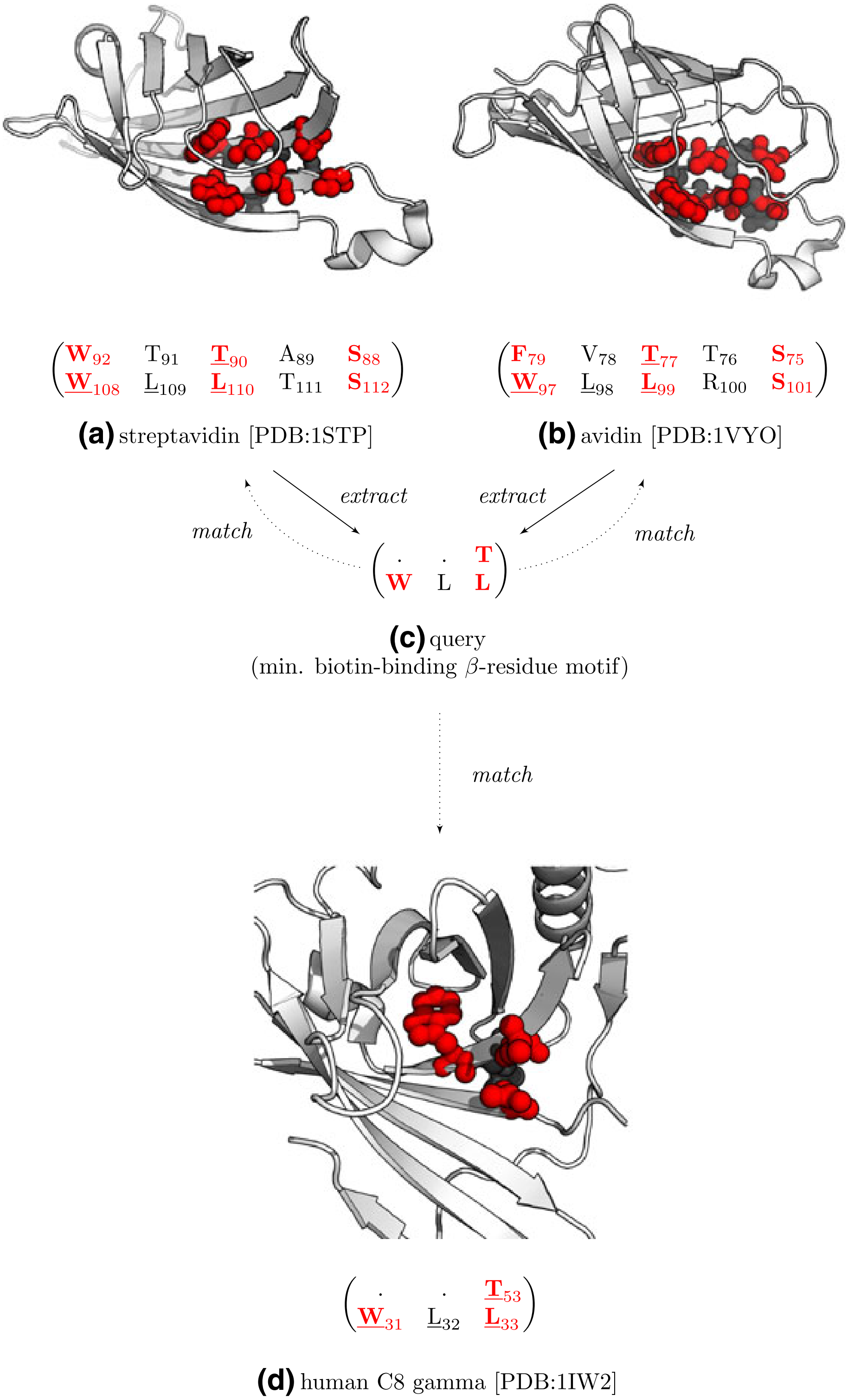

Figure 5 Querying biotin-binding β-residue motifs (Case Study 2). The (**a**) streptavidin and (**b**) avidin biotin-binding motifs are well conserved with the residues on the binding face coloured red. (**c**) The minimal biotin-binding β-residue motif used to query potential biotin-binding proteins. (**d**) The query was matched in the gamma chain of the human complement C8 protein [PDB:1IW2]. The β-matrices shown in (**a**), (**b**), and (**d**) have been abbreviated for clarity and are sub-matrices of larger β-matrices.

has four native strand pairings. A score for each possible strand pair was computed as a function of the number of hits, in a BetaSearch index, obtained for each inter-strand 4-mer query. The top four pairs ranked by pairing scores were considered as predictions. Strand pairing scores were computed for parallel and antiparallel orientations, as shown in Table 4. These results demonstrated that each of the native strand pairs were correctly predicted. The procedures used to perform these predictions are described in the Methods section. Our mechanisms for strand pair scoring are by no means a definitive solution to β-sheet topology prediction. They can, however, be used in existing algorithms such as BetaPro [47] which require preliminary β-strand or β-residue pairing scores in order for predictions to be made. A large scale evaluation of our BetaSearch-based prediction method is the topic of future work.

Conclusion

We have described a method for indexing and querying β-residue motifs, called *BetaSearch*, that is at least an order of magnitude faster than state-of-the-art graph indexing methods. These speedups are achieved by indexing β-sheets as 2D matrices of amino acids known as β-matrices. This representation leverages the inherent planar structural constraints of β-sheets, thereby avoiding much of the computational complexity involved in querying and indexing 3D or graph representations of protein structures. BetaSearch is therefore able to achieve quadratic-time querying. Filtering precisions were close to 1.0 for all datasets and query sizes, resulting in near minimal verification time.

When compared with existing 3D substructure search methods, BetaSearch achieves a 240 times speedup over the baseline (SHEBA) and a 33 times speedup over the next fastest method (LabelHash). The demonstrated efficiency of BetaSearch lends itself well to the rapid exploration of probable motif or β-sheet conformations in a matter of minutes, rather than days or weeks with 3D-based methods. Furthermore, the ability of BetaSearch to perform exact matching ensures that correct hits are not missed.

Table 4 The predicted strand pairing scores for the 1A09 chain A β-sheet (Case Study 3)

Pair	Parallel score	Antiparallel score	Total score
*4,5	4.86	4.99	9.85
*2,3	4.59	4.60	9.19
*3,4	4.32	4.31	8.64
*1,2	4.09	4.09	8.18
3,5	3.12	3.09	6.21
2,4	2.98	2.97	5.96
1,3	2.95	2.97	5.93
1,4	2.72	2.72	5.44
2,5	2.33	2.33	4.66

The strand pairing scores generated for the β-sheet in chain A of c-src tyrosine kinase. The top four scoring pairs (predicted) are delineated and the native pairs are denoted with an asterisk.

Our three case studies demonstrated the utility of BetaSearch in biological contexts. We discovered that the synthetic Top7 protein shares an identical β-residue motif with a known naturally-occurring protein—the Charcot-Leyden crystal. A small query derived from the biotin-binding motif of avidins easily identified unrelated biotin-binding proteins and is suggestive of biotin-binding in others including the gamma subunit of the human C8 complement protein. BetaSearch, with its ability to identify functional similarity from unrelated proteins can potentially help characterise the proteins in the PDB with unknown function. We also demonstrated how BetaSearch could be used to predict strand pairing in β-sheets, which could help reduce the search space of more complex supersecondary or tertiary structure prediction tasks. Although our work has focused on substructural motifs in β-sheets, our algorithm can be modified to perform querying of any substructural motif involving pairwise interactions, such as the well-characterised hydrogen-bond pairings in helices and turns. Indeed, this is an avenue of development we are currently exploring.

It is our intention for BetaSearch to be used by protein researchers to supplement conventional sequence and structural search methods. For example, the efficiency of the BetaSearch filtering and verification algorithms introduces the possibility for their use as a rapid “first-pass” filter to improve the querying performance of other methods. Such an application would be non-trivial to develop but could potentially reduce conventional structural query times from hours to minutes.

Findings

The pseudocode for each of the algorithms described in this section is provided in the Supplementary Materials.

Trimers

A *trimer* is a path of three amino acids in a β-matrix configured in the shape of an ‘L’ (an L-trimer), vertically in the same column (a V-trimer), or horizontally in the same row (an H-trimer). Trimers are the features by which β-matrices are indexed in BetaSearch. An example of the trimer extraction process is shown in Additional file 1: Figure S2.

A trimer t has a number of attributes that encode its configuration and location within a β-matrix:

- *t*.SEQ: a three letter string of residues spanned by the trimer where

$$\text{if } t.\textsc{seq} = \text{“abc''}$$
$$\text{then } t.\textsc{seq}[0] \rightarrow \text{“a''},\ t.\textsc{seq}[1] \rightarrow \text{“b''},\ t.\textsc{seq}[2] \rightarrow \text{“c''}.$$

- *t*.CLASS: an integer representing the *class* of the trimer, defined as

$$t.\textsc{class} = \begin{cases} 1 & \text{if } t \text{ is an L-trimer} \\ 3 & \text{if } t \text{ is a V-trimer and } t.\textsc{seq}[0] \neq t.\textsc{seq}[2] \\ 5 & \text{if } t \text{ is an H-trimer and } t.\textsc{seq}[0] \neq t.\textsc{seq}[2] \\ 15 & \text{if } t \text{ is a V-trimer and } t.\textsc{seq}[0] = t.\textsc{seq}[2] \\ 31 & \text{if } t \text{ is an H-trimer and } t.\textsc{seq}[0] = t.\textsc{seq}[2]\,. \end{cases}$$

- *t*.ID: a (*t*.CLASS,*t*.SEQ) tuple.
- *t*.ORIENT: an integer value such that *t*.ORIENT $\in \{0,1,2,3\}$. These values describe the possible orientations of a trimer and were chosen to allow the calculation of x- and y-axis trimer reflections using the bitwise-XOR (‘$\oplus$’) operator. Orientation reflections are calculated as

$$t'.\textsc{orient} = \begin{cases} t.\textsc{orient} \oplus 1 & \text{if reflected in the y-axis} \\ t.\textsc{orient} \oplus 2 & \text{if reflected in the x-axis} \end{cases}$$

where *t’* is the reflection of *t*. Additional file 1: Figure S2 shows how trimer orientations are determined for each trimer class.
- *t*.EQ-ORIENTS: an integer that encodes the equivalent orientations of *t*.ORIENT, defined as

$$t.\textsc{eq-orients} = \begin{cases} 2^{t.\textsc{orient}} & \text{if } t.\textsc{class} = 1 \\ 3 & \text{if } t.\textsc{class} = 3 \text{ and } t.\textsc{seq}[0] < t.\textsc{seq}[2] \\ 12 & \text{if } t.\textsc{class} = 3 \text{ and } t.\textsc{seq}[0] > t.\textsc{seq}[2] \\ 5 & \text{if } t.\textsc{class} = 5 \text{ and } t.\textsc{seq}[0] < t.\textsc{seq}[2] \\ 10 & \text{if } t.\textsc{class} = 5 \text{ and } t.\textsc{seq}[0] > t.\textsc{seq}[2] \\ 15 & \text{otherwise,} \end{cases}$$

such that

$$t.\textsc{eq-orients}\ \&\ i = \begin{cases} 1 & \text{if orientation } i \text{ is equivalent to } t.\textsc{orient} \\ 0 & \text{otherwise,} \end{cases}$$

where ‘<’ and ‘>’ are the lexicographic less-than and greater-than operators; and ‘&’ is the bitwise-AND operator. The equivalent orientations of a trimer are encoded using bitmasks such that orientation *i* is equivalent to *t*.ORIENT if the i^{th} bit of *t*.CLASS is set to 1. The *t*.CLASS value for each type of trimer is an encoding of the minimum *t*.CLASS value for the class.
- *t*.ROW: the row coordinate of *t*.SEQ[1] within its β-matrix.
- *t*.COL: the column coordinate of *t*.SEQ[1] within its β-matrix.
- *t*.COORD: a (*t*.ROW,*t*.COL) tuple.
- *t*.SPAN1, *t*.SPAN2: the row (*t*.ROW-SPAN) or column spans (*t*.COL-SPAN) of a trimer, depending on the trimer class. Each span is an ordered tuple (i,j) where *i* is the coordinate of *t*.SEQ[1] and *j* is the coordinate of either *t*.SEQ[0] or *t*.seq[2], depending on the trimer type. L-trimers have one row span and one column span, V-trimers have two row spans, and H-trimers have two column spans. Examples of the spans for each trimer are shown in Additional file 1: Figure S3.

Index construction

BetaSearch uses three indices: $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{C}$.

- $\mathcal{D}$ is an inverted index that maps each trimer *id* to the set of β-matrices in which they are contained, defined as

$$\mathcal{D}[\,id\,] \mapsto \{b \in B : id \in b.\textsc{trimer-ids}\}$$

where *B* is the set of β-matrices in the dataset.
- $\mathcal{R}$ maps a compound key $\kappa_{\mathcal{R}}$ to a trimer *t*, defined by

$$\mathcal{R}[\,\kappa_{\mathcal{R}}\,] \mapsto t$$
$$\text{where } \kappa_{\mathcal{R}} = (t.\textsc{matrix-id},\ t.\textsc{id},\ t.\textsc{eq-orients},\ t.\textsc{coord},\ t.\textsc{row-span})$$

such that *class* $\notin \{3, 15\}$.
- $\mathcal{C}$ maps a compound key $\kappa_{\mathcal{C}}$ to a trimer *t*, defined by

$$\mathcal{C}[\,\kappa_{\mathcal{C}}\,] \mapsto t$$
$$\text{where } \kappa_{\mathcal{C}} = (t.\textsc{matrix-id},\ t.\textsc{id},\ t.\textsc{eq-orients},\ t.\textsc{coord},\ t.\textsc{col-span})$$

such that *t*.CLASS $\notin \{5, 31\}$.

L-trimers are indexed in $\mathcal{R}$ and $\mathcal{C}$; whereas H-trimers are indexed only in $\mathcal{C}$ because they do not contain any row spans, conversely, V-trimers are indexed only in $\mathcal{R}$ because they do not contain any column spans. The BUILD-INDICES procedure in Additional file 1: Algorithm S1 describes the index construction algorithm.

Time complexity

Each entry in a β-matrix is the intersection of at most six trimers: four L-trimers (one in each of the four orientations), one V-trimer, and one H-trimer. BUILD-INDICES runs in $O(6mn)$ time where *m* is the maximum number of residues in a β-matrix and *n* is the number of β-matrices.

Filtering

A query is a well-formed β-matrix *Q*. Preprocessing *Q* requires the following steps:

1. Enumerate the query trimers and storing them in *Q*.TRIMERS.
2. Store the corresponding trimer IDs in *Q*.TRIMER-IDS.

Partially matching candidates are obtained by pruning the β-matrices in $\mathcal{D}$ using two filters. The FIRST-FILTER procedure, described in Additional file 1: Algorithm S2, prunes the β-matrices in $\mathcal{D}$ that do not contain the entire set of trimer IDs in *Q*.TRIMER-IDS.

β-matrices are indexed in a single arbitrary orientation, therefore a procedure for comparing a query against all orientations of a candidate is required. The naive approach enumerates and stores the x- and y-axes reflections of each β-matrix, effectively tripling the index size. Alternatively, the x- and y-axes reflections of the query are enumerated and compared with a candidate, effectively tripling the filtering time.

We have developed an algorithm that prunes invalid candidates without enumerating reflections of the candidate or the query. The algorithm is implemented as the SECOND-FILTER procedure described in Additional file 1: Algorithm S3, where only the candidates congruent to Q are retained from C_1. A query Q is congruent to a candidate c if

$$\bigcap_{\substack{q \in Q.\textsc{trimers} \\ t \in c.\textsc{trimers}}} \{q.\textsc{orient} \oplus t.\textsc{orient} : t.\textsc{id} = q.\textsc{id}\} \neq \emptyset$$

where '$\oplus$' is the bitwise-XOR operator.

(5)

The CONGRUENT procedure defined in Additional file 1: Algorithm S3 implements Equation 5 using bitwise operations that enable constant-time set unions and intersections.

Time complexity

The FIRST-FILTER algorithm runs in $O(|p_{\min}| \cdot |Q.\textsc{trimer-ids}|)$ time where $|p_{\min}|$ is the cardinality of the smallest postings set and $|Q.\textsc{trimer-ids}|$ is the number of unique trimer ids in the query. A postings set is a set in the index of β-matrices containing a particular query trimer. The SECOND-FILTER algorithm runs in $O(|Q.\textsc{trimer-ids}| \cdot |C_1|)$ time where $|C_1|$ is the number of candidates returned by FIRST-FILTER.

Verification

Most graph indexing methods use the VF2 [20] or Ullmann [19] algorithms for candidate verification. Other methods use algorithms optimised for the data structures of their features and indices.

Constraining β-graphs to the simpler structures of β-matrices has enabled us to develop a quadratic time candidate verification algorithm that does not rely on subgraph isomorphism tests.

A graph G of the query is constructed, in which each vertex is a trimer $q \in Q.\textsc{trimers}$ and each edge $e = (q_{\text{src}}, q_{\text{des}})$ indicates a span overlap (i.e. $t.\textsc{col-span}$ or $t.\textsc{row-span}$) between adjacent query trimers q_{src} and q_{des}. The algorithm to construct a query graph is implemented in the MAKE-QUERY-GRAPH procedure described in Additional file 1: Algorithm S4.

The remainder of the verification algorithm attempts to find a subgraph of a candidate c that matches G by matching each pair $(q_{\text{src}}, q_{\text{des}})$ to a pair $(t_{\text{src}}, t_{\text{des}}) \in c.\textsc{trimers}$, where a match between pairs is defined by

$$\begin{aligned}
&\textsc{Match-Pairs}\,(q_{\text{src}}, q_{\text{des}}, t_{\text{src}}, t_{\text{des}}) = \\
&\quad \textsc{Rel-Orient}\,(q_{\text{src}}, q_{\text{des}}) \iff \textsc{Rel-Orient}\,(t_{\text{src}}, t_{\text{des}}) \\
&\quad \wedge\ \textsc{Overlaps}\,(q_{\text{src}}, q_{\text{des}}) \iff \textsc{Overlaps}\,(t_{\text{src}}, t_{\text{des}}) \\
&\quad \wedge\ \textsc{Overlap-Type}\,(q_{\text{src}}, q_{\text{des}}) \\
&\qquad \iff \textsc{Overlap-Type}\,(t_{\text{src}}, t_{\text{des}}) \\
&\quad \wedge\ \textsc{Overlap-Span-Nums}\,(q_{\text{src}}, q_{\text{des}}) \\
&\qquad \iff \textsc{Overlap-Span-Nums}\,(t_{\text{src}}, t_{\text{des}})
\end{aligned} \tag{6}$$

and a match between a query Q and a candidate c occurs if

$$\bigwedge_{q_{\text{src}}, q_{\text{des}} \in G} \left(\bigvee_{t_{\text{src}}, t_{\text{des}} \in c} \textsc{Match-Pairs}(q_{\text{src}}, q_{\text{des}}, t_{\text{src}}, t_{\text{des}}) \right) \tag{7}$$

The keys by which trimers are indexed in $\mathcal{R}$ and $\mathcal{C}$ contain their locations and geometric configurations within a candidate, allowing MATCH-PAIRS to be tested in constant-time. MATCH-PAIRS is implemented using the procedures defined in Additional file 1: Algorithm S8. The VERIFY-CANDIDATE procedure in Additional file 1: Algorithm S6 describes our algorithm for verifying a single candidate.

Time complexity

The MAKE-QUERY-GRAPH procedure runs in $O(|Q.\textsc{trimers}|)$ time where $|Q.\textsc{trimers}|$ is the number of trimers in the query. The VERIFY-CANDIDATE procedure runs in $O(|Q.\textsc{trimers}|^2)$ time, which is called by the VERIFY procedure in Additional file 1: Algorithm S7 to verify each filtered candidate in C_2. Therefore, the overall time complexity of our candidate verification algorithm is $O\left(|Q.\textsc{trimers}| + |C_2| \cdot |Q.\textsc{trimers}|^2\right)$.

β-strand pairing prediction

For Case Study 3, we constructed a BetaSearch index from the ASTRAL95 dataset, as per the 3D substructure search comparisons. The secondary structures for the β-sheet were using DSSP. Each β-strand sequence was delineated as contiguous substrings of "E" secondary structure assignments. Scores for each possible β-strand pairing (i, j) were computed as follows:

1. Let $\mathcal{I}$ be the index generated from the ASTRAL95 dataset.
2. Let S^P and S^A be the strand pairing score matrices for the parallel and antiparallel strand pairs, respectively.
3. Let M be a matrix where M_{ij} contains the number of 4-mers between strands i and j. We define a 4-mer as a single occurrence of two-consecutive bridge

```
   pairings. For example, the $\beta$-matrix $\begin{pmatrix} \mathrm{A} & \mathrm{B} & \mathrm{C} \\ \mathrm{D} & \mathrm{E} & \mathrm{F} \end{pmatrix}$ has
   the 4-mers – $\begin{pmatrix} \mathrm{A} & \mathrm{B} \\ \mathrm{D} & \mathrm{E} \end{pmatrix}$ and $\begin{pmatrix} \mathrm{B} & \mathrm{C} \\ \mathrm{E} & \mathrm{F} \end{pmatrix}$.
4. For each strand pair $(i, j)$:
    (a) For each alignment $a$ of $j$ on $i$:
        (i) For each 4-mer $m$ of $a$:
            (A) Let $h_P$ and $h_A$ be the
                number of hits returned
                from querying $\mathcal{I}$ for the
                parallel and antiparallel
                orientations of $m$,
                respectively.
                Set $S_{ij}^P \leftarrow S_{ij}^P + h_P$
                Set $S_{ij}^A \leftarrow S_{ij}^A + h_A$
                Set $M_{ij} \leftarrow M_{ij} + 1$
    (b) Let seq_sep $(i, j)$ be the number of residues
        separating strands $i$ and $j$.
        Set $S_{ij}^P \leftarrow \frac{S_{ij}^P}{M_{ij}} - \log\left[\text{seq\_sep}\,(i, j)\right]$
        Set $S_{ij}^A \leftarrow \frac{S_{ij}^A}{M_{ij}} - \log\left[\text{seq\_sep}\,(i, j)\right]$
```

The total score for a strand pair was defined as

$$\text{score}\,(i, j) = S_{ij}^P + S_{ij}^A \tag{8}$$

Evaluation

The graph indexing methods were evaluated according to their filtering, verification, and overall query times. The sizes of the indices generated by each method were measured using the POSIX "stat" command. The filtering precision was calculated as the total number of hits divided by the total number of filtered candidates for all queries of a given dataset size.

A "hit" for a query against a β-sheet occurs when it contains (or is) an exact match of the query structure. Unlike BetaSearch, LabelHash and SHEBA perform approximate matching and return a list of hits ranked by a structural similarity score. For LabelHash, this is a statistically-determined p-value where a low value indicates a close match. SHEBA ranks hits according to the m-value which is the number of aligned residues between a query and a result, divided by the number of residues in the larger of the two structures. For simplicity, we denote the LabelHash score for a hit as $p' = 1 - p$. These scores need to be thresholded in order to obtain exact matches so that all hits with scores below a given threshold are ignored and those above the threshold are assigned the same rank. Once a suitable hit score threshold is chosen, the querying accuracy of each method can be computed by counting a hit as a true positive (*TP*) if the query is exactly matched within a β-sheet, a false positive (*FP*) if it is not, or a false negative (*FN*) if a correct β-sheet hit is absent from the list of results. We can then calculate the *recall*, as

$$recall = \frac{TP}{TP} + FN \tag{9}$$

which denotes the proportion of exactly-matching β-sheets out of all correct β-sheet hits. and the *precision*, as

$$precision = \frac{TP}{TP} + FP \tag{10}$$

which denotes the proportion of correct β-sheet hits in a list of hits. These two measures are commonly used to evaluate conventional document retrieval systems [24], as well as protein structural search methods [16]. The *F-score*, defined as

$$F_\beta = \left(1 + \beta^2\right) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall} \tag{11}$$

is the harmonic mean of the precision and recall values. It provides a convenient way of evaluating the query precision and recall as a single value. The β parameter allows emphasis to be placed on precision or recall depending on the query performance goals. We used the F_1 score in our 3D substructure search comparisons as a measure of query accuracy.

Datasets

For the graph indexing comparisons, the January 3, 2011 (PDB2011) snapshot of the PDB was used to generate a dataset of 209,127 β-sheets. A number of PDB files were excluded due to discrepancies in their content [50]. β-sheets exhibiting poor planarity, such as those with significantly pronounced twisting or curvature, were also excluded.

The DSSP [51] algorithm was used to assign secondary structures to each PDB file. Residues with a secondary structural assignment of "E" or "b" were considered to form part of a β-strand. DSSP also assigns bridge partner relationships between residues on adjacent β-strands, which were used to determine the bridge edges in β-graphs.

We used Pro-Origami [52] to generate β-graphs and a topological sort was used to generate the β-matrix from the peptide and bridge edges of each β-graph.

Subsets containing 1,000; 2,000; 4,000; 8,000; and 16,000 β-sheets were randomly selected from the dataset. These sizes were used in previous benchmarks [23,27]. GraphGrepSX and SING were unable to be run on datasets of 32,000 or more due to the memory consumption of their respective implementations, we therefore restricted the sizes of our datasets accordingly.

Each β-graph was preprocessed by inserting "dummy" nodes in place of a labelled edge. Each dummy node was labelled with either a “b” to denote a bridge edge or a “z” to denote a peptide edge in order to avoid conflict with the labels of residue nodes. Preprocessing was required because edge-labelled graphs were not supported by SING or GraphGrepSX.

Queries were generated in the same manner as in previous benchmarks [23]. A query was created from each β-graph by randomly selecting a root node and performing a random breadth-first traversal until the query obtained the degrees d such that $2 \leq d \leq 10$. Queries were not generated from β-graphs with insufficient edges. Each query contained a single wildcard node that matched any amino acid. To enable wildcard matching on all methods, each query was repeated by replacing the wildcard node with each of the 20 amino acids. The β-matrices for each query were generated for use by BetaSearch. The total numbers of queries generated for each query size are shown in Additional file 1: Table S1.

For the case studies, we generated the required indices from all the β-matrices in the PDB2011 dataset. Each β-matrix was assigned a sheet identifier of the format “<PDB ID><chain ID>_SHEET_<number>”. For example, the sheet ID of the first β-matrix in chain A of ubiquitin [PDB:1UBQ] is “1UBQA_SHEET_000”.

For the 3D substructure search comparisons, the ASTRAL SCOP 1.75A 95% sequence identity non-redundant dataset [53] of protein structures (ASTRAL95) were used. β-sheets were extracted and filtered as per the graph indexing datasets and a total of 29,341 β-sheets were obtained. A subset of 26,669 β-sheets containing between 4 and 50 residues, inclusive, were used as queries. The β-matrices corresponding to each β-sheet were generated and used with BetaSearch.

Implementation

The graph indexing comparisons were performed on a 2.66 Ghz Intel Nehalem 8-core processor with 48 GB of main memory running CentOS. The source code to SING and GraphGrephSX were provided by the authors of their respective publications. All methods were implemented in C++, compiled using g++ version 4.3, and depend on the Boost C++ version 1.42.0 libraries [54]. BetaSearch additionally requires Redis [55] version 2.0.4 and the official Redis C headers [56].

The indices in SING and GraphGrepSX were implemented as modified C++ STL “std::map” containers in the memory spaces of their respective processes. In contrast, BetaSearch stores its indices using Redis hash tables that are stored in (disk-based) virtual memory and operates external to BetaSearch as a concurrent process. It is therefore subject to interprocess communication overhead during filtering and indexing, which are included in our experimental timings.

The 3D substructure search comparisons were performed on the same platform with 8 GB of main memory. BetaSearch was re-implemented in Python 2.7 using the Whoosh Python Search Library [57]. LabelHash 1.0.2 [58] and SHEBA 3.1.1 [59] were downloaded and used. The LabelHash index was built from the PDB coordinates of all the ASTRAL95 β-sheets, which were extracted from their original structures using ProDy [60]. The BetaSearch index was built from the β-matrix representations of each β-sheet.

Structural renderings and alignments

We used PyMOL [61] to generate 3D renderings and structural alignments of proteins.

Availability and requirements

- **Project name:** BetaSearch
- **Project homepage:** http://www.csse.unimelb.edu.au/~hohkhkh1/betasearch
- **Operating system(s):** Ubuntu Linux 11.10+ (http://www.ubuntu.com)
- **Programming language(s):** Python
- **Other requirements:** A complete listing of Python module dependencies is provided on the project homepage.
- **License:** None
- **Any restrictions to use by non-academics:** BetaSearch can be used free-of-charge by non-academics, provided appropriate citation and credit is given to the authors of this publication.

Additional files

Additional file 1: Supplementary materials (supplement.pdf). This PDF file contains additional information about our experiments as well as the pseudocode for each algorithm referenced in this paper.

Additional file 2: Results from a BLAST query of Top7 chain A (1QYSA-BLAST-results.txt). This plain text file contains the accession numbers of the protein sequences obtained from a BLAST query of Top7 [PDB:1QYS] chain A. The query parameters are defined in this file.

Additional file 3: Results from a BLAST query of streptavidin chain A (1STPA-BLAST-results.txt). This plain text file contains the accession numbers of the protein sequences obtained from a BLAST query of streptavidin [PDB:1STP] chain A. The query parameters are defined in this file.

Additional file 4: Matching β-sheets from Case Study 2 (case-study-2-matches.txt). This plain text file contains all the matching β-sheets in our dataset from the biotin-binding β-residue motif query defined in Figure 5C.

Competing interests

The authors declare that they have no competing interests.

Authors’ contributions

HH, GG, and KR contributed to the design of the algorithms. HH implemented the algorithms and evaluation software, performed the experimental analyses, and prepared the manuscript and figures. MJK contributed to the evaluation of the case studies. All authors read and approved the final manuscript.

Funding

This work was supported by a Victorian Life Sciences Computation Initiative (VLSCI) [grant number VR0127] on its Peak Computing Facility at the University of Melbourne, an initiative by the Victorian Government. HH is supported by a NICTA PhD scholarship. NICTA (National ICT Australia) is funded by the Australian Government's Department of Communications; Information Technology and the Arts; Australian Research Council through Backing Australia's Ability; ICT Centre of Excellence programs.

Acknowledgements

We would like to thank:

- The reviewers for their feedback in improving the quality of this article.
- R. Di Natale, R. Giugno, V. Bonnici, and D. Shasha for their assistance and providing the GraphGrepSX and SING source code.
- M. Moll for his assistance with the LabelHash software.
- A. Stivala for his assistance with the Pro-Origami source code.
- The VLSCI systems support staff for their assistance with our HPC technical requests.

Author details

[1]Department of Computing and Information Systems, The University of Melbourne, Victoria, Australia. [2]National ICT Australia (NICTA), The University of Melbourne, Victoria, Australia. [3]Victorian Life Sciences Computation Initiative (VLSCI), The University of Melbourne, Victoria, Australia.

Received: 10 May 2012 Accepted: 15 June 2012
Published: 30 July 2012

References

1. Kessel A, Ben-Tal N: *Introduction to proteins: structure, function, and motion*. London: CRC Press; 2010.
2. Zaremba SM, Gregoret LM: **Context-dependence of amino acid residue pairing in antiparallel β-sheets.** *J Mol Biol* 1999, **291:**463–479.
3. Parisien M, Major F: **Ranking the factors that contribute to protein β-sheet folding.** *Proteins* 2007, **68:**824–829.
4. Wathen B, Jia Z: **Folding by numbers: primary sequence statistics and their use in studying protein folding.** *Int J Mol Sci* 2009, **10:**1567–1589.
5. Hubbard TJP: **Use of β-strand interaction pseudo-potentials in protein structure prediction and modelling.** In *Proceedings of the 27th Hawaii International Conference on System Sciences*; 1994:336–344.
6. Zhu H, Braun W: **Sequence specificity, statistical potentials, and three-dimensional structure prediction with self-correcting distance geometry calculations of β-sheet formation in proteins.** *Prot Sci* 1999, **8:**326–342.
7. Steward RE, Thornton JM: **Prediction of strand pairing in antiparallel and parallel β-Sheets using information theory.** *Proteins* 2002, **48:**178–191.
8. Rajgaria R, Wei Y, Floudas CA: **Contact prediction for beta and alpha-beta proteins using integer linear optimization and its impact on the first principles 3D structure prediction method ASTRO-FOLD.** *Proteins* 2010, **78:**1825–1846.
9. Bork P, Koonin E: **Protein sequence motifs.** *Curr Opin Struct Biol* 1996, **6:**366–376.
10. Berman HM, Westbrook J, Fend Z, Gilliland G, Bhat TN, Weissig H, Shindyalov IN, Bourne PE: **The protein data bank.** *Nucleic Acids Res* 2000, **28:**235–242.
11. Bella J, Hindle KL, McEwan PA, Lovell SC: **The leucine-rich repeat structure.** *Cell Mol Life Sci* 2008, **65:**2307–2333.
12. Liou YC, Tocilij A, Davies PL, Jia Z: **Mimicry of ice structure by surface hydroxyls and water of a β-helix antifreeze protein.** *Nature* 2000, **406:**322–324.
13. Makabe K, McElheny D, Tereshko V, Hilyard A, Gawlak G, Yan S, Koide A, Koide S: **Atomic structures of peptide self-assembly mimics.** *Proc Natl Acad Sci USA* 2006, **103:**17753–17758.
14. Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ: **Gapped BLAST and PSI-BLAST: a new generation of protein database search programs.** *Nucleic Acids Res* 1997, **25:** 3389–3402.
15. Larkin MA, Blackshields G, Brown NP, Chenna R, McGettigan PA, McWilliam H, Valentin F, Wallace IM, Wilm A, Lopez R, Thompson JD, Gibson TJ, Higgins DG: **ClustalW and ClustalX version 2.** *Bioinformatics* 2007, **23:**2947–2948.
16. Stivala A, Wirth A, Stuckey PJ: **Fast and accurate protein substructure searching with simulated annealing and GPUs.** *BMC Bioinformatics* 2010, **11**.
17. Parisien M: **Les feullets beta dans les protéines. Annotation, comparaison et construction.** *Master's thesis*. Université de Montréal 2005.
18. Amitai G, Shemesh A, Sitbon E, Shklar M, Netanely D, Venger I, Pietrokovski S: **Network analysis of protein structures identifies functional residues.** *J Mol Biol* 2004, **344:**1135–1146.
19. Ullmann JR: **An algorithm for subgraph isomorphism.** *J ACM* 1976, **23:**31–42.
20. Cordella LP, Foggia P, Sansone C, Vento M: **A (sub)graph isomorphism algorithm for matching large graphs.** *IEEE T Pattern Anal* 2004, **10:**1367–1372.
21. Zampelli S: **A constraint programming approach to subgraph isomorphism.** *PhD thesis*. Université catholique de Louvain 2008.
22. Cook SA: **The complexity of theorem-proving procedures.** In *Proceedings of the 3rd ACM Symposium on Theory of Computing*; 1971:151–158.
23. Bonnici V, Ferro A, Giugno R, Pulvirenti A, Shasha D: **Enhancing graph database indexing by suffix tree structure.** In *Pattern Recognition in Bioinformatics, Volume 6282 of* Lecture Notes in Computer Science. Springer; 2010:195–203.
24. Manning CD, Raghavan P, Schütze H: *Introduction to Information Retrieval*. Cambridge: Cambridge University Press; 2008.
25. Giugno R, Shasha D: **GraphGrep: a fast and universal method for querying graphs.** In *Proceedings of the 16th International Conference on Pattern Recognition, 2002, Volume 2*; 2002:112–115.
26. Ferro A, Giugno R, Mongiovi M, Pulvirenti A, Skripin D, Shasha D: **GraphFind: enhancing graph searching by low support data mining.** *BMC Bioinformatics* 2008, **9**.
27. Di Natale R, Ferro A, Giugno R, Mongiovi M, Pulvirenti A, Shasha D: **SING: subgraph search in non-homogeneous graphs.** *BMC Bioinformatics* 2010, **11**.
28. Yan X, Yu PS, Han J: **Graph indexing based on discrimintative frequent structure analysis.** *ACM T Database Syst* 2005, **30**(4):960–993.
29. Cheng J, Ke Y, Ng W, Lu A: **FG-Index: towards verification-free query processing on graph databases.** In *Proceedings of the 2007 ACM SIGMOD International Conference on the Management of Data*; 2007:857–872.
30. Williams DW, Huan J, Wang W: **Graph database indexing using structured graph decomposition.** In *IEEE 23rd International Conference on Data Engineering, 2007*; 2007:976–985.
31. Zhang S, Hu M, Yang J: **TreePi: a novel graph indexing method.** In *IEEE 23rd International Conference on Data Engineering, 2007*; 2007:966–975.
32. Zhao P, Yu JX, Yu PS: **Graph indexing: tree + delta <= graph.** In *Proceedings of the VLDB Endowment*; 2007:938–949.
33. He H, Singh AK: **Closure-tree: an index structure for graph queries.** In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*; 2006:38.
34. Shamir R, Tsur D: **Faster subtree isomorphism.** In *Proceedings of the 5th Israel Symposium on the Theory of Computing Systems*; 1997:267–280.
35. Zou L, Chen L, Yu JX, Lu Y: **A novel spectral coding in a large graph database.** In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT'08)*; 2008:181–192.
36. Moll M, Bryant DH, Kavraki LE: **The LabelHash server and tools for substructure-based functional annotation.** *Bioinformatics* 2011, **27**.
37. Holm L, Sander C: **Mapping the protein universe.** *Science* 1996, **273:**595–602.
38. Holm L, Park J: **DaliLite workbench for protein structure comparison.** *Bioinformatics* 2000, **16:**566–567.
39. Jung J, Lee B: **Protein structure alignment using environmental profiles.** *Prot Sci* 2000, **13:**535–543.
40. Stivala A, Wirth A, Stuckey PJ: **Tableau-based protein substructure search using quadratic programming.** *BMC Bioinformatics* 2009, **10**.
41. Moll M, Bryant DH, Kavraki LE: **The LabelHash algorithm for substructure matching.** *BMC Bioinformatics* 2010, **11**.
42. Kuhlman B, Dantas G, Ireton GC, Varani G, Stoddard BL, Baker D: **Design of a novel globular protein fold with atomic-level accuracy.** *Science* 2003, **302:**1364–1368.

43. Havranek JJ: **Specificity in computational protein design.** *J Biol Chem* 2010, **285:**31095–31099.
44. Liu Y, Kuhlman B: **RosettaDesign server for protein design.** *Nucleic Acids Res* 2006, **34:**W235–W238.
45. Rosado CJ, Kondos S, Bull TE, Kuiper MJ, Law RHP, Buckle AM, Voskoboinik I, Bird PI, Trapani JA, Whisstock JC, Dunstone MA: **The MACPF/CDC family of pore-forming toxins.** *Cell Microbiol* 2008, **10:**1765–1774.
46. Brown WM, Martin S, Chabarek JP, Strauss C, Faulon JL: **Prediction of β-strand packing interactions using the signature product.** *J Mol Model* 2006, **12:**355–361.
47. Cheng J, Baldi P: **Three-stage prediction of protein β-sheets by neural networks, alignments and graph algorithms.** *Bioinformatics* 2005, **21:**75–84.
48. Jeong JK, Berman P, Przytycka TM: **Bringing folding pathways into strand pairing prediction.** In *Lecture Notes in Computer Science, Volume 4645.* Springer; 2007:38–48.
49. Aydin Z, Altunbasak Y, Erdogan H: **Bayesian models and algorithms for protein β-sheet prediction.** In *IEEE/ACM Transactions on Computational Biology and Bioinformatics, Volume 8.* Springer; 2011:395–409.
50. Schierz AC, Soldatova LN, King RD: **Overhauling the PDB.** *Nat Biotechnol* 2007, **25:**437–442.
51. Kabsch W, Sander C: **Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features.** *Biopolymers* 1983, **22:**2577–2637.
52. Stivala AD, Wybrow M, Wirth A, Whisstock JC, Stuckey PJ: **Automatic generation of protein structure cartoons with Pro-origami.** *Bioinformatics* 2011, **27:**3315–3316.
53. Brenner M, Koehl P, Levitt M: **The ASTRAL compendium for sequence and structure analysis.** 2000, **28:**254–256.
54. **Boost v1.42.0.** [http://www.boost.org/users/history/version_1_42_0]
55. **Redis.** [http://www.redis.io]
56. **hiredis.** [https://github.com/antirez/hiredis]
57. **Whoosh Python search library.** [https://bitbucket.org/mchaput/whoosh/wiki/Home]
58. **LabelHash 1.0.2.** [http://labelhash.kavrakilab.org/downloads/python27/LabelHash-1.0.2-Linux64.tar.gz]
59. **SHEBA 3.1.1.** [https://ccrod.cancer.gov/confluence/download/attachments/63341259/sheba-3.1.1.tar.gz]
60. Bakan A, Meireles LM, Bahar I: **ProDy: protein dynamics inferred from theory and experiments.** *Bioinformatics* 2011, **27:**1575–1577.
61. Schrödinger LLC: **The PyMOL Molecular Graphics System, Version 1.3r1.** 2010.

doi:10.1186/1756-0500-5-391
Cite this article as: Ho *et al.*: BetaSearch: a new method for querying β-residue motifs. *BMC Research Notes* 2012 **5**:391.