# BiGGER Python

Artificial Intelligence Project Report

Authors: Andrea Grendene (1184985), Davide Colacicco (1175173)

## Abstract

With this project we have studied the algorithm BiGGER[1] (Bimolecular complex Generation with Global Evaluation and Ranking), which purpose is to use Constraint Programming to improve proteins docking algorithms performances. One of the most used techniques, at least when BiGGER paper was published, is based on Fast Fourier Transforms, so the aim of the document we read is to find a equally faster method that works in the same cases and is more flexible. Besides Constant Programming allows users to add new constraints, for example from experimental data, in a simple way, while the previous methods must be rewritten in order to do so.
Our purpose instead is to understand how BiGGER works, expecially with regard to constraint programming. Besides the original version of this algorithm and Chemera, which is the application GUI, are written in Pascal, so we decided to rewrite both in Python, also to improve the comprehension of BiGGER steps.

## Used applications and other useful information

- Lazarus (Pascal IDE)
- PyCharm (Python IDE)
- Python 3 (but only GUI is not compatible with version 2.7)
- Repository link: https://github.com/Cenze94/BiggerPython

- How to install Lazarus and execute Chemera:
  - download Lazarus from https://www.lazarus-ide.org/, install and start it;
  - open Package -> Install/Uninstall Packages and install ‘lazopenglcontext.lpk’ package;
  - copy Open-Chemera project from ‘https://github.com/Cenze94/BiggerPython/tree/master/Open-Chemera-master’;
  - create a folder called ‘oclibrary’ in ‘[User]/AppData/Local/’ directory, then copy there all the files inside ‘[Chemera directory]/oclibrary/data’;
  - open and run ‘[Chemera directory]/oclibrary/chemera/chemera.lpi’.
- How to install Python and execute BiggerPython:
  - download Python from ‘https://www.python.org/downloads/’ and install it, checking ‘Add Python 3.x to PATH’;
  - open Windows PowerShell as Administrator and execute ‘pip install guizero’ and ‘pip install numpy’, to install the respective packages;
  - download ‘https://github.com/Cenze94/BiggerPython/tree/master/BiggerPython’ and ‘https://github.com/Cenze94/BiggerPython/tree/master/PDB’, put the two folders in the same directory (because BiggerPython GUI test will load PDB files from ‘PDB’ folder);
  - download PyCharm from ‘https://www.jetbrains.com/pycharm/’, install and start it;
  - open ‘GUI.py’;

    - if the Interpreter is not set, click on the warning link (or go to ‘File/Settings/Project: BiggerPython/Project Interpreter’), click on the settings symbol in the top right of the window, select ‘Add...’, click on ‘System Interpreter’ and then on ‘OK’;
    - execute ‘Gui.py’;
    - with Python 2.7 open and execute ‘GUITests.py’ instead; to change the loaded PDB files modify the first two lines of ‘BiggerTest()’ function.
- Where to download PDB files: theoretically the application should be able to parse all PDB files, in any case files from ‘https://en.wikipedia.org/wiki/List_of_proteins’ and ‘https://www.rcsb.org/’ should work. Otherwise inside ‘https://github.com/Cenze94/BiggerPython/tree/master/PDB’ there are 5 working PDB files.

## Proteins and docking

The key elements of our application are proteins. They are structures made of amino acids, the monomers of the proteins, the essential structure of its. Indeed proteins are macromolecules of amino acids, combined to form complex structures. Usually about 20 types of amino acid are used to produce proteins. Proteins are large biomolecules, or macromolecules, consisting of one or more long chains of amino acid residues, performing a vast array of functions within organisms, including catalyzing metabolic reactions, DNA replication, responding to stimuli, providing structure to cells and organisms, and transporting molecules from one location to another. Proteins differ from one another primarily in their sequence of amino acids, which is dictated by the nucleotide sequence of their genes, and which usually results in protein folding into a specific three-dimensional structure that determines its activity. In our application we are inspecting proteins interaction with other proteins. The focus is on the particular case of protein docking, currently the area of greater application of these algorithms, a more complex computational problem is the prediction of intermolecular interactions, such as in molecular docking.

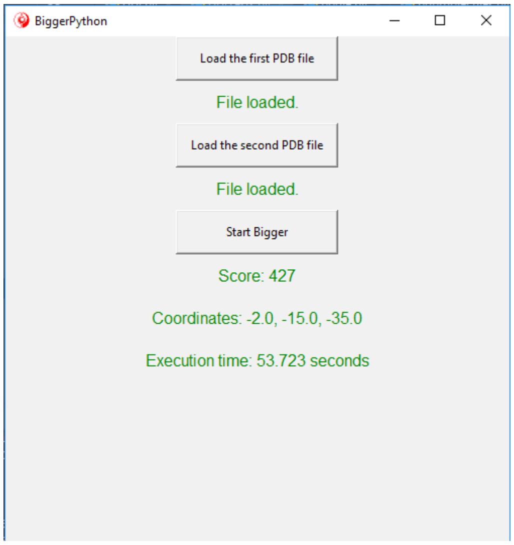
In the field of molecular modeling, docking is a method which predicts the preferred orientation of one molecule to a second one when bounded to each other to form a stable complex structure and interaction.
The general problem we address in this paper is to fit two solids of arbitrary shape maximizing the surface of contact. This problem is important in bioinformatics to model protein interactions, also known as protein docking.

The data necessary for the algorithm is provided and can be extracted from a specific file, created ad hoc for the manipulation of proteins: the PDB file (Protein Data Bank). This file format is a textual file format describing the three-dimensional structures of molecules held in the Protein Data Bank. The PDB format accordingly provides for description and annotation of protein and nucleic acid structures, including atomic coordinates, observed sidechain rotamers, secondary structure assignments, as well as atomic connectivity. Structures are often deposited with other molecules such as water, ions, nucleic acids, ligands and so on, which can be described in the PDB format as well. The Protein Data Bank also keeps data on biological macromolecules.

A typical PDB file describing a protein consists of hundreds or thousands lines, and is divided into the following sections:

- **HEADER, TITLE and AUTHOR records**: provide information about the researchers who defined the structure; numerous other types of records are available to provide other types of information.
- **REMARK records**: can contain free-form annotation, but they also accommodate standardized information; for example, the ‘REMARK 350 BIOMT’ records describe how to compute the coordinates of the experimentally observed multimer from those of the explicitly specified ones of a single repeating unit.
- **SEQRES records**: give the sequences of the three peptide chains (named A, B and C), which usually span multiple lines.
- **ATOM records**: describe data of atoms that are part of the protein. Usually there are three floating point numbers, which are atom x, y and z coordinates measured in units of Ångströms, Then there are the occupancy, the temperature factor and the element name.
- **HETATM records**: describe coordinates of hetero-atoms, that are atoms not belonging to the protein molecule.

## Chemera GUI



File loaded.

File loaded.

Score: 427

Coordinates: -2.0, -15.0, -35.0

Execution time: 53.723 seconds

The application GUI is composed by three buttons, which have an associated text below to give to the user some information about the state of Chemera. The first two buttons are used to load the two proteins, in fact, when the user click one of them, a file dialog is open, in order to find, select and load a PDB file. Texts associated with these two buttons communicate if the file has been uploaded or not. The third button executes BiGGER algorithm, and requires that the two files have been uploaded. If not so, the associated text will show an error. During the execution of BiGGER, it will communicate to the user the step reached by the execution, to improve user experience. When the execution of the algorithm has been successful, the result will be shown in this text and in two others, which normally are hidden. In the first one there will be the best score value, in the second one the relative coordinate, and in the third the execution time of BiGGER.

# BiGGER execution

The execution of BiGGER is composed by two phases:

1. first of all the proteins are transformed in grid form;
2. then the constraint programming is applied, in order to find the best docking result.

## Grid construction

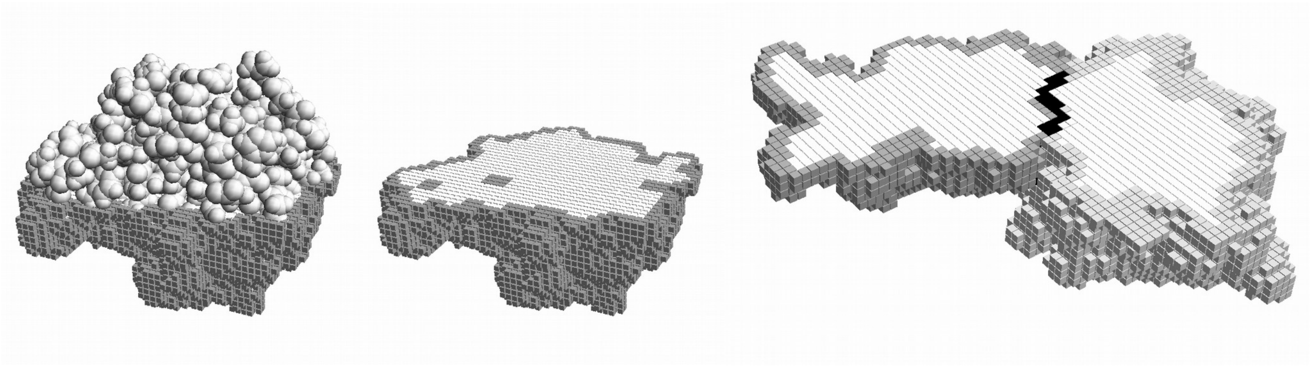


**Fig. 1.** The image on the left shows a protein structure overlaid on a cutaway of the respective grid, with grey spheres representing the atoms of the protein. The centre figure shows only the grid, cut to show the surface in grey and the core region in white. The rightmost image shows the contact between two grids as a black line of grid cells at the interface.

Every protein has to be transformed in grid form: this implies also that grid cells will be divided between surface and core grids. The main class of grids creation and management is TDockingGrid, which contains three grids: FBase, FSurf and FCore. The first is used initially to store and prepare all atoms, then its contents will be divided between the other two grids: FSurf contains surface cells and FCore core ones. Before FBase construction, all coordinates are translated, subtracting the minimum value of every component of the coordinate, which is negative and so the resulting values won't be negative.

To build and manage the grid where to put protein atoms, the application uses the TGeomHasher class: it builds a cube of cells, which size for every axis is the difference between the maximum and the minimum values of the corresponding component of atoms coordinates, divided by the grid step. The latter is the length of a side of cells, which corresponds to the maximum atom radius, as mentioned before. Then the index of every atom is added to the cell that contains its coordinate, divided by the grid step and without the fractional part; so for example if the coordinate is 42.35, 36.21, 12.98, and the grid step is 3, then the atom will be added in the cell with x = 14, y = 12 and z = 4. So there could be more than one atom index in a cell, with a quite large grid step, although we don't know if it can happen in the real world.

FBase grid is completely different from TGeomHasher one: the sizes of the first two dimensions are obtained from the maximum value of the first two components of atoms coordinates, without divisions, so they are much bigger than TGeomHasher ones. The third dimension contains a list of segments: they are the intervals of contiguous cells of FBase where there is an atom. To build it, BiGGER executes the following instructions for every index of the first two dimensions: if x, y and z are the indexes of the first, second and third dimensions (this one is obtained from the third maximum component of atoms coordinates, in the same way as the other two), and zline an array with z zeros, the algorithm checks if the coordinate x, y, z is an inner point, i. e. a coordinate located within the radius of an atom, using TGeomHasher. Then, if it is so, the value of zline in z position is set to 1, and finally zline is converted to a list of segments, where every segment indicates the first and the last indexes of an interval of contiguous 1.

FCore and FSurf are build from FBase, in fact the sizes of their first two dimensions are the same of FBase ones, and their segments are built from the other grid. To get these segments the application uses three arrays, filled with zeros: ‘neighs’, ‘surfs’ and ‘cores’. Then for every array of FBase grid, except for those with indexes equal to x = 0, x = maximum value of first dimension, y = 0 or y = maximum value of second dimension, BiGGER builds neighs. To do this it works with every segment, taking ‘bot’ and ‘top’ as the two end points, and adds 1 to the values of neighs in position bot-1 and top-1, 2 to the ones in position bot and top, and 3 to the ones between bot and top. After that it repeats this operation with x-1 and y-1, x and y-1, x+1 and y-1, x-1 and y, x+1 and y, x-1 and y+2, x and y+2, x+1 and y+2. Finally it builds surfs and cores, taking every value between the end points of every segment: calling this value as ‘zz’, if the number contained in neighs[zz] is equal or greater than 27 then cores[zz] is changed to 1, otherwise surfs[zz] is changed to 1. After that cores and surfs are converted into lists of segments, as described above with FBase, and saved respectively in FCore and FSurf grids.

Besides FBase, FCore and FSurf contain two support matrices, called ‘CellCounts’ and ‘NonEmpty’: the first is a two-dimensional matrix, which sizes are the same of grid first two dimensions, that contains the sum of the lengths of the segments of the grid; the second is an array of arrays of segments, which are the contiguous indexes of non-empty grid[x, y].

## Application of Constraint Programming

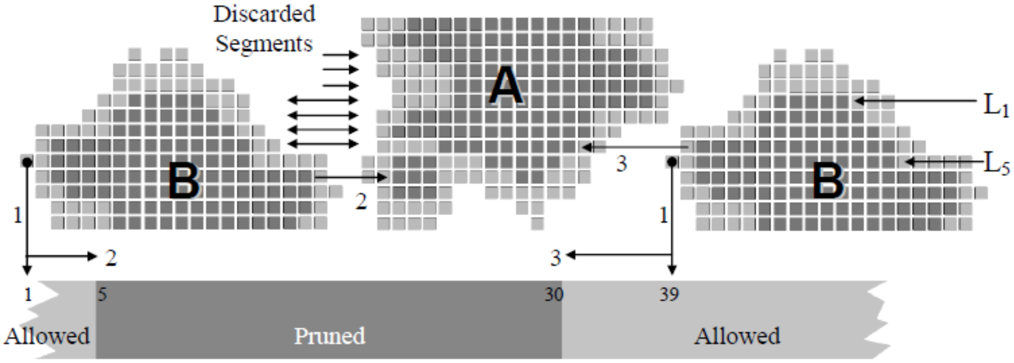


**Fig. 3.** Grid B is translated along the horizontal direction relative to grid A. The vertical arrows marked 1 indicate the position of B on the lower horizontal bar, which shows the allowed and forbidden values for the position of B. The arrows marked 2 and 3 show the allowed displacement of B. The group of horizontal arrows indicates segments to be discarded.

To check the contact between the two proteins, BiGGER builds another grid, which dimensions are the sum of the respective sizes of proteins grids. It is contained and managed in TDockDomain class, which also reduces the domains of this problem. They are represented by the intervals of cells where the proteins can overlap, which initially correspond to the sizes of the general grid. Obviously these intervals must be reduced, using Constraint Programming and Branch and Bound techniques. Domain in X axis remains equal to the domain of the general grid, instead domain in Y axis is reduced. To do so BiGGER finds the biggest intersections of the two proteins in Y for every X, saving them as intervals in the ‘NonEmpty’ object of TDockDomain grid. This is the first constraint: intervals where there aren’t intersections between the two proteins, or where intervals are smaller than others in the same X coordinate, are discarded. After that the algorithm finds core intervals in the same axis, and then saves them in a specific object, called ‘FCoreContacts’; the previous intervals are updated or deleted, in order to avoid intersections in core cells. This is the second constraint, so at this point only surface intervals in the Y axis are saved. Then BiGGER analyses these segments to find intervals in the Z axis. The previous constraints are repeated, so intervals in core cells, without proteins intersections or with smaller intersections than others in the same coordinates are not considered; besides every Y interval is scored, counting the number of possible intersections cells, and deleted when the result is less then a given value. This is the application of the Branch and Bound technique, which can be seen as a third constraint.

To calculate the score of intervals, BiGGER analyses all the intersections of proteins intervals in the Z axis for each interval in Y, then obtains the possible overlaps and calculates the total score. Each score is associated with an intersection in the Y axis, so there will be a lot of models, which are stored in TModelManager class. It manages a buffer, where there are a limited number of models in descending order, so the best result is the first. To discard not good enough results, this class has a minimum score, that is updated when the buffer is full. As mentioned in the 'Chemera GUI' section, each model has a coordinate associated, which represents the position in the first protein of the intersection with the maximum overlap score.

## Results

Since BiggerPython is a rewrite of BiGGER and Chemera in a different language, results and complexities are the same. The only difference is time execution, which we expected to be greater in the case of Python. To measure it we did 10 executions of PDB files parsing and BiGGER, in order to verify how much the difference is. The results are saved in the table below.

| | **1.<br>(Sec)** | **2.<br>(Sec)** | **3.<br>(Sec)** | **4.<br>(Sec)** | **5.<br>(Sec)** | **6.<br>(Sec)** | **7.<br>(Sec)** | **8.<br>(Sec)** | **9.<br>(Sec)** | **10.<br>(Sec)** | **Average<br>(Sec)** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Python PDB** | 0.36 | 0.375 | 0.391 | 0.375 | 0.375 | 0.359 | 0.375 | 0.359 | 0.375 | 0.375 | 0.3719 |
| **Pascal PDB** | 0.11 | 0.125 | 0.109 | 0.109 | 0.109 | 0.109 | 0.11 | 0.11 | 0.109 | 0.109 | 0.1109 |
| **Python BiGGER** | 250.498 | 249.083 | 251.96 | 252.263 | 250.304 | 251.907 | 254.264 | 251.552 | 249.15 | 255.841 | 251.6822 |
| **Pascal BiGGER** | 3.781 | 3.765 | 3.781 | 3.766 | 3.813 | 3.812 | 3.828 | 3.796 | 3.781 | 3.859 | 3.7982 |

Python is extremely slower than Pascal, in fact PDB files parsing and BiGGER are respectively three and 66 times longer. However in Python version there could be much errors than in Pascal one, or maybe it could be optimized. In fact there are some instructions that could be simplified, which have been written in this way to maintain the correlation with the code in Pascal. Furthermore Pascal is a compiled language[2], while Python is interpreted[3], so it's normal to have an important execution time difference[4].

# Conclusions

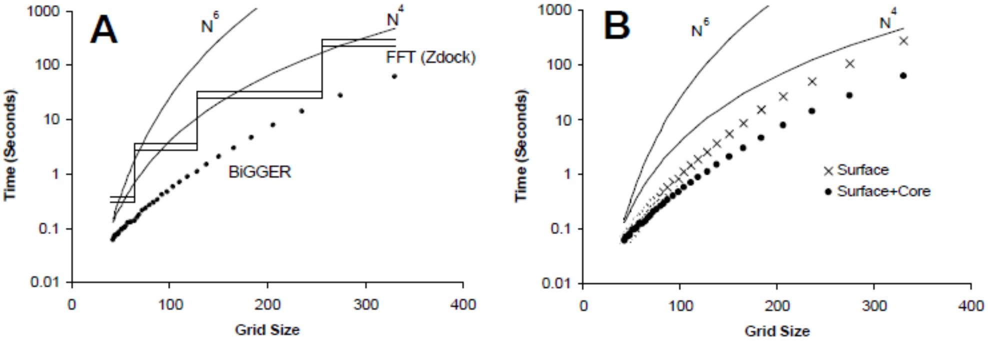


**Fig. 7.** Plots of search time as a function of the grid size. Chart A compares BiGGER (dots) to the estimated time for a FFT implementation (ZDock) on the same hardware. Chart B compares BiGGER with only the surface overlap constraint and both the surface and core overlap constraints. Both charts also show the theoretical values for O($N^6$) and O($N^4$) complexities assuming a constant scaling factor such that both correspond to the estimated factor for the ZDock implementation of the FFT method at N=40.

BiGGER is an interesting application of Constraint Programming to solve a problem that we weren't even able to imagine. Besides it provides a solution with a complexity close to the best method already existing[1], that is FFT, and sometimes it's better. In fact BiGGER complexity is O($N^4$), when FFT one is O($N^3$logN), but the other algorithm uses grids with a size equal to a multiple of 2, while BiGGER works with every value. Memory required is much less than FFT, more precisely memory complexity of this algorithm is O($N^2$), when the complexity of the other one is O($N^3$). With an N smaller than 500 BiGGER is faster than FFT, then the difference of time complexity becomes more relevant; but with a larger N memory size becomes an important problem, so for example with N=512 FFT occupies about 8 GB of memory, when BiGGER requires some MB. Another advantage of Constraint Programming is its flexibility, in fact the addiction of new constraints is easy, and doesn't requires the rewrite of the entire algorithm. Instead previous solutions can't get advantages from additional information, so to use them a user has to write his own method.

The choice of Python as a programming language to use is not good in this case, because Python is easy and fast to use, but it's not performing. Instead Pascal is faster, much more verbose and less used than the other programming language, so in this case it could be a good choice only if the authors know it very well, otherwise there are better languages. For example C++ is much more supported and flexible than Pascal, furthermore it is less verbose and equally performing.

We have achieved our goal, that is the comprehension of the use of Constraint Programming in this application, although there are some details which are not so clear. They concern the functioning of some techniques to find particular data, such as the algorithm to find if a cell belongs to the surface or the core grid. Since we understood their purpose, and therefore the obtained result, they haven't compromised the general understanding of how the algorithm works.

## Bibliography

1. Krippahl Ludwig, Barahona Pedro. 2005. Applying Constraint Programming to Rigid Body Protein Docking.
2. https://en.wikipedia.org/wiki/Pascal_(programming_language).
3. https://en.wikipedia.org/wiki/Python_(programming_language).
4. https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zappldev/zappldev_85.htm.