BIOINFORMATICS ORIGINAL PAPER Vol. 27 no. 1 2011, pages 55–62
doi:10.1093/bioinformatics/btq627

Structural bioinformatics

Advance Access publication November 25, 2010

A dynamic data structure for flexible molecular maintenance and informatics

Chandrajit Bajaj*, Rezaul Alam Chowdhury and Muhibur Rasheed
Department of Computer Science, University of Texas at Austin, Austin, TX, USA

Associate Editor: Anna Tramontano

ABSTRACT

Motivation: We present the 'Dynamic Packing Grid' (DPG), a neighborhood data structure for maintaining and manipulating flexible molecules and assemblies, for efficient computation of binding affinities in drug design or in molecular dynamics calculations.

Results: DPG can efficiently maintain the molecular surface using only linear space and supports quasi-constant time insertion, deletion and movement (i.e. updates) of atoms or groups of atoms. DPG also supports constant time neighborhood queries from arbitrary points. Our results for maintenance of molecular surface and polarization energy computations using DPG exhibit marked improvement in time and space requirements.

Availability: http://www.cs.utexas.edu/~bajaj/cvc/software/DPG.shtml

Contact: bajaj@cs.utexas.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 2, 2010; revised on October 15, 2010; accepted on October 30, 2010

1 INTRODUCTION

Many human functional processes are mediated through the interactions among proteins, a major molecular constituent of our anatomical makeup. A computational understanding of these interactions provides important clues for developing therapeutic interventions related to diseases such as cancer and metabolic disorders. Computational methods such as automated docking through shape and energetic complementarity scoring aim to gain insight and predict such molecular interactions. Docking (Bajaj *et al*., 2009b; Gilson and Zhou, 2007) involves induced complementary fit between flexible protein interfaces. The flexible docking solution space consisting of all relative positions, orientations and conformations of the proteins is searched, and each putative docking is evaluated using combinations of interface complementarity scoring and atomic pairwise charged Coulombic interactions. Also, since proteins function in predominantly watery (solvent) environment, the protein solvation energy also plays an important role in determining intermolecular binding affinities '*in vivo*' for drug screening, as well as in molecular dynamics simulations and in the study of hydrophobicity and protein folding. When computing the solvation energy for molecules, it is crucial to correctly model and sample the molecular surface.

The most common model for molecules is a collection of atoms represented by spherical balls, with radii equal to their van der Waals radii (Duncan and Olson, 1993; Mezey, 1993). The surface of the union of these spheres is known as the van der Waals surface. Accessibility to the solvent, namely the solvent accessible surface (SAS), can be defined as the locus of the center of a 'probe' sphere as it contacts the molecular surface. Usually, the 'probe' is a water molecule modeled as a sphere with radius 1.4 Å. Another definition for molecular surface is as a set of contact and reentrant patches (Richards, 1977), commonly known as the solvent contact surface (SCS), or solvent excluded surface (SES) or simply the molecular surface.

While a number of techniques have been devised for the static construction of molecular surfaces (see e.g. Bajaj *et al*., 2009c for a brief review), not much work has been done on neighborhood data structures for the dynamic maintenance of molecular surfaces under conformational changes and domain movements. Bajaj *et al.* considered limited dynamic maintenance of molecular surfaces based on Non Uniform Rational BSplines (NURBS) descriptions for the patches (Bajaj *et al*., 2003). Eyal and Halperin presented an algorithm based on dynamic graph connectivity that updates the union of balls molecular surface after a conformational change in $\mathcal{O}(\log^2 n)$ amortized time per affected (by this change) atom (Eyal and Halperin, 2005a, b). In this article, we present the *Dynamic Packing Grid* (DPG), a space and time efficient neighborhood data structure that maintains a collection of balls (atoms) in 3-space, allowing a range of spherical range queries and updates for rapid scoring of flexible protein–protein interactions (Bajaj *et al*., 2009a, 2010).

The efficiency of the data structure results from the assumption that the centers of two different balls in the collection cannot come arbitrarily close to each other, which is a natural property of molecules. A consequence of this assumption is that any ball in the collection can intersect at most a constant number of other balls. On a RAM with w-bit words, our DPG data structure can report all balls intersecting a given ball or within $\mathcal{O}(r_{max})$ distance from a given point in $\mathcal{O}(\log\log w)$ time with high probability (w.h.p.), where r_{max} is the radius of the largest ball in the collection. It can also answer whether a given ball is exposed (i.e. lies on the union boundary) or buried within the same time bound. At any time, the entire union boundary can be extracted from the data structure in $\mathcal{O}(m)$ time in the worst case, where m is the number of atoms on the boundary. There are existing techniques like Weiser *et al*. (1998, 1999), which can compute/approximate the exposed atoms and the surface area in the same time bound, but do not allow dynamic updates. On the other hand, DPG supports updates (i.e. insertion/deletion/movement of a ball) in $\mathcal{O}(\log w)$ time w.h.p.[1] The data structure uses linear space.

*To whom correspondence should be addressed.

[1]For an input of size n, an event E occurs w.h.p. (with high probability) if, for any $\alpha \geq 1$ and c independent of n, $Pr(E) \leq 1 - \frac{c}{n^{\alpha}}$.

As we show here, DPGs can be used to maintain both the van der Waals surface and the SCS of a molecule within the performance bounds mentioned above. DPGs can also be used to enable fast energetics calculation by rapidly locating the atoms close to each sampled integration point of the SCS.

Besides protein docking and molecular dynamics, the neighborhood query and surface maintenance of DPG also has potential applications in interactive computer-aided design (CAD) tools developed for *de novo* drug design, protein folding, n-body simulations, etc. All these applications often need to handle extremely large number of atoms or points.

2 THE DYNAMIC PACKING GRID DATA STRUCTURE

Let $M=\{B_1,\ldots,B_n\}$ be a collection of n balls in 3-space with c_i and r_i being the center and radius, respectively, of B_i, $i\in[1,n]$. Let $r_{max}=\max_i\{r_i\}$ and let $d_{min}=\min_{i,j}\{d(c_i,c_j)\}$, where $d(c_i,c_j)$ is the Euclidean distance between c_i and c_j.

We describe the *packing grid data structure* for maintaining M efficiently under the following set of queries and updates.

Queries

(1) INTERSECT(c,r): Returns all balls intersecting $B=(c,r)$.

(2) RANGE(p,δ): Returns all balls with centers within distance δ of point p. We assume that $\delta=O(r_{max})$.

(3) EXPOSED(c,r): Returns *true* if the ball $B=(c,r)\in M$ contributes to the boundary of the union of the balls in M.

(4) SURFACE( ): Returns the outer boundary of the union of the balls in M. If there are multiple disjoint outer boundaries defined by M, the routine returns one of them.

Updates

(1) ADD(c, r): Adds a new ball $B=(c,r)$ to the set M.

(2) REMOVE(c, r): Removes the ball $B=(c,r)$ from M.

(3) MOVE(c_1, c_2, r): Moves the ball with center c_1 and radius r to a new center c_2.

We assume that the following holds at all times.

ASSUMPTION 2.1. *If $d_{\min}$ is the minimum Euclidean distance between the centers of any two balls in M, then $r_{\max}=\mathcal{O}(d_{\min})$.*

In general, a ball in a collection of n balls in 3-space can intersect $\Theta(n)$ other balls in the worst case, and it has been shown by Clarkson *et al.* (1990) that the boundary defined by the union of these balls has a worst case combinatorial complexity of $\Theta(n^2)$. However, if M is a 'union of balls' representation of the atoms in a molecule, then assumption 2.1 holds naturally (Halperin and Overmars, 1994; Varshney *et al.*, 1994), and as proved by Halperin and Overmars (1994), both complexities improve by a factor of n. The following theorem (see Bajaj *et al.*, 2010 for a proof) states the consequences of the assumption.

THEOREM 2.1 [Theorem 2.1 in (Halperin and Overmars, 1994), slightly modified]. *Each ball in M intersects at most $216\cdot\left(r_{max}/d_{min}\right)^3=\mathcal{O}(1)$ other balls in M and the combinatorial complexity of the boundary of the union of the balls is $\mathcal{O}\left(\left(r_{max}/d_{min}\right)^3\cdot n\right)=\mathcal{O}(n)$.*

Table 1. Time complexities of the operations supported by the packing grid data structure

Operations	Time Complexity (w.h.p.)	
	Assuming $t_q=\mathcal{O}(\log\log w)$ $t_u=\mathcal{O}(\log w)$	Assuming $t_q=\mathcal{O}(\log\log n)$ $t_u=\mathcal{O}\left(\frac{\log n}{\log\log n}\right)$
RANGE, INTERSECT, EXPOSED	$\mathcal{O}(\log\log w)$	$\mathcal{O}(\log\log n)$
ADD, REMOVE, MOVE	$\mathcal{O}(\log w)$	$\mathcal{O}\left(\frac{\log n}{\log\log n}\right)$
SURFACE	$\mathcal{O}$(#balls on surface) (worst case)	

ASSUMPTIONS: (*i*) RAM with w-bit Words,
(*ii*) Collection of n balls,
(*iii*) $\delta=\mathcal{O}(r_{\max})$ and, (*iv*) $r_{\max}=\mathcal{O}$(minimum distance between two balls)

Therefore, as Theorem 2.1 suggests, one should be able to handle M more efficiently if assumption 2.1 holds. The efficiency of our data structure, listed in Table 1, also depends partly on this assumption.

2.1 Preliminaries

Before we describe our data structure, we present some definitions in order to simplify the exposition.

DEFINITION 2.1 [r-grid, grid-cell, grid-line and grid-plane]. *An r-grid is an axis-parallel infinite grid structure in 3-space consisting of cells of size $r\times r\times r$ ($r\in\mathbb{R}$) with the* root *(i.e. the corner with the smallest x, y, z coordinates) of one of the cells coinciding with the origin of the Cartesian coordinate axes. The grid cell that has its root at Cartesian coordinates (ar,br,cr) (where $a,b,c\in\mathbb{Z}$) is referred to as the (a,b,c,r)-cell or simply as the (a,b,c)-cell when r is clear from the context. The* (b,c,r)-line *(where $b,c\in\mathbb{Z}$) in an r-grid consists of all (x,y,z,r)-cells with y and z fixed to b and c, respectively. The* (c,r)-plane *(where $c\in\mathbb{Z}$) in an r-grid consists of all (x,y,z,r)-cells with z fixed to c.*

The proof of the following lemma is straightforward.

LEMMA 2.1. *If M is stored in the $2r_{max}$-grid G and if Assumption 2.1 holds, then*

(*i*) *Each grid-cell in G contains the centers of at most $\mathcal{O}(1)$ balls.*
(*ii*) *Each ball in M intersects at most eight grid-cells in G.*
(*iii*) *For a given ball $B\in M$ with center in grid-cell C, the center of each ball intersecting B lies either in C or in one of the 26 grid-cells adjacent to C.*
(*iv*) *The number of* non-empty *grid-cells in G is at most n, and the same bound holds for grid-lines and grid-planes.*

At the heart of our data structure is a fully dynamic one dimensional integer range reporting data structure for word RAM described by Mortensen *et al.* (2005). Their data structure maintains a set S of integers under updates (i.e. insertions and deletions), and answers queries of the form QUERY(l,h), which reports any or all points in S in the interval $[l,h]$. The following theorem (proved in Mortensen *et al.*, 2005) summarizes the performance bounds of the data structure which are of interest to us.

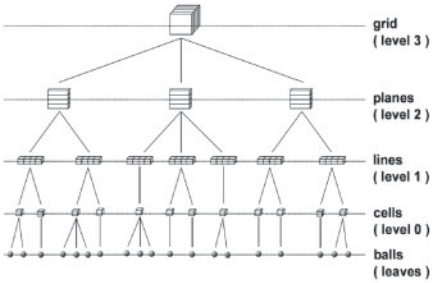

Fig. 1. Hierarchical structure of DPG.

THEOREM 2.2. *On a RAM with w-bit words, the fully dynamic one dimensional integer range reporting problem can be solved in linear space, and w.h.p. bounds of* $\mathcal{O}(t_u)$ *and* $\mathcal{O}(t_q+k)$ *on update time and query time, respectively, where k is the number of items reported, and*

(*i*) $t_u=\mathcal{O}(\log w)$ *and* $t_q=\mathcal{O}(\log\log w)$ *using the data structure in Mortensen et al. (2005); and*
(*ii*) $t_u=\mathcal{O}(\log n/\log\log n)$ *and* $t_q=\mathcal{O}(\log\log n)$ *using the data structure in (Mortensen et al., 2005) for small w and a fusion tree (Fredman and Willard, 1993) for large w.*

2.2 Description (layout) of the packing grid data Structure

We are now in a position to present the DPG data structure. DPG represents the entire 3-space as a $2r_{max}$-grid, and maintains the non-empty grid-planes, grid-lines and grid-cells. Note that a grid component (i.e. cell, line or plane) is non-empty if it contains the center of at least one ball in M. The data structure can be described as a tree with five levels: 4 internal levels (levels 3, 2, 1 and 0) and an external level of leaves (see Fig. 1). The description of each level follows (further details are available in Bajaj *et al.*, 2010).

2.2.1 The leaf level 'Ball' data structure (DPG_{-1}*)* The data structure stores the center $c=(c_x,c_y,c_z)$ and the radius r of the given ball B. It also includes a Boolean flag *exposed* which is set to *true* if B contributes to the outer boundary of the union of the balls in M, and *false* otherwise. The 3D arrangement of the spheres $B\cup\mathcal{N}(B)$, where $\mathcal{N}(B)$ is the set of balls intersecting B, divides the boundary of B into spherical patches, some of which are exposed, that is they contribute to the union boundary. We store all exposed patches (if any) of A in a set F of size $\mathcal{O}(1)$, and with each patch $f\in F$ we store pointers to the data structures of $\mathcal{O}(1)$ other balls that share edges with f and also the identifier of the corresponding patch on each ball.

2.2.2 The level 0 'Grid-Cell' data structure (DPG_0*)* The 'grid-cell' data structure stores the root (see Definition 2.1) (a,b,c) of the grid-cell it corresponds to. A grid-cell maintains a list of pointers to data structures of the $\mathcal{O}(1)$ balls whose centers lie inside the cell. Since we create 'grid-cell' data structures only for non-empty grid-cells, there will be at most n (and possibly $\ll n$) such data structures.

2.2.3 The level 1 'Grid-Line' data structure (DPG_1*)* We create a 'grid-line' data structure for a (b,c)-line provided it contains at least one non-empty grid-cell. Each (a,b,c)-cell lying on this line is identified with the unique integer a, and the identifier of each such non-empty grid-cell is stored in an integer range search data structure *RR* as described in Section 2.1 (see Theorem 2.2). We augment *RR* to store the pointer to the corresponding 'grid-cell' data structure with each identifier it stores.

2.2.4 The level 2 'Grid-Plane' data structure (DPG_2*)* A 'grid-plane' data structure is created for a c-plane provided it contains at least one non-empty grid-line. Similar to the 'grid-line' data structure, it identifies each non-empty (b,c)-line lying on the c-plane with the unique integer b, and stores the identifiers in a range reporting data structure *RR*.

2.2.5 The level 3 'Grid' data structure (DPG_3*)* This data structure maintains the non-empty grid-planes in an integer range reporting data structure *RR* in a similar way where each c-plane is identified by the unique integer c. The 'grid' data structure also stores a *surface-root* pointer which points to the 'ball' data structure of an arbitrary exposed ball in M.

We have the following lemma (proved in Bajaj *et al.*, 2010) on the space usage of the data structure.

LEMMA 2.2. *Let Assumption 2.1 hold. Then the packing grid data structure storing M uses* $\mathcal{O}(n)$ *space.*

2.3 Queries and updates

The queries and updates supported by the data structure are implemented as follows.

2.3.1 Queries

(1) Range(p, δ): Let $p=(p_x,p_y,p_z)$. First we invoke the function QUERY(l,h) of the range reporting data structure *RR* under the level 3 grid data structure with $l=\lfloor(p_z-\delta)/(2r_{max})\rfloor$ and $h=\lfloor(p_z+\delta)/(2r_{max})\rfloor$. This query returns a set S_2 of non-empty c-planes represented as pointers to level 2 grid-plane data structures. Then, for each c-plane, we perform similar queries under the corresponding level 2 data structure to obtain the set S_1 of non-empty grid-lines. Again, querying under each grid-line data structure produces the set S_0 containing non-empty grid-cells. Finally, for each cell in S_0, we collect and return each ball whose center lies within distance δ from p.

The correctness of the function follows trivially since it queries a region in 3-space, which includes the region covered by a ball of radius δ centered at p. Also, assuming $r_{max}=\mathcal{O}(d_{min})$ (i.e. Assumption 2.1) and $\delta=\mathcal{O}(r_{max})$, the complexity reduces to $\mathcal{O}(t_q)$. Details can be found in (Bajaj *et al.*, 2010).

(2) Intersect(c, r): Let $B=(c,r)$ be the given ball. First, we call RANGE(c, $r+r_{max}$) and collect the output in set S. From S we remove the data structure of each ball that does not intersect B, and return the resulting (possibly reduced) set.

The correctness follows from basic geometry and the correctness of RANGE. Under Assumption 2.1, this function runs in $\mathcal{O}(t_q)$ time.

(3) Exposed(c, r): Let $B=(c,r)$ be the given ball. We locate B's data structure by calling RANGE$(c,0)$, and return the value stored in its *exposed* field. Clearly, the function takes $\mathcal{O}\left(t_q+\left(r_{max}/d_{min}\right)^3\right)$ time (w.h.p.), which reduces to $\mathcal{O}(t_q)$ under Assumption 2.1.

(4) SURFACE( ): The *surface-root* pointer under the level 3 'grid' data structure points to the 'ball' data structure of a ball B on the union boundary of M. We scan the set F of exposed faces of B, and using the pointers to other exposed balls stored in F we perform a depth-first traversal of all exposed balls in M and return the exposed faces on each such ball. Let m be the number of balls contributing to the union boundary of M. Then according to Theorem 2.1, the depth-first search takes $\mathcal{O}\left(\left(r_{max}/d_{min}\right)^3 \cdot m\right)$ time in the worst case, which reduces to $\mathcal{O}(m)$ under Assumption 2.1.

2.3.2 Updates

(1) Add(c, r): Let $c=(c_x,c_y,c_z)$ and let $c'_u = \left\lfloor \frac{c_u}{2r_{max}} \right\rfloor$, where $u \in \{x,y,z\}$. Let G be the grid data structure. If G does not exist, then create and initialize G. Then, first we create and initialize a data structure B and add to M. Then, we query the range reporting data structure $G.RR$ to locate the data structure P for the c'_z-plane. If P does not exist, create and initialize P, and insert c'_z along with a pointer to P into $G.RR$. Similar steps are taken for the grid-line and then the grid-cell data structures to identify the (c'_x,c'_y,c'_z)-cell C and add B to C. We then use the INTERSECT query to identify $\mathcal{N}(B)$, the set of balls intersecting B. Finally we update the arrangement of each ball in $B \cup \mathcal{N}(B)$, list exposed faces on each ball and update the *surface-root* pointer if necessary.

Observe that the introduction of a new ball may affect the surface exposure of only the balls it intersects (by burying some/all of them partly or completely), and no other balls. Hence, updating the arrangements of the balls in $B \cup \mathcal{N}(B)$ (in addition to those in earlier steps) are sufficient to maintain the correctness of the entire data structure. The ADD function terminates in $\mathcal{O}(t_u)$ assuming $r_{max} = \mathcal{O}(d_{min})$. Detailed analysis is in (Bajaj *et al*., 2010).

(2) REMOVE(c, r): This function is symmetric to the ADD function, and has exactly the same asymptotic time complexity. Hence, we do not describe it here.

(3) MOVE(c_1, c_2, r): This function is implemented in the obvious way by calling REMOVE(c_1, r) followed by ADD(c_2, r). It has the same asymptotic complexity as the two functions above. Therefore, we have the following theorem.

THEOREM 2.3. *Let M be a collection of n balls in 3-space as defined in Theorem 2.1, and let Assumption 2.1 hold. Let t_q and t_u be as defined in Theorem 2.2. Then the packing grid data structure storing M on a word RAM:*

- (*i*) *uses $\mathcal{O}(n)$ space;*
- (*ii*) *supports updates (i.e. insertion/deletion/movement of a ball) in $\mathcal{O}(t_u)$ time w.h.p.;*
- (*iii*) *reports all balls intersecting a given ball or within $\mathcal{O}(r_{max})$ distance from a given point in $\mathcal{O}(t_q)$ time w.h.p., where r_{max} is the radius of the largest ball in M; and*
- (*iv*) *reports whether a given ball is exposed or buried in $\mathcal{O}(t_q)$ time w.h.p., and returns the entire outer union boundary of M in $\mathcal{O}(m)$ worst-case time, where m is the number of balls on the boundary.*

2.4 Molecular surface maintenance using DPG

In this section, we briefly describe applications of the packing grid data structure for efficient maintenance of molecular surfaces.

2.4.1 Maintaining van der Waals surface of molecules Each atom is simply treated as a ball with a radius equal to the van der Waals radius of the atom see (Batsanov, 2001) for a list of van der Waals radius of different atoms).

2.4.2 Maintaining Lee-Richards (SCS/SES) surface For the efficient maintenance of the Lee-Richards surface of a molecule within the performance bounds given in Table 1, we maintain two packing grid data structures: DPG and DPG′. The DPG data structure keeps track of the patches on the Lee-Richards surface, and DPG′ is used for detecting intersections among concave patches.

Before adding an atom to DPG, we increase its radius by r_s, where r_s is the radius of the rolling solvent atom. The DPG data structure keeps track of all solvent exposed atoms, i.e. all atoms that contribute to the outer boundary of the union of these enlarged atoms. Theorem 2.1 implies that each atom in DPG contributes $\mathcal{O}(1)$ patches to the Lee-Richards surface, and the insertion/deletion/movement of an atom results in local changes of only $\mathcal{O}(1)$ patches. We can modify DPG to always keep track of where two or three of the solvent exposed atoms intersect, and once we know the atoms contributing to a patch we can easily compute the patch in $\mathcal{O}(1)$ time (Bajaj *et al*., 2003).

The Lee-Richards surface can self-intersect in two ways: (i) a toroidal patch can intersect itself and (ii) two different concave patches may intersect (Bajaj *et al*., 2003). The self-intersections of toroidal patches can be easily detected from DPG. In order to detect the intersections among concave patches, we maintain the centers of all current concave patches in DPG′, and use the INTERSECT query to find the concave patch (if any) that intersects a given concave patch.

2.5 Energetics computation using DPG

Generally, the solvation energy G_{sol} of a molecule is decomposed into three components, namely, G_{cav} (the energy to form cavity in the solvent), G_{vdw} (the solute-solvent van der Waals interaction energy), and G_{pol} (the polarization energy or the electrostatic potential energy change due to the solvation). The first two terms G_{cav} and G_{vdw} are linearly related to the solvent accessible surface area Ω_{SAS}. The last term, G_{pol}, can be approximated using the *Generalized Born* (GB) theory as introduced by Still *et al*. (1990).

$$G_{pol} = -\frac{\tau}{2} \sum_{i,j} \frac{q_i q_j}{\sqrt{r_{ij}^2 + R_i R_j e^{-r_{ij}^2/4R_iR_j}}}, \tag{2.1}$$

where $\tau = 1 - \frac{1}{\epsilon}$, and R_i is the effective Born radius of atom i (see Fig. 2a). Either of Equations 2.2 or 2.3 can be used as discrete approximation of R_i^{-1} (Bajaj and Zhao, 2010).

$$R_i^{-1} = \frac{1}{4\pi} \sum_{k=1}^{N} w_k \frac{(\mathbf{r}_k - \mathbf{x}_i) \cdot \mathbf{n}(\mathbf{r}_k)}{|\mathbf{r}_k - \mathbf{x}_i|^4}, \tag{2.2}$$

$$R_i^{-3} = \frac{1}{4\pi} \sum_{k=1}^{N} w_k \frac{(\mathbf{r}_k - \mathbf{x}_i) \cdot \mathbf{n}(\mathbf{r}_k)}{|\mathbf{r}_k - \mathbf{x}_i|^6}, \tag{2.3}$$

where the $\mathbf{r}_k$'s are N carefully chosen integration points on the boundary of the molecule, and w_k is a weight assigned to $\mathbf{r}_k$ to ensure higher order of accuracy for small N (see Fig. 2b). Other methods have used volume integrals (Tjong and Zhou, 2007) or

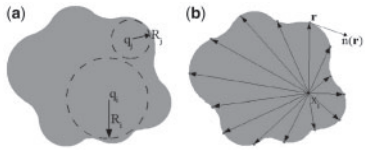

Fig. 2. (**a**) G_{pol} is computed based on Born Radii and charges of each atom pair, (**b**) Born Radii of an atom can be approximated based on integration points, shown as red dots, sampled on the surface.

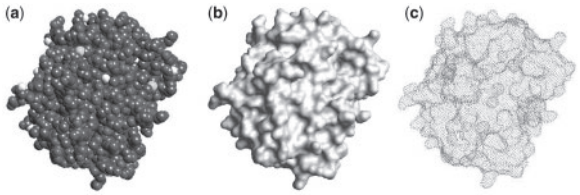

Fig. 3. Gaussian integration points (**c**) on the surface of nuclear transport factor 2 (1A2K) computed after generating a smooth surface (**b**) from the collection of balls model (**a**).

integrals over bonded and non-bonded atom pairs (Qiu *et al*., 1997) to approximate Born Radii.

The non-polar terms G_{cav} and G_{vdw} can be computed directly from the SAS area Ω_{SAS} of the molecule. The SAS of the molecule can be extracted in $\mathcal{O}(\widetilde{m}\log w)$ time and $\mathcal{O}(\widetilde{m})$ space using a DPG data structure, where $\widetilde{m}$ is the number of atoms in the molecule. The DPG data structure outputs the SAS as a set of spherical and toroidal patches, and we add up the area of each patch in order to calculate Ω_{SAS}.

2.5.1 Discrete approximation of Born Radii In order to approximate the polar term G_{pol}, first we need to approximate the Born radius R_i of each atom i. We compute the SES as A-spline patches, produce a quality improved meshing of the surface and sample integration points and their weights following (Bajaj and Zhao, 2010) (see Figure 3) and then use Equation (2.2) to approximate R_i. But observe that the direct computation of R_i requires $O(n^2)$ time, where n is the number of atoms and assuming that the number of sampled integration points is also $O(n)$. However, since the terms in the summation diminish very fast with the increase of distance, distance cutoffs can be used to approximate it.

Given the set of atoms $\mathcal{A}$, the set of integration points $\mathcal{Q}$ sampled on the surface, and two user-defined parameters $\alpha, \delta > 0$, for every integration point $q \in \mathcal{Q}$, we place each atom $a \in \mathcal{A}$ in one of the following three categories based on the distance d between q and the center of a: (*i*) *near* ($d \leq \delta$), (*ii*) *mid-way* ($\delta < d \leq \alpha\delta$) and (*iii*) *far* ($\alpha\delta < d$). Figure 4 shows an example in 2D. For the near categories, the computation is performed exactly. For the midway category, clusters of atoms and integration points are viewed as pseudo-atoms and pseudo-integration points, and hence a coarse computation is performed. For the far category, a single average distance and a single average weighted normal is used for all pairs of clusters.

Separate DPG data structures are used to store the atoms, integration points, pseudo-atoms and pseudo-integration points. DPG is used both for identifying the near, midway and far

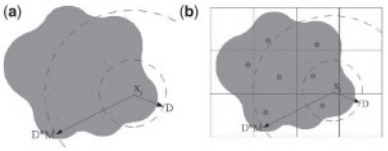

Fig. 4. (**a**) A simple 2D example depicting definition of near, medium and far atoms (centers shown as green dots) from a particular integration point x_i. In the example, two atoms are near, seven are medium and three are far. (**b**) After clustering using hierarchical DPG, each cell contains a pseudoatom (centers shown as blue circles). Now two atoms are near, three clusters are medium and two clusters are far.

atoms/pseudoatoms as well as for clustering (see Bajaj *et al*., 2010 for details).

Assuming that $\widetilde{m}_{\widetilde{\delta}}$ is an upper bound on the number of atoms within distance $\widetilde{\delta}$ from any given point in space, the time spent for computing all R_i's is $\mathcal{O}(N\log\log w + N\widetilde{m}_{\widetilde{\delta}})$, which reduces to $\mathcal{O}(N\log\log w)$ since $\widetilde{m}_{\widetilde{\delta}}$ is a constant (though could be quite large) for constant $\widetilde{\delta}$. Once all R_i's are computed, G_{pol} can be computed using Equation (2.1) in $\mathcal{O}(\widetilde{m}^2)$ time in the worst case. The space usage is $\mathcal{O}(\widetilde{m} + N\widetilde{m}_{\widetilde{\delta}})$ which is $\mathcal{O}(\widetilde{m} + N)$ for constant $\widetilde{\delta}$.

2.6 Maintenance of flexible molecules

Suppose we are given a flexible molecule decomposed into several (mostly) rigid domains which interact either through connected chain segments or large interfaces. We refer to these chain segments and interfaces as connectors. Domains may move with respect to each other through motions applied to the connectors. Two domains connected by at least one connector may undergo bending motion applied to some hinge point around some hinge axis. If they are connected by only one connector, a twisting motion can also be applied to the connector by updating torsion angles along its backbone. If two domains share a large interface area, they may undergo a shearing motion with respect to each other. However, though domains are mostly rigid they may have flexible loops and side chains on their surfaces.

We maintain a separate packing grid data structure $\mathcal{P}_i$ for each domain $\mathcal{D}_i$. If two domains $\mathcal{D}_i$ and $\mathcal{D}_j$ are connected and $i < j$, the set S_{ij} of all connectors between these two domains are included in $\mathcal{P}_i$, and a transformation matrix M_{ij} is kept with $\mathcal{P}_i$ that describes the exact location and orientation of the grid structure of $\mathcal{P}_j$ with respect to that of $\mathcal{P}_i$. Whenever some motion is applied to the connectors in S_{ij}, we update $\mathcal{P}_i$ in order to reflect the changes in the locations of the atoms in these connectors, and also update M_{ij} in order to reflect the new relative position and orientation of $\mathcal{P}_j$ with respect to $\mathcal{P}_i$. The complexities of these operations are presented in the following lemma proved in (Bajaj *et al*., 2010).

LEMMA 2.3. *The surface of a flexible molecule decomposed into (mostly) rigid domains can be maintained using packing grid data structures so that*

(*i*) *updating for a bending/shearing/twisting motion applied between two domains takes* $\mathcal{O}(1+\overline{m}\log w)$ *time (w.h.p.), where* $\overline{m}$ *is the number of atoms in the connectors between the two domains;*

(*ii*) *updating the conformation of a flexible loop or a side chain on the surface of a domain takes* $\mathcal{O}(\widetilde{m}\log w)$ *time (w.h.p.), where* $\widetilde{m}$ *is the number of atoms affected by this change; and*

(*iii*) *generating the surface of the entire molecule requires* $\mathcal{O}(\widehat{m}\log w)$ *time (w.h.p.), where* $\widehat{m}$ *is the sum of the number of atoms on the surface of each domain.*

3 RESULTS AND DISCUSSIONS

The performance of the basic functions of DPG are reported in Section 3.2. Sections 3.3 and 3.4, respectively, analyzes performance of DPG in molecular surface maintenance and energetics calculation.

3.1 Implementation details

In our current implementation, instead of the 1D integer range-reporting data structure presented in (Mortensen *et al*., 2005), we have implemented a much simpler data structure that supports both updates and distance queries in expected $\mathcal{O}(\log w)$ time and uses linear space. Since w is usually not more than 64, for most practical purposes a $\mathcal{O}(\log w)$ query time should be almost as good as $\mathcal{O}(\log\log w)$ time. This data structure builds on binary search trees, dynamic perfect hashing and y-fast trees. However, instead of dynamic perfect hashing we used 'cuckoo hashing' (Rasmus and Flemming, 2004) since it is much simpler, and still supports lookups in $\mathcal{O}(1)$ worst case time, and updates in expected $\mathcal{O}(1)$ time.

3.2 Performance analysis of updates and queries

To measures the performance of the update and query functions of DPG, we use more than 180k quadrature points, generated for energetics computations by sampling uniformly at random on the surface of PSTI (a variant of human pancreatic trypsin inhibitor: 1HPT.pdb) after protonation using PDB2PQR (Dolinsky *et al*., 2004). Experiments are performed on a 3 GHz 2× dual core (only one core was used) AMD Opteron 2222 processor with 4 GB RAM. Please refer to (Bajaj *et al*., 2010) for details of the experiment. Table 2 shows the results of this experiment. The time required is $\mathcal{O}(\log w+K)$ where K is the size of the output or in this case, the number of points returned. The last column of the table shows that as the point set becomes denser, the efficiency of the data structure remains almost the same.

Table 3 reports the performance of update functions of DPG's range reporting data structure. Four different macromolecules were used, and for each of them all atoms were first randomly inserted into the data structure followed by the random deletion of all atoms. The reported insertion and deletion times are averages of four such independent runs. The average time for a single insertion/deletion was never more than 5 μs.

Table 2. Performance of the QUERY function of packing grid

Quadr. Points	Query Distance (Å)	Average Time (ms)/Query	Average number of Points Returned (k/ms)
45654	2 | 4 | 8	0.31 | 0.57 | 1.42	0.38 | 1.37 | 3.14
91309	2 | 4 | 8	0.59 | 1.14 | 2.80	0.38 | 1.43 | 3.31
136963	2 | 4 | 8	0.97 | 1.85 | 4.44	0.34 | 1.32 | 3.27
182618	2 | 4 | 8	1.30 | 3,22 | 5.86	0.38 | 1.31 | 3.30

We randomly assign each of the 182 618 points to one of four groups and thus obtain four approximately equal-sized groups. We then run queries from the atom centers (100 queries per atom) on group 1, merge groups 1 and 2 and run queries on this merged group, and so on.

3.3 Performance of molecular surface maintenance

We compared the performance of DPG with the 3D hashing used by (Eyal and Halperin, 2005a, b) in producing and maintaining molecular surfaces. We used the same implementation of 3D arrangement and surface generation (Eyal and Halperin, 2005b), but switched between the two different range query data structures. We measured the space and time requirements for generating the surface of various macromolecules. To verify scalability, multiple chains of the same protein were inserted. For virus capsids, as multiple chains are inserted, not only the number of atoms increases but also the overall structure becomes sparser. The results of this experiment are reported in Table 4. It is clear that the space requirement of the DPG is linear in the number of atoms. The difference in space requirement becomes more pronounced for larger and sparser structures. Also, its running times are comparable with that of 3D hash. Though 3D hash performs insertions and queries in optimal constant time, using too much memory can adversely affect its running time. For example, in the case of RDV P3 with four chains, 3D hash operations run slower than DPG range reporting operations. We believe that this slowdown is due to page faults caused by excessive space requirement of 3D hash.

3.4 Performance of Born Radii and polarization energy calculation

A parallel implementation of the approximation scheme described in Section 2.5.1 was applied to compute the Born Radii, which were used to compute the polarization energy G_{Pol}. The experiments were performed on the RANGER cluster, on a single node with 16 cores.

First, three different approximations were performed by varying the δ parameter for the molecules in ZDock Benchmark 2.0 (Mintseris *et al*., 2005). We shall refer to these as *DPG_GB_g_x*, where $\delta = xD, x \in 0.5, 0.75, 1.0$ and D is the dimension of a cell in DPG, and it means that a $g \times g \times g$ grid was used to generate the surface and integration points on the surface. Both D and α are automatically selected based on the size of the molecules. For each atom i of a molecule, the approximation error is defined as $\varepsilon_i = \frac{|(R_i^{exact} - R_i^{dpg})|*100}{R_i^{exact}}$, where R_i^{dpg} and R_i^{exact} are the Born Radii of atom i approximated using DPG-based scheme and by exact (full pairwise) evaluation of Equation (2.2), respectively. The approximation error

Table 3. Insertion and deletion times of our current packing grid implementation

Molecule (PDB File)	Number of Atoms	Average Insert Time (μs)	Average Delete Time (μs)
GroEL (1GRL)	29 274	3.3	4.0
RDV P8 (1UF2: P)	193 620	3.9	4.4
RDV P3 (1UF2: A)	459 180	3.9	4.6
Dengue (1K4R)	545 040	4.0	4.5

The results are averages of four runs. In each run, all atom centers are randomly inserted into the data structure followed by random deletion of all atom centers.

Table 4. Comparison of the performance of the 3D range reporting data structure used by DPG and the 3D hash table used in (Eyal and Halperin, 2005b).

Molecule (PDB File)	Number of Chains	Number of Atoms	Number of Cells (k)		Time (s)	
			DPG	3D hash	DPG	3D hash
1I3Q	1	11 114	4.68	45.18	17.36	16.23
2GLS	1	3636	1.44	9.18	5.43	5.06
	5	18 180	7.28	41.40	37.10	34.80
2BG9	1	2991	1.20	10.75	4.44	4.29
	5	14 955	6.03	31.20	24.31	22.95
1UF2: Chain P (RDV P8)	1	3227	1.35	9.26	4.47	4.23
	2	6454	2.74	1124.04	9.23	8.56
	4	12 908	5.47	4426.11	19.36	18.14
	8	25 816	10.98	6332.16	45.22	44.44
1UF2: Chain A (RDV P3)	1	7653	3.23	38.76	10.99	10.23
	2	15 306	6.46	927.44	22.73	21.44
	3	22 959	9.74	1992.75	40.48	39.62
	4	30 612	12.99	2591.70	119.28	128.37
1K4R: Chains A and B	2	6056	2.62	20.70	8.46	7.71
	4	12 112	5.24	138.60	17.56	16.52
	6	18 168	7.85	333.06	33.73	32.62

for a molecule is the average of the ε_i's. Figure 5a reports the approximation errors for each molecule. It is clear that a larger 'near' band results in lower error. On the other hand, Figure 5b shows the speedup for each approximation, where speedup is defined as (time taken by exact computation)/(time taken by DPG-based computation). Though there is a clear speed/accuracy trade-off, it only underscores the efficacy and flexibility of the scheme. For example, *DPG_GB*_128_1.0 is almost 50 times faster than the naive pairwise computation with only 2.41% error.

In Figure 6, we report the error of G_{pol} computation where, for each molecule, the error is defined as $\frac{|(G^{exact}-G^{dpg})|*100}{G^{exact}}$, where G^{exact} and G^{dpg} are, respectively, the G_{pol} computed using R_i^{exact} and R_i^{dpg} for each atom i of the molecule. G_{pol} errors are much lower than the Born Radii errors because the integral of the G_{pol} formulation also falls off with distance and hence accuracy of G_{pol} is more dependent on the accuracy of the Born radii of atoms near the surface. In Table 5, the Born Radii of all atoms of all molecules are grouped into five bins based on R_i^{exact}. It is easy to verify that Born Radii computation errors for the atoms near the surface (having lower values of Born Radii) are indeed much lower. Another notable aspect from the results in Figure 6 is that some of the molecules, specially 1*PPE_l_b*, the G_{pol} error is considerably higher. We found that this tend to happen for molecules which are very small (for example, 1*PPE_l_b* has only 436 atoms) or very flat, in other words does not have much in the 'far' band. Our scheme for computing partial sums for 'far' bands seem to overestimate in such cases.

We also computed the Born Radii and G_{pol} for the same set of molecules using Amber (Case *et al*., 2005) and GBr^6 (Tjong and Zhou, 2007) on the same computing cluster using the same number of nodes and cores. The results in Figure 7a show that DPG-based implementations, are much faster than GBr^6 and are comparable to Amber. In Figure 7b, we report the ratio of the Born Radii

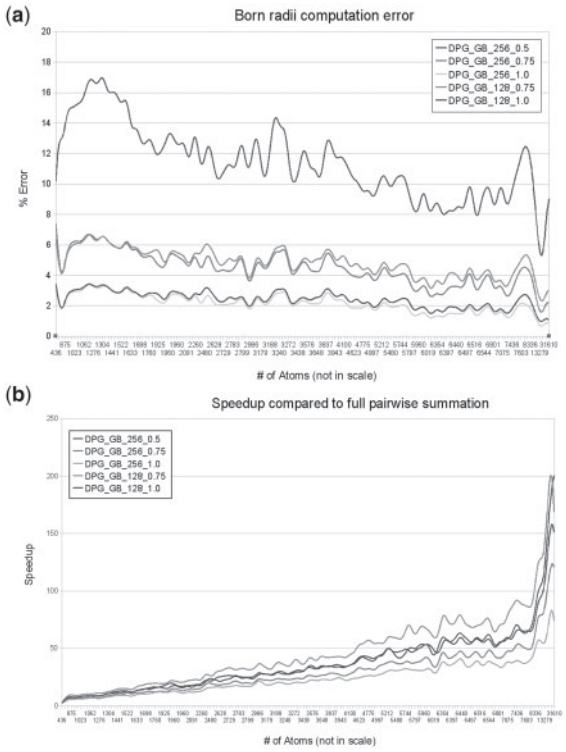

Fig. 5. (**a**) Comparison of the approximation errors for Born Radii computation at various levels of approximation. Average percentage error across all molecules for the schemes are 11.42, 4.44, 2.16, 4.84 and 4.41 (in the order shown in legend). (**b**) Comparison of the speedup (with respect to the exact implementation) for Born Radii computation at various levels of approximation. Average speedup across all molecules for the schemes are 47.96, 37.71, 30.63, 59.97 and 47.51 (in the order shown in legend). Figures appear in color in the online version of the paper.

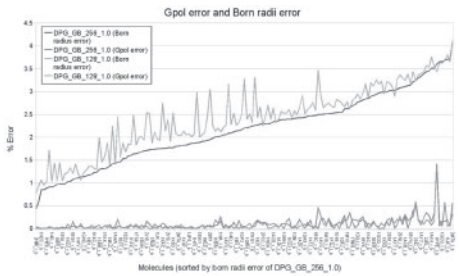

Fig. 6. Approximation errors for G_{pol} computation. The average G_{pol} error across all molecules are 0.09 and 0.1, respectively for DPG_GB_256_1.0 and DPG_GB_128_1.0. Note that G_{pol} errors are much lower than Born Radii errors.

computation time of DPG and Amber, sorted in increasing size of molecules. It is clear that DPG gets better as the size increases and outperforms Amber in a few cases. So, we experimented with Amber, GBr^6 and DPG for a very large molecule, the Cucumber Mosaic Virus (CMV) capsid, consisting 509*K* atoms. DPG completed in only 22 s, while Amber needed 172 s and GBr^6 needed about 3.6 h.

As G_{pol} obtained using different formulations often vary a lot, we decided to compare the consistency instead of the exact values.

Table 5. Distribution of errors for different ranges of Born Radii. Clearly, error is lower for atoms near the surface (smaller Born Radii).

Range of Born Radii	Number of atoms in range	Average % error
[0, 2]	17 580	0.83
(2, 4]	63 101	1.85
(4, 7]	61 640	3.82
(7, 10]	38 796	6.74
(10, …]	112 765	10.16

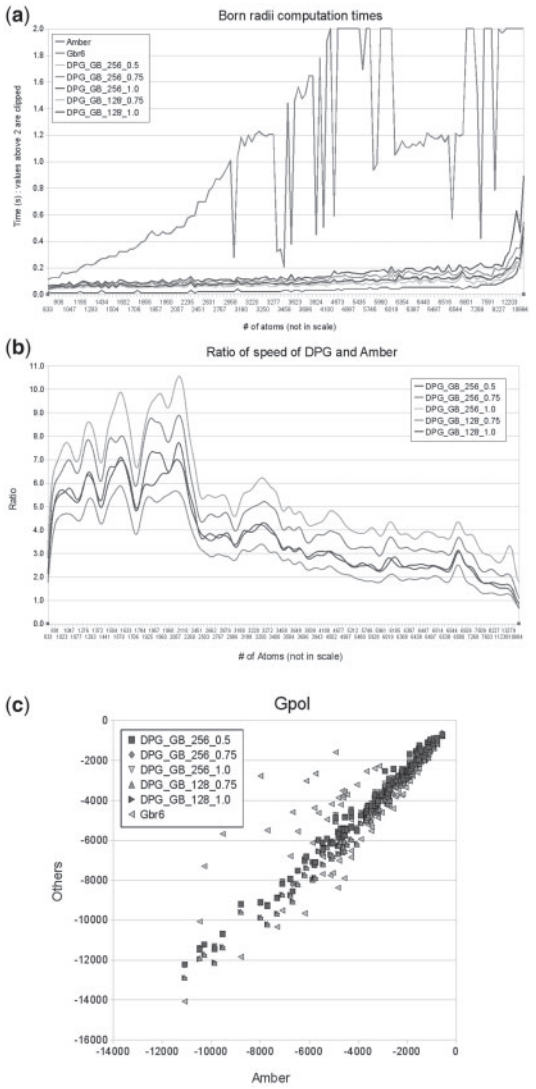

Fig. 7. (**a**) Comparison of Born Radii computation speeds of Amber, GBr^6 and DPG. DPG is almost as fast as Amber, which is the fastest. And GBr^6 is the slowest (some higher values are cropped). (**b**) Ratio of Born radii computation times of DPG and Amber, sorted by increasing size of molecules. DPG_GB_128_0.75 is the fastest, and DPG_GB_256_1.0 is the slowest as expected. But the ratio of all five schemes improve as the size of the molecules increase. (**c**) Scatter plot correlating the polarization energies computed using DPG and *GBr6* with Amber.

Figure 7c displays that DPG consistently produces G_{pol} values similar to Amber's. In fact, the average deviation of G_{pol} computed by DPG-based scheme from Amber's is less than 5%.

Funding: NIH contracts (R01-EB00487, R01-GM074258, R01-GM073087, in part); a grant from the UT-Portugal CoLab project.

Conflict of Interest: none declared.

REFERENCES

Bajaj,C. and Zhao,W. (2010) Fast molecular solvation energetics and force computation. *SIAM J. Sci. Comput.*, **31**, 4524–4552.

Bajaj,C. *et al.* (2003) Dynamic maintenance and visualization of molecular surfaces. *Dis. Appl. Math.*, **127**, 23–51.

Bajaj,C. *et al.* (2009a) A dynamic data structure for flexible molecular maintenance and informatics. In *SPM '09: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pp. 259–270.

Bajaj,C. *et al.* (2009b) F2Dock: fast fourier protein-protein docking. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics* [Epub ahead of print; doi: 10.1109/TCBB.2009.57].

Bajaj,C. (2009c) A fast variational method for the construction of resolution adaptive c^2-smooth molecular surfaces. *Comput. Methods Appl. Mech. Eng.*, **198**, 1684–1690.

Bajaj,C. *et al.* (2010) A dynamic data structure for flexible molecular maintenance and informatics. *Technical Report TR-10-31*, ICES, University of Texas at Austin, Austin, TX, USA.

Batsanov,S.S. (2001) Van der Waals radii of elements. *Inorg. Mater.*, **37**, 871–885.

Case,D.A. *et al.* (2005) The Amber biomolecular simulation programs. *J. Comput. Chem.*, **26**, 1668–1688.

Clarkson,K.L. *et al.* (1990) Combinatorial complexity bounds for arrangements of curves and spheres. *Dis. Comput. Geom.*, **5**, 99–160.

Duncan,B. and Olson,A. (1993) Approximation and characterization of molecular surfaces. *Biopolymers*, **33**, 219–229.

Dolinsky,T.J. *et al.* (2004) Pdb2pqr: an automated pipeline for the setup, execution, and analysis of poisson-boltzmann electrostatics calculations. *Nucleic Acids Res.*, **32**, 665–667

Eyal,E. and Halperin,D. (2005a) Dynamic maintenance of molecular surfaces under conformational changes. In *SCG '05: Proceedings of the 21st Annual Symposium on Computational Geometry*, pp. 45–54.

Eyal,E. and Halperin,D. (2005b) Improved maintenance of molecular surfaces using dynamic graph connectivity. *Algorithms Bioinformatics*, 401–413.

Fredman, M.L. and Willard, D.E. (1993) Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, **47**, 424–436.

Gilson,M.K. and Zhou,H.X. (2007) Calculation of protein-ligand binding affinities. *Annu. Rev. Biophys. Biomol. Struct.*, **36**, 21–42.

Halperin,D. and Overmars,M.H. (1994) Spheres, molecules, and hidden surface removal. In *SCG '94: Proceedings of the 10th Annual Symposium on Computational Geometry*, pp. 113–122.

Mezey,P.G. (1993) *Shape in Chemistry; An Introduction to Molecular Shape and Topology*. VCH Publishers, New York, USA.

Mintseris,J. *et al.* (2005) Protein-protein docking benchmark 2.0: an update. *Proteins*, **60**, 214–216.

Mortensen,C.W. *et al.* (2005) On dynamic range reporting in one dimension. In *STOC '05: Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pp. 104–111.

Qiu,D. *et al.* (1997) The GB/SA continuum model for solvation. a fast analytical method for the calculation of approximate Born radii. *J. Phys. Chem. A*, **101** , 3005–3014.

Rasmus,P. and Flemming,R. (2004) Cuckoo hashing. *J. Algorithms*, **51**, 122–144.

Richards,F. (1977) Areas, volumes, packing, and protein structure. *Annu. Rev. Biophys. Bioeng.*, **6**, 151–176.

Still,W.C *et al.* (1990) Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am. Chem. Soc.*, **112**,6127–6129.

Tjong,H. and Zhou,H.X. (2007) GBr6: a parameterization-free, accurate, analytical generalized born method. *J. Phys. Chem. B*, **111**, 3055–3061.

Varshney,A. *et al.* (1994) Computing smooth molecular surfaces. *IEEE Comput. Graph. Appl.*, **14**, 19–25.

Weiser,J. *et al.* (1998) Neighbor-list reduction: optimization for computation of molecular van der Waals and solvent-accessible surface areas. *J. Comput. Chem.*, **19**, 797–808.

Weiser,J. *et al.* (1999) Fast, approximate algorithm for detection of solvent-inaccessible atoms. *J. Comput. Chem.*, **20**, 588–596.