UNIVERSITEIT VAN AMSTERDAM

Bachelor Thesis Natuur- en Sterrenkunde

Crystacean: a novel non-epitaxial interface configuration generator, tested on the SiC/SiO_2 interface

---

door

Wessel Beumer

18 april 2024

Studentnummer
12640662

Onderzoeksinstituut
Advanced Reseach Center for NanoLithography
Onderzoeksgroep
Materials Theory and Modelling group

Verantwoordelijk docent
Emilia Olsson
Begeleider
Jonathon Cottom

ABSTRACT

As advanced computational techniques have led to significant advances in the field of material science, contemporary methods of generating hetero-interfaces involving amorphous materials have fallen short. Crystacean is a Rust based Python library which aims to fill this gap in the available computational tools. To demonstrate the method developed, the library is used to find configurations of the SiC/SiO_2 interface. Crystacean can generate multiple amorphous SiO_2 layers on top of a SiC substrate lattice. This is done by first identifying where the silicon atoms of the SiO_2 could be placed, and then creating exclusion rules. These sites and exclusion rules are then represented with bitvectors, allowing memory and compute efficient searching of configurational space. This thesis also demonstrates how Crystacean can generate more configurations for a lattice than would be practical by hand and the generation of layers beyond the direct interface, and why these capabilities might be desired. Possible broader applicability beyond the SiC/SiO_2 interface system is also discussed.

POPULAIRWETENSCHAPPELIJKE SAMENVATTING

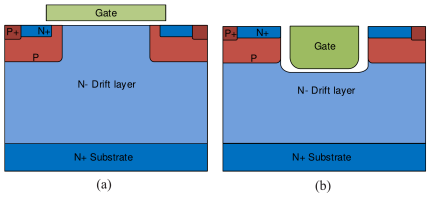
De transistor is misschien wel de belangrijkste uitvinding voor onze moderne wereld. Ze zijn natuurlijk beroemd voor rol in de microchips die computers en smartphones mogelijk maken, maar er zijn ook andere soorten transistors. Transistoren kunnen ook gemaakt worden voor toepassingen, zoals bijvoorbeeld een regelbare spanningsbron of een circuit dat gelijkspanning omzet naar een wisselspanning. Dit soort circuits zijn cruciaal voor elektrische auto's, omdat deze wisselspanning nodig hebben voor hun motoren, terwijl batterijen een gelijkspanning leveren. De transistoren die voor deze circuits nodig zijn, worden MOSFETs genoemd.

Deze MOSFETs werden traditioneel gesproken van silicium gemaakt. Dit is hetzelfde materiaal als voor computerchips gebruikt wordt, waardoor we goed weten hoe we met dit materiaal kunnen werken. Silicium heeft echter problemen bij hoge spanningen en hoge temperaturen. Elektrische componenten, zoals elektromotoren, kunnen vaak efficiënter werken, wanneer ze met hogere spanningen worden aangedreven. Silicium MOSFETs kunnen echter kapotgaan wanneer ze met te hoge spanningen of temperaturen in aanraking komen. Daarnaast verliezen silicium MOSFETs efficiëntie bij hogere temperaturen. Om deze reden is de halfgeleiderindustrie op zoek naar nieuwe materialen om MOSFETs mee te maken.

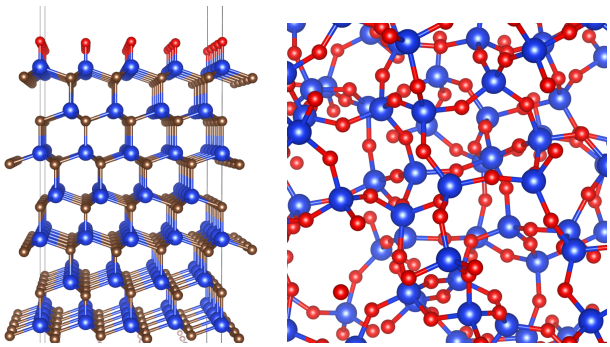
Siliciumcarbide is één van de aangewezen materialen om silicium op te volgen. Siliciumcarbide is bekend als het materiaal waar boorbitjes van gemaakt worden. Het wordt hiervoor gebruik vanwege zijn hoge hardheid en smeltpunt. Vooral deze laatste eigenschap is nuttig voor een materiaal om silicium te vervangen in hoge energiedichtheid MOSFETs. Siliciumcarbide is op atoomniveau een kristal gemaakt van een gelijke mix van silicium en koolstof en heeft de scheikundige aanduiding SiC. De belangrijkste eigenschap van SiC voor dit project is, dat het gebruikt kan worden om MOSFETs te maken die met hogere spanningen en temperaturen kunnen werken. Daarnaast blijven SiC MOSFETs ook efficiënter bij hogere temperaturen. Met de hogere spanningen en minder vereiste koeling kunnen efficiëntere, kleinere en lichtere apparaten gemaakt worden. Siliciumcarbide MOSFETs zijn hierdoor erg gewild geworden.

Er is echter één probleem met SiC MOSFETs. Een transistor kan gezien worden als een schakelaar, die stroom door laat lopen wanneer er een bepaalde spanning op hun derde pin wordt gezet. Bij veel SiC MOSFETs verandert over de loop van een jaar de spanning echter, waardoor de schakelaar of te vroeg, of te laat opengaat. Dit is slecht nieuws voor ingenieurs die met deze MOSFETs willen werken, omdat zij precieze waardes nodig hebben om effectieve circuits te ontwerpen. Het is daarom belangrijk om erachter te komen hoe deze verandering ontstaat en hoe deze vermeden kan worden.

Om deze verandering te onderzoeken, willen we de materialen in onze MOSFETs kunnen simuleren. Hiervoor moeten we de structuur van de materialen op atomair niveau weten, maar dit lastig voor SiC MOSFETs. De reden hiervan zit in de naam. De afkorting MOSFET staat voor "Metallic Oxide Semiconductor Field Effect Transistor". Belangrijk hier is het "Metallic Oxide" gedeelte, een metaaloxide. Het metaaloxide wordt gebruikt in de "gate" die te zien is in Figuur 1a en is voor een SiC MOSFET gemaakt van siliciumdioxide, SiO_2. Dit is dezelfde SiO_2 als waar glas van gemaakt is. Het raakvlak waar deze twee materialen elkaar tegenkomen wordt de SiC/SiO_2 interface genoemd.



(a) Schematische weergaves van twee verschillende technieken om MOSFETs te maken. Voor een Siliciumcarbide MOSFET is de groene 'gate' gemaakt van siliciumdioxide, ookwel bekend als glas. De blauwe en rode delen zijn gemaakt van SiC. Afbeelding van She et al. (2017)



(b) Voorbeelden van de atoomstructuren van (links) Siliciumcarbide en (rechts) het amorfe siliciumdioxide.

Figure 1

De SiO_2 vormt echter een probleem bij het maken van onze modellen. De twee materialen van de transistor zijn schematisch weergegeven in Figuur 1b. Hier is te zien dat de SiC een geordende, voorspelbare kristalstructuur heeft. Bij de SiO_2 is echter geen enkele voorspelbare structuur te bekennen. Deze vorm van SiO_2 wordt daarom ook amorf – zonder vorm – genoemd. Door deze amorfe structuur is het lastig om te voorspellen hoe een SiC/SiO_2 interface er op atomair niveau uitziet. Er zijn heel veer variaties en het liefst zouden we die allemaal af willen gaan.

Dat is waar mijn project in beeld komt. Ik heb voor mijn project een programma geschreven dat variaties van de SiC/SiO_2 interface kan genereren. Dit programma kan laag voor laag een amorfe SiO_2 laag op een kristal van SiC groeien. Door de datastructuren in mijn programma zo klein mogelijk te maken, heb ik het programma zo snel en geheugenefficiënt mogelijk gemaakt. Van deze structuren kan vervolgens de energie berekend worden die zou vrijkomen als de structuur in het echt gevormd zou worden. Deze energie kan dan tussen de structuren vergeleken worden om te bepalen welke structuren het waarschijnlijkst zijn om te vormen.

Ik heb daarnaast ook mijn programma gebruikt om te laten zien hoe het programma nuttig kan zijn voor onderzoek naar interfaces. Ik laat zien dat er SiO_2 lagen met een lagere energie te vinden zijn op grotere SiC kristallen, wanneer het vergeleken wordt met kleinere SiC kristallen. Ook demonstreer ik de mogelijkheid om meerdere lagen te maken. Ik laat hierbij zien dat een structuur die op de eerste laag de laagste energie heeft, mogelijk meer energie nodig heeft voor zijn volgende lagen dan andere structuren die een hogere energie hadden op de eerste laag.

CONTENTS

1 Introduction **6**

2 Theory **6**

2.1 The SiC/SiO_2 interface 6

2.2 Restricted Random Structure Searching 7
Application to the SiC/SiO_2 interface

2.3 Programming details and considerations 8

3 Code Development **9**

3.1 Python 9

3.2 Rust and bitvectors 10

3.3 Subsequent layers 12

4 Crystacean Results **13**

4.1 Small and large input lattices 13

4.2 Subsequent layers 15

4.3 Performance 15

5 Conclusion **15**

5.1 Future work 15

6 Acknowledgments **19**

References **19**

A Link to the code **20**

B Full Rust lib.rs file **20**

C Python code **43**

C.1 classes.py 43

C.2 findthosepoints.py 49

C.3 cull_results.py 59

1 INTRODUCTION

The advent of advanced computational techniques have led to significant advances in the field of material science, enabling the exploration of vast configurational spaces for the design and discovery of new materials. An important area of research is that of hetero-interfaces, the surface where two different materials meet, as these kinds of interfaces are important in applications like catalytics (Yang et al., 2022), energy storage (Fang et al., 2019) and transistors (She et al., 2017). Analyzing these interfaces computationally requires methods to generate valid interface structures, but these have proven challenging to develop for non-epitaxial systems.

The previous method used to determine interface configurations for epitaxial systems, is the lattice matching approach by Zur and McGill (1984). It utilizes the periodicity of the crystal lattices of both materials, by combining small crystal lattice cells into larger "superlattice cells" containing many smaller units. Two lattices are then said to match, when two of their superlattice cells coincide to within 1% in both dimensions and angle. This method is powerful, but cannot be used with non-epitaxial systems, as in these systems the materials are not aligned in crystal direction. New methods are thus needed to explore this field of material science.

Random Structure Searching (RSS) is a different approach to finding new material structures (Johnston, 2003). This method tries to find structures by assembling structures from scratch and computationally testing their stability and properties afterwards. This allows for the discovery of possible stable or metastable configurations, which could correspond to real materials under experimental conditions. There are many implementations, the most prominent being USPEX (Glass et al., 2006) and AIRSS (Pickard and Needs, 2011). The rough process of an RSS search is as follows: First, structures are generated within certain constraints, such as density or composition. These constraints are left loose enough to allow for a wide sampling of configurational space. After this, the new structures are evaluated with ab initio methods, particularly Density Functional Theory (DFT). The results of this search can be used to refine subsequent searches, to focus searching on the most promising configurations.

The fundamental ideas of RSS are both simple and broadly applicable. The large amount of samples needed to explore the entire configurational space can be challenging, however. To overcome this challenge, the concept of Restricted Random Structure Searching (RRSS) (Cottom et al., 2019) was introduced. With this new method, the searched configurational space will be shrunk by applying rules and restrictions based on physically and chemically reasonable considerations.

In this thesis, I will show how I used RRSS to develop an interface configuration generator called Crystacean, which can generate multilayered interface configurations, and use it to generate interfaces for the SiC/SiO_2 interface. In Section 2 I will cover the theory of RRSS and finding the configurations. In Section 3 I will describe the development process of Crystacean in more detail. Then, I will present some results I obtained with my generator in Section 4, which will show in what ways Crystacean can be used for the analysis of interfaces, by showing how bigger structures and multiple layers can affect the energetic properties of materials. Finally, I will summarize my findings in Section 5.

2 THEORY

2.1 The SiC/SiO_2 interface

To demonstrate Crystacean, I will determine possible atomic configurations of the interface between silicon carbide and silicon dioxide (SiC/SiO_2). Silicon carbide has long been regarded as a possible candidate for use in semiconductors. Devices made of SiC are desirable, because of their superior electronic properties compared to standard silicon (Millán et al., 2014; Roussel, 2011). This makes the material suitable for high voltage, high temperature and high frequency applications.

Silicon carbide possesses polytypism, meaning it has many crystalline forms (Choyke and Pensl, 1997). The SiC lattice unit cell can be seen as a repeating pattern of alternating silicon and carbon atoms. However, every next layer has two ways with which it can follow up the previous. This means that there are countless amounts of SiC polytypes possible. Out of the many known SiC polytypes, the 4H-SiC variant appears to be one of the most popular for the creation of devices (Presser and Nickel, 2008). An example of 4H-SiC can be seen in Figure 2. This SiC variant can be used to make a Metallic Oxide Semiconductor Field Effect Transistor, or MOSFET, possibly the most important active components in electronics. These devices require an oxide layer in their construction, and in SiC this layer is commonly made of silicon oxide, thus forming a SiC/SiO_2 interface (Dhar et al., 2005).

This interface has led to problems with device performance and lifespan (Aichinger et al., 2018; Presser and Nickel, 2008). 4H-SiC based devices suffer from threshold voltage instabilities, which are attributed to point defects in the SiC at the SiC/SiO_2 interface. These point defects include silicon vacancies and various amounts of dangling carbon bonds (Cottom et al., 2018). Analyzing these configurations requires the ability to construct a SiC/SiO_2 interface. Lattice matching would normally be used to construct such an interface, but silicon oxide is amorphous. This makes the SiC/SiO_2 interface non-epitaxial and ineligible for lattice matching, but a good candidate to demonstrate RRSS.

2.2 Restricted Random Structure Searching

The RRSS method is largely based on, and shares many similarities with the *ab initio* Random Structure Searching method by Pickard and Needs (2011). RRSS differentiates itself on the strictness of the restrictions placed on possible atom sites. A schematic representation of RRSS can be seen in Figure 3. Where AIRSS allows atoms to be placed freely within a given bound, RRSS first determines discrete allowed sites these atoms can occupy. These sites are derived from the atomic properties of the next layer's atoms, such as valence, and the geometry of the previous layer. After finding these sites, additional restrictions can be added to represent the sites' interdependencies (Cottom et al., 2019).

These sites can then be used to find possible interface structures. If the identified structures contain many unphysical interfaces, additional restrictions could be applied to reduce this amount. The identified structures will then be subjected to energy and geometry calculations to relax them to a local energy minimum. These structures can be used as they are, or used to add additional interface layers. By using this more restricted approach, this technique aims to efficiently sample the space of possible interface configurations, despite possibly missing less obvious stable solutions.

2.2.1 Application to the SiC/SiO_2 interface

In the following section, I will show how I applied RRSS to the SiC/SiO_2 interface. When seen from above, 4H-SiC has a hexagonal unit cell, see Figure 2b. Building the first layer starts with placing oxygen atoms on the lattice, as seen in Figure 2. The silicon atoms will be attached to these oxygen atoms. Following the RRSS method, the valence of the present atoms can be used to construct possible sites of different types. Silicon has a valence of four, from which three possible ways a silicon atom can connect to the previous layer can be derived. The three connection types are shown in Figure 4. A silicon atom can connect to either one (singlet), two (midpoint) or three (tripoint) oxygen atoms from the previous layer. The unconnected bonds of the atoms are then presented upwards for the construction of the next layer.

Besides the restricted placement, there are two more restriction rules which should be implemented.

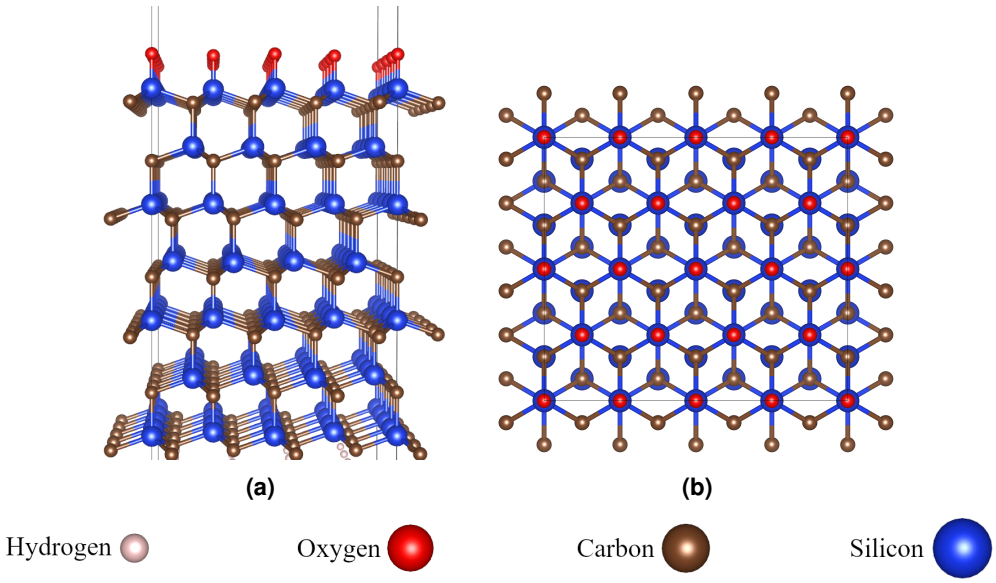


Figure 2. An example of a 4H-SiC lattice shown from the side (a) and the top (b). The crystal is hydrogen terminated at the bottom and oxygen terminated at the top.

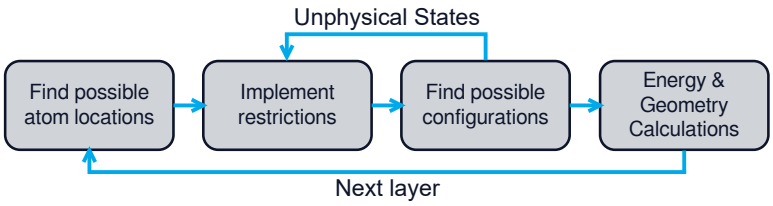


Figure 3. A schematic representation of the Restricted Random Structure Searching method.

The first rule is to limit the amount of singlet states present in final solutions. This rule exists, because a large amount of singlet states in close proximity would cause voids on the surface of the SiC, which would be unstable. The second rule applies to secondary layers and states that the formation of small loops should be prevented. An example of a small loop is shown in Figure 5. A small loop here means that two or three of the oxygen atoms, to which a midpoint or tripoint is connected, are connected to the same silicon atom of the previous layer. The formation of these loops is unfavorable as, although they are easy to detect, they are never seen experimentally in SiO_2.

2.3 Programming details and considerations

When writing any scientific program, using Python would be an obvious first choice. It is a popular programming language in the world of physics and has a large number of libraries aimed at doing many numerical calculations. A big downside of using Python is its speed: Python code generally takes orders of magnitudes longer to run than comparable code in other languages (Heer, 2023). Its generous use of pointers and runtime typing also make it inefficient with memory. There are a few ways to make Python faster. Using NumPy arrays can make calculations over many numbers faster, but this lacks the fine control needed to optimize this problem (Harris et al., 2020). Numba can speed up Python code execution by using a JIT compiler (Lam et al., 2015). However, using Numba hurts Pythons ergonomics, as it does not support classes and only partially supports NumPy.

Another option is to write an external Python library in a faster, compiled language, like Rust. The Rust programming language (Matsakis and Klock, 2014) is a compiled language focusing on memory and type safety, extendability and developer experience. The language has been growing in popularity over recent years in many fields, from embedded software to web development (Daigle and Staff, 2023), but has not yet had the years-long support from the scientific community Python has had. A Rust library can be easily compiled into a Python library by using the PyO3 crate[1] (Rust library). This is the route I decided to take, mainly because of familiarity.

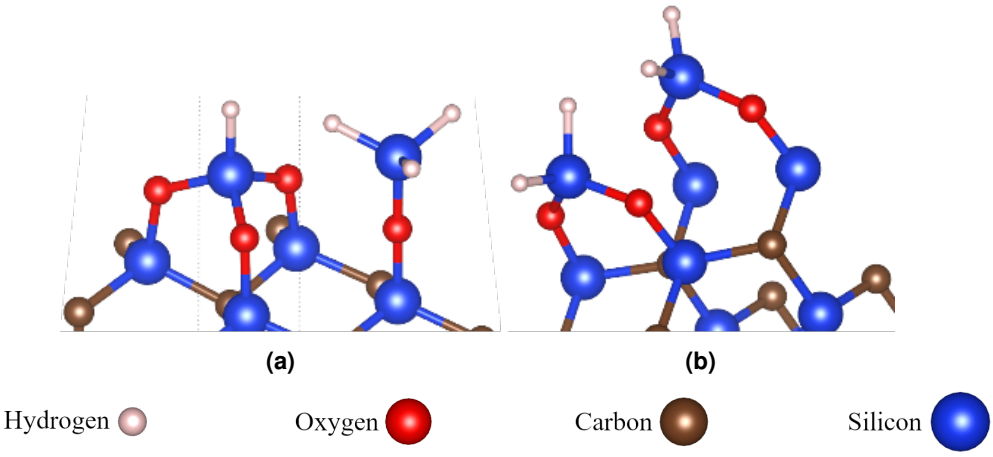


Figure 4. Examples of the three types of connections a silicon atom of an SiO_2 layer can have. a shows the triplet (left) and the singlet (right) states. b shows two midpoint examples.

[1] https://crates.io/crates/pyo3

There are many more possible options. Python also has libraries specialized in atomic modeling and simulation, like Atomic Simulations Environment (ASE) (Larsen et al., 2017), possibly in combination with a framework like Atomistic Global Optimization X (AGOX) (Christiansen et al., 2022). These libraries allow researchers to design whole experiments in a single function and interface with many popular computational science tools. While these extensive libraries might not have been the fastest for every step of my structure search, the all-inclusive experience of these libraries might be worth this trade-off for many.

RSS is commonly used together with software which can run Density Functional Theory calculations (Pickard and Needs, 2011). To this end I have used CP2K (Kühne et al., 2020). I used CP2K with the MOLOPT dataset (Li et al., 2021) and a PBE functional to calculate the energy of the system and geometrically relax the atoms in the structure. While this method can be very accurate for formation energy calculations, it is also very compute intensive; calculations on a structures of around 80 atoms can take dozens of hours on consumer grade hardware.

3 CODE DEVELOPMENT

3.1 Python

At the beginning of the project, I started by finding potential silicon sites for a given input lattice. My supervisor had given me some examples of how these sites could be found, and one used the KDTree (Maneewongvatana and Mount, 1999) data structure from SciPy (Virtanen et al., 2020). This is a data structure which orders elements based on multidimensional proximity, which enables finding points which are within a certain distance from one another. This was exactly what I needed, as the midpoints and tripoint sites could only form between attachment points within a certain distance. I also needed to make sure multiple silicon sites would not get double generated. A site could be generated multiple times when, for example, a midpoint coupled to two attachment sites was added by both of its ends. To prevent this, I sorted the list of coordinates and only allowed a coordinate to create an attachment site with its neighbors when the neighbors are placed further down the list.

Eventually I could reliably find all potential silicon sites, even for larger input lattices. I then implemented exclusion rules for the silicon sites. The oxygens of the input lattice can only connect to one silicon atom in the new layer, which means that any potential silicon site has to check if its associated lattice points are already taken when it wants to be filled itself. I implemented this check by giving every lattice point a variable, which stated whether the point was used in a connection and with which

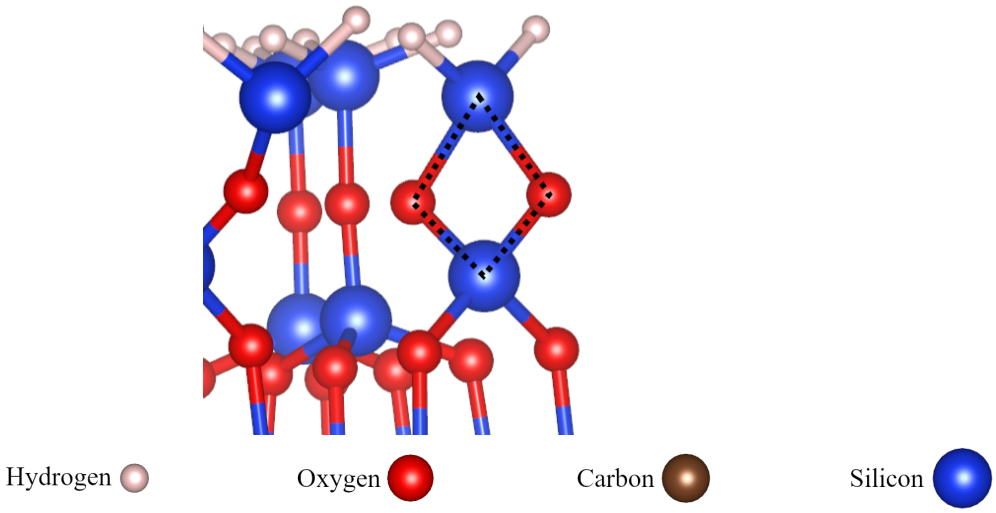


Figure 5. An example of a small loop, of which the formation should be prevented in secondary layers. The small loop marked here is caused by a midpoint connecting to two oxygen atoms, which are connected to the same silicon atom of the previous layer.

silicon site. A potential silicon site could then check and update its associated lattice points. This process has been illustrated in Figure After implementing these features, I also wrote a breadth first searching algorithm. To follow the minimize singlets restriction from Subsection 2.2.1, the algorithm first checked if any tripoints or midpoints were allowed to be filled. Only if no other sites were available, could singlet states be filled. The layouts of the interface configurations identified with this algorithm were saved to json files, which would have to be converted to atomic structures for further analysis.

After this, I also implemented periodic boundary conditions for input lattices. Periodic boundary conditions mean that an atom which would exit the unit cell of the material on one side, would reappear on the opposite side. This has the effects of simulating an infinitely large input lattice and allowing potential silicon sites to cross over into neighboring unit cells. To achieve this, I created a new kind of lattice point called a ghost point, which would act like one of the original points, but in a different place. At this point, I started to encounter significant performance and memory limitations; I could not even find all of the solutions for a sixteen atom input lattice without boundary conditions. The cause of these limitations was clear: Each step of the breadth first search algorithm required many deep copies of a rather inefficient data structure. If I still wanted to be able to find all solutions of bigger systems, I would need to take a different approach.

3.2 Rust and bitvectors

At this point I switched to writing a Rust library. This did mean that I had to rewrite the logic I had previously written for finding the possible silicon sites, but the process did not have to change much to fit this new language. I found a new Rust KDTree library in "kiddo"[2], which was almost as easy to work with as the SciPy version. After this step, my new implementation diverged.

I realized that a potential silicon site could either be occupied or unoccupied, which means that the occupancy of a site could be described with just a single bit of information. An entire lattice with N potential silicon sites could be described with a bitvector containing N bits. Therefore I define

$$\vec{v} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{1}$$

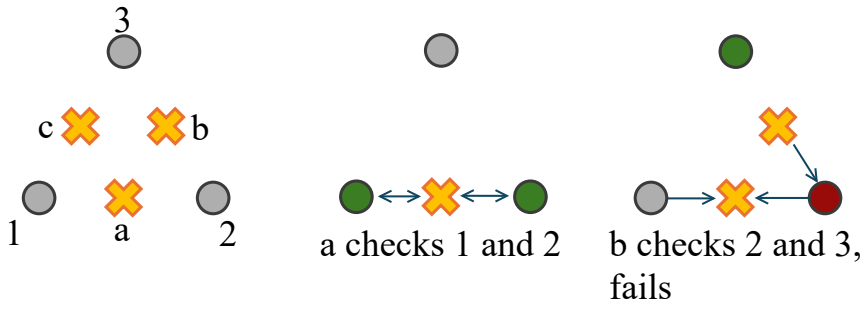


Figure 6. An example of how the exclusion rules for the Python implementation work. The example lattice has three atoms: 1, 2 and 3, and three midpoint silicon sites: a, b and c. When a is attempted to be filled in the middle diagram, it checks 1 and 2 to see if they are available, and finds that they are unallocated. A then gets filled, and 1 and 2 get allocated to a. When b is attempted to be filled in the third diagram, its checks 2 and three and finds that 2 is allocated to a. This causes the filling of b to fail.

[2]https://crates.io/crates/kiddo

as a partial solution vector. I then also made a binary representation of the exclusion rules. I did this by creating the exclusion matrix, which is an $N \times N$ bitmatrix, defined as

$$M = \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & \cdots & 1 \end{pmatrix} \tag{2}$$

where if site j is excluded by site i, this gives

$$M(i,j) = M(j,i) = 1 \tag{3}$$

and the opposite when i does not exclude j. A sample of the responsible code for this conversion is shown in Listing 1. When these definitions are made, determining which states can still be filled for a given partial solution also becomes trivial, as it can be determined with

$$\overline{M\vec{v}} = \vec{w} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{4}$$

with $\vec{w}$ the allowed vector. The code for this process is shown in Listing 2. In practice, this means that for every bit i of the new allowed vector a bitwise AND operation gets performed between the partial solution vector and the ith row of the exclusion matrix, with the new bit being set to one when all bits in the result of this operation are zero. New partial solution vectors can then be made with the identified allowed states. This bitvector approach has the following two upsides:

1. A single configuration takes up a minimal amount of memory. The bitvector library I used[3] uses 24 bytes on the stack to note the vector length and heap allocation. It then stores the bits on the heap in 32 bit chunks, which makes the total memory footprint of an N-bit long bitvector $24 + 4 \cdot \lceil N/32 \rceil$ bytes.[4]

```
416 for (number, oxygen) in self.oxygens.iter().enumerate() {
417     let mut exclusions =
        ↪  FixedBitSet::with_capacity(self.oxygens.len());
418     for exclusion in &oxygen.exclusions {
419         exclusions.set(exclusion.0, true);
420     }
421     match &oxygen.sitetype {
422         SiteType::Tripoint(_) => tripoint_mask.set(number, true),
423         SiteType::Midpoint(_) => midpoint_mask.set(number, true),
424         SiteType::Singlet(_) => singlet_mask.set(number, true),
425     };
426     exclusion_matrix.push(exclusions);
427 }
```

Listing 1. Rust code responsible for creating the bitvector representation of a lattice. Full code can be seen in Appendix B

[3] https://crates.io/crates/fixedbitset

[4] Since the writing of the Crystacean version described in this report, many changes have improved the performance and memory footprint of the FixedBitSet library. This does mean the information presented in this report is outdated in respect to the most recent versions.

2. The calculations can be performed with simple bitwise operations. The calculations for determining the allowed vector only require AND operations and comparisons to zero. These operations could theoretically even be vectorized with SIMD instructions[5], which would mean a single row-vector and operation could be performed in a single clock cycle.

With this new representation, I once again implemented a breadth first searching algorithm for the structures. This time, all solutions could be identified for the sixteen lattice input lattice, both with and without periodic boundary conditions. These solutions could again be exported as json files. At this point, the amount of solutions identified for the sixteen atom input lattice was over 100,000, many of which would likely be translations or rotations of one another. To filter out these duplicates, I created a post processing program in Python. This program created a sorted list of distances between each occupied silicon site of each structure. These lists were then compared with one another, where two structures were marked as being duplicates when all the distances on their list were within a given margin. This margin can be chosen freely, dependant on the desired culling aggressiveness. With this program and my chosen margin, about 99% of the identified structures could be eliminated.

3.3 Subsequent layers

The interface configurations identified in the previous sections could then be converted to an atomic file format which could be energetically and geometrically analyzed with CP2K. The geometrically relaxed structures would look like the structures in Figure 4 and were encoded in the XYZ format. To be able to add a second layer to these structures, Crystacean would need to be able to parse these files. I used ASE to convert these files from XYZ to json files, as these would be easier to work with in Rust. As can be seen in Figure 4, the silicon atoms of the second layer had hydrogenated ends where they could connect to a following layer. Therefore I could create a new input lattice by selecting these hydrogen atoms and extracting their coordinates. This new input lattice could then be processed similarly as the earlier input lattices.

An important thing which did have to change, was saving the resulting structures to files. As Crystacean already received a valid ASE json file, I decided to let it output copies of the original ASE json file, with the atoms of the newly generated layer added. The first step in the process of adding the new layer, was to replace the existing hydrogen atoms with oxygen atoms. After this, the new silicon atoms could be added to the file. Lastly, new hydrogen atoms had to be added to the newly added silicon

```rust
fn matrix_vector_multiply(&self, vector: &FixedBitSet) -> FixedBitSet {
    let mut output_vector = FixedBitSet::with_capacity(vector.len());
    for bit_nr in 0..output_vector.len() {
        let mut enabled = true;
        // equal to: (vector & &self.exclusion_matrix[bit_nr]).is_clear()
        for (v, m) in zip_eq(vector.as_slice(), self.exclusion_matrix[bit_nr].as_slice()) {
            if (v & m) != 0u32 {
                enabled = false;
                break;
            }
        }
        output_vector.set(bit_nr, enabled);
    }
    output_vector
}
```

Listing 2. Rust code responsible for the boolean linear algebra. The method receives a bitvector, which it then logically ands with the rows of the exclusion matrix. Full code can be seen in Appendix B

[5] SIMD has been added in newer library versions.

Table 1. The energies calculated by CP2K for the three small structures.

Sample	Total energy (a.u.)	Total energy (eV)	ΔE to lowest (eV)
Small 1	−345.80	−9409.62	0.00
Small 2	−345.76	−9408.63	0.99
Small 3	−345.76	−9408.64	0.98

atoms. Though adding the hydrogen atoms to the tripoint and singlet silicon atoms was trivial, adding the midpoint hydrogens was a bit more involved. I had to derive at which angle the hydrogens would connect by looking at the relative locations of the oxygen atoms associated to the silicon atom.

At this point I also had to implement the last restriction of the system mentioned in Subsection 2.2.1, the prevention of small loops. According to this restriction, some of the possible silicon sites could not be filled. To prevent these sites from being filled, I created a method which accepted a bitvector. The sites marked in this bitvector would then temporarily be removed from the intermediary representation. I illustrated the effects of this filter in Equation 5. Solving the problem with the new exclusion matrix and partial solution vectors would then yield smaller solution vectors, which would need to be corrected to the original size. This is done by applying the filter in reverse, as shown in Equation 7.

$$\begin{pmatrix} 1 & \underline{0} & 0 & 0 & 1 \\ \underline{0} & \underline{1} & \underline{0} & \underline{1} & \underline{0} \\ 0 & \underline{0} & 1 & 0 & 0 \\ 0 & \underline{1} & 0 & 1 & 0 \\ 1 & \underline{0} & 0 & 0 & 1 \end{pmatrix} + \text{filter} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \tag{5}$$

Where $[\underline{0}, \underline{1}]$ denotes the removed row and column. (6)

$$\begin{pmatrix} 1 \\ \\ 1 \\ 1 \\ 0 \end{pmatrix} + \text{filter} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{7}$$

4 CRYSTACEAN RESULTS

In this section, I want to show some of the reasons of why using Crystacean is useful, or possibly even necessary, for the modelling of interfaces with amorphous materials. In the first subsection, I will show how using larger input lattices can reveal solutions with lower formation energies compared to those found on smaller lattices. After this, I will demonstrate the formation of additional layers, and show how structures which had low formation energies on their first layer, can have significantly higher energy in subsequent layers. Finally, I will give a short demonstration of the performance of Crystacean.

4.1 Small and large input lattices

The first results contain four oxygen atoms. Without considering boundary conditions, this system has four interface configurations. Three of these have been chosen for energetic comparison, see Figure 7. Structure 7a contains a tripoint site and a singlet and Structures 7b and 7c both contain two midpoints. The fourth structures was left out, as this structures was a rotation of structure 7a. The formation energies of these structures have been calculated with CP2K, and their values are shown in Table 1. The formation energy here is the energy needed to form the supplied atomic structures from individual, unpaired atoms, or diatoms in the case of oxygen and hydrogen. The formation energies for the structures covered here are negative, meaning that energy would be released if these structures have been made from loose atoms. The table shows that the first structure, with the tripoint and singlet, has the lowest formation energy. The energies of the midpoint structures are about an electron volt higher. Additionally the energies of the structures 7b and 7c are nearly identical, indicating that rotations of the lattice likely have equal energies.

While the configurational space of this simple input lattice is small, the amount of possible configurations increases sharply with larger systems. To show the necessity of considering larger systems, I also prepared some larger structures, see Figure 9. Each of these structures contains sixteen attachment sites. These larger structures can be divided into three catagories: Structures 9a and 9b are small structures from Figure 7 repeated four times, structures 9c and 9d are combinations of the smaller structures and

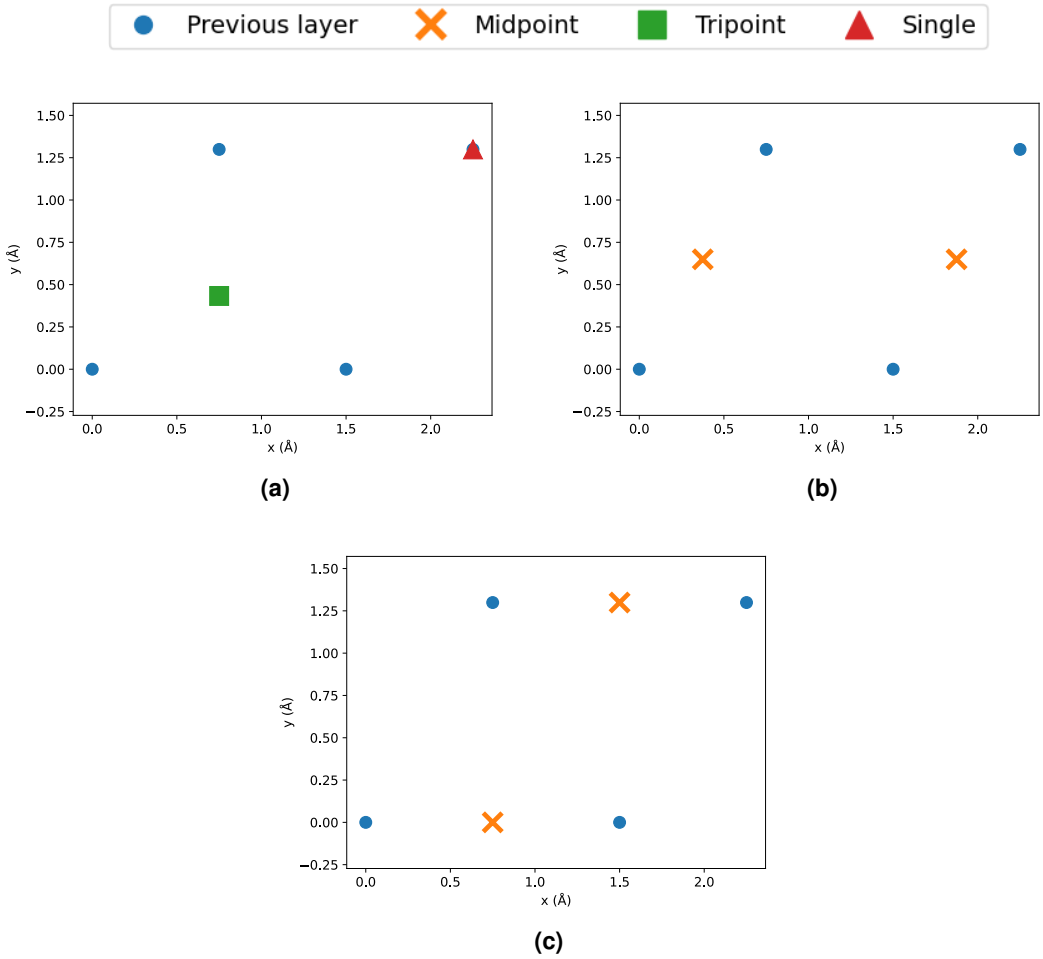


Figure 7. A small SiC substrate with four attachment points has three unique SiO_2 configurations. Structure a contains a tripoint site and a singlet and Structures b and c both contain two midpoints.

Sample	Formation energy per atom (eV)
a	−2.20
b	−2.07
c	−2.14
d	−2.15
e	−2.76
f	−3.80

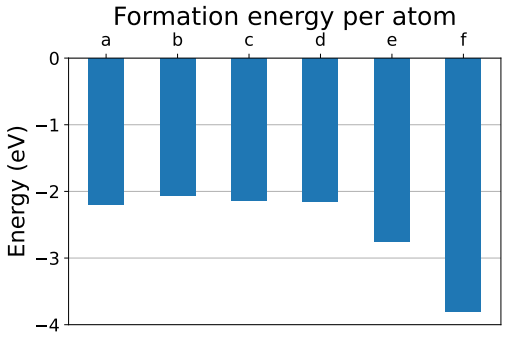


Figure 8. The formation energy of the larger structures shown in Figure 9. The energies have been normalized to amount of new atoms added in the layer.

structures 9e and 9f are structures which are not made from smaller structures and which contain larger patterns. The resulting formation energies are given in Figure 8. Because the structures covered in this part do not have the same amount of atoms added in their new layers, the formation energy has to be normalized to the amount of atoms added for a fair comparison. To do this, a theoretical formation energy is first calculated for the added silicon and hydrogen atoms. This energy is then added to the energy of a structure with the new layer removed, calculated with CP2K. By subtracting this value from the actual formation energy found, a difference in energy can be found. Dividing this energy by the amount of atoms added in the new layer gives the additional energy released forming a layer of SiO_2 per atom.

Figure 8 shows that both repetition of the small solutions and combinations of solutions have approximately the same formation energy per atom added. The differences also align, with 9b having the highest formation energy of the structures, analogous to the small structures 7b and 7c. The unique larger structures have significantly lower formation energies than the ones made from smaller configurations. For example, the difference in formation energy per atom between structure 9a and 9f is about 1.6 eV. These results show that the ability to generate larger structures is a necessary to properly analyze the interfaces of non-epitaxial systems, as small structures might not be able to represent the lowest energy configurations of an interface. Additionally, forming tripoint connections appears to be energetically favorable over forming midpoint or singlet connections.

4.2 Subsequent layers

This last point changes when a second layer is added. To evaluate the effects of adding a second layer on a generated material, I took the geometrically optimized versions of the structures from Figure 7 generating structures with a second layer from them, as shown in Figure 10. The structures 10a and 10b are based off structure 7a, structure 10c is based on 7b and structure 10d is based on 7c. Structure 10a and 10b both contain one midpoint connection and two singlet connection, while the 10c and 10d structures both have two midpoints in their second layers. The formation energies per atom are given in Figure 11. The figure shows second layers identified for the two midpoint solutions, 7b and 7c, which had the highest energies for their first layer, have significantly lower energies for their second layers. The two structures based on 7a have a second layer with an energy which is op to 0.8 eV higher. These results show that structures with multiple layers are needed for the modeling of non-epitaxial interfaces.

4.3 Performance

Finally, I want to give some indication of the performance of Crystacean. To do this, I have recorded the time taken to find all solutions for a sixteen atom input lattice with periodic boundary conditions. The measurement is taken with the "time" command in bash on a laptop running Linux Mint. The laptop has an AMD Ryzen 5 PRO 2500U and 32 GB of 2666 MHz DDR4 RAM. This laptop can find all 125,992 solutions to this input lattice in about 8.5 seconds. A plot of the profiling results of the solving method is shown in Figure 12.

5 CONCLUSION

To summarize, I developed the Crystacean Python library in Rust to identify possible atomic configurations for non-epitaxial interfaces based on the principles of RRSS. I then used it to identify interface configurations for SiC/SiO_2. The library is able to do this efficiently by using bitvectors. It first identifies possible silicon sites, after which it represents the filled status of these sites in a bitvector. I also encoded the exclusionary relations of the silicon sites in a binary exclusion matrix. The available silicon sites for a given partial configuration can then be calculated with binary operations between the vector and the matrix. With this method, I was able to identify interface structures for input lattices of four and sixteen atoms. I also showed why the ability to generate larger structures is important, as I presented larger structures which had lower formation energies than combinations of smaller structures. I also showed that Crystacean can generate subsequent layers on existing structures and demonstrated how the relative energies between structures can shift, when a second layer is introduced.

5.1 Future work

In the near future, many more configurations of the sixteen atom input lattice are going to be analyzed with CP2K. As mentioned in Subsection 3.2, Crystacean was able to generate 1,469 unique structures for the sixteen atom input lattice. These structures have been sent out for analysis, but as these calculations

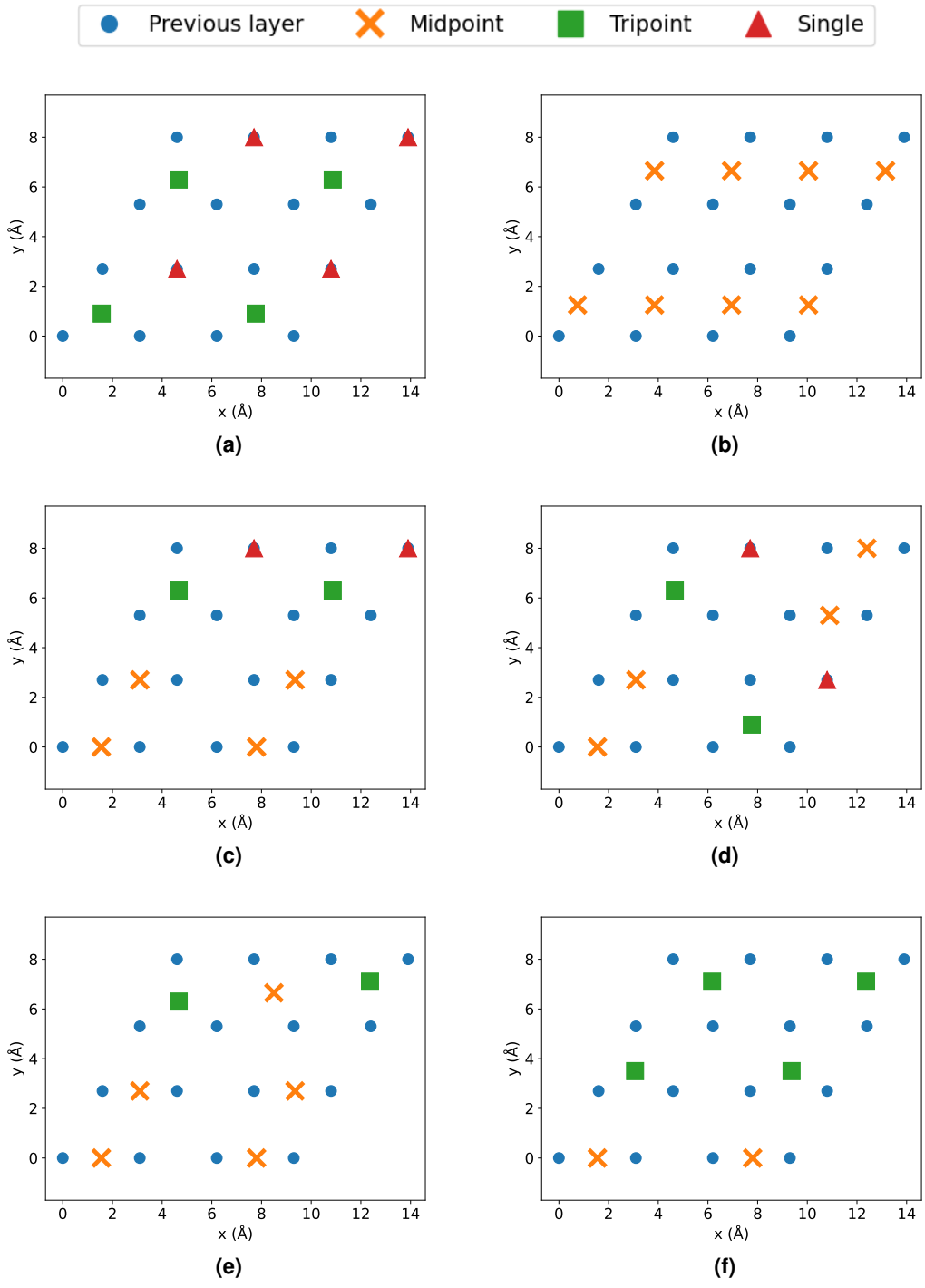


Figure 9. A larger sixteen atom input lattice has many possible solutions, of which the six shown above have been analyzed for this paper. Structures a and b are simply small structures from Figure 7 repeated four times, structures c and d are combinations of the smaller structures and structures e and f are structures which are not made from smaller structures and which contain larger patterns.

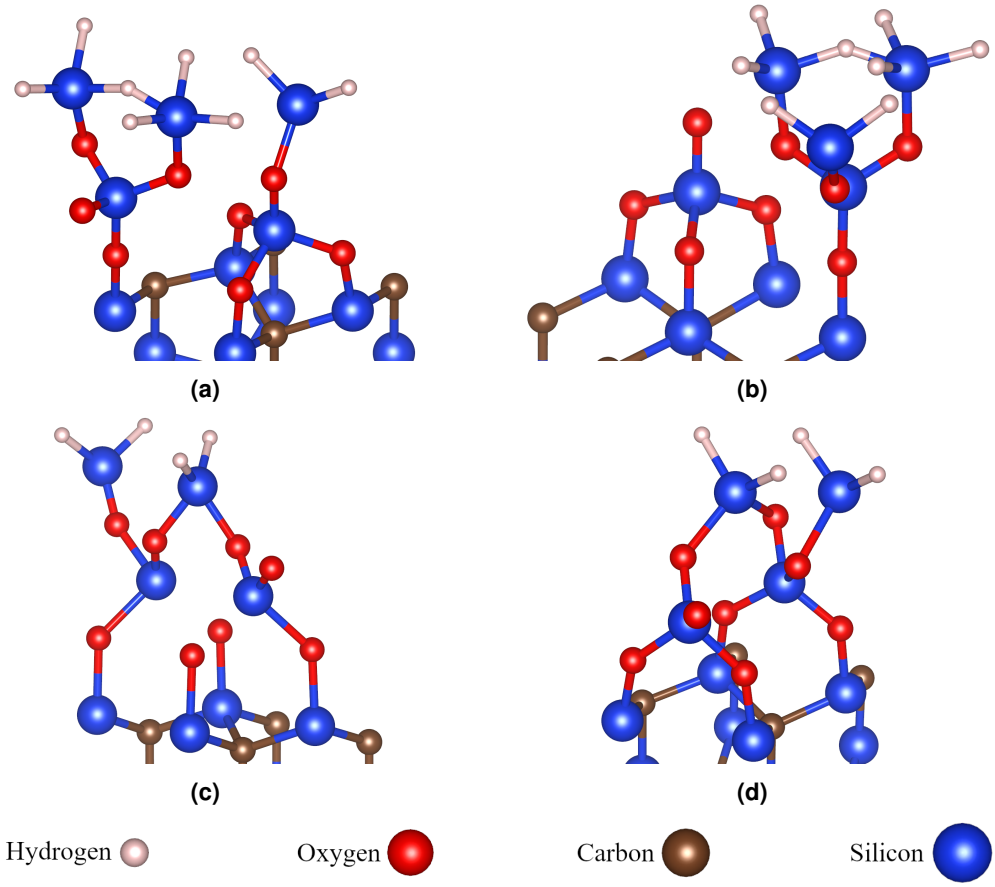


Figure 10. The second layer solutions for the structures shown in Figure 7. Structures a and b are based off structure 7a, structure c is based on 7b and structure d is based on 7c.

Sample	Formation energy per atom (eV)
a	−0.81
b	−0.93
c	−1.30
d	−1.69

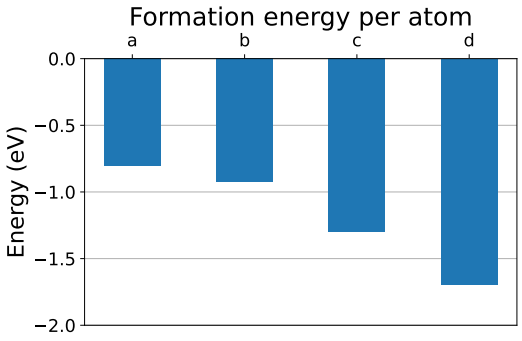


Figure 11. The formation energy of the larger structures shown in Figure 9. The energies have been normalized to amount of new atoms added in the layer.

require significant computation time, these could not be finished in the time allocated for a Bachelors project. When the analysis of these structures is finished, I would likely want to try adding second layers to these structures.

Crystacean does need better culling methods to combat the high memory usage of the library and enable the analysis of larger input lattices. Currently, the breadth first searching algorithm prevents considering duplicate states by storing the next generation of states in a hash set, a data structure which can only contain a single copy of any type of data. This is a reasonable solution for breadth first search, as this searching method needs to store all candidates for the next generation anyway. However, for a more memory efficient approach like depth first, this would only add the large memory footprint of breadth first search to the slower searching speed of depth first searching. A better method would thus need to be used if I wanted to switch to a different algorithm. Additionally, in the current implementation, Crystacean does not consider whether new structures are translations or rotations of existing ones. As mentioned in Subsection 3.2, I used a post-processing script to remove the translations and rotations from the set of solutions found by Crystacean. It might however be more efficient to perform this processing filter inside of the search itself. With both of these additional culling methods, Crystacean could analyze larger structures, and process smaller structures faster.

An additional hurdle to finding structures is the high computational cost of performing DFT calculations. The DFT calculations are needed to compute the energetic properties of our found structures and geometrically relax their atoms for the deposition of the next layer. Performing these calculations can take hours per structure and a method to speed up these calculations for found structures would be desirable. It is possible that advances in machine learning might develop methods which could be used on our structures, but currently no such methods exist.

Lastly, I have shown that the bitvector approach works well for the SiC/SiO_2 interface I covered in this paper, but it would be beneficial if this technique could be applied to more material combinations. Given that the library could theoretically process any type of input lattice, this program could be used on any substrate other than SiC to form an interface with SiO_2. A modified version of Crystacean could also be used for different oxides, given that the oxidized material only has a single oxidation state, or a non-oxide where one of the constituent atoms can only form two bonds (sulfur comes to mind). The approach taken for Crystacean will however struggle with materials, where both atoms can form more than two connections, or where one or both atoms can have a variable amount of connections. For the first case, the original process of Crystacean could possibly be altered to where the two types of atoms are alternatingly deposited. For the second case, while the different amounts of bonds of an atom could theoretically be encoded in its exclusion matrix, it would be rather inelegant and drastically increase the configurational space. The same would apply for amorphous materials with more than three constituent elements.

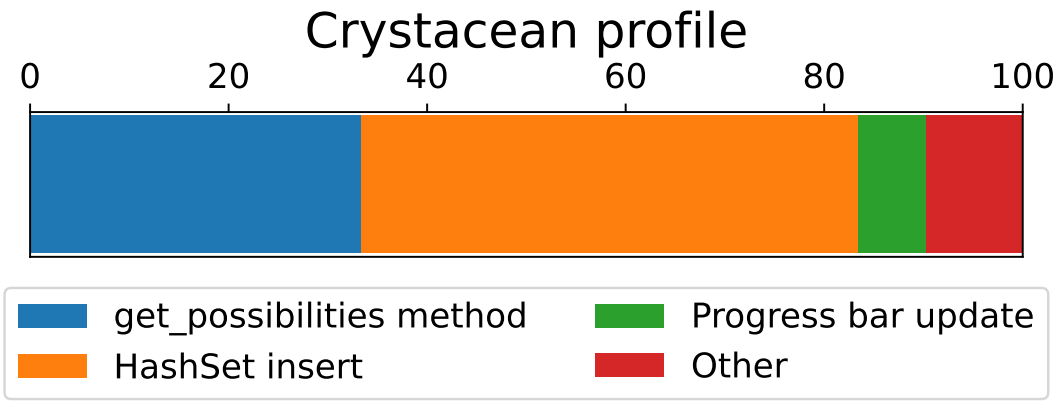


Figure 12. The profile of the solving method of Crystacean. This bar chart shows the breakdown of where the program has spent the most time.

6 ACKNOWLEDGMENTS

Thanks to my daily supervisor Jon Cottom and my supervisor Emilia Olsson from ARCNL for helping me with my thesis and project.

REFERENCES

Aichinger, T., Rescher, G., and Pobegen, G. (2018). Threshold voltage peculiarities and bias temperature instabilities of SiC MOSFETs. *Microelectronics Reliability*, 80:68–78.

Choyke, W. and Pensl, G. (1997). Physical properties of sic. *MRS BULLETIN*, 22(3):25–29.

Christiansen, M.-P. V., Rønne, N., and Hammer, B. (2022). Atomistic global optimization X: A Python package for optimization of atomistic structures. *The Journal of Chemical Physics*, 157(5):054701.

Cottom, J., Bochkarev, A., Olsson, E., Patel, K., Munde, M., Spitaler, J., Popov, M. N., Bosman, M., and Shluger, A. L. (2019). Modeling of diffusion and incorporation of interstitial oxygen ions at the tin/sio2 interface. *ACS Applied Materials & Interfaces*, 11(39):36232–36243. PMID: 31532611.

Cottom, J., Gruber, G., Pobegen, G., Aichinger, T., and Shluger, A. L. (2018). Recombination defects at the 4H-SiC/SiO2 interface investigated with electrically detected magnetic resonance and ab initio calculations. *Journal of Applied Physics*, 124(4):045302.

Daigle, K. and Staff, G. (2023). Octoverse: The state of open source and rise of ai in 2023.

Dhar, S., Wang, S., Williams, J. R., Pantelides, S. T., and Feldman, L. C. (2005). Interface passivation for silicon dioxide layers on silicon carbide. *MRS Bulletin*, 30(4):288–292.

Fang, L., Lan, Z., Guan, W., Zhou, P., Bahlawane, N., Sun, W., Lu, Y., Liang, C., Yan, M., and Jiang, Y. (2019). Hetero-interface constructs ion reservoir to enhance conversion reaction kinetics for sodium/lithium storage. *ENERGY STORAGE MATERIALS*, 18:107–113.

Glass, C. W., Oganov, A. R., and Hansen, N. (2006). Uspex—evolutionary crystal structure prediction. *Computer Physics Communications*, 175(11):713–720.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Heer, N. (2023). Github - niklas-heer/speed-comparison: A repo which compares the speed of different programming languages.

Johnston, R. L. (2003). Evolving better nanoparticles: Genetic algorithms for optimising cluster geometries. *Dalton Trans.*, pages 4193–4207.

Kühne, T. D., Iannuzzi, M., Del Ben, M., Rybkin, V. V., Seewald, P., Stein, F., Laino, T., Khaliullin, R. Z., Schütt, O., Schiffmann, F., Golze, D., Wilhelm, J., Chulkov, S., Bani-Hashemian, M. H., Weber, V., Borštnik, U., Taillefumier, M., Jakobovits, A. S., Lazzaro, A., Pabst, H., Müller, T., Schade, R., Guidon, M., Andermatt, S., Holmberg, N., Schenter, G. K., Hehn, A., Bussy, A., Belleflamme, F., Tabacchi, G., Glöß, A., Lass, M., Bethune, I., Mundy, C. J., Plessl, C., Watkins, M., VandeVondele, J., Krack, M., and Hutter, J. (2020). CP2K: An electronic structure and molecular dynamics software package - Quickstep: Efficient and accurate electronic structure calculations. *The Journal of Chemical Physics*, 152(19):194103.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA. Association for Computing Machinery.

Larsen, A. H., Mortensen, J. J., Blomqvist, J., Castelli, I. E., Christensen, R., Dułak, M., Friis, J., Groves, M. N., Hammer, B., Hargus, C., Hermes, E. D., Jennings, P. C., Jensen, P. B., Kermode, J., Kitchin, J. R., Kolsbjerg, E. L., Kubal, J., Kaasbjerg, K., Lysgaard, S., Maronsson, J. B., Maxson, T., Olsen, T., Pastewka, L., Peterson, A., Rostgaard, C., Schiøtz, J., Schütt, O., Strange, M., Thygesen, K. S., Vegge, T., Vilhelmsen, L., Walter, M., Zeng, Z., and Jacobsen, K. W. (2017). The atomic simulation environment—a python library for working with atoms. *Journal of Physics: Condensed Matter*, 29(27):273002.

Li, W.-L., Chen, K., Rossomme, E., Head-Gordon, M., and Head-Gordon, T. (2021). Optimized Pseudopotentials and Basis Sets for Semiempirical Density Functional Theory for Electrocatalysis Applications. *The Journal of Physical Chemistry Letters*, 12(42):10304–10309. PMID: 34653336.

Maneewongvatana, S. and Mount, D. M. (1999). Analysis of approximate nearest neighbor searching with clustered point sets. *CoRR*, cs.CG/9901013.

Matsakis, N. D. and Klock, F. S. (2014). The rust language. *Ada Lett.*, 34(3):103–104.

Millán, J., Godignon, P., Perpiñà, X., Pérez-Tomás, A., and Rebollo, J. (2014). A survey of wide bandgap power semiconductor devices. *IEEE Transactions on Power Electronics*, 29(5):2155–2163.

Pickard, C. J. and Needs, R. J. (2011). Ab initio random structure searching. *Journal of Physics: Condensed Matter*, 23(5):053201.

Presser, V. and Nickel, K. G. (2008). Silica on Silicon Carbide. *Critical Reviews in Solid State and Materials Sciences*, 33(1):1–99.

Roussel, P. (2011). Sic market and industry update. In *Int. SiC Power Electron. Appl. Workshop, Kista, Sweden*.

She, X., Huang, A. Q., Lucía, Ó., and Ozpineci, B. (2017). Review of silicon carbide power devices and their applications. *IEEE Transactions on Industrial Electronics*, 64(10):8193–8205.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

Yang, B., Luo, D., Wu, S., Zhang, N., and Ye, J. (2022). Nanoscale hetero-interfaces for electrocatalytic and photocatalytic water splitting. *SCIENCE AND TECHNOLOGY OF ADVANCED MATERIALS*, 23(1):587–616.

Zur, A. and McGill, T. C. (1984). Lattice match: An application to heteroepitaxy. *Journal of Applied Physics*, 55(2):378–386.

A LINK TO THE CODE

This is a link to the Github repository of the Crystacean code used for this thesis:
`https://github.com/pluumenbrownie/lattice_solver/tree/FrozenForThesis`

B FULL RUST LIB.RS FILE

The following is the Rust part of the Crystacean code.

```rust
use fixedbitset::FixedBitSet;
use itertools::{izip, zip_eq, Itertools};
use json::{object, JsonValue};
use kdam::{tqdm, Colour, Spinner};
use std::{
    collections::HashSet, f32::consts::PI, ffi::OsString, fs::File,
    ↪ io::{stderr, IsTerminal}, iter::zip, mem, sync::{Arc, RwLock}
};

use kiddo::{KdTree, SquaredEuclidean};
use std::io::prelude::*;

const TRIPLE_CROWN: [f32; 9] = [
    -0.000082766,
    -1.397718937,
    -0.517599594,

    -1.222253196,
    0.702780512,
    -0.517599594,

```

```rust
    1.204777673,
    0.711865236,
    -0.517599594,
];
const DOUBLE_CROWN: [f32; 6] = [
    2.2252961107321143 - 1.0526814404964109,
    -1.3867293215799141 - -1.0917258490752173,
    -(9.26392293591872 - 8.41953003801284),

    2.2252961107321143 - 3.3223088674933625,
    -1.3867293215799141 - -1.6248356621955473,
    -(9.26392293591872 - 8.313286837087986),
];
const SINGLE_CROWN: [f32; 3] = [0.0, 0.0, -1.7];

fn double_crown_rotated(theta: f32) -> [f32; 6] {
    // println!("{theta:?}");
    let r = ((2.2252961107321143 - 1.0526814404964109 as f32).powi(2)
     ↪  + (-1.3867293215799141 - -1.0917258490752173 as
     ↪  f32).powi(2)).sqrt();
    [
        r*(theta).cos(),
        r*(theta).sin(),
        -(9.26392293591872 - 8.41953003801284),

        r*(theta + PI).cos(),
        r*(theta + PI).sin(),
        -(9.26392293591872 - 8.313286837087986),
    ]
}

fn triple_crown_rotated(theta: f32) -> [f32; 9] {
    // println!("{theta:?}");
    let r = ((-0.000082766 as f32).powi(2) + (-1.397718937 as
     ↪  f32).powi(2)).sqrt();
    [
        r*(theta).cos(),
        r*(theta).sin(),
        -0.517599594,

        r*(theta + (2.0/3.0 * PI)).cos(),
        r*(theta + (2.0/3.0 * PI)).sin(),
        -0.517599594,

        r*(theta - (2.0/3.0 * PI)).cos(),
        r*(theta - (2.0/3.0 * PI)).sin(),
        -0.517599594,
    ]
}

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
struct OxygenIndex(usize);

#[derive(Clone, Copy, Debug)]
struct LatticeIndex(usize);
```

```rust

#[derive(Debug)]
struct LatticePoint {
    x: f32,
    y: f32,
    z: f32,
    connected_to: RwLock<Vec<OxygenIndex>>,
    ghost_to: Option<Arc<LatticePoint>>,
}

impl LatticePoint {
    fn new(x: f32, y: f32, z: f32, ghost_to:
     ↪  Option<Arc<LatticePoint>>) -> Arc<Self> {
        Arc::new(LatticePoint {
            x,
            y,
            z,
            connected_to: RwLock::new(vec![]),
            ghost_to,
        })
    }

    fn get_connections(&self) -> &RwLock<Vec<OxygenIndex>> {
        match &self.ghost_to {
            None => &self.connected_to,
            Some(point) => &point.connected_to,
        }
    }

    fn distance_squared_to(&self, other: &LatticePoint) -> f32 {
        (self.x - other.x).powi(2) + (self.y - other.y).powi(2) +
         ↪  (self.z - other.z).powi(2)
    }
}

#[derive(Clone)]
struct Oxygen {
    x: f32,
    y: f32,
    z: f32,
    sitetype: SiteType,
    exclusions: Vec<OxygenIndex>,
}

impl Oxygen {
    fn new(x: f32, y: f32, z: f32, sitetype: SiteType) -> Self {
        Oxygen {
            x,
            y,
            z,
            sitetype,
            exclusions: vec![],
        }
    }
}
```

```rust

#[derive(Clone, Copy)]
enum SiteType {
    Singlet(Singlet),
    Midpoint(Midpoint),
    Tripoint(Tripoint),
}

impl SiteType {
    fn iter(&self) -> std::slice::Iter<'_, LatticeIndex> {
        match self {
            Self::Tripoint(c) => c.0.iter(),
            Self::Midpoint(c) => c.0.iter(),
            Self::Singlet(c) => c.0.iter(),
        }
    }
}

#[derive(Clone, Copy)]
struct Singlet([LatticeIndex; 1]);

#[derive(Clone, Copy)]
struct Midpoint([LatticeIndex; 2]);

#[derive(Clone, Copy)]
struct Tripoint([LatticeIndex; 3]);

#[derive(Debug, PartialEq)]
pub struct BitArraySolution(pub FixedBitSet);

impl BitArraySolution {
    /// Convert a solution from a filtered `BitArrayRepresentation`
     ↪  back into
    /// it's full form.
    ///
    /// ```
    /// use lattice_solver::BitArraySolution;
    /// use fixedbitset::FixedBitSet;
    ///
    /// let mut compressed = BitArraySolution(
    ///     FixedBitSet::with_capacity_and_blocks(4, vec![0b10110])
    /// );
    /// let full = BitArraySolution(
    ///     FixedBitSet::with_capacity_and_blocks(6,
     ↪  vec![0b1001010000])
    /// );
    /// let filter = FixedBitSet::with_capacity_and_blocks(6,
     ↪  vec![0b1101011000]);
    ///
    /// compressed.inflate(&filter);
    /// assert_eq!(compressed, full);
    /// ```
    /// Shown schematically:
    /// ```text
    /// 10 1 10
```

```rust
    /// 1101011000
    /// 1001010000
    /// ```
    pub fn inflate(&mut self, filter_bitset: &FixedBitSet) {
        let mut new_vector =
        ↪ FixedBitSet::with_capacity(filter_bitset.len());
        for (self_number, new_location) in
        ↪ filter_bitset.ones().enumerate() {
            new_vector.set(new_location, self.0[self_number]);
        }
        self.0 = new_vector;
    }

    pub fn __str__(&self) -> String {
        format!("{}", self.0)
    }
}

pub struct BitArrayRepresentation {
    filled_sites: FixedBitSet,
    exclusion_matrix: Vec<FixedBitSet>,
    tripoint_mask: FixedBitSet,
    midpoint_mask: FixedBitSet,
    singlet_mask: FixedBitSet,
    filter: Option<FixedBitSet>,
}

impl BitArrayRepresentation {
    fn matrix_vector_multiply(&self, vector: &FixedBitSet) ->
    ↪ FixedBitSet {
        let mut output_vector =
        ↪ FixedBitSet::with_capacity(vector.len());
        for bit_nr in 0..output_vector.len() {
            let mut enabled = true;
            // equal to: (vector &
            ↪ &self.exclusion_matrix[bit_nr]).is_clear()
            for (v, m) in zip_eq(vector.as_slice(),
            ↪ self.exclusion_matrix[bit_nr].as_slice()) {
                if (v & m) != 0u32 {
                    enabled = false;
                    break;
                }
            }
            output_vector.set(bit_nr, enabled);
        }
        output_vector
    }

    pub fn get_possibilities(&self, vector: &FixedBitSet) ->
    ↪ FixedBitSet {
        let non_singlet_mask: FixedBitSet = &self.tripoint_mask |
        ↪ &self.midpoint_mask;
        let possibilities: FixedBitSet =
        ↪ self.matrix_vector_multiply(vector);
```

```
223             let masked_possibilities = &possibilities &
                ↪  &non_singlet_mask;
224
225             if masked_possibilities.is_clear() {
226                 &possibilities & &self.singlet_mask
227             } else {
228                 masked_possibilities
229             }
230         }
231
232         pub fn get_bitarray(&self) -> FixedBitSet {
233             self.filled_sites.clone()
234         }
235
236         pub fn filtered(&self, filter: SiteFilter) ->
                ↪  BitArrayRepresentation {
237             // println!("{filter:?}");
238             let mut filter_set =
                    ↪  FixedBitSet::with_capacity(self.filled_sites.len());
239             filter_set.toggle_range(..);
240             for number in filter.wrapped {
241                 filter_set.set(number.0, false);
242             }
243             let new_length = filter_set.count_ones(..);
244
245             let filled_sites = FixedBitSet::with_capacity(new_length);
246
247             let mut tripoint_mask =
                    ↪  FixedBitSet::with_capacity(new_length);
248             let mut midpoint_mask =
                    ↪  FixedBitSet::with_capacity(new_length);
249             let mut singlet_mask =
                    ↪  FixedBitSet::with_capacity(new_length);
250
251             let mut exclusion_matrix = vec![];
252
253             for (new_number, old_number) in filter_set.ones().enumerate()
                    ↪  {
254                 tripoint_mask.set(new_number,
                        ↪  self.tripoint_mask[old_number]);
255                 midpoint_mask.set(new_number,
                        ↪  self.midpoint_mask[old_number]);
256                 singlet_mask.set(new_number,
                        ↪  self.singlet_mask[old_number]);
257
258                 let mut new_matrix_row =
                        ↪  FixedBitSet::with_capacity(new_length);
259                 for (col_number, old_col_number) in
                        ↪  filter_set.ones().enumerate() {
260                     new_matrix_row.set(
261                         col_number,
262
                                ↪  self.exclusion_matrix[old_number][old_col_number].into(
263                     );
264                 }
```

```
            exclusion_matrix.push(new_matrix_row);
        }

        BitArrayRepresentation {
            filled_sites,
            exclusion_matrix,
            tripoint_mask,
            midpoint_mask,
            singlet_mask,
            filter: Some(filter_set),
        }
    }

    pub fn solve(&self, find_all: bool) -> Vec<BitArraySolution> {
        let test_lattice = self.filled_sites.clone();

        let mut current_generation = HashSet::from([test_lattice]);
        let mut next_generation = HashSet::new();
        let mut depth = 0;
        let mut solutions = vec![];
        kdam::term::init(stderr().is_terminal());

        while solutions.is_empty() | (find_all &
         ↪  (!next_generation.is_empty())) {
            depth += 1;

            next_generation.clear();

            let iterator = tqdm!(
                current_generation.iter(),
                desc = format!("Current depth: {depth}"),
                mininterval = 1.0/60.0,
                bar_format = "{desc suffix=' '}|{animation}|
                 ↪  {spinner} {count}/{total} [{percentage:.0}%] in
                 ↪  {elapsed human=true} ({rate:.1}/s, eta:
                 ↪  {remaining human=true})",
                colour = Colour::gradient(&["#5A56E0", "#EE6FF8"]),
                spinner = Spinner::new(
                    "*CENSORED UTF-8 CHARACTERS*"
                    60.0,
                    1.0,
                )
            );
            for candidate in iterator {
                let possibilities =
                 ↪  self.get_possibilities(candidate);

                if possibilities.is_clear() {

                     ↪  solutions.push(BitArraySolution(candidate.clone()));
                    continue;
                }

                for fillable_site in possibilities.ones() {
                    let mut new_candidate = candidate.clone();
```

```
                    new_candidate.set(fillable_site, true);

                    next_generation.insert(new_candidate);
                }
            }

            mem::swap(&mut current_generation, &mut next_generation);
        }
        if let Some(filter) = &self.filter {
            for solution in &mut solutions {
                solution.inflate(filter);
            }
        }
        solutions
    }

    pub fn __str__(&self) -> String {
        let mut output = String::from("BitArrayRepresentation: {\n");

        output += format!("  filled_sites = \n    {}\n",
        ↪  self.filled_sites).as_str();
        output += "  exclusion_matrix = \n";
        for (number, row) in self.exclusion_matrix.iter().enumerate()
        ↪  {
            output += format!("{number:5}. {}\n", row).as_str();
        }
        output += format!("  tripoint_mask = \n    {}\n",
        ↪  self.tripoint_mask).as_str();
        output += format!("  midpoint_mask = \n    {}\n",
        ↪  self.midpoint_mask).as_str();
        output += format!("  singlet_mask = \n    {}\n",
        ↪  self.singlet_mask).as_str();
        output += format!("  filter = \n    ").as_str();
        output += match &self.filter {
            None => "None".into(),
            Some(fbs) => format!("{}", fbs),
        }
        .as_str();
        output += "\n}";

        output
    }

    pub fn __repr__(&self) -> String {
        format!("BitArrayRepresentation[{}]", self.filled_sites)
    }
}

#[derive(Clone, Debug)]
pub struct SiteFilter {
    wrapped: Vec<OxygenIndex>,
}

impl SiteFilter {
    pub fn empty() -> Self {
```

```
364             SiteFilter { wrapped: vec![] }
365         }
366     }
367
368     pub struct Lattice {
369         points: Vec<Arc<LatticePoint>>,
370         oxygens: Vec<Oxygen>,
371         source_file: Option<JsonValue>,
372     }
373
374     impl Lattice {
375         /// Create an empty `Lattice`
376         fn new() -> Self {
377             Lattice {
378                 points: vec![],
379                 oxygens: vec![],
380                 source_file: None,
381             }
382         }
383
384         /// Push a point to `self.points`
385         fn add_point(&mut self, new_point: Arc<LatticePoint>) {
386             self.points.push(new_point);
387         }
388
389         fn generate_exclusions(&mut self) {
390             for (number, oxygen) in self.oxygens.iter().enumerate() {
391                 for index in oxygen.sitetype.iter() {
392                     let mut connections =
                         ↪ self.points[index.0].get_connections().write().unwrap();
393                     connections.push(OxygenIndex(number));
394                 }
395             }
396             for oxygen in &mut self.oxygens {
397                 for connection in oxygen.sitetype.iter() {
398                     let point = &self.points[connection.0];
399                     for ox_con in
                         ↪ point.get_connections().read().unwrap().iter() {
400                         oxygen.exclusions.push(*ox_con);
401                     }
402                 }
403                 oxygen.exclusions.dedup();
404             }
405         }
406
407         /// Turns the lattice problem in an abstracted form based on
            ↪ bitarrays.
408         fn generate_intermediary(&self) -> BitArrayRepresentation {
409             let filled_sites =
                 ↪ FixedBitSet::with_capacity(self.oxygens.len());
410             let mut exclusion_matrix: Vec<FixedBitSet> = vec![];
411
412             let mut tripoint_mask =
                 ↪ FixedBitSet::with_capacity(self.oxygens.len());
```

```rust
        let mut midpoint_mask =
         ↪  FixedBitSet::with_capacity(self.oxygens.len());
        let mut singlet_mask =
         ↪  FixedBitSet::with_capacity(self.oxygens.len());

        for (number, oxygen) in self.oxygens.iter().enumerate() {
            let mut exclusions =
             ↪  FixedBitSet::with_capacity(self.oxygens.len());
            for exclusion in &oxygen.exclusions {
                exclusions.set(exclusion.0, true);
            }
            match &oxygen.sitetype {
                SiteType::Tripoint(_) => tripoint_mask.set(number,
                 ↪  true),
                SiteType::Midpoint(_) => midpoint_mask.set(number,
                 ↪  true),
                SiteType::Singlet(_) => singlet_mask.set(number,
                 ↪  true),
            };
            exclusion_matrix.push(exclusions);
        }

        BitArrayRepresentation {
            filled_sites,
            exclusion_matrix,
            tripoint_mask,
            midpoint_mask,
            singlet_mask,
            filter: None,
        }
    }

    /// distance_margin should be 1.1 for 2D, 1.4 for 3D
    pub fn python_new(
        input_lattice: Vec<(Vec<f32>, Vec<Vec<f32>>)>,
        distance_margin: f32,
        autodetect_margin: bool,
    ) -> Self {
        // Convert 2D structures to 3D
        let lattice_3d = turn_2d_3d(input_lattice);

        // Create the silicon lattice
        let mut out_lattice = create_silicon_lattice(lattice_3d);

        let first_point_location = {
            let first_point = &out_lattice.points[0];
            [first_point.x, first_point.y, first_point.z]
        };

        // Fill in the oxygens
        let silicon_iterator = out_lattice
            .points
            .iter()
            .map(|p| [p.x, p.y, p.z])
            .collect_vec();
```

```rust
        let kdtree: KdTree<_, 3> = (&silicon_iterator).into();
        let node_search_distance = if autodetect_margin {
            kdtree.nearest_n::<SquaredEuclidean>(&first_point_location, 2)[1].distance
                * distance_margin
        } else {
            distance_margin.powi(2)
        };
        // println!("{node_search_distance:?}");

        // Tripoints
        let mut covered_sites = HashSet::new();
        for (number, silicon) in out_lattice.points.iter().enumerate() {
            let mut close_points = kdtree.within::<SquaredEuclidean>(
                &[silicon.x, silicon.y, silicon.z],
                node_search_distance,
            );
            // Sort results on lattice number
            close_points.sort_by_key(|p| p.item);
            let sites = close_points
                .iter()
                .filter(|s| s.item as usize != number)
                .combinations(2)
                .filter(|a| {
                    let mut identifier = [number as u64, a[0].item, a[1].item];
                    identifier.sort();
                    covered_sites.insert(identifier)
                })
                .filter(|a| {
                    out_lattice.points[a[0].item as usize].x
                        != out_lattice.points[a[1].item as usize].x
                })
                .filter(|a| {
                    out_lattice.points[number].ghost_to.is_none()
                        || out_lattice.points[a[0].item as usize].ghost_to.is_none()
                        || out_lattice.points[a[1].item as usize].ghost_to.is_none()
                })
                .filter(|a| {
                    out_lattice.points[a[0].item as usize]
                        .distance_squared_to(&out_lattice.points[a[1].item as usize])
                        <= node_search_distance
                });

            for site in sites {
                let x = (out_lattice.points[number].x
                    + out_lattice.points[site[0].item as usize].x
                    + out_lattice.points[site[1].item as usize].x)
                    / 3.0;
```

```
                let y = (out_lattice.points[number].y
                    + out_lattice.points[site[0].item as usize].y
                    + out_lattice.points[site[1].item as usize].y)
                    / 3.0;
                let z = (out_lattice.points[number].z
                    + out_lattice.points[site[0].item as usize].z
                    + out_lattice.points[site[1].item as usize].z)
                    / 3.0
                    - 1.1;
                let sitetype = SiteType::Tripoint(Tripoint([
                    LatticeIndex(number),
                    LatticeIndex(site[0].item as usize),
                    LatticeIndex(site[1].item as usize),
                ]));
                out_lattice.oxygens.push(Oxygen::new(x, y, z,
                    sitetype))
            }
        }

        // Midpoints
        for (number, silicon) in
            out_lattice.points.iter().enumerate() {
            let mut close_points = kdtree.within::<SquaredEuclidean>(
                &[silicon.x, silicon.y, silicon.z],
                node_search_distance,
            );
            // Sort results on lattice number
            close_points.sort_by_key(|p| p.item);
            let sites = close_points
                .iter()
                .skip(1)
                .filter(|s| s.item as usize > number)
                .filter(|s| {
                    out_lattice.points[number].ghost_to.is_none()
                        || out_lattice.points[s.item as
                            usize].ghost_to.is_none()
                });

            for site in sites {
                let x =
                    (out_lattice.points[number].x +
                        out_lattice.points[site.item as usize].x) /
                        2.0;
                let y =
                    (out_lattice.points[number].y +
                        out_lattice.points[site.item as usize].y) /
                        2.0;
                let z = (out_lattice.points[number].z +
                    out_lattice.points[site.item as usize].z)
                    / 2.0
                    - 1.4;
                let sitetype = SiteType::Midpoint(Midpoint([
                    LatticeIndex(number),
                    LatticeIndex(site.item as usize),
                ]));
```

```
                out_lattice.oxygens.push(Oxygen::new(x, y, z,
                    ↪ sitetype))
            }
        }

        // Singles
        for (number, silicon) in out_lattice
            .points
            .iter()
            .enumerate()
            .filter(|s| s.1.ghost_to.is_none())
        {
            out_lattice.oxygens.push(Oxygen::new(
                silicon.x,
                silicon.y,
                silicon.z - 1.7,
                SiteType::Singlet(Singlet([LatticeIndex(number)])),
            ));
        }

        out_lattice.generate_exclusions();
        out_lattice
    }

    fn add_source_file(&mut self, source_file: JsonValue) {
        self.source_file = Some(source_file);
    }

    pub fn from_dft_json(filename: String, distance_margin: f32,
    ↪ autodetect_margin: bool) -> Self {
        let mut buffer = String::new();
        let mut file = File::open(filename).expect("Opening file
        ↪ failed.");
        file.read_to_string(&mut buffer)
            .expect("Reading file failed.");
        let parsed = json::parse(&buffer).expect("Parsing file
        ↪ failed/");

        let last_id = &parsed["ids"][parsed["ids"].len() -
        ↪ 1].to_string();

        // let hydrogen_amount =
        ↪ &parsed[last_id]["numbers"]["__ndarray__"][2]
        //     .members()
        //     .filter_map(|n| n.as_usize())
        //     .filter(|n| n == &1)
        //     .count();

        let numbers =
        ↪ &parsed[last_id]["positions"]["__ndarray__"][2];
        let atoms = numbers
            .members()
            .map(|j| j.as_f32().unwrap())
            .collect_vec()
            .chunks_exact(3)
```

```
            .map(|c| c.to_vec())

            .zip(parsed[last_id]["numbers"]["__ndarray__"][2].members().map(|i| i.as_i32()))
            .collect_vec();

        let hydrogenated_ends = atoms
            .iter()
            .filter(|(_, i)| i == &Some(1))
            .map(|(v, _)| v)
            .filter(|v| v[2] < 20.0)
            .collect_vec();

        let cell = &parsed[last_id]["cell"]["array"]["__ndarray__"][2];
        let (x_vec, y_vec, _z_vec) = cell
            .members()
            .map(|v| v.as_f32().unwrap())
            .tuples::<(_, _, _)>()
            .collect_tuple()
            .expect("Json 'cell' property format is incorrect.");

        let input_lattice = {
            let mut input_lattice = vec![];
            for end in hydrogenated_ends {
                let mut new_point = (end.clone(), vec![]);
                new_point
                    .1
                    .push(vec![end[0] + x_vec.0, end[1] + x_vec.1, end[2] + x_vec.2]);
                new_point
                    .1
                    .push(vec![end[0] + y_vec.0, end[1] + y_vec.1, end[2] + y_vec.2]);
                new_point.1.push(vec![
                    end[0] + x_vec.0 + y_vec.0,
                    end[1] + x_vec.1 + y_vec.1,
                    end[2] + x_vec.2 + y_vec.2,
                ]);
                new_point.1.push(vec![
                    end[0] + x_vec.0 - y_vec.0,
                    end[1] + x_vec.1 - y_vec.1,
                    end[2] + x_vec.2 - y_vec.2,
                ]);
                new_point.1.push(vec![
                    end[0] - x_vec.0 + y_vec.0,
                    end[1] - x_vec.1 + y_vec.1,
                    end[2] - x_vec.2 + y_vec.2,
                ]);
                input_lattice.push(new_point);
            }
            input_lattice
        };
```

```
        let mut lattice = Lattice::python_new(input_lattice,
         ↪  distance_margin, autodetect_margin);
        lattice.add_source_file(parsed.clone());

        lattice
    }

    pub fn diagnostic_ase(&self) {
        let parsed = self.source_file.as_ref().unwrap();
        let oxygens = &self.oxygens;
        let last_id = &parsed["ids"][parsed["ids"].len() -
         ↪  1].to_string();

        let mut new_numbers = parsed[last_id]["numbers"].clone();
        let mut new_positions = parsed[last_id]["positions"].clone();

        println!("Added {} oxygens.", oxygens.len());

        for oxygen in oxygens {
            new_numbers["__ndarray__"][2]
                .push(8)
                .expect("new_numbers[\"__ndarray__\"][2].push(8)");
            new_positions["__ndarray__"][2]
                .push(oxygen.x)

                 ↪  .expect("new_positions[\"__ndarray__\"][2].push(oxygen.x)")
            new_positions["__ndarray__"][2]
                .push(oxygen.y)

                 ↪  .expect("new_positions[\"__ndarray__\"][2].push(oxygen.y)")
            new_positions["__ndarray__"][2]
                .push(oxygen.z)

                 ↪  .expect("new_positions[\"__ndarray__\"][2].push(oxygen.z)")
        }
        new_numbers["__ndarray__"][0][0] =
         ↪  new_numbers["__ndarray__"][2].len().into();
        new_positions["__ndarray__"][0][0] =
         ↪  new_numbers["__ndarray__"][2].len().into();

        let mut export_data = json::JsonValue::new_object();
        export_data["1"] = object! {
            cell: parsed[last_id]["cell"].clone(),
            ctime: parsed[last_id]["ctime"].clone(),
            mtime: parsed[last_id]["mtime"].clone(),
            numbers: new_numbers,
            pbc: parsed[last_id]["pbc"].clone(),
            positions: new_positions,
            unique_id: "Not unique",
            user: parsed[last_id]["user"].clone(),
        };

        let mut file = File::create("output.json").unwrap();
        file.write_all(export_data.pretty(4).as_bytes()).unwrap();
    }
```

```rust

    pub fn no_rings_plot(&self) -> Vec<(usize, f32, f32)> {
        let parsed = self.source_file.as_ref().unwrap();

        let last_id = &parsed["ids"][parsed["ids"].len() -
        ↪  1].to_string();

        let numbers =
        ↪  &parsed[last_id]["positions"]["__ndarray__"][2];
        let atoms = numbers
            .members()
            .map(|j| j.as_f32().unwrap())
            .collect_vec()
            .chunks_exact(3)
            .map(|c| c.to_vec())
            .collect_vec();

        let top_silicon_locations = atoms
            .iter()
            .sorted_by(|&a, &b| a[2].total_cmp(&b[2]))
            .filter(|&s| (s[2] > 8.4) & (s[2] < 10.0))
            .map(|v| [v[0], v[1], v[2]])
            .collect_vec();

        let points_vector = self.points.iter().map(|p| [p.x, p.y,
        ↪  p.z]).collect_vec();
        println!("{points_vector:?}");
        println!("length: {}", points_vector.len());
        let points_tree: KdTree<_, 3> = (&points_vector).into();

        let point_group_vector = {
            let mut point_group_vector = vec![1000;
            ↪  self.points.len()];

            for (number, atom) in
            ↪  top_silicon_locations.iter().enumerate() {
                let close_points =
                ↪  points_tree.within::<SquaredEuclidean>(atom,
                ↪  1.7f32.powi(2));
                println!("{:?} --- {:?}", atom, close_points);
                for point in close_points {
                    point_group_vector[point.item as usize] = number;
                }
            }
            point_group_vector
        };
        println!("{point_group_vector:?}");

        izip!(
            point_group_vector,
            self.points.iter().map(|o| o.x),
            self.points.iter().map(|o| o.y)
        )
        .collect_vec()
    }
```

```rust

    pub fn no_rings(&self) -> SiteFilter {
        let parsed = self.source_file.as_ref().unwrap();

        let last_id = &parsed["ids"][parsed["ids"].len() -
        ↪  1].to_string();

        let numbers =
        ↪  &parsed[last_id]["positions"]["__ndarray__"][2];
        let atoms = numbers
            .members()
            .map(|j| j.as_f32().unwrap())
            .collect_vec()
            .chunks_exact(3)
            .map(|c| c.to_vec())
            .collect_vec();

        let top_silicon_locations = atoms
            .iter()
            .sorted_by(|&a, &b| a[2].total_cmp(&b[2]))
            .filter(|&s| (s[2] > 8.4) & (s[2] < 10.0))
            .map(|v| [v[0], v[1], v[2]])
            .collect_vec();

        let points_vector = self.points.iter().map(|p| [p.x, p.y,
        ↪  p.z]).collect_vec();
        // println!("{points_vector:?}");
        // println!("length: {}", points_vector.len());
        let points_tree: KdTree<_, 3> = (&points_vector).into();

        let point_group_vector = {
            let mut point_group_vector = vec![1000;
            ↪  self.points.len()];

            for (number, atom) in
            ↪  top_silicon_locations.iter().enumerate() {
                let close_points =
                ↪  points_tree.within::<SquaredEuclidean>(atom,
                ↪  1.7f32.powi(2));
                // println!("{:?} --- {:?}", atom, close_points);
                for point in close_points {
                    point_group_vector[point.item as usize] = number;
                }
            }
            point_group_vector
        };
        // println!("{point_group_vector:?}");

        let mut disabled_oxygens = vec![];

        for (number, oxygen) in self.oxygens.iter().enumerate() {
            match oxygen.sitetype {
                SiteType::Singlet(_) => {}
                SiteType::Midpoint(p) => {
                    let connections = p.0;
```

```
                    let same_group = |a: usize, b: usize| {
                        point_group_vector[connections[a].0] ==
                         ↪  point_group_vector[connections[b].0]
                    };

                    if same_group(0, 1) {
                        disabled_oxygens.push(OxygenIndex(number));
                    };
                }
                SiteType::Tripoint(p) => {
                    let connections = p.0;
                    let same_group = |a: usize, b: usize| {
                        point_group_vector[connections[a].0] ==
                         ↪  point_group_vector[connections[b].0]
                    };

                    if same_group(0, 1) | same_group(1, 2) |
                     ↪  same_group(0, 2) {
                        disabled_oxygens.push(OxygenIndex(number));
                    };
                }
            }
        }

        SiteFilter {
            wrapped: disabled_oxygens,
        }
    }

    /// Returns the coordinates of the lattice points in two lists.
    /// Use with the * star operator in a `plt.plot` function:
    ///
    /// ```python
    /// plt.plot(*solved_lattice.points_to_plot(), "o")
    /// ```
    pub fn points_to_plot(&self) -> (Vec<f32>, Vec<f32>) {
        let x_points = self.points.iter().map(|p| p.x).collect_vec();
        let y_points = self.points.iter().map(|p| p.y).collect_vec();
        (x_points, y_points)
    }

    /// Returns the coordinates of the oxygen points in two lists.
    /// Use with the * star operator in a `plt.plot` function:
    ///
    /// ```python
    /// plt.plot(*solved_lattice.oxygens_to_plot(), "o")
    /// ```
    pub fn oxygens_to_plot(&self) -> (Vec<f32>, Vec<f32>) {
        let x_points = self.oxygens.iter().map(|p|
         ↪  p.x).collect_vec();
        let y_points = self.oxygens.iter().map(|p|
         ↪  p.y).collect_vec();
        (x_points, y_points)
    }

```

```rust
    /// Returns the coordinates of the tripoints in two lists.
    /// Use with the * star operator in a `plt.plot` function:
    ///
    /// ```python
    /// plt.plot(*solved_lattice.tripoints_to_plot(), "o")
    /// ```
    pub fn tripoints_to_plot(&self) -> (Vec<f32>, Vec<f32>) {
        let x_points = self
            .oxygens
            .iter()
            .filter(|p| matches!(p.sitetype, SiteType::Tripoint(_)))
            .map(|p| p.x)
            .collect_vec();
        let y_points = self
            .oxygens
            .iter()
            .filter(|p| matches!(p.sitetype, SiteType::Tripoint(_)))
            .map(|p| p.y)
            .collect_vec();
        (x_points, y_points)
    }

    /// Returns the coordinates of the tripoints in two lists.
    /// Use with the * star operator in a `plt.plot` function:
    ///
    /// ```python
    /// plt.plot(*solved_lattice.midpoints_to_plot(), "o")
    /// ```
    pub fn midpoints_to_plot(&self) -> (Vec<f32>, Vec<f32>) {
        let x_points = self
            .oxygens
            .iter()
            .filter(|p| matches!(p.sitetype, SiteType::Midpoint(_)))
            .map(|p| p.x)
            .collect_vec();
        let y_points = self
            .oxygens
            .iter()
            .filter(|p| matches!(p.sitetype, SiteType::Midpoint(_)))
            .map(|p| p.y)
            .collect_vec();
        (x_points, y_points)
    }

    /// Returns the coordinates of the tripoints in two lists.
    /// Use with the * star operator in a `plt.plot` function:
    ///
    /// ```python
    /// plt.plot(*solved_lattice.singlets_to_plot(), "o")
    /// ```
    pub fn singlets_to_plot(&self) -> (Vec<f32>, Vec<f32>) {
        let x_points = self
            .oxygens
            .iter()
            .filter(|p| matches!(p.sitetype, SiteType::Singlet(_)))
```

```
            .map(|p| p.x)
            .collect_vec();
        let y_points = self
            .oxygens
            .iter()
            .filter(|p| matches!(p.sitetype, SiteType::Singlet(_)))
            .map(|p| p.y)
            .collect_vec();
        (x_points, y_points)
    }

    /// Generates a more efficient representation of the lattice
    /// problem for the given lattice.
    pub fn get_intermediary(&self) -> BitArrayRepresentation {
        self.generate_intermediary()
    }

    /// Returns a solved version of the lattice. Usefull for plotting
    /// and exporting.
    pub fn to_solved_lattice(&self, solution: &BitArraySolution) ->
     ↪  Self {
        let mut solved_oxygens = vec![];
        for number in 0..self.oxygens.len() {
            if solution.0[number] {
                solved_oxygens.push(self.oxygens[number].clone());
            }
        }
        Lattice {
            points: self.points.clone(),
            oxygens: solved_oxygens,
            source_file: self.source_file.clone(),
        }
    }

    /// Export the solved lattice to a json file.
    ///
    /// - path must be a valid path name.
    /// - name should end with ".json".
    pub fn export(&self, path: OsString, name: String) {
        let mut data = json::JsonValue::new_object();
        {
            let mut points = vec![];
            for point in &self.points {
                let mut new_obj = json::JsonValue::new_object();
                new_obj["x"] = point.x.into();
                new_obj["y"] = point.y.into();
                new_obj["ghost"] = point.ghost_to.is_some().into();
                points.push(new_obj);
            }
            data["lattice_points"] = points.into();
        }

        {
            let mut tripoints = vec![];
            let mut midpoints = vec![];
```

```
            let mut singles = vec![];
            for oxygen in &self.oxygens {
                let mut new_obj = json::JsonValue::new_object();
                new_obj["x"] = oxygen.x.into();
                new_obj["y"] = oxygen.y.into();

                match oxygen.sitetype {
                    SiteType::Tripoint(_) => tripoints.push(new_obj),
                    SiteType::Midpoint(_) => midpoints.push(new_obj),
                    SiteType::Singlet(_) => singles.push(new_obj),
                };
            }
            data["tripoints"] = tripoints.into();
            data["midpoints"] = midpoints.into();
            data["singles"] = singles.into();
        }

        let mut filename = path.clone();
        filename.push("/");
        filename.push(name);

        let mut file =
         ↪  File::create(&filename).expect(&*format!("{:?}",
         ↪  &filename));
        file.write_all(data.pretty(4).as_bytes()).unwrap();
    }

    pub fn export_as_ase_json(&self, filename: String) {
        let parsed = self.source_file.as_ref().unwrap();
        let oxygens = &self.oxygens;
        let last_id = &parsed["ids"][parsed["ids"].len() -
         ↪  1].to_string();

        let old_numbers = parsed[last_id]["numbers"].clone();
        let mut new_numbers = parsed[last_id]["numbers"].clone();
        let mut new_positions = parsed[last_id]["positions"].clone();

        let z_coords =
         ↪  new_positions["__ndarray__"][2].members().skip(2).step_by(3);

        new_numbers["__ndarray__"][2].clear();
        for (old_number, z) in
         ↪  zip(old_numbers["__ndarray__"][2].members(), z_coords) {
            if (old_number.as_usize() == Some(1usize)) & (z.as_f64()
             ↪  < Some(20.0)) {
                new_numbers["__ndarray__"][2]
                    .push(8)
                    .expect("Pushing new number failed.");
            } else {
                new_numbers["__ndarray__"][2]
                    .push(old_number.clone())
                    .expect("Pushihng old number failed");
            }
        }
```

```
        for oxygen in oxygens {
            new_numbers["__ndarray__"][2]
                .push(14)
                .expect("new_numbers[\"__ndarray__\"][2].push(8)");
            new_positions["__ndarray__"][2]
                .push(oxygen.x)
                 ↪  .expect("new_positions[\"__ndarray__\"][2].push(oxygen.x)")
            new_positions["__ndarray__"][2]
                .push(oxygen.y)
                 ↪  .expect("new_positions[\"__ndarray__\"][2].push(oxygen.y)")
            new_positions["__ndarray__"][2]
                .push(oxygen.z)
                 ↪  .expect("new_positions[\"__ndarray__\"][2].push(oxygen.z)")
            self.add_crown(oxygen, &mut new_numbers, &mut
             ↪  new_positions);
        }
        new_numbers["__ndarray__"][0][0] =
         ↪  new_numbers["__ndarray__"][2].len().into();
        new_positions["__ndarray__"][0][0] =
         ↪  new_numbers["__ndarray__"][2].len().into();

        let mut export_data = json::JsonValue::new_object();
        export_data["1"] = object! {
            cell: parsed[last_id]["cell"].clone(),
            ctime: parsed[last_id]["ctime"].clone(),
            mtime: parsed[last_id]["mtime"].clone(),
            numbers: new_numbers,
            pbc: parsed[last_id]["pbc"].clone(),
            positions: new_positions,
            unique_id: "Not unique",
            user: parsed[last_id]["user"].clone(),
        };

        let mut file = File::create(&filename).expect("Folder does
         ↪  not exist!");
        file.write_all(export_data.pretty(4).as_bytes()).unwrap();
        println!("Saved {filename}");
    }

    fn add_crown(&self, oxygen: &Oxygen, new_numbers: &mut JsonValue,
     ↪  new_positions: &mut JsonValue) {
        let double_crown;
        let single_crown;
        let new_crown = match oxygen.sitetype {
            SiteType::Singlet(_) => {
                single_crown = triple_crown_rotated(0.0);
                single_crown.iter()
            },
            SiteType::Midpoint(p) => {
                double_crown = double_crown_rotated(
                    double_angle(&self.points[p.0[0].0],
                     ↪  &self.points[p.0[1].0])
```

```
                );
                double_crown.iter()
            }
            SiteType::Tripoint(_) => SINGLE_CROWN.iter(),
        };

        for (x, y, z) in new_crown.tuples() {
            new_numbers["__ndarray__"][2]
                .push(1)
                .expect("Adding to borrowed new_numbers failed");
            new_positions["__ndarray__"][2]
                .push(oxygen.x + x)
                .expect("Adding to borrowed new_positions failed");
            new_positions["__ndarray__"][2]
                .push(oxygen.y + y)
                .expect("Adding to borrowed new_positions failed");
            new_positions["__ndarray__"][2]
                .push(oxygen.z + z)
                .expect("Adding to borrowed new_positions failed");
        }
    }
}

fn double_angle(p1: &LatticePoint, p2: &LatticePoint) -> f32 {
    PI - ((p2.y-p1.y) /
     ↪  ((p2.x-p1.x).powi(2)+(p2.y-p1.y).powi(2)).sqrt() ).acos()
}

fn create_silicon_lattice(lattice_3d: Vec<(Vec<f32>, Vec<Vec<f32>>)>)
 ↪  -> Lattice {
    let mut out_lattice = Lattice::new();

    for (location, ghosts) in lattice_3d {
        let new_point = LatticePoint::new(location[0], location[1],
         ↪  location[2], None);
        out_lattice.add_point(new_point.clone());

        for ghost in ghosts {
            out_lattice.add_point(LatticePoint::new(
                ghost[0],
                ghost[1],
                ghost[2],
                Some(new_point.clone()),
            ))
        }
    }
    out_lattice
        .points
        .sort_by_key(|p| (100.0 * p.x + p.y).round() as u32);
    out_lattice
}

fn turn_2d_3d(input_lattice: Vec<(Vec<f32>, Vec<Vec<f32>>)>) ->
 ↪  Vec<(Vec<f32>, Vec<Vec<f32>>)> {
    if input_lattice[0].0.len() == 2 {
```

```rust
        let mut new_lattice = vec![];
        for (point, ghosts) in input_lattice {
            let mut new_point = point.clone();
            new_point.push(0.0);
            let mut new_ghosts = ghosts.clone();
            for g in &mut new_ghosts {
                g.push(0.0);
            }
            new_lattice.push((new_point, new_ghosts));
        }
        new_lattice
    } else if input_lattice[0].0.len() == 3 {
        input_lattice
    } else {
        panic!("Input lattice layout is incorrect: points must be two or threedimentional.")
    }
}
```

C PYTHON CODE

This section contains several Python files used in the development of Crystacean.

C.1 classes.py

```python
from __future__ import annotations
from itertools import chain
import json
from typing import Sequence
from dataclasses import dataclass, field
import matplotlib.pylab as plt


SINGLE_POINT_ENERGY = 1.4
MID_POINT_ENERGY = 0.7
TRI_POINT_ENERGY = 0.4


@dataclass
class LatticePoint:
    x: float
    y: float
    _connected_to: None | SinglePoint | MidPoint | TriPoint = field(default=None, init=False)

    def get_location(self) -> list[float]:
        return [self.x, self.y]

    def get_real_location(self) -> list[float]:
        return self.get_location()

    def get_link(self) -> LatticePoint:
        return self

    @property
    def connected_to(self) -> None | SinglePoint | MidPoint | TriPoint:
```

```
        return self._connected_to

    def set_connection(self, connection: None | SinglePoint |
    ↪ MidPoint | TriPoint):
        if not self._connected_to and not connection:
            raise ValueError("Connection already None")
        if self._connected_to and connection:
            raise ValueError("Point already connected")
        self._connected_to = connection


    def __hash__(self) -> int:
        return hash((self.x, self.y, self.connected_to))


@dataclass
class GhostPoint(LatticePoint):
    linked_point: LatticePoint

    def get_link(self) -> LatticePoint:
        return self.linked_point

    @property
    def connected_to(self) -> None | SinglePoint | MidPoint |
    ↪ TriPoint:
        return self.linked_point.connected_to

    def set_connection(self, connection: None | SinglePoint |
    ↪ MidPoint | TriPoint):
        self.linked_point.set_connection(connection)

    def __hash__(self) -> int:
        return hash((self.x, self.y, self.connected_to))


@dataclass
class SinglePoint:
    x: float
    y: float
    _connections: list[LatticePoint] = field(init=False)
    populated: bool = False

    def set_connections(self, connections: list[LatticePoint]) ->
    ↪ None:
        assert len(connections) == 1
        self._connections = connections

    def get_connections(self) -> list[LatticePoint]:
        return self._connections

    def available(self) -> bool:
        if self.populated:
            return False

        for connection in self.get_connections():
```

```
            if connection.connected_to:
                return False
        return True

    def populate(self) -> bool:
        """
        Connect oxygen to connections, if possible.
        Returns `True` if all connections are available, else returns
        ↪ `False` and
        doesn't connect.
        """
        assert not self.populated, "This point is already populated."

        for connection in self.get_connections():
            if connection.connected_to:
                return False

        for connection in self.get_connections():
            connection.set_connection(self)
        self.populated = True
        return True

    def depopulate(self) -> None:
        """
        Reset the point and connected lattice sites.
        """
        assert self.populated, "Point not populated."

        for connection in self.get_connections():
            connection.set_connection(None)
        self.populated = False

    def __hash__(self) -> int:
        return hash((self.x, self.y))


@dataclass
class MidPoint(SinglePoint):
    def set_connections(self, connections: list[LatticePoint]) ->
    ↪ None:
        assert len(connections) == 2
        self._connections = connections

    def __hash__(self) -> int:
        return hash((self.x, self.y))


@dataclass
class TriPoint(SinglePoint):
    def set_connections(self, connections: list[LatticePoint]) ->
    ↪ None:
        assert len(connections) == 3
        self._connections = connections

    def __hash__(self) -> int:
```

```
134         return hash((self.x, self.y))
135
136
137 def points_to_plot(input_points: Sequence[Plottable]) ->
    ↪ tuple[list[float], list[float]]:
138     """
139     `Plottable = LatticePoint | SinglePoint | MidPoint | TriPoint`
140     """
141     x_points = [point.x for point in input_points]
142     y_points = [point.y for point in input_points]
143     return x_points, y_points
144
145
146 @dataclass
147 class FullLattice:
148     points: list[LatticePoint]
149     singles: list[SinglePoint]
150     midpoints: list[MidPoint]
151     tripoints: list[TriPoint]
152
153
154     def energy(self) -> float:
155         total = 0
156         for tri in self.tripoints:
157             if tri.populated:
158                 total += TRI_POINT_ENERGY
159         for mid in self.midpoints:
160             if mid.populated:
161                 total += MID_POINT_ENERGY
162         for single in self.singles:
163             if single.populated:
164                 total += SINGLE_POINT_ENERGY
165         return total
166
167     def is_solved(self) -> bool:
168         for point in self.points:
169             if not point.connected_to:
170                 return False
171         return True
172
173     def empty_amount(self) -> int:
174         total = 0
175         for point in self.points:
176             if not point.connected_to:
177                 total += 1
178         return total
179
180     def plot(self, draw_single: bool = False):
181         plt.plot( *points_to_plot(self.points), "o")
182         for num, point in enumerate(self.points):
183             plt.annotate(str(num), (point.x, point.y))
184
185         plt.plot( *points_to_plot(self.midpoints), "x")
186         plt.plot( *points_to_plot(self.tripoints), "s")
187         if draw_single:
```

```
188             plt.plot( *points_to_plot(self.singles), "^",
                ↪  markersize=10)
189         plt.show()
190
191     def plot_report(self, lattice_size: int, site_size: int):
192         plt.plot( *points_to_plot(self.points), "o",
                ↪  markersize=lattice_size, label="Previous layer")
193         plt.plot( *points_to_plot(self.midpoints), "x",
                ↪  markersize=site_size, label="Midpoint",
                ↪  markeredgewidth=4)
194         plt.plot( *points_to_plot(self.tripoints), "s",
                ↪  markersize=site_size, label="Tripoint")
195         plt.plot( *points_to_plot(self.singles), "^",
                ↪  markersize=site_size, label="Single")
196         plt.axis('equal')
197         plt.xlabel("x (Å)")
198         plt.ylabel("y (Å)")
199         plt.show()
200
201     def plot_ghost_connections(self):
202         plt.plot( *points_to_plot(self.points), "o")
203         for num, point in enumerate(self.points):
204             plt.annotate(str(num), (point.x, point.y))
205
206         for point in self.points:
207             linked_point = point.get_link()
208             if not linked_point == point:
209                 plt.plot( *points_to_plot([linked_point, point]),
                        ↪  "--")
210         plt.show()
211
212     def as_spg_tuple(self, oxygens: bool = True) ->
        ↪  tuple[list[tuple[float, float, float]], list[tuple[float,
        ↪  float, float]], list[int]]:
213         basis_vectors = [(0.0, 0.0, 1.5)]
214         return_points = []
215         return_elements = []
216
217         for silicon in self.points:
218             if isinstance(silicon, GhostPoint):
219                 print("skipped")
220                 continue
221             return_points.append(( *silicon.get_location(), 0.0))
222             return_elements.append(14)
223             if len(basis_vectors) == 1:
224                 if not silicon.y == 0.0:
225                     basis_vectors.append((silicon.x, silicon.y, 0.0))
226             elif len(basis_vectors) == 2:
227                 if silicon.y == 0.0:
228                     basis_vectors.append((silicon.x, silicon.y, 0.0))
229
230         if oxygens:
231             for oxygen in chain(self.tripoints, self.midpoints,
                    ↪  self.singles):
232                 return_points.append((oxygen.x, oxygen.y, 1.5))
```

```
                return_elements.append(8)

        return basis_vectors, return_points, return_elements

    def export(self, filename: str):
        export_dict = {
            "lattice_points" : [],
            "tripoints": [],
            "midpoints": [],
            "singles": []
        }
        for point in self.points:
            export_dict["lattice_points"].append(
                {
                    "x": point.x,
                    "y": point.y,
                    "ghost": isinstance(point, GhostPoint)
                }
            )
        for tri in self.tripoints:
            export_dict["tripoints"].append(
                {
                    "x": tri.x,
                    "y": tri.y
                }
            )
        for mid in self.midpoints:
            export_dict["midpoints"].append(
                {
                    "x": mid.x,
                    "y": mid.y
                }           )
        for single in self.singles:
            export_dict["singles"].append(
                {
                    "x": single.x,
                    "y": single.y
                }
            )
        with open(f"exports/{filename}.json", mode="w") as file:
            json.dump(export_dict, file, indent=4)

    def __hash__(self) -> int:
        return hash(tuple(self.points))


def from_file(filename: str) -> FullLattice:
    points: list[LatticePoint] = []
    singles: list[SinglePoint] = []
    midpoints: list[MidPoint] = []
    tripoints: list[TriPoint] = []
    with open(filename) as file:
        structure = json.load(file)
    for point in structure["lattice_points"]:
        points.append(LatticePoint(point["x"], point["y"]))
```

```
    for single in structure["singles"]:
        singles.append(SinglePoint(single["x"], single["y"],
        ↪ populated=True))
    for mid in structure["midpoints"]:
        midpoints.append(MidPoint(mid["x"], mid["y"],
        ↪ populated=True))
    for tri in structure["tripoints"]:
        tripoints.append(TriPoint(tri["x"], tri["y"],
        ↪ populated=True))
    return FullLattice(points, singles, midpoints, tripoints)


type Plottable = LatticePoint | SinglePoint | MidPoint | TriPoint

if __name__ == '__main__':
    import os
    path = "exports/sizeable_unique_1/(4, 0, 4)"
    for structure in os.listdir(path):
        lattice = from_file(f"{path}/{structure}")
        lattice.plot(draw_single=True)
    # lattice = from_file("exports/sizeable_unique_wrong/(5, 0,
    ↪ 1)/sizeable_rust_0022.json")
    # lattice.plot(draw_single=True)
```

C.2 findthosepoints.py

```
from __future__ import annotations
from copy import deepcopy
from typing import Sequence
import matplotlib.pyplot as plt
from matplotlib import rcParams
from itertools import combinations, permutations, product
from scipy.spatial import KDTree
from dataclasses import dataclass, field
from classes import *
import numpy as np


def find_ox_sites(lattice: list[list[float]]) -> FullLattice:
    lattice.sort(key=lambda x: 100*x[0] + x[1])

    lattice_points = [LatticePoint(point[0], point[1]) for point in
    ↪ lattice]
    single_points = [SinglePoint(point.x, point.y) for point in
    ↪ lattice_points]
    for s_point, point in zip(single_points, lattice_points):
        s_point.set_connections([point])

    point_kdtree = KDTree(lattice) # type: ignore
    node_search_distance =
    ↪ point_kdtree.query(lattice_points[0].get_location(),
    ↪ k=[2])[0][0] * 1.1
    unsorted_pairs: set[tuple[int, int]] =
    ↪ point_kdtree.query_pairs(node_search_distance,
    ↪ output_type='set')
    pairs = sorted(unsorted_pairs)
```

```
    print(node_search_distance)

    midpoints: list[MidPoint] = []
    for pair in pairs:
        point1 = lattice[pair[0]]
        point2 = lattice[pair[1]]
        new_point = MidPoint((point1[0] + point2[0]) / 2, (point1[1]
         ↪  + point2[1]) / 2)
        new_point.set_connections([
            lattice_points[pair[0]],
            lattice_points[pair[1]]
        ])
        midpoints.append(new_point)

    tripoints: list[TriPoint] = []
    for pair1, pair2 in combinations(pairs, 2):
        if (not pair1[0] == pair2[0]):
            continue
        point1 = lattice[pair1[0]]
        point2 = lattice[pair1[1]]
        point3 = lattice[pair2[1]]
        # if abs(point2[0] - point3[0]) > node_search_distance or
         ↪  abs(point2[1] - point3[1]) > node_search_distance:
        if not (pair1[1], pair2[1]) in unsorted_pairs:
            continue

        # print(f"{pair1=}, {pair2};      {point1=}, {point2=},
         ↪  {point3=}")
        new_point = TriPoint((point1[0] + point2[0] + point3[0]) / 3,
         ↪  (point1[1] + point2[1]+ point3[1]) / 3)
        new_point.set_connections( [
            lattice_points[pair1[0]],
            lattice_points[pair1[1]],
            lattice_points[pair2[1]]
        ] )
        tripoints.append(new_point)

    return FullLattice(lattice_points, single_points, midpoints,
     ↪  tripoints)


def point_sorting_key(point: LatticePoint) -> float:
    return point.x * 100 + point.y


def find_ox_sites_bounds(lattice: list[tuple[list[float],
 ↪  list[list[float]]]]) -> FullLattice:
    """
    Where `ghost_lattice` is a superset of `lattice`.
    """
    lattice.sort(key=lambda x: 100*x[0][0] + x[0][1])

    lattice_points: list[LatticePoint] = []
    for point in lattice:
        point_object = LatticePoint(point[0][0], point[0][1])
```

```
        lattice_points.append(point_object)
        for ghost in point[1]:
            lattice_points.append(GhostPoint(ghost[0], ghost[1],
             ↪  point_object))


    single_points = [SinglePoint(point.x, point.y) for point in
     ↪  lattice_points if not isinstance(point, GhostPoint)]
    for s_point, point in zip(single_points, [point for point in
     ↪  lattice_points if not isinstance(point, GhostPoint)]):
        s_point.set_connections([point.get_link()])

    lattice_points.sort(key=point_sorting_key)
    new_lattice: list[list[float]] = []
    for point in lattice:
        new_lattice.append(point[0])
        for ghost in point[1]:
            new_lattice.append(ghost)
    new_lattice.sort(key=lambda x: 100*x[0] + x[1])

    for one, other in zip(lattice_points, new_lattice):
        assert one.x == other[0]
        assert one.y == other[1]

    point_kdtree = KDTree(new_lattice) # type: ignore
    node_search_distance =
     ↪  point_kdtree.query(lattice_points[0].get_location(),
     ↪  k=[2])[0][0] * 1.1

    unsorted_pairs: set[tuple[int, int]] =
     ↪  point_kdtree.query_pairs(node_search_distance,
     ↪  output_type='set')
    pairs = sorted(unsorted_pairs)
    print(node_search_distance)

    midpoints: list[MidPoint] = []
    for pair in pairs:
        point1 = lattice_points[pair[0]]
        point2 = lattice_points[pair[1]]
        if isinstance(point1, GhostPoint) and isinstance(point2,
         ↪  GhostPoint):
            continue
        new_point = MidPoint((point1.x + point2.x) / 2, (point1.y +
         ↪  point2.y) / 2)
        new_point.set_connections([
            point1.get_link(),
            point2.get_link()
        ])
        midpoints.append(new_point)

    tripoints: list[TriPoint] = []
    for pair1, pair2 in combinations(pairs, 2):
        if (not pair1[0] == pair2[0]):
            continue
        if not (pair1[1], pair2[1]) in unsorted_pairs:
```

```
            continue

        point1 = lattice_points[pair1[0]]
        point2 = lattice_points[pair1[1]]
        point3 = lattice_points[pair2[1]]
        # if abs(point2[0] - point3[0]) > node_search_distance or abs(point2[1] - point3[1]) > node_search_distance:
        if isinstance(point1, GhostPoint) and isinstance(point2, GhostPoint) and isinstance(point2, GhostPoint):
            continue

        # print(f"{pair1=}, {pair2};      {point1=}, {point2=}, {point3=}")
        new_point = TriPoint((point1.x + point2.x + point3.x) / 3, (point1.y + point2.y+ point3.y) / 3)
        new_point.set_connections( [
            point1.get_link(),
            point2.get_link(),
            point3.get_link()
        ] )
        tripoints.append(new_point)

    return FullLattice(lattice_points, single_points, midpoints, tripoints)


def points_to_plot(input_points: Sequence[Plottable]) -> tuple[list[float], list[float]]:
    """
    `Plottable = LatticePoint | SinglePoint | MidPoint | TriPoint`
    """
    x_points = [point.x for point in input_points]
    y_points = [point.y for point in input_points]
    return x_points, y_points


def make_site_plot(
    lattice: FullLattice,
    draw_single: bool = False
):
    plt.plot( *points_to_plot(lattice.points), "o")
    for num, point in enumerate(lattice.points):
        plt.annotate(str(num), (point.x, point.y))

    plt.plot( *points_to_plot(lattice.midpoints), "x")
    plt.plot( *points_to_plot(lattice.tripoints), "s")
    if draw_single:
        plt.plot( *points_to_plot(lattice.singles), "^", markersize=10)
    plt.show()


def fill_oxygen_maxtri(lattice: FullLattice) -> FullLattice:
    """
```

```
    Fill the lattice with points by first trying to populate all
     ↪ `TriPoint`s.
    """
    # shuffle(tripoints)
    filled_tri: list[TriPoint] = []
    for tri in lattice.tripoints:
        if tri.populate():
            filled_tri.append(tri)

    filled_mid: list[MidPoint] = []
    for mid in lattice.midpoints:
        if mid.populate():
            filled_mid.append(mid)

    filled_single: list[SinglePoint] = []
    for single in lattice.singles:
        if single.populate():
            filled_single.append(single)

    return FullLattice(lattice.points, filled_single, filled_mid,
     ↪ filled_tri)


def fill_oxygen_midonly(lattice: FullLattice) -> FullLattice:
    """
    Fill the lattice with points by first trying to populate all
     ↪ `MidPoint`s.
    """
    # shuffle(midpoints)
    filled_mid: list[MidPoint] = []
    for mid in lattice.midpoints:
        if mid.populate():
            filled_mid.append(mid)

    filled_single: list[SinglePoint] = []
    for single in lattice.singles:
        if single.populate():
            filled_single.append(single)

    return FullLattice(lattice.points, filled_single, filled_mid, [])


@dataclass
class Solver:
    lattice: FullLattice
    lowest_energy = 1_000_000.0
    archive: set[int] = field(default_factory=set, init=False)
    depth: int = field(default=0, init=False)

    def start_solve(self, depth: int = 0) -> list[FullLattice]:
        current_gen: list[FullLattice] = []
        current_gen.append(deepcopy(self.lattice))
        gen_nr = 0
        solutions: list[FullLattice] = []
        while not solutions or (depth and gen_nr <= depth):
```

```
            if depth:
                print(f"Generation {gen_nr}/{depth}")
            else:
                print(f"Generation {gen_nr}")

            next_gen = []
            for lattice in current_gen:
                options_found = 0
                spare_lattice = deepcopy(lattice)
                for number, tripoint in enumerate(lattice.tripoints):
                    if not tripoint.available():
                        continue
                    spare_lattice.tripoints[number].populate()
                    spare_lattice_hash = hash(spare_lattice)
                    if spare_lattice_hash in self.archive or
                     ↪  spare_lattice.energy() >= self.lowest_energy:
                        spare_lattice.tripoints[number].depopulate()
                        continue

                    self.archive.add(spare_lattice_hash)
                    if spare_lattice.is_solved():
                        solutions.append(deepcopy(spare_lattice))
                    elif not solutions or (depth and gen_nr < depth):
                        next_gen.append(deepcopy(spare_lattice))
                    options_found += 0
                    spare_lattice.tripoints[number].depopulate()

                for number, midpoint in enumerate(lattice.midpoints):
                    if not midpoint.available():
                        continue
                    spare_lattice.midpoints[number].populate()
                    spare_lattice_hash = hash(spare_lattice)
                    if spare_lattice_hash in self.archive or
                     ↪  spare_lattice.energy() >= self.lowest_energy:
                        spare_lattice.midpoints[number].depopulate()
                        continue

                    self.archive.add(spare_lattice_hash)
                    if spare_lattice.is_solved():
                        solutions.append(deepcopy(spare_lattice))
                    elif not solutions or (depth and gen_nr < depth):
                        next_gen.append(deepcopy(spare_lattice))
                    options_found += 0
                    spare_lattice.midpoints[number].depopulate()

                # do not try to fit singles in if not necessary
                if options_found:
                    continue
                for number, single in enumerate(lattice.singles):
                    if not single.available():
                        continue
                    spare_lattice.singles[number].populate()
                    spare_lattice_hash = hash(spare_lattice)
                    if spare_lattice_hash in self.archive or
                     ↪  spare_lattice.energy() >= self.lowest_energy:
```

```
                        spare_lattice.singles[number].depopulate()
                        continue

                    self.archive.add(spare_lattice_hash)
                    if spare_lattice.is_solved():
                        solutions.append(deepcopy(spare_lattice))
                    elif not solutions or (depth and gen_nr < depth):
                        next_gen.append(deepcopy(spare_lattice))
                    spare_lattice.singles[number].depopulate()
            current_gen = next_gen
            gen_nr += 1

        if not solutions:
            raise ValueError("Depth was to shallow. Deepen search or remove depth requirement.")
        return solutions

    # def start_solve_accelerated(self, depth: int = 0) -> list[FullLattice]:



def reduce_lattice(lattice: FullLattice) -> FullLattice:
    return FullLattice(
        lattice.points,
        [single for single in lattice.singles if single.populated],
        [mid for mid in lattice.midpoints if mid.populated],
        [tri for tri in lattice.tripoints if tri.populated],
    )


def full_lattice_from_basis_vectors(size: int = 1) -> list[tuple[list[float], list[list[float]]]]:
    points = []
    bottom_points = {}
    left_points = {}
    vec_x = np.array([1.5, 0.0, 0.0]) / 1.5 * 3.076
    vec_y = np.array([0.75, 1.299038105676658, 0.0]) / 1.5 * 3.076
    vec_z = np.array([0.0, 0.0, 1.5])

    for x, y in product(range(0, size), range(0, size)):
        # create main cluster
        new_point_lb = (list((2*x - y)     * vec_x + 2*y      * vec_y)[0:2], [])
        new_point_rb = (list((2*x - y  + 1)* vec_x + 2*y      * vec_y)[0:2], [])
        new_point_lt = (list((2*x - y)     * vec_x +(2*y + 1) * vec_y)[0:2], [])
        new_point_rt = (list((2*x - y  + 1)* vec_x +(2*y + 1) * vec_y)[0:2], [])
        points.append(new_point_rb)
        points.append(new_point_lb)
        points.append(new_point_rt)
        points.append(new_point_lt)

```

```
319             # note left/bottom boundaries
320             if y == 0:
321                 bottom_points[new_point_lb[0][0]] = new_point_lb
322                 bottom_points[new_point_rb[0][0]] = new_point_rb
323             if x == 0:
324                 left_points[new_point_lb[0][1]] = new_point_lb
325                 left_points[new_point_lt[0][1]] = new_point_lt
326 
327             # create and link top/right boundaries
328             if y == size - 1:
329                 bound_point_1 = list((2*x - y - 1) * vec_x + 2*(y +
                    ↪  1)*vec_y)[0:2]
330                 bount_point_2 = list((2*x - y) * vec_x + 2*(y + 1)*
                    ↪  vec_y)[0:2]
331                 bottom_points[bound_point_1[0]][1].append(bound_point_1)
332                 bottom_points[bount_point_2[0]][1].append(bount_point_2)
333             if x == size - 1:
334                 bound_point_3 = list((2*x - y  + 2)* vec_x + 2*y      *
                    ↪  vec_y)[0:2]
335                 bount_point_4 = list((2*x - y  + 2)* vec_x +(2*y + 1) *
                    ↪  vec_y)[0:2]
336                 left_points[bound_point_3[1]][1].append(bound_point_3)
337                 left_points[bount_point_4[1]][1].append(bount_point_4)
338                 if y == size - 1:
339                     corner_point = list((2*x - y  + 1)* vec_x +(2*y + 2)
                        ↪  * vec_y)[0:2]
340                     bottom_points[0.0][1].append(corner_point)
341 
342 
343     # points.append(list(0*vec_x + 0*vec_y)[0:2])
344     # points.append(list((0 + 1)*vec_x + 0*vec_y)[0:2])
345     # points.append(list(0*vec_x + (0 + 1)*vec_y)[0:2])
346     # points.append(list(1*vec_x + 1*vec_y)[0:2])
347     # print(points)
348     return points
349 
350 
351 
352 if __name__ == '__main__':
353     litterally_just_a_line = [
354         [0.0, 0.0],
355         [1.5, 0.0]
356     ]
357     equilateral_triangle_points = [
358         [0.0, 0.0],
359         [0.75, 1.299038105676658],
360         [1.5, 0.0]
361     ]
362     flipped_triangle = [
363         [0.0, 1.299038105676658],
364         [0.75, 0.0],
365         [1.5, 1.299038105676658]
366     ]
367     double_triangle_points = [
368         [0.0, 0.0],
```

```
        [0.75, 1.299038105676658],
        [1.5, 0.0],
        [2.25, 1.299038105676658]
    ]
    big_lattice_points = [
        [0.0, 5.3],
        [1.6, 13.4],
        [3.1, 5.3],
        [9.3, 0.0],
        [10.8, 2.7],
        [7.7, 2.7],
        [1.6, 8.0],
        [6.2, 0.0],
        [13.9, 8.0],
        [12.4, 5.3],
        [3.1, 10.7],
        [0.0, 10.7],
        [4.6, 8.0],
        [13.9, 2.7],
        [4.6, 2.7],
        [3.1, 0.0],
        [10.8, 8.0],
        [9.3, 5.3],
        [10.8, 13.4],
        [6.2, 5.3],
        [1.6, 2.7],
        [0.0, 0.0],
        [6.2, 10.7],
        [13.9, 13.4],
        [12.4, 10.7],
        [7.7, 13.4],
        [7.7, 8.0],
        [12.4, 0.0],
        [4.6, 13.4],
        [9.3, 10.7]
    ]
    boundary_points = [
        ([0.0, 0.0], [
            [0.0, 5.3],
            [6.2, 5.3],
            [6.2, 0.0],
        ]),
        ([1.6, 2.7], [
            [7.7, 2.7],
        ]),
        ([3.1, 0.0], [
            [3.1, 5.3],
        ]),
        ([4.6, 2.7], []),
    ]
    bigger_boundary_points = [
        ([0.0, 0.0], [
            [0.0, 10.7],
            [9.3, 0.0],
            [9.3, 10.7],
```

```
        ]),
        ([0.0, 5.3], [
            [9.3, 5.3],
        ]),
        ([1.6, 2.7], [
            [10.8, 2.7],
        ]),
        ([1.6, 8.0], [
            [10.8, 8.0],
        ]),
        ([3.1, 0.0], [
            [3.1, 10.7],
        ]),
        ([3.1, 5.3], []),
        ([4.6, 2.7], []),
        ([4.6, 8.0], []),
        ([6.2, 0.0], [
            [6.2, 10.7],
        ]),
        ([6.2, 5.3], []),
        ([7.7, 2.7], []),
        ([7.7, 8.0], []),
    ]
    big_lattice_points = [point for point in big_lattice_points if
     ↪  point[0] < 12 and point[1] < 9]
    rcParams.update({'font.size': 11})
    lattice_size = 10
    site_size = 16

    filled_lattice = find_ox_sites(big_lattice_points)
    filled_lattice.plot_report(lattice_size, site_size)
    # filled_lattice = find_ox_sites_bounds(bigger_boundary_points)
    # points = full_lattice_from_basis_vectors(size=2)
    # filled_lattice = find_ox_sites_bounds(points)
    # filled_lattice.plot_ghost_connections()
    # make_site_plot(filled_lattice, draw_single=True)

    # solved_lattice = fill_oxygen_maxtri(filled_lattice)
    # solved_lattice = fill_oxygen_midonly(filled_lattice)
    # shuffle(filled_lattice.midpoints)
    # shuffle(filled_lattice.tripoints)
    solution_list = Solver(filled_lattice).start_solve(depth=0)

    # Use the "just look at it" theorum to select for unique
     ↪  solutions
    # exportset = {0, 1, 2, 3, 25, 26, 37, 70}
    # exportset = {0, 9, 10}
    for number, lattice in enumerate(solution_list):
        print(f"Showing {number+1} of {len(solution_list)}")
        print(f"Energy: {lattice.energy()}")

        reduced = reduce_lattice(lattice)
        reduced.plot_report(lattice_size, site_size)
        # if number in exportset:
        #     reduced.export(f"bigger_lattice_{number:0>4}")
```

C.3 cull_results.py

```python
import os
import json
from itertools import chain, combinations
from math import sqrt
from alive_progress import alive_bar


# FOCUS = {"sizeable_rust_32232.json", "sizeable_rust_35469.json"}
FOCUS = set()


def nearly_in(new: list[float], uniques: list[list[float]], margin: float) -> bool:
    """
    Returns true if `new` list has matching list in `uniques`, where all elements
    differ by no more than `margin`:

    For every `x`:
    ```
    new[x] - uniques[:][x] <= margin
    ```
    """
    for unique in uniques:
        if not len(new) == len(unique):
            continue
        for a, b in zip(new, unique):
            if (a - b) > margin:
                break
        else:
            return True
    return False


def sorting_key(input) -> float:
    return 100 * input["x"] + input["y"]


def cull(path: str, margin: float, postfix: str):
    list_of_files = os.listdir(path)
    new_path = path + "_unique_" + postfix
    os.mkdir(new_path)
    unique_spacings = {}
    unique_files = {}
    total_amount = 0

    with open(f"{path}/{list_of_files[0]}") as file:
        lattice = json.load(file)
        max_x = 0
        max_y = 0
        for number, lattice_point in enumerate(lattice["lattice_points"]):
            if lattice_point["x"] > max_x and lattice_point["y"] == 0:
```

```
                print(f"Choose point {number} for {max_x=}")
                max_x = lattice_point["x"]
            if lattice_point["y"] > max_y:
                print(f"Choose point {number} for {max_y=}")
                max_y = lattice_point["y"]
    # extremely important progress bar
    with alive_bar(
        len(list_of_files), spinner="dots_waves", title="Analyzing
        ↪ files..."
    ) as bar:
        for filename in list_of_files:
            with open(f"{path}/{filename}") as file:
                lattice = json.load(file)

                connection_type_count = (
                    len(lattice["tripoints"]),
                    len(lattice["midpoints"]),
                    len(lattice["singles"]),
                )
                oxygens = list(
                    chain(
                        lattice["tripoints"], lattice["midpoints"],
                        ↪ lattice["singles"]
                    )
                )

                oxygen_spacing = []
                for o1, o2 in combinations(oxygens, 2):
                    dx = abs(o1["x"] - o2["x"])
                    if dx > max_x/2:
                        dx -= max_x
                    dy = abs(o1["y"] - o2["y"])
                    if dy > max_y/2:
                        dy -= max_y

                    spacings = sqrt((dx)**2 + (dy)**2)
                    oxygen_spacing.append(spacings)
                oxygen_spacing.sort()
                if filename in FOCUS:
                    print([round(number, 4) for number in
                    ↪ oxygen_spacing])

                if not unique_spacings.get(connection_type_count):
                    unique_spacings[connection_type_count] =
                    ↪ [oxygen_spacing]
                    unique_files[connection_type_count] = [filename]
                    total_amount += 1
                elif not nearly_in(
                    oxygen_spacing,
                    ↪ unique_spacings[connection_type_count],
                    ↪ margin
                ):

                    ↪ unique_spacings[connection_type_count].append(oxygen_sp
```

```
98
                        ↪ unique_files[connection_type_count].append(filename)
99                      total_amount += 1
100             bar()
101
102     # print(total_amount)
103     # print(f"{len(unique_files[(4, 0, 4)])}")
104     with alive_bar(
105         total_amount, spinner="dots_waves", title="Copying
            ↪ uniques..."
106     ) as bar:
107         for conn_type, files in unique_files.items():
108             for filename in files:
109                 if f"{conn_type}" not in os.listdir(new_path):
110                     os.mkdir(f"{new_path}/{conn_type}")
111                 with open(f"{path}/{filename}") as file:
112                     with open(f"{new_path}/{conn_type}/{filename}",
                        ↪ "w") as new_file:
113                         json.dump(json.load(file), new_file)
114                 bar()
115
116
117 if __name__ == "__main__":
118     cull("exports/sizeable_remade", 0.2, "2")
```